POLITECNICO
DI TORINO

DEPARTMENT OF ELECTRONICS AND
TELECOMMUNICATIONS
Master of Science in Electronic Engineering (Embedded Systems)

Methodologies for SOC verification

Candidate:
Imane EL FENTIS

Academic Advisor:
Prof. GRAZIANO Mariagrazia

Academic Advisor:
Prof. RIENTE Fabrizio

Academic Year 2019-2020

Polytechnic University Of Turin
DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS
Master of Science in Electronic Engineering (Embedded Systems)

Acknowledgments

I would like to show my gratitude to my Supervisors: M.GRAZIANO and F.RIENTE for enabling this greatly appreciated thesis and allowing me to participate in it and for guiding me throughout the whole process of the development of this thesis. I would also like to recognize the invaluable assistance and support provided from my family and friends during my studies.

Torino, April 2019

Imane EL FENTIS

Abstract

Nowadays, with he increase of the complexity of system on chips SOC, the ASIC industry struggles to meet schedules of time to market TTM. System on chips market is complex in term of the business and technology point of view. However, the time to market enforces a huge pressure to this industry. As a consequence of these factors: appearance of new challenges, among them the top one which is verification. This last one consumes more than 70 percent of design effort.
Verifying the correctness of the final design is the key to design more and more complex SOCs and exploiting `leading_edge` process technologies.
The better to discover the hidden bugs in earlier stages the better in term of the cost, companies often end up with costly mistake. Hence, it is important for the companies to select the suitable tool and techniques for verification. One of the modern and effective methodologies is universal verification methodology UVM.
UVM provides to verification engineer a layered architecture. It is based hierarchy that allows the verification engineer to decompose the problem to sub-problem that can be solved in several steps, hence, not dealing with huge and complex problem. In the next chapters, it is described the importance and efficiency of this methodology and its tools used to verify a SOC and how to use its language.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

Nowadays, the field of integrated systems has known an evolution ,which means the complexity of digital systems has became very complex .In one system on chip (SOC) may include different parts of other electronic systems ,such as the memory ,the DSP,the A/D and D/A converters up to the microprocessor.The aim of this evolution is to make a product faster,more efficient also less expensive in terms of area ,power consumption and obviously in terms of money,taking in consideration time to market when produce the product to the client (final user) with low cost ,It is very important to minimize time and effort invested in life cycle of the product. However,a digital design before arriving to the market ,must pass within several steps starting from the original set of specifications. In the process of manufacturing a Very Large scale integrated circuit three different step:

1. - Design : the design phase is when an idea is transformed to a real working system
2. - verification: to ensure that the functionality of the system is the same as the specification of the design.
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.

One of the largest and the more complex domain is design verification (DV)which contains many languages,technologies and methodologies .A DV engineer must not get pigeonholed in only one of many technologies that fall under DV umbrella . He/she should have largest knowledge about systemverilog,UVM and hardware micro-architecture .At the least, the following technologies fall under DV domain:

* UVM (Universal Verification Methodology).
* UPF (Unified Power Format) low-power verification using UPF.
* AMS (analog/mixed signal) verification. Real number modeling, etc.
* SystemVerilog Assertions (SVA) and functional coverage (SFC) languages and methodology.
* Coverage-driven verification (CDV) and constrained random verification (CRV).
* Static verification technologies. Static formal verification (model checking), static + simulation hybrid methodology, X-state verification, CDC (clock domain crossing), etc.
* Logic equivalency check (LEC). Design teams mostly take on this task. But the DV (design verification) team also needs to have this expertise.

- ESL—Electronic System Level (TLM 2.0) virtual platform development (for both software development and verification tests/reference model development).
- Hardware/software co-verification (hint: use virtual platform methodology).
- SoC interconnect (bus-based and NoC—network-on-chip) verification.
- Simulation speedup using hardware acceleration, emulation, and prototyping

The verification methodologies are continuously developed since the complexity of a design is increased, as a result the verification flows become fractured and in some cases inefficient. Though each technology presented has more features and advantages with respect to the previous version of the same technology or an other technology especially faster bug detection rate . The most challenge for engineer is to find an answer of this question: what can we do during pre-silicon verification to guarantee post-silicon validation a first pass success? Moreover,the biggest challenge that companies face is time to market which is short,in order to deliver first pass working silicon of increasing complexity. Power management ,performance and massive functional capacity are embedded in recent SOC realizations.The challenges of the verification on an increased complexity scale are growing, so traditional verification methodologies are losing field as time to market should be reached and in same time the cost must be met .another burning point nowadays is exploding software content ,so a methodology needed to allow in same time easier software development and faster silicon realization. In the past, it was possible to have a early silicon sample for software development. On a recent report of ITRS (International Technology Roadmap for Semiconductors) we can clearly observe 1.1 the increasing HW-SW design gap.

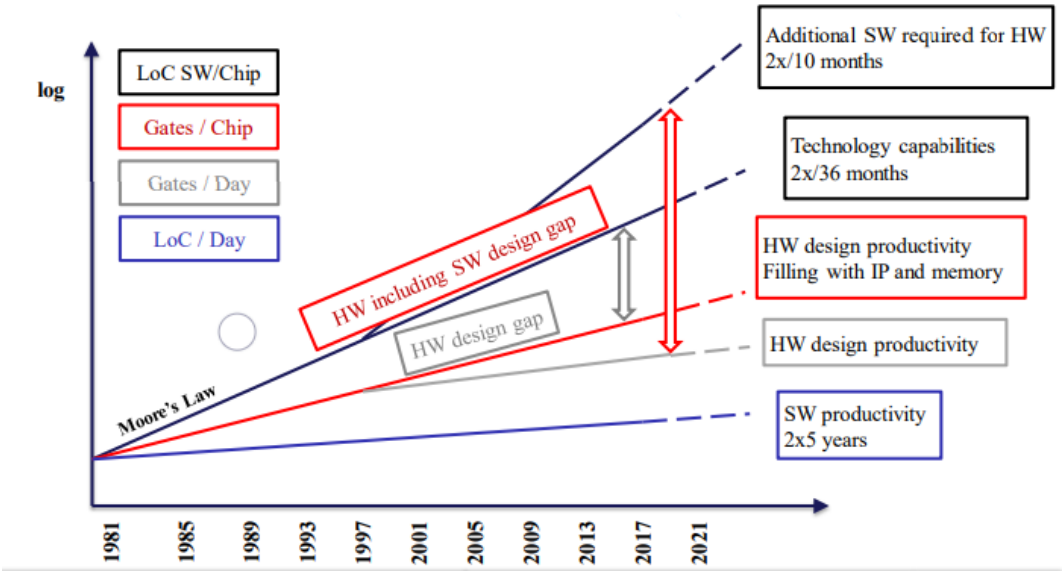


Figure 1.1.: Hw/Sw Design gap-ITRS report

The starting point of such methodology which enables a unified software and hardware development is the specification analysis of the whole system for each of the three directions:design,verification,software development .

In such a unified development environment, the verification role has increased, from developing classical test benches, to complete architecture of transaction-level models that

enable architecture testing, performance metrics, software development, and accurate and efficient design verification.

Chapter 2.

Technology challenges

The embedded systems know a quick shifting from system on board to system on chip ,where all components are in the same die. As an interaction with the market, the system on chips SOCs become more complex and challenging. The velocity of semiconductor processes evolution and the need of the complex applications let thee design and verification engineers think to find and build an efficient future design methodologies and verification methodologies to meet the constraints such time to market TTM and handle the complexity of the design . By integrating several pre-designed cores on one and same die ,realizes more soc so it becomes state-of-the-art. Nevertheless ,this evolution raises lot of challenges with respect to previous and traditional methodologies . In this chapter will highlight the :

- Technology challenges
- Verification technology options
- Verification methodology
- Verification languages
- Verification approaches
- Verification plans

2.1. challenges of a technology

The physical dimensions of silicon structures got continuously shrunk by silicon technology foundries. Due to this shrinkage ,both circuit capacity and performance got improved significantly. Moor's law characterizes this technology evolution, It states that integrating logic gates (transistors) onto a single silicon chip doubles every 18 months. When silicon technology reached level 0.25 and bellow of deep sub-micron dsm, significant challenges face the design community. These challenges can be grouped to :

1. **timing closure :** Designing a chip is not enough ,so timing closure must be taken in consideration ,therefor timing closure is important ;because we need to know how fast the chip is going to run, how much time need to receive the responses after applying inputs , how fast the chip is going to interact with the other chips, etc. In semiconductor market , timing closure and time taken to achieve it can dramatically impact the success of the product . therefore, methodologies employed are addressed and strategized to obtain in a faster timing closure along with reasonable design metrics .the main reasons of making this item a challenge:

- increased size of the project :following the moor's law as a result the computational resources required for achieving the timing closure for billion gates become high.
- trade off timing for design techniques :most of design techniques such as power gating, multiple clock and dynamic voltage and frequency scaling are trade off with low power consumption [2].

2. **capacity :** a capacity challenges introduced in DSM technology when millions of gates are integrated onto a single IC using 0.15 um and below technology.in order to cope with this challenges ,the DSM design system must contain the following solutions:
   - block based design :in system on chip solutions, which combine several chips into one device that have thousands of gates .in order to complete the design of the project successfully, the design engineer must carefully plan to meet design timing and specifications, therefore the design must be partitioned. The top level provides the interconnection of the blocks and in the design level down, provides the details of these blocks either in terms of interconnection of sub-blocks or library elements.
   - Design reuse :reusing already designed components for a class of the applications is a method to reduce the design-effort and time, since these blocks are pre-designed and have been certified or validated then there is no need to validate them again, and they are considered as black boxes.

3. **Physical properties :** main challenging key in this feature are signal integrity SI such as cross talk noise, and IR drop and design integrity such as hot electron, electro-migration and self heating. Theses keys were ignored at relaxed geometries, while this last have shrunk, they become more critical so that require a sign-off screen in order to check if any violation exist [3].

4. **Design productivity gap :** the increase of the complexity of Ics is accompanied with introducing more challenges to both design engineer and verification engineer .The ITRS identified a critical design gap [4]. The design productivity lags the design density . See figure 2.1 . As a solution to this challenge is using design reuse strategies.

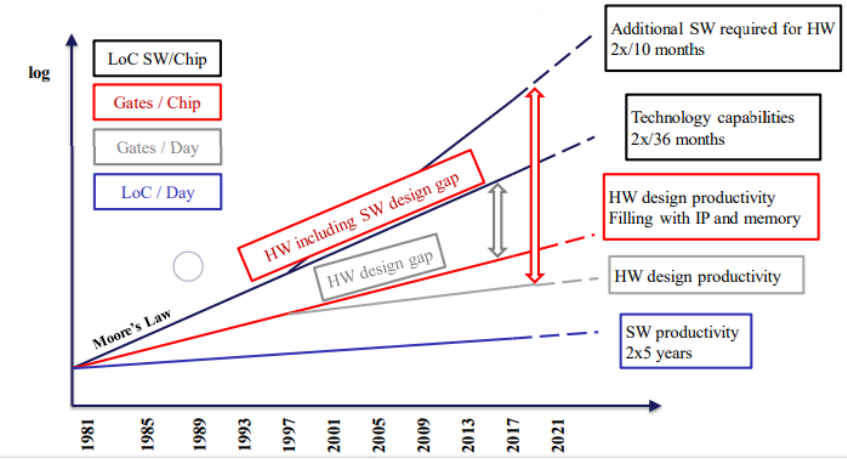


Figure 2.1.: design productivity gap

By reusing pre-existing blocks (also known as virtual component (vc) or intellectual

property (IP)) is reducing time and effort. By adopting platform-based-design ,a set of core elements that are common ,identified ,integrated and verified as a single entity .by adding either newly authored elements or additional IP blocks to this core, the actual products are then built and realized [5].

5. **Time to market trend :** Nowadays ,the development of a product is based on change rapidly,and increasing percentage of demands of new products as the number of competitors for market share do. In the market ,there are core elements which are affecting the revenue,for example ,if your product launched with a delay of 6 months .theses 6 months for your competitors are a chance to grape market share, and less revenue for you to persue when finally go to market. So the better control you have on your development of your products,the better you will have a control and ability to predict TTM and get new technologies out while it is still new and same time have good revenue.
   With market and time pressures plus the evolution affect badly on verification methodologies and tools.the studies and experiences showed that from 40 to 70 percentage of the total development of a product is dedicated to verification tasks. Clearly, these verification activities must be efficient with respect to time.

6. **SOC technology :** The traditional verification methods are neither enough nor efficient for verifying complex and developed SOC which make an other kind of challenges to design and verification engineers.A SOC contains hardware elements,software elements and programmable elements and power distribution ,clock and buses and test structures. Nowadays,a SOC contains different design disciplines (AMS, digital, embedded software (ESW)) That are co-existed in a single design.as a result the verification methodology must deal with analog ,mixed digital signals and mixed hardware/esw verification.

2.2. Technology options

The aim of the verification is to ensure that the design meets the functional requirements as defined in the functional specification. From 40 to 70 percent of the total development effort for the design is dedicated to the verification of SOC devices.There are many issues that challenge both the verification solution providers and the verification engineer such as : is the device verified enough?,what technology options and strategies to use for verification ?and how to plan for it to minimize verification time?.
In the industry,different technology options are available. These options can be divided to four classifications: static technologies, simulation based technologies, formal technologies, and physical verification and analysis. A combination of these methods must be used in order to achieve the SOC verification goals.

2.2.1. Static Technologies

The technologies of static verification such as lint checking and static timing verification verification don't require test vectors or testbench for performing verification.

Lint checking :

Lint checking is a technology of static verification used to carry out a static check of the design code in order to verify the correctness of the syntax. The types of uncovered errors

include unsupported constructs, uninitialized variables, and port mismatches. Simple errors that would be time-consuming are identified by lint checking. So it would be better to perform it in the earlier design cycle.

Static Timing Verification :

Each storage element and latch have timing requirements in a design, such as setup, hold, and various delay timings. Timing verification determines whether the timing requirements are being met. Timing verification is challenging for a complex design, since each input can have multiple sources, and the timing can vary depending on the circuit operating condition.

2.2.2. Simulation Technologies

Simulation technologies include code coverage, event-based and cycle-based simulators, transaction-based verification, AMS simulation, HW/SW co-verification,accelerators, such as rapid prototype systems, hardware accelerators, hardware modelers, and emulation .

Code Coverage :

Code coverage analysis provides you the capability to know the quantity of the functional coverage, that a particular test suite is applied to a specific design. This can at the full-chip level or at the individual block level.

Event-based Simulators :

It performs the simulation by taking events, one at a time and propagating them until a steady state condition is achieved ,through the design. This can be slow for large designs.

Cycle-based simulators :

Cycle-based simulators only function on synchronous logic, because they have no notion for time within a clock cycle maybe this can speed up the process of the simulation but it leads to erroneous results(for combinational circuits) because they evaluate the logic between state element and/or ports.

Transaction-based Verification

Transaction-based verification allows simulation and debug of the design at the transaction level. A detailed testbench with large vectors is not required in the transaction based verification.

Emulation Systems

Emulation systems perform at speeds faster than software simulators, in some instances, can approach the target design speeds, because they are done in hardware

HW/SW Co-verification

In HW/SW co-verification, integration and verification of the hardware and software occurs concurrently. The co-verification environment provides a graphical user interface (GUI) that is consistent with the current hardware simulators and software emulators/debuggers.

Hardware Accelerators

In most common cases, the actual design to be verified is run in the hardware accelerator and the testbench keeps running in software.

Rapid Prototyping Systems

It is offering the ability for developing and debugging software, with a real view of hardware of SOC

AMS Simulation

Due to the complexity of analog designs, less automation provided by analog tools that are available in the industry, it is more complex than both digital-only or analog-only simulation.

2.2.3. Formal Technologies

Usually, it is difficult to detect bugs happening in a specific sequences of events. When we are not able to detect bugs earlier in the verification phase, they may cause serious impact on the design cycle. The exhaustive nature of formal verification and detecting these bugs earlier become main driving forces toward using formal verification techniques. Formal verification methods do not require testbenches or vectors for verification. They theoretically promise a very fast verification time and 100 percent coverage. The formal verification methods are:

* Formal model checking
* Theorem proving technique
* Formal equivalence checking

Formal model checking :

A model checking tool compares the design behavior to a set of logical properties defined by the user,which are extracted from the design specifications. Formal model checking is used to verify behavioral properties of a design using formal mathematical techniques.

2.3. Verification Methodology

Design verification planning should start in parallel with the creation of specifications for the system. System specifications drive the verification strategy. The figure 2.2 shows the high-level verification flow for an SOC device.

2.3.1. System-Level Verification

According to the specifications, the system behavior is modeled in the system design. The system behavior is verified using a behavioral simulation testbench. This last might be created in HDL, C, C++.
After validating the system behavior,the system is mapped to an existed and suitable library. The hardware and software partitioning is done. The testbench should be converted to a suitable format, so it can be used for hardware RTL code simulation and software verification.

	Event based simulation	Cycle based simulation	Hw accelerators	Emulation	Formal verification	Static timing verification
Function	Yes	Yes	Yes	Yes	No	No
Abstraction level	Behavioral, RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	RTL, Gate	Gate
Functional equivalence	Yes	Yes	Yes	Yes	Yes	No
Timing	Yes	No	Yes/No	No	No	Yes
Gate capacity	Low	Medium	High	Very high	High	Medium
Run time	<10 cycles	1k cycles	1K cycles	1M cycles	Medium	High
Cost	Low	Medium	Medium	High	Medium	Low

Table 2.1.: Comparison of verification options

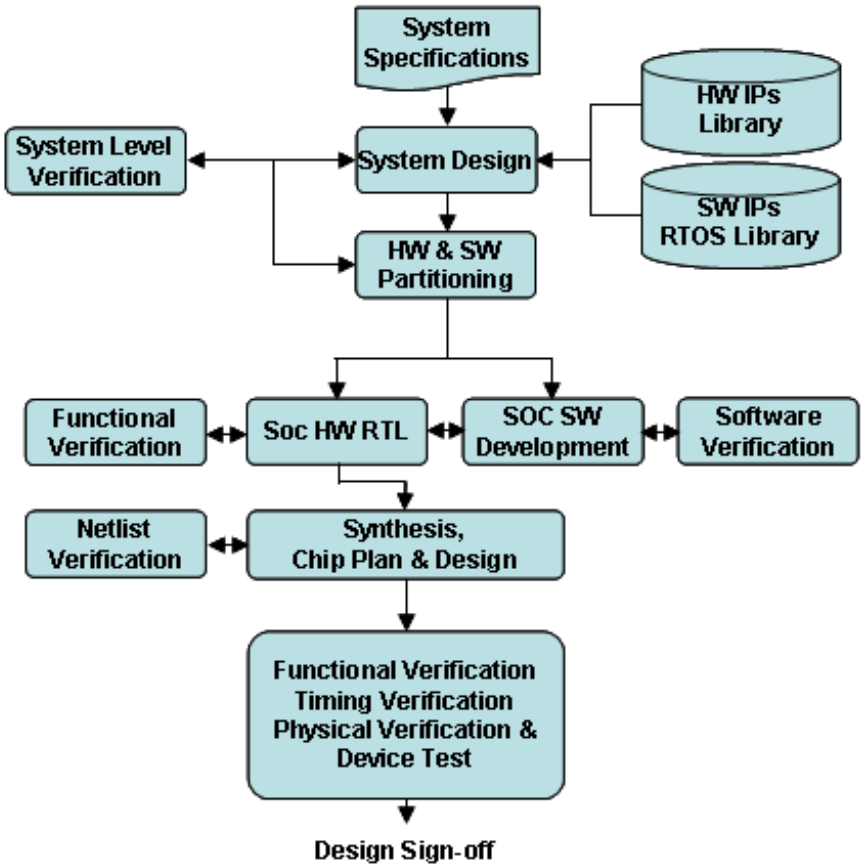


Figure 2.2.: SOC verification methodology

2.3.2. SOC Software Verification

The software team provides the software and test files in order to perform the software verification that is performed against the specifications obtained from the system design.

2.3.3. SOC Hardware RTL Verification

The system design sends the testbench and RTL code to the Hardware verification. The testbench is converted or migrated to a suitable format to verify the RTL code and the design is verified for functionality. The verification mainly focuses on the functional aspects of the design. RTL verification includes lint checking, formal model checking, logic simulation, transaction-based verification, and code coverage analysis.

2.3.4. Netlist Verification

In this phase, the hardware RTL is synthesized and a gate-level netlist is generated. Using formal equivalence checking tool with the reference design that is the RTL code to verify it and the netlist of the gate-level as the implementation design. So this is used in order to ensure that RTL and gate-level are equivalent logically.

2.3.5. Physical Verification

In order to ensure that there are no violations in the implemented design, a physical verification is performed on the chip design. The physical verification includes design rules checking, layout versus schematic, process antenna effects analysis, SI checking, including crosstalk, and current-resistance (IR) drop.

2.3.6. Device Test

The final device test uses the test vectors that are generated during the functional verification. The device test checks if the device was manufactured correctly. In this stage, it is focused on the structure of the chip like the gate truth tables, connections. By using an automatic test pattern generator(ATPG) tool Vectors are generated for manufacturing the device test using the testbench created during functional verification. After getting satisfied about the results then the design is ready for fabrication sign-off and tape-out.

2.3.7. Verification IP Reuse

Because of the pressures of the TTM on product design cycles,It forces SOC designers and integrators to reuse available design blocks. Verification teams put a lot of effort into developing testbenches. If the testbenches developed for one design can be used for other similar designs, a significant amount of the verification time is saved for subsequent designs.

2.4. Verification Approaches

There are different verification approaches. These include top-down design and verification, bottom-up verification, platform-based verification, and system interface-driven verification.

2.4.1. Top-Down Design and Verification Approach

The functional specification is the starting point for any top-down design. A detailed verification plan is developed from the functional specification.
we are functionally verifying the system level model by exercising it with the system level test bench. The design can be decomposed through any number of abstraction level until the detailed design is complete. Using the system testbench, the design is verified a the upper abstraction levels.

2.4.2. Bottom-Up Verification Approach

Nowadays, this approach is widely used in the design field. The first step in this approach is validating the incoming data of the design by passing the files through a parser to ensure that they are compatible with target tools. The second step is passing the files of the design through a lint checker.
The next steps depend on the abstraction level of the design as it is shown in the figure. Verification levels are defined as follows:

- **Level 0**:the blocks, individual components, or units are verified in this level separately. The goal is to test the component exhaustively without considering the environment into which it will be integrated. The technologies and techniques used in unit test are similar to those applicable to an integrated design : directed random simulation, lint, deterministic simulation, and formal checking.
- **Level 1**: in this level the system memory map and The internal interconnect of the design are verified. By performing writes and read-backs all the interconnect within the design are verified.
- **Level 2**: At this level, it is verified basic functionality of the design and the external interconnect.
- **Level 3**:The intent is to test the functionality of the integrated design exhaustively.

After the above tests, the netlist verification, timing verification, physical verification, and device tests are performed to ensure correct chip implementation.

2.4.3. Platform-based Verification Approach

for verifying the derivative designs that using a verified preexisting platform.

2.4.4. System Interface-driven Verification Approach

In this approach, at the interface level, of the blocks that are planned to be used are modeled during system design. These models, along with the specifications for the blocks to be designed and verified, are handed off to the design team members. The interface models can be used by the verification engineers to verify the interface between the designed block and the system. This eases the final integration efforts and enables early detection of errors in the design.

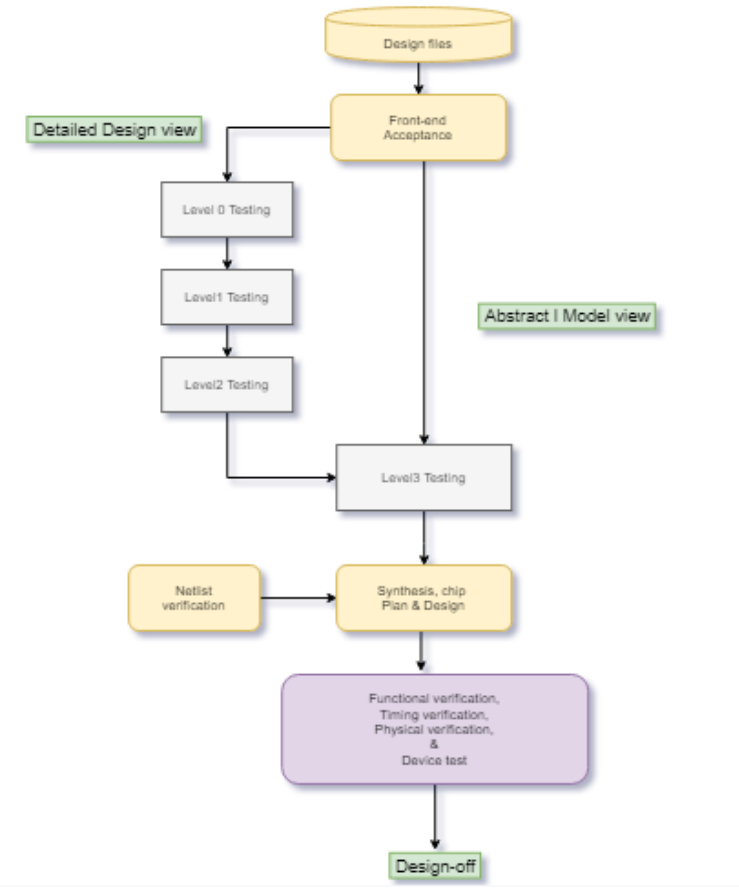


Figure 2.3.: Bottom-UP verification approach

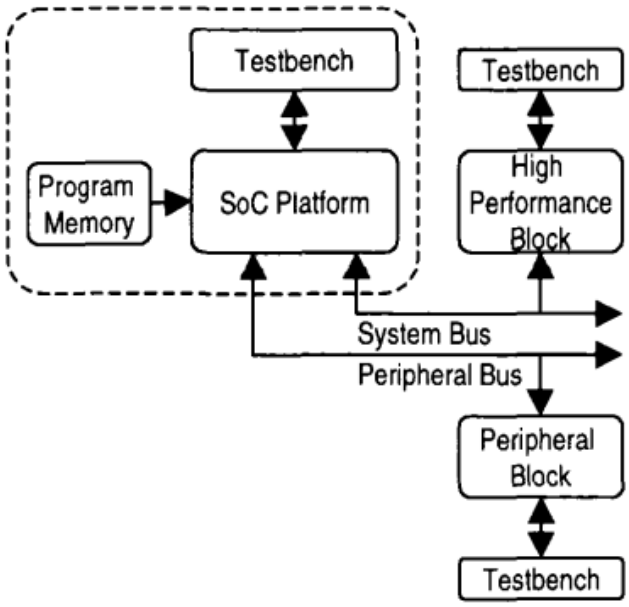


Figure 2.4.: Platform-based verification approach

Chapter 3.

Universal verification methodology UVM

In verification technology,the latest advancement is UVM.UVM is a new verification methodology .It is designed to enable creation of robust,reusable,interoperable verification IP and testbench components .
UVM has lot of exiting aspect like how is developed .UVM is not rolled out as a part of a marketing campaign, because a collection of industry experts participated on developing this technology like verification consultants, networking companies, microprocessor companies ,as well as EDA vendors.Accellera was the responsible of taking the auspices of this great work.UVM succeed to unify lot of competitors in the market place in order to collaborate to build a sophisticated verification methodology see 3.1 .

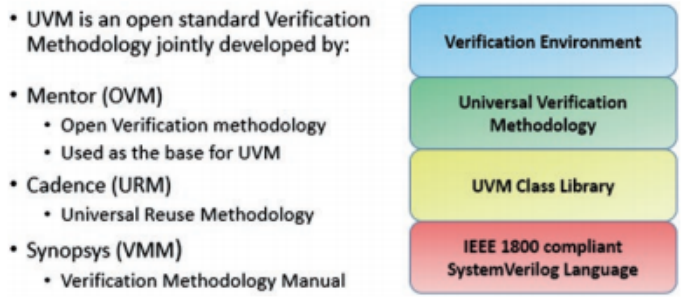


Figure 3.1.: UVM evolution

The result is a powerful, multi-dimensional software layer and methodology for building verification environments [6].The culmination of lot of indepent efforts in the verification space represent UVM,which means based on union of other technologies ,we get as a result a powerful verification methodology.Its heritage includes AVM, URM, VMM, and OVM figure 3.2.
UVM is very close to OVM ,because OVM was the starting point to build UVM and UVM is compatible with OVM .In UVM ,the register facility is a transformation of the RAL package which was part of VMM.UVm is the fruit of combining these methodologies and it is not just a conglomeration of code drawn from its predecessors.This fruit provides new facilities and new use models for testbench construction,as a result the stat-of-the-art is moved forward.
UVM is transaction level methodology (TLM). UVM is a derived class library that makes it easy to write a configurable and reusable code.It is based on Object Oriented Programming (OOP), but UVM designers did the whole hard work to simplify it .so you don't need to be an OOP expert; by creating the so co-called class library whose components an be used to develop a testbench.In other words,when you put together the required code in place,you will be able to go forward to next project because you still can reuse the previous code

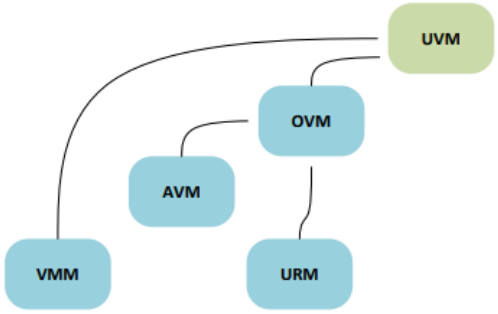


Figure 3.2.: heritage of UVM

whiteout modifying it ,you just derive from that class. However ,only few components such driver,scoreboard and the basic transaction(sequence) need to be changed.Hence,as a solution to the challenge of communication between interfaces ,the UVM designers make UVM classe based, so it communicates between these classes via transaction .

3.1. UVM Hierarchy

Don't you understand some of this? hold on, we will go into UVM hierarchy and examples to solidify the concepts.
The UVM class library gives you the possibility to use them for generic utilities like component hierarchy,transaction library model(TLM) ,configuration of database,etc..,which enable the user to create any structure he wants for the testbench. In this figure 3.3 is from [(Accelera, Universal Verification Methodology (UVM) 1.2 User's guide)] which shows a simple hierarchy which is composed of

* UVM Testbench
* UVM Test
* UVM Environment

3.1.1. UVM Testbench

Testbench instanciates the Device Under Test (DUT) and Test class and configures the connection between them.In UVM ,TLM interfaces provide a set of communication methods that is consist to send and receive transactions between components.These components are instantiated and connected in the testbench in order to perform the different operations required to verify a design.
The UVM Test is instantiated dynamically at run-time, so it allows the UVM testbench to be compiled once and run with many different tests.

Transacton level Testbench of UVM

figur 3.4 Thisis the most basic testbench using a UVM agent that comprises of the sequencer,the driver and the monitor A scoreboard is used to analyze data is also instantiated. The components of this Tesbench are:

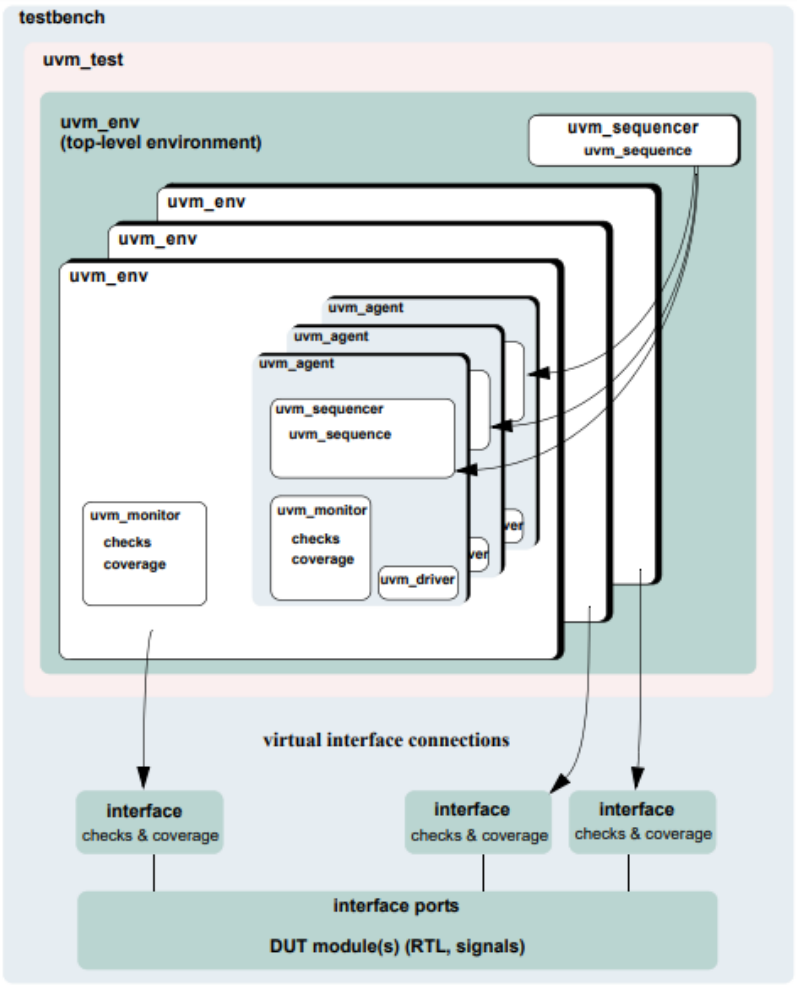


Figure 3.3.: The hierarchy of UVM

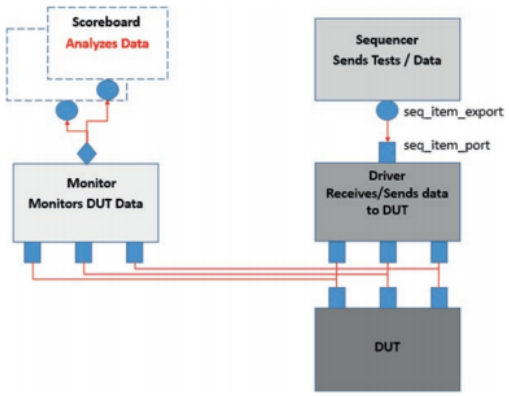


Figure 3.4.: UVM transaction level Testbench

1. the Sequencer: Stimulus Generator,creates transaction -level traffic to send them to the driver.
2. The driver: It takes these transactions from the sequencer and then converts them into pin signal -level activity,and drives the DUT
3. The monitor: snoops the signal -level activity and converts them back transactions that are sent to a scoreboard.
4. The scoreboard: gets the monitored transactions from the monitor comparesthem with expected transactions (response transactions).

3.1.2. UVM Test

In the UVM testbench, the UVM-test is the top level UVM component. A test is a class that encapsulates test-specific instructions written by the test writer. Tests in UVM are classes,they are derived from uvm -test class.By using classes ,inheritance and reuse of test is allowed. The test instantiates the top-level environment just like any other verification component.
The uvm test :

1. Instantiate the top-level environment.
2. Configure the environment (via factory overrides or the configuration database).
3. Apply stimulus by invoking UVM sequences through the environment (typically one per DUT interface) to the DUT.

3.1.3. UVM Environment

The UVM environment is a component that groups together other verification components that are interrelated.The components that are usually instantiated inside the UVM environment are UVM agents, UVM scoreboards, or even other UVM environments because the top-level environment contains one or more environments. Each environment contains an agent for a specific DUT interface which means that each interface to the DUT,might have separate environment per interface.This top -level environment instantiates and configures the reusable verification IP and defines the default configuration of that IP required by the application. Some of these IP environments can be grouped together into cluster environments (e.g., an IP interface environment, CPU environment, etc.).

3.1.4. UVM Agent

The UVM agent is a component that groups together other verification components that are dealing with a specific DUT interface. Agent contains a UVM sequencer to manage stimulus flow, a UVM driver to apply stimulus to the DUT interface, and a UVM monitor to monitor the DUT interface. UVM agents might include other components, like coverage collectors, protocol checkers, and a TLM model.
As mentioned before, the UVM agent is the component that drives the signal-level interface of the DUT. The agent can operate in an active mode or a passive mode. In the active mode, it can generate the stimulus (i.e., the driver drives DUT input and senses DUT outputs). In the passive mode, the driver and the sequencer remain silent (disabled) and only the monitor

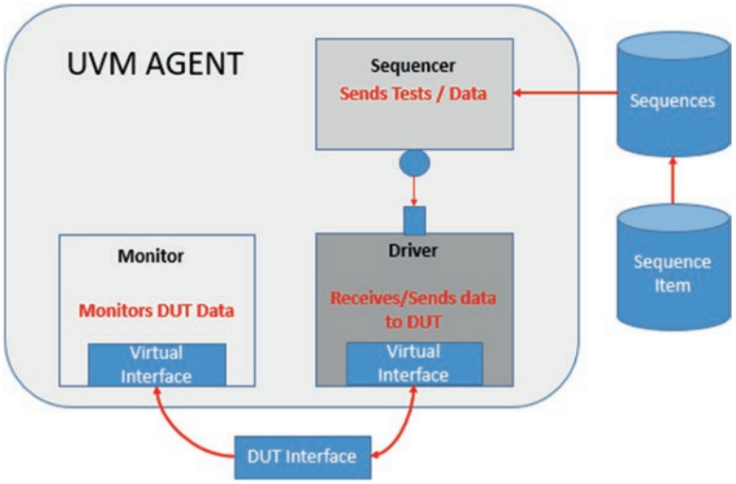


Figure 3.5.: UVM Agent

remains active. Monitor simply monitors the outputs of DUT;it cannot control the IO of the DUT. You can dynamically configure an agent in either an active mode or a passive mode. Monitor is an unidirectional interface,while driver is a bidirectional interface. This is depicted in Figure 3.5

3.1.5. UVM Driver

Driver is where the TLM transaction-level world meets the DUT signal/clock/ pinlevel world. Driver receives sequences from the sequencer, converts the received sequences into signal -level activities, and drives them on the DUT interface as per the interface protocol. Or the driver pulls sequences from the sequencer and sends them to the signal-level interface. An other block (The monitor) will observe and evaluate This interaction.As a result, functionality of the driver should only be limited to send the necessary data to the DUT. Note that nothing prevents the Driver from monitoring the transmitted/received data from DUT—but that violates the rules of modularity. Also, if you embed the monitor in the driver, you can't turn the monitor ON/OFF.
The driver has a TLM port to receive transactions from the sequencer and access to the DUT interface to drive the DUT signals.
Driver is written by extending uvm-driver.uvm-driver is inherited from uvm-component; Methods and TLM port (seq-item-port) are defined for communication between sequencer and driver.The uvm-driver is a parameterized class; and it is parameterized with the type of the request sequence-item and the type of the response sequence-item.

3.1.6. UVM Monitor

Monitor, is reverse of the driver. It takes the DUT signal/pin-level activities and converts them back into transactions to be sent out to the rest of the UVM testbench (e.g., to the scoreboard) for analysis. Monitor broadcasts the created transactions through its analysis port. Note that comparing of the received output from the DUT to that with expected output is normally done in the scoreboard and not directly in the monitor.
The reason is to preserve modularity of the testbench. Monitor, as the name suggests,monitors the DUT signals and coverts them to transactions. That's it. It's the job of the scoreboard

to receive the broadcasted transaction from the Driver and do the comparison with the expected outputs.

3.1.7. UVM Scoreboard

The scoreboard simply means that it is a checker (not to be confused with SystemVerilog SVA "checker"). It checks the response of the DUT against expected response. The UVM scoreboard usually receives transactions from the monitor through UVM agent analysis ports and the transactions through a reference model to produce expected transactions and then compares the expected output versus the received transaction from the monitor.
There are many ways to implement a scoreboard. For example, if you are using a reference model, you may use SystemVerilog–DPI API to communicate with the scoreboard, pass transactions via DPI to the reference model, convert reference model response into transactions, and compare the DUT output transaction with the one provided by the reference model. Reference model can be a C/C++ model or a TLM2.0 SystemC model or simply another SystemVerilog model.

3.1.8. UVM Sequencer

The sequencer controls the flow of request and response sequence items between sequences and the driver. UVM sequencer is a simple component that serves as an arbiter for controlling transaction flow from multiple stimulus sequences. The sequencer and driver use TLM interface to communicate. uvm-sequencer and uvm-driver base classes have seq-item-export and seq-item-port defined respectively. The user needs to connect them using TLM connect method.

3.1.9. UVM Sequence

After a basic uvm-sequence-item has been created, the verification environment will need to generate sequences using the sequence item to be sent to the sequencer.
Sequences are an ordered collection of transactions (sequence items); they shape transactions to our needs and generate as many as we want. Since the variables in the transaction (sequence item) are of type "rand," if we want to test just a specific set of addresses in a master-slave communication topology, we could restrict the randomization to that set of values instead of wasting simulation time in invalid (or redundant) values.
Sequences are extended from uvm-sequence, and their main job is generating multiple transactions. After generating those transactions, there is another class that takes them to the sequencer.

3.1.10. UVM Sequence item

UVM sequence item (i.e., a transaction) is the fundamental lowest denominator object in the UVM hierarchy. It is the definition of the basic transaction that will then be used to develop UVM sequences.
The sequence item defines the basic transaction data items and/or constrains imposed on them. While the driver deals with signal activities at the bit level, it doesn't make sense to keep this level of abstraction as we move away from the DUT, so the concept of transaction was created.
UVM sequence items, i.e., transactions are the smallest data transfers that can be executed in

a verification model. They can include variables, constraints, and even methods for operating on themselves.

3.2. UVM Class Library

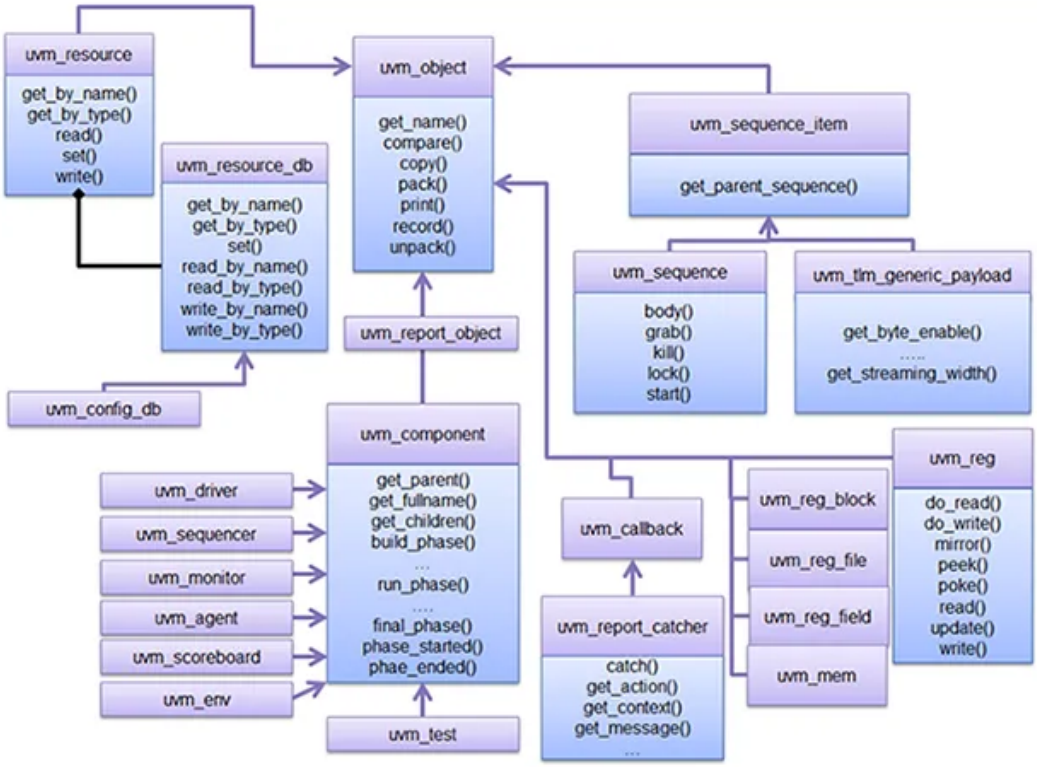


Figure 3.6.: UVM Class library

shows the building blocks of UVM class library that you can use to quickly build well constructed, reusable, configurable components and testbenches. The library contains base classes, utilities, and macros.
The advantages of using the UVM class library [7] include:

(a) Many features are provided by the UVM class library, those features are required for verification.

(b) The component can be derived from a corresponding UVM class library component. By using these base class elements, it increases the readability of your code since each component's role is predetermined by its parent class.

3.3. UVM Phases

In UVM ,phases are defined as callback methods,uvm-component provides a set of predefined phases and corresponding callbacks.The Method can be either a function or task.Methods that consumes simulation time are Tasks while methods that they don't consume simulation time are functions.May more than one callback will be implemented if the class is derived

from uvm-component.These callbacks are executed in order.
Basically,the UVM phases have three phases:

* Build phase:builds top-level testbench topology.
* Connect phase:connects environment topology
* Run phase: run the test
* Cleanup phase. gathers details on the final DUT state processes the simulation results, and does simulation results analysis and reporting

Run phase includes many different sub-phases ,all of them are run in Zero time ,except of course run() phase.

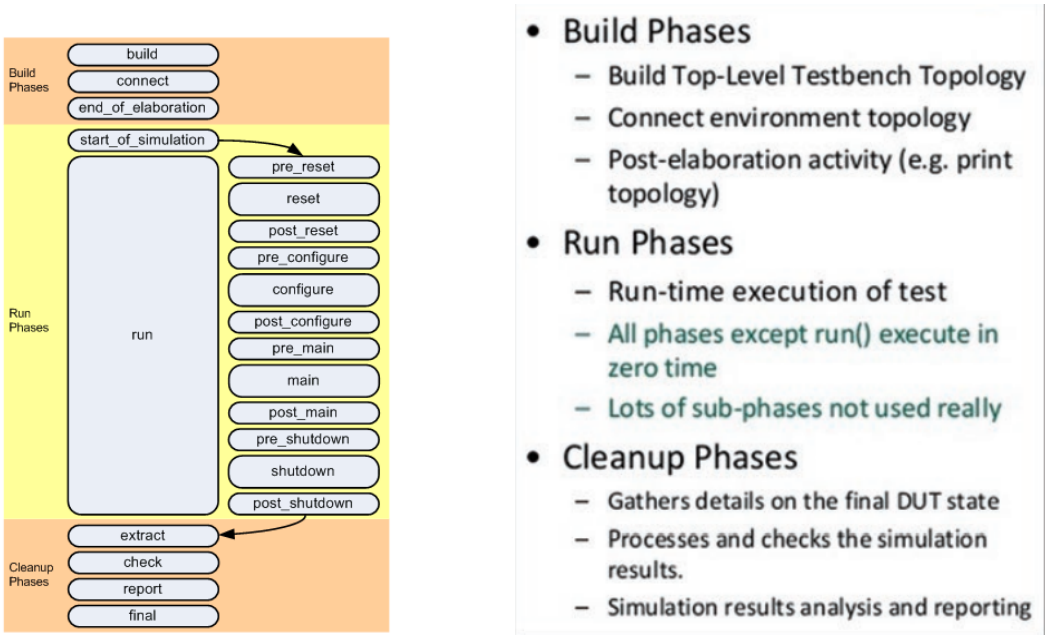


Figure 3.7.: UVM phases

3.3.1. Build phase

At the start of the UVM testbench simulation,the builds phases are executed ,so the aim of these phases is to construct, configure and connect the testbench component hierarchy.All the build phase methods are functions and therefore execute in zero simulation time.[8]

build

Once the UVM testbench root node component is constructed, the build phase starts to execute. It constructs the testbench component hierarchy from the top downwards. The construction of each component is deferred so that each layer in the component hierarchy can be configured by the level above. During the build phase uvm-components are indirectly constructed using the UVM factory [8].

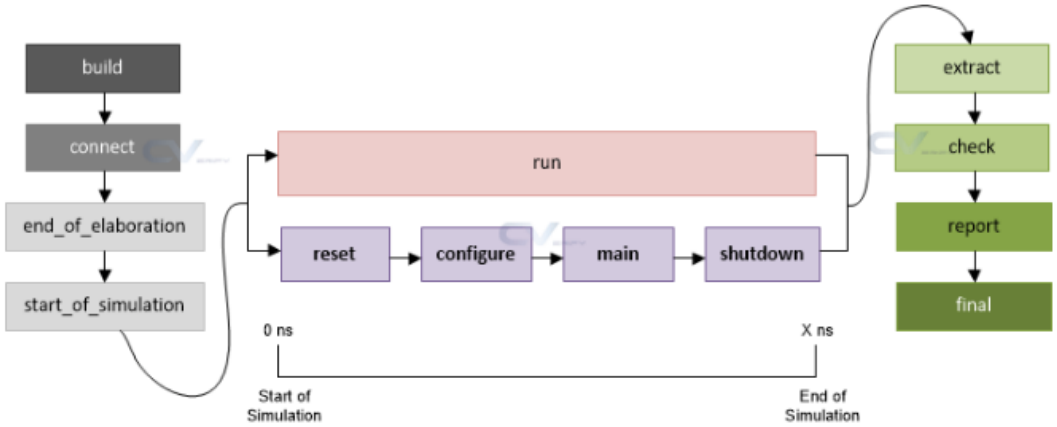


Figure 3.8.: Run phase

connect

The connect phase is used to make TLM connections between components or to assign handles to testbench resources. It has to occur after the build method has put the testbench component hierarchy in place and works from the bottom of the hierarchy upwards [8].

end-of-elaboration

The end-of-elaboration phase is used to make any final adjustments to the structure, configuration or connectivity of the testbench before simulation starts. Its implementation can assume that the testbench component hierarchy and inter-connectivity is in place. This phase executes bottom up.

3.3.2. Run-Time Phases

The testbench stimulus is generated and executed during the run time phases which follow the build phases. After the start-of-simulation phase, the UVM executes the run phase and the phases pre-reset through to post-shutdown in parallel. The run phase was present in the OVM and is preserved to allow OVM components to be easily migrated to the UVM. It is also the phase that transactors will use. The other phases were added to the UVM to give finer run-time phase granularity for tests, scoreboards and other similar components. It is expected that most testbenches will only use reset, configure, main and shutdown and not their pre and post variants [8].

start-of-simulation

The start-of-simulation phase is a function which occurs before the time consuming part of the testbench begins. It is intended to be used for displaying banners, testbench topology, or configuration information. It is called in bottom-up order[8].

Run

The run phase occurs after the start-of-simulation phase and is used for the stimulus generation and checking activities of the testbench. The run phase is implemented as a task, and all

uvm-component run tasks are executed in parallel.Transactors such as drivers and monitors will nearly always use this phase [8].

pre-reset

The pre-reset phase starts at the same time as the run phase. Its purpose is to take care of any activity that should occur before reset, such as waiting for a power-good signal to go active [8].

Reset

The reset phase is reserved for DUT or interface-specific reset behavior. For example, this phase would be used to generate a reset and to put an interface into its default state [8].

post-reset

The post-reset phase is intended for any activity required immediately following reset. This might include training or rate negotiation behavior [8].

pre-configure

The pre-configure phase is intended for anything that is required to prepare for the DUT's configuration process after reset is completed, such as waiting for components (e.g., drivers) required for configuration to complete training and/or rate negotiation. It may also be used as a last chance to modify the information described by the test/environment to be uploaded to the DUT [8].

Configure

The configure phase is used to program the DUT and any memories in the testbench so that it is ready for the start of the test case. It can also be used to set signals to a state ready for the test case start [8].

post-configure

The post-configure phase is used to wait for the effects of configuration to propagate through the DUT or for it to reach a state where it is ready to start the main test stimulus. I do not anticipate much use for this phase [8].

pre-main

The pre-main phase is used to ensure that all required components are ready to start generating stimulus [8].

Main

This is where the stimulus specified by the test case is generated and applied to the DUT. It completes when either all stimulus is exhausted or a time-out occurs. Most data throughput will be handled by sequences started in this phase [8].

post-main

This phase is used to take care of any finalization of the main phase [8].

pre-shutdown

This phase is a buffer for any DUT stimulus that needs to take place before the shutdown phase [8].

Shutdown

The shutdown phase is used to ensure that the effects of stimulus generated during the main phase have propagated through the DUT and that any resultant data has drained away[8].

post-shutdown

Perform any final activities before exiting the active simulation phases. At the end of the post-shutdown phase, the UVM testbench execution process starts the cleanup phases [8].

3.3.3. Cleanup Phases

The cleanup phases are used to extract information from scoreboards and functional coverage monitors to determine whether the test case has passed and/or reached its coverage goals [9]. The cleanup phases are implemented as functions and therefore take zero time to execute. They work from the bottom upwards in the component hierarchy [8].

Extract

The extract phase is used to retrieve and process information from scoreboards and functional coverage monitors. This may include the calculation of statistical information used by the report phase. This phase is usually used by analysis components [8].

Check

The check phase is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the testbench. This phase is usually used by analysis components [8].

Report

The report phase is used to display the results of the simulation or to write the results to file. This phase is usually used by analysis components [8].

Final

The final phase is used to complete any other outstanding actions that the testbench has not already completed. Here's a very simple example of a basic uvm-component showing different UVM phases [8].

3.4. summary

The system used to verify the functionality of a circuit design DUT comprises :

* A control station which comprises at least one graphical user interface (GUI).
* An emulator that is in communication with the control station.The emulator is composed of verification component and a register abstraction layer(RAL), where the verification component is configured to implement the DUT and where the RAL is configured to implement one or more communication interface of DUT. The emulator generates a transaction stream for a communication interface, the transaction stream is composed of many transactions. The transaction are associated with the commands of the communication interface and test data associated with commands.
* Sending the transaction stream to the dut via the communication interface.
* One or many monitors that are associated responses sent from the DUT via the communication interface.
* A RAL painter that classifies the transaction and responses based responses based upon one or more characteristics of the transactions and the responses.
* Generating a graphical representation of the transactions and responses based upon the classification.
* Displaying the graphical representation on the control station GUI.[9]

Chapter 4.

UVM specifications

The chapter 3 describes the UVM components and their hierarchy, in this chapter we will go deeply in each component and know how to describe them and then write their code.
All these details and information are from Mentor-SEIMENS videos, the `UVM_guide_user` and UVM-cookbook, and it is an explanation of the code of next chapter.

4.1. UVM-SV-Glossoary

Before going on, It would be better to give a definition to most important vocabularies used in OOP:

Class : contains related features and functionality that are in an usable block. It contains definitions for variables and routines that operate on those variables. No memory allocated. **To simplify it, It is like a blueprint of a house.**

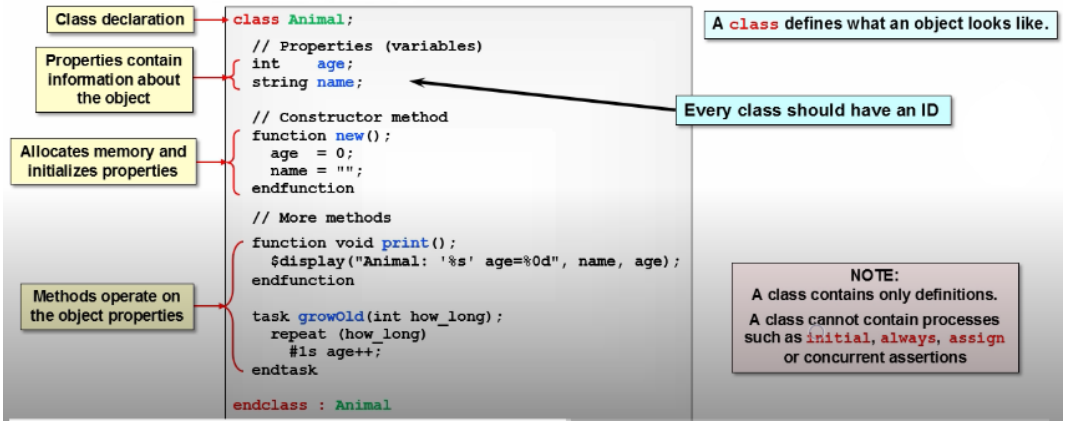


Figure 4.1.: structure of a class [1]

Property : variable declared in the class.**light switches**.

Method : task or function in a class. **turn ON/OFF switches**.

Members : methods and properties in a class (because, it is unfeasible to use the class by it self, so it should be instantiated. The instance of the class is an object).

Object : instance of class that can be used. Memory allocated. **a complete house**.

Handle : type-safe printer to an object (systemverilog). So no worries about interrupting accidentally the handler.

Encapsulation : As mentioned before, a class contains properties and methods that operate on them. After defining a class, we declare the handle variable then construct the object using new() method. It allocates memory for the object. new() returns the object location which assigned to the handle.

4.1.1. Accessing members

An object's properties and methods are accessed with the handle and dot_operator(.)
OOP languages such C++ and JAVA recommend that properties should only be accessed

```
module top();
  class Animal;
    int    age;
    string name;
    function print();
      $display("Animal: '%s' age=%0d", name, age);
    endfunction
    task growOld(int how_long);
      repeat (how_long) #1s age++;
    endtask
  endclass
  Animal a_h;              // Handle to Animal objects
  initial begin
    a_h = new();           // Construct an Animal object
    a_h.name = "Fluffy";   // Assign an object's property
    a_h.growOld(3);        // Access an object's method
    a_h.print();           // Access an object's method
  end
endmodule
```

```
Animal: 'Fluffy' age=3
```

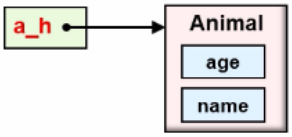


Figure 4.2.: structure [1]

through methods and never directly; with systemverilog testbench, you need direct access properties for generating randomized values and injecting errors.

Aggregation Composition : a class contains a reference to a handler class `has_a` relationship.

Construct : build an object directly by calling " new()" method

Base class or Derived class : extend a base class to make a derived class

OOP hierarchy : relationship between base and derived class `is_a` relationship.

Create : construct an object indirectly with the UVM factory instead of calling new(). there are two major groups of classes: transaction and component

Transaction : stimulus item, continually created and destroyed during the simulation. Test generates stimulus, send them to DUT, monitor catches the results and sends them to scoreboard.

Component : testbench unit (driver,monitor..) created at start of simulation and it exists for entire simulation

UVM hierarchy : relationship between UVM components;test has an environment, env has an agent,...

Testbench topology : is not an official term for testbench but it describes the relationship between UVM components `has_a` relationship.

4.2. UVM : sequence_item, sequence, sequencer, transaction, virtual sequence

First of all, we should distinguish between sequence_item, sequence, sequencer and transaction:

- **sequence_item**: contains the properties and methods of the transaction .
- **Sequence**: generates a series of sequence items.
- **Sequencer**: arbitrates/routes one or more sequences.
- **Driver**: sends sequence item objects to the DUT.

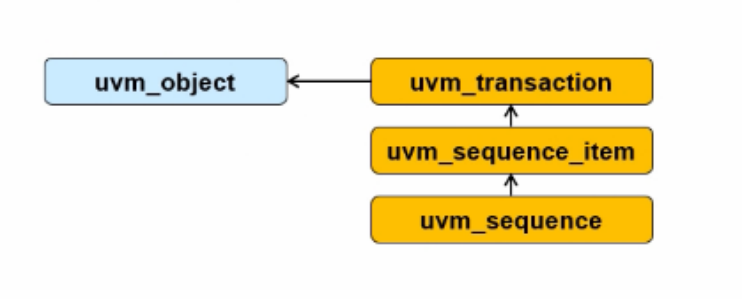


Figure 4.3.: sequencer and sequence

Creating individual transactions and connecting them together to a sequence is generated in the test level then sent them to the lower testbench components in the design under test DUT.
The base class of all UVM classes, is UVM_object and the individual transaction is known as UVM_sequence_item and the sequence is a collection of sequence items collected together to describe a stimulus. The transactions are flown to the sequencer which arbitrates between sequences.
When first time read about UVM component, is easy to confuse between **sequencers** and

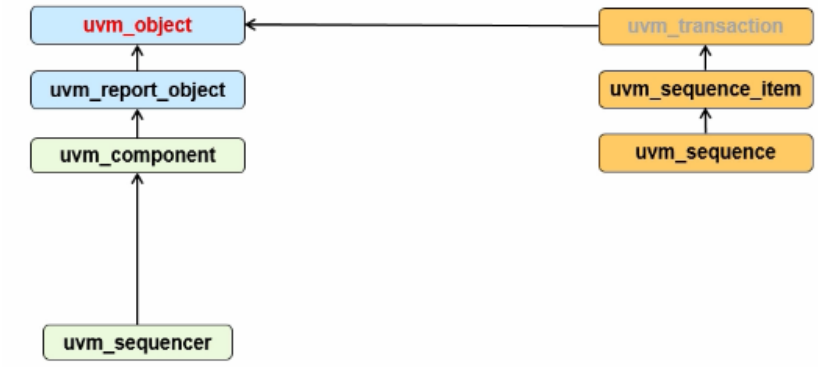


Figure 4.4.: The base object

sequences. Sequencer is actually a pipe that connects sequences to the driver then sends them to the DUT.

A virtual sequence is a sequence that controls the execution of other sequences and almost never generates `sequence_items` itself. This is different from a sequence library which is a sequence that lets you pick from a group of sequences that are registered with the library. A virtual sequence contains pointers to agent sequencers so you start other sequences on those sequencers and is not recommended to use virtual sequences. If you want to create configurable streams of stimuli, coordinated with other streams:

- The `UVM_sequence_item` class has no member to access its context and configuration, eventually, building a `UVM_sequence` that can be generated by a sequencer and send to multiple drivers.
- `UVM_transaction` is an isolated object with no context.

4.2.1. Code styling of Transactions

A transaction class hold single unit of stimuli such as bus transaction that work in packets or a processor instruction.

```
1          class tx_item extends UVM_sequence_item
2           `uvm_object_utils(tx_item)
3            function new (string name = "tx_item");
4            super.new(name);
5            endfunction
6            rand bit [31:0] src;
7            rand bit [31:0] dst;
8            rand command_t cmd;
9            logic [31:0] result;
10           ...// sequence  in the method class
11         endclass
```

- Transactions are extended from `UVM_sequence_item`
- The first line in transaction class should be a macro: `` `uvm_object_utils(tx_item) ``, it creates the code directs new UVM factory, no semicolon needed in the end of the line. This macro is used to build and substitute the object.(register the class in UVM factory)
- The constructor that has a single argument name which must have a default value :
  - should be the same as the class name
  - the actual value is passed into the function by the factory

```
1                function new (string name = "tx_item");
2                   super.new(name);
3                endfunction
```

- This piece of code it is used for describing properties; So it stores the transaction values and send them to the DUT and read back from DUT.

```
1              rand bit [31:0] src;
2              rand bit [31:0] dst;
```

All inputs should be randomized for maximum controllability, it is unfeasible to use randomized values by yourself. However, if you don't write **rand**, it cannot be randomized rather.
Since the randomization is based on writing 0s or 1s then it is better to declare DUT inputs properties as randomized 2-states types and for output don't need to be randomized, but you need to declare them 4-states type since it may contain not only 0s and 1s but also Xs and Zs

- class methods is used for : copy(),compare(),...

```
1                ...// sequence  in the method class
```

A transaction might contain many properties into the DUT so to organize them:

- store stimulus values to send into the DUT
- store outputs that are read back from the DUT
- store predicted results that are expected from DUT output

4.2.2. code style of the transaction class

Style 1

Same sequence item has both DUT inputs and outputs, however the class may become huge.

```
1      class tx_item extends UVM_sequence_item
2           //DUT inputs(stimulus)
3       rand bit [31:0] src;
4       rand bit [31:0] dst;
5           //DUT outputs and predicted output values
6      logic [31:0] result;
7       ...// transaction methods
8       endclass
```

Style 2

The problem here, that you may need to access both sets predictor and scoreboard.

```
1       class tx_in extends UVM_sequence_item
2          rand bit [31:0] src;
3          rand bit [31:0] dst;
4             ...// transaction methods
5       endclass
```

```
1       class tx_out extends UVM_sequence_item
2          logic [31:0] result;
3          ...// transaction methods
4       endclass
```

Style 3

So as a solution to the previous style 2, you can use a base sequence item that it is extended for inputs and outputs. So you share common methods in the base class and declare the inputs and outputs transaction in different classes.

```
class tx_base extends UVM_sequence_item
      ...//  common variables / methods
endclass
```

```
class tx-in extends tx_base;
   rand bit [31:0] src;
   rand bit [31:0] dst;
      ...// extended methods
endclass
```

```
class tx_out extends  tx_base;
    logic [31:0] result;
       ...// extended methods
 endclass
```

4.2.3. Methods on transaction class

operating on transactions

for example: a scoreboard compares an expected and actual transactions. UVM provides standard sets of methods that can build and operate on a new transaction classes.
transaction inherit these methods from `UVM_object`

- **compare()**: deep compare two transactions, it returns 1 for match and 0 for mismatch.

```
if(actual.compare(expected))...
```

- **copy()**: deep copy of transaction.

```
dst.copy(src);
```

- **create()**: creates new object and returns a handle
- **clone()**: is a method that calls **create()** then **copy()**. This method created in `UVM_object` and it returns the handle of type `UVM_object`.

```
dst=src.clone();
```

  Waiiit!!!! don't you see anything wrong in the syntax ? Because clone() is method from `UVM_base_object` and it returns back the handle of type `UVM_object` it is impossible to assign that handl to the drive handle dst.

```
if(!$cast(dst,src,clone()))....;
```

  A dynamic cast needed to check the type of the object in run time and see if compared with dst handle . Remember to check $ cast results in case of testbench bugs.

- **convert2string()**: is a method that converts to a string. like print in language C. Its format is

```
1            `uvm_info("BUILD", tx.convert2string()...);
```

- **print()** and **sprint()**: display transactions properties except that sprint() returns the resulting string.
- **record()**: deep records() the transaction information and is used by simulators for debugging analysis.
- **pack()**: concatenates the transaction into a packed array of bits.
- **pack_bytes()** and **pack_int()** : is the same as pack() but the first one concatenates byte by byte and second one int.
- **unpack()**: is the opposite of pack(). Used to extract the transaction properties from a concatenated packed array of bits.
- **unpack_bytees()**, **unpack_int()**.

These methods are important !!
UVM enables scalability and reusability by requiring that all transactions have a standard set of methods. **compare(), copy() and convert2string()** are the most methods used in transaction class.
DO I have to write 14 methods for every transaction type? IT'S TOOO MUCH
each of these pre_defined methods are non_virtual and they can call a set of virtual methods. We only need to create the virtual ones.
UVM provides two ways to create these virtual methods:

- use do_() method by hand which is flexible and more precise.
- The alternative way is by using a set of macros which are more quick to implement, slower, less debug needed and quirky syntax.
  Don't ever mix between these two methodes in one class.

4.2.4. Relationship between non_virtual and virtual methods

in the following, it shows the relationship between virtual and non_virtual methods, so when a transaction class is created, it is possible to call them. The difference only in convert2sting that it can be called directly.

Implementing do_() methods

As first question might come to mind where do the do_() methods go? let's take this example :

```
1           class tx_item extends UVM_sequence_item
2            `uvm_object_utils(tx_item)
3            function new (string name = "tx_item");
4             super.new(name);
5            endfunction
6             rand bit [31:0] src;
7             rand bit [31:0] dst;
8             rand command_t cmd;
9             rand tx_playload pay_h;     // contaned object
10                logic [31:0] result;
```

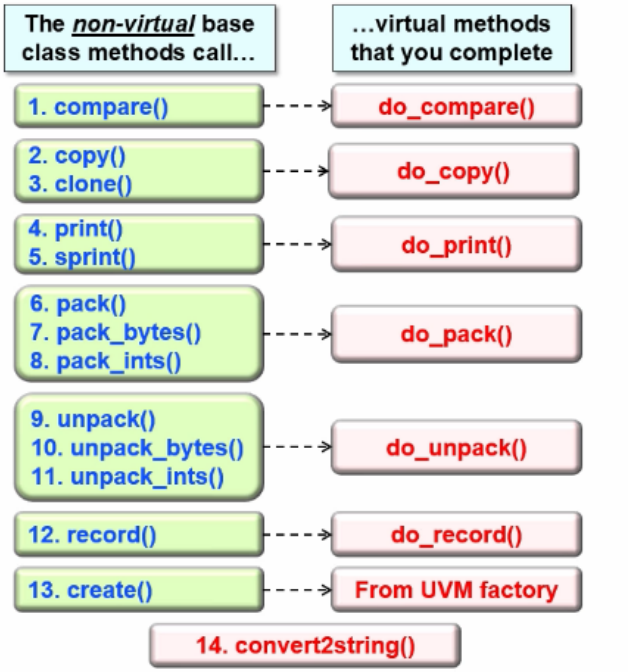


Figure 4.5.: Relationship between virtual and non virtual methods [1]

```
11
12          virtual function bit do_compare(...);  // do deep compare
13          virtual function void do_copy(...)   ;  // do deep copy
14          virtual function bit do_print(...)   ;  // do deep print
15          virtual function bit do_pack(...)    ;  // do deep pack
16          virtual function bit do_unpack(...) ;  // do deep unpack
17          virtual function bit do_record(...) ;  // do deep record
18          virtual function string conevrt2string(); // your method
19            endclass
```

`tx_item` is a class extended from `UVM_sequence_item` and after the registration factory and constructing the function, the properties are defined. The `pay_h` is defined from another class called `tx_playload` which extended from `UVN_sequence_item`

```
1           class tx_payload extends UVM_sequence_item
2            //UVM object macro and constructor
3            // properties
4            //do_*() & convert2string() method
5           endclass
```

In order to copy transaction object including the following properties:

```
1           class tx_item extends UVM_sequence_item
2            `uvm_object_utils(tx_item)
3            function new (string name = "tx_item");
4             super.new(name);
5            endfunction
6             rand bit [31:0] src;
7             rand bit [31:0] dst;
8             rand command_t cmd;
```

```
9               rand tx_playload pay_h;     // contaned object
10                  logic [31:0] result;
```

The systemverilog does not have deep copy operator, it must be written manually. There 3 three steps that do most deep methods and must be followed to do deep copy:

```
1           class tx_item extends UVM_sequence_item
2            virtual function void d0_copy(UVM_object rhs);
3            tx_item tx_rhs;
4            if(!$cast(tx_rhs, rhs) )
5           `uvm_fatal(get_type_name(), "Illegal rhs argument")
6            super.do_copy(rhs);
7             src    = tx_rhs.src;
8             dst    = tx_rhs.dst;
9             cmd    = tx_rhs.cmd;
10            result = tx_rhs.result;
```

create `do_copy()` method

- called to copy objects for scoreboards,TLM,etc
- void `do_copy()` has one argument "rhs" : right hand side, because of the rules of OOP consist that the type of this argument must much the type of the base class where was declared as UVM object handle. The problem that `tx_item` properties are not visible with UVM object handle. So :

1. the first step to do is to create new handle `tx_item tx_rhs` and cast the argument to this type. Always check the results from $ cast, never use it as a task.

```
1               if(!$cast(tx_rhs, rhs) )
2              `uvm_fatal(get_type_name(), "Illegal rhs argument")
```

2. The second step involves the basic class, there maybe properties in sequence UVM that they need to be copied so call **super.do_copy()** passing rhs handle .

3. The third step is to copy the object properties.

```
1               this.src    = tx_rhs.src;
2               this.dst    = tx_rhs.dst;
3               this.cmd    = tx_rhs.cmd;
4               this.result = tx_rhs.result;
```

So now we get from where rhs comes from: is in the right hand side of the assignment. We are copying from another object to **this** one. So no need for these handles src,dst and cmd should be visible in this method.

```
1               src    = tx_rhs.src;
2               dst    = tx_rhs.dst;
3               cmd    = tx_rhs.cmd;
4               result = tx_rhs.result;
```

Until now we were looking only to copy the properties of this object which is shallow copy. The `tx_item` class has a handle to payload, we need to copy rhs payload to this one.Don't forget to check for null handles.

A class has a handle to testbench object like an agent, we would not copy it, we need only to know what handles to follow and to rewrite the do methods for `tx_payload` class.

comparing two deep UVM item objects

virtual `do_compare()` method is called by scoreboards and returns the bit that is true or false and it has a rhs handle just like `do_copy()`.

```
1  class tx_item extends UVM_sequence_item
2      virtual function bit do_compare(UVM_object rhs, UVM_compare
           compare);
3              tx_item   tx_rhs;
4              if(!$cast(tx_rhs, rhs) )
5             `uvm_fatal(get_type_name(), " Illegal rhs argument ")
6             return(super.do_compare(rhs , comparer)) &&
7              ( src    === tx_rhs.src    ) &&
8              (dst     === tx_rhs.dst    ) &&
9              (cmd     === tx_rhs.cmd    ) &&
10             (result === tx_rhs.result ));
11
12 endfunction
13 endclass
```

1. The first step is to cast the UVM handle and do `tx_item`

```
1  class tx_item extends UVM_sequence_item
2    virtual function bit do_compare(UVM_object rhs, UVM_compare
         compare);
3              tx_item   tx_rhs;
4              if(!$cast(tx_rhs, rhs) )
5             `uvm_fatal(get_type_name(), " Illegal rhs argument ")
6  endfunction
7  endclass
```

2. The second step is to call super.`do_compare()` since it is a function with returned value it needs to return saved.

```
1             return(super.do_compare(rhs , comparer))
```

3. The third step is to compare your properties using logic & which is a short circuit evaluation. we should use 4 states operator " === " . Deep compare the properties payload being careful to avoid null handles, your transactions classes may treat handles differently

4.2.5. The convert2string method

Printing a transaction in the way that you want

In the class `tx_item`, writing the virtual function that returns the string with content of this object must be done in following steps :

1. Step 1: create the string with the base object properties

```
1   class tx_item extends UVM_sequence_item
2      virtual function string convert2string();
3         string s =super.convert2string();
```

```
4  endfunction
5  endclass
```

2. Step 2: have more values with $ format,

```
1   class tx_item extends UVM_sequence_item
2      virtual function string convert2string();
3         string s =super.convert2string();
4
5         $sformat (s, "%s\n tx_item valeus are:", s)
6
7  endfunction
8  endclass
```

convert the payload have to check for null handles, we have to use a simple convert2string for the `tx_payload` class.

```
1   (pay_h == null) ? "null": pay_h.convert2string();
```

the complete code for convert2string():

```
1   class tx_item extends UVM_sequence_item
2      virtual function string convert2string();
3         string s =super.convert2string();
4          $sformat (s, "%s\n tx_item valeus are:", s);
5          (pay_h == null) ? "null" : pay_h.convert2string();
6          return s;
7  endfunction
8  endclass
```

Printing the do() method and convert2string()

- create and call **convert2string()** in he message macro

```
1   `uvm_info ("DBG", tx.convert2string(), "UVM_DEBUG")
2   `uvm_error ("DRV", $formatf("BAD :", tx.convert2string()))
3   `uvm_fatal ("cast",{"$cast failed ", tx.convert2string()})
```

- avoid **sprint()** and **print()**) as both ignore the verbosity.
  - sprint() calls `do_print` and returns the string
  - print() is non virtual method so it should call `do_print` and print with $display()
  - implement `do_print` is in your base class

```
1   virtual function void tx_item :: do_print(UVM_printer
       printer);
2   printer.m_string = convert2string;
3   endfunction
```

- Since convert2string is **virtual**,extended classes don't need to be override **`do_print()`**. In other words, calling always convert2string will print properties of the object even in the derived class. So means we only need to write it once per transaction type
- The `UVM_printer` formats values,primarily used by fields Marcos.(called by field macros).

Copy a data in a transaction in a new format

In UVM, the methods pack(),unpack(),can transform sequence items into arrays of bits,bytes and ints.

UVM testbench can record transaction by packing the object into an array and write it into a file. In another simulation, an other testbench can read the file, unpack the data and the transaction and replay them. Writing pack and unpack methods depend on your specifics of your protocol. Just make empty versions for now ,we may need them later.

```
virtual function void do_pack(UVM_packer packer);
return;
endfunction
virtual function void do_unpack(UVM_packer packer);
return;
endfunction
virtual function void do_record(UVM_recorder recorder);
return;
endfunction
```

writing `do_*()` methods when extending a transaction class

```
class tx_dst_fixed extends tx_item
`uvm_object_utils(tx_dst_fixed)

 function new (string name = "tx_dst_fixed");
 super.new(name);
 endfunction

 bit [31:0] fixed_dst;
 constraint   c_dst_fixed { soft dst == fixed_dst};

 virtual function void do_copy(UVM_object rhs);
 tx_dst_fixed  tx_rhs;
 if(!$cast(tx_rhs, rhs) )   `uvm_fatal(get_type_name(), " Illegal
     rhs argument ")
 // copy base properties
 super.do_copy(rhs);
 // copy derived properties
 fixed_dst = tx_rhs.fixed_dst ;
 endfunction

 virtual function bit do_compare(UVM_object rhs, UVM_comparer
     comparer);
 tx_dst_fixed  tx_rhs;
 if(!$cast(tx_rhs, rhs) )    `uvm_fatal(get_type_name(), " Illegal
      rhs argument ")
 // compare base properties
 return(super.do_compare(rhs, comparer) &&
 // compare derived properties
 (fixed_dst === tx_rhs.fixed_dst));
 endfunction

 virtual function string convert2string();
```

```
30        string s = super.convert2string();
31        // show base properties
32        $sformat (s, "%s\n tx_ds_fixed valeus are: \n", s);
33        // show derived properties
34        $sformat (s, "%s  fixed_dst = %0x\n", s ,fixed_dst);
35         endfunction
36  endclass
```

`do_pack()`, `do_unpack()`, `do_record()` and `do_print()` are all one line methods implemented in the base transaction class `tx_item`, we don't need to override them in the extended class for this simple protocol.

```
1  tx_item :: do_print(),d0_pack(), do_unpack() and do_record()
```

Write the sequence item virtual methods

- **Always** implement all 6 sequence item virtual methods.
  - `do_copy()`, `do_compare()`, `do_print()`, `do_pack()`,`do_unpack()`,`do_record()` plus convert2string().
  - Even if the project does not use all these methods,future projects might.
  - Exception, IP that already has field macros-stay with macros when extending those classes.
- **Always** call `non_virtual` methods, like compare()in the testbench, not `do_compare()`.
  - That sequence item method calls its `do_*()` counterpart.
  - Never mix field macros and `do_*()` method; both are called, with bad results.
- **Always** call super.`do_*()` in a `do_*()` method: allows a sequence item to be extended from another sequence item and properly chain any base class functionality. In other words, we need just to add a new thin layer and reuse all the current code by extending classes that are built on the existed functionality of the parents.

Stimulating the design with a sequence item

A sequence item represents a single transaction object. It contains the values to communicate with UVM components such as drivers and scoreboards and it provides standard methods to print, copy, compare etc.
As a question might you ask yourself: how do I use these sequence item as stimulus for my DUT? the answer is hat a sequence contains one or more items that are generated together. So, we can create complex sequences that can be generated in groups, with feedback of Streams of related transactions, processor instructions, commands and responses. This can be done only by creating multiple transactions with context between them.

Generating transactions

- A UVM sequence class is derived from `UVM_sequence base class`
- It contains a body () task that generates one or more sequence items.

```
1     class tx_sequence extends UVM_sequence #(tx_item);
2     `uvm_object_utils(tx_sequence)
3
```

```
4       function new (string name = "tx_sequence");
5       super.new(name);
6       endfunction
7
8       virtual task body ();
9       repeat (50) begin
10      tx_item  tx;
11      tx = tx_item :: type_id :: create("tx");
12      start_item(tx);
13      if(! tx.randomize())  `uvm_fatal(get_type_name(), " Illegal rhs
            argument ")
14      finish_item(tx);
15      end
16    endtask
17  endclass
```

The class `tx_sequence` is extended from the class `UVM_sequence` and it is parametrized class and it specialized for specific sequence item type

```
1  #(tx_item);
```

A sequence is not just an array of sequence items, the transactions that are generated in the body methods know this is a task not a function so it can have delays. There are 4 four steps to generate a transaction:

1. create the individual transaction object (when see **`type_id`** think of the UVM factory; The macro `uvm_object_utils`), `tx_item` class declare it class `type_id` you can think that this is a small factory with a method to create **tx__item** object. The string "tx"is the instance of the object. In general use Handle name such tx for the instance name to simplify debugging.

```
1       tx = tx_item :: type_id :: create("tx");
```

2. Wait for `start_item` (tx) to be requested from the driver for next item.The driver might no be ready or need to reset the DUT or still be busy with the previous transaction.

```
1        start_item(tx);
```

3. Assign the transaction values, randomizing values of transaction object.

4. Send the transaction to the driver by passing tx handle and `finish_item`(tx) blocks waiting for the driver to complete. The driver sends this transaction to DUT and when finishes, releases the call `finish_item`

NB: when always randomize values object, always check for the returned status in case of randomization fails. Check with if statement not with assert statement; The problem with the assert is that maybe someone wants to speedup the simulation so he disables the assertions in which case the code inside the assert is never executed, in this case thee UVM object transactions won't be randomized.

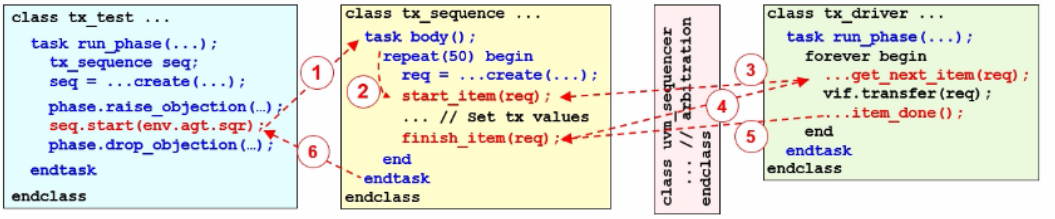


Figure 4.6.: handshake between Test/Driver/Sequence [1]

The handshake between Test/Driver/Sequence

1. The test calls the sequence's **start()** method, which initializes its properties, then invokes **body()**.
   The test **blocks** (waits at that point) in the **start()** method until **body()** exits.

2. The sequence **body()** calls `start_item()`
   `start_item()` blocks until the driver asks for a transaction

3. The driver calls `seq_item_port.get_next_item` method to pull in a transaction.
   The driver then blocks until a transaction is received from the sequence

4. The sequence fills in the item, calls **`finish_item`** to send transaction to the driver, and blocks

5. the driver calls **`item_done`** to tell the sequence it is done with that object.

6. when the sequence **body()** exits, control is returned back to the test.

7. the test continues with its next statement(such as allowing the `run_phase` to end)

4.3. Drivers and sequencers

As we have seen in previous chapters, a transaction flows from test level to the sequencer through the driver passing by the interface to the DUT. So in this section, it is highlighted how a transaction passes from sequencer to the driver and from driver to interface and from interface to the DUT.

4.3.1. UVM TLM communication

In the testbench, the DUT limits how fast the stimuli can be applied. Since the driver is connected to the DUT, it can only accept new transaction when the DUT is ready, as a result, in the connection in the left, the driver controls flow not the generator. The driver

Figure 4.7.: generator-driver-DUT

pulls transaction from generator. This last one is made up of classes in this example shown in the figure. In case if you are writing the classes from scratch without UVM you might have the following pseudo code.

```
1  class driver;
2      generator g;
3      task run();
4          forever begin ()
5              g.get(tx);
6              transfer(tx);
7          end
8      endtask
9  endclass
```

The driver class has run method that continually pulls the transactions from the generator by calling **get()**, then the driver send out those transactions by calling **transfer(tx)**.

```
1  class generator;
2      task get(tx_item tx);
3          tx = new ();
4          if(!tx.randomize()).... ;
5      endtask
6  endclass
```

In the example of the generator class, which combines `UVM_sequence` that creates random transactions and *UVM_sequencer* that write them to the driver. The driver calls **get()** task in the generator which creates the transaction, randomizes it and returns the handle to the driver.

```
1  class agent;
2      generator g = new();
3      driver    d = new(g);
4      task run();
5          d.run();
6      endtask
7  endclass
```

The agent instantiates the components and runs the driver. This is just a simplified version of real `UVM_agent`. Transactions flow from left to right and controls flow from right to left. In UVM, it is said that the driver is the initiator of the transfer, and generator is the target

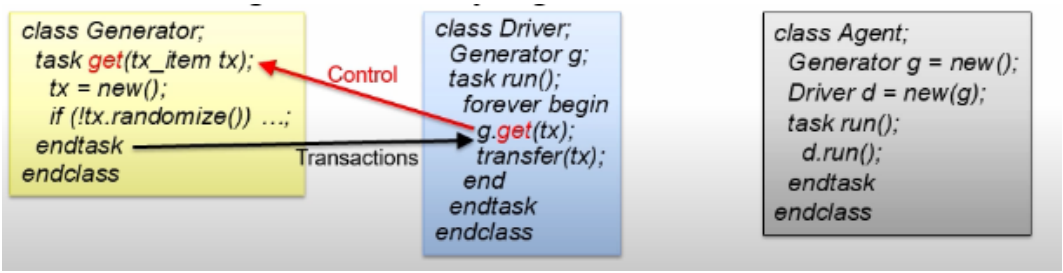


Figure 4.8.: Transaction and control flow [1]

for transfer. The problem with this approach is that the connection between these components has `hard_wired(coded)` and difficult to change. The driver can only get transactions from

single component which must be a class called "generator". As a solution to this problem is to add a layer of abstraction between the components.
The communication between the components called TLM: transaction level Modeling, it is based from TLM standard from systemC. As it is explained in the previous chapters, the TLM connection has two ports A **port** calls a communication method, such as calling **get()**

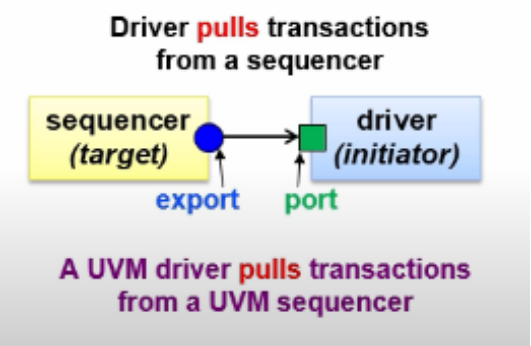


Figure 4.9.: Target and initiator communication [1]

NB: the term **port** is confusing, it has nothing to do with port in systemverilog term port. The driver requests the transaction to the port, the target, such as `UVM_sequencer` class contains an object called **export** which has the implementation of the port's method, such as the **get()** body.
A port must be paired with exactly one export (`one_to_one`) connection.

```
1  DRIVE ==>SEQUENCER
```

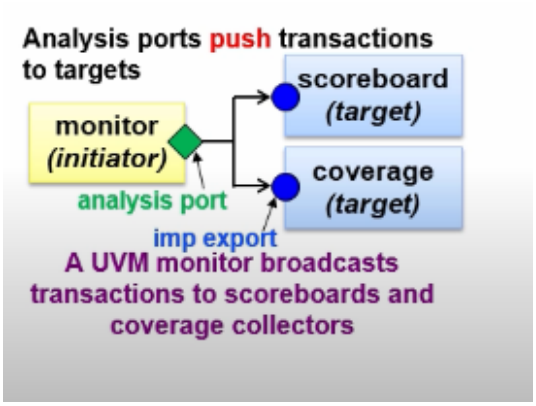


Figure 4.10.: Analysis port-export [1]

An analysis port as shown in the figure 4.10 : pairs with zero or more exports (`one_to_many`)

```
1  MONITOR ==> SCOREBOARD & COVERAGE
```

Without TLM the driver who controls the generator like is shown in the figure 4.11 while in the other figure 4.12 it is with TLM, the driver has a handle `seq_item_port` and calls get(), this is a blocking connection called `UVM_blocking_port` as the driver's run task is blocked until the get() method returns.
TLM connections are components so the export is the child of the sequencer. The sequencer has the final get() method.

Figure 4.11.: Control direction without using TLM [1]

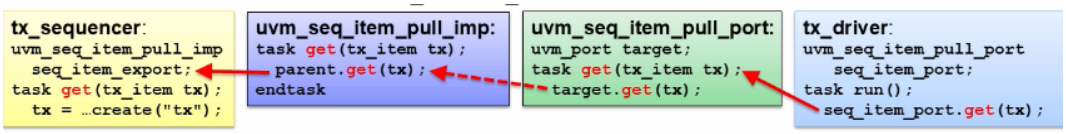


Figure 4.12.: Control direction using TLM [1]

So to go from **port** to **export**, the `tx_agent` calls the **port.connect** with the handle to export.

```
1 tx_agent:
2    drv.seq_item_port.connect(sqr.seq_item_export);
```

The general formula is:

```
1    initiator.port.connect(target.export)
```

Even though there are three 3 calls of "get()" instead of one, the TLM connection still efficient as its functional call is very fast because it passes only handles not entire objects.
There are many TLM connections flavors like blocking, non blocking, FIFOs and others...
The most common TLM connections are the blocking and analysis port.

4.3.2. Working way of the sequencer and the driver together

The sequencer is a part of the agent, it receives transaction by sequence and then send them to the driver in this agent (the driver is fed by a single sequence and a sequencer is fed by

Figure 4.13.: The sequencer and driver [1]

multiple sequences and arbitrates between them). See figure 4.13. In the following code, it is a way describing how to declare a sequencer handle **sqr** specialized with `tx_item` in the agent.

```
1     // Declare a sequencer handle - fas !
2     class tx_agent extend UVM_agent;
3     uvm_sequencer #(tx_item)  sqr;
4     tx_driver                 drv;
5     endclass
```

The other way is shown in the following code which is defining a new type **tx_sequencer** then using this type to declare the handle and is helpful in case it is needed to define multiple handles of sequencer in multiple places.

```
1              // Declare a separate sequencer type
2          typedef uvm-sequencer #(tx_item)  tx_sequencer;
3          class tx_agent extend UVM_agent;
4          tx_sequencer sqr;
5          tx_driver    drv;
6          .....
7          endclass
```

However the preferred style is the first style of coding as it is short and does not require to define other type.
Both components communicate with a special TLM connection, the base class is `UVM_driver` which is parametrized with : the first parameter is " REQ " for request type and " RSP " for response type which has a s default value RSQ type and the declaration of special TLM connection is `seq_item_port`, which is declared in the base class. see the following example:

```
1      class tx_driver extend UVM_driver #(tx_item);
2      `uvm_component_utils(tx_driver)
3      //constructor, no default needed
4       function new (string name, uvm_component parent);
5       super.new(name, parent);
6       endfunction
7          .....
8          endclass


1  ////////////////////////////////////
2      class tx_driver #(type REQ=uvm_sequence_item, type RSP=REQ)
          extends uvm_component;
3      uvm_seq_item_pull_port #(REQ, RSP) seq_item_port;
4      endclass
```

`tx_driver` is extended from `UVM_driver` and specialized with **tx__item** type(since `tx_-driver` is a component we need to register it in the UVM factory); the constructor has two formal arguments: the instance and the handle to the parent, we don't have default values unlike `sequence_item`.
In the figure 4.14 where the driver requests a sequencer item from a sequencer and send the

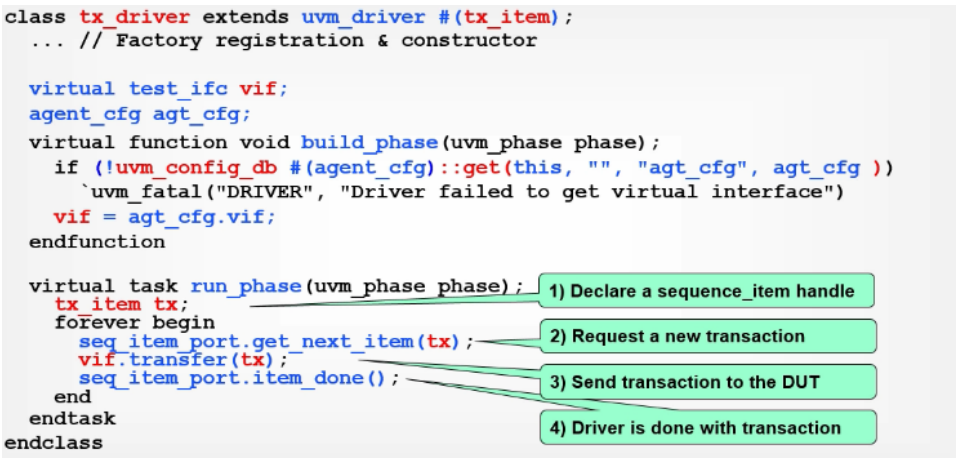


Figure 4.14.: The driver example -complete the code [1]

item to DUT inputs. The driver needs a handle to the interface " vif " so it can send the transactions. In the run phase, the driver:

- declares a handle to the transaction
- requests a new transaction by calling `get_next_item()`
- calls the method in the interface such as **transfer(tx)** to send transaction to the DUT. When a task returns, the driver tells the sequencer is done with transaction by calling `item_done`.

The UVM agent creates and connects a driver and a sequencer. `tx_agent` class has handle for driver and sequencer, during the build phase, components create lower layer components. The `UVM_driver` base class has a `built_in` TLM port called `seq_item_port` and `UVM_sequencer` base has a `built_in` TLM export called `seq_item_export`. In build phase the components are built **TOP DOWN**

```
1     class tx_agent extend UVM_agent;
2         ...// Factory registration & constructor
3       tx_driver                drv;
4       uvm_sequencer #(tx_item) sqr;
5
6       virtual function void build_phase(uvm_phase phase);
7       drv = tx_driver :: type_id :: create("drv",this);
8       sqr = new("sqr",this); //don't call factory
9       enfunction
10
11      virtual function void connect_phase(uvm_phase phase);
12      drv.seq_item_port.connect(sqr.seq_item_export);
13      endfunction
14    endclass
```

4.3.3. The communication between modules and the interface

In verilog,ports connecting the testbench and the modules is too low level, and gives error prone plus adding more port is time consuming. Systemverilog SV introduces the interface which contains all signal and code to describe the communication protocol.

```
1     interface test_if (input  logic clk);
2     logic       reset_n;
3     logic       en;
4     logic  [31:0] a, b;
5     logic  [31:0] result;
6
7     task automatic transfer(tx_in t);
8     ...
9     endtask
10    endinterface
```

An interface is like a module that contains signals and the code that read and write signals, it receives handle to an object that describes the transaction to be sent to DUT. The testbench classes become more reusable with this layer of abstraction, the driver can call a method and sends the transaction so it reduces the dependencies between the driver and DUT.See the

corresponding code of the interface.

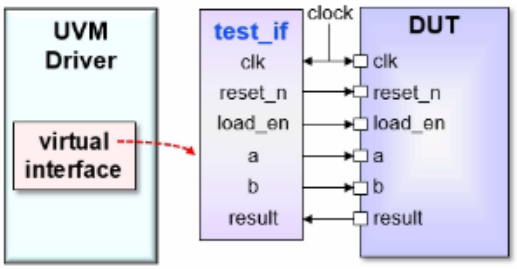


Figure 4.15.: virtual interface

Sometimes in SV classes, `UVM_driver` may contain virtual interfaces. In general, in SV " virtual " means reference to something else.
A dynamic class cannot contain a static interface which is made of wires, the virtual interface value is passed through in a configuration database.

```
1     class tx_driver ....;
2        virtual test_ifc vif;
3        virtual task run_phase(...);
4        tx_item  tx;
5        forever begin
6        seq_item_port.get_next_item(tx);
7        vif.transfer(tx);
8        seq_item_port.item_done();
9        end
10      endtask
11    endclass
```

4.3.4. The Emulation performance

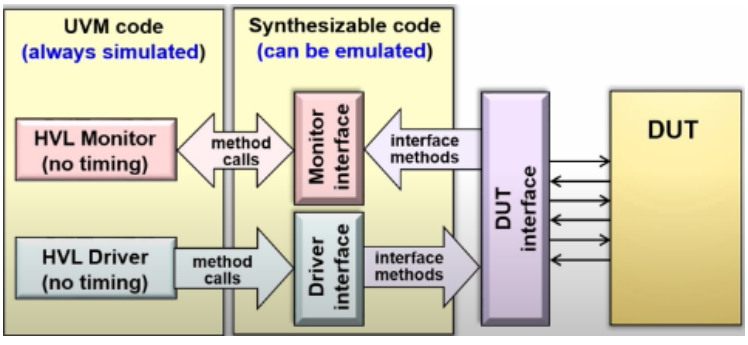


Figure 4.16.: The emulation [1]

Emulation performance is always a concern. The code of verification is written in UVM OOP and simulation run in a simulator such **Questasim**. In design time it is not declared any delay, it passes transactions and interfaces methods and it is not assigned directly to DUT ports. The emulation side is written in synthesizable modules or interfaces. It is important to synchronize the DUT when drives the pins.[10]

4.4. Monitor

The interface reads DUT pin wiggles and collects them into UVM transactions, then the monitor receives them and broadcasts them to be analyzed. If the scoreboard and `coverage_collector` are present(they are optional), they receive those transactions and work on them. The `coverage_collector` can go inside or outside of the agent. The monitor never analyzes

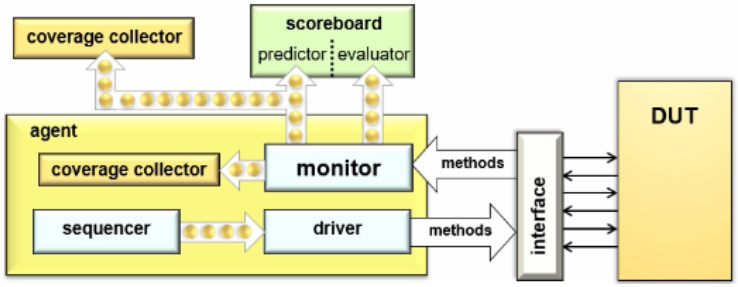


Figure 4.17.: The coverage-collector,agent,monitor [1]

these collected transactions, it just drops them to the proper destinations. A monitor is always passive component because it does not drive DUT ports.

4.4.1. The Monitor and type of the transactions

The monitor observes the values going into the DUT and creates input transactions object. This broadcasts the analysis port(they maybe seen by the scoreboard predictor or/and coverage collector). The monitor sees also the transactions coming out of the DUT and creates outputs transactions object(that broadcasts the scoreboard evaluator and coverage collector).

4.4.2. Gathering the input transactions for analysis

The driver and sequencer cannot send these transactions, because the values sent by them may not be the actual values on the interface due to deliberated errors or accidental.
An agent cannot send these transactions because it might be a passive agent which means that it does not contain neither the sequence nor the driver(no streams).
Only the monitor captures the values and sends them to the scoreboard and coverage collector because it :

- ensures that both UVM verification IP and `non_UVM` work in the UVM-testbench.
- ensures vertical scaling of testbenches.

The monitor broadcasts transactions with TLM analysis ports(`one_to_many`). The number of analysis port designed is dependent. Typically, the monitor broadcasts the input transactions in one port and the output transactions in an other.

4.4.3. The communication between the monitor and scoreboard

The monitor is tied to the DUT until it limits how fast the transactions are received. The monitor sends out the transactions to the scoreboard. Writing in SV the code of these component without UVM, see figure 4.19.

Figure 4.18.: DUT-monitor-scoreboard

Figure 4.19.: code of monitor-scoreboard [1]

The monitor will have a forever loop receiving a transaction from the DUT and then passing a handle to the write() method in the scoreboard. The scoreboard's write() method receives a handle and might store in an array the expected transactions. These two components are instantiated and connected in agent class.
As may you notice, this connection has problems such that, the monitor communicates only with the component called scoreboard(not with the `coverage_collector`) (so its connection is hard-coded class name and with a fix topology). So if we add function " coverage ", we need to add more code to the monitor after write() call. The scoreboard and coverage are optional, so we need to write this code using if statement. Hence, the monitor needs to know about the configuration information. We can use mailbox in SV but may not work well with multiple optional receivers. So, as a solution, add a layer of abstraction between the components. Both control and transactions flow from left to right. The monitor is the initiator of the transfer, scoreboard is the target for the transfer.

4.4.4. TLM analysis port flow

Figure 4.20.: control and transaction flow direction [1]

In the figure4.20 shows the connection without UVM, where the monitor passes the handle directly to class scoreboard's write() function.
In UVM, as shown in the figure 4.21, the `tx_monitor` reads the transaction from the DUT and passes the handle to `TLM_analysis_port` write() function. The scoreboard write() function is called from `analysis_imp_export`. In analysis port can connect multiple component so, it needs a list of component handles called " imp ", then for each component, it calls its write() method. The final connection is made in higher hierarchical level such as the **the environment** or **the agent**.

```
1          initiator.port.connect(target.export)
```

TLM rule: write() is `non_blocking` and always completes. This is because the monitor must passes the transactions to the other components without any delay and this is because of if the scoreboard has a delay, the monitor can miss the following transaction.

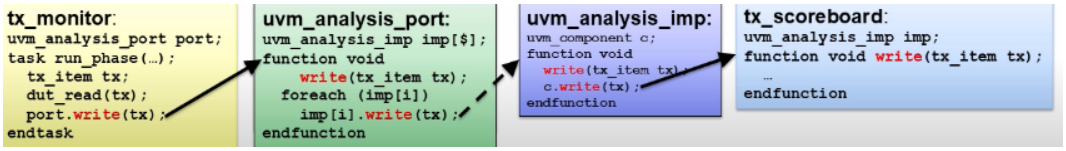


Figure 4.21.: Monitor-scoreboard with UVM [1]

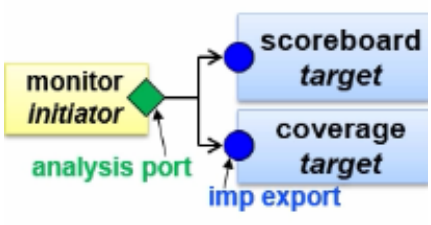


Figure 4.22.: TLM analysis port-export [1]

4.4.5. Monitor code example

```
1  class tx_monitor extends uvm_monitor;
2      ...// Factory registration & constructor
3          virtual tb_if vif;
4          agent_config agt_cfg;
5          uvm_analysis_port #(tx_item) dut_in_tx_port;
6          uvm_analysis_port #(tx_item) dut_out_tx_port;
7
8          function void build_phase(uvm_phase phase);
9          dut_in_tx_port = new("dut_in_tx_port", this);
10         dut_out_tx_port = new("dut_out_tx_port", this);
11         if(!uvm_config_db #(agent_config) :: get(this,"","agt_cfg",
               agt_cfg))
12         'uvm_fatal("MONITOR","No agent configuration found!")
13         vif = agt_cfg.vif;
14         endfumction
15
16         virtual task run_phase(uvm_phase phase);
17         fork
18         get_inputs();
19         get_outputs;
20         join
21         endtask
22
23         virtual task get_inputs();
24         tx_item tx_in;
25         forever begin
26         tx_in =  tx_item :: type_id ::  create("tx_in");
27         vif.get_an_input(tx_in);
28         'uvm_info("TX_IN",tx_in.convert2string(), UVM_DEBUG)
29         dut_in_tx_port.write(tx_in);
30         end
31         endtask
32
```

```
33          virtual task get_outputs();
34          tx_item tx_out;
35          forever begin
36          tx_out =  tx_item :: type_id ::  create("tx_out");
37          vif.get_an_output(tx_out);
38          'uvm_info("TX_out",tx_out.convert2string(), UVM_DEBUG)
39          dut_out_tx_port.write(tx_out);
40          end
41          endtask
42
43      endclass: tx_monitor
```

4.5. Agent

We want to verify a device with multiple ports using UVM code and make it reusable on future projects, we start with making UVM-sequencer,monitor,driver and call the methods in the interface and check the result for the scoreboard, lastly, we instantiate all these component in a test, but this still is not reusable. When we write second test we will have to manually instantiate all these low level components in a new test all over again.
In case if this device has multiple interfaces such as USB ports it should be easy to write the test component because it contains all these low level components.
So better way to organize this is by wrapping all these component in an agent which is a reusable container that creates and connects the component inside.Hence, the test controls all low level components without reaching down to them. So, make a configuration class

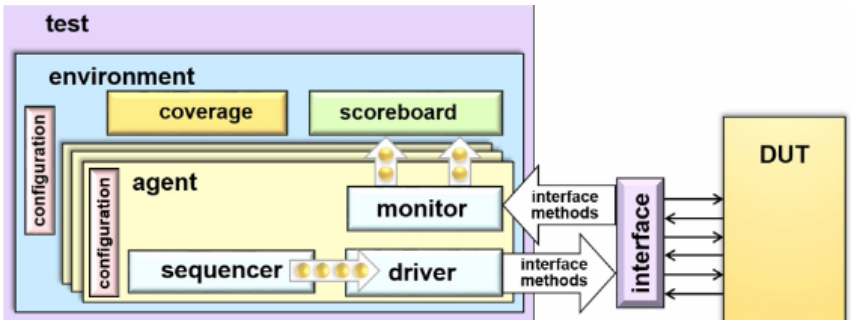


Figure 4.23.: multiple agents

containing addresses and other properties. This agent let us easily reuse port (USB for example) specific component. If the design has 3 USB ports, just instantiate 3 agents. The scoreboard receives the results from all agents, so it is not included inside any agent. By wrapping agent(s) and all theses components in the environment level, it becomes reusable either. So it has its own configuration, test can customize its behavior.

An agent contains the components for a specific DUT protocol. Each agent is connected to an interface for that protocol (for the DUT side, most of these signals may be discrete ports). Related DUT signals are grouped together in an interface that has its own UVM agent. see figure 4.24.An environment can contain any number of agent.

A passive agent only monitors the DUT(the driver and sequencer are not created) while an active agent both drives and monitors the DUT. UVM methodology recommends that agents should be configurable to be either active or passive . The figure shows an example of

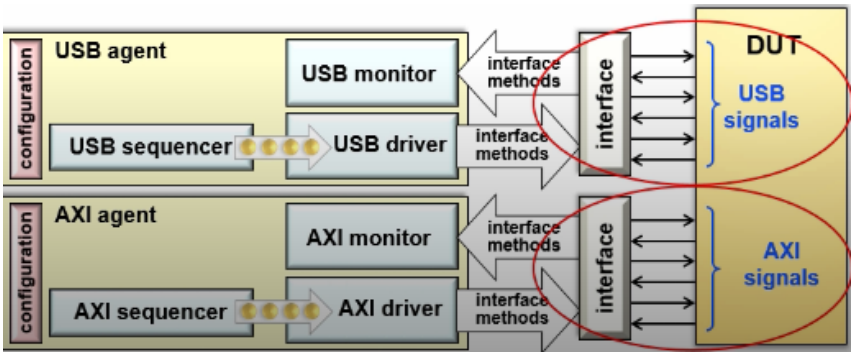


Figure 4.24.: DUT with multiple ports

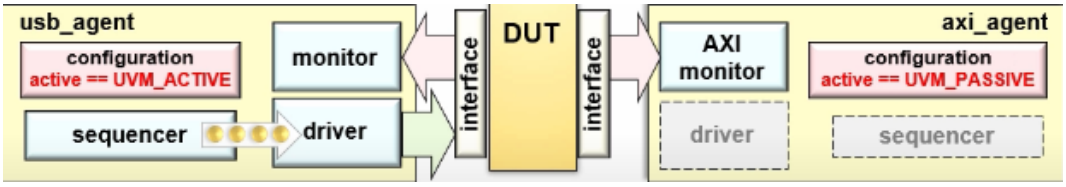


Figure 4.25.: Active and passive agent

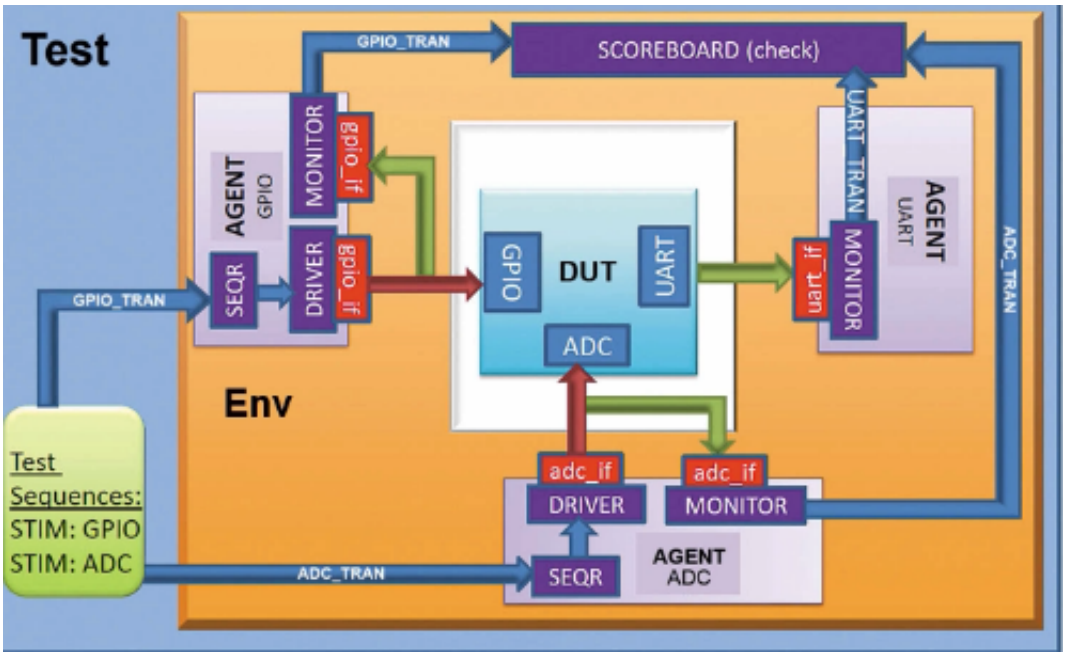


Figure 4.26.: Real example of multiple agent with other components

multiple agents that has :

- A separate interface is used for each logic group of DUT signals
- A separate agent is paired with each DUT interface
- Agents can be active (drive and monitor) or passive (just monitor)

the source of this figure 4.26 is taken from "A verification Horizons Magazine an evaluation of the advantages of moving fomA vhdl to a UVM test bench.(verificationacademy.com)"

4.6. Scoreboard

Any discussion of scoreboards should have a brief pr-description of the design. A scoreboard is comparing the actual outputs of the design with expected values. The design engineer

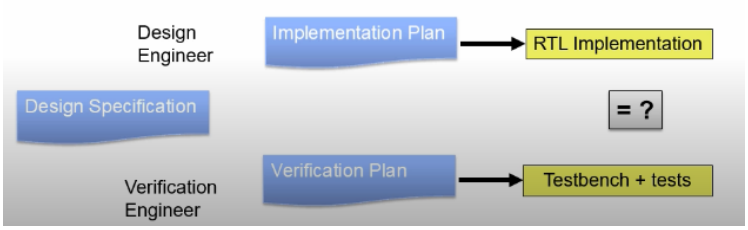


Figure 4.27.: Implementation and verification plan

reads the specifications and implement the design and writes RTL code. The verification engineer reads same specifications and makes verification plan and writes the test for the design. The comparison between these two threads can be a scoreboard. In UVM, it is preferable to apply random stimulus that requires automatic checking. For a single test, the scoreboard job is to let you know if the test is pass or fail, in some other scoreboards,they can let you know when finishes.

In initial testbenches verify single blocks. As the design progresses from individual blocks

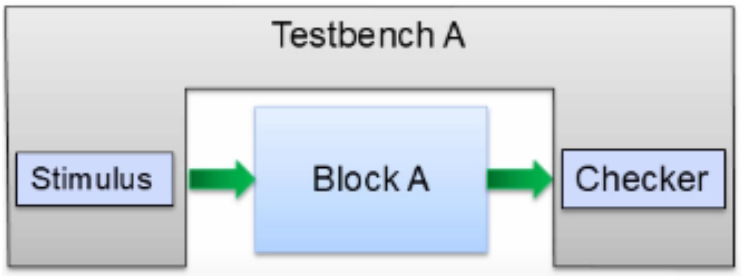


Figure 4.28.: Single block

to `sub_systems`, chips and beyond, checking evolves too. Each configuration may need a different set of scoreboards. The environment needs to be flexible enough to handle these various configurations.

4.6.1. Individual parts of the scoreboard

In general, the scoreboard should keep track of the differences between outputs of the design and the testbench. A monitor captures the DUT inputs and outputs. The design outputs

called the actual results. A predictor transforms the input transactions , perhaps with an abstraction model. The predictor outputs called expected results and they will be stored in a buffer until the actual results are ready. The evaluator compares the expected and actual results, this could be with the following method:

```
1     uvm_sequence_item :: compare()
```

or with another algorithm. The scoreboard is a reusable component. Common scoreboard

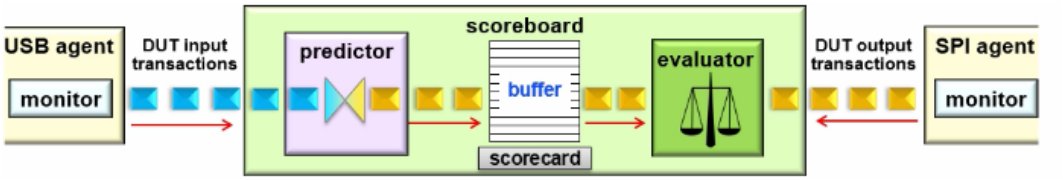


Figure 4.29.: The component of the scoreboard [1]

bugs might be caused by improper synchronization of all these threads.

4.6.2. Scoreboard TLM communication

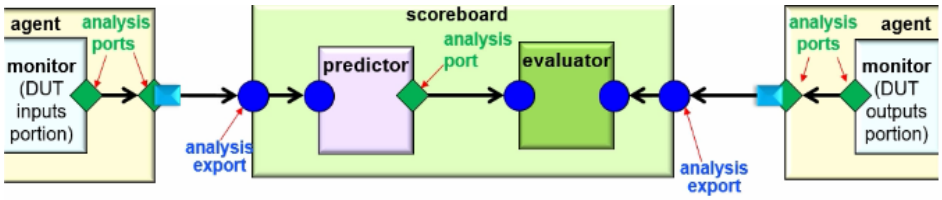


Figure 4.30.: Scoreboard TLM connection [1]

A scoreboard needs to communicate transactions in several ways:
In high level, the monitor sends transactions to the scoreboard. Inside the scoreboard, the predictor is optional,hence, it must be connected to the monitor with analysis port. The monitor calls a write() method in the TLM method which then calls the write() method in the agent port. This broadcasts the transaction up through agent ports and into any receiving exports. The scoreboard receives the handle with analysis export(because, it exports its write() method making visible to component outside the block) the final connection is made with `analysis_imp_export` which sends the handle to the final implementation of the write() method. Scoreboard is instantiated in the environment level. In this document, will be shown to kinds of scoreboards:
hierarchical scoreboard The predictor and the evaluator are different component contained

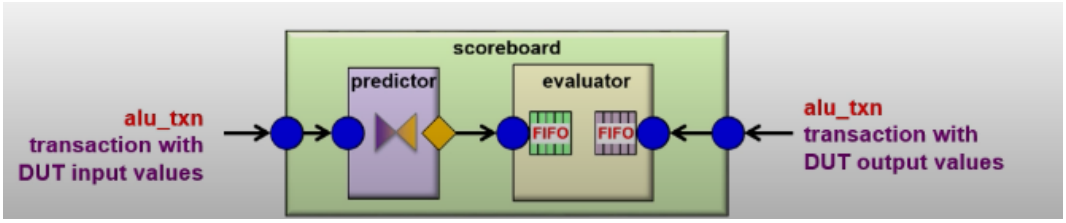


Figure 4.31.: hierarchical scoreboard [1]

inside the scoreboard, the DUT is an ALU, the sequence item type of the input with an

opcode and operands is used for DUT inputs and outputs, the transactions are held on FIFOs and then compared. The predictor is a reference model.

Flat scoreboard

The second example is a flat scoreboard: predictor and evaluator coded directly in the

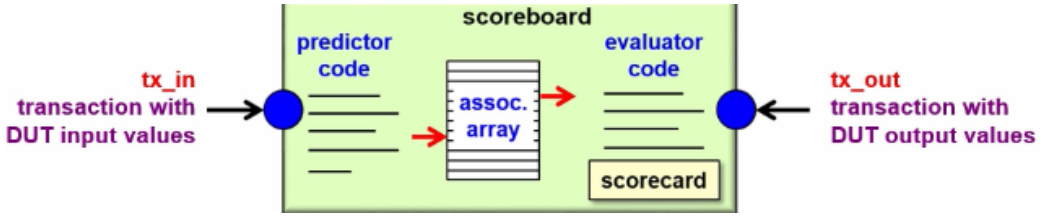


Figure 4.32.: Flat scoreboard [1]

scoreboard and it has different sequence item types for inputs and outputs, the expected values are stored in an associative array storage.

4.6.3. Scoreboard storage

Typically,the predictor genrates the expected transactions in 0 Zero time. The scoreboard needs to store them until the actual results available later. There are lot of ways to store the transactions, among them :

- **FIFOs** : the predictor and evaluator can be connected by means of `TLM_analysis` port, this function is like SV mailbox with blocking get() and put() methods, the content of FIFO is not visible and it is automatically connected to analysis export so there is no need to connect between the write() and put() methods. `First_in`,`First_out` ordering.
- **Queues** : it is a sv dynamic array `built_in` access methods it is more flexible than FIFO.
- **Associative arrays** : when DUT outputs can occur in an unpredictable order. The storage created only for the locations used.

4.6.4. Safer testbenches

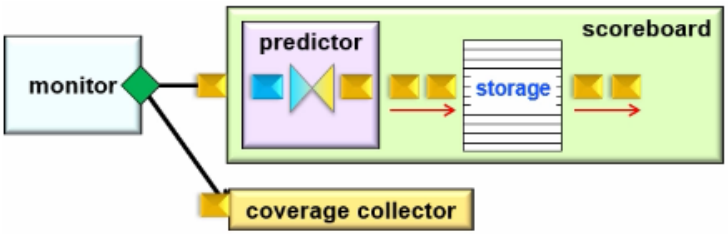


Figure 4.33.: Testbench-copy-clone [1]

Among the common software project problems is memory corruption, the monitor stores the DUT transactions in an object then sends the handle to the predictor and coverage collector. A predictor can write the expected result into the transaction. The coverage collector will immediately see the changes and if the collector sample the changed values,

it could get wrong coverage result. The best way to avoid this, by making sure that the testbench using copy()/clone() before writing into it. This principle is known as copy On Write COW.
Let's consider the following code, we have write() method that receives `tx_item` handls and dst handle.

```
1       function void write(input tx_item t);
2       tx_item dst;
3       dst = t.clone(); //??
4       ...
```

We cannot clone t to dst, we get the following compiler error :

```
1      Error Questa: illegal assignment to type 'class tx_item' from type
           'class uvm_object': types are not assignment compatible.
```

this is because `tx_item` eventually derived from `uvm_object` which is, where the clone() method is defined(turned to uvm base handle) the assignment of a base handle to a derived handle is not legal, instead, we must use $cast to check at run time the type of the object returned by clone() methd and ensured the compatibility with dst handle. It would be better if we use if statement to check the results and gives fatal in case of errors. As it is mentioned before in the monitor section, always created a new object for every transaction broadcast. Otherwise, the scoreboard storage array will have many handles outputting to the single object.Sharing and reusing objects can result in data corruption bugs.

4.7. Environment

The agent contains the component for a single protocol and it is a reusable class, the environment contains multiple agent plus scoreboards and coverage collectors and configuration object. This forms a high level reusable block. An environment can contain multiple lower environment, the best practice is to have a single top environment and multiple sub environment, this allows you to hide details of lower level blocks as an agent hides details of the driver and monitor.

```
1      class tx_env extends uvm_env;
2       ...// factory registration & constructor
3       tx_agent                agt;
4       tx_scoreboard           scb;
5       tx_coverage_collector   cov;
6       env_config            env_cfg;
7
8       virtual function void build_phase(uvm_phase phase);
9       // get env_cfg from configuration database
10      agt = tx_agent:: type_id:: create("agt", this);
11      if (env_cfg.enable_scoreboard)
12           scb = tx_scoreboard:: type_id:: create("scb", this);
13           if (env_cfg.enable_coverage)
14           cov = tx_coverage_collector:: type_id:: create("cov", this);
15     endfunction
16
17     virtual function void connect_phase(uvm_phase phase);
18     if (env_cfg.enable_scoreboard) begin
```

```
19      agt.dut_in_tx_port.connect(scb.dut_in_imp_export);
20      agt.dut_out_tx_port.connect(scb.dut_out_imp_export);
21      end
22      if (env_cfg.enable_coverage)
23      agt.dut_in_tx_port.connect(cov.dut_in_imp_export);
24      endfunction
25      endclass: tx_env
```

`tx_env` class is extended from `uvm_env`, it has handles to the components agent, scoreboard, coverage and configuration class which contains variables that describe the environment. The build phase is where you build the component, the first step to do is to get the configuration of the environment then create the agent, since the scoreboard is optional, it depends on the configuration object if it must be crated or no,likewise the coverage collector is optional. The last step is to connect the components. The environment is just a container of the components so it does not have a `Run_phase`.

4.8. configuration

UVM testbench is composed from components and stimulus ,they should be configurable for maximum re-usability, Instead of hardwired values create variables that we can vary to change the class behavior, allow the user to change the testbench topology, the following configuration values must be set during the build phase

- Number of master and slave agents
- Active o passive agents
- Interface location
- Bus sizes
- Address ranges for slave devices

The configuration values that must be set during the run phase:

- Stimulus specification:
  - Transaction generation iteration
  - Transaction delays
  - Randomization constraints
- Verification specification
  - Enabling or disabling message printing
  - Enabling or disabling specific checks in the scoreboard

Passing configuration values through OOP constructors in hierarchical references does not work(there are too many values and testbench topology can change from run to run).This is solved in UVM by passing them through a separate database DB which is not a part of testbench topology, using set() and get() methods.
The database is made up of three entries:

```
1       scope     name    value
```

UVM configuration DB is stored in a class called `UVM_config_db`. A set of values ,addressed by strings stored in an associative array, each entry consists of scope and name and value, this class has two main functions set(..) and get(..) which have four 4 arguments : the first and second " context and inst" are combined to give scope and the last two are the name of the entry and the actual value respectively.
For further information return back to `uvm_users_guide`.

4.9. UVM factory

One of the fundamental role of UVM is reusing of testbench over and over without making any changes (one reason to avoid changes is that can break the existing tasks).
If a component is constructed with new(), then it is not possible to derive from it.

```
1      class usb_agent extends uvm_agent
2       usb30_driver drv;
3       function void build_phase(...);
4       drv = new();
5       endfunction
6       endclass
```

When building an object a hook needed to optionally build an alternative one, this allows writing the testbench code once and inject new behavior later. In uvm, building the object with create()method that looks up the class in the factory and build the object.

```
1      class usb_agent extends uvm_agent
2       usb30_driver drv;
3       function void build_phase(...);
4       drv = usb30_driver :: type_id :: create("drv", this); //factory
            create driver
5       endfunction
6       endclass
```

To connect factory with classes either use the first way which used for component `'uvm_component_utils` macro registers classes derived from `uvm_component`,such as `uvm_test`, `uvm_env`, `uvm_agent`, `uvm_sequencer`, `uvm_monitor`...; a component is constructed with two arguments: name and parent. The second way is used for non-component class `'uvm_object_utils` macros register classes derived from `uvm_object` and any other non-component class like `uvm_sequence`, `uvm_sequence_item`, and configuration object class.
Theses macros create a proxy class `type_id` that can build the class by registering its class name and its proxy in the factory.

Chapter 5.

Examples in UVM

5.1. First style of coding

5.1.1. Full adder

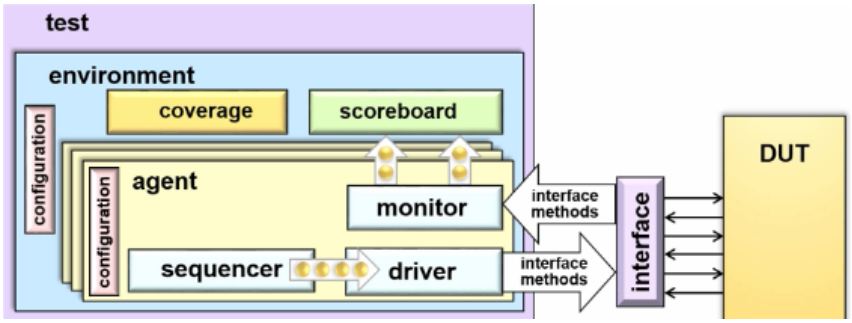


Figure 5.1.: circuit to implement

The figure 5.1 shows the layered levels of verifying the DUT. In this chapter we will verify some simple circuits using UVM and the all component explained in the previous chapters in general and the code explained in the last chapter in particular. All the examples used are implemented in VHDL except the DUT in FSM.
The following code implements the full adder in vhdl,the figure 5.2 shows the full adder circuit

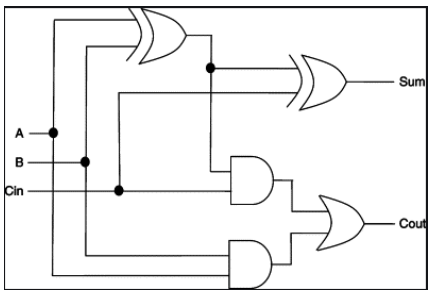


Figure 5.2.: Full-adder

where :

```
1     S = a ^ b ^ ci // sum
2     co = (a & b) | (a & ci) | (b & ci)
```

The code in VHDL:

```
1  //--------------------**fa.vhdl**-----
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  entity adder is
6          port(
7               clk    : in  std_logic;
8                   reset : in  std_logic;
9                   a     : in  std_logic;
10                  b     : in  std_logic;
11                  ci    : in  std_logic;
12                  s     : out std_logic;
13                  co    : out std_logic);
14 end adder;
15 architecture description of adder is
16 begin
17                 s <=  a xor b xor ci;
18                 co <= ( a and b) or(a and ci) or (ci and b);
19 end description;
```

The interface is spitted to two : **input_if** and **output_if**, the first one used to describe the inputs going to the DUT and second one is the interface of signals coming out from DUT. The interface using "modport" which is used to restrict the access within an interface. It is not necessary to split the interface to input and output, is just about the style coding. The interfaces of full adder

```
1  //--------------------**input_if.sv**--------
2  interface input_if(input clk, rst);
3      logic  A, B;
4      logic ci;
5      modport port(input clk, rst, A, B, ci);
6  endinterface
7  //-----------------------------------------
```

```
1  //--------------------**output_if.sv**-------
2  interface output_if(input clk, rst);
3      logic  data;
4      logic co;
5      modport port(input clk, rst, output  data, co);
6  endinterface
7  //-----------------------------------------
```

the transaction are extended from **uvm_sequence_item**.

```
1  `uvm_object_utils_begin(packet_in)
2
3  `uvm_field_int(A, UVM_ALL_ON|UVM_HEX)
```

they are just two lines macro expand to over 100 lines of code just to support the field automation macros [11]. It is useful when building your own comparer so do not need to create your own **do_compare()** and **do_copy()**... then build the class, since it is derived from object then it has no parent only name and register it in the factory.

```
//--------------------**packet_in.sv**-------
class packet_in extends uvm_sequence_item;
    rand logic A ;
    rand logic B ;
    rand logic ci;
    `uvm_object_utils_begin(packet_in)  // register in the factory
        `uvm_field_int(A, UVM_ALL_ON|UVM_HEX)
        `uvm_field_int(B, UVM_ALL_ON|UVM_HEX)
        `uvm_field_int(ci, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end
//----------------------------------------
    function new(string name = "packet_in" );
        super.new(name);
    endfunction: new
//----------------------------------------
endclass: packet_in
//----------------------------------------
```

`Packet_in` and `packet_out` are transactions sent to the DUT and received from it respectively. In this code are defined both in different classes so to make it easy for me to distinguish which packet do I need to use. Both are extended from `uvm_sequence_item` and registered in the factory then built using new() method, each of the packets has its own properties

```
//--------------------**packet_out.sv**------
class packet_out extends uvm_sequence_item;
    rand logic data;
    rand logic co;
    `uvm_object_utils_begin(packet_out)
        `uvm_field_int(data, UVM_ALL_ON|UVM_HEX)
        `uvm_field_int(co, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end
//---------------------------------------
    function new(string name="packet_out");
        super.new(name);
    endfunction: new
//---------------------------------------
endclass: packet_out
//----------------------------------------
```

Generating the sequences that sequence class is derived from `uvm_sequence` base class and it is parametrized with #(`packet_in`). It is also registred in the factory and built, it uses a task to use `start_item` which tells the sequencer that the sequence is available to be arbitrated by him then, it randomizes the returned value also to use `finish_item` which sends the randomized `sequence_item` to the driver.

```
//--------------------**sequence_in.sv**-----
class sequence_in extends uvm_sequence #(packet_in);
    `uvm_object_utils(sequence_in)
//----------------------------------------
    function new(string name="sequence_in");
        super.new(name);
    endfunction: new
//----------------------------------------
```

```
9       task body;
10          packet_in tx;
11          forever begin
12              tx = packet_in::type_id::create("tx");
13              start_item(tx);
14              assert(tx.randomize());
15              finish_item(tx);
16          end
17      endtask: body
18  //-----------------------------------------
19  endclass: sequence_in
20  //-----------------------------------------
```

The sequencer class is derived from `uvm_sequencer` parametrized with `packet_in`, it is registered in the factory using `'uvm_component_utils` not`'uvm_object_utils` so to build it, it must have name and parent. It will arbitrate the randomized transactions and then send them to the driver.

```
1   //--------------------**sequencer.sv---------
2   class sequencer extends uvm_sequencer #(packet_in);
3       `uvm_component_utils(sequencer)
4   //-----------------------------------------
5   function new (string name = "sequencer", uvm_component parent = null);
6           super.new(name, parent);
7   endfunction
8   //-----------------------------------------
9   endclass: sequencer
10  //-----------------------------------------
```

The driver class is derived from `uvm_driver` always parametrized with `packet_in`, it is registered in the factory and built with a name and parent. In this class is defined the virtual interface, because in SystemVerilog a class cannot make a reference to a signal without being declared within a module or interface scope where those signals are defined. For the purposes of developing reusable testbenches this is very restrictive. However, SystemVerilog classes can reference signals within an interface via a virtual interface handle. This allows a class to either assign or sample values of signals inside an interface, or to call tasks or functions within an interface. A virtual interface is a peculiar concept. It behaves like a class variable, but an interface gets defined and instantiated like a module. An interface is not a data type, but a virtual interface is.
In task run phase, it uses fork-join to run in parallel three tasks that are defined out side of this task. The virtual protected task used when we don't want the methods and members be accessible from outside only by the child inherited. In this task, the inputs are reset when reset=1.
In the `get_and_drive` task, it waits until reset gets activated low and when the clock is positive, It loops forever to get multiple transactions from sequencer using `seq_item_port`.get(req).
In the `drive_transfer` task, the transactions are transferred to the virtual variables. It waits for one clock cycle to generate and record then It ends record after another clock cycle that is used for hold time, the task is ended.

```
1   //--------------------**driver.sv------------
2   typedef virtual input_if input_vif;
```

```
3  class driver extends uvm_driver #(packet_in);
4      `uvm_component_utils(driver)
5      input_vif vif;
6      event begin_record, end_record;
7  //-----------------------------------------
8  function new(string name = "driver", uvm_component parent = null);
9      super.new(name, parent);
10 endfunction
11 //-----------------------------------------
12 virtual function void build_phase(uvm_phase phase);
13     super.build_phase(phase);
14     assert(uvm_config_db#(input_vif)::get(this, "", "vif", vif));
15 endfunction
16 //-----------------------------------------
17 virtual task run_phase(uvm_phase phase);
18     super.run_phase(phase);
19     fork
20         reset_signals();
21         get_and_drive(phase);
22         record_tr();
23     join
24 endtask
25 //-----------------------------------------
26 virtual protected task reset_signals();
27         wait (vif.rst == 1);
28     forever begin
29         vif.A <= 0;
30         vif.B <= 0;
31         vif.ci <= 0;
32         @(posedge vif.rst);
33     end
34 endtask
35 //-----------------------------------------
36 virtual protected task      get_and_drive(uvm_phase phase);
37     wait(vif.rst === 1);
38     @(negedge vif.rst);
39     @(posedge vif.clk);
40     forever begin
41         seq_item_port.get(req);
42         -> begin_record;
43         drive_transfer(req);
44     end
45 endtask
46 //-----------------------------------------
47 virtual protected task drive_transfer(packet_in tr);
48     @(posedge vif.clk)
49     vif.A = tr.A;
50     vif.B = tr.B;
51     vif.ci = tr.ci;
52     @(posedge vif.clk);
53     -> end_record;
```

```
54      @(posedge vif.clk); //hold time
55  endtask
56  //-----------------------------------------
57  virtual task record_tr();
58      forever begin
59          @(begin_record);
60          begin_tr(req, "driver");
61          @(end_record);
62          end_tr(req);
63      end
64  endtask
65  //-----------------------------------------
66  endclass
67  //-----------------------------------------
```

The `driver_out` is extended from `uvm_driver`, driver and `driver_out` can be grouped in one class, all is about coding style.

```
1   //------------------**driver_out.sv**-------
2   typedef virtual output_if output_vif;
3   class driver_out extends uvm_driver #(packet_out);
4       `uvm_component_utils(driver_out)
5       output_vif vif;
6   //-----------------------------------------
7   function new(string name = "driver_out", uvm_component parent = null);
8       super.new(name, parent);
9   endfunction
10  //-----------------------------------------
11  virtual function void build_phase(uvm_phase phase);
12      super.build_phase(phase);
13      assert(uvm_config_db#(output_vif)::get(this, "", "vif", vif));
14  endfunction
15  //-----------------------------------------
16  virtual task run_phase(uvm_phase phase);
17      super.run_phase(phase);
18  endtask
19  //-----------------------------------------
20  endclass
21  //-----------------------------------------
```

The monitor class is defined from `uvm_monitor`, like previously described it receives data from the DUT and sends them to scoreboard

```
1   //--------------------**monitor.sv**---------
2   class monitor extends uvm_monitor;
3       input_vif  vif;
4       event begin_record, end_record;
5       packet_in tr;
6       uvm_analysis_port #(packet_in) item_collected_port;
7       `uvm_component_utils(monitor)
8   //-----------------------------------------
9   function new(string name, uvm_component parent);
10      super.new(name, parent);
11      item_collected_port = new ("item_collected_port", this);
```

```
endfunction
//-----------------------------------------
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    assert(uvm_config_db#(input_vif)::get(this, "", "vif", vif));
    tr = packet_in::type_id::create("tr", this);
endfunction
//-----------------------------------------
virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    fork
        collect_transactions(phase);
        record_tr();
        join
endtask
//-----------------------------------------
virtual task collect_transactions(uvm_phase phase);
    wait(vif.rst === 1);
    forever begin
        -> begin_record;
        tr.A = vif.A;
        tr.B = vif.B;
        tr.ci = vif.ci;
        item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
//-----------------------------------------
virtual task record_tr();
    forever begin
        @(begin_record);
        begin_tr(tr, "monitor");
        @(end_record);
        end_tr(tr);
    end
endtask
//-----------------------------------------
endclass
//-----------------------------------------
```

Like the driver, the `monitor_out` uses virtual variables to get data from DUT.

```
//--------------------**monitor_out.sv-------
class monitor_out extends uvm_monitor;
    `uvm_component_utils(monitor_out)
    output_vif  vif;
    event begin_record, end_record;
    packet_out tr;
    uvm_analysis_port #(packet_out) item_collected_port;
//-----------------------------------------
function new(string name, uvm_component parent);
    super.new(name, parent);
```

```
11      item_collected_port = new ("item_collected_port", this);
12  endfunction
13  //---------------------------------------------
14  virtual function void build_phase(uvm_phase phase);
15      super.build_phase(phase);
16      assert(uvm_config_db#(output_vif)::get(this, "", "vif", vif));
17      tr = packet_out::type_id::create("tr", this);
18  endfunction
19  //---------------------------------------------
20  virtual task run_phase(uvm_phase phase);
21      super.run_phase(phase);
22      fork
23          collect_transactions(phase);
24          record_tr();
25      join
26  endtask
27  //---------------------------------------------
28  virtual task collect_transactions(uvm_phase phase);
29      @(negedge vif.rst);
30      forever begin
31          -> begin_record;
32          tr.data = vif.data;
33          tr.co = vif.co;             item_collected_port.write(tr);
34          @(posedge vif.clk);
35          -> end_record;
36      end
37  endtask
38  //---------------------------------------------
39  virtual task record_tr();
40      forever begin
41          @(begin_record);
42          begin_tr(tr, "monitor_out");
43          @(end_record);
44          end_tr(tr);
45      end
46  endtask
47  //---------------------------------------------
48  endclass
49  //---------------------------------------------
```

The agent instantiates the components : sequencer, driver and monitor. It consists of the handles of these components, after registration in the factory and constructing it, it uses `uvm_analysis_port` to get transactions then create the handls of sequencer,driver and monitor then connect monitor to `item_collected_port` and the driver with `seq_item_export`.

```
1  //--------------------**agent.sv-------------
2  class agent extends uvm_agent;
3      sequencer sqr;
4      driver    drv;
5      monitor   mon;
6      uvm_analysis_port #(packet_in) item_collected_port;
7      `uvm_component_utils(agent)
```

```
8  //-----------------------------------------
9  function new(string name = "agent", uvm_component parent = null);
10     super.new(name, parent);
11     item_collected_port = new("item_collected_port", this);
12 endfunction
13 //-----------------------------------------
14 virtual function void build_phase(uvm_phase phase);
15     super.build_phase(phase);
16     mon = monitor::type_id::create("mon", this);
17     sqr = sequencer::type_id::create("sqr", this);
18     drv = driver::type_id::create("drv", this);
19 endfunction
20 //-----------------------------------------
21 virtual function void connect_phase(uvm_phase phase);
22     super.connect_phase(phase);
23     mon.item_collected_port.connect(item_collected_port);
24     drv.seq_item_port.connect(sqr.seq_item_export);
25 endfunction
26 //-----------------------------------------
27 endclass: agent
28 //-----------------------------------------
```

The agent_out is used to connect the driver_out and monitor_out,again it is all about code style.

```
1  //--------------------**agent_out.sv**-------
2  class agent_out extends uvm_agent;
3      driver_out    drv;
4      monitor_out   mon;
5
6      uvm_analysis_port #(packet_out) item_collected_port;
7
8      `uvm_component_utils(agent_out)
9  //-----------------------------------------
10 function new(string name = "agent_out", uvm_component parent = null);
11     super.new(name, parent);
12     item_collected_port = new("item_collected_port", this);
13 endfunction
14 //-----------------------------------------
15 virtual function void build_phase(uvm_phase phase);
16     super.build_phase(phase);
17     mon = monitor_out::type_id::create("mon_out", this);
18     drv = driver_out::type_id::create("drv_out", this);
19 endfunction
20 //-----------------------------------------
21 virtual function void connect_phase(uvm_phase phase);
22     super.connect_phase(phase);
23     mon.item_collected_port.connect(item_collected_port);
24 endfunction
25 //-----------------------------------------
26 endclass: agent_out
27 //-----------------------------------------
```

The reference model is extended from `uvm_component`, it is registered in the factory and constructed. It has two handles of `packet_in` and `packet_out`. In run phase, it calls a function called "sum" and "sum1" in c++(can be in c++, c, systemverilog or matlab,...), these functions are imported in the top of the file using "import "DPI -C" context function", it gets the transaction inputs and then pass them as arguments to these functions.

```
1  //--------------------**refmod.sv**----------
2  import "DPI-C" context function int sum(int x, int y , int carry_in);
3  import "DPI-C" context function int sum1(int x, int y , int carry_in);
4
5  class refmod extends uvm_component;
6      `uvm_component_utils(refmod)
7      packet_in tr_in;
8      packet_out tr_out;
9      integer a, b;
10     uvm_get_port #(packet_in) in;
11     uvm_put_port #(packet_out) out;
12 //-----------------------------------------
13 function new(string name = "refmod", uvm_component parent);
14     super.new(name, parent);
15     in = new("in", this);
16     out = new("out", this);
17 endfunction
18 //-----------------------------------------
19 virtual function void build_phase(uvm_phase phase);
20     super.build_phase(phase);
21     tr_out = packet_out::type_id::create("tr_out", this);
22 endfunction: build_phase
23 //-----------------------------------------
24 virtual task run_phase(uvm_phase phase);
25     super.run_phase(phase);
26     forever begin
27         in.get(tr_in);
28         tr_out.data = sum(tr_in.A, tr_in.B, tr_in.ci);
29 //uvm_report_info("DAAATAAAAAA", "");
30         tr_out.co = sum1(tr_in.A, tr_in.B, tr_in.ci);
31         out.put(tr_out);
32     end
33 endtask: run_phase
34 //-----------------------------------------
35 endclass: refmod
36 //-----------------------------------------
```

The comparator or scoreboard is extended from `uvm_scoreboard`, it defines some string (to be shown later during the simulation) that are parametrized to accept a data object of type T. In run phase, it raises the objection and drops it in order to coordinate status information between the participant components. The put() is used in refmod to get the expected result and since the comparator is the receiver with respect to reference model, it must define the put task which is a blocking task. The `try_put()` is a non blocking function, will attempt to perform a put operation and will return true if it succeeds, and false if does not. If it fails, then you have to try again. The `can_put()` function is just a test to see if a non-blocking put operation would succeed without actually performing the operation.

```
1  //--------------------**comparator.sv--------
2  class comparator #(type T = packet_out) extends uvm_scoreboard;
3    typedef comparator #(T) this_type;
4    `uvm_component_param_utils(this_type)
5    const static string type_name = "comparator #(T)";
6    uvm_put_imp #(T, this_type) from_refmod;
7    uvm_analysis_imp #(T, this_type) from_dut;
8    typedef uvm_built_in_converter #( T ) convert;
9    int m_matches, m_mismatches;
10   T exp;
11   bit free;
12   event compared, end_of_simulation;
13 //-----------------------------------------
14 function new(string name, uvm_component parent);
15     super.new(name, parent);
16     from_refmod = new("from_refmod", this);
17     from_dut = new("from_dut", this);
18     m_matches = 0;
19     m_mismatches = 0;
20     exp = new("exp");
21     free = 1;
22 endfunction
23 //-----------------------------------------
24 virtual function string get_type_name();
25     return type_name;
26 endfunction
27 //-----------------------------------------
28 task run_phase(uvm_phase phase);
29     phase.raise_objection(this);
30     @(end_of_simulation);
31     phase.drop_objection(this);
32 endtask
33 //-----------------------------------------
34 virtual task put(T t);
35     if(!free) @compared;
36     exp.copy(t);
37     free = 0;
38     @compared;
39     free = 1;
40 endtask
41 //-----------------------------------------
42 virtual function bit try_put(T t);
43     if(free) begin
44         exp.copy(t);
45         $display("exp ",exp);
46         free = 0;
47         return 1;
48     end
49     else return 0;
50 endfunction
51 //-----------------------------------------
```

```
52  virtual function bit can_put();
53      return free;
54  endfunction
55  //-----------------------------------------
56  virtual function void write(T rec);
57      if (free)
58          uvm_report_fatal("No expect transaction to compare with", "");
59      if(!(exp.compare(rec))) begin
60          uvm_report_warning("Comparator Mismatch", "");
61          m_mismatches++;
62      end
63      else begin
64          uvm_report_info("Comparator Match", "");
65          m_matches++;
66      end
67      if(m_matches+m_mismatches > 100)
68        -> end_of_simulation;
69      -> compared;
70    endfunction
71  //-----------------------------------------
72  endclass
73  //-----------------------------------------
```

The environment class connects and instantiates the component that are in bottom layer such as agent, agent_out,refmod and comparator.

```
1   //--------------------**env.sv**-------------
2   class env extends uvm_env;
3       agent       mst;
4       refmod      rfm;
5       agent_out   slv;
6       comparator #(packet_out) comp;
7       uvm_tlm_analysis_fifo #(packet_in) to_refmod;
8       `uvm_component_utils(env)
9   //-----------------------------------------
10  function new(string name, uvm_component parent = null);
11      super.new(name, parent);
12      to_refmod = new("to_refmod", this);
13  endfunction
14  //-----------------------------------------
15  virtual function void build_phase(uvm_phase phase);
16      super.build_phase(phase);
17      mst = agent::type_id::create("mst", this);
18      slv = agent_out::type_id::create("slv", this);
19      rfm = refmod::type_id::create("rfm", this);
20      comp = comparator#(packet_out)::type_id::create("comp", this);
21  endfunction
22  //-----------------------------------------
23  virtual function void connect_phase(uvm_phase phase);
24      super.connect_phase(phase);
25      // Connect MST to FIFO
26      mst.item_collected_port.connect(to_refmod.analysis_export);
27      // Connect FIFO to REFMOD
```

```
28      rfm.in.connect(to_refmod.get_export);
29      //Connect scoreboard
30      rfm.out.connect(comp.from_refmod);
31      slv.item_collected_port.connect(comp.from_dut);
32  endfunction
33  //-----------------------------------------
34  virtual function void end_of_elaboration_phase(uvm_phase phase);
35      super.end_of_elaboration_phase(phase);
36  endfunction
37  //-----------------------------------------
38  virtual function void report_phase(uvm_phase phase);
39      super.report_phase(phase);
40      `uvm_info(get_type_name(), $sformatf("Reporting matched %0d", comp
            .m_matches), UVM_NONE)
41      if (comp.m_mismatches) begin
42          `uvm_error(get_type_name(), $sformatf("Saw %0d mismatched
                samples", comp.m_mismatches))
43      end
44  endfunction
45  //-----------------------------------------
46  endclass
47  //-----------------------------------------
```

Simple_test is where the container environment and transactions are instantiated.

```
1   //--------------------**simple_test.sv**-----
2   class simple_test extends uvm_test;
3       env env_h;
4       sequence_in seq;
5       `uvm_component_utils(simple_test)
6   //-----------------------------------------
7   function new(string name, uvm_component parent = null);
8       super.new(name, parent);
9   endfunction
10  //-----------------------------------------
11  virtual function void build_phase(uvm_phase phase);
12      super.build_phase(phase);
13      env_h = env::type_id::create("env_h", this);
14      seq = sequence_in::type_id::create("seq", this);
15  endfunction
16  //-----------------------------------------
17  task run_phase(uvm_phase phase);
18      seq.start(env_h.mst.sqr);
19  endtask: run_phase
20  //-----------------------------------------
21  endclass
22  //-----------------------------------------
```

Top is the top level where DUT is instantiated. All included files can grouped and packed in one package then import it.

```
1   //--------------------**top.sv**-------------
2   import uvm_pkg::*;
3   `include "uvm_macros.svh"
```

```
4  `include "./input_if.sv"
5  `include "./output_if.sv"
6  `include "./packet_in.sv"
7  `include "./packet_out.sv"
8  `include "./sequence_in.sv"
9  `include "./sequencer.sv"
10 `include "./driver.sv"
11 `include "./driver_out.sv"
12 `include "./monitor.sv"
13 `include "./monitor_out.sv"
14 `include "./agent.sv"
15 `include "./agent_out.sv"
16 `include "./refmod.sv"
17 `include "./comparator.sv"
18 `include "./env.sv"
19 `include "./simple_test.sv"
20 //Top
21 module top;
22     logic clk;
23     logic rst;
24     initial begin
25         clk = 0;
26         rst = 1;
27         #22 rst = 0;
28     end
29     always #5 clk = !clk;
30     input_if in(clk, rst);
31     output_if out(clk, rst);
32     adder sum(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .ci(in.ci),.
           s(out.data), .co(out.co));
33     initial begin
34         `ifdef INCA
35             $recordvars();
36         `endif
37         `ifdef VCS
38             $vcdpluson;
39         `endif
40         `ifdef QUESTA
41             $wlfdumpvars();
42             set_config_int("*", "recording_detail", 1);
43         `endif
44         uvm_config_db#(input_vif)::set(uvm_root::get(), "*.env_h.mst
               .*", "vif", in);
45         uvm_config_db#(output_vif)::set(uvm_root::get(), "*.env_h.slv
               .*",  "vif", out);
46         run_test("simple_test");
47     end
48 endmodule
49 //-----------------------------------------
```

to compile a code in QuestaSim we run the commands :

- vcom : for vhdl files

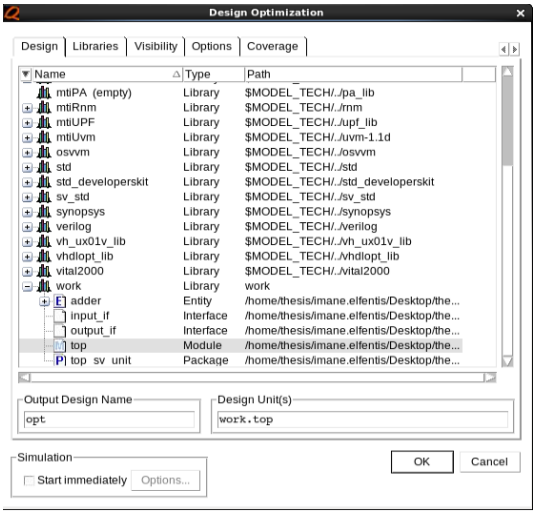


Figure 5.3.: design optimization window

- vlog : for verilog and systemverilog files
- -mixedsvvh : for packages either are defined in vhdl or systemverilog.

∗ So as a first step run command step" vcom fa.sv"

∗ second step run command "vlog top.sv external.cpp -dpiheader external.h"

we start with optimization: **simulate** –> **design optimization** –> **work** –> select top and give a name to output design name(I gave the name : opt). Go to visibility and check "apply full visibility to all modules" figures: 5.3 and 5.4. Go to coverage and enable for the

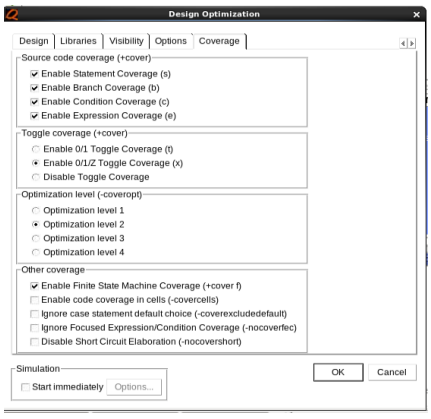


Figure 5.4.: Enabling coverage-Questasim

moment source code coverage and enable 0/1/z toggle coverage(x).
we start with simulation: **simulate** –> **start simulation** –> **work** –> **opt**. Go to others and enable code coverage then OK. Run 400 ns.
In the transcript we can read and observe if we have match or mismatch and in what clock cycle see figure 5.5 and the figure 5.6 shows the wav forms.

Code coverage is the only verification metric generated automatically from design source in RTL or gates. While most verification plans require a high level of code coverage, it does not

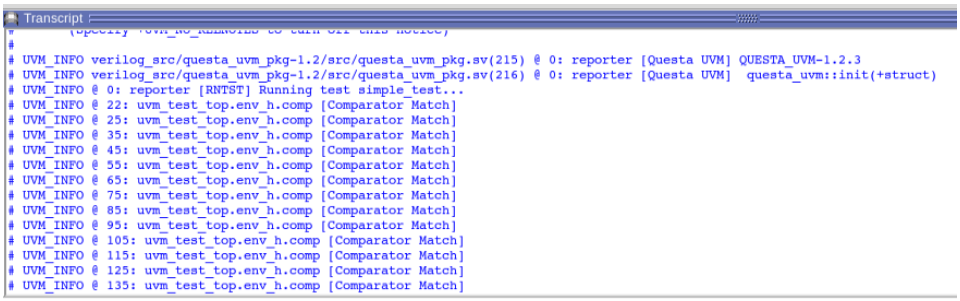


Figure 5.5.: Results after simulation

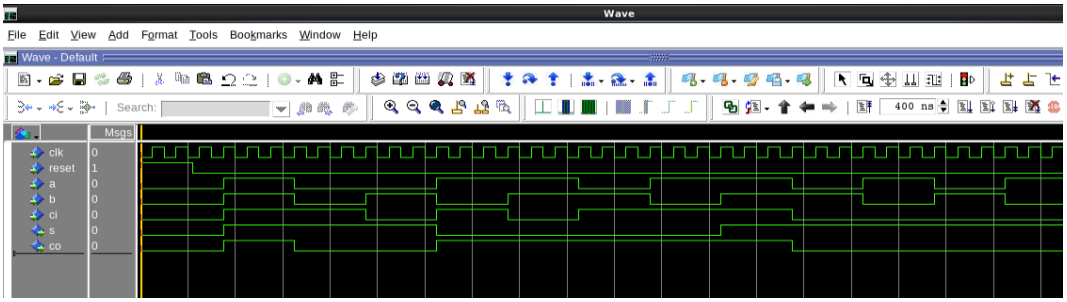


Figure 5.6.: waves form after simulation

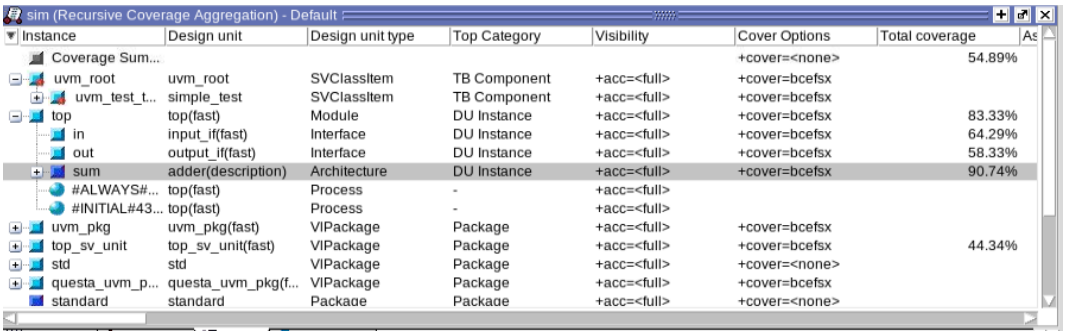


Figure 5.7.: Structural window

necessarily indicate correctness of your design. Code coverage measures only how often a suite of tests exercises certain aspects of the source.
Missing code coverage is usually an indication of one of two things: either unused code, or holes in the tests. Because it is automatically generated, code coverage is a metric achieved with relative ease, obtained early in the verification cycle. A sophisticated exclusions mechanism enables you to achieve 100% code coverage, even for designs where unused code would otherwise make it impossible to achieve coverage. Code coverage statistics are collected and can be saved into the Unified Coverage DataBase for later analysis. The structural window

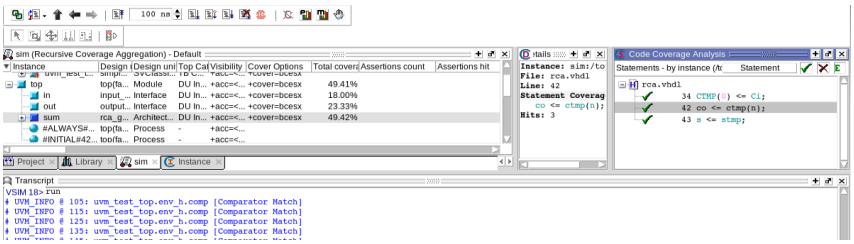


Figure 5.8.: structural-code coverage analysis-coverage details

figure 5.7 shows the results in hierarchical tree, it is mainly used as design navigation aid, it is not specific of code coverage but it adds additional information when the simulation is invoked in the code coverage mode, it displays coverage data and graphs for each design object or file including coverage from child instances compiled with coverage argument, in this example we consider **sum** instance that has 56,52% for statement. In this example, we have only one instance (not child instances).
Coverage analysis window is key to navigate through coverage data that can be enabled

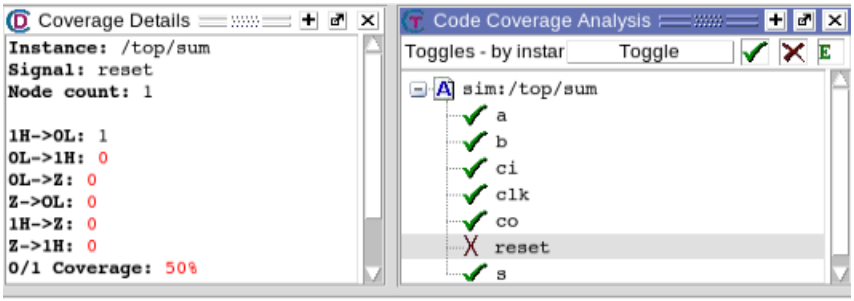
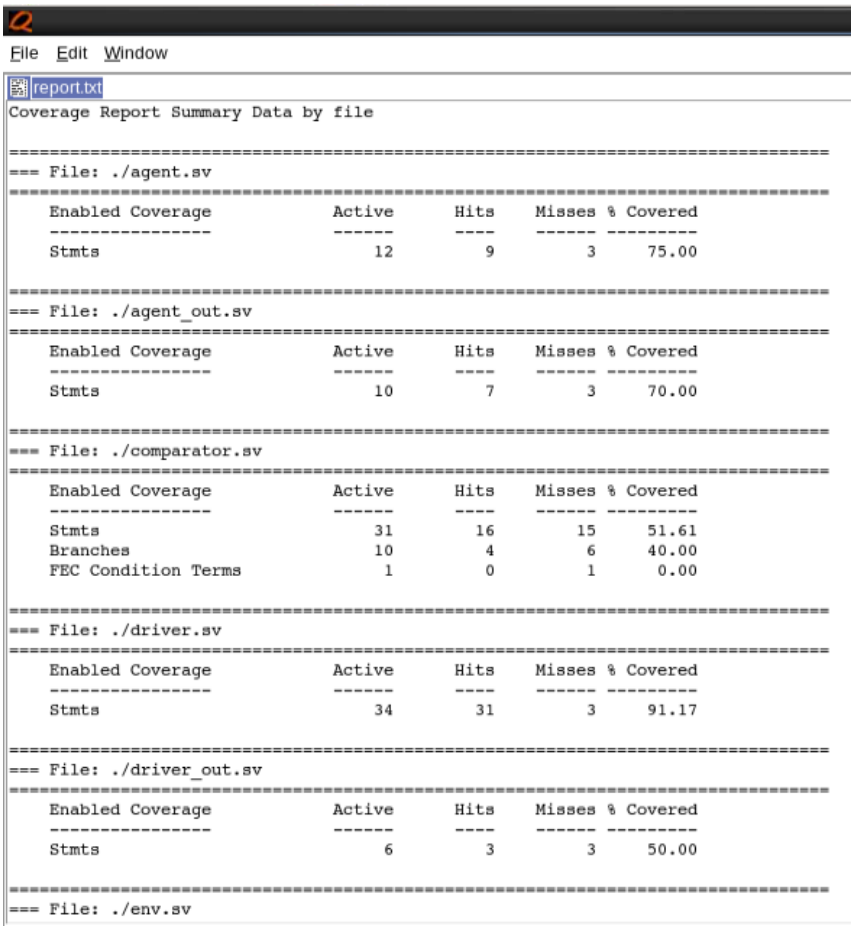


Figure 5.9.: Code coverage analysis and coverage details

from **View** –> **coverage** –>**code coverage analysis**. It is context dependent, it displays the line numbers of covered, uncovered and excluded items in the file underlined selected in structure or instance window.
The coverage details window also is available in same menu **View** –> **coverage** –> **details**, it shows complementary information based on selection from analysis window. On the coverage analysis we can set what type of metrics displayed can be set and filters are available to show or hide covered, missed or excluded lines.
The report of code coverage reports the percentage of each file figur 5.10
In **sim** window go to **structure** –> **code coverage** –> **code coverage reports**



File Edit Window

report.txt

```
Coverage Report Summary Data by file

================================================================================
=== File: ./agent.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           12         9         3     75.00

================================================================================
=== File: ./agent_out.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           10         7         3     70.00

================================================================================
=== File: ./comparator.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           31        16        15     51.61
    Branches                        10         4         6     40.00
    FEC Condition Terms              1         0         1      0.00

================================================================================
=== File: ./driver.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           34        31         3     91.17

================================================================================
=== File: ./driver_out.sv
================================================================================
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                            6         3         3     50.00

================================================================================
=== File: ./env.sv
```

Figure 5.10.: Code coverage Report

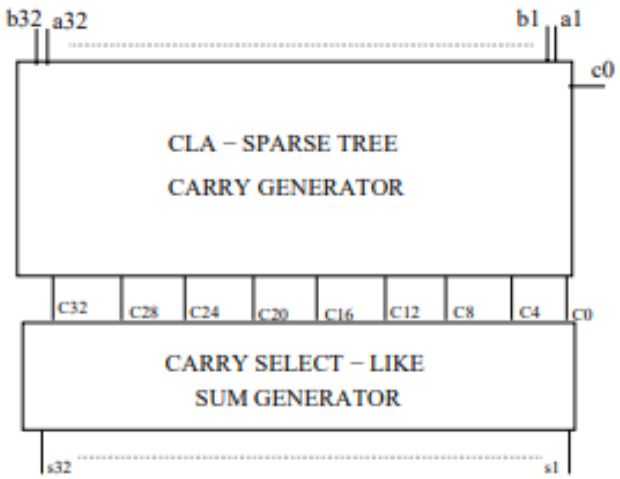


Figure 5.11.: P4Adder

5.1.2. Example of P4Adder

The general structure of the adder used in the pentium 4 is described in the figure 5.11. The sub-blocks are a carry select for the sum generation and a sparse tree for the carry generation. The first stage receives the inputs(operands) A and B. this stage consists of blocks that are used to create the prefix signals i.e Generate and Propagate signals. The Prefix phase consist of sparse Carry merge block [10]. The Carry merge block is used to generate the first predetermined number of carry signals based on the generate and propagate signals. Similarly predetermined number of carry signals will be generated using another carry merge circuit. Those predetermined number of carry signals will be again merged to calculate the group carry. The Group carry should be calculated for every 4 bit summation block. The Summation Stage is the second step, in the parallel prefix adder to perform the addition operation. It consists of Carry select adder to calculate the output sum. It consists of a ripple carry adder block, and a multiplexer.The code used to implement and verify is in Appendix.

5.1.3. Sequential circuit: D register

In this section we are going to verify a sequential circuit starting from simple one : D register

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3  entity register_generic is
4          Generic (N: integer:=4);
5          Port (  DIN:    In      std_logic_vector(N-1 downto 0) ;
6                  Reset:  In      std_logic;
7                  clk:    In      std_logic;
8                  DOUT:   Out     std_logic_vector(N-1 downto 0));
9  end register_generic;
10 architecture BEH of register_generic is
11 begin
12   p1:process(clk, RESET)
13       begin
14          if RESET='1' then
15                 DOUT <= (others => '0');
16          ELSIF rising_edge(clk) THEN
17                         DOUT <= DIN;
18          END IF;
19 end process;
20 end BEH;
```

The code for the components in UVM are the same(examples of fulladder and in Appendix), the main modifications in the components are:

```
1  //------------------**input_if.sv**------
2  interface input_if(input clk, rst);
3      logic [3:0] A;
4      modport port(input clk, rst, A);
5  endinterface
```

```
1  //------------------**output_if.sv**-----
2  interface output_if(input clk, rst);
3      logic [3:0] data;
```

```
    modport port(input clk, rst, output  data);
endinterface
```

```
//----------------**packet_in.sv**-------
class packet_in extends uvm_sequence_item;
    rand logic A;
    logic clk;
    logic rst;
    `uvm_object_utils_begin(packet_in)
        `uvm_field_int(A, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end
     //...code
endclass: packet_in
//---------------------------------------
```

```
//------------------**packet_out.sv-----
class packet_out extends uvm_sequence_item;
    rand logic data;
    `uvm_object_utils_begin(packet_out)
        `uvm_field_int(data, UVM_ALL_ON|UVM_HEX)
    `uvm_object_utils_end
    //...code
endclass: packet_out
//---------------------------------------
```

```
//--------------------**driver.sv**-----
    //...code
virtual protected task drive_transfer(packet_in tr);
    @(posedge vif.clk);
    vif.A = tr.A;
    @(posedge vif.clk);
    @(posedge vif.clk);
    -> end_record;
    @(posedge vif.clk); //hold time
    endtask
    //...code
```

```
//--------------------**monitor.sv**----
    //...code
virtual task collect_transactions(uvm_phase phase);
    wait(vif.rst === 1);
    forever begin
        -> begin_record;
        tr.A = vif.A;              item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
    //...code
```

```
//--------------------**monitor_out.sv---
    //...code
```

```
3  virtual task collect_transactions(uvm_phase phase);
4      @(negedge vif.rst);
5      forever begin
6          -> begin_record;
7          tr.data = vif.data;            item_collected_port.write(tr);
8          @(posedge vif.clk);
9          -> end_record;
10     end
11 endtask
12     //...code
```

For sequential model, the easiest way to build a reference model in my opinion is to use systemverilog because,I can use timing notations.

```
1  //--------------------**refmod.sv**----
2  class refmod extends uvm_component;
3      //...code
4  virtual task run_phase(uvm_phase phase);
5      super.run_phase(phase);
6      forever begin
7          in.get(tr_in);
8          fork
9              wait(tr_in.rst === 0);
10             @(posedge tr_in.clk);
11             tr_out.data = tr_in.A;
12         join_none
13         out.put(tr_out);
14     end
15 endtask: run_phase
16 //---------------------------------------
17 endclass: refmod
18 //---------------------------------------
```

5.1.4. Serial In Serial Out SISO

Serial-in, serial-out shift registers delay data by one clock time for each stage. They will store a bit of data for each register.The implementation and verification code used are in Appendix.

5.1.5. Control unit

The circuit is a simple control unit connected to an Alu that consists of addition, subtraction, shifting using SISO shifter. The implementation and code used for reference model are described in Appendix.

5.1.6. FSM

As an example for FSM is an adder that has six 6 inputs :clk, rst, two operands, and two signals ready and valid and the same for outputs, it has two signals to send data and receive it. Since the example is in systemverilog [12], we add it in"top.sv" like other components in systemverilog. Following the previous steps to compile it, simulate it and observe the

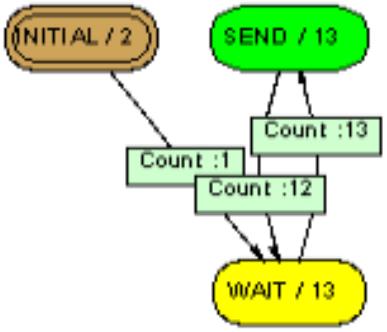


Figure 5.12.: FSM

total coverage, we can also have a window that shows a drown FSM figure 5.12. **view** –> **coverage** –> **code coverage analysis** –> **FSM** –> double click on state.

```
1  module adder(input_if.port inter, output_if.port out_inter,
2     output enum logic [1:0] {INITIAL,WAIT,SEND} state)
3      always_ff @(posedge inter.clk)
4          if(inter.rst) begin
5              inter.ready <= 0;
6              out_inter.data <= 'x;
7              out_inter.valid1 <= 0;
8              state <= INITIAL;
9          end
10         else case(state)
11                 INITIAL: begin
12                     inter.ready <= 1;
13                     state <= WAIT;
14                 end
15                 WAIT: begin
16                     if(inter.valid) begin
17                         inter.ready <= 0;
18                         out_inter.data <= inter.A + inter.B;
                                              out_inter.valid1 <= 1;
19                         state <= SEND;
20                     end
21                 end
22                 SEND: begin                    if(out_inter.ready1)
                       begin                        out_inter.valid1 <= 0;
23                         inter.ready <= 1;
24                         state <= WAIT;
25                     end
26                 end
27         endcase
28 endmodule: adder
```

```
1  //--------------------**refmod.sv**-----
2      ///...code
3  import "DPI-C" context function int sum(int a, int b);
```

```
4       virtual task run_phase(uvm_phase phase);
5           super.run_phase(phase);
6           forever begin
7               in.get(tr_in);
8               tr_out.data = sum(tr_in.A, tr_in.B);
9               out.put(tr_out);
10          end
11      endtask: run_phase
12      //...code
```

```
1   //--------------------**external.cpp**--
2   #include <stdio.h>
3   extern "C" int sum(int a, int b){
4     return a+b;
5   }
```

5.2. Second coding style

Writing the code without using predefined classes in UVM might look like: l let 'us take a simple example of an adder described in VHDL used in the previous code style

```
1   //----------------**transaction.sv**----
2   class transaction;
3     //declaring the transaction items
4       rand bit  a;
5       rand bit  b;
6       rand bit  ci;
7            bit  s;
8            bit  co;
9   //---------------------------------------
10  function void display(string name);
11      $display("-------------------------");
12      $display("- %s ",name);
13      $display("-------------------------");
14      $display("- a = %0d, b = %0d,ci = %0d",a,b,ci);
15      $display("- s = %0d,co = %0d",s,co);
16      $display("-------------------------");
17  endfunction
18  endclass
```

```
1   //--------------------**interface.sv**---
2   interface intf(input logic clk,reset);
3       //declaring the signals
4       logic  valid;
5       logic  a;
6       logic  b;
7       logic  ci;
8       logic  s;
9       logic  co;
10  endinterface
11  //---------------------------------------
```

```
1  //--------------------**generator.sv**--
2  class generator;
3      //declaring transaction class
4      rand transaction trans;
5      //repeat count, to specify number of items to generate
6      int  repeat_count;
7      //mailbox, to generate and send the packet to driver
8      mailbox gen2driv;
9      //event, to indicate the end of transaction generation
10     event ended;
11     //constructor
12 //---------------------------------------
13 function new(mailbox gen2driv);
14     //getting the mailbox handle from env,
15     //in order to share the transaction packet between
16     // the generator and driver, the same mailbox is shared between
           both.
17     this.gen2driv = gen2driv;
18 endfunction
19 //---------------------------------------
20     //main task, generates(create and randomizes) the
21     //repeat_count number of transaction packets and puts into mailbox
22     task main();
23         repeat(repeat_count) begin
24         trans = new();
25         if( !trans.randomize() ) $fatal("Gen:: trans randomization
               failed");
26         trans.display("[ Generator ]");
27         gen2driv.put(trans);
28     end
29     -> ended; //triggering indicatesthe end of generation
30 endtask
31 //---------------------------------------
32 endclass
33 //---------------------------------------
```

```
1  //--------------------**driver.sv**---
2      //gets the packet from generator and drive the transaction paket
3      //items into interface (interface is connected to DUT, so the
           items
4      //driven into interface signal will get driven in to DUT)
5  class driver;
6      //used to count the number of transactions
7      int no_transactions;
8      //creating virtual interface handle
9      virtual intf vif;
10     //creating mailbox handle
11     mailbox gen2driv;
12     //constructor
13 //---------------------------------------
14 function new(virtual intf vif,mailbox gen2driv);
15     //getting the interface
```

```
16      this.vif = vif;
17      //getting the mailbox handles from  environment
18      this.gen2driv = gen2driv;
19  endfunction
20  //-------------------------------------
21      //Reset task, Reset the Interface signals to default/initial
            values
22  task reset;
23      wait(vif.reset);
24      $display("[ DRIVER ] ----- Reset Started -----");
25      vif.a      <= 0;
26      vif.b      <= 0;
27      vif.ci     <= 0;
28      vif.valid <= 0;
29      wait(!vif.reset);
30      $display("[ DRIVER ] ----- Reset Ended   -----");
31  endtask
32  //-------------------------------------
33      //drivers the transaction items to interface signals
34  task main;
35      forever begin
36          transaction trans;
37          gen2driv.get(trans);
38          @(posedge vif.clk);
39          vif.valid <= 1;
40          vif.a      <= trans.a;
41          vif.b      <= trans.b;
42          vif.ci     <= trans.ci;
43          @(posedge vif.clk);
44          vif.valid <= 0;
45          trans.s    = vif.s;
46          trans.co   = vif.co;
47          @(posedge vif.clk);
48          trans.display("[ Driver ]");
49          no_transactions++;
50      end
51  endtask
52  //-------------------------------------
53  endclass
54  //-------------------------------------
```

```
1   //--------------------**monitor.sv**--
2       //Samples the interface signals, captures into transaction packet
            and send the packet to scoreboard.
3   class monitor;
4       //creating virtual interface handle
5       virtual intf vif;
6       //creating mailbox handle
7       mailbox mon2scb;
8       //constructor
9   //-------------------------------------
10  function new(virtual intf vif,mailbox mon2scb);
```

```
11      //getting the interface
12      this.vif = vif;
13      //getting the mailbox handles from  environment
14      this.mon2scb = mon2scb;
15  endfunction
16  //---------------------------------------
17      //Samples the interface signal and send the sample packet to
            scoreboard
18  task main;
19      forever begin
20          transaction trans;
21          trans = new();
22          @(posedge vif.clk);
23          wait(vif.valid);
24          trans.a    = vif.a;
25          trans.b    = vif.b;
26          trans.ci    = vif.ci;
27          @(posedge vif.clk);
28          trans.s    = vif.s;
29          trans.co    = vif.co;
30          @(posedge vif.clk);
31          mon2scb.put(trans);
32          trans.display("[ Monitor ]");
33      end
34  endtask
35  //---------------------------------------
36  endclass
37  //---------------------------------------
```

```
1   //--------------------environment.sv**--
2   `include "transaction.sv"
3   `include "generator.sv"
4   `include "driver.sv"
5   `include "monitor.sv"
6   `include "scoreboard.sv"
7   class environment;
8       //generator and driver instance
9       generator   gen;
10      driver      driv;
11      monitor     mon;
12      scoreboard  scb;
13      //mailbox handle's
14      mailbox gen2driv;
15      mailbox mon2scb;
16      //virtual interface
17      virtual intf vif;
18      //constructor
19  //---------------------------------------
20  function new(virtual intf vif);
21      //get the interface from test
22      this.vif = vif;
23      //creating the mailbox (Same handle will be shared across
```

```
        generator and driver)
24      gen2driv = new();
25      mon2scb  = new();
26      //creating generator and driver
27      gen  = new(gen2driv);
28      driv = new(vif,gen2driv);
29      mon  = new(vif,mon2scb);
30      scb  = new(mon2scb);
31  endfunction
32  //---------------------------------------
33  task pre_test();
34      driv.reset();
35  endtask
36  //---------------------------------------
37  task test();
38      fork
39          gen.main();
40          driv.main();
41          mon.main();
42          scb.main();
43      join_any
44  endtask
45  //---------------------------------------
46  task post_test();
47      wait(gen.ended.triggered);
48      wait(gen.repeat_count == driv.no_transactions); //Optional
49      wait(gen.repeat_count == scb.no_transactions);
50  endtask
51  //---------------------------------------
52      //run task
53  task run;
54      pre_test();
55      test();
56      post_test();
57      $finish;
58  endtask
59  //---------------------------------------
60  endclass
61  //---------------------------------------
```

```
1   //--------------------**scoreboard.sv**--
2   //gets the packet from monitor, Generated the expected result and
        compares with the //actual result recived from Monitor
3   class scoreboard;
4       int x =0;
5       int y =0;
6       //creating mailbox handle
7       mailbox mon2scb;
8       //used to count the number of transactions
9       int no_transactions;
10      //constructor
11  //---------------------------------------
```

```
12  function new(mailbox mon2scb);
13      //getting the mailbox handles from  environment
14      this.mon2scb = mon2scb;
15  endfunction
16  //---------------------------------------
17      //Compares the Actual result with the expected result
18  task main;
19      transaction trans;
20      forever begin
21          mon2scb.get(trans);
22          if(((trans.a ^trans.b ^ trans.ci) == trans.s) && ((trans.a &
                trans.b) |(trans.a & trans.ci) |( trans.ci & trans.b)) ==
                trans.co)
23              $display("Result is as Expected and x = %0d, count is %0d,
                     ",x,y++);
24          else
25              $error("Wrong Result.\n\tExpeced:  x = %0d",x++);
26          no_transactions++;
27          trans.display("[ Scoreboard ]");
28      end
29  endtask
30  //---------------------------------------
31  endclass
32  //---------------------------------------
```

```
1   //--------------------**directedtest.sv**--
2   `include "environment.sv"
3   program test(intf i_intf);
4   class my_trans extends transaction;
5       bit [1:0] count;
6   //---------------------------------------
7   function void pre_randomize();
8       a.rand_mode(0);
9       b.rand_mode(0);
10      // a = 10;
11      //b = 12;
12  endfunction
13  //---------------------------------------
14  endclass
15  //---------------------------------------
16      //declaring environment instance
17      environment env;
18      my_trans my_tr;
19      initial begin
20          //creating environment
21          env = new(i_intf);
22          my_tr = new();
23          //setting the repeat count of generator as 4, means to
                generate 4 packets
24          env.gen.repeat_count = 16;
25          env.gen.trans = my_tr;
26          //calling run of env, it interns calls generator and driver
```

```
            main tasks.
27          env.run();
28      end
29  endprogram
```

```
1   //--------------------**testbench.sv**--
2   //tbench_top or testbench top, this is the top most file, in which DUT
        and Verification environment are connected.
3   //including interfcae and testcase files
4   `include "interface.sv"
5   module tbench_top;
6       //clock and reset signal declaration
7       bit clk;
8       bit reset;
9       //clock generation
10      always #5 clk = ~clk;
11      //reset Generation
12      initial begin
13          reset = 1;
14          #5 reset =0;
15      end
16      //creatinng instance of interface, inorder to connect DUT and
            testcase
17      intf i_intf(clk,reset);
18      //Testcase instance, interface handle is passed to test as an
            argument
19      test t1(i_intf);
20      //DUT instance, interface signals are connected to the DUT ports
21      adder DUT (
22      .clk(i_intf.clk),
23      .reset(i_intf.reset),
24      .a(i_intf.a),
25      .b(i_intf.b),
26      .ci(i_intf.ci),
27      .valid(i_intf.valid),
28      .s(i_intf.s),
29      .co(i_intf.co)
30      );
31
32      //enabling the wave dump
33      initial begin
34          $dumpfile("dump.vcd"); $dumpvars;
35      end
36  endmodule
```

Chapter 6.

Conclusion

This thesis is focused in the study and evaluation of verification of devices. The first part is dedicated to technology challenges explaining technology options such as static technologies,simulation technologies and formal technologies and explaining SOC verification methodology showing the different approaches in verification and when they are used such as top-down verification, bottom-up verification, platform-based verification.
The second part, the focus is brought to the latest advancement in the verification methodology which is UVM, it is designed to enable creation of robust, reusable, interoperable verification IP and testbench components.
In the third section of this thesis, it shows the hierarchy of UVM and discusses all the components of the testbench : sequencer, driver, monitor, agent, scoreboard, sequence items, environment and test then testbench. This section goes through all these components and explains them in details how the connection between all of them is done and how to move from layer to layer using transaction level modeling TLM and Constrained Random Verification which show also different UVM phases. While going through each component, a starting point of how to write a code in UVM using Systemverilog is given, and how to compare the module under verification MUV(device under test: DUT) with a golden model that it might be written in other language like C, C++ or MATLAB ..., so to enable the reader to write his own code for its own project.
In the next section, an explained example is discussed showing all the steps discussed in the previous section are shown, this example, the device under test DUT is a full adder that is implemented in VHDL and the code used to verify it in SV, deriving the classes from pre-defined component in UVM also mentioning the steps followed for using Questasim. In the last section, it is discussed and given the code for combinational, sequential, finite state machine and a simple control unit in UVM.
The Appendix consists of different examples are built during the thesis period, those examples are extended from the devices used in the previous section.
One of the work to be extended in the future is to verify more complex circuit like having multiple agents, interfaces, specially different kind of transactions, also dedicate some effort to Hardware/Software Co-verification because an SoC is ready only when both, its hardware and software components are ready. We cannot ship silicon until its software is ready because without software, hardware is pretty useless. The Current designs invariably have both the digital and analog components within a block and also at SoC level. Without correct verification of analog voltage levels to digital binary and vice versa also known as Analog/Mixed Signal (AMS) Verification, the design will be dead on arrival.

Appendix A.

Appendix

For the following circuits, it is shown only the portion of code that will change, for the rest won't be changed like env.sv , `agent_out.sv`, sequencer.sv, `sequence_in.sv` and comparator.sv..

rca.vhdl

```
1  //---------------------**rca.vhdl**----------
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  entity RCA_generic is
6          Generic (   N : integer := 4);
7          Port    ( clk   : in  std_logic;
8                    reset : in  std_logic;
9                    A     : In  std_logic_vector(N-1 downto 0);
10                   B     : In  std_logic_vector(N-1 downto 0);
11                   Ci    : In  std_logic;
12                   S     : Out std_logic_vector(N-1 downto 0);
13                   Co    : Out std_logic);
14 end RCA_generic;
15
16 architecture STRUCTURAL of RCA_generic is
17 signal STMP : std_logic_vector(N-1 downto 0);
18 signal CTMP : std_logic_vector(N downto 0);
19   component adder
20     port(         clk   : in  std_logic;
21                   reset : in  std_logic;
22                   a     : in  std_logic;
23                   b     : in  std_logic;
24                   ci    : in  std_logic;
25                   s     : out std_logic;
26                   co    : out std_logic);
27   end component;
28 begin
29 CTMP(0) <= Ci;
30 fai4:for i in 1 to N generate
31 fai5:adder
32           Port Map (clk,reset,a(i-1), b(i-1), ctmp(i-1), STMP(i-1),
                 ctmp(i));
33 end generate ;
34 co <= ctmp(n);
35 s <= stmp;
```

```
36 end structural;
```

```
1  //--------------------**packet_in.sv**------
2  class packet_in extends uvm_sequence_item;
3      rand logic A ;
4      rand logic B ;
5      rand logic ci;
6  `uvm_object_utils_begin(packet_in) // register in the factory
7      `uvm_field_int(A , UVM_ALL_ON|UVM_HEX) `uvm_field_int(B ,
           UVM_ALL_ON|UVM_HEX)
8       `uvm_field_int(ci, UVM_ALL_ON|UVM_HEX)
9  `uvm_object_utils_end
10     //...code
11 endclass: packet_in
12 //---------------------------------------
```

```
1  //--------------------**packet_out.sv**-----
2  class packet_out extends uvm_sequence_item;
3      rand logic data;
4      rand logic co;
5  `uvm_object_utils_begin(packet_out)
6      `uvm_field_int(data, UVM_ALL_ON|UVM_HEX)
7      `uvm_field_int(co, UVM_ALL_ON|UVM_HEX)
8  `uvm_object_utils_end
9      //...code
10 endclass: packet_out
11 //---------------------------------------
```

```
1  //--------------------**driver.sv**---------
2      //...code
3  virtual protected task drive_transfer(packet_in tr);
4      @(posedge vif.clk);
5      vif.A = tr.A;
6      vif.B = tr.B;
7      vif.ci = tr.ci;
8      @(posedge vif.clk);
9      @(posedge vif.clk);
10     -> end_record;
11     @(posedge vif.clk); //hold time
12 endtask
13     //...code
14 endclass
15 //---------------------------------------
```

```
1  //--------------------**monitor.sv**---------
2      //...code
3  virtual task collect_transactions(uvm_phase phase);
4      wait(vif.rst === 1);
5      forever begin
6          -> begin_record;
7          tr.A = vif.A;
8          tr.B = vif.B;
```

```
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  endclass
```

```
1   //--------------------**monitor_out.sv**----
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           -> begin_record;
7           tr.data = vif.data;
8           tr.co = vif.co;             item_collected_port.write(tr);
9           @(posedge vif.clk);
10          -> end_record;
11      end
12  endtask
13      //...code
14  endclass
```

```
1   //--------------------**refmod.sv**----------
2   import "DPI-C" context function int rca_sum(int x,int y,int carry_in);
3   import "DPI-C" context function int rca_carry(int x, int y, int
        carry_in);
4       //...code
5   virtual task run_phase(uvm_phase phase);
6       super.run_phase(phase);
7       forever begin
8           in.get(tr_in);
9           tr_out.data = rca_sum(tr_in.A, tr_in.B, tr_in.ci);
10          tr_out.co = rca_carry(tr_in.A, tr_in.B, tr_in.ci);
11          out.put(tr_out);
12      end
13  endtask: run_phase
14  //-----------------------------------------
15  endclass: refmod
```

```
1   //--------------------**top.sv**-------------
2   import uvm_pkg::*;
3   `include "uvm_macros.svh"
4   `include "./input_if.sv"
5   `include "./output_if.sv"
6   `include "./packet_in.sv"
7   `include "./packet_out.sv"
8   `include "./sequence_in.sv"
9   `include "./sequencer.sv"
10  `include "./driver.sv"
11  `include "./driver_out.sv"
```

```
`include "./monitor.sv"
`include "./monitor_out.sv"
`include "./agent.sv"
`include "./agent_out.sv"
`include "./refmod.sv"
`include "./comparator.sv"
`include "./env.sv"
`include "./simple_test.sv"
    //...code
rca_generic sum(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .ci(in.ci)
    ,.s(out.data), .co(out.co));
    //...code
endmodule
```

```
//--------------------**external.cpp**-------
#include <stdio.h>
//-----------------------------------------
int getBit(int a, int i)
{
return ((a & (1 << i)) >> i);
}
//-----------------------------------------
int sum(int carry_in,int a,int b)
{
return ((carry_in ^ a) ^ b);
}
//-----------------------------------------
int carry(int carry_in,int a,int b)
{ int d;
d=(carry_in & a) | (a & b) | (carry_in & b) ;
return d;
}
//-----------------------------------------
int setBit(int result, int i, int s)
{
    if (s == 1)
        return result | (1 << i);
return result & ~(1 << i);
}
//-----------------------------------------
extern "C" int rca_sum(int x, int y , int carry_in){
int a,b,i , result=0;
for (i = 0; i < 4; i++) // probably can't use the increment op
{
        a = getBit(x, i);
        b = getBit(y, i);
        int s = sum(carry_in, a, b);
         int carry_out = carry(carry_in, a, b);
        result = setBit(result, i, s);
        carry_in = carry_out;}
return result;
}
```

```
//-----------------------------------------
extern "C" int rca_carry(int x, int y , int carry_in){
int a,b,i,result=0 ;
for (i = 0; i < 4; i++) // probably can't use the increment op
{
        a = getBit(x, i);
        b = getBit(y, i);
        int s = sum(carry_in, a, b);
        int carry_out = carry(carry_in, a, b);
        result = setBit(result, i, s);          carry_in = carry_out;}
return carry_in;
}
```

Multiplexer

```
//--------------------**mux21.sv**-----------
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX21 is
        Generic ( N     : integer := 31);
         Port    ( clk   : in  std_logic;
                   reset : 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;
                   Y     : Out std_logic_vector(N-1 downto 0));
end entity;
architecture behavioral of MUX21 is
BEGIN
         Y <= A when S='0' else B;
end behavioral;
```

```
//------------------**driver.sv**------------
     //....code
virtual protected task drive_transfer(packet_in tr);
    @(posedge vif.clk);
    vif.A = tr.A;
    vif.B = tr.B;
    vif.s = tr.s;
    @(posedge vif.clk);
    -> end_record;
    @(posedge vif.clk); //hold time
    @(posedge vif.clk);

endtask
     //...code
//-----------------------------------------
endclass
//---------------------------------------
```

```
//--------------------**monitor.sv**---------
    //....code
virtual task collect_transactions(uvm_phase phase);
```

```
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.s = vif.s;              item_collected_port.write(tr);
10          @(posedge vif.clk);
11          -> end_record;
12      end
13  endtask
14      //....code
15  //-----------------------------------------
16  endclass
17  //---------------------------------------
```

```
1   //------------------**monitor_out.sv**-------
2       //... code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           -> begin_record;
7           tr.y = vif.y;            item_collected_port.write(tr);
8           @(posedge vif.clk);
9           -> end_record;
10      end
11  endtask
12     //....code
13  //-----------------------------------------
14  endclass
15  //---------------------------------------
```

```
1   //--------------------**refmod.sv**----------
2   import "DPI-C" context function int mux(int a, int b, int sel);
3       //...code
4   virtual task run_phase(uvm_phase phase);
5       super.run_phase(phase);
6       forever begin
7           in.get(tr_in);
8           tr_out.y = mux(tr_in.A, tr_in.B, tr_in.s);              out.put(
                tr_out);
9       end
10  endtask: run_phase
11  //-----------------------------------------
12  endclass: refmod
```

```
1   //--------------------**top.sv**-------------
2     //... code
3   mux21 mux(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .s(in.s),.y(out.
        y));
4     //... code
5   //-----------------------------------------
```

```
//--------------------**external.cpp**-------
#include <stdio.h>
//-----------------------------------------
extern "C" int mux(int a, int b, int sel){
int y;
if (sel == 0){
   y=a;}
else y = b;
return y;
}
```

Carry Select Adder

```
//--------------------**csa.vhdl**-----------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity csa is
        Generic (N    : integer := 32);
        Port (  clk   : in std_logic;
                reset : in  std_logic;
                A     : In  std_logic_vector(N-1 downto 0);
                B     : In  std_logic_vector(N-1 downto 0);
                Cin   : In  std_logic;
                Si    : Out std_logic_vector(N-1 downto 0));
end ENTITY;
architecture STRUCTURAL of csa is
signal STMP0          : std_logic_vector(N-1 downto 0);
signal STMP1          : std_logic_vector(N-1 downto 0);
signal STMP2          : std_logic_vector(N-1 downto 0);
signal Carry0,Carry1  : std_logic;
signal valid1         : std_logic;
component RCA_generic is
        Generic (   N : integer := 32);
        Port    ( clk  : in  std_logic;
                reset : in  std_logic;
                A     : In  std_logic_vector(N-1 downto 0);
                B     : In  std_logic_vector(N-1 downto 0);
                Ci    : In  std_logic;
                S     : Out std_logic_vector(N-1 downto 0);
                Co    : Out std_logic);
end component;
 component MUX21 is
        Generic ( N     : integer := 32);
         Port   ( clk   : in  std_logic;
                  reset : 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;
                  Y     : Out std_logic_vector(N-1 downto 0));
        end component;
begin
CSA0:  RCA_generic
```

```
41            generic map (N=> N)
42            Port Map (clk =>clk, reset =>reset, a =>a, b=>b, ci => '0',
                  s =>STMP0,co=>Carry0);
43  CSA1:  RCA_generic
44            generic map (N=> N)
45            Port Map (clk =>clk, reset =>reset,a =>a, b => b,ci =>'1', S
                   => STMP1,co=>Carry1);
46  mux:  MUX21
47            generic map (N=> N)
48            port map(clk =>clk, reset =>reset,A => stmp0,B => stmp1,s=>
                  Cin,  y=>stmp2);
49  si <= stmp2;
50  end STRUCTURAL;
```

```
1  //--------------------**driver.sv**----------
2      //....code
3  virtual protected task drive_transfer(packet_in tr);
4          @(posedge vif.clk);
5          vif.A = tr.A;
6          vif.B = tr.B;
7          vif.ci = tr.ci;
8          @(posedge vif.clk);
9          -> end_record;
10         @(posedge vif.clk); //hold time
11         @(posedge vif.clk); //hold time
12         @(posedge vif.clk); //hold time
13         @(posedge vif.clk); //hold time
14 endtask
15     //....code
16 //-----------------------------------------
17 endclass
18 //---------------------------------------
```

```
1  //--------------------**monitor.sv**---------
2      //...code
3  virtual task collect_transactions (uvm_phase phase);
4      wait(vif.rst === 1);
5          forever begin
6              -> begin_record;
7              tr.A = vif.A;
8              tr.B = vif.B;
9              tr.ci = vif.ci;
10             item_collected_port.write(tr);
11             @(posedge vif.clk);
12             -> end_record;
13         end
14 endtask
15     //...code
16 //-----------------------------------------
17 endclass
18 //---------------------------------------
```

```
//--------------------**monitor_out.sv**-----
    //.....code
virtual task collect_transactions(uvm_phase phase);
    wait(vif.rst === 1);
    forever begin
        -> begin_record;
        tr.A = vif.A;
        tr.B = vif.B;
        tr.ci = vif.ci;
        item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
    //...code
//-----------------------------------------
endclass
//---------------------------------------
```

```
//--------------------**refmod.sv**----------
import "DPI-C" context function int csa(int a, int b, int ci);
    //...code
virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        in.get(tr_in);
        tr_out.data = csa(tr_in.A, tr_in.B, tr_in.ci);
        out.put(tr_out);
    end
endtask: run_phase
//-----------------------------------------
endclass: refmod
```

```
//--------------------**top.sv**-------------
    //...code
csa sum(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .cin(in.ci),.si(
    out.data));
    //...code
```

```
//--------------------**external.cpp---------
#include <stdio.h>
#include <iostream>
using namespace std;
#include <cmath>
#include <stdlib.h>
int getBit(int a, int i)
{
        return ((a & (1 << i)) >> i);
}
//-----------------------------------------
int sum(int carry_in,int a,int b)
{
```

```
14          return ((carry_in ^ a) ^ b);
15  }
16  //-----------------------------------------
17  int carry(int carry_in,int a,int b)
18  { int d;
19  d=(carry_in & a) | (a & b) | (carry_in & b) ;
20          return d;
21  }
22  //-----------------------------------------
23  int setBit(int result, int i, int s)
24  {
25          if (s == 1)
26                  return result | (1 << i);
27          return result & ~(1 << i);
28  }
29  //-----------------------------------------
30  int rca_sum(int x, int y , int carry_in){
31  int a,b,i , result=0;
32  for (i = 0; i < 4; i++) // probably can't use the increment op
33          {
34                  a = getBit(x, i);
35                  b = getBit(y, i);
36                  int s = sum(carry_in, a, b);
37                   int carry_out = carry(carry_in, a, b);
38                  result = setBit(result, i, s);
39                  carry_in = carry_out;}
40  return result;
41          }
42  //-----------------------------------------
43  int rca_carry(int x, int y , int carry_in){
44  int a,b,i, result=0 ;
45  for (i = 0; i < 4; i++) // probably can't use the increment op
46          {
47                  a = getBit(x, i);
48                  b = getBit(y, i);
49                  int s = sum(carry_in, a, b);
50                   int carry_out = carry(carry_in, a, b);
51                  result = setBit(result, i, s);
52                  carry_in = carry_out;}
53  return carry_in;
54          }
55  //-----------------------------------------
56   int mux(int a, int b , int sel){
57  int y;
58   if (sel == 0){
59     y=a;}
60   else y = b;
61  return y;
62          }
63  //-----------------------------------------
64  extern "C" int csa(int a, int b , int ci){
```

```
int si;
int sum1, sum0;
int carry0,carry1;
int mux_out;
sum0   =   rca_sum   (a, b, 0);
carry0 =   rca_carry (a, b, 0);
sum1   =   rca_sum   (a, b, 1);
carry1 =   rca_carry (a, b, 1);
mux_out=   mux       (sum0,sum1,ci);
cout <<mux_out;
return mux_out;
}
```

Sum Generator

```
//--------------------**sum_generator.vhdl**-
 library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity CSA_generic is
        Generic (N: integer := 32);
        Port (
                clk   : in  std_logic;
                reset : in  std_logic;
                A     : In  std_logic_vector(N-1 downto 0);
                B     : In  std_logic_vector(N-1 downto 0);
                Cin   : In  std_logic_vector(N/4-1 downto 0);
                Si    : Out std_logic_vector(N-1 downto 0));
end ENTITY;
architecture STRUCTURAL of CSA_generic is
  signal STMP0 : std_logic_vector(N-1 downto 0);
  signal STMP1 : std_logic_vector(N-1 downto 0);
component csa is
        Generic (N    : integer := 32);
        Port (  clk   : in std_logic;
                reset : in  std_logic;
                A     : In  std_logic_vector(N-1 downto 0);
                B     : In  std_logic_vector(N-1 downto 0);
                Cin   : In  std_logic;
                Si    : Out std_logic_vector(N-1 downto 0));
end component;
begin
SUMGEN1: for i in 1 to N/4 generate
         CSi :  csa
         generic map (N => 4)
        port map (clk,reset,a(4*i-1 downto 4*(i-1)), b(4*i-1 downto 4*
            (i-1)), Cin(i-1), si(4*i-1 downto 4*(i-1)));
        end generate;
end STRUCTURAL;
```

```
//------------------**driver.sv**-----------
    //....code
virtual protected task drive_transfer(packet_in tr);
```

```
4       @(posedge vif.clk);
5       vif.A = tr.A;
6       vif.B = tr.B;
7       vif.ci = tr.ci;
8       @(posedge vif.clk);
9       @(posedge vif.clk);
10      @(posedge vif.clk);
11      @(posedge vif.clk);
12      -> end_record;
13      @(posedge vif.clk); //hold time
14      endtask
15      //....code
16  //-----------------------------------------
17  endclass
18  //---------------------------------------
```

```
1   //--------------------**monitor.sv**---------
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  endclass
17  //---------------------------------------
```

```
1   //--------------------** monitor_out.sv**----
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           -> begin_record;
7           tr.data = vif.data;
8           item_collected_port.write(tr);
9           @(posedge vif.clk);
10          -> end_record;
11      end
12  endtask
13      //...code
14  endclass
15  //---------------------------------------
```

```
1   //------------------**refmod.sv**------------
```

```
2  import "DPI-C" context function int sum_generator(int a, int b, int ci
       );
3      //...code
4  virtual task run_phase(uvm_phase phase);
5      super.run_phase(phase);
6      forever begin
7          in.get(tr_in);
8          tr_out.data = sum_generator(tr_in.A, tr_in.B, tr_in.ci);
9          out.put(tr_out);
10     end
11 endtask: run_phase
12 endclass: refmod
13 //---------------------------------------
```

```
1  //------------------**top.sv**--------------
2      \\ ....code
3  CSA_generic sum(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .cin(in.ci
       ),.si(out.data));
4      \\..code.
```

```
1  //--------------------**external.cpp**-------
2  #include <stdio.h>
3  #include <iostream>
4  using namespace std;
5  #include <cmath>
6  #include <stdlib.h>
7  int getBit(int a, int i)
8  {
9          return ((a & (1 << i)) >> i);
10 }
11 //-----------------------------------------
12 int sum(int carry_in,int a,int b)
13 {
14         return ((carry_in ^ a) ^ b);
15 }
16 //-----------------------------------------
17 int carry(int carry_in,int a,int b)
18 { int d;
19 d=(carry_in & a) | (a & b) | (carry_in & b) ;
20         return d;
21 }
22 //-----------------------------------------
23 int setBit(int result, int i, int s)
24 {
25 if (s == 1)
26         return result | (1 << i);
27         return result & ~(1 << i);
28 }
29 //-----------------------------------------
30 int rca_sum(int x, int y , int carry_in){
31 int a,b,i , result=0;
32 for (i = 0; i < 4; i++)
```

```
{
        a = getBit(x, i);
        b = getBit(y, i);
        int s = sum(carry_in, a, b);
        int carry_out = carry(carry_in, a, b);
        result = setBit(result, i, s);
        carry_in = carry_out;}
return result;
}
//-----------------------------------------
int rca_carry(int x, int y , int carry_in){
int a,b,i ;
for (i = 0; i < 4; i++)
{
        a = getBit(x, i);
        b = getBit(y, i);
        int s = sum(carry_in, a, b);
         int carry_out = carry(carry_in, a, b);
        int     result = setBit(result, i, s);
        carry_in = carry_out;}
return carry_in;
}
//-----------------------------------------
int mux(int a, int b , int sel){
int y;
if (sel == 0){
   y=a;}
else y = b;
return y;
}
//-----------------------------------------
int csa(int a, int b , int ci){
int si;
int sum1, sum0;
int carry0,carry1;
int mux_out;
sum0   =   rca_sum   (a, b, 0);
carry0 =   rca_carry (a, b, 0);
sum1   =   rca_sum   (a, b, 1);
carry1 =   rca_carry (a, b, 1);
mux_out=   mux       (sum0,sum1,ci);
return mux_out;
}
//-----------------------------------------
extern "C" int sum_generator(int a, int b, int ci){
int si;
int sum_gen;
sum_gen= csa(a, b, ci);
return sum_gen;

}
```

G block

```
//---------------------**g.vhdl**-------------
library ieee;
use ieee.std_logic_1164.all;
entity G is
        port (  clk   : in std_logic;
                reset : in std_logic;
                G_ik  : in  std_logic;
                P_ik  : in  std_logic;
                G_k_j : in  std_logic;  -- G_k-1_j
                G_ij  : out std_logic);
end G;
architecture Behav of G is
begin
            G_ij <= G_ik or (P_ik and G_k_j);
end Behav;
```

```
//---------------------**driver.sv**----------
    //...code
virtual protected task drive_transfer(packet_in tr);
    @(posedge vif.clk)
    vif.G_ik = tr.G_ik;
    vif.P_ik = tr.P_ik;
    vif.G_k_j = tr.G_k_j;
    @(posedge vif.clk);
    -> end_record;
    @(posedge vif.clk); //hold time
endtask
    //...code
endclass
```

```
//---------------------**monitor.sv**---------
   //...code
virtual task collect_transactions(uvm_phase phase);
    wait(vif.rst === 1);
    forever begin
        -> begin_record;
        tr.G_ik = vif.G_ik;
        tr.P_ik = vif.P_ik;
        tr.G_k_j = vif.G_k_j;
        item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
    //....code
```

```
//---------------------**monitor_out.sv-------
    //...code
virtual task collect_transactions(uvm_phase phase);
    @(negedge vif.rst);
    forever begin
```

```
6           -> begin_record;
7           tr.G_ij = vif.G_ij;
8           item_collected_port.write(tr);
9           @(posedge vif.clk);
10          -> end_record;
11      end
12  endtask
13      //...code
14  //-----------------------------------------
15  endclass
```

```
1   //--------------------**refmod.sv**----------
2   import "DPI-C" context function int g(int a, int b, int c);
3       //...code
4   virtual task run_phase(uvm_phase phase);
5       super.run_phase(phase);
6       forever begin
7           in.get(tr_in);
8           tr_out.G_ij = g(tr_in.G_ik, tr_in.P_ik, tr_in.G_k_j);
9           out.put(tr_out);
10      end
11  endtask: run_phase
12  //-----------------------------------------
13  endclass: refmod
14  //---------------------------------------
```

```
1   //------------------**top.sv**---------------
2       //....code
3   g sum(.clk(clk), .reset(rst), .G_ik(in.G_ik), .P_ik(in.P_ik), .G_k_j(
        in.G_k_j),.G_ij(out.G_ij));
4       //...code
```

```
1   //--------------------**external.cpp**-------
2   #include <stdio.h>
3   extern "C" int g(int a, int b , int c){
4   int d;
5   d = a or (b and c);
6   return d;
7   }
```

P block

```
1   //--------------------**P.vhdl**-------------
2   library ieee;
3   use ieee.std_logic_1164.all;
4   entity P is
5           port (
6               clk   : in  std_logic;
7                   reset : in  std_logic;
8                   P_ik  : in  std_logic;
9                   P_k_j : in  std_logic;  --P_k-1_j
10                  P_ij  : out std_logic);
11  end P;
```

```
12 architecture Behavioral of P is
13 begin
14                 P_ij <= P_ik and P_k_j;
15 end Behavioral;
```

```
1  //--------------------**driver.sv**----------
2      //....code
3  virtual protected task drive_transfer(packet_in tr);
4      @(posedge vif.clk)
5      vif.P_ik = tr.P_ik;
6      vif.P_k_j = tr.P_k_j;
7      @(posedge vif.clk);
8      -> end_record;
9      @(posedge vif.clk); //hold time
10 endtask
11     //...code
12 //-----------------------------------------
```

```
1  //------------------**monitor.sv**-----------
2      //....code
3  virtual task collect_transactions(uvm_phase phase);
4      wait(vif.rst === 1);
5      forever begin
6          -> begin_record;
7          tr.P_ik = vif.P_ik;
8          tr.P_k_j = vif.P_k_j;
9          item_collected_port.write(tr);
10         @(posedge vif.clk);
11         -> end_record;
12     end
13 endtask
14     // ...code
15 //-----------------------------------------
```

```
1  //------------------** monitor_out.sv**------
2      //....code
3  virtual task collect_transactions(uvm_phase phase);
4      @(negedge vif.rst);
5      forever begin
6          @(posedge vif.clk);
7          -> begin_record;
8          tr.P_ij = vif.P_ij;
9          item_collected_port.write(tr);
10         @(posedge vif.clk);
11         -> end_record;
12     end
13 endtask
14     //...code
15 //-----------------------------------------
```

```
1  //--------------------**refmod.sv**----------
2   import "DPI-C" context function int p(int a, int b );
```

```
3       //...code
4   virtual task run_phase(uvm_phase phase);
5       super.run_phase(phase);
6       forever begin
7           in.get(tr_in);
8           tr_out.P_ij = p(tr_in.P_ik, tr_in.P_k_j);
9           out.put(tr_out);
10      end
11  endtask: run_phase
12  //-----------------------------------------
13  endclass: refmod
```

```
1   //--------------------**top.sv**-------------
2       //...code
3   p P(.clk(clk), .reset(rst), .P_ik(in.P_ik), .P_k_j(in.P_k_j),.P_ij(out
        .P_ij));
4       //...code
5   //-----------------------------------------
```

```
1   //--------------------**external.cpp**-------
2    #include <stdio.h>
3   extern "C" int p(int a, int b ){
4   int d;
5   d = a and b 5
6   return d;
7   }
```

PG_BLOCK

```
1   //--------------------**pg_block.vhdl--------
2   library ieee;
3   use ieee.std_logic_1164.all;
4   entity pg is
5           port (
6               clk     : in std_logic;
7                   reset   : in std_logic;
8                   Gik     : in  std_logic;  --a
9                   Pik     : in  std_logic;  --b
10                  G_k_j   : in  std_logic;  --c    -- G_k-1_j
11                  P_k_j   : in  std_logic;  --m
12                  G_ij    : out std_logic;  --d
13                  P_ij    : out std_logic); --v
14  end pg;
15  architecture Behav of pg is
16  begin
17          G_ij <= Gik OR (G_k_j and Pik) ;
18          p_ij <= pik and p_k_j;
19  end Behav;
```

```
1   //--------------------**driver.sv**----------
2       //...code
3   virtual protected task drive_transfer(packet_in tr);
4       @(posedge vif.clk);
```

```
    vif.Gik = tr.Gik;
    vif.Pik = tr.Pik;
    vif.G_k_j = tr.G_k_j;
    vif.P_k_j = tr.P_k_j;
    @(posedge vif.clk);
    -> end_record;
    @(posedge vif.clk); //hold time
endtask
    //...code
//-----------------------------------------
```

```
//-------------------**monitor.sv**----------
    //...code
virtual task collect_transactions(uvm_phase phase);
    wait(vif.rst === 1);
    forever begin
        -> begin_record;
        tr.Gik = vif.Gik;
        tr.Pik = vif.Pik;
        tr.G_k_j = vif.G_k_j;
        tr.P_k_j = vif.P_k_j;
        item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
    //...code
//-----------------------------------------
```

```
//--------------------**monitor_out.sv**-----
    //....code
virtual task collect_transactions(uvm_phase phase);
    @(negedge vif.rst);
    forever begin
        -> begin_record;
        tr.G_ij = vif.G_ij;
        tr.P_ij = vif.P_ij;
        item_collected_port.write(tr);
        @(posedge vif.clk);
        -> end_record;
    end
endtask
    //...code
```

```
//-------------------**refmod.sv**----------
import "DPI-C" context function int sum(int a, int b, int c , int m);
import "DPI-C" context function int sum1(int a, int b, int c , int m);
    //...code
virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        in.get(tr_in);
```

```
9           tr_out.G_ij = sum(tr_in.Gik, tr_in.Pik, tr_in.G_k_j, tr_in.
                P_k_j);
10          tr_out.P_ij = sum1(tr_in.Gik, tr_in.Pik, tr_in.G_k_j, tr_in.
                P_k_j);
11          out.put(tr_out);
12      end
13  endtask: run_phase
14  //-----------------------------------------
15  endclass: refmod
16  //---------------------------------------
```

```
1   //----------------------**top.sv**----------
2      //...code
3   pg m(.clk(clk), .reset(rst), .Gik(in.Gik), .Pik(in.Pik), .G_k_j(in.
        G_k_j), .P_k_j(in.P_k_j),.G_ij(out.G_ij), .P_ij(out.P_ij));
4       //...code
```

```
1   //-------------------**external.cpp**--------
2   #include <stdio.h>
3   extern "C" int sum(int a, int b, int c, int m){
4   int d;
5   d = a or c and b;
6    return d;
7   }
8   //-----------------------------------------
9   extern "C" int sum1(int a, int b , int c ,int m){
10  int v;
11  v= b and m ;
12  return v;
13  }
```

PG_NETWORK

```
1   //-----------------**pg_network.sv**---------
2   library ieee;
3   use ieee.std_logic_1164.all;
4   entity pg_network is
5           Generic (N: integer := 4);
6           port (  clk     : in  std_logic;
7                   reset   : in  std_logic;
8                   A       : in  std_logic_vector (N-1 downto 0);
9                   B       : in  std_logic_vector (N-1 downto 0);
10                  Cin     : in  std_logic;
11                  p       : out std_logic_vector (N-1 downto 0);
12                  g       : out std_logic_vector (N-1 downto 0);
13                  G_10    : out std_logic);
14  end pg_network;
15  architecture Behav of pg_network is
16  signal tmp_p, tmp_g : std_logic_vector(N-1 downto 0);
17  begin
18  pg_net: for i in 0 to N-1 generate
19          tmp_p(i) <= a(i) xor b(i);
20          tmp_g(i) <= a(i) and b(i);
```

```
21          G_10 <= tmp_g(0) or (tmp_p(0) and cin);
22   end generate;
23  p <= tmp_p;
24  g <= tmp_g;
25  end Behav;
```

```
1   //---------------------**driver.sv**---------
2     //...code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           -> begin_record;
7           tr.p = vif.p;
8           tr.g = vif.g;
9           tr.G_10 = vif.G_10;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  //-----------------------------------------
```

```
1   //---------------------**monitor.sv**----------
2       //....code.sv
3   virtual task collect_transactions(uvm_phase phase);
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  //-----------------------------------------
```

```
1   //---------------------**monitor_out.sv**------
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           @(posedge vif.clk);
7           -> begin_record;
8           tr.p = vif.p;
9           tr.g = vif.g;
10          tr.G_10 = vif.G_10;
11          item_collected_port.write(tr);
12          @(posedge vif.clk);
```

```
        -> end_record;
    end
endtask
    //...code
//-----------------------------------------
```

```
//----------------------**refmod.sv**--------
import "DPI-C" context function int sum(int a, int b );
import "DPI-C" context function int sum1(int a, int b );
import "DPI-C" context function int sum2(int a, int b, int ci );
    //...code
virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        in.get(tr_in);
        tr_out.p    = sum (tr_in.A, tr_in.B);
        tr_out.g    = sum1(tr_in.A, tr_in.B);
        tr_out.G_10 = sum2(tr_in.A, tr_in.B, tr_in.ci);
        out.put(tr_out);
    end
endtask: run_phase
//-----------------------------------------
endclass: refmod
//---------------------------------------
```

```
//-----------------**external.cpp**-----------
#include <stdio.h>
//-----------------------------------------
extern "C" int sum(int a, int b ){
int k;
k= a xor b;
return k;
}
//-----------------------------------------
extern "C" int sum1(int a, int b ){
int d;
d = a & b ;
return d;
}
//-----------------------------------------
extern "C" int sum2(int a, int b ,int ci){
int j;
j= sum1(a,b) | (sum(a,b) & ci);
return j;
}
```

```
//------------------**top.sv**---------------
    //...code
pg_network pg_net(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .cin(in.
    ci),.p(out.p), .g(out.g), .G_10(out.G_10));
   //...code
//-----------------------------------------
```

CARRY GENERATOR

```
1  //-------------**generic_carry_generator.sv**----
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  use IEEE.math_real.all;
6  use WORK.log2Funct.all;
7  entity CARRYGEN_GENERIC is
8          Generic ( N: integer := 4);
9           port   ( clk    : in std_logic;
10                   reset  : in std_logic;
11                   A      : in  std_logic_vector (N-1 downto 0);
12                   B      : in  std_logic_vector (N-1 downto 0);
13                   Cin    : in  std_logic;
14                   Co     : out std_logic_vector (N/4-1 downto 0));
15 end CARRYGEN_GENERIC;
16 architecture Struct_generatorcar of CARRYGEN_GENERIC is
17 component G is
18         port (  clk    : in std_logic;
19                 reset  : in std_logic;
20                 G_ik   : in  std_logic;
21                 P_ik   : in  std_logic;
22                 G_k_j  : in  std_logic;  -- G_k-1_j
23                 G_ij   : out std_logic);
24 end component;
25 component P is
26         port (  clk    : in  std_logic;
27                 reset  : in  std_logic;
28                 P_ik   : in  std_logic;
29                 P_k_j  : in  std_logic;  --P_k-1_j
30                 P_ij   : out std_logic);
31 end component;
32 component pg_network is
33         Generic (N: integer := 4);
34         port (  clk     : in  std_logic;
35                 reset   : in  std_logic;
36                 A       : in  std_logic_vector (N-1 downto 0);
37                 B       : in  std_logic_vector (N-1 downto 0);
38                 Cin     : in  std_logic;
39                 p       : out std_logic_vector (N-1 downto 0);
40                 g       : out std_logic_vector (N-1 downto 0);
41                 G_10    : out std_logic);
42 end component;
43 component  pg is
44         port (  clk     : in std_logic;
45                 reset   : in std_logic;
46                 Gik     : in  std_logic;  --a
47                 Pik     : in  std_logic;  --b
48                 G_k_j   : in  std_logic;  --c    -- G_k-1_j
49                 P_k_j   : in  std_logic;  --m
50                 G_ij    : out std_logic;  --d
```

```
51                  P_ij    : out std_logic); --v
52  end component;
53  type Matrix is array (N-1 downto 0) of std_logic_vector (N-1 downto 0)
        ;
54          signal GTree : Matrix;
55          signal PTree : Matrix;
56          signal p_n   : std_logic_vector(N-1 downto 0);
57          signal g_n   : std_logic_vector(N-1 downto 0);
58          signal G10   : std_logic;
59  begin
60  pgNETWORK : pg_NETWORK
61              generic map (N)
62              port map (clk,reset,a, b, Cin,p_n, g_n, G10);
63  righe : for riga in -2 to log2(N)-3 generate
64  riga0 : if riga=-2 generate
65  array0 : for I in 1 to N/2 generate
66  array01 : if I=1 generate
67  G_20 : G
68  port map(clk,reset,g_n(1),p_n(1),G10, GTree(1)(0));
69  end generate array01;
70  array0n : if I/=1 generate
71  P_ij : P
72  port map(clk,reset,p_n(I*2-1), p_n(I*2-2), PTree(I*2-1)(I*2-2));
73  G_ij : G
74  port map(clk,reset,g_n(I*2-1),p_n(I*2-1),g_n(I*2-2),GTree(I*2-1)(I*2-2
        ));
75  end generate array0n;
76  end generate array0;
77  end generate riga0;
78  riga1:  if riga=-1 generate
79  array1 : for I in 1 to N/4 generate
80  G_ij :  G
81  port map(clk,reset,GTree(I*4-1)(I*4-2),PTree(I*4-1)(I*4-2),GTree(I*4-3
        )(I*4-4),GTree(I*4-1)(I*4-4));
82  array1n : if I>1 generate
83  P_ij :  Pg
84  port map(clk,reset,GTree(I*4-1)(I*4-2), PTree(I*4-1)(I*4-2),GTree(I*4-
        3)(I*4-4), PTree(I*4-3)(I*4-4),GTree(I*4-1)(I*4-4),PTree(I*4-1)(I*
        4-4));
85  end generate array1n;
86  end generate array1;
87  end generate riga1;
88  ----------------------***Generates only the G blocks ***---------------
89  rigaxG: if (riga/=-2 and riga/=-1) generate
90  arrayxG: for I in 2**riga+1 to N/4 generate
91  IfxG:   if (I=2**riga+1) generate
92  ForxG:  for n in 0 to 2**riga-1 generate
93  --generate the G window (I=index of the window)
94  G_ij :  G
95  port map(clk,reset,GTree((I+n)*4-1)(2**riga*4), PTree((I+n)*4-1)(2**
        riga*4),GTree(2**riga*4-1)(0), GTree((I+n)*4-1)(0));
```

```
96  end generate ForxG;
97  end generate IfxG;
98  end generate arrayxG;
99  end generate rigaxG;
100 ------------------------*** Generates the PG blocks ***--------
101 rigaxPG:if (riga/=-2 and riga/=-1 and riga/=log2(N)-3) generate
102 arrayxPG: for I in 1 to N/4 generate
103 -- x is the windows number containing 2^rows PG
104 PG :    for x in (N/(2**(riga+3))-1) downto 1 generate --for x in
        Num_finestre_PG downto 1 (ex: 32bit -> {riga=0 x=3,riga=1 x=1})
105 IfxPG:  if (I=(2**riga+x*2**(riga+1)+1)) generate
106 ForxPG: for n in 0 to 2**riga-1 generate
107 G_ij :  G
108 port map(clk,reset,Gtree((I+n)*4-1) ((I-1)*4), Ptree((I+n)*4-1) ((I-1)
        *4), GTree((I-1)*4-1) ((I-1)*4-(2**(riga+2))), GTree((I+n)*4-1) ((
        I-1)*4-(2**(riga+2))));
109 P_ij :  P
110 port map(clk,reset,Ptree((I+n)*4-1) ((I-1)*4), PTree((I-1)*4-1) ((I-1)
        *4-(2**(riga+2))), PTree((I+n)*4-1) ((I-1)*4-(2**(riga+2))));
111 end generate ForxPG;
112 end generate IfxPG;
113 end generate PG;
114 end generate arrayxPG;
115 end generate rigaxPG;
116 end generate righe;
117 CoX:    for i in 1 to N/4 generate
118 Co(i-1) <= GTree(i*4-1) (0);
119 end generate CoX;
120 end Struct_generatorcar;
```

```
1  //----------------------**11.log2function.vhd**-
2  package log2Funct is
3          function log2( i : natural) return integer;
4  end log2Funct;
5  package body log2Funct is
6          function log2( i : natural) return integer is
7                  variable temp    : integer := i;
8                  variable ret_val : integer := 0;
9          begin
10                 while temp > 1 loop
11                         ret_val := ret_val + 1;
12                         temp    := temp / 2;
13                 end loop;
14                 return ret_val;
15         end log2;
16 end log2Funct;
```

```
1  //----------------------**driver.sv**----------
2      //...code
3  virtual protected task drive_transfer(packet_in tr);
4      vif.A = tr.A;
5      vif.B = tr.B;
```

```
6       vif.ci = tr.ci;
7       @(posedge vif.clk);
8       -> end_record;
9       @(posedge vif.clk); //hold time
10  endtask
11      //...code
12  //------------------------------------------
```

```
1   //--------------------**monitor.sv**----------
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  //------------------------------------------
```

```
1   //--------------------**monitor_out.sv**-----
2       //...code
3   virtual task collect_transactions(uvm_phase phase);
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  //------------------------------------------
```

```
1   //---------------------**refmod.sv**----------
2   import "DPI-C" context function int addBits( int num_1,  int num_2,
        int carry_out);
3       //...code
4   virtual task run_phase(uvm_phase phase);
5       super.run_phase(phase);
6       forever begin
7           in.get(tr_in);
8           tr_out.co    = addBits (tr_in.A, tr_in.B, tr_in.ci);
9           out.put(tr_out);
```

```
    end
endtask: run_phase
//------------------------------------------
endclass: refmod
```

```
//-------------------**external.cpp**--------
#include <iostream>
#include <stdio.h>
#define N_CSA 4
using namespace std;
//------------------------------------------
extern "C"  int addBits( int num_1,  int num_2,  int carry_out){
    int output = 0;
    int index = 0;
    int size = sizeof(unsigned int);
    int maxPow = 1;//<<(size*8-1);
    // = 0;
    bool first_time = true;
    for(int i=0;i<size*8;++i){
    // print last bit and shift left.
    int bit_num_1 = (num_1&maxPow ? 1 : 0);
    int bit_num_2 = (num_2&maxPow ? 1 : 0);
    int sum = bit_num_1 + bit_num_2 + carry_out;
    carry_out = ((sum & 1<<1)>>1 ? 1 : 0);
    if (((i+1) % N_CSA == 0) && (!first_time))
    {
          carry_out = carry_out << index++;
          output = output | carry_out;
    }
    num_1 = num_1>>1;
    num_2 = num_2>>1;
    first_time = false;
}
    //print_Bits(output);
    return output;
}
```

```
//------------------**top.sv**---------------
    //...code
carrygen_generic carrygen(.clk(clk), .reset(rst), .a(in.A), .b(in.B),
    .cin(in.ci),.co(out.co));
    //...code
//------------------------------------------
```

P4ADDER

```
//----------------**p4adder.sv**-------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity P4Adder is
        Generic (N: integer := 32);
         port   (
```

```
 8                   clk    : in  std_logic;
 9                   reset  : in  std_logic;
10                   A      : in  std_logic_vector (N-1 downto 0);
11                   B      : in  std_logic_vector (N-1 downto 0);
12                   Cin    : in  std_logic;
13                   SUM    : out std_logic_vector (N-1 downto 0);
14                   Co     : out std_logic);
15 end P4Adder;
16 architecture Struct_P4Adder of P4Adder is
17 signal cc   :  std_logic;
18 component CARRYGEN_GENERIC is
19 --this component used tfor generating carry tree
20         Generic ( N: integer := 32);
21          port   ( clk    : in std_logic;
22                   reset  : in std_logic;
23                   A      : in  std_logic_vector (N-1 downto 0);
24                   B      : in  std_logic_vector (N-1 downto 0);
25                   Cin    : in  std_logic;
26                   Co     : out std_logic_vector (N/4-1 downto 0));
27 end component;
28 component CSA_generic is
29 --  with carry generated in CARRYGEN_GENERIC,the sum can be done!
30         Generic (N: integer := 32);
31         Port (
32                  clk    : in  std_logic;
33                  reset  : in  std_logic;
34                  A      : In  std_logic_vector(N-1downto 0);
35                  B      : In  std_logic_vector(N-1 downto 0);
36                  Cin    : In  std_logic_vector(N/4-1 downto 0);
37                  Si     : Out std_logic_vector(N-1 downto 0));
38 end component;
39 component  register_generic
40         Generic (N: integer:=32);
41         Port (
42             DIN    : In std_logic_vector(N-1 downto 0) ;
43             Reset  : In std_logic;
44             clk    : In std_logic;
45             DOUT   : Out std_logic_vector(N-1 downto 0));
46 end component;
47 signal Cout   : std_logic_vector (N/4-1 downto 0);
48 signal SumCin : std_logic_vector (N/4-1 downto 0);
49 signal Ct   : std_logic;
50 begin
51 CARRYGEN  : CARRYGEN_GENERIC
52             generic map(N)
53             port map(clk,reset,a, b, Cin, Cout);
54             SumCin <= Cout(N/4-2 downto 0) & Cin;
55
56 SUMGEN    : CSA_generic
57             generic map (N)
58             port map (clk,reset,a, b, SumCin, sum);
```

```
59              co<= Cout(N/4-1);          -- final carry out
60  end Struct_P4Adder;
```

```
1   //------------------**driver.sv**------------
2       //...code
3   virtual protected task drive_transfer(packet_in tr);
4       @(posedge vif.clk);
5       vif.A = tr.A;
6       vif.B = tr.B;
7       vif.ci = tr.ci;
8       @(posedge vif.clk);
9       -> end_record;
10      @(posedge vif.clk); //hold time
11  endtask
12      //...code
13  //-----------------------------------------
```

```
1   //------------------**monitor.sv**-----------
2      //....code
3   virtual task collect_transactions(uvm_phase phase);
4       wait(vif.rst === 1);
5       forever begin
6           -> begin_record;
7           tr.A = vif.A;
8           tr.B = vif.B;
9           tr.ci = vif.ci;
10          item_collected_port.write(tr);
11          @(posedge vif.clk);
12          -> end_record;
13      end
14  endtask
15      //...code
16  //-----------------------------------------
```

```
1   //-----------------**monitor_out.sv**--------
2      //...code
3   virtual task collect_transactions(uvm_phase phase);
4       @(negedge vif.rst);
5       forever begin
6           -> begin_record;
7           tr.data = vif.data;
8           tr.co = vif.co;
9           item_collected_port.write(tr);
10          @(posedge vif.clk);
11          -> end_record;
12      end
13  endtask
14      //....code
15  //-----------------------------------------
```

```
1   //----------------**refmod.sv**--------------
2   import "DPI-C" context function int p4_sum(int a, int b, int ci);
```

```
3  import "DPI-C" context function int p4_carry(int a, int b, int ci);
4      //...code
5  virtual task run_phase(uvm_phase phase);
6      super.run_phase(phase);
7      forever begin
8          in.get(tr_in);
9          tr_out.data = p4_sum(tr_in.A, tr_in.B, tr_in.ci);
10         tr_out.co = p4_carry(tr_in.A, tr_in.B, tr_in.ci);
11         out.put(tr_out);
12     end
13 endtask: run_phase
14 //-----------------------------------------
15 endclass: refmod
```

```
1  //------------------**external.cpp**---------
2  #include <stdio.h>
3  #include <iostream>
4  using namespace std;
5  #include <cmath>
6  #include <stdlib.h>
7  #define N_CSA 4
8  //-----------------------------------------
9  int getBit(int a, int i)
10 {
11         return ((a & (1 << i)) >> i);
12 }
13 //-----------------------------------------
14 int sum(int carry_in,int a,int b)
15 {
16         return ((carry_in ^ a) ^ b);
17 }
18 //-----------------------------------------
19 int carry(int carry_in,int a,int b)
20 { int d;
21 d=(carry_in & a) | (a & b) | (carry_in & b) ;
22         return d;
23 }
24 //-----------------------------------------
25 int setBit(int result, int i, int s)
26 {
27         if (s == 1)
28                 return result | (1 << i);
29         return result & ~(1 << i);
30 }
31 //-----------------------------------------
32 int rca_sum(int x, int y , int carry_in){
33 int a,b,i , result=0;
34 for (i = 0; i < 4; i++)
35         {
36                 a = getBit(x, i);
37                 b = getBit(y, i);
38                 int s = sum(carry_in, a, b);
```

```
39                  int carry_out = carry(carry_in, a, b);
40                 result = setBit(result, i, s);
41                 carry_in = carry_out;}
42 return result;
43         }
44 //-----------------------------------------
45 int rca_carry(int x, int y , int carry_in){
46 int a,b,i ;
47 for (i = 0; i < 4; i++)
48         {
49                 a = getBit(x, i);
50                 b = getBit(y, i);
51         int s = sum(carry_in, a, b);
52                  int carry_out = carry(carry_in, a, b);
53         int     result = setBit(result, i, s);
54                 carry_in = carry_out;}
55 return carry_in;
56         }
57 //-----------------------------------------
58 int mux(int a, int b , int sel){
59 int y;
60  if (sel == 0){
61    y=a;}
62  else y = b;
63 return y;
64         }
65 //-----------------------------------------
66 int csa(int a, int b , int ci){
67 int si;
68 int sum1, sum0;
69 int carry0,carry1;
70 int mux_out;
71 sum0   =   rca_sum   (a, b, 0);
72 carry0 =   rca_carry (a, b, 0);
73 sum1   =   rca_sum   (a, b, 1);
74 carry1 =   rca_carry (a, b, 1);
75 mux_out=   mux       (sum0,sum1,ci);
76 //cout <<mux_out;
77 return mux_out;
78 }
79 //-----------------------------------------
80 int sum_generator(int a, int b , int ci){
81 int si;
82 int sum_gen;
83 sum_gen= csa(a, b, ci);
84 return sum_gen;
85 }
86 //-----------------------------------------
87 int pg1(int a, int b , int c , int m,int d, int v){
88 d = a |(c & b) ;
89   return d;
```

```
90  }
91  //-----------------------------------------
92  int pg2(int a, int b , int c , int m,int d, int v){
93  v= b & m ;
94  return v;}
95  //-----------------------------------------
96  int pg_network1(int a, int b ,int p ){
97  p = a ^b ;
98  return p;
99  }
100 //-----------------------------------------
101 int pg_network2(int a, int b ,int g){
102 g = a & b;
103 return g;
104 }
105 //-----------------------------------------
106 int pg_network3(int a, int b , int c , int G_10){
107 G_10 = (a & b) | (( a ^ b) & c);
108 return G_10;
109 }
110 //-----------------------------------------
111 int P(int a, int b , int d ){
112 d = a & b ;
113 return d;
114 }
115 //-----------------------------------------
116 int G(int a, int b , int ci , int v){
117 v = a | (b & ci);
118 return v;
119 }
120 //-----------------------------------------
121 struct Point {
122   int  Gtree, Ptree;
123 };
124 //-----------------------------------------
125 int get_bits(int N ,int bits_wanted){
126 int k;
127 int bits;
128 for (k=0; k<bits_wanted;k++){
129     int mask=1<<k;
130     int masked_n = N& mask;
131     int thebit =masked_n>>k;
132     bits = thebit;
133 }
134 return bits;
135 }
136 //-----------------------------------------
137 int addBits( int num_1,  int num_2,  int carry_out){
138     int output = 0;
139     int index = 0;
140     int size = sizeof(unsigned int);
```

```
141     int maxPow = 1;//<<(size*8-1);
142     // = 0;
143     bool first_time = true;
144     for(int i=0;i<size*8;++i){
145     // print last bit and shift left.
146     int bit_num_1 = (num_1&maxPow ? 1 : 0);
147     int bit_num_2 = (num_2&maxPow ? 1 : 0);
148     int sum = bit_num_1 + bit_num_2 + carry_out;
149     carry_out = ((sum & 1<<1)>>1 ? 1 : 0);
150     if (((i+1) % N_CSA == 0) && (!first_time))
151     {
152           carry_out = carry_out << index++;
153           output = output | carry_out;
154     }
155     num_1 = num_1>>1;
156     num_2 = num_2>>1;
157     first_time = false;
158 }
159     //print_Bits(output);
160     return output;
161 }
162 //-----------------------------------------
163 extern "C" int p4_sum(int a, int b , int ci)
164 {   int sum;
165 sum = sum_generator(a,b,ci);
166 return sum;
167 }
168 //-----------------------------------------
169 extern "C" int p4_carry(int a, int b , int ci)
170 {   int carry;
171 carry = addBits(a,b,ci);
172 return carry;
173 }
```

```
1  //-----------------**top.sv**----------------
2       //...code
3  P4Adder sum(.clk(clk), .reset(rst), .a(in.A), .b(in.B), .cin(in.ci),.
       sum(out.data), .co(out.co));
4       //...code
5  //-----------------------------------------
```

Serial In Serial Out SISO

```
1  //--------------------**siso.vhdl**----------
2  library IEEE;
3  use IEEE.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  entity siso is
6          Generic (N: integer:=4);
7          Port (
8              din_siso :  In  std_logic;
9              Reset    :  In  std_logic;
10             clk      :  In  std_logic;
```

```
11              Dout_siso:  Out     std_logic);
12  end siso;
13  architecture BEH of siso is
14  signal tmp : std_logic_vector(N-1 downto 0);
15  begin
16  process(clk)
17  begin
18      if ReSeT='1' then
19         DOUT_siso <= '0';
20      ELSIF rising_edge(clk) THEN
21            for i in 0 to 2 loop
22              tmp(i+1) <= tmp(i);
23            end loop;
24            tmp(0) <= din_siso;
25      end if;
26  dout_siso <= tmp(3);
27  end process;
28  end BEH;
```

```
1  //-----------------------**input_if.sv**------
2  interface input_if(input clk, rst);
3      logic  A;
4      modport port(input clk, rst, A);
5  endinterface
```

```
1  //---------------------**output_if.sv**------
2  interface output_if(input clk, rst);
3      logic  data;
4      modport port(input clk, rst, output  data);
5  endinterface
```

```
1  //--------------------**packet_in.sv**-------
2  class packet_in extends uvm_sequence_item;
3      rand logic A;
4       logic clk;
5       logic rst;
6      `uvm_object_utils_begin(packet_in)
7          `uvm_field_int(A, UVM_ALL_ON|UVM_HEX)
8      `uvm_object_utils_end
9  //-----------------------------------------
10 function new(string name="packet_in");
11     super.new(name);
12 endfunction: new
13 //-----------------------------------------
14 endclass: packet_in
```

```
1  //---------------------**packet_out.sv**------
2  class packet_out extends uvm_sequence_item;
3      rand logic data;
4      `uvm_object_utils_begin(packet_out)
5          `uvm_field_int(data, UVM_ALL_ON|UVM_HEX)
6      `uvm_object_utils_end
```

```
7  //-----------------------------------------
8  function new(string name="packet_out");
9      super.new(name);
10 endfunction: new
11 //-----------------------------------------
12 endclass: packet_out
```

```
1  //----------------------**driver.sv**---------
2    //...code
3  virtual protected task drive_transfer(packet_in tr);
4      vif.A = tr.A;
5      @(posedge vif.clk); //hold time
6      @(posedge vif.clk); //hold time
7      -> end_record;
8      @(posedge vif.clk);
9  endtask
10     //...code
```

```
1  //----------------------**monitor.sv**-------
2      //...code
3  virtual task collect_transactions(uvm_phase phase);
4      wait(vif.rst === 1);
5      forever begin
6          -> begin_record;
7          tr.A = vif.A;            item_collected_port.write(tr);
8          @(posedge vif.clk);
9          -> end_record;
10     end
11 endtask
12     //...code
13 //-----------------------------------------
```

```
1  //--------------------**monitor_out.sv**-----
2      //...code
3  virtual task collect_transactions(uvm_phase phase);
4      @(negedge vif.rst);
5      forever begin
6          -> begin_record;
7          tr.data = vif.data;
8          item_collected_port.write(tr);
9          @(posedge vif.clk);
10         -> end_record;
11     end
12 endtask
13     //...code
14 //-----------------------------------------
```

```
1  //-----------------**refmod.sv**-------------
2  class refmod extends uvm_component;
3      `uvm_component_utils(refmod)
4      packet_in tr_in;
5      packet_out tr_out;
```

```
6           logic a,rst,clk,b;
7       uvm_get_port #(packet_in) in;
8       uvm_put_port #(packet_out) out;
9   //-----------------------------------------
10  function new(string name = "refmod", uvm_component parent);
11      super.new(name, parent);
12      in = new("in", this);
13      out = new("out", this);
14  endfunction
15  //-----------------------------------------
16  virtual function void build_phase(uvm_phase phase);
17      super.build_phase(phase);
18      tr_out = packet_out::type_id::create("tr_out", this);
19  endfunction: build_phase
20  //-----------------------------------------
21  virtual task run_phase(uvm_phase phase);
22      super.run_phase(phase);
23      forever begin
24          in.get(tr_in);
25          fork
26              if (tr_in.rst===1)
27                  sig <=0;
28              else
29                  @(posedge clk);
30                  sig = sig << 1;
31                  for(int i=0; i<2; i++) begin
32                      sig[i+1] <= sig[i];
33                  end
34                  sig[0] = tr_in.A;
35                  tr_out.data = sig[3];
36          join_none
37          out.put(tr_out);
38      end
39  endtask: run_phase
40  //-----------------------------------------
41  endclass
```

```
1   //---------------------**top.sv**------------
2       //...code
3   siso sum( .din_siso(in.A), .reset(rst),  .clk(clk), .dout_siso(out.
        data));
4       //...code
5   //-----------------------------------------
```

ALU

```
1   //--------------------**Alu.vhdl**--
2   library IEEE;
3   use IEEE.std_logic_1164.all;
4   use IEEE.std_logic_arith.all;
5   use WORK.GLOBALS.all;
6   entity ALU is
7     generic (N : integer:=4);
```

```
 8    port    (
 9          clk              : in  std_logic;
10          reset            : in  std_logic;
11          FUNC             : IN ALUOP;
12          DATA1, DATA2     : IN std_logic_vector(N-1 downto 0);
13          OUTALU           : OUT std_logic_vector(N-1 downto 0));
14  end ALU;
15  architecture BEHAVIOR of ALU is
16  COMPONENT P4Adder is
17  Generic (N: integer := 4);
18           port    (
19              clk          : in  std_logic;
20              reset        : in  std_logic;
21              A            : in  std_logic_vector (N-1 downto 0);
22              B            : in  std_logic_vector (N-1 downto 0);
23              Cin          : in  std_logic;
24              SUM          : out std_logic_vector (N-1 downto 0);
25              Co           : out std_logic);
26  end COMPONENT;
27  component siso is
28          Generic (N: integer:=4);
29          Port (
30               din_siso    :        In       std_logic;
31               Reset       :        In       std_logic;
32               clk         :        In       std_logic;
33               Dout_siso   :        Out      std_logic);
34  end component;
35  SIGNAL CARRY_IN  : STD_LOGIC;
36  SIGNAL out_shft1 : STD_LOGIC;
37  SIGNAL ADD_IN2, OUT_SUM, OUT_shft,  OUT_nop: STD_LOGIC_VECTOR(3 downto
        0);
38  begin
39  CARRY_IN <= '1'         WHEN FUNC = F_SUB ELSE '0'    ;
40  ADD_IN2  <= NOT(DATA2) WHEN FUNC = F_SUB ELSE  DATA2;
41  out_shft <=  "000"& out_shft1;
42  ADD : P4ADDER
43  generic MAP (N =>4)
44  Port MAP( clk, reset , DATA1, ADD_IN2, CARRY_IN, OUT_SUM);
45  shift: siso
46  generic  map(N =>4)
47  port map ( data1(0), reset, clk, out_shft1  );
48  out_nop <= "000"&"0";
49  OUTALU <= OUT_SUM  WHEN FUNC = F_ADD  ELSE
50            OUT_nop  WHEN FUNC = nop  ELSE
51            OUT_SUM  WHEN FUNC = F_SUB  ELSE
52            OUT_SHFT WHEN FUNC = F_SHFT ELSE
53            (OTHERS=> '0');
54  end BEHAVIOR;
```

CONTROL-UNIT-HW

```
1  //---------------------**cu_hw.sv**---
2  library ieee;
```

```
3  use ieee.std_logic_1164.all;
4  use ieee.std_logic_unsigned.all;
5  use ieee.std_logic_arith.all;
6  use work.GLOBALS.all;
7
8  entity CONTROL_UNIT is
9    generic (
10         MICROCODE_MEM_SIZE :     integer := 4;  -- Microcode Memory
               Size
11         FUNC_SIZE          :     integer := 3;  -- Func Field Size for
                R-Type Ops
12         OP_CODE_SIZE       :     integer := 3;   -- Op Code Size
13         IR_SIZE            :     integer := 12;   -- Instruction
               Register Size
14         CW1_SIZE            :     integer := 4
15          ); -- Control Word Size
16   port (
17         Clk                : in  std_logic;  -- Clock
18         Rst                : in  std_logic;  -- Reset:Active-Low
19         IR_IN              : in  std_logic_vector(IR_SIZE - 1 downto 0
               );
20         ALU_OPCODE         : out aluOp      -- implicit coding
21     );
22 end CONTROL_UNIT;
23 architecture dlx_cu_hw of CONTROL_UNIT is
24   type mem_array is array (integer range 0 to MICROCODE_MEM_SIZE - 1)
       of std_logic_vector(CW1_SIZE - 1 downto 0);
25   CONSTANT cw_mem : mem_array := (
26                                 "0000",
27                                 "0001",
28                                 "0010",
29                                 "0011"
30                                  );
31   signal IR_sig : std_logic_vector(func_size -1 downto 0);
32   signal IR_func : std_logic_vector(FUNC_SIZE -1  downto 0);
33   signal aluOpcode_i: aluOp ;
34   signal aluOpcode1: aluOp ;
35 begin
36   IR_sig <= IR_IN( 10 downto 8);
37  -- IR_opcode <= "010";
38   IR_func  <= IR_IN(5 downto 3);
39 CW_PIPE: process (Clk, Rst)
40          begin
41              if Rst = '1' then
42                aluOpcode1 <= nop;
43              elsif rising_edge(clk) then
44                aluOpcode1 <= aluOpcode_i;
45              end if;
46 end process CW_PIPE;
47 ALU_OPCODE <= aluOpcode1;
48 ALU_OP_CODE_P : process (ir_sig,ir_func)
```

```
49                  begin
50                  case conv_integer(unsigned( IR_func)) is
51                      when TYPE_ADDI => aluOpcode_i <= F_ADD;
52
53                      when TYPE_SUBI => aluOpcode_i <= F_SUB;
54                      when TYPE_SHFT => aluOpcode_i <= F_SHFT;
55
56                      when TYPE_NOP  => aluOpcode_i <= NOP;

57                          when others    => aluOpcode_i <= NOP;
58                  end case;
59  end process ALU_OP_CODE_P;
60  end dlx_cu_hw;
```

DATA

```
1   //-------------**data.sv**-----------
2   library ieee ;
3   use ieee.std_logic_1164.all;
4   use ieee.numeric_std.all;
5   use WORK.GLOBALS.all;
6   entity Data is
7     port (
8           Clk       : in std_logic;
9           Rst       : in std_logic;
10          IRam_DOut : IN std_logic_vector(11 downto 0);
11          alu_out   : OUT std_logic_vector(3 downto 0)
12          );
13  end data;
14  architecture data_r of Data is
15  COMPONENT alu
16    generic (N : integer:=4);
17    port   (
18        clk  : in  std_logic;
19        reset: in  std_logic;
20        FUNC : IN ALUOP;
21        DATA1, DATA2: IN std_logic_vector(3 downto 0);
22        OUTALU: OUT std_logic_vector(3 downto 0));
23  end component;
24  component CONTROL_UNIT
25   generic (
26          MICROCODE_MEM_SIZE :     integer := 4;
27          OP_CODE_SIZE       :     integer := 3;
28          IR_SIZE            :     integer := 12;
29          CW1_SIZE            :     integer := 4
30          );
31    port (
32          Clk                : in  std_logic;
33          Rst                : in  std_logic;
34          IR_IN              : in  std_logic_vector(IR_SIZE - 1 downto 0
                );
35          ALU_OPCODE         : out aluOp
36      );
```

```
end component;
  signal ALU_OPCODE_i : aluOp;
  signal siga      :  std_logic_vector(3 downto 0);
  signal sigb      :  std_logic_vector(3 downto 0);
  begin
 siga <= IRam_dout (7 DOWNTO 4);
 sigb <= IRam_dout (3 DOWNTO 0);
CU_I: CONTROL_UNIT
       port map (Clk, Rst, IRam_DOut, ALU_OPCODE_I );
DP1: alu
     port MAP( Clk, Rst, ALU_OPCODE_I , siga, sigb, alu_out );
end data_r;
```

```
//--------------------**driver.sv**---------------
virtual protected task drive_transfer(packet_in tr);
    @(posedge vif.clk)
    vif.iram = tr.iram;
    @(posedge vif.clk);
    @(posedge vif.clk);
    @(posedge vif.clk);
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    @(posedge vif.clk); //hold time
    -> end_record;
    @(posedge vif.clk); //hold time
endtask
```

```
//--------------------**refmod**-----
    //...code
virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
        in.get(tr_in);
        fork
ir_opcode =tr_in.iram [5:3];
a    =  tr_in.iram [7:4];
b  =  tr_in.iram [3:0];

wait(tr_in.rst === 0)begin

```

```
14  case (ir_opcode)
15  0: funci = 0;
16  1 : funci = a+b ;
17  2:  funci = a-b ;
18  3:  funci = a[0];
19  default    : funci =0;
20  endcase
21              tr_out.alu_out= funci;
22              end
23          join_none
24      out.put(tr_out);
25      end
26  endtask: run_phase
```

Bibliography

[1] https://www.mentor.com/training, "functional-verification-videos," *https://learn.sw.siemens.com/library/functional-verification/VyX0-NUrf/learning-paths/system-verilog-uvm.*

[2] S. Saurabh, H. Shah, and S. Singh, "Timing closure problem: Review of challenges at advanced process nodes and solutions„" *IETE Technical Review*, vol. Volume 36, 2019 - Issue 6, 2018.

[3] E. Tuncer, "Signal integrity a challenge in ic design," *EE Times*, June 23, 2003.

[4] X. LI and O. HAMMAMI, "Fast design productivity for embedded multiprocessor through multi-fpga emulation: The case of a 48-way multiprocessor with noc," *Desig-Reuse.*

[5] P. Rashinkar, P. Paterson, and L. Singh, *SYSTEM-ON-A-CHIP VERIFICATION Methodology and Techniques.* KLUWER ACADEMIC PUBLISHERS, 2002.

[6] M. Glasser, "Uvm: The next generation in verification methodology," February 4, 2011.

[7] "Accelera, universal verification methodology (uvm) 1.2 user's guide," *(Accelera, Universal Verification Methodology (UVM) 1.2 User's guide).*

[8] "Uvm: phasing: https://verificationacademy.com/cookbook/phasing,"

[9] A. U. Shah, Neel (Marana, "Universal verification methodology (uvm) register abstraction layer (ral) painter," 03/01/2018.

[10] A. S. M. Graphics, "From simulation to emulation – a fully reusable uvm framework," *verification academy.*

[11] D. Rich, "Parameterized classes, static members and the factory macros," *https://blogs.mentor.com/verificationhorizons/blog/2011/02/13/parameterized-classes-static-members-and-the-factory-macros/.*

[12] N. Campos, "A basic tutorial of uvm an introduction to functional verification," *https://sistenix.com/basic_uvm*, September 11, 2016.