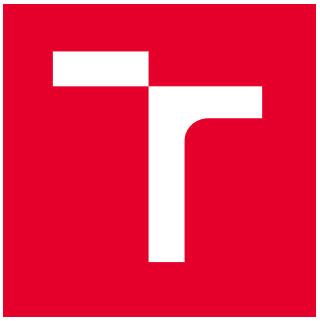

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF MICROELECTRONICS

ÚSTAV MIKROELEKTRONIKY

VERIFICATION ENVIRONMENT FOR BLDC MOTOR CONTROLLER

VERIFIKAČNÍ PROSTŘEDÍ PRO SYSTÉM ŘÍZENÍ BLDC MOTORŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Vít Kalocsányi**
AUTOR PRÁCE

SUPERVISOR — **Ing. Vojtěch Dvořák, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Microelectronics**

Department of Microelectronics

Student: Bc. Vít Kalocsányi ***ID:*** 220877

Year of study: 2 ***Academic year:*** 2023/24

TITLE OF THESIS:

Verification environment for BLDC motor controller

INSTRUCTION:

Familiarize yourself with the UVM verification methodology and the typical structure of a BLDC motor control system. Define the method of verification of the motor control system, design a suitable structure of the verification environment based on the UVM standard, implement all the necessary verification components and perform the verification of the example motor control system. In the Master's thesis conclusion, evaluate the benefit of using the UVM methodology in this application.

RECOMMENDED LITERATURE:

According to recommendations of supervisor

Date of project specification: 5.2.2024 ***Deadline for submission:*** 21.5.2024

Supervisor: Ing. Vojtěch Dvořák, Ph.D.

doc. Ing. Lukáš Fujcik, Ph.D.
Chair of study program board

ABSTRACT

This thesis addresses the need for thorough verification in the design of BLDC motor controllers. This paper explains functional verification of digital circuits and Universal Verification Methodology (UVM), and it focus on the design of verification environment using this methodology. In this work a typical structure of BLDC motor controller is explained and the verification method for this controller is suggested. Furthermore, implementation of the verification environment is described, and benefits of introducing the UVM into the verification workflow are discussed.

KEYWORDS

Functional verification, UVM, BLDC motor controller, FPGA, SystemVerilog

ABSTRAKT

Tato práce se věnuje požadavku na důkladnou verifikaci při návrhu systému řízení BLDC motorů. V práci je vysvětlena funkční verifikace číslicových obvodů a univerzální verifikační metodika (UVM) a práce je zaměřena na návrh verifikačního prostředí s využitím této metodologie. Dále je v této práci vysvětlena typická struktura systému řízení BLDC motoru a definován způsob verifikace takového systému řízení. Dále je popsána implementace verifikačního prostředí a diskutovány přínosy zavedení UVM do verifikačního procesu.

KLÍČOVÁ SLOVA

Funkční verifikace, UVM, Systém řízení BLDC motoru, FPGA, SystemVerilog

ROZŠÍŘENÝ ABSTRAKT

Tato práce se zabývá návrhem prostředí pro funkční verifikaci systému řízení BLDC motoru implementovatelného do obvodu FPGA. Vzhledem k rostoucím nárokům na efektivitu a spolehlivost takovýchto systémů řízení je ověření jednotlivých parametrů návrhů pomocí funkční verifikace velmi aktuálním tématem.

Pro verifikaci je v této práci využita univerzální verifikační metodika (UVM), což je standardizovaná metodika funkční verifikace, která se hojně využívá napříč odvětvím návrhu číslicových obvodů. Velkým přínosem využití této metodiky je její flexibilnost, standardizace a přenositelnost jednotlivých komponent mezi jednotlivými projekty, což z dlouhodobého hlediska šetří čas při tvorbě verifikačního prostředí, ovšem tvorba počátečního prostředí při přechodu na UVM může být časově velmi náročná.

V první části práce je obecně popsána verifikace číslicových obvodů a dále jsou popsány základní principy UVM. Je zde představena struktura verifikačního prostřední podle této metodiky a jednotlivé komponenty, které jsou v tomto prostředí využívány. V práci jsou popsány i další prvky UVM, jako je například rozhraní pro modelování na úrovni transakcí nebo mechanismus fázování simulace tak, aby bylo možné synchronizovat jednotlivé události v rámci testu. Dále jsou představeny možnosti konfigurace v rámci UVM a základní makra, které je možné v UVM použít pro specifické požadavky. V práci nejsou obsaženy všechny nástroje, které tato metodika nabízí, ale jsou představeny všechny základní nástroje k jejímu pochopení.

V druhé části práce jsou představeny komutační metody pro BLDC motory, mezi které patří lichoběžníková metoda, sinusová metoda a metoda vektorové regulace (FOC), a dále je představena struktura typického systému řízení BLDC motoru využívající metodu FOC. V rámci tohoto systému řízení je obsažen blok řízení motoru, který zpracovává požadavky řídicího systému a ovládá motor pomocí třífázového PWM signálu. Nastavení PWM je prováděno na základě informací o poloze a rychlosti z jednoduchého senzoru pozice a zpětné elektromotorické síly, která je vypočtena ze snímaných fázových proudů motoru pomocí externího ADC.

V další části této práce je navrhnuta metoda verifikace takovéhoto systému řízení pomocí UVM. Verifikace je zaměřena na blok řízení motoru a nejsou verifikovány ostatní periferie celého systému řízení. Nicméně pro možnost interakce mezi verifikačním prostředím a DUT je nutné vytvoření komponent pro interakci s jednotlivými periferiemi.

Vzhledem k tomu, že motor má definovanou zpětnou vazbu, je nutné vygenerovat data pomocí referenčního modelu. Model je vytvořen v prostředí Matlab a data jsou vygenerována pro každý typ regulační smyčky samostatně. Vzhledem k tomu, že z důvodu přiměřeného času simulace není možné přesně vygenerovat referenční data ze senzoru pozice pro blok odhadu pozice a rychlosti, je nutné v simulaci data o pozici a rychlosti přímo vnutit do bloku řízení motoru.

Dále je v práci popsáno navrhnuté verifikační prostředí. Základem verifikačního prostředí jsou jednotlivé komponenty, které přímo interagují s DUT. Pro primární

komunikaci je definována komponenta Modbus, která umožňuje příkazy pro zápis a čtení mezi prostředím a DUT. Dále je implementována komponenta NVM, která modeluje funkci externí paměti, ve které je uložena konfigurace DUT. V rámci komunikace je NVM pasivní, a tedy pouze reaguje na požadavky DUT. Další komponentou je ADC, které předává DUT referenční data o fázových proudech motoru.

Verifikační prostředí obsahuje také komponenty, které monitorují výstupy DUT, konkrétně třífázový PWM signál a telemetrické rozhraní. V případě PWM je kontrolováno výstupní napětí nastavené na každé lince prostřednictvím měření periody, střídy a ochranné doby (dead-time). Telemetrické rozhraní je konfigurováno pomocí sběrnice Modbus a verifikační prostředí přes něj čte přesné hodnoty napětí, které jsou nastavovány na výstup. Tyto hodnoty jsou dále porovnány s referenčními.

Jednotlivé komponenty předávají relevantní data do výsledkového modulu (scoreboard) pro ověření správnosti funkce DUT. Tento modul obsahuje metody pro čtení referenčních dat v závislosti na zvoleném pracovním módu DUT a jejich následné porovnání s daty simulace. V rámci celého verifikačního prostředí jsou implementovány automatické kontroly, které v případě chyby simulace vypíšou chybu a výsledkem bude selhání dané simulace.

V rámci tohoto prostředí jsou definovány základní testy pro ověření funkce bloku řízení motoru vzorového systému řízení. PWM test nastavuje fixní střídu na jednotlivých PWM signálech a kontroluje, že po zapnutí výstupu DUT je tato střída správně nastavena. Test otevřené smyčky spustí DUT v tomto módu a kontroluje výstupní data vůči referenčním. V rámci otevřené smyčky nejsou využívány žádné vstupy referenčních dat pomocí ADC. Dále je obsažen hlavní test FOC metody řízení, kdy v rychlostní smyčce jsou referenční fázové proudy jednotlivých vinutí motoru předány DUT pomocí komponenty ADC, sekvenčně jsou vnuceny referenční data o pozici a rychlosti motoru do vnitřní paměti DUT a je provedena kontrola nastavování výstupních napětí proti referenčnímu modelu.

Dále jsou obsaženy dva testy dalších funkcí systému řízení. První je test aktivního brždění, který opět porovnává referenční data pro daný mód. Druhý test ověřuje funkčnost systému detekce, izolace a obnovy z poruch. Tento test nastaví limit pro maximální teplotu, která nesmí být překročena po určitou dobu a následně mění měřenou teplotu pomocí komponenty ADC. Když je tato teplota nastavena dostatečně dlouhou dobu, test očekává, že DUT přejde do chybového režimu.

V rámci této práce byla provedena verifikace vzorového systému řízení na úrovni RTL a všechny testy prošly bezchybně. Navržené prostředí funguje správně a přináší některé výhody spojené s využitím UVM, jako je standardizace a sjednocení celého prostředí, možnost relativně jednoduchého rozšiřování a upravování prostředí pro budoucí projekty a některé benefity v rámci simulace, jako je záznam transakcí v simulačním prostředí a přehledná struktura výstupního logu.

Je však vhodné podotknout, že nebylo možné využít všech benefitů, které UVM nabízí. Vzhledem k tomu, že bylo potřeba generovat referenční data pomocí externího modelu, nebylo možné naplno využít randomizovanou generaci vstupních dat nebo testování pomocí omezujících podmínek.

Zavedení tohoto verifikačního prostředí přináší jednu nevýhodu, kterou je prodloužení běhu simulací v porovnání s přímým testováním, což je způsobeno zvýšenými nároky na zdroje díky režii, kterou UVM vyžaduje. Celkově však přínosy zavedení UVM tento nedostatek převyšují, zvláště pak v kontextu možného dalšího rozšiřování verifikačního prostředí.

BIBLIOGRAPHIC CITATION

KALOCSÁNYI, V. *Verification environment for BLDC motor controller*. Brno, 2024. Available also from: https://www.vut.cz/studenti/zav-prace/detail/159933. Master's thesis. Brno University of Technology, Faculty of electrical engineering and communications, Dept. of microelectronics, Advised by Vojtěch Dvořák.

AUTHOR'S DECLARATION

Author:	*Vít Kalocsányi*
Author's ID:	*220877*
Paper type:	*Master's Thesis*
Academic year:	*2023/24*
Topic:	*Verification environment for BLDC motor controller*

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the project and listed in the comprehensive bibliography at the end of the project.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation S 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

.

Brno, May 21, 2024 ----------

Author's signature

ACKNOWLEDGEMENT

I would like to thank my supervisor Vojtěch Dvořák for his effective methodical, pedagogical, and professional help, for excellent communication and invaluable advice and support during the elaboration of this thesis.

Brno, May 21, 2024 ----------

Author's signature

CONTENTS

FIGURES

INTRODUCTION

Modern electric propulsion systems are commonly powered by brushless DC (BLDC) motors. Such motor is driven by a motor controller that is responsible for the performance of the engine. With rising demand for efficient and reliable propulsion systems, ensuring the performance and reliability of these systems becomes essential. The complexity of these systems requires a development of challenging verification methods to verify functionality and compliance with the specified requirements.

This thesis focuses on a development of verification environment designed for BLDC motor controllers, using Universal Verification Methodology (UVM) to provide standardized and comprehensive approach to the functional verification. The designed verification environment is implemented in SystemVerilog language using the UVM. The simulation data are generated using a golden reference model of the propulsion system to ensure the accuracy down to the last bit.

The purpose of this thesis is to introduce the UVM which is the industry standard for functional verification, familiarize with the typical structure of a BLDC motor controller and propose a verification method for such controller. The following objective is to design the verification environment using the UVM and to discuss the advantages using this methodology.

The body of the thesis is divided into six chapters. Chapter 1 introduces basics of verification of digital circuits. The Universal Verification Methodology is described in chapter 2. The following chapter introduces BLDC motors, types of commutation methods and the typical structure of a motor controller. Chapter 4 suggests a verification method for the motor controller. Last two chapters describe the designed verification environment and discuss the advantages and disadvantages of introducing UVM into the verification flow for a sample project of the motor controller. At the end there is a conclusion to summarize the whole work done during processing of this thesis.

1 VERIFICATION OF DIGITAL CIRCUITS

During the development process of digital circuit, whether it is an implementation in FPGA or a design of ASIC, any mistakes in design can be both time consuming and costly. To mitigate as much risks as possible, verification plays a vital role. Verification is a process of verifying that the designed circuit meets the required specification. As the level of complexity of designs increased, the time required for verification increased as well, and it can consume approximately 70 % of the development time. This is the reason why design teams contain many dedicated verification engineers. [1]

The foundation of verification process is a functional verification which verifies that the circuit behaves as expected. Functional verification compares the outcome of the designed circuit with the specification of the function and checks that the results match expectations. This process can be very time-consuming due to the development of comprehensive tests, and runtime of simulations and analysis, but in the end, it cannot provide an absolute proof of correctness. The functional simulation can only provide a certain confidence in the correctness of the design. [1]

Formal verification is a process used to mitigate the risks that there are some bugs left after the functional verification. Formal verification performs a mathematical proof of the correctness of the design, and it can be divided into equivalence checking and model checking. Equivalence checking proves that the logic function is retained after a transformation process such as synthesis. Model checking mathematically proves whether conditions specified as assertions can occur. [1]

Generally, verification process starts with a planning phase. Outcome of this phase is a verification plan which defines all the features of the design to be tested, methodologies and strategies for verifying the design and expected outcomes of the verification process. The process continues with testbench development which is a test environment that applies input stimuli on the design under test (DUT) and checks its outputs. Testbench is a code usually written in a hardware verification language (HVL) like SystemVerilog or SystemC, but in can be also written in hardware description language (HDL) like VHDL or Verilog. Basic testbench architecture is visualized in Figure 1.1.

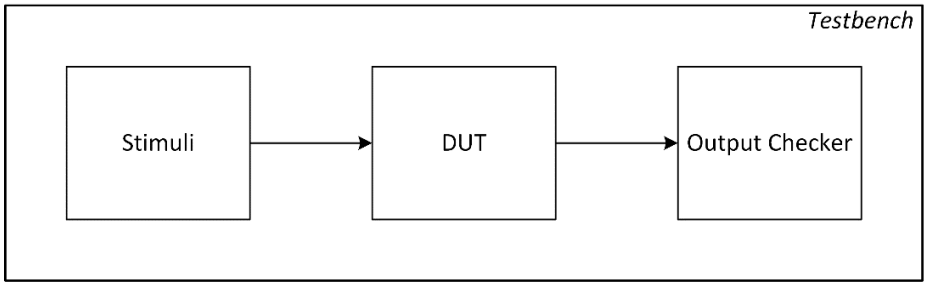

Fig. 1.1 Basic testbench architecture

Once the testbench is prepared, a simulator software is used to execute the testbench and the emulated DUT function. During simulation process, the outputs of the DUT are compared against expected results in the testbench. When an error is found during the simulation, debugging techniques are used to identify the source of the error to fix it. During the simulation a coverage metrics are usually collected as defined in the verification plan. There are two types of coverage. The first one is code coverage that is collected automatically by the simulator, and it measures the percentage of the code executed during the simulation. It can be measured as line, block, branch, or conditional coverage. Second type is functional coverage which is user defined and it measures which functions or specifications of the DUT were tested. [2]

2 UNIVERSAL VERIFICATION METHODOLOGY

Universal Verification Methodology (UVM) is a standardized verification methodology and a library written in SystemVerilog language developed by Accellera. The UVM inherits parts of several methodologies from different vendors, such as OVM, AVM, VMM and eRM to create a standardized, powerful, and flexible methodology for complex functional verification of digital circuits. It sets out guidelines on how to develop, integrate and expand verification environments and components. [3]

2.1 UVM Testbench

Testbench is a verification environment which is used to verify the function of a design under test (DUT). Typical architecture of UVM testbench is visualized in Figure 2.1. The DUT stands in the lowest level of the testbench, and its pins are connected via an interface directly to the transactors of the testbench, such as monitors and drivers. Transactors stands in the middle and make the conversion between a signal level (visualized with black arrows) and a transaction level (visualized with white arrows). All the components that are in the highest level above the transactors communicate only at transaction level using TLM connections. These components are sequencers, scoreboards, and others. [3]

UVM components (UVCs) integrate behavior of a particular object that needs to interact with DUT. UVCs are usually very well reusable, but they are also specific for each use case. Typically, there are multiple UVCs within the testbench, one for each communication interface of the DUT and one for each subsystem connected to the DUT. Besides UVCs, there are common components in the testbench environment for more or all the UVCs, like scoreboards or coverage collectors. These components are modified according to expected function of DUT and the test hence they are not easily reusable.

Usually, sequences of multiple UVCs need to run in a specific order and be synchronized between each other. This is achieved by using virtual sequencer and virtual sequences that are declared inside the test scope. Virtual sequencer launches sequences on the downstream sequencers according to the current virtual sequence.

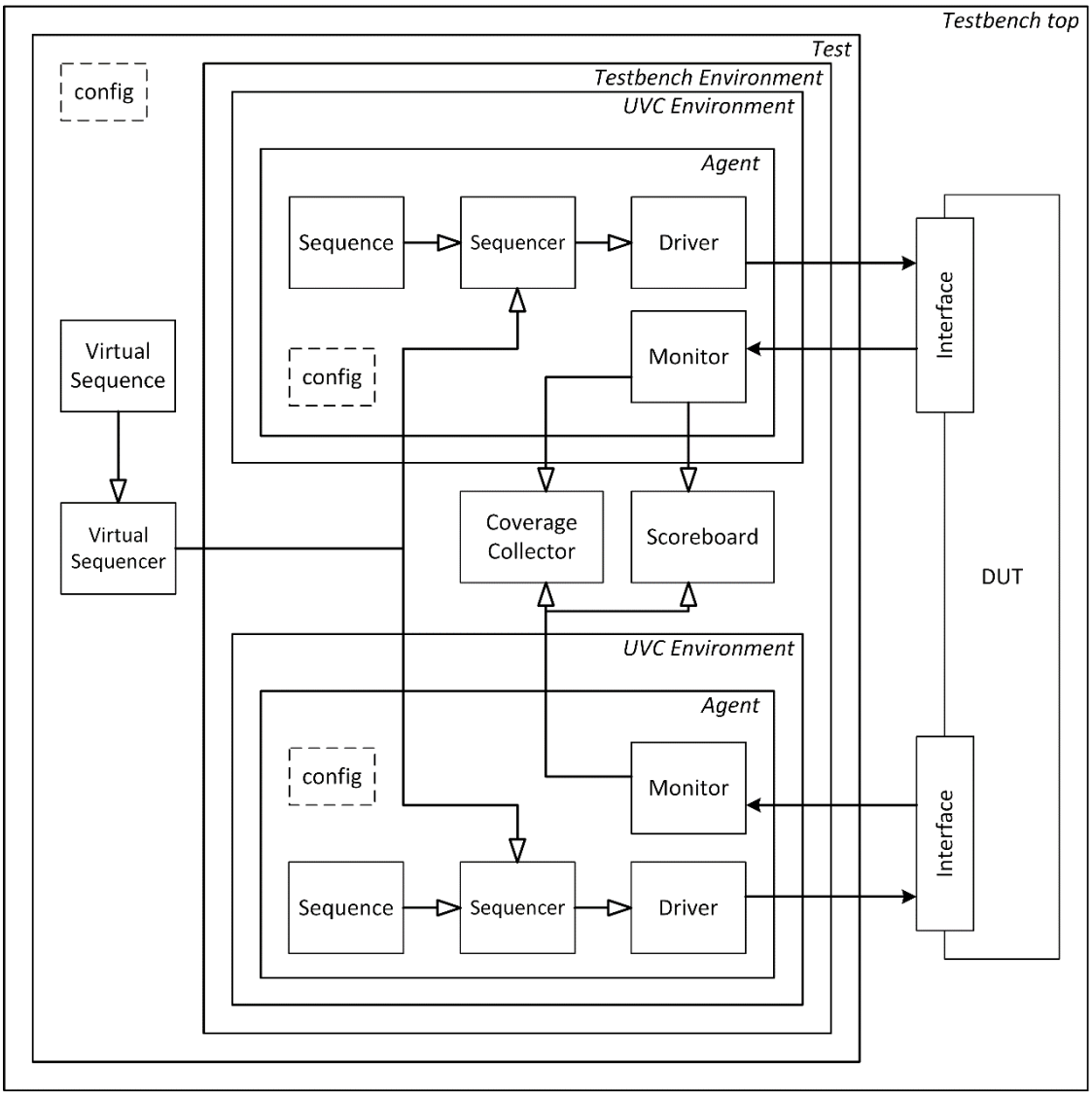

Fig. 2.1 Typical architecture of UVM Testbench

2.2 UVM Classes

Class Library is the cornerstone of UVM. The Class Library is written in SystemVerilog language, and it provides both basic and complex classes and utilities for developing of modular, scalable, reusable test environments and UVM verification components (UVCs). [4]

Basic building block of all UVM classes is *uvm_void* class which has no data members nor functions. Basic UVM functions are defined in *uvm_object* class which is derived directly from the *uvm_void* class and from which two important classes are derived – *uvm_component* and *uvm_transaction*.

The *uvm_component* class is the base for all UVM components which are objects that exist throughout the simulation. This class allows establishment of structural hierarchy and callbacks are defined for each test phase to execute the test in precise order.

The *uvm_transaction* class is the base for all UVM transactions which exist only for a limited time during the simulation. Most important part of transactions is timing and recording interface. [4]

2.2.1 UVM Test

This component enables execution of any user-defined test derived from this class. The name of the test is specified with the run simulation command and then executed by *run_test* task which shall be called at initial block inside the top module at time zero. UVM test is where the building of the whole testbench starts, and the test determines which components are created and how are they configurated. It also assigns any interface configuration to connect to DUT and invokes building of other lower-level components. UVM test also define the base sequence that is executed. [3] [4]

2.2.2 UVM Driver

Main purpose of the driver is to receive transactions from the sequence, convert them to a signal level and send them to the DUT via a virtual interface. For this purpose, the *uvm_driver* class includes *seq_item_port* which is used to request items from the sequencer to be sent to DUT. Also, the class includes *rsp_port* which can be used to send responses from the DUT back to the sequencer. [3] [4]

2.2.3 UVM Monitor

Monitor is another component with a direct access to an interface between UVM environment and DUT. Monitor is only passive component, and it observes activity on the interface. It needs to implement code to recognize patterns of the transactions and passes them to other components for further analysis via *uvm_analysis_port*. [3] [4]

2.2.4 UVM Sequencer

Sequencer is an object that manages the flow of transactions to the driver and the *uvm_sequencer* class includes methods that can manage the flow precisely. When an item is requested by the driver, the sequencer selects available sequence to generate next transaction item. When the item is ready, the sequencer uses the *seq_item_export* to pass the item to the driver. [4]

2.2.5 UVM Agent

Agent is a container component which encapsulates other verification objects that are interacting with a specific interface. An agent contains a configuration object that defines the specific interface and an active or passive agent configuration. Default configuration is active, which means it will drive data to the interface, hence it will typically contain

driver, sequencer, and monitor. If the agent is configurated as passive it will only monitor the transactions on the interface and omit the driver and sequencer instantiation. [3] [4]

2.2.6 UVM Environment

UVM environment is a container object that groups together related components, and it is usually used to enclose UVC. Base low-level environment can encapsulate a single verification component while other high-level environments are used to contain whole testbench and other environments. [4]

2.2.7 UVM Sequence

UVM Sequence defines the transaction flow during the test. The *uvm_sequence* class is derived from the *uvm_transaction* class. It is created only after it is used during the simulation and discarded after its body has finished execution. This approach enables flexible and constrained randomized generation of stimuli. The stimuli are generated in a form of sequence item that are passed via the sequencer to the driver. The sequence can be called from another sequence to form the more complex one. The *uvm_sequence* class declares variables for request and response which are of specified user-defined sequence item type, and it declares methods to operate with the variables. [3] [4]

2.2.8 UVM Sequence Item

Sequence items define structure of transaction data and different constraints that are associated with them. The *uvm_sequence_item* class provides fundamental functionality for sequence items and sequences to work in the sequence mechanism. The data of the sequence item contains payload to be driven to the DUT, but also any other information to control, configurate or analyze the flow. When the items are created by the sequence, they are randomized. That is why they usually contains constraints to ensure correctness of the data. [3] [4]

2.2.9 UVM Scoreboard

A scoreboard is a supervisory component that checks whether received transactions match expected values. Scoreboard is very complex component and is one of the hardest pieces of the code to write as it must predict the correct function of the DUT and observe and evaluate that the transactions collected by the monitors match the predictions. The transactions from monitors are collected via TLM analysis ports, which are described in chapter 2.3, and a compare method is usually used to evaluate the results. [3] [4]

2.3 TLM interfaces

In the UVM, transaction level modeling (TLM) library is implemented to define abstract interfaces to pass whole transaction objects. Using the TLM interfaces designed components are more reusable and modular.

TLM interfaces contain both unidirectional and bidirectional interfaces which allows passing transactions between components whose interfaces do not match exactly. These interfaces can use blocking or non-blocking methods or their combination. There are variations of TLM interfaces, such as unidirectional *put*, *get, peek* or *analysis* interfaces, or bidirectional *transport* or *master and slave* interfaces. These TLM interfaces are defined by a declaration of a corresponding port or export that are used to pass the transactions. [4]

Ports are instantiated by components that require the interface to transmit transactions, such as monitors. They use implementation of a corresponding interface method, such as a *put* method, and pass the transaction as an argument to the corresponding method. Exports are observers which are used to retrieve transactions that were passed into the ports. Each export needs to be connected to a corresponding port that it receives the transactions from. The connection is made when an export is passed as an argument to the *connect* method of the dedicated port. [4]

There are two different types of exports. A hierarchical export is used to propagate the TLM interface between parent component and its child component, meaning it is used only to forwards the transaction. An implementation exports ("imps") enables the component to access and change the implementation of the methods directly. Imps can be connected either to an export or directly to a port. Connections of ports, exports, and imps of TLM analysis interface are displayed in figure 2.2. [3]

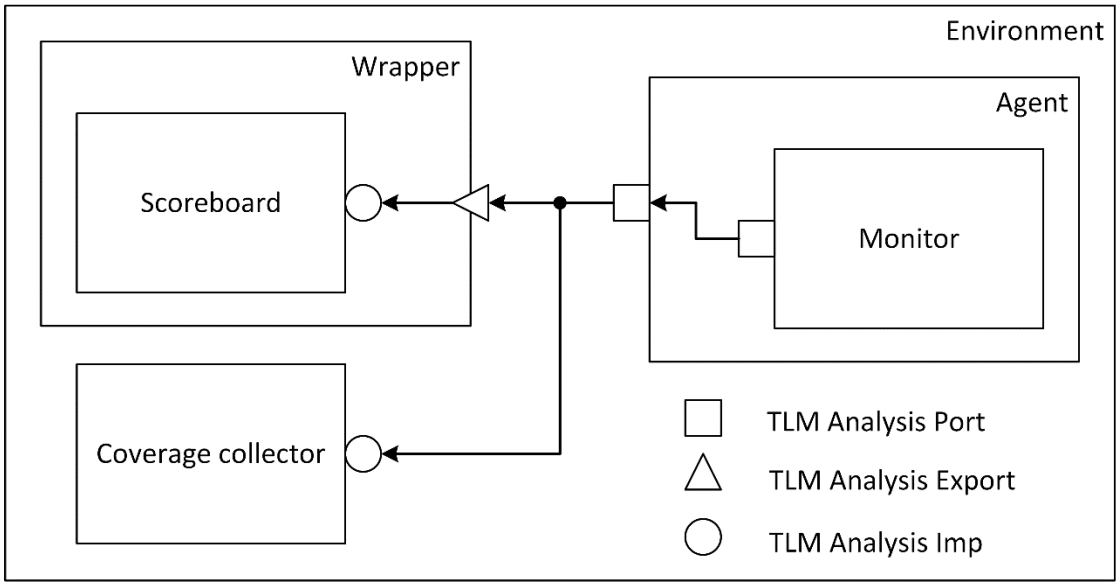

Fig. 2.2 TLM analysis interface connections

Unlike the others, TLM analysis interface allows propagation of transactions from one port to any number of exports, including zero. For this purpose, analysis interface provides only non-blocking *write* method. When this method is called by a port, the analysis port will go through a list of all connected exports and call the implementation of the *write* method associated with each export. This broadcasting is widely used by monitors to deliver all transactions it has collected to all its subscribers such as scoreboards and coverage collectors, but using TLM analysis interface also makes it possible to leave the monitor alone without any connection. [3]

2.4 Phasing

In UVM, an automated system is incorporated for the synchronization and coordination of all components inside the UVM testbench. All the components which are derived from the *uvm_component* class incorporates this phasing mechanism, and they are synchronized to the common phases. The flow of the phases is managed by implicit top-level *uvm_root* class that is automatically instantiated with the start of the simulation.

Common phases are classes derived from *uvm_phase* object and are executed in a predefined order as is visualized in Figure 2.3. Execution of each component's function for the current phase must be completed before the UVM testbench moves to the next phase.

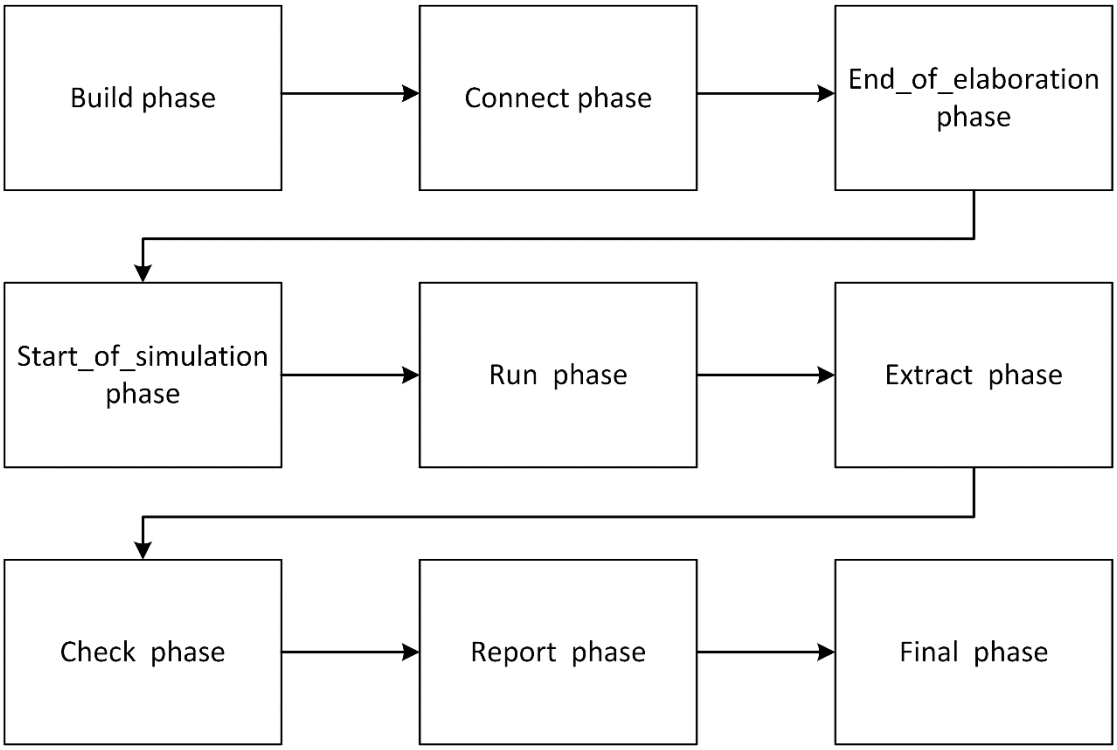

Fig. 2.3 UVM phases

The phase flow starts with *build* phase during which the testbench structure is created and all the components are instantiated and configurated. The testbench continues with *connect* phase followed by *end_of_elaboration* phase which functions are to set up intercomponent TLM connections and to fine-tune the testbench.

When the testbench is ready, *start_of_simulation* phase begins when last preparations are made, such as debugger or other run-time tools are started. Until this point, the simulation time is still zero and the *run* phase can start during which the simulation is performed. During this phase run-time phases are executed in parallel, and the *run* phase is completed when all the run-time phases are ready to end, and no more simulation needs to be performed. Each of the run-time phases has its corresponding *pre-* and *post-* phase to add more flexibility. The run-time phases are displayed in Figure 2.4.

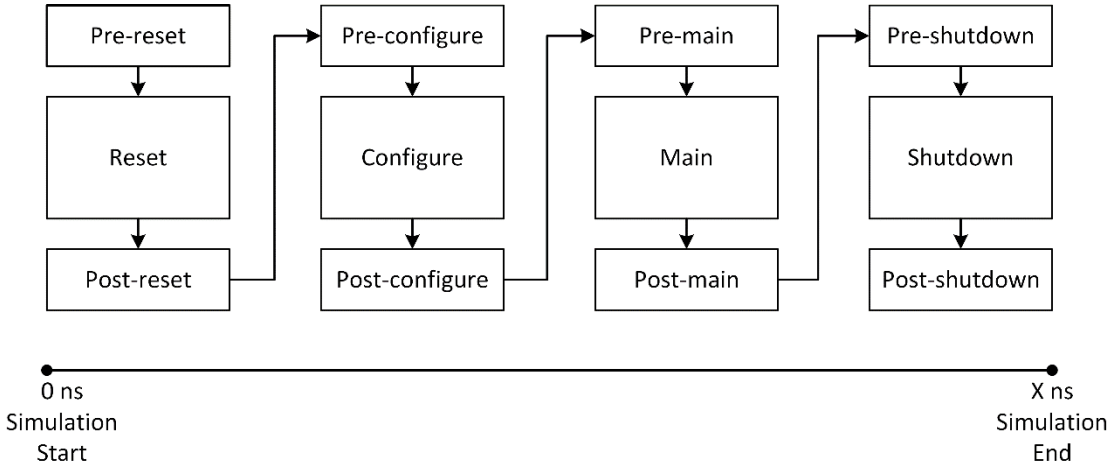

Fig. 2.4 Sequence of run-time phases

After the simulation is finished, the simulation time stops, and the *extract* phase starts to retrieve any remaining data from verification components to get the final state information. Usually, the statistics are calculated during this phase. Final inspection is performed during the *check* phase. It is checked if there is no data left to be considered, and it is known whether the test was successful or not. The testbench follows with the *report* phase. The test results are reported and written to file. At last, the testbench is ended with the *final* phase which closes all opened files and end any co-simulation engines. [4]

2.5 UVM Factory, configuration, synchronization

For improved flexibility, scalability, and memory efficiency the UVM factory is introduced. Purpose of the factory is to create components and other objects when requested.

A component or an object type needs to be registered with the factory for the possibility to be created. The registration process is typically performed

via a utility macro. When the components are properly designed to delegate the creation process to the factory, it is possible to use the factory overrides to change the type or the instance of the component when creating it. These overrides can be declared inside or outside components.

For component configuration a centralized database is provided which can be used throughout the simulation. In this *uvm_config_db* a type-specific data can be written and read at any time. This configuration can be stored with a specific hierarchical scope, or it may be stored as global to be visible to all components.

In UVM a run-time synchronization can be done by a mechanism of global events and barriers. The *uvm_event* class is provided for extending the SystemVerilog event datatype with the UVM functionality. The *uvm_barrier* class can be used to prevent processes from continuing until all reached specified simulation point. For instantiations of these classes to be accessible globally, each needs to be registered with specific *uvm_pool* class, which are used to store and pass the data by reference. [4]

2.6 UVM Macros

UVM includes a set of macros to simplify writing the code without specifying multiple SystemVerilog constructs. Macros are used to report messages and errors, specify object behavior, or specify sequence calling.

Usually, each component and object contain a utility macro which enables the correct factory operation. Field macros can be used together with a utility macro to enable some core data methods, such as copy, compare, clone and print.

For starting a sequence, *`uvm_do* macro is defined to start a sequence on the default sequencer. Modifications of this macro are also included to set constraints for the sequence called, specify a priority of the sequence, or perform the sequence on a different sequencer which is declared by *`uvm_declare_p_sequencer* macro. [4]

3 DRIVING BLDC MOTOR

BLDC stands for brushless DC motor which works on the same principle as a standard brushed DC motor, which converts electromagnetic force to a rotary movement. Compared to brushed DC motors, BLDC motors have rotor made of permanent magnets, which eliminates the need to drive the electrical current to the rotor via brushes. Stator contains windings to create magnetic field to move the rotor.

This construction of a motor brings many advantages like higher efficiency, longer lifetime than brushed DC motors due to less wearing parts and that BLDC motors can be precisely controlled. Working principle of BLDC motor is visualized in Figure 3.1. [5]

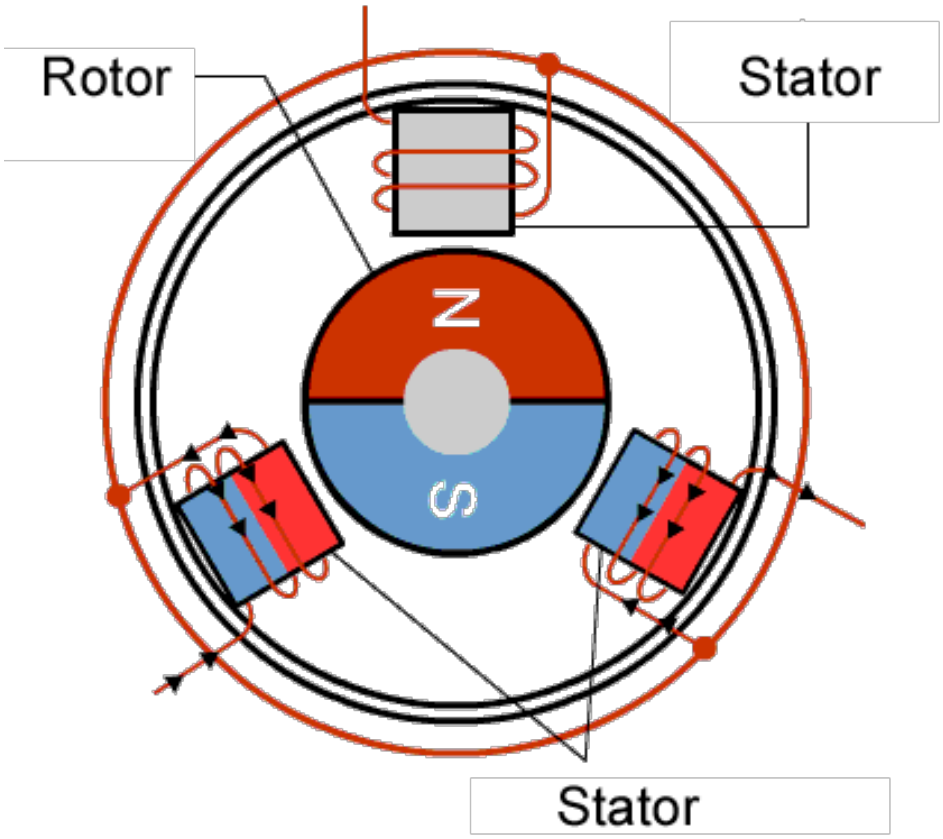

Fig. 3.1 Working principle of a BLDC motor [5]

3.1 Commutation methods

BLDC motor require a motor controller that electronically commutate the rotor and properly regulate speed and torque. The control is done by switching current to different windings in a proper sequence. This switching is commonly done with three pairs of transistors which are controlled with three-phase PWM signal from the motor controller.

Controller drives current to each winding in a proper sequence to run the motor. There are three main commutation methods: trapezoidal, sinusoidal, and field oriented control (FOC). Each of these methods differ in efficiency, smoothness, and simplicity of implementation. [6]

3.1.1 Trapezoidal method

Trapezoidal commutation is the most basic commutation method, and it is mainly used for its simplicity. It consists of six-step sequence. During each step, there is always one of the windings driven high, second is driven low and the third is left floating. While it is very simple to implement, it causes torque ripple, and it can cause motor vibrations at low speeds. [7][8]

3.1.2 Sinusoidal method

When implementing sinusoidal commutation, all three windings are always energized, and the controller adjust the current in each winding smoothly and sinusoidally always 120° apart from the others. Smooth current adjustment fixes the issue with torque ripple in trapezoidal commutation, but it leads to more comprehensive design. [8]

3.1.3 Field Oriented Control

FOC is by far the most complex commutation method, and it is used mostly for high-end application. During FOC motor phase currents feedback is observed to calculate voltage and current vectors. Based on calculation it is possible to drive the phase currents, so the motor torque stays always perpendicular to the rotor. This method allows for highest power output, smooth and precise operation in both low and high speeds and great dynamic load performance. [6][8]

3.2 Typical BLDC motor controller

While there are different approaches to design BLDC motor controller, such as having dedicated IC or using an MCU, this work deals with a FPGA core design of a BLDC motor controller which is ideal for precision and high-performance application. Simplified block structure of example FPGA core is shown in Figure 3.2.

For the correct operation it is necessary for the controller to know the position of the rotor, which can be done with or without usage of position sensor. Sensor-based motors typically use Hall-effect sensors, rotary encoders, or resolvers. Sensor-less BLDC motor control method estimates the rotor position by sensing back electromotive force (Back-EMF) on the undriven winding or by utilizing an observer in the control logic. [7]

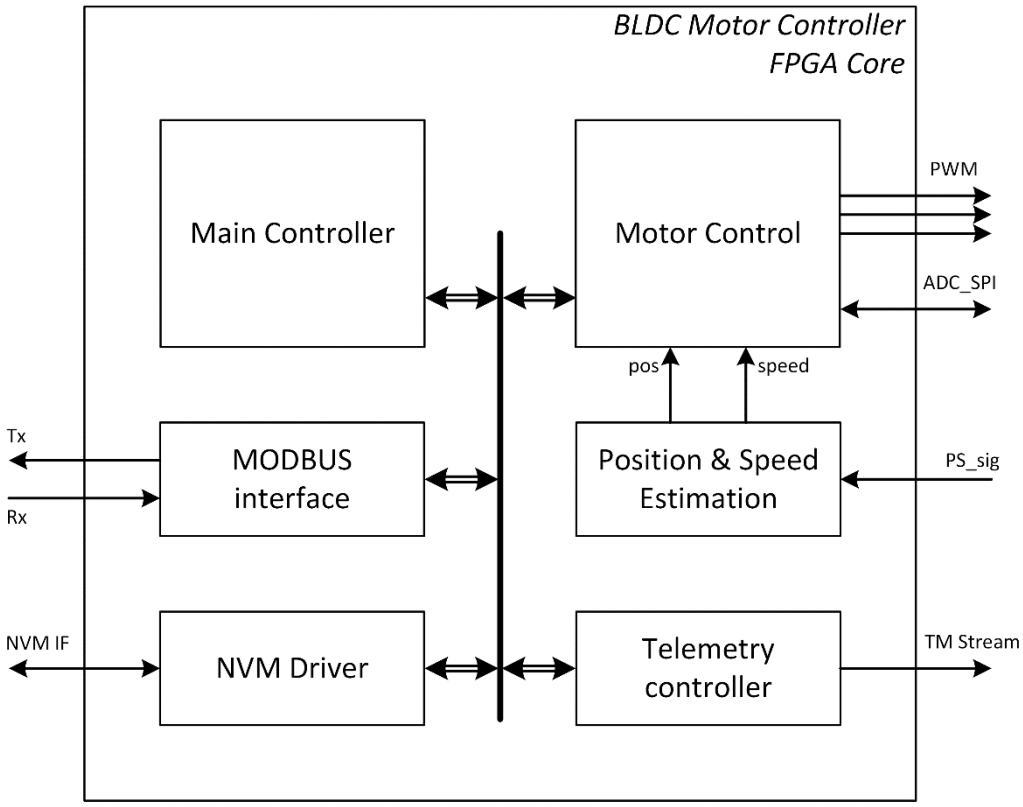

Fig. 3.2 Simplified structure of FPGA core of BLDC Controller

3.2.1 Motor Control block

Motor Control block is the most vital component of the design which implements the control method to drive the motor. Internal structure of the Motor Control block is displayed in Figure 3.3.

The control method is implemented in the Motor Control Sequencer which implements open loop method, FOC and PWM test mode. Current control schema is chosen by the control signals coming from the Main controller. Each of the methods periodically calculates voltages that should be applied on the motor.

Test mode serves only for testing PWM output with constant setting of the voltages. Open loop is used mainly for starting the motor before the position of the motor can be estimated. This method allows for setting acceleration or target frequency. FOC implementation is the most complex and it considers position and speed feedback from the motor. FOC calculate amplitude of voltages based on calculated error from required current, speed or position of the motor.

All the arithmetical operations to calculate required data are implemented in internal fixed-point arithmetic unit. The operation of the sequencer is controlled by the Data controller which is switching the control signals between Memory and Arithmetic unit based on the current method selected in Sequence controller.

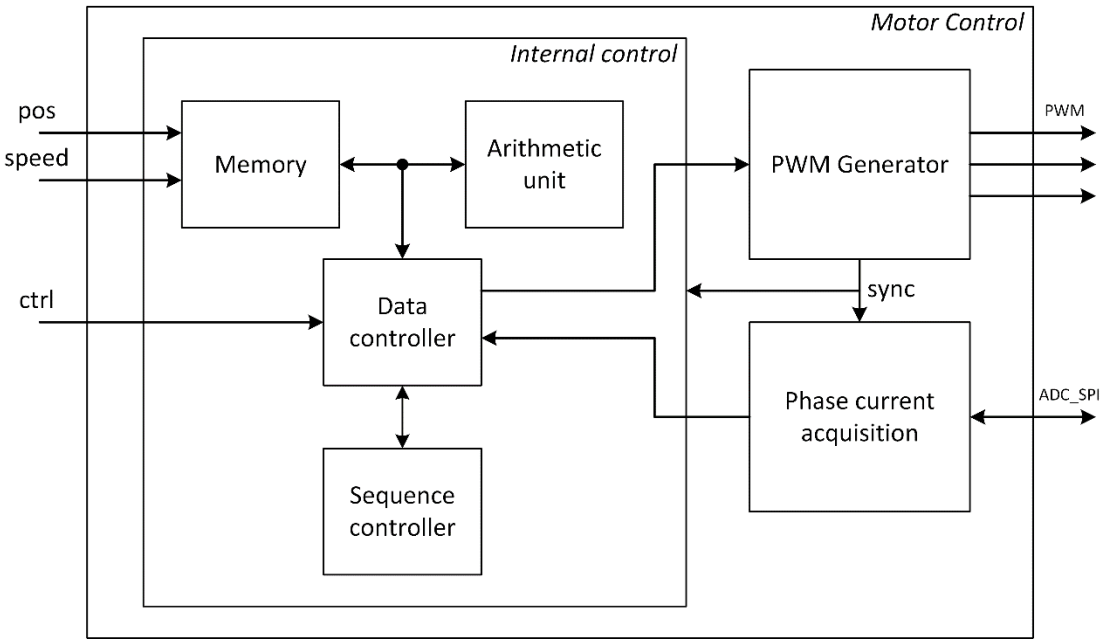

Fig. 3.3 Simplified structure of Motor Control block

PWM generator block converts calculated voltages from fixed-point internal representation to three-phase PWM voltages applied on the motor. Each phase of the PWM output is a pair of complementary signals and its duty cycle is calculated based on the calculated voltages. For preventing short-circuit in the motor, dead-time can be configurated. PWM generator also generates synchronization signal to ensure current acquisition is enabled when low-side transistors of the half-bridge are open.

Phase current acquisition block samples date from external ADC and the sampling sequence is initialized by the synchronization signal from the PWM generator. Acquired values of the currents are passed to the Motor Control Sequencer for the calculation of the voltages when FOC method is used.

3.2.2 Peripherals

All the peripherals as shown in Figure 3.2 communicate on internal bus. Main Controller is the top module that controls the function of the whole controller. It implements operational mode FSM, and it includes internal memory to store configuration of the system. The Main Controller also implements functions to decode commands received via Modbus interface and failure detection, isolation, and recovery (FDIR) function.

The Modbus interface communicates with outside world to receive commands and report status. It only supports simplified Modbus protocol which means that only read and write commands are supported and does not implement all the commands that Modbus protocol offers.

NVM Driver performs read and write operations between FPGA and external NVM to store the configuration of the FPGA. This configuration is always read after reset or power-up of the device.

Telemetry controller is used for testing of the system. It uses proprietary protocol to periodically send telemetry data packets. It is possible to configurate the Telemetry controller via Modbus interface or setting in NVM to change the packet content, the frequency of sending the packet in relation to PWM synchronization period, or to set a counter of how many packets should be sent.

Position and Speed estimation block uses signal from a simple position sensor to estimate position and speed of the motor. As the position generates one pulse per revolution of the motor, the speed is calculated based on time measurement between two pulses. The position is calculated by integrating current speed over one period.

4 VERIFICATION APPROACH

This chapter discuss an approach to verification of the BLDC motor controller described in chapter 3.2. The aim of this work is to verify the functionality of the Motor Control block within the motor controller with UVM. All other peripherals shall be considered fully functional and will not be verified even though it is necessary to use them to mediate communication between the DUT and the verification environment.

The most feasible option to verify the function of the Motor Control is to compare the results of the design simulation against a golden reference. As the golden reference a model of the control loop system shall be used. To ensure the most representative results it is necessary that the golden reference is a bit-accurate model to match exactly numbers from the RTL simulation.

The golden reference is designed in Matlab simulation environment that tends to use floating-point arithmetic. The RTL is utilizing a fixed-point arithmetic unit for a simpler implementation which is why a Matlab model of this arithmetic unit is needed to match the results to the last bit. This model has been designed in [9] to match the fixed-point arithmetic implemented in the FPGA exactly.

The motor controller can run different control methods and each control method of the system requires separate data to be extracted from the model. For this, separate models must be created to simulate different control methods as described in chapter 3.2.1. The test mode does not require reference data as the output of the DUT shall be constant according to the setting. Block structure of the golden reference model of FOC current control method is in Figure 4.1.

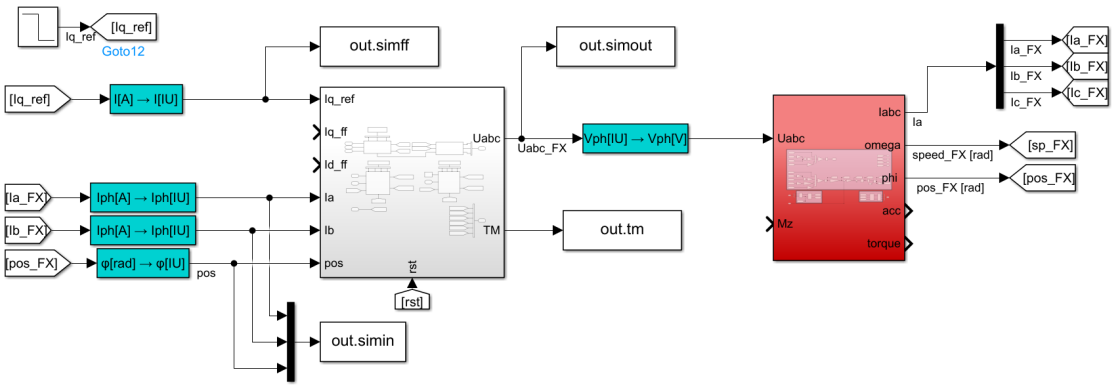

Fig. 4.1 Golden reference model of FOC current control method

The main part of the model is the Motor Control block. The blue blocks represent conversion functions between real values and the internal unit representation. The red block represents the BLDC motor. Reference value of the current and the feedback from the motor are applied on the Motor Control block which generates phase voltage that is directly applied on the motor without PWM.

The reference data extracted from the model include reference current value, motor currents and the position of the motor as the input data and the output voltages and telemetry data. The reference data are extracted in internal unit representation value at the beginning of each PWM period and are stored in a plain text format to be easily imported into the RTL simulation.

General flow of the verification of a single control method is visualized in Figure 4.2. After running the golden reference model simulation in Matlab to extract the reference data, the RTL simulation shall be executed.

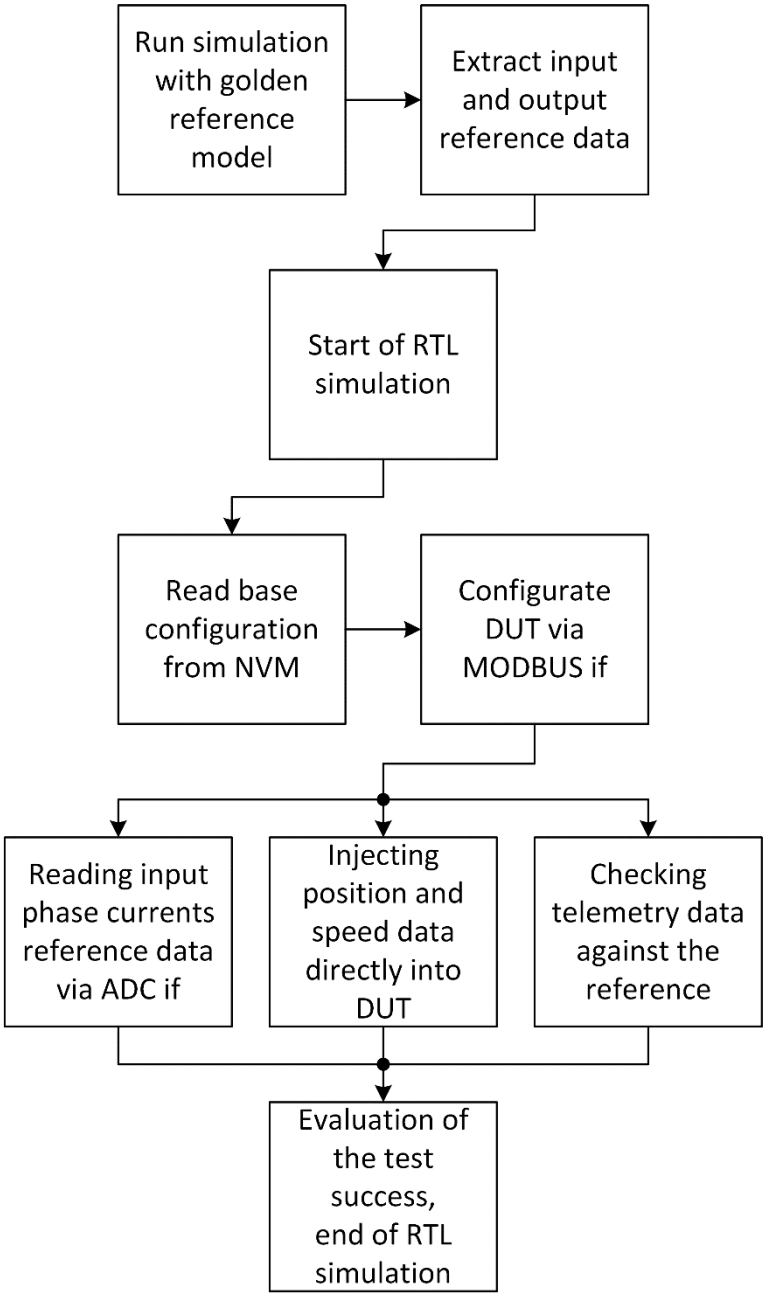

Fig. 4.2 Verification flowchart of a single control method

At the beginning of each simulation, a proper configuration of the system must be ensured. General configuration of the DUT shall be read from the NVM which will include configuration of the telemetry interface as it should be common for most of the tests. Also, configuration via NVM should be faster than using Modbus interface. Test specific configuration must be performed via Modbus to start the currently tested control method in the Motor Control block.

Once the control loop is started, reference input phase currents values must be available for the DUT to access via ADC SPI interface. The DUT shall autonomously access this data according to its configuration. At the same time reference position and speed values shall be forced directly into the Motor Control block. This is because the golden reference model does not implement bit-accurate position and speed estimation block. Reason why this block is omitted from the model is that it would prolong the simulation time significantly due to fine time resolution to match the DUT clock frequency.

In parallel with applying the input data on the DUT, the output voltages shall be compared in the scoreboard. Because the golden reference provides output voltages, it is not possible to ensure bit-level precision when comparing the data with PWM outputs. For this reason, the output voltages that are input of the PWM generator inside the Motor Control block will be read via Telemetry interface.

The Telemetry interface has a limited throughput, and it is not able to send all the required data each PWM period, hence it will not be possible to compare every reference value that will be available. To ensure the data are compared with correct value, it will be necessary to filter the reference data inside the scoreboard according to the configuration of the Telemetry interface. Once no more reference data are available, the simulation shall stop, and the result shall be evaluated.

As the goal is to verify the function of the whole Motor Control block, it is also necessary to verify the correctness of the PWM generator. The PWM lines shall be monitored all the time to measure the duty cycle. From this value the voltage can be calculated with a certain margin of error that is given by the value of LSB. It is then possible to check whether the calculated value does not differ from the reference by more than the expected maximum deviation.

The verification approach described above brings advantage the different scenarios are simulated in Matlab, where the simulation is significantly faster compared to RTL testing while the changes in verification tests are negligible, usually just different dataset to be used and different operational modes commands.

5 VERIFICATION ENVIRONMENT

In this chapter, designed verification environment based on UVM methodology is described. The environment is visualized in Figure 5.1. The first part of the chapter describes reusable UVCs that are used in the verification environment and the sequences that these UVCs use. The second part describes the motor controller specific components that are used, such as scoreboard and reference module. The last part describes implemented tests for the verification of the Motor Control block.

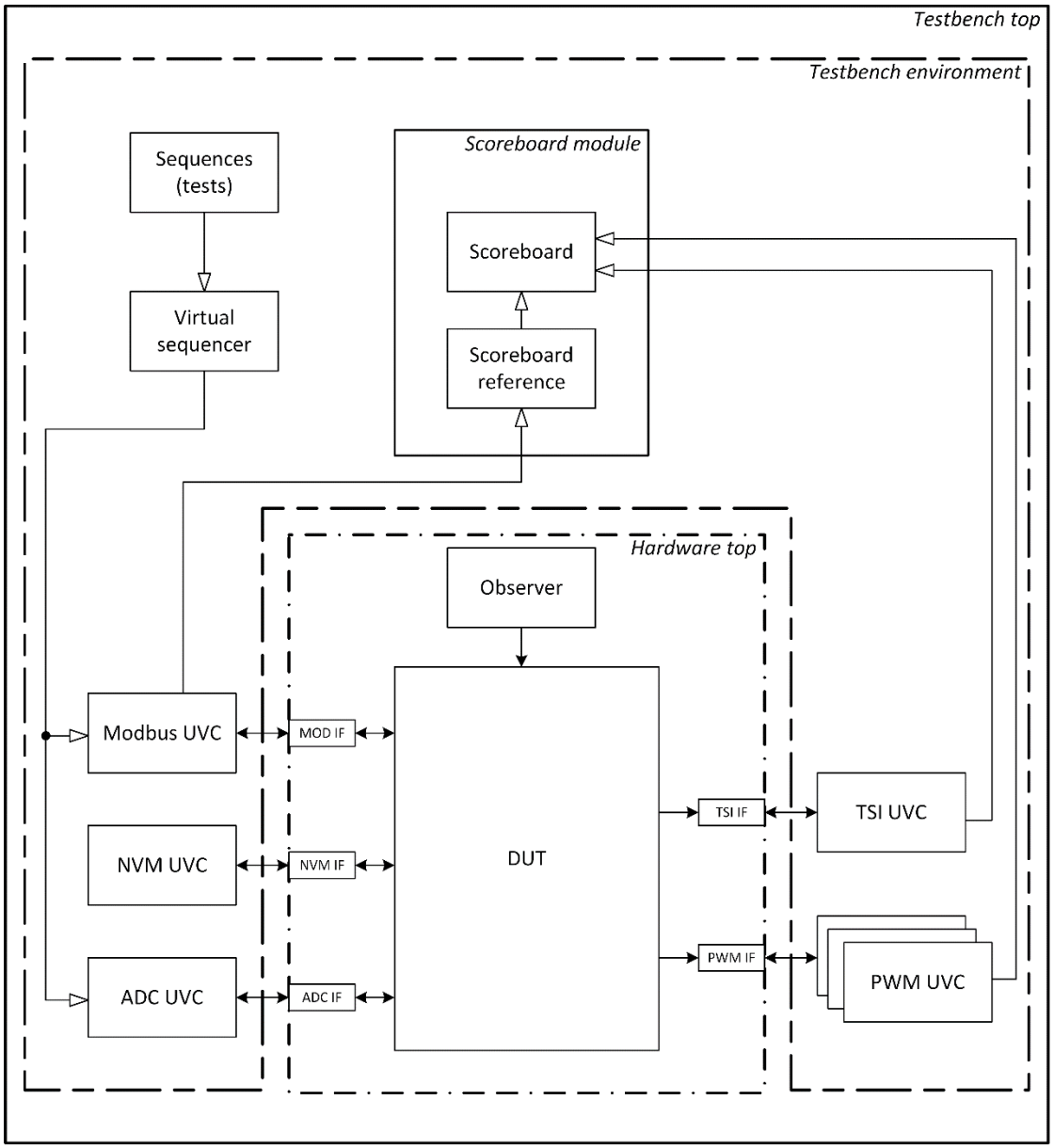

Fig. 5.1 Block structure of designed UVM verification environment

5.1 Reusable components

Each interface of the example BLDC motor controller uses a separate UVC. The reason for this is mainly that the UVCs can be reused for different controllers or in case of design change in the controller that affect only one interface, only single UVC needs to be replaced or modified.

The interfaces are instantiated inside hardware top module and they are not directly part of the UVC that they interact with. The design choice was made to implement some of the methods into the interfaces, such as sending data to DUT, monitoring the transaction or check timings on the interfaces. These methods are called by the monitors and drivers of each UVC which makes the interfaces dependent to a specific UVC. Description of each interface is included later in this chapter in the section describing the UVC that the interface interacts with.

Phasing during the simulation is common to all UVC and it will be shortly described here. First the initial block of the testbench top module sets the configuration of all the virtual interfaces to the configuration database. During Build phase all the components are created by the Factory which starts with creating environment of the UVC inside the testbench environment. The UVC's environment then creates an agent which creates a monitor and, in case the agent is active, it creates also a driver and a sequencer. Also, all the TLM ports, exports and imps are created here.

During Connect phase, all the TLM interfaces are connected using dedicated function of the ports. This connects all the sequence item ports and exports between each driver and corresponding sequencer which is performed at the agent's level. All the analysis ports, exports and imps between monitors and scoreboard module are also connected during this phase which happens inside the testbench environment. Here, also, handles are passed to the virtual sequencer to connect to each sequencer of the UVCs that it needs to interact with. During this phase the monitors and drivers of all the UVCs get the handle for the virtual interface from the configuration database which would put an error in case the handle would be unavailable.

The Run phase is executed only inside the monitors and drivers, and it describes the main function of these components. These also contain Report phase to report number of transactions driven or collected by them. The Check phase is only used by the scoreboard to check if there are no more reference data left unchecked and other phases are not used by the environment. These later phases are in detail described in the following chapters as they are specific to each of the UVCs.

5.1.1 Modbus UVC

Modbus UVC implement simplified function of Modbus data communication protocol which is specified in [10]. The architecture of this UVC is in Figure 5.2. The communication interface is based on a serial line containing only TX and RX wires. As the implemented example controller supports only read and write functions, this UVC implement only these two functions. The UVC encapsulates two different agents, of which one is receiver and the second is transmitter. Both agents use a separate line of the same interface that is connected to the physical wires during the simulations.

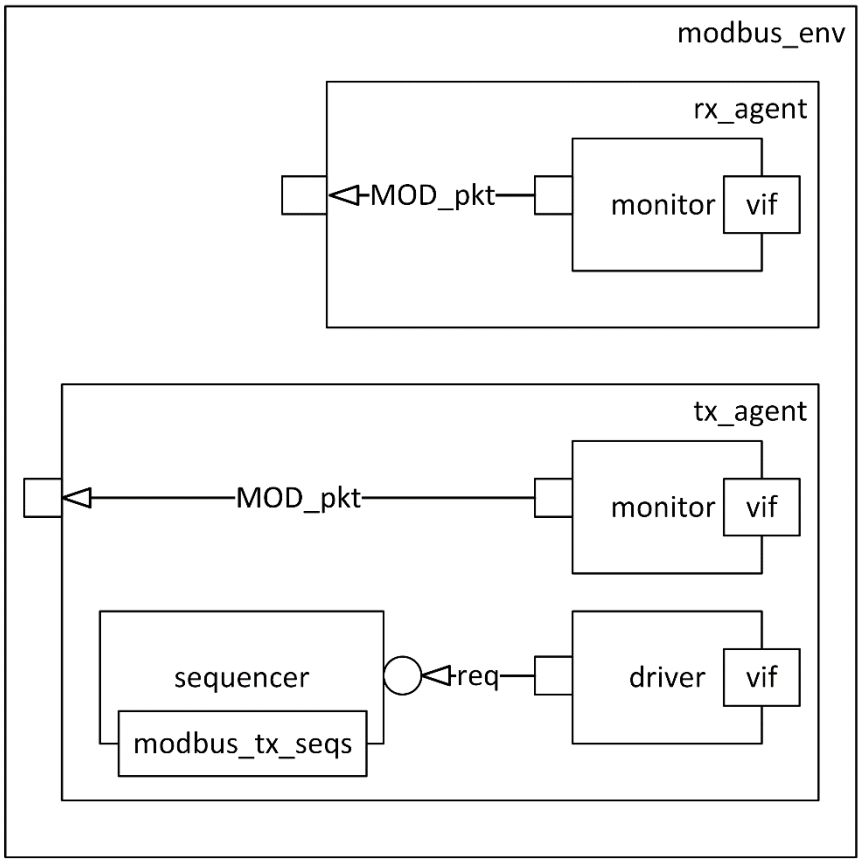

Fig. 5.2 Block diagram of Modbus UVC architecture

The MODBUS interface is parametrized with baud rate which is by default set to *19200 Baud/s* that is commonly used but it is not specified by the Modbus protocol. This configuration needs to match configuration of the DUT and the expected application.

Besides the physical RX and TX signals, the MODBUS interface has a container *modbus_pkt_mem* for data to be transmitted and some other internal variables and events to help with management of the transactions. There are two main tasks implemented in the interface. Task *send_to_dut* is called by the transmitter's driver. This task sends the data passed from the driver to the *modbus_pkt_mem* over the TX line, which is cross-wired to DUT's RX line at the hardware top level. Transaction on the line based on UART with one start bit, 8 bits of data and two stop bits.

The second main task is *collect_received_packet* ant it is called by the receiver's monitor. The receiver on the interface is built as an always block that monitors the RX line and triggers events to manage the flow of the whole transaction. As the Modbus protocol is composed of packets which content vary by the type of function specified by the function code at the beginning of the packet, this function code is recognized by the task, and it builds the content of the packet accordingly.

In the UVC, the **MODBUS packets** are built as UVM sequence item called as *modbus_packet* (visualized as MOD_pkt in Figure 5.2) according to the current transaction required. The packet has a container for each part of the packet, which is slave address, function code, start address, number of registers, number of bytes, payload, error code and CRC. Not all of these field are used for each transaction. The packet also contains a queue of bytes of data and a number of bytes to easily split the data into the transactions on the interface. Because not all of the fields can be randomized, the sequence item contains *post_randomize* function which sets the fixed fields, such as slave address, calculate the CRC and builds the queue of data based on the function code.

Receiver agent is passive and thus contains only monitor to record transactions on the RX line which means this agent reads transactions sent by the DUT. The monitor waits for the reset to be dropped and starts to monitor the transactions in a *forever* loop at the beginning of the Run phase. It creates a new *modbus_packet* for each transaction and in parallel it calls the *collect_received_packet* task and task to record the transaction start. After a packet is collected, is it stored in the *modbus_packet*, end of transaction is recorded, and the received packet is always written to the UVM analysis port of the monitor. Also, counter is incremented for the number of collected packets.

Transmitter agent is active. The driver sends data on the TX line that is connected to the RX of the DUT where it listens. The transmitter implements only write and read sequences. The driver uses two tasks in parallel inside the Run phase, one of which is *get_and_drive* task that always waits to receive the *modbus_packet* when it is requested by the running sequence. This task then passes the content of packet to the interface and calls the *send_to_dut* task in MODBUS Interface. It also uses recording of the start and end of the transaction and a counter of sent packets. The second task that is running in parallel is *reset_signals*. This task disables currently running transactions when a reset of the DUT occurs.

Monitor of the transmitter agent records the sent data and writes each packet to the analysis port for the whole environment to know the configuration of the DUT that can be set via the Modbus interface. This monitor is very similar to the receiver monitor but it uses dedicated task *collect_sent_packet* which monitors the data that are transmitted to the DUT.

5.1.2 NVM UVC

UVC of a non-volatile memory needs to be based on a specification of an NVM used in the example application, as the communication interface and timing parameters of the interface can be different for each NVM. The example controller is designed to work with MRAM MR0A08B from EVERSPIN Technologies. [11]

The main function of the NVM is modeled inside the **NVM interface** where the memory space is included. This choice was made due to the fact that the NVM needs to be passively accessed by the DUT at any time. The DUT as a controller on the line initializes transmissions and can perform read or write operation. The interface contains three control signals, CE, OE, and WE, and parallel bus for address and data content which are all connected to the DUT.

The NVM interface contains reset block that calls *init* function after the reset is dropped. This function reads content of the NVM stored in an external file. The interface is mainly built on *always* blocks that model the behavior of the interface based on the control signals as specified in [11]. This interface is designed to response with the worst-case timing parameters according to the specification and it implements checkers for the minimum timing requirements of the NVM.

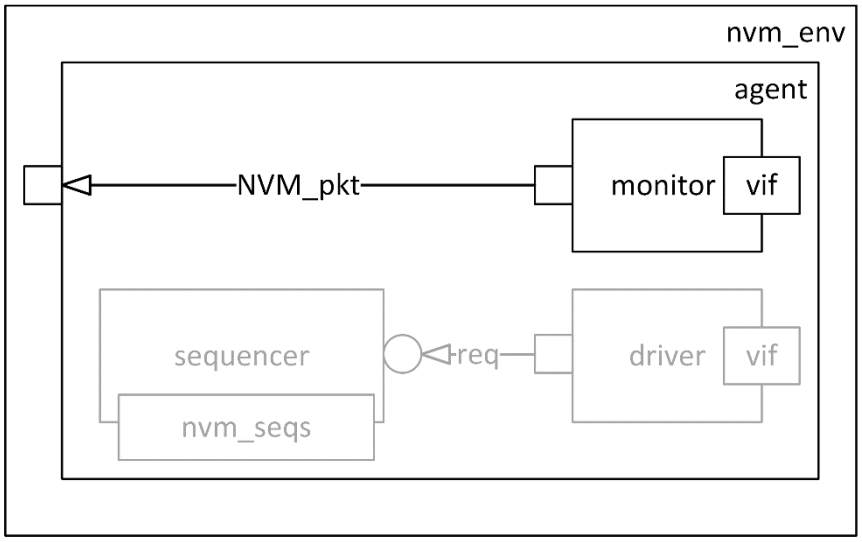

Fig. 5.3 Block diagram of NVM UVC architecture

The NVM UVC's architecture is in Figure 5.3. The **NVM packet** used for TLM interface are built on a sequence item called *nvm_packet*. This packet only contains an address and a data of the transaction. The **monitor** collects the transaction by calling a *collect_packet* task of the interface. When the DUT initializes a transaction, this task passes the corresponding address and data to the monitor. The monitor also records start and end of the transaction and the number of packets collected. The transaction is written to the analysis port, but it is not used by the verification environment.

The UVC also contains instantiations of active components (**driver**) and the necessary functions to write data to the NVM model from the TB for testing the NVM controller in DUT. There are currently no sequences created for the performed tests and driver is not used during the simulations.

5.1.3 PWM UVC

As the PWM signals is direct output of the motor controller, PWM UVC is passive and it acts only as a monitor of the PWM line. The architecture of the UVC is visualized in Figure 5.4. PWM UVC will check one phase of the PWM line that consist of high-side and low-side of the PWM signal, meaning total of three instantiations of this UVC will be implemented to check the whole interface. The main purpose of this components is to measure parameters of the PWM signal which are period, duty cycle and dead time. These measured parameters are essential for the scoreboard to check that the output voltages measured from the PWM correspond to the reference and telemetry data.

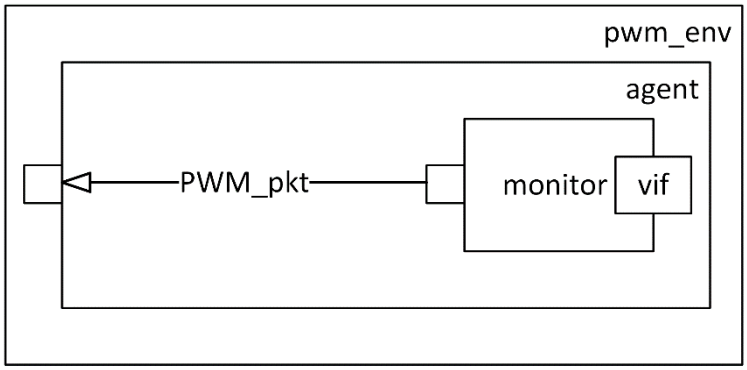

Fig. 5.4 Block diagram of PWM UVC architecture

The **PWM interface** is connected to the PWM signals and a synchronization signal of the PWM line for the precision of the measurement. Inside the interface times of rising and falling edge of each of the PWM signals are sampled. Inside *collect_packet* function there is a calculation performed to determine all the parameters of the PWM signal based on the sampled timing of edges. The calculation is done with the start of each PWM period signalized by the synchronization signal.

The period of the PWM signal is directly measured from the distance between the synchronization pulses. The duty cycle is defined by the relation

$$DC = (T_{H1} + T_{H2})/(T_{H1} + T_{H2} + T_L), \tag{5.1}$$

where DC is the duty cycle of the PWM signal, T_{H1} and T_{H2} represent the time of the high-side of the PWM line at logic one during one PWM period and T_L is the time of the low-side at logic one. All of these times are measured from the sampled times of the rising and falling edges of each the signal as visualized in Figure 5.5. The dead

time means the time when both signals are at logic zero. This is calculated as a sum of measured times DT_1 and DT_2 as visualized in the figure below.

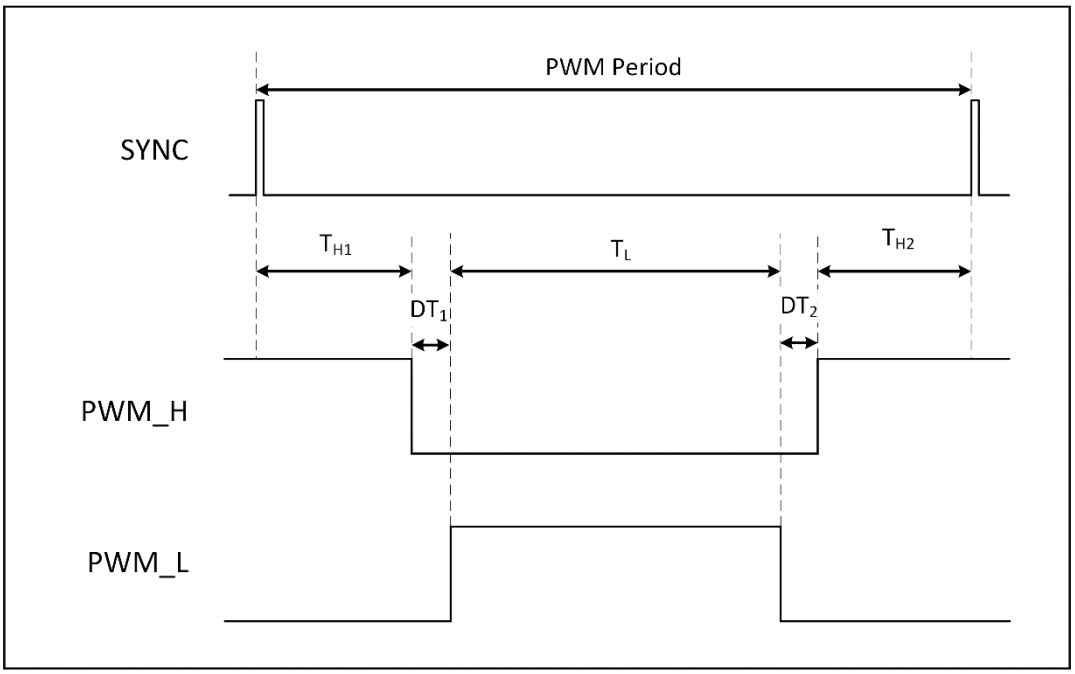

Fig. 5.5 Measurement of the PWM parameters

The **monitor** of the PWM UVC creates sequence item called *pwm_packet* which has containers for the parameters measured by interface – period of PWM, duty cycle and dead-time. After the reset is dropped, the Run phase of the monitor waits for the first edge on any of the PWM signals to trigger a global event. This event is used as a synchronization with the tests. After this, the monitor uses *forever* loop to call *collect_packet* task from the interface, record the start and end of the transaction, count the number of the transactions, and pass each transaction to the analysis port.

Further analysis of the PWM signals is not performed within this UVC because it would reduce the reusability of the UVC. The only checker implemented in the UVC is a checker of a failure of the PWM line in case when both high-side and low-side signals are in logical one at a same time which could shortcut the system.

5.1.4 ADC UVC

The block diagram of the ADC UVC is in Figure 5.6. This UVC implements a model of a specific ADC used in the system which is adc128s102qml-sp [12]. This ADC contains eight 12-bits channels and communicates with the controller on SPI interface as the responder. Even though the ADC cannot initiate communication with the DUT, the UVC is an active component because it needs to read the reference data from the golden reference and pass them sequentially via correctly set channel.

The basic behavior of the SPI is implemented in the **ADC interface**. It contains four wires connected to the DUT, which are CS, SCLK, DOUT and DIN. The output data is stored inside a variable and are always transmitted to the DUT when the transaction is initialized. The interface contains checks for all the timing parameters of the SPI line and responses with the worst-case timing as specified in the datasheet for the used ADC. [12]

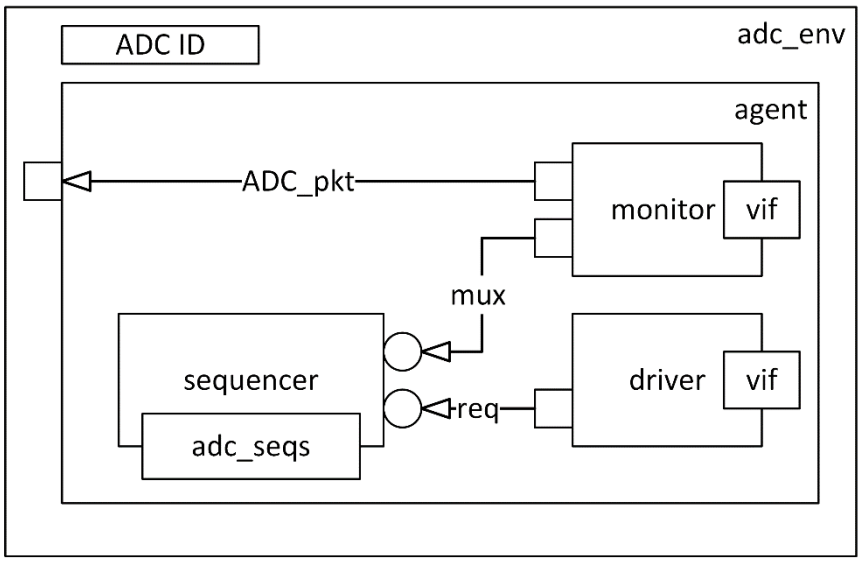

Fig. 5.6 Block diagram of ADC UVC architecture

During each transaction between the controller and ADC, the controller must always send the data indicating the selection of input channel for the subsequent transaction. The ADC responds with the measured value of previously requested channel. Inside the UVC the input data are collected with the monitor, and they are passed to the sequencer via a TLM port.

The ADC UVC's sequence item ***adc_packet*** contains 12-bit value of representing the measured value of a corresponding channel, an index of the channel and an index of a channel selected by the DUT for the subsequent transaction which is only used by the monitor. The **monitor** of this UVC creates a packet and uses task *collect_packet* of the ADC interface which samples both the output and the input data. The output data are stored together with the index of the current transaction. The input data corresponding with the index of subsequent transaction are stored and, also, passed to the sequencer to communicate the following transaction.

The **driver** calls the *send_to_dut* task declared inside the interface which only updates the value of data to be sent stored there and it waits until the transaction is initialized by the DUT and finishes. After the transaction is done it also resets the output data to zero. The driver also calls the *adc_reset* task in parallel to disable the transaction in case a reset occurs.

The **sequencer** has a container for the represented measured values of all the eight channels and the index of selected channel for the next transaction. The sequence called

on the sequencer always select the corresponding container and if the container is supposed to contain the reference data, the value in the container is updated with the new data from a specified external file. The path to this file needs to be configurated by the test sequence. If there are not any reference data on the selected channel, the UVC will fill the data with zero. The sequence finishes when all the reference data from the external file has been read and transmitted to the DUT.

The ADC UVC also includes an ID field at the environment level which is passed to all the other components. This is included in case the DUT interacts with multiple ADCs at the same time. The ID number must be passed to the specific instance at the Build phase from the testbench environment.

5.1.5 TSI UVC

Telemetry Stream Interface (TSI) is an interface through which the DUT sends telemetry data. TSI is based on UART interface with only one line utilized, which an output from the DUT. The TSI interface is configurable with baud rate with the default configuration of *3 MBaud/s* and it always listens as a receiver. As the packet content does not end with specific stop word or character, the configuration also includes number of words inside the packet. The UART part of the interface is used with fixed configuration of one start bit, 8 bits of data and one stop bit.

The packet content varies depending on the configuration. The TSI inside the DUT can be configurated through Modbus interface. Because the Modbus is slow for this configuration to be performed at the beginning of each test, the standard configuration is stored in the NVM. The UVC is by default counting with the standard configuration of the TSI packet consisting of three words, but this configuration can be modified when a new configuration of the TSI is written to the DUT via Modbus interface.

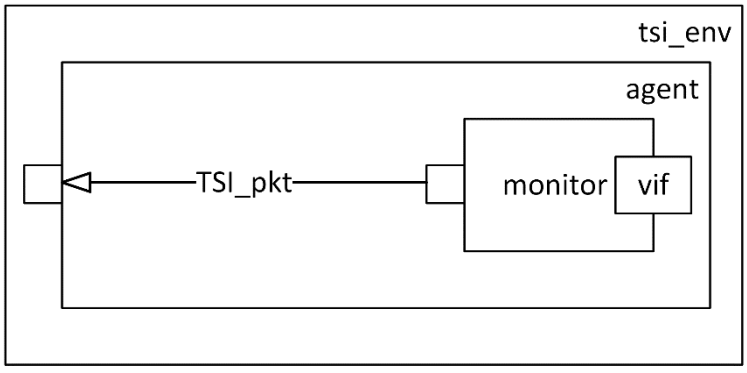

Fig. 5.7 Block diagram of TSI UVC architecture

The **TSI interface** contains local storage for packet content as a queue of 32-bits words. The *collect_packet* task inside the interface collects first the start identifier of the packet which is passed separately as an output value of the task. After that it

collects the configurated number of words and the index of the packet which are stored inside the local data storage as it is not possible to pass a queue directly.

The TSI UVC's architecture is visualized on block diagram in Figure 5.7. The **monitor** creates a sequence item *tsi_packet* which contains the containers for the start identifier and the queue of data. Then it calls the *collect_packet* task and with the output of the task it fills the start identifier container. When the task is finished, it transfers the content of the packet from the local storage inside the interface to the *tsi_packet* to pass it to the analysis port. Also, it contains recording of the transaction and a counter of them as all the other monitors used in the environment. The UVC sends the collected TSI packet to the scoreboard through the analysis port to be checked against the reference data.

5.1.6 Observer

Observer is an interface that inject reference values of position and speed directly into the memory of the motor controller. It is not implemented as the whole UVC because there is no reason for monitoring the injected data or to utilize any other benefit from using all the components of UVC.

The Observer needs a synchronization signal from the DUT to inject the data periodically in a precise time. For this reason, the same signal is used as in the case of PWM UVC. The Observer needs defined memory hierarchical path for the specific DUT, and it requires impulse from PWM UVC to start which makes the reusability a bit worse than the other components. Also, this interface is designed for a specific format of the file with reference data.

The format of the reference data file and the Observer allows to bypass the function of ADC with direct injecting of phase currents into the memory which is automatically enabled when no reference data for the ADC has been identified. The function of the Observer must be enabled directly from the test as it requires test-specific data, and it would jeopardize the test if wrong data has been injected inside the DUT's memory.

5.2 Top-level components

On the top level, all the used components and the DUT need to be instantiated and connected. Data acquired from the monitors have to be evaluated to be able to get a result of each test. This is performed by the following components which are specific to the designed environment, and they are reusable only with a large intervention in the code.

5.2.1 Scoreboard module

The scoreboard functionality is implemented inside a module that contains two parts. Architecture of this module is in Figure 5.8. The first part of the module is a scoreboard reference module, and the second part is the main scoreboard.

The main purpose of **scoreboard reference module** is to simplify the implementation of the scoreboard module by delegating some of the functions here. This reference module is connected to the analysis port of Modbus TX monitor, and it is responsible for decoding communication between the TB and the DUT on the Modbus interface. It reads the transaction and pass the operating mode of the DUT to the scoreboard or modify the configuration of the TSI UVC if the configuration changed via Modbus.

When a command changes the operating mode, the reference module is responsible for reading the reference data from the external file and passing them to the scoreboard. The path to the correct file for each test is configurated at the beginning of the test and is passed to the reference module via UVM configuration database. The reference data are here filtered according to the setting of the TSI to ensure the coherence between the reference data and the output from the DUT.

The **main part of the scoreboard** implements functions to compare reference data with received data from TSI and with real value of voltages set on each PWM line. The reference data passed by scoreboard reference module are stored in a queue of TSI packets. When TSI UVC collets a packet, it is processed via implementation function of the analysis port in scoreboard.

The compare functions put out warning messages in case receiving unexpected TSI packet and the scoreboard does not have any reference data to compare. The packets received from the DUT contains timestamp, which is not part of the reference data. When the first packet arrives, the timestamp is stored, and it is checked that the following packets contain incremental value from the first generated timestamp.

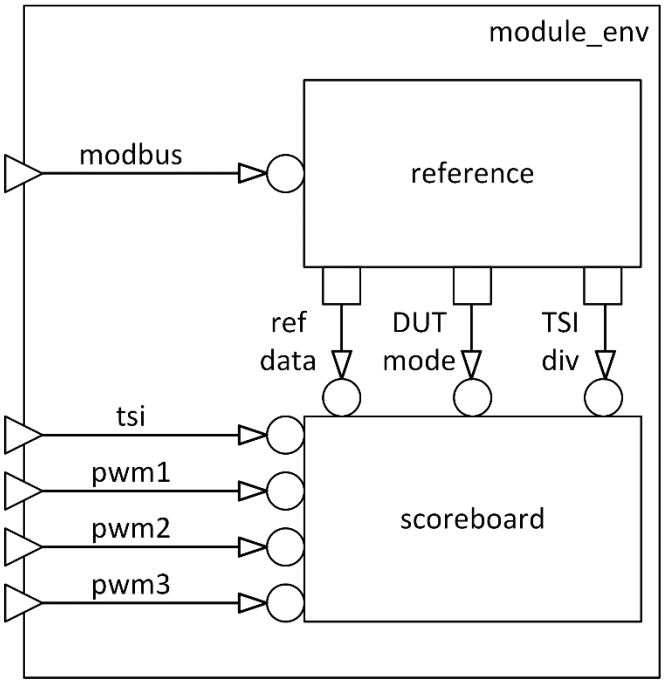

Fig. 5.8 Block diagram of scoreboard module architecture

The header of the packet should be identical based of the configuration of the packets, and it is always compared when new packet arrives. Each payload word of the packet should be comparable with the reference data, and it is checked that they match.

The real value of the voltage on each PWM line is counted with each period of the PWM signal based on the observed duty cycle. The count is performed inside implementation function of analysis ports connected to monitors of PWM UVCs. The comparison with the reference data is performed during the comparison with TSI packet, meaning that not every single PWM period is checked. It is checked that the counted real value matches reference data with a smaller deviation than the value of LSB.

The scoreboard is checked that it does not contain any reference data left to compare at the end of the simulation during the UVM check phase. During the UVM report phase, the scoreboard reports whether the simulation was successful or not base on internal counters of detected errors. Also, the UVM mechanism reports number of warnings and errors that occurred during the simulation.

5.2.2 Testbench environment and virtual sequencer

Testbench environment component contains instantiations of all the UVCs and scoreboard module used in the verification environment. Also, instantiation of a virtual sequencer is included as shown on 5.1. This virtual sequencer is used to synchronize behavior of sequencers of individual UVCs. All the tests used during the verification process are sequences called on the virtual sequencer.

All the components are created by the UVM Factory during the build phase. During the connect phase the connections between virtual sequencer and all other sequencers are created. Also, connections between analysis ports of monitors and analysis exports of scoreboard module are created.

5.2.3 Top modules

The verification environment is split into two top modules for better clarity. **Hardware top module** contains instantiations of the DUT and all the interfaces needed for the connection between the TB and the DUT. Each physical port of the DUT is connected to the specific interface here. The hardware top module also contains basic clock and reset module which generates initial reset of the DUT and TB and generates clock signal.

Testbench top module is the second top module used in the verification environment and it is highest in the environment's hierarchy. All the files necessary for the verification are imported into this module. It contains initial block that is started at the beginning of each simulation. In this block configuration of the UVCs' virtual interfaces is set, the clock signal and initial reset are kickstarted, and the test is started by UVM *run_test* task from this block.

5.3 Tests

The following tests were implemented for a basic functional verification of an example BLDC motor control system. The goal of these tests is to run the DUT in different modes and verify the function of these modes. Tests are implemented as virtual sequences that shall be run on virtual sequencer.

5.3.1 PWM test

PWM test checks basic function of PWM generator with direct setting of duty cycle. The test does not randomize the values of duty cycle on each PWM line. The values are fixed on different corner case values to cover all corner cases with single run of the test. During this test configuration of TSI must be performed via Modbus as it is needed to configurate number of packets sent by the DUT as it is not limited with the size of reference data file.

The test writes the values to the DUT's memory via Modbus to correct addresses. These addresses are not recognized by the scoreboard reference and the values are passed to the scoreboard by the UVM configuration database. Then the test enables PWM test mode on the DUT via Modbus command.

When the scoreboard reference detects this command, it builds a virtual TSI packets that serves as a reference data as there are no external reference data used for this test. Those packets are filled with the values of duty cycles for each PWM line and passed to scoreboard. The scoreboard checks the same value is received from the DUT as it was set from TB. The test ends after the scoreboard receives specified number of TSI packets.

5.3.2 Open loop test

This test verifies the function of the open control loop. The open loop parameters, such as an acceleration or a target frequency, are stored inside the controller's memory and they must be set the same as the setting in the golden reference model for this test. For this reason, these parameters are stored inside the NVM and are not configurated by the test. The output reference data are extracted from the golden reference and are stored in a separate file.

The test passes the reference data file to the UVM configuration database for the scoreboard module to prepare the reference TSI packets. The test starts the open loop mode of the controller via Modbus interface and waits until the scoreboard checked all the reference data it has prepared.

5.3.3 FOC test

The FOC test is the most complex test used as it verifies the function of FOC speed control loop. The FOC method requires feedback current from two phases of the motor to perform necessary transforms related to this commutation method. The controller is able to use any combination of two out of three phase currents available. As the data produced

by the golden reference is tied to the selection of which combination is used, only one set of data has been generated to test this operation mode.

The generated reference data consist of three separate files. First file is the input data for the ADC UVC that contains data of all the phase currents. Second file contains speed and position data for the Observer. The last file contains output reference data of the phase voltages for the scoreboard.

The DUT sequentially changes the combination of the phase currents it uses in its calculations based on current setting of the PWM signal and, hence, it is not possible to configurate this combination to be the same as in the golden reference for the specific set of data. It is necessary to force the value of an internal switch signal via a hierarchical access for the whole time the simulation runs with a single set of reference data.

The test starts from operation mode of the controller called NOOP. As the first step the Observer is enabled so it can inject position and speed reference data. The test sends command to start the FOC speed control loop and waits until the PWM UVC detects start of PWM signal to synchronize the input data for the DUT. When the PWM signal starts, the value of the switch is forced and the sequence on the ADC UVC is started to load the input data. The Observer detects the start of PWM signal autonomously to inject the first data and it is not needed to start it synchronously. The test runs until there are no more reference data inside the scoreboard. At the end the forced value of the switch is released, and the Observer is disabled.

5.3.4 Active braking test

Active braking mode is a special mode of the motor controller that requires testing. This mode is configurable in a similar way to the open loop mode. The configuration consists of different PWM duty cycle parameters and it has to correspond to the configuration of the golden reference model. This configuration is set in the NVM content, and the test passes the output reference data file to the scoreboard module via the UVM configuration database and starts the active braking mode on the DUT. The test finishes when the scoreboard has no more reference data to check.

5.3.5 FDIR threshold test

This test is not designed to check the function of the Motor control block, but to check the correct behavior of the function of the whole DUT. The controller can be configurated with different thresholds to limit the operation modes of the motor during extreme conditions. As an example of the thresholds a temperature setting has been chosen to test. The temperature can be read by the DUT from the ADC.

If the motor controller is in any operation mode and the set threshold value is exceeded for a configurated number of synchronization periods, the controller is expected to switch to a failure mode. The test starts with NOOP operation mode of the controller.

The temperature threshold is set via Modbus interface to specific value and specific timeout.

This function is tested in the open control loop mode. The testing is performed with setting the temperature value inside the ADC UVC over the threshold value for an incremental number of control periods. After each time the temperature value drops below the threshold. The test checks that the controller does not switch to the failure mode when the temperature exceed the threshold for shorter time than it was configurated at the beginning of the test. It also checks that the DUT reports failure mode when the temperature exceeded the limit for one period more than it was configurated.

6 VERIFICATION RESULTS

The functional verification of a motor control block of an example BLDC motor control system has been performed with HDL simulator QuestaSim. The verification process was limited only to RTL. Gate-level simulations were not included in the process. All of the tests described in chapter 5.3 were executed, with each test passing without detecting any errors. This testing provides confidence in the functionality of the motor control block within the tested system.

The verification environment is designed and ready to be used for a basic functional verification of a similar project. The environment is able to communicate with the DUT via Modbus interface where it can perform read and write operations. The NVM UVC models function of external NVM, it stores default DUT's configuration and passively communicates with the DUT. The ADC UVC interacts with the DUT as a responder, and it passes the reference response data from the motor's model to the controller. The environment monitors the PWM output of the DUT and also transactions on TSI to compare the telemetry against reference data.

This environment offers flexibility for modification and extension for next revision of the project or for different one. Minor internal update of the system, such as a modification of the used motor control algorithm, does not require an update of the environment. The only thing necessary to verify the function is to regenerate the reference data and update the path to this data in the tests used.

In case of major design modification in the system, the verification environment has to be modified as well. For instance, the design can be updated to incorporate a different communication interface. In that case the verification environment would require a new UVC for this interface, connection of this UVC in the environment and modification of the tests to use sequences of this UVC.

While the designed top-level components are not easily reused as they are built for project-specific use, they can be used as a template or be modified to work properly in different projects. Also, the observer component exhibits slightly reduced reusability as it forces values hierarchically to DUT's memory. This could potentially pose a challenge in the gate level simulations where a modification of the path to the memory might be required according to the netlist.

In this case, the benefit of using the UVM is mainly future proofing the verification environment with the possibility to extend the environment quite easily for comprehensive testing of all the components of the motor control system or to scale up the environment for additional functions in the future design. The unified structure of the verification environment also brings the benefit of easier understanding of the environment to anyone else.

Using UVM also brings some benefits during debugging the tests. One of the benefits is using transactions recording. This transaction can be visualized during waveform

debugging and an example of a Modbus TX write packet visualized in Figure 6.1. It is much easier to understand the content of the packet directly from waveform compared to deciphering the content solely from the TX wire behavior.

Furthermore, designed environment provides a quick evaluation of the tests with the UVM summary report at the end of the log of each test and the standardized structure of the log file. As the designed environment uses only UVM macros to write info messages to console or put out warnings and errors, the summary report will quantify each by the categories and report it in the log file.

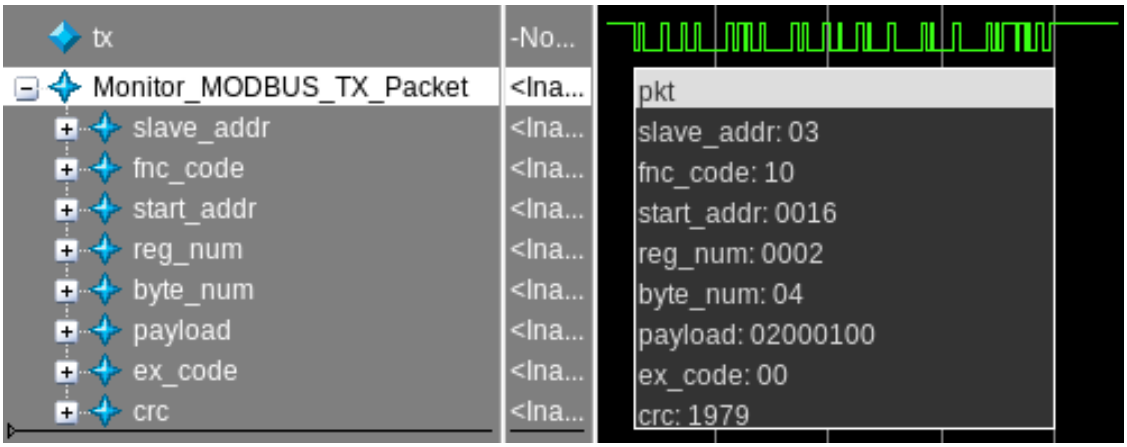

Fig. 6.1 Example of recording Modbus transaction

However, it is appropriate to consider the suitability of using the UVM in the context of this particular project. The relatively small size and simplicity of the DUT do not necessarily need to utilize the UVM, which typically excel with larger and more complex designs. The overhead associated with the UVM seems to be excessive for a project of this scale.

The response of the motor, which cannot be randomized and has to be generated from an external reference model, limits the application of the UVM. It is not possible to benefit from randomized stimulus generation or constraint-driven testing in this context when the behavior of the DUT is deterministic and predictable. The verification is focused on a specific use-case scenarios and functional correctness and, in this case, the UVM provides mainly a structured and systematic framework for the verification.

Additionally, it is worth noting that the simulations take longer compared to direct testing methods used previously to verify the same design. This increase in simulation duration can be caused by utilizing more complex components in the verification environment with more overhead needed due to the UVM.

7 CONCLUSION

This work discusses verification of digital circuits based on UVM. First part of this thesis presents verification and basics of UVM which is the industry standard for functional verification. This methodology is heavily used for its reusability, but the main issue is the initial creation of the testbench as it takes a lot of time and effort to switch from older approach of using direct testing methods. Once the testbench is created, it is easy to modify and reuse for similar projects or some parts can be reused for completely different designs which saves time and effort in the long run.

The following part of this work introduced a typical structure of BLDC motor controller which implement FOC commutation method. This controller is designed for implementation in FPGAs. The main focus of this work is on designing verification environment for verification of the Motor Control block function. To verify the function correctly, data from golden reference model must be extracted and imported to the testbench. Even though this work does not focus on verifying other blocks of the controller, the testbench must be able to communicate with the DUT and therefore must include UVCs for each communication interface.

The designed verification environment and all the components used are described in the chapter 5. The environment is working correctly. The verification of the example system has passed without detecting any errors during the designed tests. The advantage of using the UVM for this specific project is mainly the future proofing of the verification environment for extension and modification during different projects. Also, it is easier to comprehend due to the unified structure of UVM. Furthermore, it brings some benefits during debugging and test evaluation, such as recording transactions and report summary.

The main disadvantage of using the UVM, which is long development of the verification environment, has been taken away by this work. The UVM environment is resource intensive, which causes the simulation runtime to increase. In addition to that, the verification process cannot benefit much from randomized stimulus generation or constraint-driven testing for verification of the Motor Control block. This is mostly caused by the deterministic response from the motor which has to be generated from the reference model. Despite these drawbacks, the overall benefits of using the UVM prevail, especially in the context of the future development of the verification environment.

LITERATURE

[1] *BERGERON, Janick. Writing testbenches: functional verification of HDL models*. Boston: Kluwer Academic, 2000. ISBN 0-7923-7766-4.

[2] *TYPES OF COVERAGE METRICS. THE ART OF VERIFICATION* [online]. 2021 [cit. 2023-06-28]. Available from: https://www.theartofverification.com/types-of-coverage-metrics/

[3] *UVM Cookbook. Verification Academy* [online]. Plano, TX, USA: Siemens, 2021 [cit. 2023-05-31]. Available from: https://verificationacademy.com/cookbook/uvm

[4] *Universal Verification Methodology (UVM) 1.2 Class Reference*. *Accellera.org* [online]. Elk Grove, CA, USA: Accellera Systems Initiative, 2014 [cit. 2023-05-31]. Available from: https://www.accellera.org/downloads/standards/uvm

[5] *RENESAS ELECTRONICS CORPORATION. What are Brushless DC Motors. Renesas.com* [online]. 2023 [cit. 2023-10-21]. Available from: https://www.renesas.com/us/en/support/engineer-school/brushless-dc-motor-01-overview

[6] *LEE, S., T. LEMLEY a G. KEOHANE. A comparison study of the commutation methods for the three-phase permanent magnet brushless DC motor.* Electrical Manufacturing Technical Conference 2009: Electrical Manufacturing and Coil Winding Expo EMCWA 2009. 2009, 49-55. Available also from: https://www.magnelab.com/wp-content/uploads/2015/02/A-comparison-study-of-the-commutation-methods-for-the-three-phase-permanent-magnet-brushless-dc-motor.pdf

[7] *INTEGRA SOURCES. BLDC Motor Controller: How It Works, Design Principles & Circuit Examples.* Integrasources.com [online]. 2021, Last update: 2023-09-22 [cit. 2023-10-21]. Available from: https://www.integrasources.com/blog/bldc-motor-controller-design-principles/

[8] *DIGIKEY. How to Power and Control Brushless DC Motors.* Digikey.de [online]. 2016, 2016-12-07 [cit. 2023-10-21]. Available from: https://www.digikey.de/de/articles/how-to-power-and-control-brushless-dc-motors

[9] *KALOCSÁNYI, Vít. Návrh aritmetické jednotky v pevné řádové čárce pro obvody FPGA*. Brno, 2022. Available also from: https://www.vutbr.cz/studenti/zav-prace/detail/142779. Bachelor‘s thesis. Brno University of Technology, Faculty of electrical engineering and communications, Dept. of microelectronics, Advised by Vojtěch Dvořák.

[10] *MODBUS Application Protocol Specification V1.1b3.* Modbus.org [online]. Andover, MA, USA: Modbus Organization, Inc., 2012-04-26 [cit. 2024-01-23]. Available from: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf

[11] *EVERSPIN TECHNOLOGIES. MR0A08B* [online]. 2021 [cit. 2024-01-23]. Available from: https://www.everspin.com/getdatasheet/MR0A08B

[12] *TEXAS INSTRUMENTS. ADC128S102QML-SP Radiation Hardened 8-Channel, 50 kSPS to 1 MSPS, 12-Bit A/D Converter* [online]. 2008, rev 2017-04 [cit. 2024-01-26]. Available from: https://www.ti.com/lit/gpn/adc128s102qml-sp

SYMBOLS AND ABBREVIATIONS

Abbreviations:

ADC	Analog-to-digital converter
ASIC	Application specific integrated circuit
AVM	Advanced Verification Methodology by Mentor
BLDC	Brushless direct current
CE	Chip Enable
CRC	Cyclic redundancy check
DC	Duty Cycle
DIN	Data input
DOUT	Data output
DT	Dead Time
DUT	Design under test
eRM	The e Reuse Methodology by Verisity
FDIR	Failure detection, isolation, and recovery
FOC	Field-oriented control
FPGA	Field programmable gate array
FSM	Finite state machine
HDL	Hardware description language
HVL	Hardware verification language
IC	Integrated circuit
LSB	Least significant bit
MCU	Microcontroller unit
NOOP	No-operation mode
NVM	Non-volatile memory
OE	Output Enable
OVM	Open Verification Methodology by Mentor and Cadence
PWM	Pulse-width modulation
RTL	Register-transfer level
SCLK	Serial clock
SPI	Serial Peripheral Interface
TLM	Transaction level modelling
TSI	Telemetry Stream Interface
UART	Universal asynchronous receiver/transmitter
UVC	UVM component
UVM	Universal Verification Methodology
VMM	Verification Methodology Manual by Synopsys
WE	Write Enable

LIST OF APPENDICES

Appendix A - Attached directory structure

```
/.....................................Root directory of the appendix
├─ uvm.................................Designed UVM environment directory
│  ├─ adc128.....................................ADC UVC source files
│  │  ├─ sv
│  │  │  ├─ adc128_agent.sv
│  │  │  ├─ adc128_driver.sv
│  │  │  ├─ adc128_env.sv
│  │  │  ├─ adc128_if.sv
│  │  │  ├─ adc128_monitor.sv
│  │  │  ├─ adc128_packet.sv
│  │  │  ├─ adc128_pkg.sv
│  │  │  ├─ adc128_seqs.sv
│  │  │  ├─ adc128_sequencer.sv
│  │  │  └─ timing_macro_uvm.inc
│  ├─ modbus.....................................Modbus UVC source files
│  │  ├─ sv
│  │  │  ├─ modbus_env.sv
│  │  │  ├─ modbus_if.sv
│  │  │  ├─ modbus_packet.sv
│  │  │  ├─ modbus_pkg.sv
│  │  │  ├─ modbus_rx_agent.sv
│  │  │  ├─ modbus_rx_monitor.sv
│  │  │  ├─ modbus_tx_agent.sv
│  │  │  ├─ modbus_tx_driver.sv
│  │  │  ├─ modbus_tx_monitor.sv
│  │  │  ├─ modbus_tx_seqs.sv
│  │  │  └─ modbus_tx_sequencer.sv
│  ├─ nvm........................................NVM UVC source files
│  │  ├─ sv
│  │  │  ├─ nvm_agent.sv
│  │  │  ├─ nvm_content.txt
│  │  │  ├─ nvm_driver.sv
│  │  │  ├─ nvm_env.sv
│  │  │  ├─ nvm_if.sv
│  │  │  ├─ nvm_monitor.sv
│  │  │  ├─ nvm_packet.sv
│  │  │  ├─ nvm_pkg.sv
│  │  │  ├─ nvm_seqs.sv
│  │  │  ├─ nvm_sequencer.sv
│  │  │  └─ timing_macro_uvm.inc
│  ├─ obs..................................Observer interface source file
│  │  ├─ sv
│  │  │  └─ obs_if.sv
│  ├─ rep.........................Project specific components source files
│  │  ├─ sv.....................................Scoreboard module source files
│  │  │  ├─ rep_module_env.sv
│  │  │  ├─ rep_module_pkg.sv
│  │  │  ├─ rep_reference.sv
│  │  │  └─ rep_scoreboard.sv
│  │  ├─ tb.........................Testbench top components source files
│  │  │  ├─ hw_top.sv
│  │  │  ├─ rep_tb.sv
│  │  │  ├─ rep_test_lib.sv
│  │  │  ├─ rep_uvm_conf_pkg.sv
│  │  │  ├─ rep_virtual_seqs.sv
│  │  │  ├─ rep_virtual_sequencer.sv
│  │  │  └─ tb_top.sv
```


pwm.....PWM UVC source files
sv
pwm_agent.sv
pwm_env.sv
pwm_if.sv
pwm_monitor.sv
pwm_packet.sv
pwm_pkg.sv
tsi.....TSI UVC source files
sv
tsi_agent.sv
tsi_env.sv
tsi_if.sv
tsi_monitor.sv
tsi_packet.sv
tsi_pkg.sv
ver.....Verification directory
logs.....Logs from executed tests
pwm_test.log
ol_test.log
foc_test.log
ab_test.log
fdir_test.log
scripts.....Script to start simulation in Questa
all.do
README.txt

Appendix B - Test execution

A.1 How to execute tests

Simulation tool and version: Questa Sim-64, Version 2023.4 linux_x86_64 Oct 9 2023
UVM version: 1.1d

NOTE: DUT compilation is not included in the *all.do* script file and DUT source files are not part of the attachment. The design must be compiled separately, or the script must be modified. The simulation tool must have access to the complied design. Also, the golden reference model and generated reference data are not part of the attachment. The verification environment must have access to the reference files. Paths to the files has to be modified inside the corresponding tests in **./uvm/rep/tb/rep_virtual_seqs.sv**

Execution steps:

1. Modify *simple_vseqr_test* to execute required test sequence
    a. Open file **./uvm/rep/tb/rep_test_lib.sv**
    b. Change the name of the test sequence in *uvm_config_wrapper* inside build phase of *simple_vseqr_test* class, line 74
2. Open simulation tool with terminal opened inside **./ver/scripts**/ folder
3. Execute the all.do script using Questa terminal with the following command: **do all.do**

A.2 Table of executed tests

Test	Sequence name	Status	Log file
PWM	test_run_pwm_vseq	Passed	./ver/logs/pwm_test.log
Open Loop	test_run_ol_vseq	Passed	./ver/logs/ol_test.log
FOC	test_run_foc_sw0_vseq	Passed	./ver/logs/foc_test.log
Active Braking	test_run_ab_vseq	Passed	./ver/logs/ab_test.log
FDIR Threshold	test_run_fdir_th_vseq	Passed	./ver/logs/fdir_test.log