ABSTRACT

Developing Industrial Strength UVM for Academic Research and Teaching

George M. Onwubuya, M.S.E.C.E.

Mentor: Keith E. Schubert, Ph.D.

The Universal Verification Methodology (UVM) has been getting attention from researchers and the functional verification community for a little over decade. Its flexibility, reusability and reliability features are suitable for the design verification of multifaceted chip systems thus making it attractive for the verification industry. Similarly researchers frequently explore and utilize UVM to enhance its verification capabilities of system-on-chip (SoC) and application specific integrated circuits (ASIC). For a long time UVM learning and training has been tailored to suit the needs of seasoned verification engineers. Recent books have sought to address the needs of novice verification engineers, however UVM testbenches lack the standard required by the verification industry. This thesis outlines steps required in building a typical UVM testbench while also highlighting the important industry standards that must be maintained. In the first lab a UVM testbench is built to verify a trivial design. UVM components are constructed using a step-by-step guide with a detailed description of the UVM code. In the second lab a simple APB finite state machine is verified. This lab includes a verification plan, assertion coverage and functional coverage. These features are widely used by the verification industry to create a more robust verification environment

Developing Industrial Strength UVM for Academic Research and Teaching

by

George M. Onwubuya, B.S.E.E.C.E.

A Thesis

Approved by the Department of Electrical and Computer Engineering

Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

Keith E. Schubert, Ph.D., Chairperson

Linda J. Olafsen, Ph.D.

Stephen T. McClain, Ph.D.

Accepted by the Graduate School
August 2020

J. Larry Lyon, Ph.D., Dean

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Firstly, I would like to express my special appreciation and gratitude to my advisor, Dr. Keith Schubert. His continuous guidance, patience, career advice and immense knowledge helped me in all the time spent researching and writing this thesis. I would also like to thank friends and co-workers for all the interesting and stimulating discussions we had about the necessary skills that a verification engineer must have and more importantly what constitutes a quality testbench environment. I would also like to thank my dear wife who has bore the brunt of my frustrations but has never stopped encouraging me. She has urged me on to the finish line. Lastly and by no means the least, I would like to thank God for his wonderful and ever gracious providence. To him be all the glory.

DEDICATION

To my life-coaches, my parents, I owe it all to you.

CHAPTER ONE

Introduction

Verification is a process that takes place on a daily basis and it can range from verifying a food order to balancing accounting books. Formally, verification is a process used to demonstrate that the specifications of a design are preserved throughout its implementation. In the era of reusable intellectual property (IP), system-on-chip (SoC) devices, and multi-million gate application specific integrated circuits (ASICs) the process of verification has taken a central role. Verification is estimated to take approximately 70% of the design effort. In order to cope with the increase in verification load, companies are forced to hire as many as twice the number of design verification engineers to design engineers.

Given the amount of time and effort required for the verification process, it is no surprise that the verification process often forms the critical path of the project plan. Verification engineers and the industry as a whole have sought methods and tools that would reduce verification time, enable parallelism between design and verification efforts, and increase automation. In order to parallelize the verification process it is vitally important to avoid any interferences with the low level design. Higher levels of abstraction enable the verification engineers to work more efficiently without having to worry about the low level design and any changes that could be made to it. Higher levels of abstraction do present their own unique problems. It leads to less control over the design and therefore it is important to choose the method of abstraction wisely. Automation requires a standard with well-defined inputs and outputs for it to take place. Not every aspect of the verification process can be automated because of the various functions, interfaces, protocols and transformations. Therefore it is not possible to create a general-purpose automated solution for verification.

The reconvergence model is a conceptual representation of the verification process. Verification begins at the end point of the transformation process, and ends at the start point of the transformation process. Therefore the verification process is used to reconcile the result with the beginning of transformation. This is why a line with a double arrow is used to represent the verification process in Figure 1.1. According to [1], transformation could be any process that takes an input and produces an output. Transformation could be taking a specification and creating the appropriate circuit using register transfer level (RTL) code. Verification checks whether the RTL implementation meets the specification. The verification process is prone to human

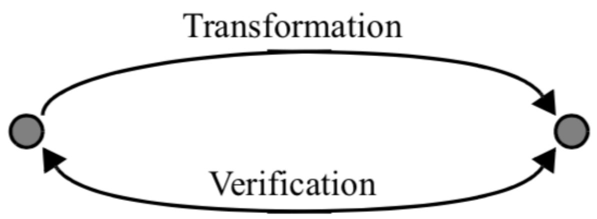


Figure 1.1: The reconvergence model shows that without a common starting point verification is not possible. There must therefore be two common points for the convergence to be possible.

error and therefore much thought must be given to eliminating as much human error as possible. As previously mentioned, automation is one of the mechanisms used to error proof a verification process. In the field of RTL coding, complete automation is not possible because at some level human intervention is required. Figure 1.2 describes the scenario where a designer interprets user requirements from a specifications document. During this process, the verification effort is focused on verifying the designers interpretation of the user specifications. Therefore if the interpretation is wrong in anyway this verification scenario will not reveal it. Redundancy is added to the verification process to guard against the misinterpretation of the user specifications. Figure 1.3 describes a scenario where a designer engineer creates RTL code that is meant to address the specification and the verification engineer will independently

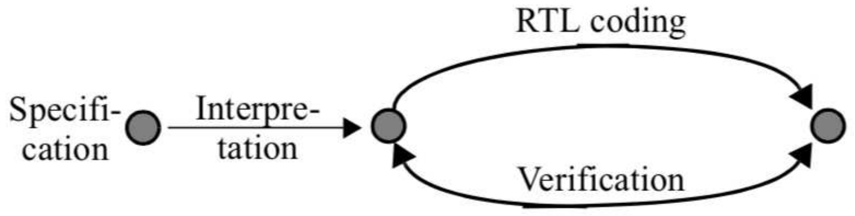


Figure 1.2: The common origin for this scenario is at the interpretation point and not at specification. This method is prone to the misinterpretation of user specifications.

test the transformation. It is important to note that the verification engineer is also aware of the user specification and therefore is able to detect an error in the designer's interpretation. This is the verification process that has been widely adopted by the ASIC industry. Origin and reconvergence points in the verification process are deter-

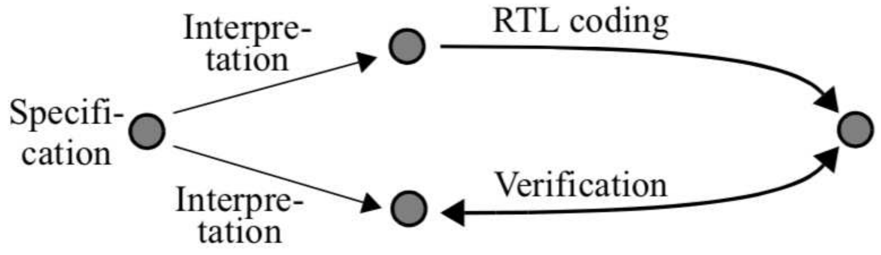


Figure 1.3: The reconvergence model shows the redundancies used to protect against the misinterpretation of the user specifications.

mined by the method or tool used for verification. These tools and methods include formal verification, functional verification, property checking and rule checkers.

Formal verification mathematically proves that the origin and output are logically equal thus showing the transformation maintained its functionality. Equivalence checking is a common formal verification method and its most common use is found when comparing two netlists. It is also used to check whether a netlist correctly implements the original RTL code. A netlist is a connection of gates derived from synthesizing the RTL code. History has shown that synthesis tools are prone to error and equivalence checking is used to keep the synthesis tools honest.

Property checking is another application of formal verification technology and involves proving or disproving the assertions or characteristics of a design. Characteristics such as checking state machines for isolated states or assertions about the signal interfaces of a design are some of the property checks that can be done. Property checks have their challenges, one major obstacle is the interpretation of design specifications which assertions are required to prove. Furthermore, for assertions to prove useful, they are not to be written as trivial restatements of the design behavior, but rather they should be based on the user requirements. Assertions will be discussed in further detail in Chapter 4.

Functional verification is used to verify the design intent at the design, unit, chip and system level. It's used to reconcile a design with its specification. Functional verification checks that the transformation of a specification document was done without any misrepresentation. Functional verification can be approached using three methods: black-box, white-box and grey-box. These approaches will be discussed in further detail in Chapter 2. It is important to note that for functional verification to take place the specifications document must be written in formal language with precise semantics.

It is worth mentioning that testing and verification are not the same concept. The two are often confused and used interchangeably. Testing is used to verify that a design component was manufactured correctly and verification is used to ensure that design meets its functional intent. During testing the finished silicon is reconciled with the netlist used for manufacturing. Testing uses test vectors to test physical locations and observe for changes from 0 to 1.

A Brief History of UVM

The verification industry has made large strides in building reusable and robust testbench environments. The Verilog Hardware Description language was created to model the design behavior of a circuit. It also has a few constructs that could be used for creating tests. This makes it rather cumbersome for carrying out the verification of complex chip or system-on-chip (SoC) designs.

Hardware Verification Languages such as ‘OpenVera’, ‘e’ and ‘SystemC’ were designed to make the verification process easier. SystemC was particularly useful because of its software approach to verification. RTL requires pin level activity to communicate. As the RTL design grows, the number of pins required for communication increases and this slows down simulation significantly. SystemC uses transaction level modeling (TLM) to communicate by making simple function calls. By replacing a lot of pin wiggling with function calls, a dramatic speed up in simulation is realized [2].

Over time it became apparent that there was a growing need for a unified language and a consortium of electronic design automation (EDA) companies was created. The result of this effort was SystemVerilog, an extension of verilog which inherited the verification constructs of OpenVera and makes use of TLM from SystemC. System Verilog has proved to be very valuable in the creation of quality designs as well as implementing a reliable verification environment that can be used across different projects.

It is important to note that the evolution of a verification standard and SystemVerilog were taking place at the same time. In 2000, Verification Advisor (vAdvisor) was introduced to the verification industry. It was a collection of best known practices for verification design. It touched many aspects of verification such as stimuli creation, self-checking testbenches and coverage model creation [3]. Although this collection was useful it did not address the need for automation and reuse. In 2002, the creators of vAdvisor released the eReuse Methodology (eRM) which became the first verification library.

In response, Synopsis created and released the Reuse Verification Methodology Library (RVM). RVM lacked a lot of the architectural guidelines that came with eRm and therefore users always saw it as a subset of eRM [3]. Overtime Synopsis converted RVM to the SystemVerilog Verification Methodology Manual (VMM) which was used to support the ever evolving SystemVerilog standard.

Mentor Graphics created the Advanced Verification Methodology and although it made use of TLM found in SystemC it left out key verification needs such as complex stimuli generation, test classes and more [4]. AVM was important because it was the first open-source verification solution.

Cadence released the Universal Reuse Methodology (URM) in early 2007. It was open source and made use of TLM communications, but most importantly it migrated proven solutions from the eRM architecture into the SystemVerilog URM. It also provided a significant upgrade by adding new features such as an abstract factory, class automations, test classes and configuration mechanisms [5].

In 2008, Cadence and Mentor Graphics released the Open Verification Methodology (OVM). The library unification was made rather smooth because both URM and AVM were already using TLM as the standard communication protocol. However, OVM like all its predecessors was a standard that was simulator dependent. In 2010, OVM was chosen as the basis for the Universal Verification Methodology. UVM is currently the industry standard and has been tested by all vendors who are a part of the Accellera group. It is meant to be simulator independent and has several other advantages.

A Survey of Current Literature

Verification is a complex process and therefore it comes as no surprise that there is a steep learning curve that is associated with UVM. The Accellera organization has produced the UVM 1.2 class reference manual which is used to provide a detailed description of the SystemVerilog classes used in UVM. According to [6] it is important to make use of this document but only after one has a firm grasp on the fundamentals of UVM.

UVM resources follow two general approaches. The focus of the first approach is to tailor UVM training to design verification engineers who are already a part of the industry. Verification books released before or during the development of UVM focused heavily on the practical application of advanced verification techniques for the everyday verification engineer [7].

Industry experts realized that the goal of UVM was to help verification engineers find bugs earlier in the design process and the best way to do this was through the use of controlled randomness [3]. Therefore the UVM testbench is composed of reusable UVM-compliant components which are ready to be used and configured to suit any design interface protocol. This approach led to further developments in constrained-random verification which has proved to be very useful in ASIC verification.

As the introduction of verification libraries grew, industry experts began to adopt verification techniques for system-level design. Verification for block-level design is very different from the verification of system-level design. For the latter, it is important to consider how individual blocks that make up the system interact with one another [8].

Scalable techniques for Formal verification was introduced by Ray Sandip. Its aim was to give a broad overview of the spectrum of formal verification techniques, as well as combining such techniques into a single framework [9]. More importantly, Gaurav Bhatnagar et al states how formal verification and portable stimulus standards could compliment a dynamic verification standard like UVM [10].

This first approach is useful for the seasoned verification engineer by focusing on the finer elements of UVM testbench. This approach discourages a beginner who seeks a simpler overview of UVM without being burdened with the enormous amount of information that is often associated with industry sources. The second approach was tailor made for the novice verification engineer. According to [11], it is important to have a solid foundation of object-oriented programming constructs (OOP) in SytemVerilog before learning UVM.

Resources that adopt this approach make use of fairly simple designs to illustrate a fully functional UVM testbench. The source [12] adopts this approach and provides multiple design examples to illustrate the many features of UVM. However, too many design examples distract the beginner from focusing on the important elements of a fully functioning testbench.

A single design is used by [13] as the only design example to illustrate the different components required to build a fully functional testbench. The Ray Salemi builds the UVM testbench around a tiny arithmetic logic unit and supplements high-level discussions of code behavior with detailed videos. This method of presenting UVM to a beginner seems to work however it leaves out the basic industry standards required by companies seeking to hire verification engineers such as coverage and assertions.

Some resources such as [14] have sought to address the lack of industry standards. It provides advanced best practices on how to apply UVM beyond an introduction of the verification methodology. Although this resource is very useful, one must go beyond the basic introduction of UVM to fully grasp these concepts.

There are resources that have attempted to balance industry standards with simple UVM testbench examples. The source [6] provides a good method for achieving this balance. The aim is to present the building of the UVM testbench around a simple design by separating the different parts into stimulus generation, stimulus delivery and checking the response from the design-under-test (DUT). It also provides a brief introduction into industry methods such as the register abstraction layer and coverage with example code. Although this is one of the best resources for beginners, it still lacks an overview of assertion coverage, how to write a standard verification plan and there is no access to the code examples found in the book.

The main focus of this work is centered around the universal verification methodology (UVM) architecture used in functional verification. This paper will outline the steps involved in setting up a typical UVM testbench that meets the standard required by the verification industry. It centers around two designs, the first is rather trivial and is meant to keep the focus on the features of the UVM testbench. The second design steps up the level of complexity and includes a complete verification strategy, UVM testbench, assertion and functional coverage.

Industry Desired Skills

A verification engineer must have solid knowledge about the design-under-test (DUT). This knowledge is often derived from the design specification document. It is imperative that a thorough understanding of the design specification document is achieved and sometimes even a greater understanding than the design engineer. The knowledge is important and fundamental for all aspects of verification. This includes:

- Defining a proper verification plan and identifying all the features and corner cases for testing
- Identifying the right methodology for verifying different design block features. A one size fits all approach will not work for different features of the design.
- Implementation of efficient testbenches using components like drivers, checkers, monitors, coverage and efficient stimulus patterns.
- Identifying and triaging simulation failures and recommending fixes.

Debugging is going to be part of the everyday job of a verification engineer. Verification engineers spend a lot of time debugging when compared to other tasks. Mastering the art of debug is a quality often overlooked among verification engineers who are just starting off, but it is a hugely desirable industry skill. The modern verification engineer is required to have more software skills on top of a fundamental understanding of the hardware design concepts. Gaining good knowledge in a verification language like SystemVerilog and a widely used methodology like UVM is only possible with good fundamentals in programming and software engineering concepts. Finally, demonstrating results with quality is a core value in any engineering discipline and it is important that a verification engineer carry out their work with the highest standards of quality. Anything less, and the company could be made to suffer huge losses. It is important that verification engineers take responsibility of tasks and finish them on time.

Content Structure of this Thesis

Chapter Two

Chapter Two gives a detailed background of the functional verification approaches, benefits of UVM, verification components, features of UVM, 3 C's of a good verification methodology and tests and coverage.

Chapter Three

Chapter Three introduces the basic features behind a fully functional UVM testbench environment. The trivial design to be verified adds two 16-bit numbers.

Chapter Four

Chapter Four introduces assertions and coverage. These are industry methods used to improve the quality of UVM testbenches. It details the use of assertions, types of assertions and how to write a good assertion. It provides a brief overview of coverage and cross coverage.

Chapter Five

Chapter Five introduces the second design which much more complex than the first. It details a verification strategy which includes features to be verified, stimulus generation and checkers. It also adds functional coverage and other testbench features that were intentionally left out in the first UVM environment.

Chapter Six

Chapter Six outlines a conclusion for the thesis. It recommends learning to use the register abstraction layer (RAL) and virtual sequencer as next steps.

CHAPTER TWO

Background

Functional Verification Approaches

Functional verification can be accomplished using three complementary approaches, black-box, white-box and grey-box.

In the black-box approach, functional verification is performed without any knowledge of the design implementation. Verification is accomplished through the available interfaces without direct access to the internal state of the design. This approach suffers from a lack of controllability and visibility, this means that it is often difficult to set up interesting state combinations to isolate specific functionality. It is also difficult to debug a response from the input if an error is produced. This problem often arises from the long delay experienced between the occurrence of the problem and the appearance of the symptoms [1, 15]. However, a huge advantage of the black-box verification is that it does not depend on any design implementation. This means that whether a design is implemented in ASIC, RTL code, transaction-level model, or entirely in software, the approach to verification remains the same. Therefore the black-box verification model serves as a good model to develop golden testbenches. The black-box approach is very impractical in the today's large and complex ASIC systems. There too many internal signals and states to effectively verify the ASIC using only this approach.

The white-box approach provides full visibility and controllability to the internal implementation of the design. It allows the person carrying out verification to setup interesting combinations of states and inputs quickly [1, 15]. It therefore becomes easier to observe the results and to treat any bugs in the design. This approach is heavily dependent on the particular design implementation. Therefore any change in the design would require a change in the testbench. It also requires the verification engineer to have a significant amount of knowledge of the design implementation to know which test conditions to create and which results to expect. The white-box verification approach is a useful complement to the black-box approach to ensure low-level design behaves correctly. Assertions are ideal for this type of approach, the assertion states that at a particular event (clock or a signal), under a given condition, a certain result is expected of the design. For example, when checking the rollover of a counter, the assertion would check that when the count reaches maximum count the next count must be zero.

The grey-box verification approach is a compromise between the first two methods. It aims to fully exercise all the parts of the design while still being portable [1, 15]. Like the black-box verification method it has the benefits of observing the design entirely through is top-level interfaces. However, the verification is intended to fully exercise low-level implementations of the design. A typical grey-box strategy is to include some non-functional registers in the design to increase visibility and controllability. A good example would be to include a signal that can be used to force a counter to its maximum value in order to speed up simulation time. These registers and features would not be used during normal operation of the design but are often useful in providing more control to the verification engineer.

If verification is to be done in parallel with design implementation using TLM as stated in chapter one, the black-box and grey-box approaches are the only possible avenues for design and verification parallelization. They both do not require a detailed implementation of the design beforehand in order for verification to take place.

UVM

Benefits of UVM

According to [16], the benefits of UVM include:

- Modularity and Reusability - The methodology is designed as modular components and this allows for reusing of components across multiple units as well as across projects.
- Separating Tests from Testbenches - Tests that are generated using sequencers and other stimulus can be separated from the actual testbench hierarchy and hence tests can be reused across different units and the project
- Simulator Independent - The UVM base class library and the methodology as a whole is supported by all simulators and hence is not dependent on any simulator.
- Sequence control gives good control on stimulus generation. Sequences can be developed in several ways using randomization, layered sequences, virtual sequences, etc. This provides good control and rich stimulus generation capability.
- Configuration mechanisms simplify the configuration of objects with deep hierarchy. The configuration mechanism helps in easily configuring different testbench components based on the verification environment using it without having to worry about how deep any component is within the testbench hierarchy.
- Factory mechanism is an OOP method used to simplify the modification of components. Creating a component using factory enables them to be overridden in different tests or environments without changing the underlying code base.

UVM does have its drawbacks. As previously mentioned, UVM has a steep learning curve to understand all the details of the library. The methodology is still developing and has lots of overhead that can sometimes slow down RTL code simulation.

UVM Components

Verification components can be divided into stimulus generating components, and components that monitor and check the effects of injecting stimulus into a DUT. The transaction, sequence, sequencer and driver form part of the stimulus generation of the verification environment. The monitor and scoreboard are used to monitor effects of injecting stimulus into the DUT. Components such as the agent are used to make the verification environment re-usable from one design to the next [17].

Sequence Item. The sequence item contains "transaction" data [18] . Networking packets and processor instructions are some examples of transactions in UVM. Put simply, a sequence item is a packet that contains data.

Sequence. The sequence is used to take care of the creation, randomization and finalization of the transaction objects. The sequence uses a transaction object and creates multiple iterations of the object to form a stream of input data [19]. These complex sequences can be used to simulate operations like bus read and write protocols. It is possible to create virtual sequences that contain several normal sequences. This is normally useful for a verification environment that deals with several verification components that require multiple sequences. If the sequence item is a packet, then think about the sequence as the packet generator.

Sequencer. The sequencer acts as an inter-agent between the production and use of transactions. It is used to implement the handshaking methods between the sequence and the driver [19]. Think of the sequencer as a packet router that sends packets (transactions) to the driver.

Driver. Driver as the name suggests is used to drive the DUT signals. It receives the transaction object from the sequencer and converts it into pin level activity [20]. The driver is active part of the of the verification environment and uses clock inputs to time the events. A driver can drive a simple single transmission or it can be used to drive multi-cycle transfers such as in the case of bus transmissions. In keeping with the network analogy, the driver is the packet injector because it drives packets to the DUT.

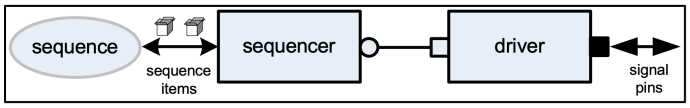


Figure 2.1. Connection between sequence item, sequence, sequencer and driver

Monitor. The monitor is a packet sniffer and is responsible for sampling signals at the DUT. The monitor collects the data items from the DUT and translates it into a transaction making it available for other verification components [21]. This information is normally sent to the scoreboard to test for correctness or subscriber for coverage. Monitors can divided into bus monitors and agent monitor. All bus signals and bus related transactions are handled by a bus monitor. Signals and transactions related to a specific agent are handled by an agent monitor. It always recommended to create a monitor that does not depend on driver for information.

Agent. The agent component is a container that encapsulates the sequencer, driver and monitor [22]. All these components can be called through the agent. Verification environments can have more than one agent and some agents can be master agents involved in initiating transactions to the DUT. Other agents can be slave agents that react to the transaction requests. Agents can be configured to act as either an active or a passive agent. Active agents are responsible for driving transactions, while the passive agents are responsible for only monitoring the DUT behavior.

Scoreboard. The scoreboard performs high-level validation checks based on the transaction received from the monitors of the different verification components. It predicts what DUT outputs should occur based on the DUT inputs and then compares the predicted results to the actual DUT outputs [23]. It may contain a functional model of the DUT. By collecting traffic from all the DUT ports the scoreboard enables verification of the device behavior.

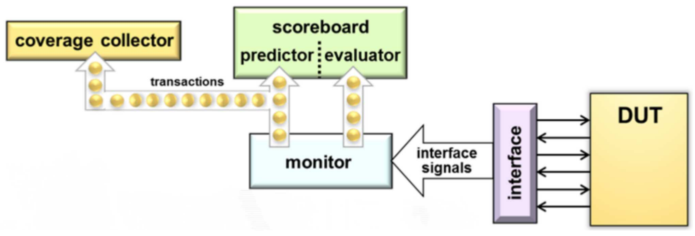


Figure 2.2. Connection between a monitor, scoreboard and coverage collector

Environment. The environment is another container. This particular container holds one or more agents and other environments [24]. It also contains other objects that are needed for simulation such as a scoreboard, register model, memory model, coverage objects, etc. It's called an environment because it contains the components that are necessary to build an effective verification environment.

Test. The test component is a top-level element of the UVM based simulation. It defines a testbench scenario, by scheduling execution of the high level sequences on the respective virtual sequencers [25]. The UVM test allows for configuration of its verification components and customization of the reusable environment.

Features of UVM

Transaction-Level Modeling (TLM) Protocol. UVM uses the SystemC TLM 1.0 standard for communication between components [26]. A "port" or "analysis port" specifies a set of communication methods used for a particular connection. An "export" or "imp export" implements the ports methods. A port must be paired with exactly one export(one-to-one). An analysis port is paired with zero or more imp exports (one-to-many).

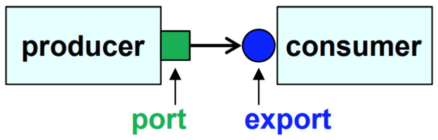


Figure 2.3: A simple example of how two components can communicate using a TLM port and export.

In Figure 2.3, the producer pushes transactions to the consumer. The producer can create a transaction and place the transaction in the TLM port.

A TLM FIFO(first-in-first-out) is used for transactional communication when both the producing component and consuming component need to operate independently [26]. In this case as shown in Figure 2.4, the producing component generates a transaction and places it into the FIFO, while the consuming component pulls one transaction at a time from the FIFO.

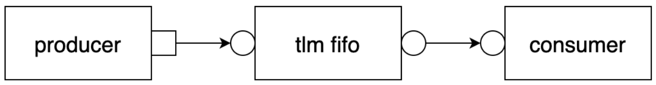


Figure 2.4. Components communicating using the TLM FIFO

TLM is normally used when connecting the driver component to the sequencer and the monitor to the scoreboard or coverage report as shown in Figure 2.5

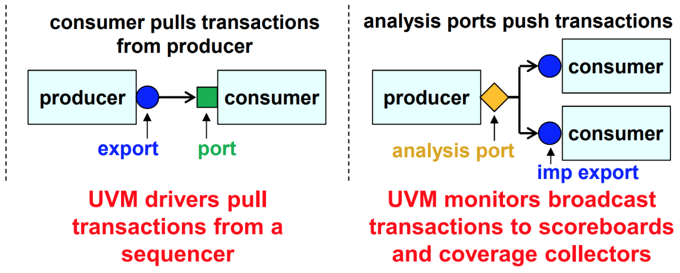

consumer pulls transactions from producer
producer
consumer
export
port
UVM drivers pull transactions from a sequencer
analysis ports push transactions
producer
consumer
consumer
analysis port
imp export
UVM monitors broadcast transactions to scoreboards and coverage collectors

Figure 2.5. Typical usage of TLM in UVM

UVM Class Libraries. UVM is a collection of SystemVerilog classes. It provides 3 core base classes as shown by Figure 2.6. This includes the uvm object, uvm component and uvm transaction.

The *uvm_ object* is used to define a core of class-based operations such as create, copy, compare, print, sprint, record, etc. It also defines interfaces for instance identification and random seeding. All components and transactions are derived from the uvm_object class [27].

The *uvm_ component* class is the root base class for all UVM components. Components are quasi-static objects that exist throughout the simulation [27]. Therefore they can be used to establish a structural hierarchy like modules and program blocks. The uvm component also defines a phased test flow that components must follow during the course of the simulation. Phases are discussed later in this chapter. Every component is uniquely addressable via a hierarchical path name. Finally, uvm_component can also be used to define configuration, reporting, transaction recording and factory interfaces.

The *uvm_ transaction* class is the root base class for all the UVM transactions [27]. Unlike components they are transient in nature. Simple transactions can be derived directly from the uvm transaction but transactions that are sequence- enabled will be derived from the uvm sequence item class.

The *uvm_root class* is a special form of the uvm component class. It serves as the top level component for all the other components found in the UVM environment [27]. It provides phasing control for all the UVM components.

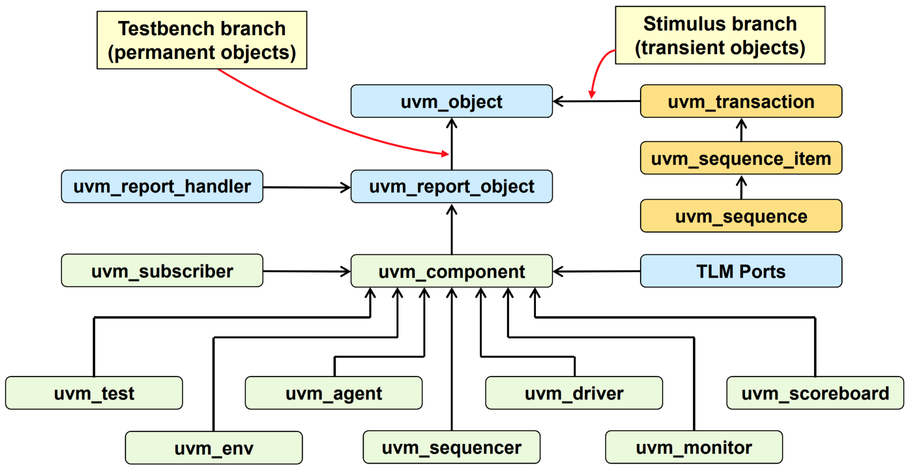


Figure 2.6. UVM class hierarchy

UVM Factory. The UVM factory is used to build the UVM object hierarchy. It is used to construct the class objects using the factory create() method instead of the class constructor new(). It allows tests to swap out testbench components for specific tests. It can be used to change drivers, scoreboards and other components as well as method behavior and constraints.

Think of the UVM factory as having an assembly line and creating objects required during simulation. UVM is a dynamic testbench environment, this means that tests are constantly begin swapped in or swapped out. In Figure 2.7, the UVM testbench begins the execution of test usb2_test. This test requires a USB 3 agent, however the UVM environment is making use of the USB 1 agent. The factory is used for building the components corresponding to the USB 3 agent (sequence item, sequencer, driver etc.) and swapping the USB 1 agent for the USB 3 agent.

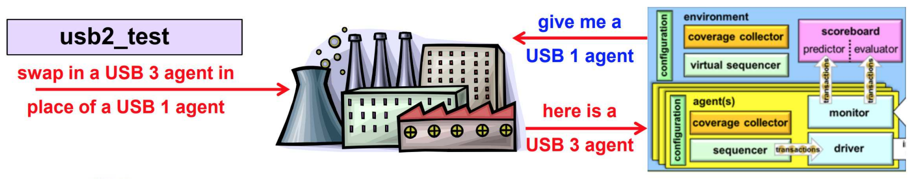


Figure 2.7. UVM factory

UVM Phases. UVM phases are used to define the simulation phase for static objects derived from the uvm component class. UVM components synchronize with each other using UVM phases. There are 3 phases in the UVM testbench; construction phases, Run phases and Cleanup phases.

The *construction phase* is where the testbench is configured and constructed. All construction phases execute in zero time and at simulation time zero [28, 29]. The construction phase includes the build phase, connect phase, end of elaboration-phase and start of simulation-phase. The build phase is used to construct the components of the UVM testbench from the top-level hierarchy to the bottom using the UVM factory. The connect phase is responsible for TLM connections. Handles to test bench resources are also assigned during this method call. This method is called after a successful construction of UVM testbench components. The end of elaboration phase is used to make final adjustments to the testbench structure, connectivity, or configuration before the start of simulation. Start of simulation phase occurs before the start of the time consuming part of simulation and is used to set initial run-time configurations.

The *run phase* category is executed after the start of simulation phase. The run phase is the only phase included in this category, and it is where the actual execution of simulation takes place. It is important to note that this is the only phase that is defined as a task because it consumes time [28, 29]. Stimulus generation and test bench activity checks are handled by this method. To add greater control and flexibility UVM callbacks are added to the run phase. UVM callbacks are methods used to alter the behavior of the transactor(driver) with modifying it. The callback methods in this phase include pre-reset, reset, post-reset, pre-configure, configure, post-configure, pre-main, main, post-main, pre-shutdown, shutdown, post-shutdown. Pre-reset is used to take care of all the activities before reset. Its execution begins at the same time as the run task. Reset is used to put all the DUT related interface signals into their respective reset states. Post-reset is used to execute any activity following the reset. Pre-configure is used for the preparation of the DUT configuration following reset. This includes waiting for the creation of components responsible for driving the DUT signals. Pre-configure is intended for any activity that is required to prepare the DUTs for the configuration process after reset is completed. Configure is responsible for programming the DUT and setting the signals that would initiate the test case. Post-configure is used to wait for the effects of configuration to propagate through the DUT or for the DUT to reach a state where it is ready to start the main test stimulus. Pre-main is used to ensure that all components are ready to begin generating stimulus. Pre-main is used to ensure that all required components are ready to begin generating stimulus. Main is where the stimulus specified by the test case is generated and applied to the DUT. It completes when either all stimulus is generated or a timeout has occurred. Post-main is used to take care of any activity required for the finalization of the main. Pre-shutdown is a buffer for any DUT stimulus activity that needs to take place before the shutdown phase. Shutdown is used to ensure that any effects of stimulus generation during the main has propagated through the DUT and that any resultant data has drained away. Finally, post-shutdown is used to perform any final activities before exiting the active simulation phase. At the end of the post shutdown the UVM testbench starts the cleanup phase.

The *cleanup phases* are where the results of the tests are collected and reported [28, 29] . All cleanup phases are executed in zero time. The cleanup phases category is meant for information extraction from monitors and scoreboards to prove that the test case was successful. It also checks if the coverage goals have been achieved. The extract phase is used to retrieve and process information from the scoreboards and functional coverage monitors. The check phase is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the simulation of the testbench. The report phase is used to display the results of the simulation or to write the results to a file. The final phase is used to complete any other outstanding actions that the testbench has not already completed.

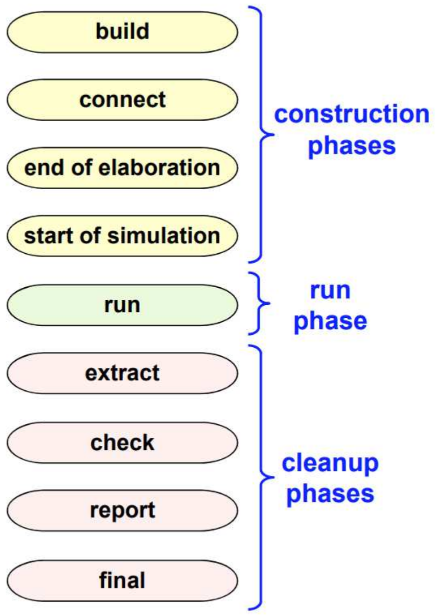


Figure 2.8. UVM phases

UVM Macros. Macros are provided in UVM to semi-automate generation of required UVM code [27]. The different type of macros include report, utility, sequence-related and TLM macros.

The *report macros* are used to provide wrappers around the uvm report* functions. Some commonly used report macros include uvm report info, uvm report warning, uvm report error and uvm report fatal.

The *utility macros* are used inside any user-defined uvm object derived classes. They are meant to define the infrastructure for all the uvm components to ensure correct factory operation. Some commonly used utility macros include uvm object utils and uvm component utils.

The *sequence-related macros* are used for starting the sequence items and sequences on the m sequencer (default sequencer). Some commonly used sequence-related macros include uvm create, uvm do, uvm do pri, uvm do with and uvm do pri with.

The *TLM macros* are used to provide multiple implementation tasks to a port. For example any component that implements a put() function call must implement a corresponding put imp port. If many put() functions are implemented then as many put imp ports must also be declared. Some commonly used TLM macros include uvm analysis imp decl, uvm put imp decl, uvm get imp decl, uvm master imp decl and uvm slave imp decl.

Three C's of a Good Verification Process

Verification of complex systems must not be dependent on the manual inspection of detailed waveforms and vector sets. Functional checking and coverage must be an automated process. The process of verification begins with a verification plan which is derived from the system and design requirements. The verification plan together with automated checking, functional coverage collection and analysis are the foundation of a good verification methodology. The best way to approach a verification process is to start with a simple directed test also known as a bring-up test for the design block. The bring-up test is used to test basic feature of a block and is normally added to smoke tests to ensure that any future changes to the RTL code does not break the verification test. The bring-up test is followed up by random tests to explore the design block space in broad fashion and detect as many bugs as possible. Random stimulus has two significant benefits, it is great for uncovering bugs and it allows for compute resources to be maximally utilized by running nightly or weekly regression tests. Random tests typically cannot achieve 100% coverage and the latter part of the verification process is used to define a series of tests, each of which uses constrained random stimulus to push the design into interesting corner cases. In order to properly direct verification resources, priorities must be set according to the verification plan.

Checkers, coverage and constraints are known as the 3 C's of a good verification process. Constrained verification relies on checkers, coverage and constraints. Each is used to play a key role in the verification process.

Checkers

Checkers ensure functional correctness, and is used to make sure that as more and more random stimulus is generated the DUT is being checked automatically for functional correctness [30]. Checkers can be implemented using SystemVerilog assertions or normal procedural code. Assertions can be directly embedded within the DUT, the external interface or as part of the verification process.

Coverage

Coverage is used to provide a measure of functional completeness of the testing and is used to tell when the goals of verification plan have been met [30]. SystemVerilog offers two separate mechanisms for functional coverage collection; property-based coverage and sample-based coverage. Coverage models are specified in the verification plan and its execution is intimately tied to it as well.

Constraints

Constraints provide the means to reach coverage goals by shaping the random stimulus to push the DUT into interesting corner cases [30]. Random stimulus alone is not sufficient to exercise many of the deeper states of the DUT. Constrained random stimulus is still random, the distribution of vectors is shaped such that interesting cases are reached.

Tests and Coverage

The features in the verification plan should be captured as a set of checker and coverage statements. Many simulation tools provide ways to link coverage and checks directly to the verification plan. This provides direct feedback o the effectiveness of any test. The verification plan is not part of UVM but it is a vital element in the verification process.

Direct testing is written with the purpose of pushing the design into specific cases. For example a customer might request to check whether a general-purpose timer used for profiling rolls over after the maximum count is reached. In this case the maximum count is 32'hFFFFFFFF and is too big a number to capture using constrained random stimulus. A test for this specific case would include forcing the count to a value closer to the max count and observing the results in the simulation waveform. Constrained random test can be written with specific coverage goals in mind, however the tests are not assumed to exercise only one particular feature but rather tests are graded against the coverage model. Tests that achieve the highest coverage in the fewest number of cycles can be used to form the basis of a regression test set.

CHAPTER THREE

Lab 1

Verification Plan

Verification plan details the process of translating the verification requirement specifications into verifiable descriptions. Verification plan is one of the key deliverables of the functional verification process. The testbench architecture is wholly dependent on the verification plan so cutting corners will directly impact functional verification quality. A verification plan is primarily driven by three plans: test plan, coverage plan and checks plan. Fundamentally functional verification is about stimulus generation and creating checks for verifying the response to stimulus. The test and check plans are meant to cover these two primary aspects of functional verification. A coverage plan is added as a third part of the verification plan because of constrained random verification. There is uncertainty introduced by constrained random verification and therefore the coverage plan is meant to ensure that all the design and system requirements are covered during randomization.

Test Plan

The test plan is used to capture the various scenarios to be verified. Test plans often include tests for specific use cases required by a design or system requirement. These include tests that could provide stimulus for configuring the DUT into interesting corner cases. An example could be testing the accuracy of a real-time counter when switching between a 32 kHz clock source and a 40kHz clock source. A specific test could be devised to randomize the number of times the real-time counter(RTC) switches between the two clock sources. This randomization might not be otherwise realized using UVM sequences because the RTC is a small part a larger chip design.

An assertion is used check that the RTC maintains its count accuracy after clock switching has taken place.

Checks Plan

The checks plan lists all the response checks to be implemented for each verification requirement. This is the second key element of a good verification plan. If the DUT has multiple components or physical interfaces, response checks have to be implemented on all interfaces. Response checks are used to flag as errors any deviation from the design or system specifications. Checks could be simple signal level checks, complex sequence of events and data integrity checks. Checks can be implemented as assertions, scoreboards and a combination of the two.

Coverage Plan

The coverage plan should capture the functional coverage requirements. Individual parameters, transactions, sequential and concurrent scenarios have to be captured. The coverage plan can be used to ensure that constrained random verification hits all the required 'bins' for each signal, transaction or register field. When considering coverage, special attention must be paid to micro-architecture coverage and their intersection. For example, watchdog timer(WDT) is sub-block within a microprocessor unit(MCU) block, coverage could be implemented to observe the interaction between the MCU enable and the WDT enable. Functional coverage has to be carefully balanced without missing important coverage goals and ruthlessly weeding out irrelevant coverage.

A good verification plan is essentially an art of balancing between theoretically enumerating all cases while ensuring that irrelevant cases are kept to a minimum. Sequential and concurrent scenarios as well as state machine transitions are but a few important points to keep in mind when writing a good verification plan. Practically only about 80% of the verification plan can be completed during the initial planning phase, the remaining 20% evolves during the course of the project as specifications undergo changes [30].

Testbench Architecture

A bottom-up approach has been used in the design of the testbench architecture. The design begins with design of DUT and ends with a top module. Another way to look at it, is to think about the verification process. The design and interface must be implemented first before sequences are generated. After sequences are generated there needs to be a driver to inject stimulus into the DUT. Thereafter, a monitor broadcasts the DUTs response to the stimulus to a scoreboard which carries out the appropriate checks.

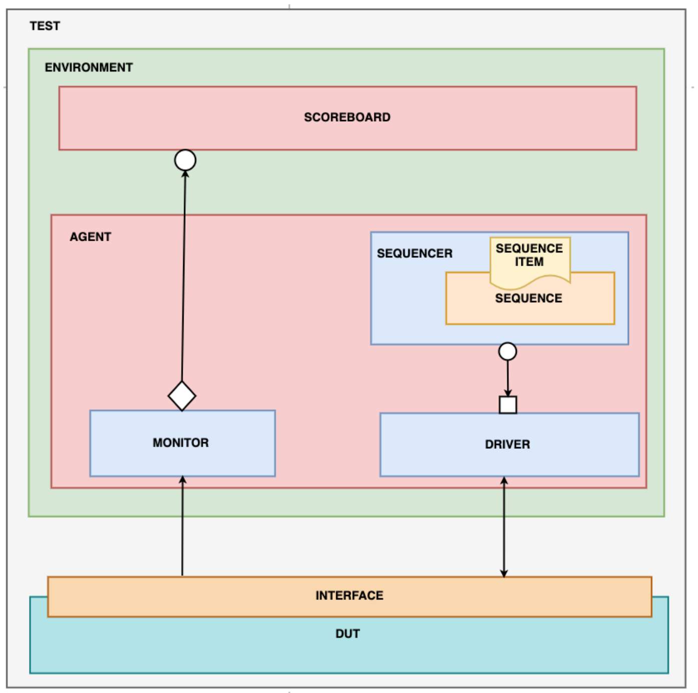


Figure 3.1. Testbench architecture for lab 1

DUT

```
module adder(
  input  logic clk,
  input  logic reset,
  input  logic [15:0] a,
  input  logic [15:0] b,
  output logic [31:0] sum
);

  always_ff@(posedge clk, posedge reset)
    if(reset)begin
      sum <= 32'h00000000;
    end
    else begin
      sum <= a + b;
    end

endmodule: adder
```

Figure 3.2. adder DUT

The DUT used in lab 1 is rather simplistic because the major focus of this section is to become familiar with the UVM testbench architecture. The DUT has an asynchronous reset, two 16 bit inputs and one 32-bit output. The DUT carries out a sum of the two inputs and yields an output.

Step 1: Build Interface

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

  logic [15:0] a;
  logic [15:0] b;
  logic [31:0] sum;

  clocking driver_cb @(posedge clk);
    default input #1 output #1;
    output a;
    output b;
    input  sum;
  endclocking: driver_cb

  clocking monitor_cb @(posedge clk);
    default input #1 output #1;
    input a;
    input b;
    input sum;
  endclocking: monitor_cb

  modport DRIVER  (clocking driver_cb, input clk, reset);
  modport MONITOR (clocking monitor_cb,input clk, reset);

endinterface: adder_if
```

Figure 3.3. adder_if

The following can be used to implement an interface:

(1) Derive adder_if from interface

(2) Implement driver clocking block

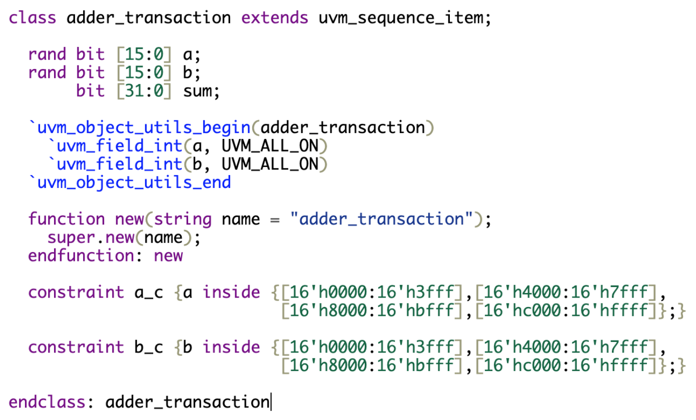
(3) Implement monitor clocking block

(4) Implement driver and monitor modport

The adder_interface consists of two clocking blocks and two modports. A clocking block is a set of signals that are synchronous to a common clock. The clocking blocks in SystemVerilog were introduced to address the problem of specifying the timing and the synchronization requirements of a design. Clocking blocks become especially important for complex designs with multiple architectural design blocks. Design blocks have a myriad number of flip-flops that come with setup and hold times. Trying to recreate these timing paradigms in simulation is annoying and often not scalable. Clocking blocks are used to specify the skew time using the octothorp or pound key(#) construct to ensure events are delayed by a certain number of clock cycles.

The driver and monitor clocking blocks are triggered at the positive edge of the clock. Signals are sampled to or from the DUT after one clock cycle. One important consideration to make when writing clocking blocks is to examine whether a signal is an input or an output to the DUT. The two 16-bit inputs (a and b) are defined as outputs in the driver clocking block because they are inputs to the DUT. Likewise the sum is defined as input in the driver clocking block because it is sampled as an output from the DUT. All the signals in the monitor clocking block are inputs because the monitor samples stimulus inputs and outputs at the DUT. Modports are used to provide direction information for interface ports. The DRIVER and MONITOR modports are used to specify the direction of the signals clock, reset and the clocking bus.

Step 2: Build Sequence Item/Transaction



```
class adder_transaction extends uvm_sequence_item;

  rand bit [15:0] a;
  rand bit [15:0] b;
       bit [31:0] sum;

  `uvm_object_utils_begin(adder_transaction)
    `uvm_field_int(a, UVM_ALL_ON)
    `uvm_field_int(b, UVM_ALL_ON)
  `uvm_object_utils_end

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

  constraint a_c {a inside {[16'h0000:16'h3fff],[16'h4000:16'h7fff],
                            [16'h8000:16'hbfff],[16'hc000:16'hffff]};}

  constraint b_c {b inside {[16'h0000:16'h3fff],[16'h4000:16'h7fff],
                            [16'h8000:16'hbfff],[16'hc000:16'hffff]};}

endclass: adder_transaction
```

Figure 3.4. adder_transaction

The following steps can be used to implement a transaction:

(1) Derive transaction/sequence item from uvm_sequence_item using the "extends" keyword

(2) Add signals (information) to be randomized

(3) Register transaction class with factory using uvm_object_utils

(4) Add class constructor

(5) Add signal constraints

The adder_transaction is a UVM class that is derived from uvm_sequence_item. It is recommended to use the uvm_sequence_item for implementing sequence based stimulus. The adder_transaction is used to group DUT information together, and add constrained randomization to the information. The utility macro uvm_object_utils is used to register the adder_transaction class with the factory. A factory in UVM is a special look up table where all UVM components and transactions are registered. Creating objects using the factory helps in substituting an object of one type with an object of a derived type without having to change the structure of the testbench.

Field automation macros can be defined by adding the words "begin" and "end" at the end of a utility macro. Field automation macros are in the form uvm_-field_*(data member, flag) and they allow access to methods such as copy, compare, pack, unpack, clone, record, print, etc. These methods can be tedious and repetitive to code manually. There are macros of various data types such as integer, enumeration, queues, etc. The flag indicates what type of automation to enable for that data member.

Table 3.1: A few of the possible flag values are listed below. Multiple flag values can be bitwise ORed together.

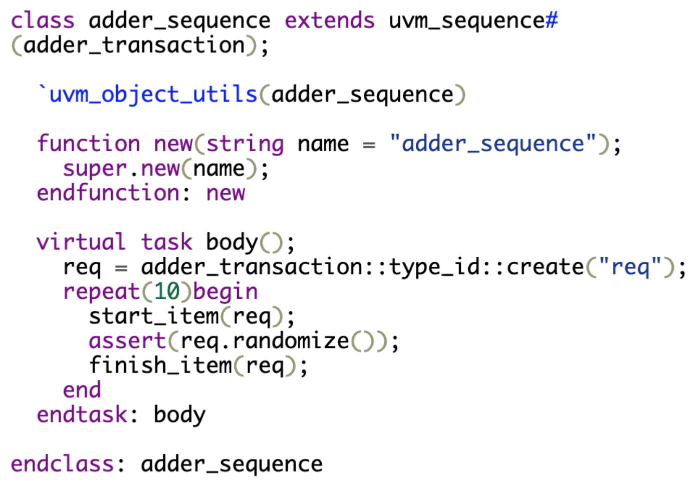
Field Automation Flags

Flag	Interpretation
UVM_ALL_ON	Set all operations on
UVM_DEFAULT	Use the default flag settings
UVM_NOCOPY	Do not copy this field
UVM_NOCOMPARE	Do not compare this field
UVM_NOPRINT	Do not print this field
UVM_NOPACK	Do not pack or unpack this field

The constructor "new" is a virtual method found in the uvm_object class, hence adder_transaction has to include a constructor that follows its prototype template. UVM components/objects are constructed during the build phase but factory constructors should contain default arguments in the definition of these classes. This allows factory registered components/objects to be created initially inside factory and later be passed to the class properties via the create() command as arguments. The default arguments are different for components and objects, in this case the definition is *string name="adder_transaction"* (string name = "name of derived class").

The constraints a_c and b_c ensure that there is comprehensive randomization of the two input numbers. The constraints are divided into four ranges so that when coverage is implemented, all four ranges of the constraints will be used as "bins" and coverage is designated as a success when all four bins are hit.

Step 3: Build Sequence



```
class adder_sequence extends uvm_sequence#
(adder_transaction);

  `uvm_object_utils(adder_sequence)

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

  virtual task body();
    req = adder_transaction::type_id::create("req");
    repeat(10)begin
      start_item(req);
      assert(req.randomize());
      finish_item(req);
    end
  endtask: body

endclass: adder_sequence
```

Figure 3.5. adder_sequence

The following steps can be used to implement a sequence:

(1) Derive sequence from uvm_sequence using the “extends” keyword and pass the transaction as a parameter

(2) Register sequence class with factory using uvm_object_utils

(3) Add class constructor

(4) Define sequence generation in task body()

A sequence specifies one or more sequences to be sent to the driver. The adder_sequence class is derived from uvm_sequence and is parameterized with the uvm_sequence_item type, adder_transaction. Its is registered with the factory using the utility macro uvm_object_utils and It has the same constructor as the adder_transaction because both classes belong to base class uvm_object. There are several ways to define a sequence such as using uvm_do, uvm_do_with, etc. For building the adder_sequence, the start_item and finish_item application program interfaces(APIs) were used.

The sequence order is defined within a task body() which is a virtual method found in the uvm_sequence class. A “req” transaction is first created and then the method start_item is called. The API start_item requests for access to the driver via the sequencer and returns when the driver gets access. The “req” is the randomized and finish_item is called. This type of randomization that waits for arbitration from the sequencer is known as late randomization. The finish_item API results in the driver receiving the sequence item, and is a blocking method which returns only after driver calls item_done. Ten sequence items are generated using SystemVerilog repeat construct.

Step 4: Build Sequencer

```
class adder_sequencer extends uvm_sequencer#
(adder_transaction);

  `uvm_component_utils(adder_sequencer)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

endclass: adder_sequencer
```

Figure 3.6. adder_sequencer

The following steps can be used to create a sequencer:

(1) Derive sequencer from uvm_sequencer using the “extends” keyword and pass the transaction as a parameter

(2) Register sequencer class with factory using uvm_component_utils

(3) Add class constructor

The adder_sequencer is derived from uvm_sequencer and is parameterized with adder_transaction. It is registered with the factory using the utility macro uvm_component_utils. The constructor is defined as *string name, uvm_ component parent* because uvm_sequencer belongs to the uvm_component base class. The sequencer supports an arbitration mechanism to ensure that at any point in time only one sequence has access to the driver. The beauty of UVM is that you can write six lines of code to implement the sequencer. Having the sequencer mechanism prepared for you in the UVM library allows the testbench to be created faster and more effectively.

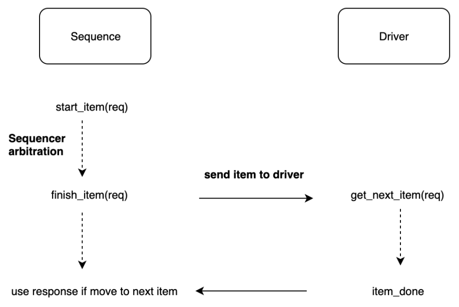


Figure 3.7. Illustration of the handshake protocol between the sequencer and the driver

Step 5: Build Driver

```
`define DRV_IF vif.DRIVER.driver_cb

class adder_driver extends uvm_driver#(adder_transaction);

  `uvm_component_utils(adder_driver)

  virtual adder_if vif;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual adder_if)::get(this, "", "vif", vif))begin
      `uvm_fatal("[DRIVER]", "virtual interface failed at build phase")
    end
    else begin
      `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
    end
  endfunction: build_phase
```

Figure 3.8. Code snippet 1 of adder_driver

The following steps can be used to implement the driver:

(1) Derive driver from uvm_driver using the “extends” keyword and pass transaction as a parameter

(2) Register driver class with factory using uvm_component_utils

(3) Declare virtual interface variable

(4) Add class constructor

(5) Implement function build_phase()

(6) Implement task run_phase()

The adder_driver is derived from the uvm_driver class and is parameterized with adder_transaction. It is registered with the factory using the same utility macro as the sequencer, uvm_component_utils. The constructor is defined as *string name, uvm_ component_ parent* just like the sequencer. A virtual interface is defined in this class and in any class that intends to make used of the interface, adder_if. The virtual interface "vif" is defined using the SystemVerilog construct virtual and the name of the real interface (virtual adder_if vif). In traditional directed testbench environments, all components are static in nature and information is also exchanged in the form signals/wire/net at all levels. This is not the case in UVM where DUT is static(module based) in nature and testbench is SystemVerilog OOP based. We use the the virtual interface as a handle pointing to the interface instance. Virtual interface acts as a medium to connect the DUT and the testbench. Testbench accesses the DUT signals via virtual interface and vice versa.

UVM components are quasi-static, this means they are executed slowly and exist throughout the life cycle of simulation. UVM phases act as a synchronizing mechanism in the life cycle of a simulation. In the build_phase function, the uvm_config_db is used to obtain the virtual interface. The uvm_config_db is a library that is used for data sharing. The uvm_config_db is built from the uvm_resource_db and it is used to store hierarchical configuration values such as the interface. The uvm_config_db is encapsulated in an if statement and when the call is unsuccessful, the uvm_fatal macro will print out the message "virtual interface failed at build phase". If the call is successful the uvm_info prints "build stage complete" with the full hierarchical name of the driver component using get_full_name().

```
 virtual task run_phase(uvm_phase phase);
   forever begin
     adder_transaction driver_trans;
     seq_item_port.get_next_item(req);
     drive();
     seq_item_port.item_done();
   end
 endtask: run_phase

 virtual task drive();
   `DRV_IF.a <= req.a;
   `DRV_IF.b <= req.b;
   @(posedge vif.DRIVER.clk);
   req.sum = `DRV_IF.sum;
 endtask: drive

endclass: adder_driver
```

Figure 3.9. Code snippet 2 of adder_driver

The run_phase is defined as a task because this is where actual simulation takes place. In the run_phase task the heart of execution is encased in the forever block. This is necessary so that these threads execute continuously throughout simulation. The seq_item_port is a TLM port defined for communication between sequencer and driver. It calls the blocking method get_next_item in the driver that blocks until a sequence item is received on the port connected to the sequencer. This method returns the sequence item which can be translated to pin level protocol by the driver. There is a task drive() which is used to drive randomized values for a and b on the interface using the clocking block driver_cb. The line of code $DRIV\_IF.a <= req.a$ used to drive a input to the DUT. It is important to note the a non-blocking assignment is used because both a and b have to be loaded at the same time. DRV_IF is a macro that is predefined and is meant to represent vif.DRIVER.driver_cb. The sum is recorded from the interface using DRV_IF.sum. The item_done() is a non-blocking method call to signal to the sequencer that it can unlock the sequences finish_item method either when the driver accepts the sequences request or it has executed it.

Step 6: Build Monitor

```
`define MON_IF vif.MONITOR.monitor_cb

class adder_monitor extends uvm_monitor;

  `uvm_component_utils(adder_monitor)

  virtual adder_if vif;

  uvm_analysis_port#(adder_transaction) item_collected_port;

  adder_transaction mon_trans;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new
```

Figure 3.10. Code snippet 1 of adder_monitor

The following steps can be used to implement the monitor:

(1) Derive monitor from uvm_monitor using the “extends” keyword

(2) Register monitor class with factory using uvm_component_utils.

(3) Declare virtual interface variable

(4) Declare TLM analysis port

(5) Add adder_transaction variable

(6) Add class constructor

(7) Implement function build_phase()

(8) Implement task run_phase()

The adder_monitor is derived from the uvm_monitor class. It is registered with factory using the utility macro uvm_component_utils. Just like the adder_driver, the adder_monitor has a virtual interface “vif” that is used to monitor information from the DUT. The adder_monitor class must contain a way to broadcast the information collected from the DUT via “vif”. A TLM analysis port is a port definition that is used by the monitor to carry out broadcasting. Think of the analysis port as a drop box where the transaction collected can be dropped off and picked up by any component that needs it such as the scoreboard or a coverage object. The analysis port is parameterized to the adder_transaction and is defined as *uvm_analysis_port#(adder_transaction) item_collected_port*. The line of code *adder_transaction mon_trans* defines a variable used to store data collected from the interface.

```
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual adder_if)::get(this, "", "vif", vif))begin
      `uvm_fatal("[MONITOR]", "virtual interface failed at build phase")
    end
    else begin
      `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
    end
    item_collected_port = new("item_collected_port", this);
    mon_trans = adder_transaction::type_id::create("mon_trans");
  endfunction: build_phase

  virtual task run_phase(uvm_phase phase);
    forever begin
      collected_data();
    end
  endtask: run_phase

  virtual task collected_data();
    forever begin
      @(posedge vif.MONITOR.clk);
      mon_trans.a = `MON_IF.a;
      mon_trans.b = `MON_IF.b;
      @(posedge vif.MONITOR.clk);
      mon_trans.sum = `MON_IF.sum;
      item_collected_port.write(mon_trans);
    end
  endtask: collected_data

endclass: adder_monitor
```

Figure 3.11. Code snippet 2 of adder_monitor

Like the build_phase found in adder_driver, the build_phase function is used to obtain the interface from uvm_config_db. However, this build_phase is different because it includes creating objects that the monitor will use in the run_phase. The analysis port item_collected_port is not registered with the factory so the constructor is used to create it. The analysis port is created using the line of code *item_collected_port=new("item_collected_port",this)*. The factory is used to create mon_trans although for this case it is not necessary because the object will not be overridden. It is good practice to always create objects with the factory whenever possible because the assumption is objects tend to be overridden in large scale testbench implementations.

The run_phase simply calls the task collected_data where the transaction collection is done. In the forever loop found in task collected_data, input values are collected at the interface at the positive edge of the clock. The output sum is also collected at the positive edge of the next clock. The MON_IF is a macro that is predefined and is meant to represent vif.MONITOR.monitor_cb. The transaction collected is written to the analysis using the write method. This is a simple monitor class suitable for this lab. Lab 2 in chapter 5 includes a monitor that adds a few more layers of complexity.

Now that the sequencer, driver and monitor have been created, it is time to create a container class to hold the aforementioned components.

Step 7: Build Agent

```
class adder_agent extends uvm_agent;

  `uvm_component_utils(adder_agent)

  adder_sequencer sequencer;
  adder_driver    driver;
  adder_monitor   monitor;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(get_is_active == UVM_ACTIVE)begin
      sequencer = adder_sequencer::type_id::create("sequencer", this);
      driver    = adder_driver::type_id::create("driver", this);
    end
    monitor = adder_monitor::type_id::create("monitor", this);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    if(get_is_active == UVM_ACTIVE)begin
      driver.seq_item_port.connect(sequencer.seq_item_export);
    end
  endfunction: connect_phase

endclass: adder_agent
```

Figure 3.12. adder_agent

As previously discussed the agent is a container class used to hold the sequencer, driver and monitor classes. The agent is important because it facilitates a key characteristic of a good testbench, reusability. The following steps can be used to implement the agent:

(1) Derive agent from uvm_agent using the “extends” keyword

(2) Register agent class with factory using uvm_component_utils

(3) Declare sequencer, driver and monitor variables

(4) Add class constructor

(5) Implement function build_phase()

(6) Implement function connect_phase()

The adder_agent class is derived from the uvm_agent class. It is registered with the factory using the utility macro ‘uvm_component_utils. The objects are created in the build_phase function such that when an agent is active the sequencer, driver and monitor are created and when it is passive only the monitor is created. The variable get_is_active is of the type uvm_active_passive_enum and it has two possible values uvm_passive and uvm_active. Component construction takes place using *left-hand-side (LHS) = <type>::type_id::create(“<name>”).* The construct type_id is used to pick the factory component wrapper for the class, construct its contents and pass the resultant handle back to the LHS variable. In the connect_phase function the driver’s communication port is connected to the sequencer’s communication export when an agent is active. This is another illustration of the power of UVM such that in one line of code the driver is connected to the sequencer so that sequence items can be passed and executed. The library does the hard part for the verification designer. There is an underlying TLM that handles the handshake between the driver and sequencer, and it is wise for the verification engineer to understand the code under the hood.

Step 8: Build Scoreboard

```
class adder_scoreboard extends uvm_scoreboard;

  `uvm_component_utils(adder_scoreboard)

  uvm_analysis_imp#(adder_transaction, adder_scoreboard) item_collected_export;

  adder_transaction queue_trans[$];

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    item_collected_export = new("item_collected_export", this);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction: build_phase

  virtual function void write(adder_transaction scb_trans);
    queue_trans.push_back(scb_trans);
  endfunction: write
```

Figure 3.13. Code snippet 1 of adder_scoreboard

The basic function of the scoreboard is to check the correctness of the output data of the DUT. The following steps can be used to implement the scoreboard:

(1) Derive scoreboard from uvm_scoreboard using the “extends” keyword

(2) Register scoreboard class with factory using uvm_component_utils

(3) Declare a transaction queue

(4) Add class constructor

(5) Implement function build_phase()

(6) Implement function write()

(7) Implement task run_phase()

The adder_scoreboard is derived from the uvm_scoreboard; however there is no current functionality of the uvm_scoreboard. It is important to mention that uvm_scoreboard has no functionality because scoreboards are design specific and the important aspect is how the scoreboard retrieves its data for comparison. In this scoreboard a queue is used to store transaction data from monitor. In adder_monitor there is an analysis port used to broadcast the collected data therefore in the scoreboard a uvm_analysis_imp must be implemented. The analysis implementation is parameterized with adder_transaction and adder_scoreboard. The line of code *uvm_analysis_imp#(adder_transaction, adder_scoreboard) item_collected_export)* declares an implementation of the analysis port. The line of code *adder_transaction queue_trans[$]* declares the transaction queue used to store data.

In the build_phase, the item_collected_export is created using the constructor. The analysis export of the scoreboard or subscriber must implement the write function. One way to do this is by using a uvm_tlm_analysis_fifo. The benefit of using the fifo is that it has an analysis export, it implements the much needed write function and has an unbounded queue for storing transactions. Another way to do this is using the uvm_analysis_imp and implementing a write function as shown in figure 3.13. The write function is used to push incoming transactions into the queue using the queue method push_back.

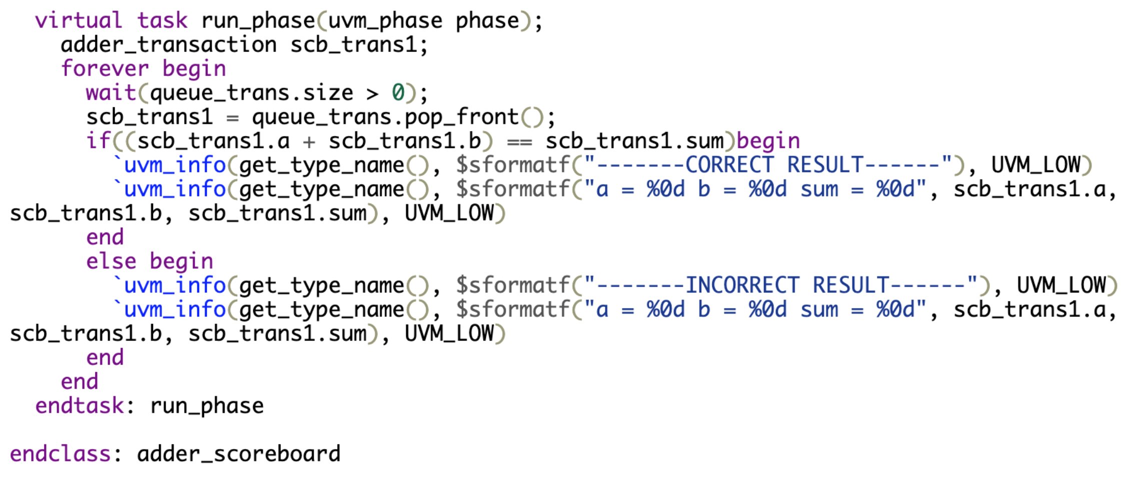

```
  virtual task run_phase(uvm_phase phase);
    adder_transaction scb_trans1;
    forever begin
      wait(queue_trans.size > 0);
      scb_trans1 = queue_trans.pop_front();
      if((scb_trans1.a + scb_trans1.b) == scb_trans1.sum)begin
        `uvm_info(get_type_name(), $sformatf("-------CORRECT RESULT------"), UVM_LOW)
        `uvm_info(get_type_name(), $sformatf("a = %0d b = %0d sum = %0d", scb_trans1.a,
scb_trans1.b, scb_trans1.sum), UVM_LOW)
      end
      else begin
        `uvm_info(get_type_name(), $sformatf("-------INCORRECT RESULT------"), UVM_LOW)
        `uvm_info(get_type_name(), $sformatf("a = %0d b = %0d sum = %0d", scb_trans1.a,
scb_trans1.b, scb_trans1.sum), UVM_LOW)
      end
    end
  endtask: run_phase

endclass: adder_scoreboard
```

Figure 3.14. Code snippet 2 of adder_scoreboard

In the run_phase data comparison is implemented in a forever loop. When there is a transaction in the queue, the transaction is popped from the queue. A check is carried out compare the sum of the inputs to the output. The line of code “scb_trans1.a + scb_trans1.b = scb_trans1.sum” implements this check. If the sum is equal to a + b, a message “CORRECT RESULT” is printed with information contained in the transaction. If the sum is not equal to a + b the message “INCORRECT RESULT” is printed.

Step 9: Build Environment

```
`include "adder_agent.sv"
`include "adder_scoreboard.sv"

class adder_environment extends uvm_env;

  `uvm_component_utils(adder_environment)

  adder_agent agent;
  adder_scoreboard scoreboard;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agent      = adder_agent::type_id::create("adder_agent", this);
    scoreboard = adder_scoreboard::type_id::create("adder_scoreboard", this);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction: build_phase

  function void connect_phase(uvm_phase phase);
    agent.monitor.item_collected_port.connect(scoreboard.item_collected_export);
  endfunction: connect_phase

endclass: adder_environment
```

Figure 3.15. adder_environment

An environment is another container. This container holds agents and other environments such as a scoreboard, register model, memory models and coverage objects. It’s called an environment because it contains the components that are used build an effective verification environment. The environment in this lab is used to simply instantiate the agent and the scoreboard. The following steps can be used to implement the environment:

(1) Include adder_agent.sv and adder_scoreboard.sv files

(2) Derive environment from uvm_env using the “extends” keyword

(3) Register class with factory using uvm_component_utils

(4) Declare objects agent and scoreboard

(5) Add class constructor

(6) Implement build_phase()

(7) Implement connect_phase()

The adder_environment is derived from uvm_env. The class definition should include the adder_agent and adder_scoreboard files. Like all the other classes, a constructor is added, and agent and scoreboard variables are added. In the build_phase the agent and scoreboard objects are created using the factory. Component construction in the build_phase uses the piece of code *LHS = <type>::type_ id::create(“<name>”)* just like the adder_agent. When object creation is complete, the function prints "build stage complete" with full hierarchical name using the the get_full_name() method. The connect phase is used to connect the monitor’s uvm analysis port and the scoreboard’s uvm analysis implementation.

Step 10: Build Base Test

```
`include "adder_environment.sv"

class adder_base_test extends uvm_test;

  `uvm_component_utils(adder_base_test)

  adder_environment environment;

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    environment = adder_environment::type_id::create("environment", this);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction: build_phase

  virtual function void end_of_elaboration();
    print();
  endfunction: end_of_elaboration
```

Figure 3.16. Code snippet 1 of adder_base_test

A test instantiates the environment and each test is a class that is derived from uvm_test. A test library is simply a collection of test that stimulate the DUT. When building a test library it is good practice to begin with a base test from which other tests can derive. This base test would include elements that are required by all tests, such as the environment.

The following steps can be used to implement a base test:

(1) Include adder_environment.sv file

(2) Derive base test from uvm_test using the “extends” keyword

(3) Register class with factory using uvm_component_utils

(4) Declare object environment

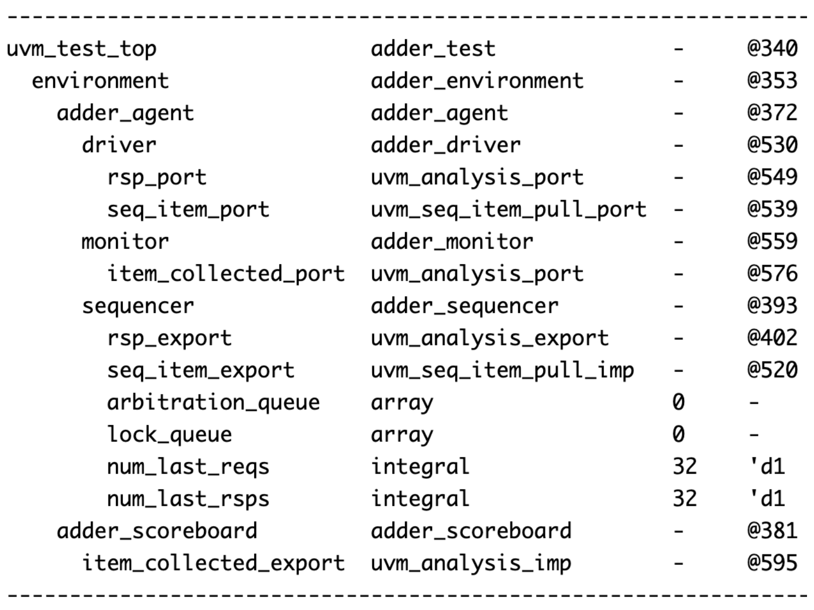
(5) Add class constructor

(6) Implement build_phase()

(7) Implement end_of_elaboration()

(8) Implement report_phase()

The adder_base_test is derived from the uvm_test. The class is registered with the factory using the utility macro uvm_component_utils. The object environment is created from the adder_environment class. The constructor definition is the same as the other constructors found in previously created components. The build_phase function is used to create the object environment using the factory. In the run_phase of the base test the drain time is set and is used to add to the simulation time to allow all elements to complete after the final objection has been lowered. The end_of elaboration function is used to print the topology of the testbench as shown in figure 13.17.



```
----------------------------------------------------------------
uvm_test_top                 adder_test              -     @340
  environment                adder_environment       -     @353
    adder_agent              adder_agent             -     @372
      driver                 adder_driver            -     @530
        rsp_port             uvm_analysis_port       -     @549
        seq_item_port        uvm_seq_item_pull_port  -     @539
      monitor                adder_monitor           -     @559
        item_collected_port  uvm_analysis_port       -     @576
      sequencer              adder_sequencer         -     @393
        rsp_export           uvm_analysis_export     -     @402
        seq_item_export      uvm_seq_item_pull_imp   -     @520
        arbitration_queue    array                   0     -
        lock_queue           array                   0     -
        num_last_reqs        integral                32    'd1
        num_last_rsps        integral                32    'd1
    adder_scoreboard         adder_scoreboard        -     @381
      item_collected_export  uvm_analysis_imp        -     @595
----------------------------------------------------------------
```

Figure 3.17. Testbench topology

The function report_phase is used to keep track of the uvm_error and uvm_fatal message count. If this count exceeds zero then function prints “TEST FAIL” and if this condition is not met the function prints “TEST PASS”.

```
  virtual task run_phase(uvm_phase phase);
    phase.phase_done.set_drain_time(this, 1500);
  endtask: run_phase

  function void report_phase(uvm_phase phase);
    uvm_report_server svr;
    super.report_phase(phase);
    svr = uvm_report_server::get_server();

    if(svr.get_severity_count(UVM_FATAL) + svr.get_severity_count(UVM_ERROR) > 0)begin
      `uvm_info(get_type_name(), "---------------------------------------", UVM_NONE)
      `uvm_info(get_type_name(), "----            TEST FAIL          ----", UVM_NONE)
      `uvm_info(get_type_name(), "---------------------------------------", UVM_NONE)
    end
    else begin
      `uvm_info(get_type_name(), "---------------------------------------", UVM_NONE)
      `uvm_info(get_type_name(), "----            TEST PASS          ----", UVM_NONE)
      `uvm_info(get_type_name(), "---------------------------------------", UVM_NONE)
    end
  endfunction: report_phase

endclass: adder_base_test
```

Figure 3.18. Code snippet 2 of adder_base_test

Step 11: Build Test

```
class adder_test extends adder_base_test;

  `uvm_component_utils(adder_test)

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction: build_phase

  task run_phase(uvm_phase phase);
    adder_sequence seq;
    phase.raise_objection(this);
    seq = adder_sequence::type_id::create("seq");
    seq.start(environment.agent.sequencer);
    phase.drop_objection(this);
  endtask:run_phase

endclass: adder_test
```

Figure 3.19. adder_test

The following steps can be used to implement a test:

(1) Derive test from base test using the "extends" keyword

(2) Register class with factory using ‘uvm_component_utils

(3) Add class constructor

(4) Implement function build_phase()

(5) Implement task run_phase()

The adder_test is derived from the adder_base_test. In the run_phase of this test a handle to the adder_sequence is declared. This class adder_sequence creates a random data transaction that is sent to the sequencer. An objection is raised using the method raise_objection. The objection mechanism is used to communicate when it is safe to end a phase. By raising an objection it indicates that the phase is still in progress. After the objection is raised, the sequence is created using the factory. Notice that the argument for the start method is the “seq” that has been constructed. After the sequence has completed, the drop_objection method is called to indicate the the phase has ended. Notice that the sequence is created in the run phase and not in the build phase. Sequences do not have phases because they are elements that do not persist throughout simulation. Although you can create them in the build phase, it is more appropriate to do so in the run phase so that they can be created and destroyed as needed.

Step 12: Build Module Top

```
`include "adder_interface.sv"
`include "adder_base_test.sv"
`include "adder_test.sv"

module testbench();
  bit clk;
  bit reset;

  always begin
    #5;
    clk = ~clk;
  end

  initial begin
    reset = 1;
    #5;
    reset = 0;
  end
```

Figure 3.20. Code snippet 1 of top module

The module testbench is the top module and it includes the adder_interface.sv, adder_base_test.sv and adder_test.sv files. A clock and reset signals are declared in this module as well as the addition of a virtual interface.

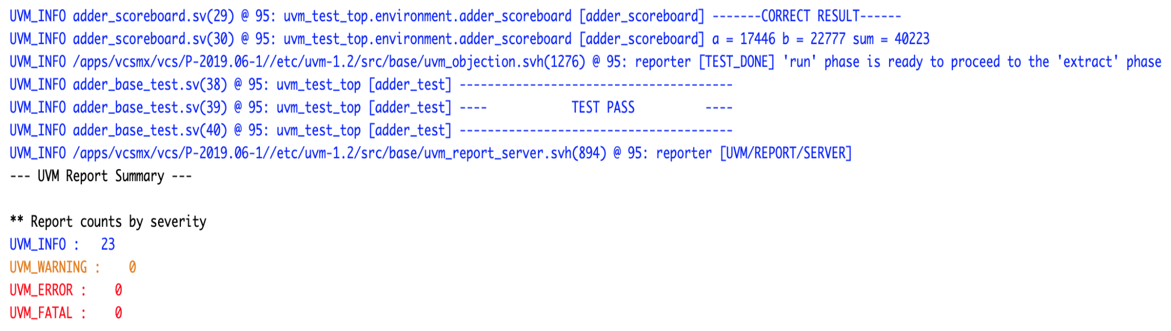
```
  adder DUT(
    .clk(intf.clk),
    .reset(intf.reset),
    .a(intf.a),
    .b(intf.b),
    .sum(intf.sum)
  );

  initial begin
    uvm_config_db#(virtual adder_if)::set(uvm_root::get(), "*", "vif", intf);
  end

  initial begin
    run_test("adder_test");
  end

endmodule: testbench
```

Figure 3.21. Code snippet 2 of top module

A DUT is instantiated in the top module and the configuration data base(uvm_config_db) is used to store the interface which can be retrieved by other elements top down. Finally the call to the run_test creates the test based on the name and then the components in the various build phase methods top down.

Simulation Results

The EDA software online tool was used for writing testbench code and Aldec Pro Riviera was the simulator of choice. Figure 3.22 shows the results of simulation with the number of UVM errors.



```
UVM_INFO adder_scoreboard.sv(29) @ 95: uvm_test_top.environment.adder_scoreboard [adder_scoreboard] -------CORRECT RESULT------
UVM_INFO adder_scoreboard.sv(30) @ 95: uvm_test_top.environment.adder_scoreboard [adder_scoreboard] a = 17446 b = 22777 sum = 40223
UVM_INFO /apps/vcsmx/vcs/P-2019.06-1//etc/uvm-1.2/src/base/uvm_objection.svh(1276) @ 95: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO adder_base_test.sv(38) @ 95: uvm_test_top [adder_test] ---------------------------------------
UVM_INFO adder_base_test.sv(39) @ 95: uvm_test_top [adder_test] ----           TEST PASS          ----
UVM_INFO adder_base_test.sv(40) @ 95: uvm_test_top [adder_test] ---------------------------------------
UVM_INFO /apps/vcsmx/vcs/P-2019.06-1//etc/uvm-1.2/src/base/uvm_report_server.svh(894) @ 95: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   23
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
```

Figure 3.22. Simulation results

CHAPTER FOUR

Assertions and Coverage

SystemVerilog Assertions

An assertion is a statement that a certain property (design or system requirement) must be true. Assertions are integral to design and verification and can be used to document the functionality of a design, check that the functionality of a design over a simulation period is met and determine whether verification tested the design(coverage) [31]. Assertions can be written by a design engineer as part of the DUT model or by a verification engineer as part of the verification test program.

The Verilog language does not have an assertion construct. Verification checks must be coded with programming statements. Simple assertions are difficult to write and maintain, require several lines of Verilog code and cannot be turned off during reset or during other don't care simulation points. Assertion checks look like RTL code and therefore synthesis compilers cannot distinguish between the DUT model and embedded assertion code. Verilog assertion code must be hidden from synthesis compilers and this means even more code.

SystemVerilog assertions are important for several reasons. Dozens of lines of Verilog code can be represented in one line of SystemVerilog assertions(SVA) code. It is ignored by synthesis and hence there is no need to hide checker code. SVA can be turned off during reset, or when a block is disabled and when the simulation reaches a certain point. SVA failures can be categorized according severity level from non-fatal to fatal.

According to [31], using SystemVerilog assertions are important for the following reasons;

- It is a verification technique that is embedded in the language to give white box visibility into the DUT.
- It can be used to specify design requirements using an executable language.
- It enables easier detection of design problems . Error reports show when error has occurred with insight as to why.
- It can be used to report how effective random stimulus was at covering all aspects of the design. Assertion coverage can be used to report on the number of assertions that never triggered and the number that only had vacuous pass.

SystemVerilog supports three general categories of assertions. These are invariant, sequential and eventuality assertions. Invariant assertions are assertions with conditions that should always be true or never true. A good example is a FIFO should never indicate full and empty at the same time. Sequential assertions have a set of conditions that occur in a specific order and over a defined number of cycles. For example, a request signal from a bus should be followed in 1 to 3 clock cycles by a grant signal. Eventuality assertions are assertions with a condition that should be followed by another condition, but with any number of clock cycles in between. For example, if an active-low reset goes low it should eventually go high.

There are two types of SVA, immediate and concurrent assertions. Before getting into the finer details used to construct both types of assertions it is important to comprehend how assertions are scheduled throughout the SystemVerilog simulator.

SystemVerilog Scheduling Semantics

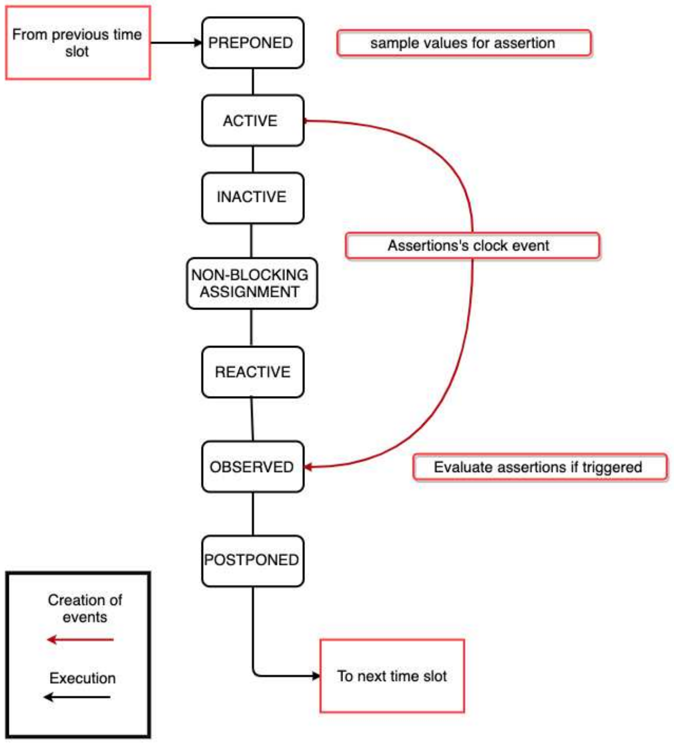


Figure 4.1. Regions inside a SystemVerilog simulation time step

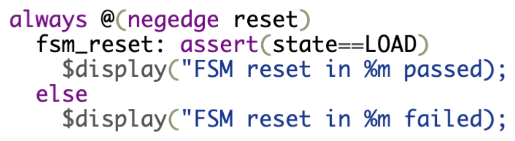
A SystemVerilog simulator has different scheduling regions where events are scheduled and subsequently evaluated. All event evaluations occur in the scheduler's Active region. Events include RTL and behavioral code, blocking assignments and right-hand-side for non-blocking assignments. Events with #0 delays are scheduled in the Inactive region and assignment (left-hand-side) for non-blocking assignments are scheduled in the Non-Blocking Assignment region. Once all the events in the Active region are exhausted, the events in the Inactive region are promoted and likewise when these events are exhausted, the events in the Non-Blocking assignment region are promoted to the Active Region.

The Preponed region has been introduced in SystemVerilog in order to sample all of an assertion's inputs and the Observed region is used for their evaluation [32].

Since assertion inputs are sampled in the Preponed region before any clock or reset events are generated, assertion always sample their input values before the sampling event occurs. This is why when we write assertions we always need to go 1 clock cycle into the future and then observe what happened on the last clock edge. The Reactive region is used to evaluate and execute all program activity to avoid race conditions between design and testbench.

Immediate Assertions

An immediate assertion is a test of an expression the moment the statement is executed. It can be used in initial and always procedures, tasks and functions. Immediate assertion performs a boolean(true/false) test such that when the test result is true, the pass statement is executed and when the test result is false or unknown the fail statement is executed. An immediate assertion can be written in the form: [name:] **assert**(expression) [pass_statement] [**else** fail_statement]. Figure 4.2 is an example of an immediate assertion.



```
always @(negedge reset)
  fsm_reset: assert(state==LOAD)
    $display("FSM reset in %m passed);
  else
    $display("FSM reset in %m failed);
```

Figure 4.2. Example of an immediate assertion

Concurrent Assertions

Concurrent assertions are used to test for a sequence of events spread over multiple clock cycles. A concurrent assertion can be written in the form; [name:] **assert property** (property_specfication) pass_statement [**else** fail_statement]. The pass fail messages are optional in concurrent assertions. Figure 4.3 is an example of a property assertion written checking sequential events.

```
property prop_cause_to_effect(check_clk,trigger,start_condition,effect,kill,TIMEOUT=1);
  @(trigger)disable iff(kill)
    start_condition |-> @(posedge check_clk)##TIMEOUT effect;
endproperty: prop_cause_to_effect
```

Figure 4.3. Example of a concurrent assertion property

The property definition “prop_cause_to_effect” is written to be reused. This means that only the first line in figure 4.3 is required when writing an assertion. The “trigger” is used to activate the assertion. This could be a signal, an event or a clock. The “kill” condition is used to disable the assertion during periods in simulation when the assertion is not required.

The implication construct(|->) allows a user to monitor sequences based on satisfying a pre-condition, and only evaluating the sequence when the condition is successful. On the left-hand side of the implication construct is the “start_condition” also known as the antecedent sequence expression. Evaluating the expression only occurs when the antecedent is true. On the right-hand side of the implication operator is the “effect” also known as the consequent sequence expression. For each successful match of the antecedent sequence expression the the consequent sequence expression is evaluated. The variable “TIMEOUT” defines the number of clock cycles after which the “effect” is expected and check_clk defines the clock.

There are two forms of the implication construct, overlapped operator (|->) and non-overlapped operator(|=>). For the overlapped implication operator, if there is a match for the antecedent sequence expression then the consequent sequence expression is evaluated on the same clock. For non-overlapped operator, the consequent sequence expression is evaluated on the next clock tick. To avoid any confusion it is always advised to use the overlapped operator with a defined number of clock ticks as is shown in figure 4.3.

```
CHK_RTC_ALARM1_EN: assert property(prop_cause_to_effect(clk_rtc,!alarm1_en,1,!alarm1,(rst_clk_rtc|cp_rtc_en),0))
```

Figure 4.4. Example of a concurrent assertion

The concurrent assertion CHK_RTC_ALARM1_EN in figure 4.4 uses the "prop_cause_to_effect" property. This is an assertion that checks that alarm1 of a real-time clock(RTC) is not triggered when alarm1_en is not set. The trigger is !alarm1_en and the kill condition is a combination of a rst_rtc_clk and cp_rtc_en ORed together. The antecedent is one, this means that the consequent is always evaluated when the assertion is triggered. The consequent is !alarm1 after zero clock ticks.

There are a few important points to remember when writing assertions. Firstly, the assertion must be efficient. Unless absolutely necessary, it is always advisable not to trigger an assertion at every edge of a clock . If a number of assertions are written in this way, it will cause multiple simulation timeouts. It is more time efficient to use an event as a trigger, that is triggered from an always block. Secondly, when evaluating the pass and fail rate of assertions it is important to distinguish passes from vacuous passes. Vacuous passes occur when an assertion is triggered and there is no match of the antecedent sequence expression. The assertion passes vacuously by returning true. Assertions need to examine for why the antecedent sequence expression has not been met. Lastly, assertion must sometimes be written in twos or even more to cover a sequence. For example, an assertion can be written to check that an alarm is triggered when alarm_en is set and another assertion is written to check that it is not triggered when alarm_en is disabled.

Coverage

Functional coverage is a measure of what features of the design have been exercised by the tests. This is useful for constrained random verification to know what features have been covered by a set of tests in a regression. Functional coverage in SystemVerilog samples variables in the testbench and analyzes if they have reached certain values.

How to write coverage

```
covergroup rtc_cg();
    cp_alarm1 : coverpoint alarm1 {
                    bins bin_alarm1_0={0};
                    bins bin_alarm1_1={1};
    }
endgroup
```

Figure 4.5. Example of a covergroup

Figure 4.4 shows a concurrent assertion used to check alarm1 for an RTC design. Figure 4.5 shows how a coverpoint for alarm1 is written. Coverpoints are put together in a covergroup block [33]. Variables from the DUT are mentioned as coverpoint, in this case the variable is alarm1 and the name of the coverpoint is cp_alarm1. Bins in a coverpoint are said to be hit or covered when the variable reaches the corresponding value. So, the bin_alarm1_0 is hit when alarm1 value is zero and bin_alarm1_1 is hit when alarm1 value is one.

```
covergroup rtc_cg();
    cp_alarm1 : coverpoint alarm1 {
                    bins bin_alarm1_0={0};
                    bins bin_alarm1_1={1};
    }
    cp_alarm1_en : coverpoint alarm1_en {
                    bins bin_alarm1_en_0={0};
                    bins bin_alarm1_en_1={1};
    }
    cross_cp_alarm1_X_cp_alarm1_en : cross alarm1, alarm1_en {
    }
endgroup
```

Figure 4.6. Example of cross coverage

Cross coverage is also possible in functional coverage. Cross coverage is specified between the coverpoints or variables and is defined using the cross construct. In figure

4.6, each coverpoint has two bins and therefore a cross between alarm1 and alarm1_en will yield four bins. Cross coverage becomes a useful tool in complex systems where it is important that a combination of functional points are verified [33].

CHAPTER FIVE
Lab 2

Advanced Peripheral Bus Protocol

The advanced peripheral bus (APB) interfaces to any peripheral that is low-bandwidth and does not require the high performance of a pipelined bus interface. It is connected to the system bus via a bridge. There is a single bus master on the APB, thus there is no need for an arbiter. The master drives the address and write buses, and also performs a combinatorial decode of the address to decide the type of transfer. It is also responsible for driving the enable signal to time the transfer and driving the APB data onto the system bus during a read transfer.

APB finite state machine (FSM) has three operating states, Idle, Setup and Access. The Idle state is the default state of the APB. When a transfer is required the bus protocol moves in to the setup state where the appropriate select signal (PSELx) is asserted. The bus only remains in this state for one clock cycle and always moves to the Access state on the next rising edge of the clock. In the Access state the enable signal(PENABLE) is asserted. The address(PADDR), write(PWRITE), select(PSELx) and write data (PWDATA) must remain stable during the transition from the Setup to Access state. Exit from the Access state is controlled by the ready signal(PREADY) from the slave. If the ready signal is held low by the slave then the peripheral bus remains in the Access state. If the signal is driven high by the slave then the Access state is exited and bus will return to the Idle state state and remain in that state if no further transfers are required. Alternatively, the bus moves directly to the Setup state if another transfer follows.

APB Transfer

There are 3 types of transfers used in the APB protocol; read, write and error response. There are two types of read and write transfers, write and read with no Wait state and read and write with a Wait state. For the purpose of this lab the read and write transactions are implemented without any Wait states.

The following steps are used to make an APB Write with no wait state

- The write transfer starts with the address, write data, write signal and select signal changing after the rising edge of the clock.
- The first clock cycle of the write transfer is the Setup phase.
- After that clock edge the enable signal is asserted which indicates that the Access phase is taking place.
- The address, data and control signals all remain valid during this phase and transfer completes at the end of the clock cycle.
- The enable signal is then de-asserted and the select signal also goes low unless the transfer is immediately followed up by another transfer to the same peripheral.

The following steps are used to make an APB Read with no waits state;

- The read transfer starts with the address, write data, write signal and select signal all changing after the rising edge of a clock.
- Like the write transfer, the first clock cycle of the read transfer is also the Setup phase.
- After the clock edge the enable signal is asserted to indicate the transfer is in the Access phase.

- The address, data and control signals all remain valid during this phase. The slave must provide the data before the end of the read transfer. The transfer completes at the end of this cycle.
- The enable signal is de-asserted at the end of the transfer.

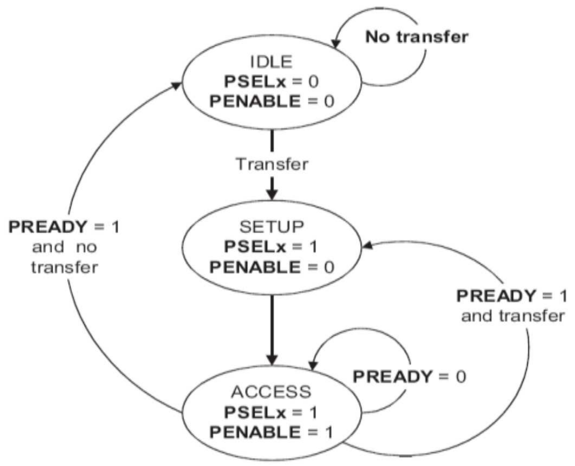


Figure 5.1. APB FSM state diagram

Verification Plan

Verification Strategy

The stimulus to the APB FSM block includes paddr, pwdata and pwrite. Each stimulus is used to illicit a response from the block that can be checked with a scoreboard and assertions. Verification of this block will be done at block level.

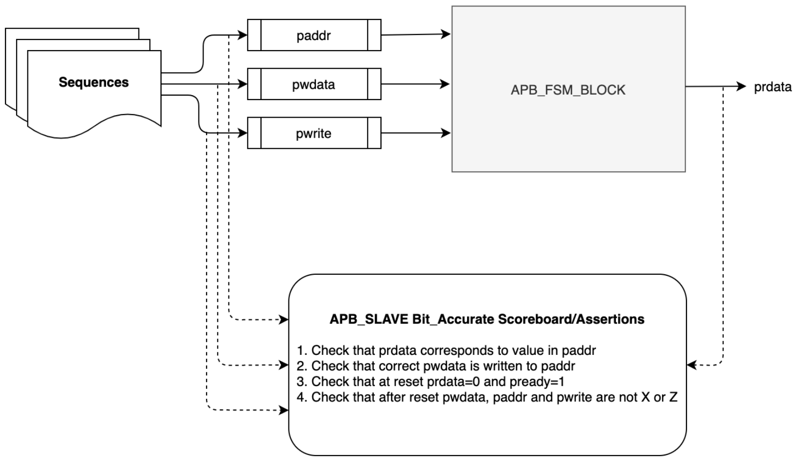

Sequences
paddr
pwdata
pwrite
APB_FSM_BLOCK
prdata
APB_SLAVE Bit_Accurate Scoreboard/Assertions
1. Check that prdata corresponds to value in paddr
2. Check that correct pwdata is written to paddr
3. Check that at reset prdata=0 and pready=1
4. Check that after reset pwdata, paddr and pwrite are not X or Z

Figure 5.2. Diagram used to illustrate the verification strategy

Features to Verify

APB block operates in reset and enable mode.

- APB FSM reset
- APB FSM read transaction
- APB FSM write transaction

Stimulus and Randomization

The sequence shall generate the following type of signals with constrained random values; address and data. The pwrite value shall be randomized as an enumerated data type.

Table 5.1. Checkers used to test features of the ABP FSM DUT

Checker scope for the verification plan

Checker	Checker Type
Check that when pwrite is set, correct data in pwdata is written to paddr	Scoreboard
Check that when pwrite is zero, data in prdata is the same as paddr	Scoreboard/Assertion
Check at reset prdata=0 and pready=1	Assertion
Check that reset, psel and penable are not X or Z values after a defined simulation period	Assertion
Check that when pwrite is high, value in prdata is zero	Assertion
Check that when psel is active, penable goes high on the next clock cycle	Assertion
Check that pwdata is not X or Z from the rising edge to the falling edge of pwrite	Assertion
Cover address and data sizes	Coverage
Cross pwrite with data and address	Coverage

Testbench Architecture

The testbench architecture for lab two is similar to lab one in many ways. New features added to class components are explained in this section. The steps for creating each testbench component remains relatively the same.

DUT

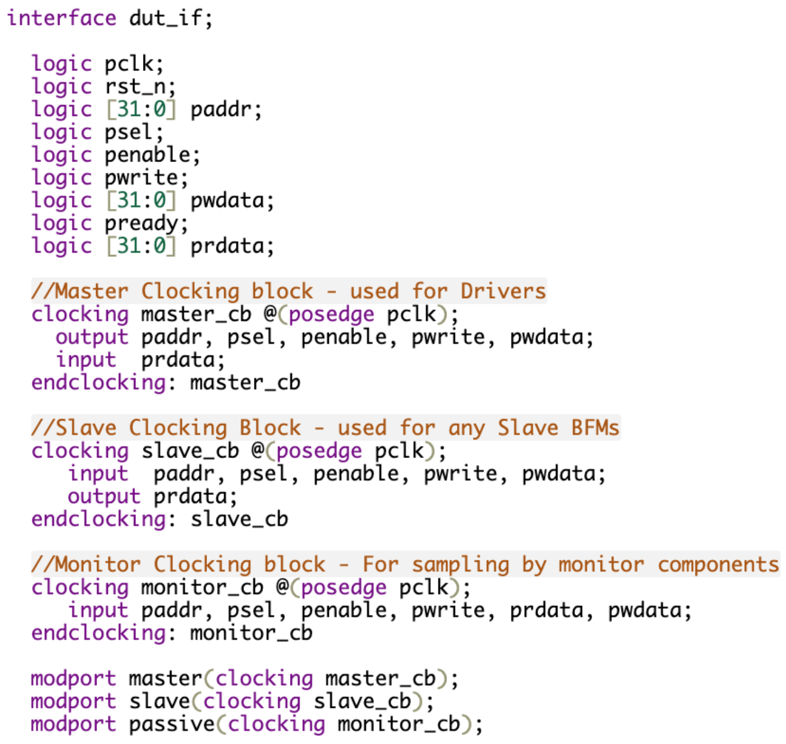
The DUT is a simple APB FSM state machine. The FSM state machine has a SETUP, R_ENABLE and W_ENABLE states. APB read takes place in the SETUP state and APB write in the W_ENABLE state.

```
module apb_slave(dut_if dif);

  logic [31:0] mem [0:256];
  logic [1:0] apb_st;
  const logic [1:0] SETUP=0;
  const logic [1:0] W_ENABLE=1;
  const logic [1:0] R_ENABLE=2;

  always @(posedge dif.pclk or negedge dif.rst_n) begin
    if (dif.rst_n==0) begin
      apb_st <=0;
      dif.prdata <=0;
      dif.pready <=1;
      for(int i=0;i<256;i++) mem[i]=i;
    end
    else begin
      case (apb_st)
        SETUP: begin
          dif.prdata <= 0;
          if (dif.psel && !dif.penable) begin
            if (dif.pwrite) begin
              apb_st <= W_ENABLE;
            end
            else begin
              dif.prdata <= mem[dif.paddr];
              apb_st <= R_ENABLE;
            end
          end
        end
        W_ENABLE: begin
          if (dif.psel && dif.penable && dif.pwrite) begin
            mem[dif.paddr] <= dif.pwdata;
          end
          apb_st <= SETUP;
        end
        R_ENABLE: begin
          apb_st <= SETUP;
        end
      endcase
    end
  end

endmodule
```

Figure 5.3. APB DUT

APB Interface

The APB interface has the collection of signal inputs and outputs found in the DUT. There is a clocking bus for the master, slave and monitor. The clocking bus for the APB slave is used if a slave BFM is added to this testbench.



```
interface dut_if;

  logic pclk;
  logic rst_n;
  logic [31:0] paddr;
  logic psel;
  logic penable;
  logic pwrite;
  logic [31:0] pwdata;
  logic pready;
  logic [31:0] prdata;

  //Master Clocking block - used for Drivers
  clocking master_cb @(posedge pclk);
    output paddr, psel, penable, pwrite, pwdata;
    input  prdata;
  endclocking: master_cb

  //Slave Clocking Block - used for any Slave BFMs
  clocking slave_cb @(posedge pclk);
     input  paddr, psel, penable, pwrite, pwdata;
     output prdata;
  endclocking: slave_cb

  //Monitor Clocking block - For sampling by monitor components
  clocking monitor_cb @(posedge pclk);
     input paddr, psel, penable, pwrite, prdata, pwdata;
  endclocking: monitor_cb

  modport master(clocking master_cb);
  modport slave(clocking slave_cb);
  modport passive(clocking monitor_cb);
```

Figure 5.4. APB interface

The interface is also a good place to write assertions that will be used to check for some functionality of the design. Figure 5.5 shows the different assertions used to check the functionality of the design.

```
// event used to trigger CHK_APB_PRDATA_ON_PWRITE assertion
event ev_pwrite_high;
always @(pclk)begin
  if(pwrite)
      ->ev_pwrite_high;
end

// Checks that when pwrite is high prdata is zero
property prop_apb_prdata_on_pwrite();
  @(ev_pwrite_high)disable iff(!rst_n)
      1 |-> @(posedge pclk)##0 prdata==32'd0;
endproperty: prop_apb_prdata_on_pwrite

CHK_APB_PRDATA_ON_PWRITE: assert property(prop_apb_prdata_on_pwrite);

// Checks that when psel is active, penable goes high on the next clock cycle
property prop_apb_psel_on_penable();
  @(posedge psel)disable iff(!rst_n)
      1 |-> @(negedge pclk)##1 penable;
endproperty: prop_apb_psel_on_penable

CHK_APB_PSEL_ON_PENABLE: assert property(prop_apb_psel_on_penable);

// Checks the safe state of a signal at posedge of a trigger
// Takes into account being a Z at the start of simulation
property prop_apb_signal_valid_posedge(trigger,signal,clock,timeout);
  @(posedge trigger)
      1 |-> @(negedge clock)##timeout !$isunknown(signal);
endproperty:  prop_apb_signal_valid_posedge

CHK_APB_RESET_VALID:     assert property(prop_apb_signal_valid_posedge(pclk,rst_n,pclk,0));
CHK_APB_PSEL_VALID:      assert property(prop_apb_signal_valid_posedge(pclk,psel,pclk,0));
CHK_APB_PENABLEL_VALID:  assert property(prop_apb_signal_valid_posedge(pclk,penable,pclk,0));

// Checks the value of prdata = paddr
// Used to catch data mismatches by scoreboard
property prop_apb_prdata();
  @(negedge psel)disable iff(!rst_n)
      !pwrite |-> @(negedge pclk)##0 prdata==paddr;
endproperty: prop_apb_prdata

CHK_APB_PRDATA: assert property(prop_apb_prdata);

// Checks that pwdata is valid from the rising edge to the falling edge of pwrite
property prop_apb_pwrite_valid();
  @(posedge pwrite)disable iff(!rst_n)
      1 |-> !$unknown(pwdata) [*1:$] ##1 $fell(pwrite);
endproperty: prop_apb_pwrite_valid

CHK_APB_PWDATA_VALID: assert property(prop_apb_pwrite_valid);

// Checks safe state of a signal at the negedge of the trigger
property prop_apb_signal_valid_negedge(trigger,signal,clock,timeout);
  @(negedge trigger)
  1 |-> @(posedge clock)##timeout !$isunknown(signal);
endproperty:  prop_apb_signal_valid_negedge

CHK_APB_PRDATA_VALID:    assert property(prop_apb_signal_valid_negedge(pwrite,prdata,pclk,0));
```

Figure 5.5. APB assertions

APB Transaction

```
class apb_transaction extends uvm_sequence_item;

  `uvm_object_utils(apb_transaction)


  typedef enum {READ, WRITE} kind_e; // typedef read/write transaction type
  rand bit [31:0] addr;              // address
  rand bit [31:0] data;              // data for write or read response
  rand kind_e  pwrite;               // pwrite command type

  constraint addr_c{
    addr inside {[32'd0:32'd256]};
  }
  constraint data_c{
    data inside {[32'd0:32'd256]};

  }
  constraint pwrite_c{
    pwrite dist {READ:=8, WRITE:=4};
  }

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

  // prints transaction information
  function string convert2string();
    return $psprintf("pwrite=%s paddr=%0h data=%0h",pwrite,addr,data);
  endfunction

endclass
```

Figure 5.6. APB transaction

The data and address random variables are restricted to a range of values using the “inside” SystemVerilog construct. The pwrite random variable is specified as weighted distribution using the “dist” Systemverilog construct. The value with more weight will get allocated more often to the random variable. The := operator assigns the specified weight to the item, the item can be a single value or a range of values. The possibility that pwrite will be assigned the value “READ” is 4 times more likely than the value “WRITE”.

The function convert2string is a new addition which was left out in lab 1. It is recommended to implement this function which returns a string representation of the transaction. It is useful for printing debug information to the simulator transcript or log file.

```
class apb_sequence extends uvm_sequence#(apb_transaction);

  `uvm_object_utils(apb_sequence)

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

  task body();
    apb_transaction rw_trans;
    repeat (80) begin
      rw_trans=new();
      start_item(rw_trans);
      assert(rw_trans.randomize());
      finish_item(rw_trans);
    end
  endtask
endclass
```

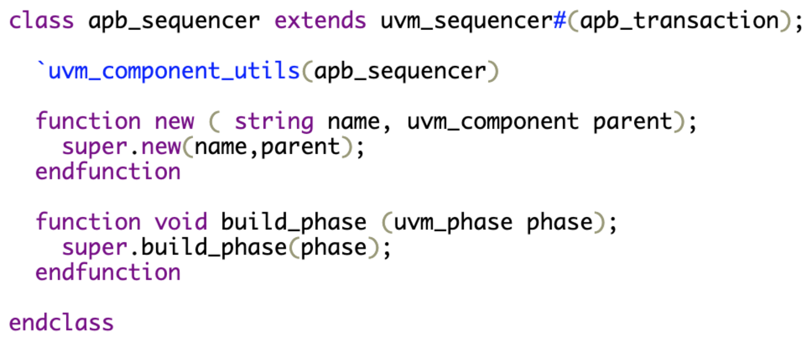
Figure 5.7. APB sequence

The APB stimulus is made up of the apb_sequence and apb_direct_sequence. The first sequence is used to generate random sequence items and will be sent to the driver. It is similar to the sequence generated in lab 1 using start_item and finish_item. The second sequence in figure 5.7 is used to generate fixed values for the data and address. This sequence is created using the macro uvm_do_with which randomizes data and address using the inline constraints provided in the curly brackets. This is useful to check that the DUT performs reads and writes as intended.

```
class apb_direct_sequence extends uvm_sequence#(apb_transaction);

  `uvm_object_utils(apb_direct_sequence)

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

  virtual task body();
    `uvm_do_with(req, {req.addr==32'h64;  req.data==32'hc8;  req.pwrite==READ;})
    `uvm_do_with(req, {req.addr==32'h32;  req.data==32'h20;  req.pwrite==READ;})
    `uvm_do_with(req, {req.addr==32'h70;  req.data==32'h50;  req.pwrite==WRITE;})
    `uvm_do_with(req, {req.addr==32'h00;  req.data==32'h90;  req.pwrite==WRITE;})
    `uvm_do_with(req, {req.addr==32'h99;  req.data==32'h40;  req.pwrite==READ;})
  endtask: body

endclass
```

Figure 5.8. APB direct sequence

APB Sequencer

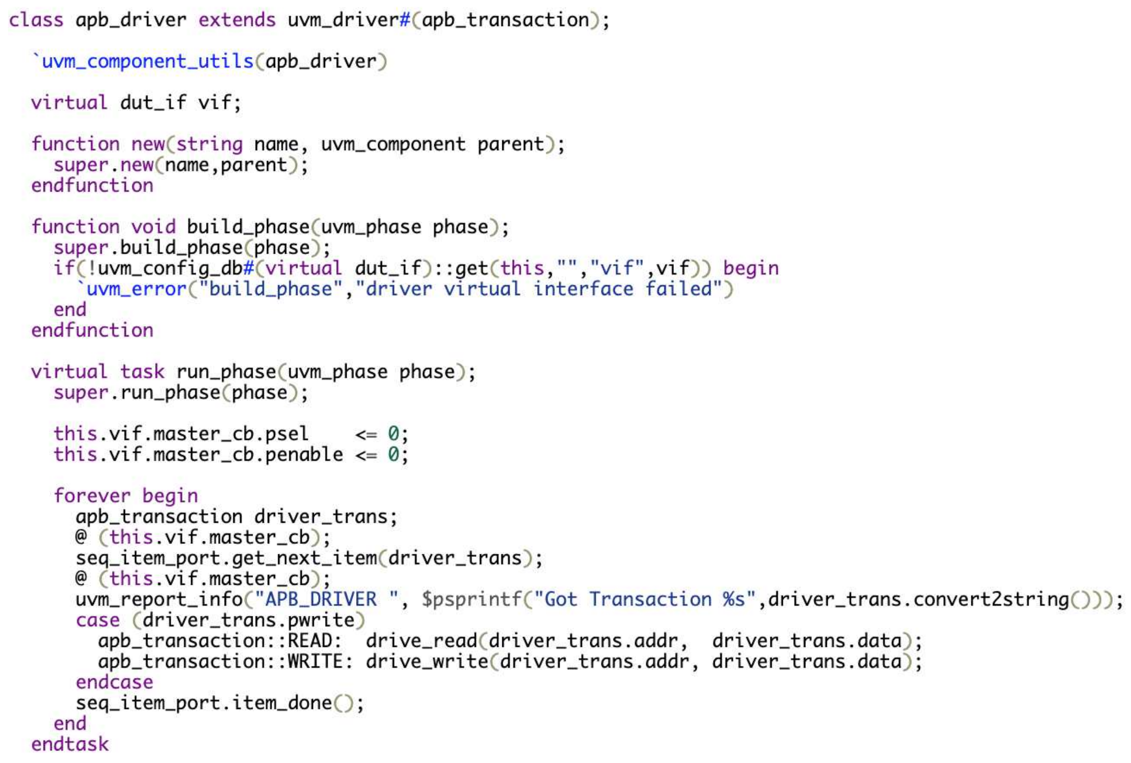


```
class apb_sequencer extends uvm_sequencer#(apb_transaction);

  `uvm_component_utils(apb_sequencer)

  function new ( string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase (uvm_phase phase);
    super.build_phase(phase);
  endfunction

endclass
```

Figure 5.9. APB sequencer

The sequencer remains the same as the sequencer in lab1 and as already highlighted before, UVM takes care of the underlying implementation of the sequencer.

APB Driver



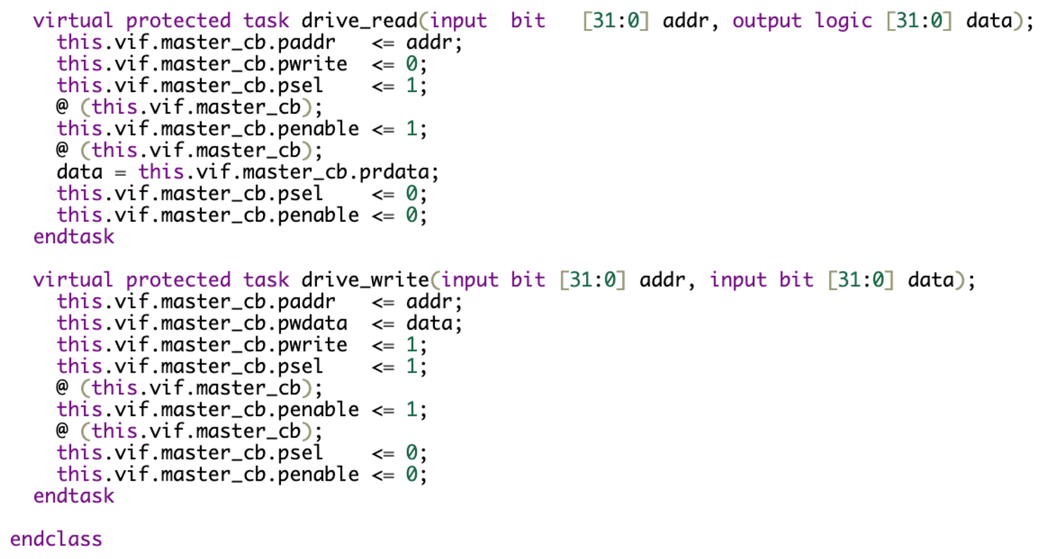
```
class apb_driver extends uvm_driver#(apb_transaction);

  `uvm_component_utils(apb_driver)

  virtual dut_if vif;

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual dut_if)::get(this,"","vif",vif)) begin
      `uvm_error("build_phase","driver virtual interface failed")
    end
  endfunction

  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);

    this.vif.master_cb.psel    <= 0;
    this.vif.master_cb.penable <= 0;

    forever begin
      apb_transaction driver_trans;
      @ (this.vif.master_cb);
      seq_item_port.get_next_item(driver_trans);
      @ (this.vif.master_cb);
      uvm_report_info("APB_DRIVER ", $psprintf("Got Transaction %s",driver_trans.convert2string()));
      case (driver_trans.pwrite)
        apb_transaction::READ:  drive_read(driver_trans.addr,  driver_trans.data);
        apb_transaction::WRITE: drive_write(driver_trans.addr, driver_trans.data);
      endcase
      seq_item_port.item_done();
    end
  endtask
```

Figure 5.10. Code snippet 1 of APB driver

The APB driver is implemented using the non-pipelined model. The driver is designed to model only one transaction at a time. In this case, sequence sends one transaction to the driver and driver might take several clock cycles (based on the interface protocol) to finish driving the transaction. Only after that will the driver accept a new transaction from the sequencer. This is the same method used to implement the driver in lab 1. It is important to mention this because some interface protocols follow a pipelined model. The driver would be required to drive more than one active transaction at a time. In this case sequence can keep sending transactions to the driver without waiting for driver to complete the previous transaction. A good example would be the advanced high performance bus (AHB) lite protocol.

The APB driver has all the same components that the adder driver has in lab 1. The only difference is in how the transaction is driven to the DUT. At the beginning of the run phase the psel and penable signals are reset. This is to ensure that the DUT enters into the Idle state. After one clock cycle a transaction is received from the sequencer and its contents reported by the uvm_report_info. The uvm_report_info has the following format, uvm_info(ID,MSG,VERBOSITY). The "APB_DRIVER" is the ID and $sprintf("Got Transaction %s, driver_trans.covert2string()" is the message. If the verbosity is not specified as is the case in figure 7, UVM_LOW is used as the default verbosity.

The case statement is used to decode whether the transaction contains a read or write depending on the value of pwrite. If pwrite is 1, the transaction specifies a write and if 0 the transaction specifies a read.



```
  virtual protected task drive_read(input  bit   [31:0] addr, output logic [31:0] data);
    this.vif.master_cb.paddr   <= addr;
    this.vif.master_cb.pwrite  <= 0;
    this.vif.master_cb.psel    <= 1;
    @ (this.vif.master_cb);
    this.vif.master_cb.penable <= 1;
    @ (this.vif.master_cb);
    data = this.vif.master_cb.prdata;
    this.vif.master_cb.psel    <= 0;
    this.vif.master_cb.penable <= 0;
  endtask

  virtual protected task drive_write(input bit [31:0] addr, input bit [31:0] data);
    this.vif.master_cb.paddr   <= addr;
    this.vif.master_cb.pwdata  <= data;
    this.vif.master_cb.pwrite  <= 1;
    this.vif.master_cb.psel    <= 1;
    @ (this.vif.master_cb);
    this.vif.master_cb.penable <= 1;
    @ (this.vif.master_cb);
    this.vif.master_cb.psel    <= 0;
    this.vif.master_cb.penable <= 0;
  endtask

endclass
```

Figure 5.11. Code snippet 2 of APB driver

The read task is defined as protected. If a task is protected, then it will be available in the inherited class but it cannot be publicly accessed. The following are steps to carry out an APB read:

(1) Drive address to DUT.

(2) Set pwrite to 0 and psel to 1 (read)

(3) Set penable to 1

(4) Retrieve value from prdata and assign it to the output data

(5) Set psel and penable to 0

The write task is also protected. The following are steps to carry out an APB write:

(1) Drive address and data to DUT

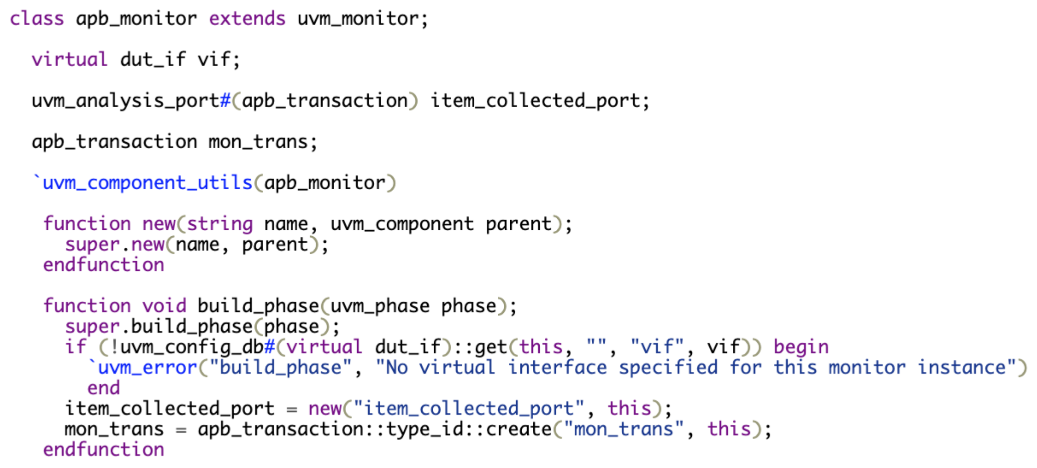
(2) Set pwrite and psel to 1 (write)

(3) Set penable to 1

(4) Set psel and penable to 0

The aforementioned steps are important to for accessing all the states in the APB FSM state machine. Care must be taken when injecting transactions to a DUT that follows a specific sequence of events.

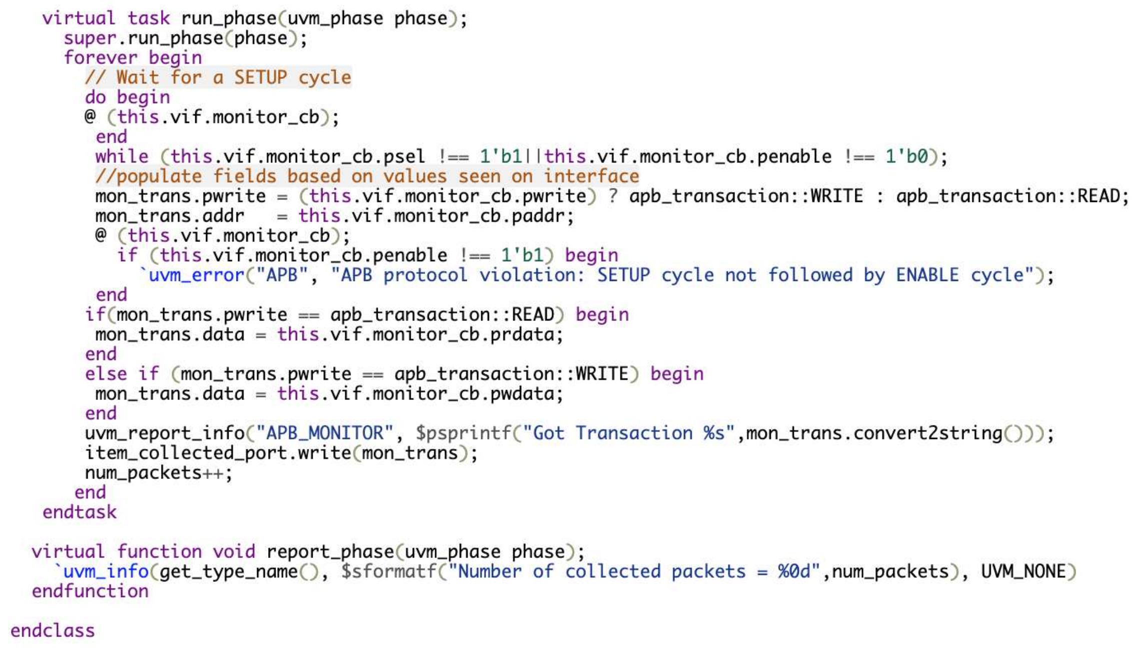
APB Monitor



```
class apb_monitor extends uvm_monitor;

  virtual dut_if vif;

  uvm_analysis_port#(apb_transaction) item_collected_port;

  apb_transaction mon_trans;

  `uvm_component_utils(apb_monitor)

   function new(string name, uvm_component parent);
     super.new(name, parent);
   endfunction

   function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     if (!uvm_config_db#(virtual dut_if)::get(this, "", "vif", vif)) begin
       `uvm_error("build_phase", "No virtual interface specified for this monitor instance")
       end
     item_collected_port = new("item_collected_port", this);
     mon_trans = apb_transaction::type_id::create("mon_trans", this);
   endfunction
```

Figure 5.12. Code snippet 1 of APB monitor

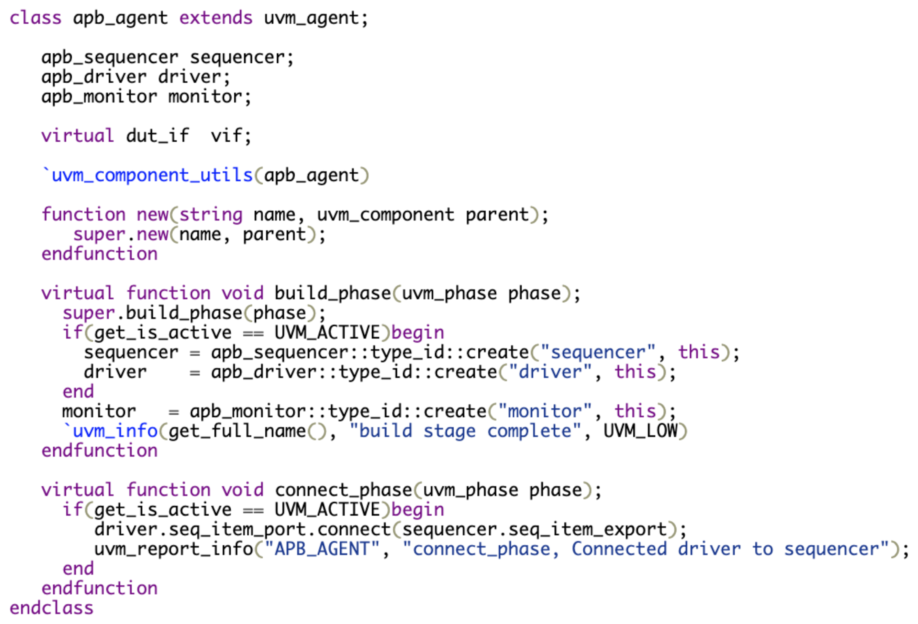
The APB monitor has the same components as the monitor in lab 1. The important components to remember in the monitor are the UVM analysis port and the virtual interface.



```
  virtual task run_phase(uvm_phase phase);
    super.run_phase(phase);
    forever begin
      // Wait for a SETUP cycle
      do begin
      @ (this.vif.monitor_cb);
       end
       while (this.vif.monitor_cb.psel !== 1'b1||this.vif.monitor_cb.penable !== 1'b0);
       //populate fields based on values seen on interface
       mon_trans.pwrite = (this.vif.monitor_cb.pwrite) ? apb_transaction::WRITE : apb_transaction::READ;
       mon_trans.addr   = this.vif.monitor_cb.paddr;
       @ (this.vif.monitor_cb);
         if (this.vif.monitor_cb.penable !== 1'b1) begin
           `uvm_error("APB", "APB protocol violation: SETUP cycle not followed by ENABLE cycle");
       end
      if(mon_trans.pwrite == apb_transaction::READ) begin
       mon_trans.data = this.vif.monitor_cb.prdata;
      end
      else if (mon_trans.pwrite == apb_transaction::WRITE) begin
       mon_trans.data = this.vif.monitor_cb.pwdata;
      end
      uvm_report_info("APB_MONITOR", $psprintf("Got Transaction %s",mon_trans.convert2string()));
      item_collected_port.write(mon_trans);
      num_packets++;
     end
  endtask

 virtual function void report_phase(uvm_phase phase);
   `uvm_info(get_type_name(), $sformatf("Number of collected packets = %0d",num_packets), UVM_NONE)
 endfunction

endclass
```

Figure 5.13. Code snippet 2 of APB monitor

At the beginning of the run phase, the monitor waits for when psel is not equal to one or when penable is not equal to zero. This is ensures that the monitor waits for the SETUP state in the FSM state machine. When one of the conditions in the while loop is satisfied, the fields in monitor are populated with the values seen on the interface. The value of address is written to the address field in monitor and the ternary operator is used to populate the pwrite field. A check of penable is carried out to ensure that an APB protocol violation has not take place. The data field is populated depending on the value of pwrite. If pwrite is one, the data field is populated with the value of pwdata and if pwrite is zero, the data field is populated with the value of prdata. The function convert2string is used to display the transaction that has been acquired from the DUT. A variable num_packets is incremented after every transaction and is used to monitor the number of transactions generated by the APB sequence. The report_phase is used to display the number of packets.

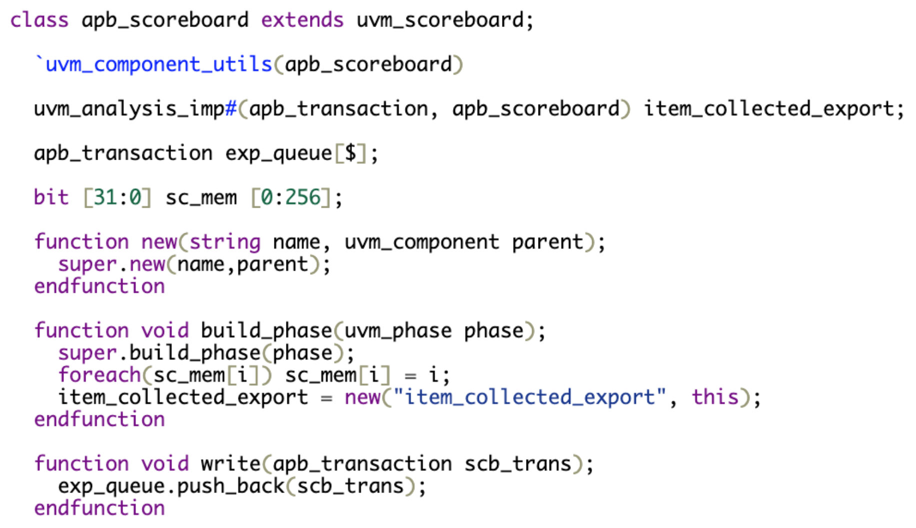


```
class apb_agent extends uvm_agent;

   apb_sequencer sequencer;
   apb_driver driver;
   apb_monitor monitor;

   virtual dut_if  vif;

   `uvm_component_utils(apb_agent)

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   virtual function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     if(get_is_active == UVM_ACTIVE)begin
       sequencer = apb_sequencer::type_id::create("sequencer", this);
       driver    = apb_driver::type_id::create("driver", this);
     end
     monitor   = apb_monitor::type_id::create("monitor", this);
     `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
   endfunction

   virtual function void connect_phase(uvm_phase phase);
     if(get_is_active == UVM_ACTIVE)begin
        driver.seq_item_port.connect(sequencer.seq_item_export);
        uvm_report_info("APB_AGENT", "connect_phase, Connected driver to sequencer");
     end
   endfunction
endclass
```

Figure 5.14. APB agent

The APB agent has the usual components like the sequencer, driver and monitor. The most important part to remember for the agent is to connect the sequencer to the driver.

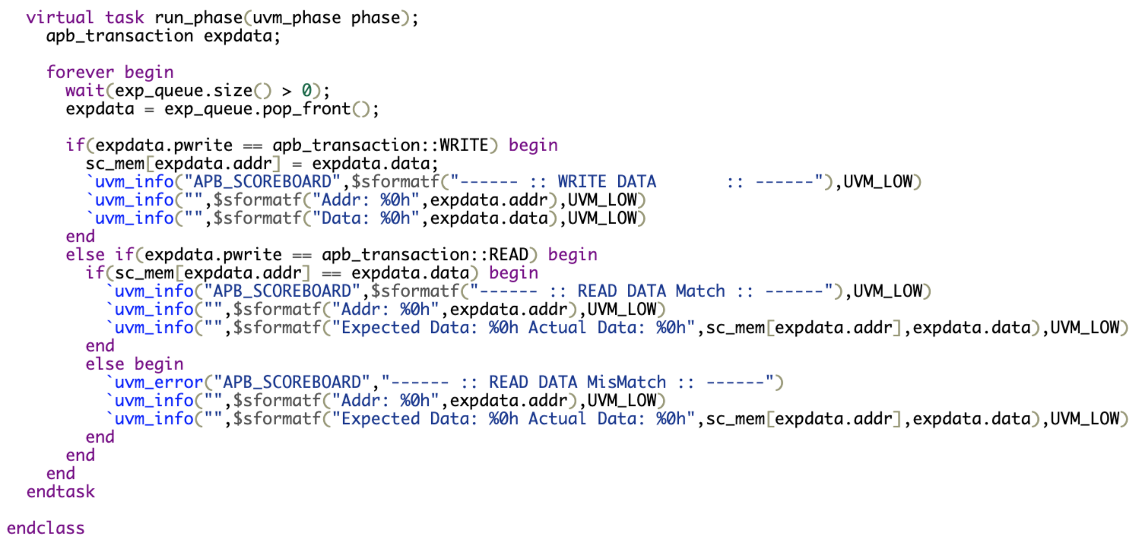
APB Scoreboard



```
class apb_scoreboard extends uvm_scoreboard;

  `uvm_component_utils(apb_scoreboard)

  uvm_analysis_imp#(apb_transaction, apb_scoreboard) item_collected_export;

  apb_transaction exp_queue[$];

  bit [31:0] sc_mem [0:256];

  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    foreach(sc_mem[i]) sc_mem[i] = i;
    item_collected_export = new("item_collected_export", this);
  endfunction

  function void write(apb_transaction scb_trans);
    exp_queue.push_back(scb_trans);
  endfunction
```

Figure 5.15. Code snippet 1 of APB scoreboard

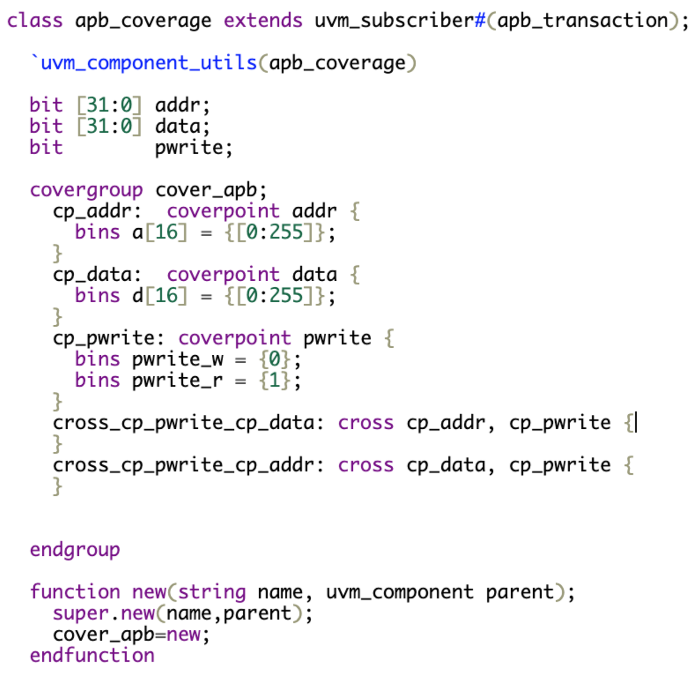
The APB scoreboard has most of the same components that the adder scoreboard has. The APB scoreboard has a queue that is used to store transactions received from the monitor. It has memory that is used to check if the correct data and address values are received from the DUT. In the build phase the uvm_analysis_imp is created using the constructor. In figure 5.16, the scoreboard waits for a transaction in the queue and then checks if the transaction is a write or a read. If is a read, the scoreboard checks that data in the address is the same as the data in prdata. If it is a write transaction the scoreboard prints the information of pwdata and paddr. The contents of this transaction are not checked in the scoreboard because this testbench does not include a slave bus functional model and a slave DUT. The convert2string function is used to print the transactions sent to the driver and collected by the monitor against those reported by the scoreboard.



```
  virtual task run_phase(uvm_phase phase);
    apb_transaction expdata;

    forever begin
      wait(exp_queue.size() > 0);
      expdata = exp_queue.pop_front();

      if(expdata.pwrite == apb_transaction::WRITE) begin
        sc_mem[expdata.addr] = expdata.data;
        `uvm_info("APB_SCOREBOARD",$sformatf("------ :: WRITE DATA       :: ------"),UVM_LOW)
        `uvm_info("",$sformatf("Addr: %0h",expdata.addr),UVM_LOW)
        `uvm_info("",$sformatf("Data: %0h",expdata.data),UVM_LOW)
      end
      else if(expdata.pwrite == apb_transaction::READ) begin
        if(sc_mem[expdata.addr] == expdata.data) begin
          `uvm_info("APB_SCOREBOARD",$sformatf("------ :: READ DATA Match :: ------"),UVM_LOW)
          `uvm_info("",$sformatf("Addr: %0h",expdata.addr),UVM_LOW)
          `uvm_info("",$sformatf("Expected Data: %0h Actual Data: %0h",sc_mem[expdata.addr],expdata.data),UVM_LOW)
        end
        else begin
          `uvm_error("APB_SCOREBOARD","------ :: READ DATA MisMatch :: ------")
          `uvm_info("",$sformatf("Addr: %0h",expdata.addr),UVM_LOW)
          `uvm_info("",$sformatf("Expected Data: %0h Actual Data: %0h",sc_mem[expdata.addr],expdata.data),UVM_LOW)
        end
      end
    end
  endtask

endclass
```

Figure 5.16. Code snippet 2 of APB scoreboard

APB Coverage



```
class apb_coverage extends uvm_subscriber#(apb_transaction);

  `uvm_component_utils(apb_coverage)

  bit [31:0] addr;
  bit [31:0] data;
  bit        pwrite;

  covergroup cover_apb;
    cp_addr:  coverpoint addr {
      bins a[16] = {[0:255]};
    }
    cp_data:  coverpoint data {
      bins d[16] = {[0:255]};
    }
    cp_pwrite: coverpoint pwrite {
      bins pwrite_w = {0};
      bins pwrite_r = {1};
    }
    cross_cp_pwrite_cp_data: cross cp_addr, cp_pwrite {
    }
    cross_cp_pwrite_cp_addr: cross cp_data, cp_pwrite {
    }


  endgroup

  function new(string name, uvm_component parent);
    super.new(name,parent);
    cover_apb=new;
  endfunction
```

Figure 5.17. APB coverage

The coverage object apb_coverage is derived from the uvm_subscriber class and is parameterized with the apb_transaction. Since the object is of type uvm_subscriber, it has an analysis_export and can implement the write function. This APB coverage class has a typical covergroup with coverpoint of the elements in the transaction. The covergroup cover_apb is created using the class constructor and data, address and pwrite are defined as coverpoints with bins. If all the bins are hit, the coverpoint is said to be covered. There is also cross coverage between cp_pwrite and cp_data, and cp_write and cp_addr. This coverage is a simple example that can be used as a building block and coverage is much larger and complicated for ASIC being manufactured today. The coverage class must be instantiated in the environment and the connect function is used to enable communication between the analysis port and export. Modern simulators often provide coverage analysis tools to examine the holes that might have been missed. These holes can be covered by adding to the test library.

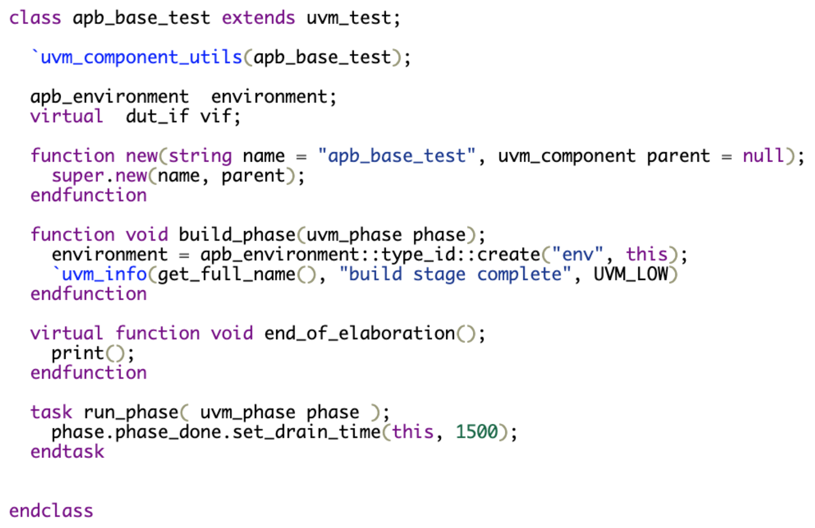
APB Environment

```
class apb_environment  extends uvm_env;

  `uvm_component_utils(apb_environment);

   apb_agent  agent;
   apb_scoreboard scoreboard;
   apb_coverage   coverage;

   function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction

   function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     agent      = apb_agent::type_id::create("agent", this);
     scoreboard = apb_scoreboard::type_id::create("scoreboard", this);
     coverage   = apb_coverage::type_id::create("coverage",this);
   endfunction

   function void connect_phase(uvm_phase phase);
     super.connect_phase(phase);
     agent.monitor.item_collected_port.connect(scoreboard.item_collected_export);
     agent.monitor.item_collected_port.connect(coverage.analysis_export);
   endfunction
endclass
```

Figure 5.18. APB environment

The class APB environment is container that holds the agent, scoreboard and coverage. As previously mentioned the coverage class should be instantiated in the environment. The instantiation is done in the connect phase by connecting the item_collected_port(from apb_monitor) to the analysis_export.

APB Base Test



```
class apb_base_test extends uvm_test;

  `uvm_component_utils(apb_base_test);

  apb_environment  environment;
  virtual  dut_if vif;

  function new(string name = "apb_base_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    environment = apb_environment::type_id::create("env", this);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction

  virtual function void end_of_elaboration();
    print();
  endfunction

  task run_phase( uvm_phase phase );
    phase.phase_done.set_drain_time(this, 1500);
  endtask


endclass
```

Figure 5.19. APB base test

The APB base has several elements required by the random and direct tests that derived from it. It has an object environment created in the build phase and drain time is set in the run phase.

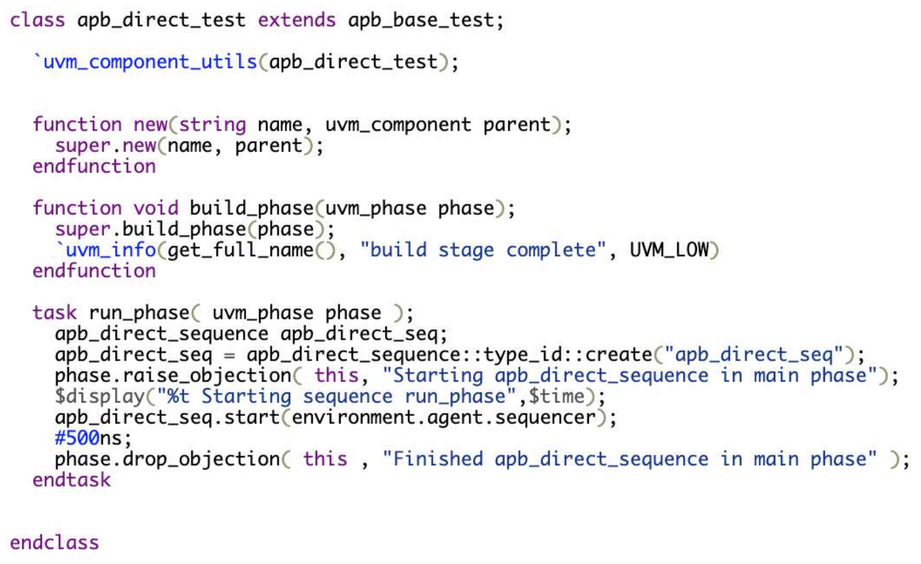
APB Random Test

```
class apb_random_test extends apb_base_test;

  `uvm_component_utils(apb_random_test);

  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction

  task run_phase( uvm_phase phase );
    apb_sequence apb_seq;
    apb_seq = apb_sequence::type_id::create("apb_seq");
    phase.raise_objection( this, "Starting apb_base_seq in main phase");
    $display("%t Starting sequence apb_seq run_phase",$time);
    apb_seq.start(environment.agent.sequencer);
    #500ns;
    phase.drop_objection( this , "Finished apb_seq in main phase");
  endtask

endclass
```

Figure 5.20. APB random test

The class apb_random_test is derived from the apb_base_test. In the run phase of this test, a handle to the sequence object apb_sequence is declared. The class apb_sequence simply creates random apb transactions that are sent to the sequencer. A objection is raised and dropped to communicate when it is safe to end a phase.

APB Direct Test



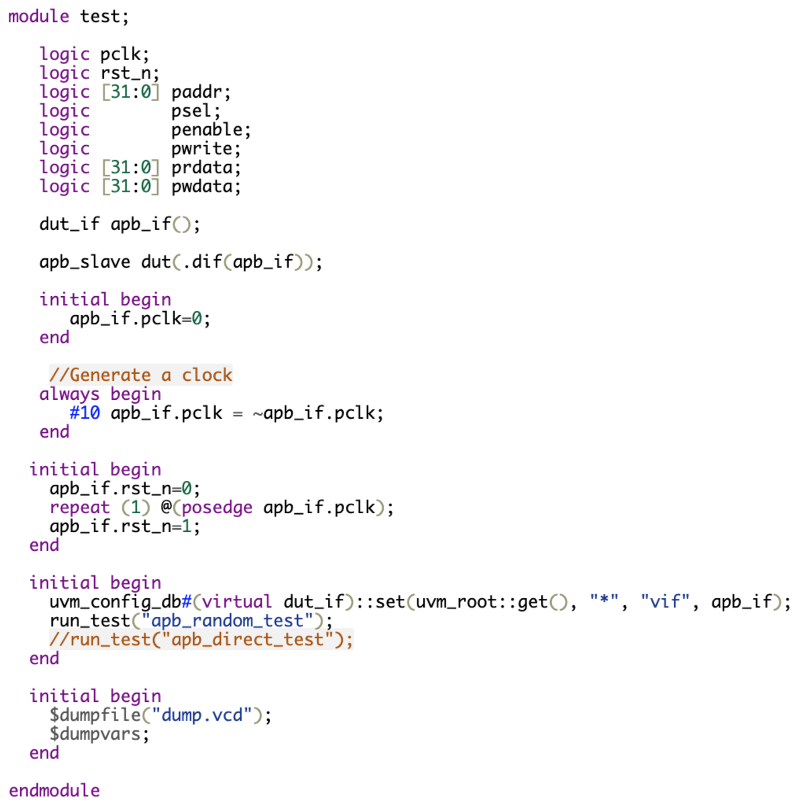
```
class apb_direct_test extends apb_base_test;

  `uvm_component_utils(apb_direct_test);


  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info(get_full_name(), "build stage complete", UVM_LOW)
  endfunction

  task run_phase( uvm_phase phase );
    apb_direct_sequence apb_direct_seq;
    apb_direct_seq = apb_direct_sequence::type_id::create("apb_direct_seq");
    phase.raise_objection( this, "Starting apb_direct_sequence in main phase");
    $display("%t Starting sequence run_phase",$time);
    apb_direct_seq.start(environment.agent.sequencer);
    #500ns;
    phase.drop_objection( this , "Finished apb_direct_sequence in main phase" );
  endtask


endclass
```

Figure 5.21. APB direct test

The class apb_direct_test is also derived from the apb_base_test. In the run phase of this test, a handle to the sequence object apb_direct_sequence is declared. The class apb_direct_sequence creates transaction with fixed values for data and address. This test could be used as bring up test because the transaction is made up of known values of address and data and not randomized values.

Testbench Top

The testbench top has the same features as the testbench top in lab 1. The run_test is called twice for the apb_random_test and the apb_direct_test.

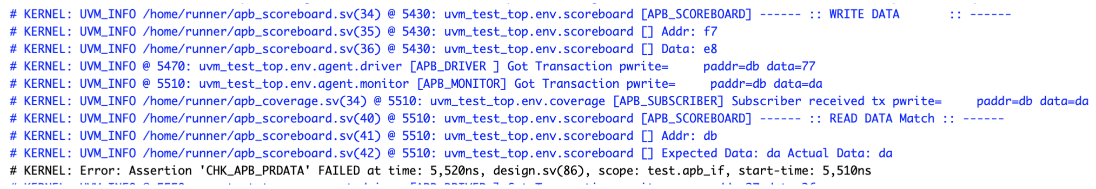


```
module test;

   logic pclk;
   logic rst_n;
   logic [31:0] paddr;
   logic        psel;
   logic        penable;
   logic        pwrite;
   logic [31:0] prdata;
   logic [31:0] pwdata;

   dut_if apb_if();

   apb_slave dut(.dif(apb_if));

   initial begin
      apb_if.pclk=0;
   end

    //Generate a clock
   always begin
      #10 apb_if.pclk = ~apb_if.pclk;
   end

  initial begin
    apb_if.rst_n=0;
    repeat (1) @(posedge apb_if.pclk);
    apb_if.rst_n=1;
  end

  initial begin
    uvm_config_db#(virtual dut_if)::set(uvm_root::get(), "*", "vif", apb_if);
    run_test("apb_random_test");
    //run_test("apb_direct_test");
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars;
  end

endmodule
```

Figure 5.22. APB top module

Simulation Results

The simulation results are shown in figure 5.23. For each transaction there is a value for data and address. The convert2string is used to display the transactions received by the driver, monitor and subscriber.



```
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(34) @ 5430: uvm_test_top.env.scoreboard [APB_SCOREBOARD] ------ :: WRITE DATA       :: ------
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(35) @ 5430: uvm_test_top.env.scoreboard [] Addr: f7
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(36) @ 5430: uvm_test_top.env.scoreboard [] Data: e8
# KERNEL: UVM_INFO @ 5470: uvm_test_top.env.agent.driver [APB_DRIVER ] Got Transaction pwrite=     paddr=db data=77
# KERNEL: UVM_INFO @ 5510: uvm_test_top.env.agent.monitor [APB_MONITOR] Got Transaction pwrite=     paddr=db data=da
# KERNEL: UVM_INFO /home/runner/apb_coverage.sv(34) @ 5510: uvm_test_top.env.coverage [APB_SUBSCRIBER] Subscriber received tx pwrite=     paddr=db data=da
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(40) @ 5510: uvm_test_top.env.scoreboard [APB_SCOREBOARD] ------ :: READ DATA Match :: ------
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(41) @ 5510: uvm_test_top.env.scoreboard [] Addr: db
# KERNEL: UVM_INFO /home/runner/apb_scoreboard.sv(42) @ 5510: uvm_test_top.env.scoreboard [] Expected Data: da Actual Data: da
# KERNEL: Error: Assertion 'CHK_APB_PRDATA' FAILED at time: 5,520ns, design.sv(86), scope: test.apb_if, start-time: 5,510ns
```

Figure 5.23. APB simulation results

The assertion CHK_APB_PRDATA caught an error that the scoreboard was unable to catch. During a read transaction the value in the address is 'db' but the value read is 'da'. This is clearly a data read mismatch. This is a good example of how an assertion can be used in conjunction with scoreboard to catch errors missed by the latter.

CHAPTER SIX

Conclusion

In this thesis steps to writing a typical UVM testbench were outlined. The first UVM testbench environment was designed to test a trivial adder. Emphasis was placed on describing in detail the steps required for creating a fully functional UVM testbench. The second UVM testbench was designed to verify a APB FSM. Details on how to write a good verification plan, concurrent multi-clock assertions and basic functional coverage were outlined. This two labs can be used to form a foundation for teaching the basics of UVM. Moving forward the aspiring verification engineer must be introduced to other parts of the environment such as the register package. Most designs have a set of registers. UVM has introduced a register package that allows for control of registers in the DUT. Virtual sequence is another part of the of UVM environment that must be quickly adopted beyond the introduction of UVM. Most chip design systems are made up of several IP components that are connected together. Virtual sequences are used to start multiple sequences on different sequencers corresponding to each IP block.

REFERENCES

[1] J. Bergeron, *Writing Testbenches Using SystemVerilog.* Springer Science+Business Media, 2013.

[2] Duolos. Systemc versus sytemverilog. [Online]. Available: https://www.doulos.com/knowhow/systemc/

[3] K. A. M. Sharon Rosenberg, *A Practical Guide to Adopting Universal Verification Methodology.* Cadence Design Systems, 2010.

[4] G. Renuka, U. V, and C. P, "Advanced verification methodology for complex system on chip verification," vol. 6, pp. 11–21, 12 2015.

[5] A. B. Mehta, *ASIC/SoC Functional Design Verification.* Springer International Publishing, 2018.

[6] V. R. Cooper, *Getting Started with UVM.* Verilab Publishing, 2013.

[7] P. Wilcox, *Professional Verification.* Kluwer, 2004.

[8] M. Fujita, I. Ghosh, and M. Prasad, *Verification Techniques for System-Level Design 1st Edition.* Morgan Kaufmann, 2007.

[9] S. Ray, *Scalable Techniques for Formal Verification*, 01 2010.

[10] G. Bhatnagar and D. Brownell, "Portable stimulus vs formal vs uvm a comparative analysis of verification methodologies throughout the life of an ip block," 02 2018.

[11] G. T. Chris Spear, *SystemVerilog for Verification.* Springer Science+Business Media, 2012.

[12] S. Vasudevan, *Practical UVM.* Scotts Valley, CA: CreateSpace, 2016.

[13] R. Salemi, *UVM Primer: A step-by-Step Introduction to the Universal Verification Methodology.* RaySalemi, 2013.

[14] B. Hunter and J. Bergeron, *Advanced Uvm.* CreateSpace Independent Publishing Platform, 2015. [Online]. Available: https://books.google.com/books?id=YppVjwEACAAJ

[15] M. Ehmer and F. Khan, "A comparative study of white box, black box and grey box testing techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, 06 2012.

[16] R. G. Ramdas Mozhikunnath, *Cracking Digital VLSI Verification Interview.* Ramdas Mozhikunnath, Robin Garg, 2016.

[17] A. Fiergolski, "Simulation environment based on the universal verification methodology," *Journal of Instrumentation*, vol. 12, no. 01, pp. C01 001–C01 001, jan 2017. [Online]. Available: https://doi.org/10.1088%2F1748-0221%2F12%2F01%2Fc01001

[18] C. E. Cummings, Ed., *Proc. UVM Transaction-Definitions, Methods and Usage (SNUG 2014).* Synopsys User Group, 2014.

[19] J. B. Clifford E. Cummings, Ed., *Proc. Using UVM Virtual Sequencers and Virtual Sequences (DVcon 2016).* Design Verification Conference, 2016.

[20] M. Peryer, Ed., *Proc. There's something wrong between Sally Sequencer and Dirk Driver - why UVM sequencer and drivers need some relationship counselling (DVcon 2012).* Design Verification Conference, 2012.

[21] H. C. Clifford E. Cummings, Ed., *Proc. UVM Analysis Port Functionality and Using Transaction Copy Commands (SNUG 2018).* Synopsys User Group, 2018.

[22] M. Graphics, *UVM Cook Book.* Mentor Graphics, 2018.

[23] C. E. Cummings, Ed., *Proc. OVM/UVM Scoreboards - Fundamental Architectures(SNUG 2013).* Synopsys User Group, 2013.

[24] R. A. Rich Edelman, Ed., *Proc. Monitors, Monitors Everywhere - Who is Monitoring the Monitors(DVCon 2013).* Design Verification COnference, 2013.

[25] T. F. Clifford E. Cummings, Ed., *Proc. OVM and UVM Techniques for Terminating Tests(DVCon 2011).* Design Verification COnference, 2011.

[26] D. S. Dr David Long, John Aynsley, Ed., *Proc. A Beginner's Guide to Using SystemC TLM-2.0 IP with UVM(SNUG 2012).*

[27] Accellera, "Universal verification methodology (uvm) 1.2 user's guide," https://www.accellera.org/downloads/standards/uvm, 2020.

[28] M. Litterick, "Sva encapsulation in uvm - enabling phase and configuration aware assertions," 02 2013.

[29] R. L. Brian Hunter, Ben Chen, Ed., *Proc. Reset Testing Made Simple with UVM Phases(SNUG 2013).* Synopsys User Group, 2013.

[30] J. Aynsley. Uvm verification primer. [Online]. Available: https://www.doulos.com/knowhow/sysverilog/uvm/tutorial_0/

[31] S. Sutherland, Ed., *Proc. Getting Started with SystemVerilog Assertions(DesignCon 2006).* Design Conference, 206.

[32] S. Das, R. Mohanty, P. Dasgupta, and P. P. Chakrabarti, "Synthesis of system verilog assertions," in *Proceedings of the Design Automation Test in Europe Conference*, vol. 2, 2006, pp. 1–6.

[33] C. Elakkiya, N. S. Murty, C. Babu, and G. Jalan, "Functional coverage - driven uvm based jtag verification," in *2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, 2017, pp. 1–7.