UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DIGITAL CIRCUIT VERIFICATION IN THE AUTOMOTIVE SPACE USING UVM

Diploma Thesis

Sotirios Giannakos

Supervisor: Nikolaos Bellas

January 2025

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

DIGITAL CIRCUIT VERIFICATION IN THE AUTOMOTIVE SPACE USING UVM

Diploma Thesis

Sotirios Giannakos

Supervisor: Nikolaos Bellas

January 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΠΑΛΗΘΕΥΣΗ ΨΗΦΙΑΚΩΝ ΚΥΚΛΩΜΑΤΩΝ ΣΤΗΝ ΑΥΤΟΚΙΝΗΤΟΒΙΟΜΗΧΑΝΙΑ ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ UVM

Διπλωματική Εργασία

Σωτήριος Γιαννακός

Επιβλέπων/πουσα: Νικόλαος Μπέλλας

Ιανουάριος 2025

Approved by the Examination Committee:

Supervisor **Nikolaos Bellas**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Christos Antonopoulos**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Spyros Lalis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

I would like to thank Professor Nikos Bellas for his continued help and support throughout this endeavor. I would also like to thank my family for giving me the courage to pursue my interests no matter the circumstances. Finally I thank A. for making my days better and always being there for me through these years.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Sotirios Giannakos

Diploma Thesis

DIGITAL CIRCUIT VERIFICATION IN THE AUTOMOTIVE SPACE USING UVM

Sotirios Giannakos

Abstract

In today's automotive industry, the widespread use of digital circuits has necessitated thorough testing and verification of these designs. This thesis aims to demonstrate a method for verifying the correctness of a digital circuit using UVM (Universal Verification Methodology). Firstly, we discuss the added features of SystemVerilog and the basic classes found in UVM. We then analyze a given SPI (Serial Peripheral Interface) to APB (Advanced Peripheral Bus) converter and build an environment that will be able to cover most of its functions. Our approach is a bottom-up one, where we build small separate UVM environments for the various interfaces of our Device Under Test (DUT) and then connect them to a larger environment where we instantiate our DUT and check its output. By using XCELIUM and VManager we are able to showcase the results of our simulations and how far we came in terms of functional coverage. Finally we list the requirements that were met regarding the various DUT interfaces.

Keywords:

SystemVerilog, UVM, SPI, APB, interface, coverage, classes

Διπλωματική Εργασία

ΕΠΑΛΗΘΕΥΣΗ ΨΗΦΙΑΚΩΝ ΚΥΚΛΩΜΑΤΩΝ ΣΤΗΝ ΑΥΤΟΚΙΝΗΤΟΒΙΟΜΗΧΑΝΙΑ ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ UVM

Σωτήριος Γιαννακός

Περίληψη

Στις μέρες μας η ευραία χρήση ψηφιακών κυκλωμάτων στην αυτοκινητοβιομηχανία έχει φέρει στο προσκήνιο την ανάγκη για τον έλεγχο και την επαλήθευση τους. Σε αυτή την διπλωματική εργασία ο σκοπός μας είναι να αναδείξουμε ένα τρόπο επαλήθευσης ενός τέτοιου κυκλώματος μέσω της χρήσης UVM (Ενιαία Μεθοδολογία Επαλήθευσης). Αρχικά, παρουσιάζουμε τα χαρακτηριστικά της SystemVerilog και τις βασικές κλάσεις της UVM που θα χρησιμοποιήσουμε. Στην συνέχεια αναλύουμε την λειτουργία ενός μετατροπέα SPI (Σειριακή Περιφερειακή Διεπαφή) σε APB (Προηγμένος Περιφερειακός Δίαυλος) και σχεδιάζουμε ένα περιβάλλον το οποίο θα καλύπτει τις περισσότερες πιθανές χρήσεις του κυκλώματος αυτού. Για τον σκοπό αυτό ακολουθούμε μια κάτω προς πάνω προσέγγιση, όπου σχεδιάζουμε πολλαπλά μικρά UVM περιβάλλοντα για τις διάφορες διεπαφές της συσκευής μας και στην συνέχεια τα συνδέουμε σε ένα μεγαλύτερο περιβάλλον το οποίο με την σειρά του συνδέεται με την συσκευή και ελέγχει την ορθότητα των αποτελέσματων. Χρησιμοποιώντας το XCELIUM και το VManager μπορούμε να αποδώσουμε τα αποτελέσματα των διαφόρων προσομοιώσεων και πόσες από τις λειτουργικές απαιτήσεις των διαφόρων διεπαφών και σημάτων καταφέραμε να καλύψουμε.

Λέξεις-κλειδιά:

SystemVerilog, UVM, APB, διεπαφή, κλάσεις

Table of contents

List of figures

List of tables

Abbreviations

e.g.	for example
UVM	Universal Verification Methodology
SPI	Serial Peripheral Interface
APB	Advanced Peripheral Bus
DUT	Device Under Test
MOSI	Master Out Slave In
MISO	Master In Slave Out
CPOL	Clock Polarity
CPHA	Clock Phase
CRC	Cyclic Redundancy Check
HNDSHK	Handshake
SPI_SYS_CTRL	SPI System Control
TM_EN_CTRL	Test Mode Enable Control

Chapter 1

Introduction

In the automotive space, to verify the correctness of a digital circuit, we use a testbench. Through the testbench, we drive signals to the ports of our design and check its response. This task is relatively simple for small designs, however in real life applications, the process of writing a testbench can prove to be tedious and time-consuming. Nowadays companies employ testing and verification engineers, whose only job is to verify the correctness of a given circuit. This circuit is normally a small part of a larger design and can be used in more than one project, which leads to the issue of reusability. Engineers face the challenge of testing various components independently and in a way that these tests can be used in the future in different designs.

1.1 The Subject of This Thesis

In this thesis we will showcase how the aforementioned problems can be solved through the usage of UVM, by trying to verify a large and significant part of a circuit used in the automotive industry. We will try to break down our verification environment in smaller reusable parts and also to solve the scaling problem that arises when designing a common Verilog testbench for a large circuit with numerous ports.

1.1.1 Contributions

The contributions of this thesis can be summarized as:

1. A quick overview of SystemVerilog and UVM

2. An overview of the SPI protocol
3. Usage of UVM for building a testbench environment
4. Usage of XCELIUM tool for simulation purposes
5. Usage of VManager tool for functional coverage metrics
6. Verification of an SPI to APB circuit

1.2 Thesis Organization

In Chapter 2 we present the additions to SystemVerilog that make it more suited for complex verification environments and the basic UVM objects and components that will be used for the purpose of this thesis. Chapter 3 presents our verification environment with a bottom-up approach after explaining the function of our DUT and its numerous ports. The results of multiple simulations are showcased in Chapter 4 along with a quick explanation of the two tools we used to get them. The specific requirements we managed to cover and the tests we used are also listed on this Chapter. Important SystemVerilog code for the creation of our components is showcased in the Appendices section.

Chapter 2

SystemVerilog and UVM

2.1 Introduction

SystemVerilog is a language that was built upon Verilog-2001 with the desire to improve the productivity, readability and reusability of Verilog based code. SystemVerilog features some extended data types from Verilog, C like void functions, enhanced process control, classes and introduces coverage. [1]

In 2009 the development of a Universal Verification Methodology (UVM) was proposed [2]. UVM is essentially a library of SystemVerilog classes that we use as the bases of our verification environment. It helps us structure reusable environments for multiple components quickly, due to the usage of the aforementioned classes and their methods.

In this chapter, we showcase the additions to SystemVerilog that are regularly used in this project, the basic UVM classes and the structure of a generic UVM environment.

2.2 SystemVerilog

SystemVerilog's data types are a combination of Verilog and C data types. It introduces new integer types like the bit, byte and logic types. Bit and byte are two-state data types which means that they can only have a value of 0 or 1 , while logic is a four-state data type and can also have a value of X or Z. The void data type is used similarly to C and is often used in UVM built-in methods.

The `always` block from Verilog is expanded upon through the introduction of the `always_comb`, `always_ff` and `always_latch` blocks . The differences between an

`always_comb` block and an `always @*` block are [1]:

- `always_comb` automatically executes once at time zero, whereas `always @*` waits until a change occurs on a signal in the inferred sensitivity list.
- `always_comb` is sensitive to changes within the contents of a function, whereas `always @*` is only sensitive to changes to the arguments of a function.
- Variables on the left-hand side of assignments within an `always_comb` procedure, including variables from the contents of a called function, shall not be written to by any other processes, whereas `always @*` permits multiple processes to write to the same variable.

The introduction of `join_any` and `join_none` enables finer control over process execution. When `join_any` is used, the parent process pauses and waits until at least one of the child processes finishes. With `join_none`, the parent process does not wait for any child processes and proceeds immediately with its execution. [1].

SystemVerilog interfaces also enable greater control over a module's ports. An interface is a block which can encapsulate multiple ports and also include tasks, functions, parameters and variables. Interfaces are a big contributor to the reusability that SystemVerilog offers and it is always preferable to view a design's ports through the lens of potential interfaces. We can also define port directions through `modport` as shown in the example below [3]:

```
interface myBus (input clk);
    logic [7:0]  data;
    logic        enable;

    // From TestBench perspective, 'data' is input and
    // 'write' is output
    modport TB  (input data, clk, output enable);

    // From DUT perspective, 'data' is output and
    // 'enable' is input
    modport DUT (output data, input enable, clk);
endinterface
```

Randomization is also a big part of SystemVerilog's appeal for the purposes of testing. With the keyword `rand` we can enable the randomization of a certain variable and have way bigger coverage of potential values without changing its value manually. If for one test or sequence we desire a specific value for a randomized variable we use the `constraint` block. An example is shown below [4]:

```
class seq_item #(parameter int p1 = 10, p2 = 20);
  rand bit [7:0] value1;
  rand bit [7:0] value2;
  rand bit [7:0] value3;
  rand bit [7:0] value4;
  rand bit [7:0] value5;
  rand bit [7:0] value6;
  rand bit [7:0] value7;

   // constant value based range
  constraint value1_c {value1 inside {[10:20]};}
  // Set of values
  constraint value2_c {value2 inside {40,70, 80};}
  // Mix
  constraint value3_c {value3 inside {[10:20], 21, 23, [25:30],
                                      40, 70, 80};}
  // Inverted Range
  constraint value4_c {!(value4 inside {[100:200]});}
  // range using variable
  constraint value5_c {value5 inside {[value1:value2]};}
  // Define based range
  constraint value6_c {value6 inside {[`START_RANGE:`END_RANGE]};}
  // parameter based range
  constraint value7_c {value7 inside {[p1:p2]};}
endclass
```

One of the biggest additions to SystemVerilog was Object Oriented Programming building blocks such as objects and classes. The existence of classes in SystemVerilog, which

enabled the creation of UVM, has made the latter's features possible; these features will be detailed in the next subsection of the chapter.

Finally SystemVerilog introduces the idea of coverage. Coverage shows us how many of the design specifications have been verified by a test or multiple tests. Coverage metrics can either derive from the actual design code like code coverage or be user specified with the goal of meeting the given requirements. The latter is achieved through the usage of the `covergroup` structure. A `covergroup` can contain one or more coverage points. A coverage point can refer to a variable or an expression. Each coverage point includes a set of bins associated with its sampled values or its value-transitions. The bins can be explicitly defined by the user or automatically created by the tool [1]. An example [5] of user defined coverage is shown below:

```
class myTrns;
  rand bit [3:0] mode;
  rand bit [1:0] key;

  covergroup CovGrp;
    coverpoint mode {
        bins featureA   = {0};
        bins featureB   = {[1:3]};
        bins common []  = {4:$};
        bins reserve    = default;
    }
    coverpoint key;
  endgroup
endclass
```

2.3 UVM

As mentioned above UVM is a library of classes that are used as the building blocks of our verification environment. In this section we will approach this environment with a bottom up perspective.

2.3.1 UVM Driver

A big part of every testbench is providing stimulus to the DUT (Device Under Test). In UVM driving takes place in the `uvm_driver` class. The driver receives data and transmits it to the DUT via ports, which are usually defined within a SystemVerilog interface.

2.3.2 UVM Monitor

The `uvm_monitor` class, like its name suggests, monitors the interface that connects the DUT and the driver. Every time the driver sends data or the DUT responds, the monitor captures the data and sends it to the scoreboard through an analysis port. The monitor can check the data's correctness before sending it to the scoreboard or withholding it entirely.

2.3.3 UVM Sequences and Sequence Items

The class that facilitates the generation of a stimulus for the DUT is the `uvm_sequence` class. The stimulus is created in the form of a `uvm_sequence_item`. A sequence can create multiple sequence items with multiple, different and sometimes randomized values.

2.3.4 UVM Sequencer

This transaction of sequence items is realized through the `uvm_sequencer`, who is connected to the driver through a port called `uvm_seq_item_pull_export`.

2.3.5 UVM Agent and Environment

All of the components mentioned above are connected in the `uvm_agent` class. The `uvm_agent` is also facilitating the connection between the scoreboard and the monitor and can be in either an active or passive state. When active the agent consists of a driver, sequencer and monitor, while on a passive state it only contains a monitor. The agent is part of the `uvm_environment` component.

2.3.6 UVM Scoreboard

The `uvm_scoreboard` class is where we check the validity of the DUT output. This is usually achieved by producing the expected output using the given input and then comparing the result to the actual output. It is important to note that the scoreboard can be connected to multiple monitors at the same time.

2.3.7 UVM Test and Testbench

In the `uvm_test` component, one or multiple sequences are created and invoked according to our needs. The test is ran in the `uvm_testbench` class, in which the DUT instantiation resides as well. Figure 2.1 shows all the aforementioned components and their connections [6]:

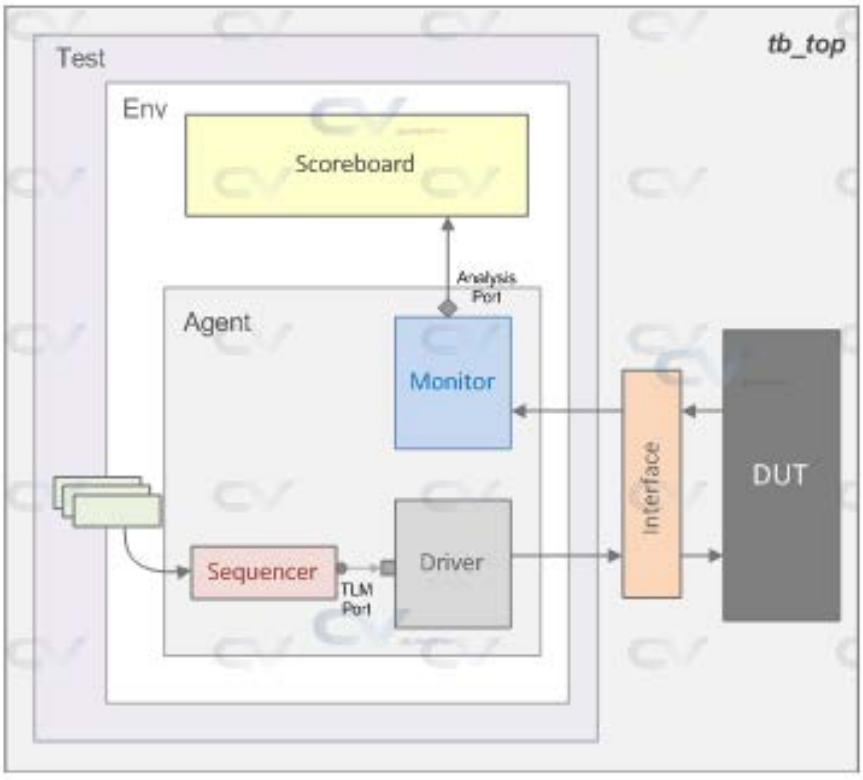

Figure 2.1: Testbench Top Architecture

2.3.8 UVM Phases

In UVM, a simulation is viewed in phases. These phases are functions or tasks that every UVM component recognizes and has to go through. In order for the simulation to progress to

the next phase, all components must complete the current phase. There are a lot of phases in UVM and the existence of user defined ones is also possible, but the ones that are used more frequently are [7]:

1. `build_phase`: Used to build testbench components and create their instances
2. `connect_phase`: Used to connect between different testbench components via TLM ports
3. `run_phase`: Actual simulation that consumes time happens in this UVM phase and runs parallel to other UVM run-time phases.
4. `report_phase`: Used to display result from checkers, or summary of other test objectives

2.3.9 UVM Factory and UVM Macros

The UVM Factory is responsible for the creation of all UVM objects that we want to create and it can also override the type of a component with the `set_type_override` command. While using UVM we might not be aware of the factory's existence, but we do use it indirectly all the time because it is the place where all objects are registered. In order to register those objects we use the UVM utility macros `` `uvm_object_utils `` and `` `uvm_component_utils ``, as well as the field macro `` `uvm_field_* `` that provides automatic implementations of some core methods [8]:

```
class ABC extends uvm_object;
    rand bit [15:0]     m_addr;
    rand bit [15:0]     m_data;

    `uvm_object_utils_begin(ABC)
        `uvm_field_int(m_addr, UVM_DEFAULT)
        `uvm_field_int(m_data, UVM_DEFAULT)
    `uvm_object_utils_end

    function new(string name = "ABC");
        super.new(name);
```

```
        endfunction
    endclass
```

We also use the `uvm_info and `uvm_error macros to print out general information, debugging information or error messages. The former macro also makes use of verbosity with which we denote the importance of a message. There are five types of verbosity:

- UVM_NONE: the message will always be printed.
- UVM_LOW: the message will be printed if verbosity is set to LOW or upwards.
- UVM_MEDIUM: the message will be printed if verbosity is set to MEDIUM or upwards.
- UVM_HIGH: the message will be printed if verbosity is set to HIGH or upwards.
- UVM_FULL: the message will be printed only when verbosity is set to FULL.

2.3.10 TLM Ports

Transaction Level Modeling (TLM) is an approach for creating abstract, high-level representations of systems and their components. In this methodology, data is structured as transactions—objects (often class-based) that encapsulate randomized, protocol-specific details. These transactions are transferred between components using dedicated communication channels referred to as TLM interfaces [9]. In this project we will see ports that are used by the scoreboard, the monitor and the agent and finally exports that are used by the sequencer. Analysis ports allow the monitor and the agent to send transactions with or without an actual target.

Chapter 3

Building our Environment

3.1 Our Device Under Test

The device under test or, as we will refer to it, the DUT, is an SPI to APB module. The DUT receives a 32-bit frame serially through the SPI interface, checks its correctness and then sends it to an APB component. We can also operate our device in a test mode through the usage of several `tm_en_ctrl` signals and request the parent SoC for resources through the `spi_sys_ctrl` signals. Finally, there are several sideband signals, such as the clock and reset signals, that are imperative for the correct functionality of the DUT. An overview of the device is shown in Figure 3.1. In the following sections there will be an overview of all these interfaces and how we test their functionality using UVM. We will create six environments that will each contain an agent for a corresponding interface.

3.2 The SPI Interface

The Serial Peripheral Interface (SPI) is a widely used standard for synchronous serial communication. It is based on a Master-Slave relationship in which the master sends a frame and then the slave replies accordingly. In our case the frame will be a 32-bit message. Our SPI interface consists of the following signals:

```
logic sck;
logic csn;
logic mosi;
logic miso;
```

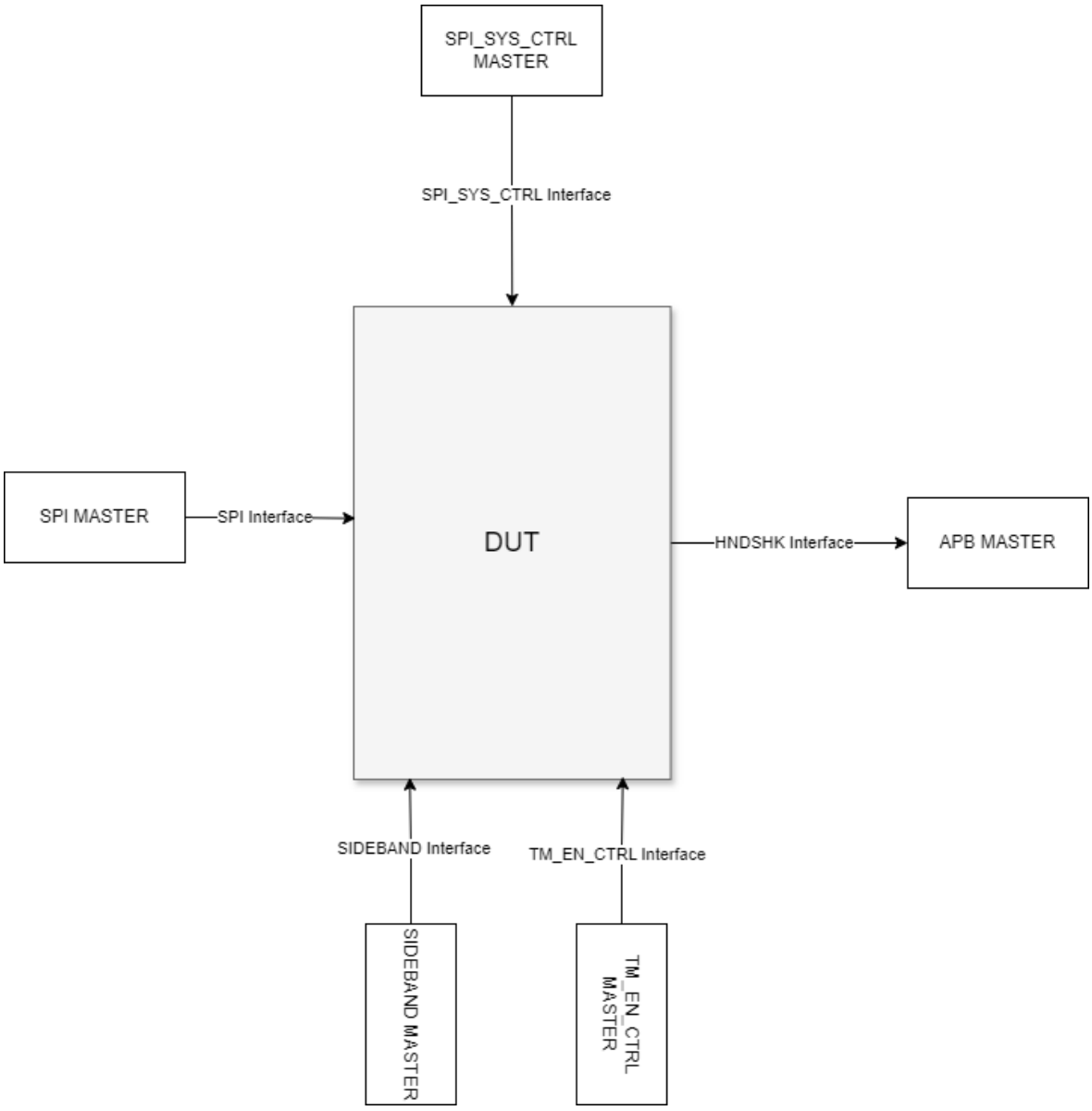

Figure 3.1: The SPI to APB DUT

```
logic miso_en;
```

The `sck` is the clock used for the bit-by-bit sending of our frame and it is different from the overall system clock that is part of the sideband interface. Chip selection is done through `csn` which in this case is active low. The master sends the frame through the `mosi` signal and the slave responds through `miso` only when `miso_en` is asserted. The testbench has the role of Master in this case and the DUT operates as Slave. The `mosi` and `miso` bits are sent at every clock cycle depending however on the phase and the polarity in which the SPI mechanism is operated on. Clock polarity (CPOL) and clock phase (CPHA) are terms set by Motorola [10]. Polarity denotes the idleness of the clock either on low voltage ($CPOL = 0$)

SPI Mode	CPOL	CPHA	Data sent on	Data sampled on
0	0	0	falling SCK and when CSN is asserted	rising SCK
1	0	1	rising SCK	falling SCK
2	1	0	rising SCK and when CSN is asserted	falling SCK
3	1	1	falling SCK	rising SCK

Table 3.1: Polarity and Phase combinations

or high voltage ($CPOL = 1$). When $CPHA = 0$ the first bit is sent immediately after the chip select signal is asserted, subsequent bits are sent when the clock transitions to its idle value and are sampled when it transitions from that same value. With $CPHA = 1$ the first bit is sent on the first clock edge after `csn` is asserted, the subsequent bits are sent when the clock transitions from its idle value and are sampled when it transitions to its idle value. Table 3.1 shows these characteristics and the four possible combinations that may occur. Our DUT specifically operates on SPI Mode number 1.

The structure and the contents of our 32 bit frames change according to the sender and the nature of the requested operation (read or write). The MISO frames have the same form in the case of either a read or a write as shown in Figure 3.2. The bit named OFC indicates a corrupted frame and bit 22 is zero in case of an error.

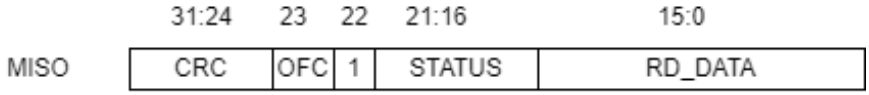

Figure 3.2: Standard MISO 32 bit frame

However, MOSI frames do change depending on the nature of the sent request. If it is a write request then the MOSI frame is as shown in Figure 3.3.

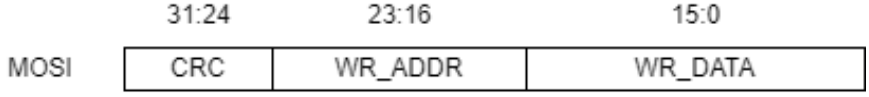

Figure 3.3: MOSI 32 bit frame in case of write

If we have a read request the WR_ADDR field has a value of $8'hFF$ and the WR_DATA field is now representative of the read address value as shown in Figure 3.4.

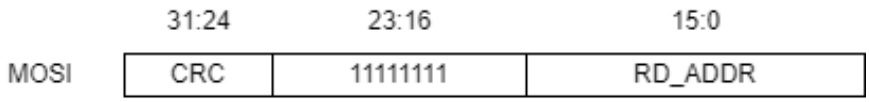

Figure 3.4: MOSI 32 bit frame in case of read

One field that is prevalent in all frame types is the CRC field. CRC (Cyclic Redundancy Check) helps us verify the correctness of the data in case of disturbances, due to electromagnetic radiation, that may occur in the environment inside the vehicle. CRC generates a checksum based on the data that are transmitted. This checksum is located on the CRC field that is showcased here. If after the transmission the CRC field does not match there is an error in the data. There are 3 big components that take part in the calculation of the CRC field:

- The generator polynomial is used for division and is associated with the error detection capabilities of the CRC. In our case we use CRC-8-SAE J1850 and our polynomial is $x^8 + x^4 + x^3 + x^2 + 1$ or $0x1D$ in binary form.
- We can initialize the computation with a non zero value which in our case is $0xFF$
- In the end we XOR the result with $0xFF$ as well

There are 4 CRC modes that our device can operate on, that change the data that CRC checks and if we ignore the check completely. These modes refer to the computation that our DUT does for the MISO field and are:

- CRC-24 mode. In CRC-24 mode the CRC field is calculated for the 24 bits of the message excluding the CRC field. The word therefore has the form of {frame[7:0], frame[15:8], frame[23:16]}.
- CRC-40 mode. In CRC-40 mode we calculate our value with a 40 bit word (hence the name) that, in case of a read, consists of {frame_{n-1}[7:0], frame_{n-1}[15:8], frame_n[7:0], frame_n[15:8], frame_n[23:16]}. In this case frame_{n-1} refers to a MOSI frame and frame_n refers to a MISO frame.
- NO-CRC mode. With NO-CRC mode in both MOSI and MISO frames the CRC field has a fixed value of $0xF1$.
- IGN-CRC mode. In IGN-CRC mode MOSI frames don't have a CRC field and the 8 most significant bits are used to extend the write address. MISO frames have a CRC

field that is calculated like we are in CRC-24 mode.

We have 2 functions that compute the CRC field for the 24 and 40 modes as shown in Appendix A.

3.2.1 The SPI Agent

The agent that is responsible for driving data to and monitoring data from the SPI interface is composed of the `spi_agent`, `spi_driver`, `spi_monitor` and `spi_sequencer` components. The sequences are located in an entirely separate file named `spi_seq_lib` and the sequence item that contains the data is called `spi_seq_item`. The first thing that we need to think about are the fields of the sequence item that will help us cover the needed requirements of the DUT. In this case our sequence item consists of:

```
rand logic [7:0] mosi_crc;
rand logic [7:0] miso_crc;
rand logic [7:0] write_addr;
rand logic [7:0] status;
rand logic [15:0] write_data;
rand logic [15:0] read_data;
rand logic [15:0] read_addr;
```

We have two different fields for the MISO and MOSI CRC due to potential differences between the two. The status field encapsulates bits 23 to 16 of the standard MISO frame in Figure 3.2. Pay attention to the `rand` descriptor. In this item we want all of these values to be randomized. This may not be the case in other instances that we will see below. Various sequences are placed in the `spi_seq_lib` file. We have eight sequences in order to cover all possible combinations of a read or write and a CRC mode. This leads to the usage of constraints in order to either signal a read operation or to signal to the driver the desired CRC mode. For example if we want to create a sequence item that will be used for a read frame on NO_CRC mode we will write:

```
`uvm_do_with(s_item, {s_item.write_addr == 8'hff;
                      s_item.read_addr <= 16'h01ff;
                      s_item.mosi_crc == 8'hf1;})
```

In our case while on IGN_CRC mode address extension is disabled therefore the CRC field will be zero. For a write frame on IGN_CRC we write:

```
`uvm_do_with(s_item, {s_item.write_addr <= 8'hfc;
                      s_item.mosi_crc == 8'h00;})
```

We have also created `mult_reads` and `mult_writes` sequences that create 1000 read frames and 100 write frames respectively, using `repeat(NUM)`. The creation of sequence items on a sequence object is done in a task called `body`.

As mentioned before the role of the driver is to drive all signals of a given interface. In the `spi_driver` we have to drive the clock, chip select and mosi signals. For these three signals we will need four separate tasks (two for chip select) to run simultaneously in different threads. We accomplish this by using a fork join in the `run_phase` task as shown:

```
task spi_driver::run_phase(uvm_phase phase);

  //-START-USER-REGION: "IMPLEMENTATION-RUN"
  init(phase);

  fork
    drive_clock(phase); // task for driving the spi clock
    drive_enable(phase); // task for initially asserting
                         // the csn signal
    drive_csn(phase); // task for asserting the csn signal in
                      // subsequent transactions
    get_and_drive(phase); // start transaction loop
  join
  //-END-USER-REGION: "IMPLEMENTATION-RUN"
endtask: run_phase
```

The `drive_clock` task is responsible for driving the `sck` signal of frequency $10\ MHz$ with a simple forever block:

```
forever #50ns vif.sck = ~vif.sck;
```

Notice that `vif` is a virtual interface handle to the SPI interface. The `drive_enable` task is for driving the `csn` signal after lowering the reset sideband signal. The reset signal is made

1 after 1000 ns and so `csn` is driven to 0 after 1050 ns. For subsequent assertions of `csn` we use the `drive_csn` task that in a forever block at every positive edge of `csn` drives it to 0 after 1250 ns:

```
task spi_driver::drive_csn(uvm_phase phase);
  // According to requirements we drop the csn 1250ns
  // after raising it
  forever begin
    @(posedge vif.csn)
    #1250ns vif.csn = 0;
  end
endtask
```

Finally the `get_and_drive` task contains a forever loop in which we pull an item from the sequencer, drive the data in a task named `drive` and then signal the end of this transaction to the sequencer:

```
task spi_driver::get_and_drive(uvm_phase phase);
  REQ req;
  RSP rsp;

  forever begin
    seq_item_port.get_next_item(req); // pull item from sequencer
    drive(req, rsp);  // drive transaction to interface
    seq_item_port.item_done(rsp); // signal end of transfer to
                                  // sequencer
  end
endtask: get_and_drive
```

The `req` and `rsp` variables are referring to the request to a sequencer and the response from the driver. The drive task is responsible for a parallel to serial conversion of our data. This is done in a simple for loop that waits for `csn` to drop and then proceeds with the driving of the signal as shown in Appendix B. While sending the data the CRC value is calculated by the 24 remaining bits in exactly the same way that was mentioned for the CRC-24 mode.

The monitor is responsible for detecting traffic on the interface and creating a sequence item with the data that is being sent to or from the DUT, therefore in the case of the SPI

interface it implements a serial to parallel conversion. In the run phase we use the `collect_transactions` task which is composed of a forever block, in which we create the sequence item with the `collect` task and then send it through a port to the scoreboard.

```
task spi_monitor::collect_transactions();
  int count = 0;

  forever begin
    item = new("item"); // one per transfer
    collect(count);
    count++;
    `uvm_info("DEBUG", {"\n", item.sprint()}, UVM_MEDIUM)
    aport.write(item);

    // if (checks_enable)
    //   perform_checks();

    // if (coverage_enable)
    //  perform_coverage();

  end
endtask: collect_transactions
```

We can also check the validity of data through the `perform_checks` and `perform_coverage` tasks but in our case this is done through the scoreboard.

In the `collect` task, shown in Appendix C, we use a for loop to capture not only the MOSI data but the MISO data as well. We also use count due to a timing difference between the first and subsequent transactions.

In the `spi_agent` object the connection between the three components is facilitated in the `connect_phase` function:

```
function void spi_agent::connect_phase(uvm_phase phase);
  super.connect_phase(phase);
```

```
  // connect monitor analyis port
  monitor.aport.connect(aport);

  // connect sequencer and driver (if existing) sequence item ports
  if(cfg.is_active) begin
    driver.seq_item_port.connect(sequencer.seq_item_export);
    driver.rsp_port.connect(sequencer.rsp_export);
  end

endfunction: connect_phase
```

The final result is schematically similar to the agent shown in Figure 2.1.

3.3 The HNDSHK (APB) Interface

The DUT is connected to an APB bus arbiter through the HNDSHK interface that we specifically named `apb_if` and it consists of the following ports:

```
  logic valid;
  logic write;
  logic [15:0] addr;
  logic [15:0] wdata;
  logic ack;
  logic nack;
  logic err;
  logic [15:0] rdata;
```

When the DUT, in our case, asserts the valid signal, if `write` is 0 then we read data in address of value `addr`. The testbench then returns the read data through `rdata` and asserts the `ack`. We can assert `nack` if we were unable to process the request or `err` for any possible issues that have arisen during processing. When `write` has a value of 1 we must write `wdata` in `addr`. APB supports a read after write functionality in which after the write operation is acknowledged the DUT will issue a read request for the same address.

3.3.1 The HNDSHK (APB) Agent

The sequence item for the hndshk interface is comprised of the following signals:

```
rand logic [15:0] rdata;
logic [15:0] wdata;
logic [15:0] addr;
logic ack;
rand logic nack;
rand logic err;
logic valid;
logic write;
```

The only signals that we want to be able to randomize (or constraint) are `rdata`, `nack` and `err`. We implement a soft constraint on `nack` and `err` in order to keep them at 0 unless instructed otherwise by a sequence's constraint.

```
constraint default_error_c {soft err == 0;}
constraint default_nack_c {soft nack == 0;}
```

We have created 4 sequences in total. Two that send multiple apb sequence items for read and write respectively and two that check the functionality in case of failure by asserting `err` or `nack`. An example for the former and the latter are shown below:

```
repeat(1000) begin
    `uvm_do_with(s_item, {s_item.err == 0; s_item.nack == 0;})
end

repeat(1000) begin
    `uvm_do_with(s_item, {s_item.err == 0; s_item.nack == 1;})
end
```

When driving our signals we must check if our sequence item is an error or not acknowledged item. If it is neither then we assert `ack` and in case of read pass the data for the address that was requested. In case of a successful write we pass `wdata` to `logic [15:0] data` in order to respond correctly to the upcoming read request. The run phase of our driver consists of two tasks. The first implements the aforementioned behavior and the second task

called `drive_ack` is responsible for driving any acknowledgment related signal to 0. The drive task is showcased at Appendix D. The simpler `drive_ack` task makes use of a forever block:

```
task apb_driver::drive_ack(uvm_phase phase);
  // When valid drops we must also drop ack, err and nack
  forever begin
    @(negedge vif.valid)
    vif.ack = 0;
    vif.nack = 0;
    vif.err = 0;
  end
endtask
```

When the valid signal is driven to 0 by the DUT we must also drive these three signals to 0 as well. Both of these tasks are executed simultaneously with the use of `fork` and then `join`. Our monitor is relatively simple for this interface. When `valid` is asserted we capture all interface ports' values on a sequence item and then send it through a port to the scoreboard.

```
task apb_monitor::collect(); // collect a single transaction
  // trace signals and translate to transactions
  @ (posedge vif.valid)

  // When valid is asserted we capture data in
  // the respective sequence items. The ack signal
  // is captured with a slight delay.
  item.addr = vif.addr;
  item.valid = vif.valid;
  item.write = vif.write;
  #5ns item.ack = vif.ack;
  item.nack = vif.nack;
  item.err = vif.err;

  if (vif.write == 1) begin
```

```
    item.wdata = vif.wdata;
  end
  else begin
    item.rdata = vif.rdata;
  end
endtask: collect
```

The agent connects the driver, sequencer and monitor in the same way that was shown in the SPI section above.

3.4 The Sideband Interface

There are 7 non-interface signals that we have grouped in a sideband interface shown below:

```
logic reset_n_i;
logic scan_mode_i;
logic [1:0] crcmd_i;
logic [5:0] sys_stat_i;
logic [31:0] tm_entry_key_init_i;
logic [31:0] tm_entry_key_finish_i;
logic [31:0] default_reply_i;
```

We did not include the system clock since it is on a separate agent in contrast to the active low `reset_n_i` that is included and softly held at 1 unless changed by a sequence. We control the CRC mode of the operation through `crcmd_i`. Table 3.2 shows the correlation between its value and the mode we operate on.

Field value	CRC Mode
$2'b00$	CRC-24
$2'b01$	CRC-40
$2'b10$	NO-CRC
$2'b11$	IGN-CRC

Table 3.2: Correlation of crcmd_i values and CRC mode

The status field mentioned in the SPI section in Figure 3.2 takes the value of the `sys_stat_i` signal. While test mode is enabled `tm_entry_key_init_i` and `tm_entry_key_finish_i` are used to define the first and last SPI frames in a way that causes a CRC error. In case of a reset `default_reply_i` defines the first MISO frame completely and the data field for subsequent frames.

3.4.1 The Sideband Agent

Our sequence item defines all these signals as `rand` and defaults `reset_n_i` to 1 unless changed explicitly. We have created 4 sequences for each possible CRC mode as well as a sequence for asserting the reset signal. Our driver uses one task to drive reset to 1 in order to begin the transaction and then a drive task to pass all sequence item fields to the interface signals.

```
task sideband_driver::drive(REQ req, RSP rsp);
  `uvm_info("DEBUG", {"\n", req.sprint()}, UVM_FULL)

  //- runs transaction cycle on interface
  // TODO: replace the following by protocol specific code
  `uvm_info("ITEM", {"\n", req.sprint()}, UVM_NONE)
  // set signals according to 'req'
  vif.reset_n_i = 0;
  vif.crcmd_i = req.crcmd_i;
  vif.sys_stat_i = req.sys_stat_i;
  vif.tm_entry_key_init_i = req.tm_entry_key_init_i;
  vif.tm_entry_key_finish_i = req.tm_entry_key_finish_i;
  vif.default_reply_i = req.default_reply_i;

  if (req.reset_n_i == 0) begin
    #7000ns vif.reset_n_i = 0;
  end

  // capture any output (including input) in 'rsp'
  #100ns; // avoid infinite loop for template
```

```
endtask: drive
```

For the sideband interface int the `collect` task, all interface signals are used to create a sequence item that is then passed on to the scoreboard.

```
task sideband_monitor::collect(); // collect a single transaction
  // trace signals and translate to transactions
  item.reset_n_i = vif.reset_n_i;
  item.crcmd_i = vif.crcmd_i;
  item.sys_stat_i = vif.sys_stat_i;
  item.tm_entry_key_init_i = vif.tm_entry_key_init_i;
  item.tm_entry_key_finish_i = vif.tm_entry_key_finish_i;
  item.default_reply_i = vif.default_reply_i;
  #100ns; // avoid 0 loop
endtask: collect
```

Like always the driver, monitor and sequencer are connected in the agent.

3.5 The Clock Generator

We use a separate agent for generating the system clock. We drive the signal `clk_i` with a frequency of 10 MHz in the `clk_gen_driver` and capture it in the monitor. Using a separate clock generator is common practice because it enables the potential usage of the same agent in another design.

3.6 The SPI _SYS _CTRL Interface

The spi_sys_ctrl interface is comprised of just one signal named `en_com_clk`. This signal is asserted by the DUT in order to enable the system clock frequence needed for bus communication. This happens when a falling edge of the chip select signal is detected.

3.6.1 The SPI _SYS _CTRL Agent

The `en_com_clk` signal as mentioned is driven by the DUT therefore the driver and sequencer are not needed in this case. The monitor receives the interface signal and created a sequence item that is then sent to the scoreboard.

3.7 The TM _EN _CTRL Interface

The tm_en_ctrl interface contains the following ports:

```
logic tm_key_1_received;
logic tm_key_2_received;
logic tm_non_key_message_received;
logic [1:0] spi_crc_mode;
logic [15:0] tm_max_read_addr;
logic [15:0] tm_max_write_addr;
```

The first three ports indicate to the test mode control module if a key SPI frame was received or not. The `spi_crc_mode` has the value of the mode that we are currently operating on as shown in Table 3.2. The last two ports operate as inputs to the SPI module and indicate the max values for read address and write address respectively.

3.7.1 The TM _EN _CTRL Agent

The sequence item contains every signal from the interface with only the last two being randomized and constrained to a default value:

```
constraint default_max_read_addr {soft tm_max_read_addr
                                   == 16'h01ff;}
constraint default_max_write_addr {soft tm_max_write_addr
                                   == 16'h00fc;}
```

The only ports that need to be driven are those related to the max address values. In the monitor we gather all signal values and then send the created sequence item to the scoreboard. Like always the connection between driver, monitor and sequencer happens in the agent.

3.8 The Scoreboard

After creating all smaller environments that pertain to each interface or signal, we need to create an environment that encapsulates our scoreboard. As mentioned before the goal of the scoreboard is to compare the actual output to the expected output and report on the number of

transactions and whether they are correct or not. The transactions are received through analysis ports that are connected to the driver who receives items from the monitor. The analysis ports we use in the scoreboard have a unique function named `write_{port_name}` that is called when a sequence item is sent through its dedicated analysis port. The implementation of these functions can be found at Appendix E. It is then up to us to handle the item accordingly, depending on our needs. There are two common ways to perform checks on a scoreboard, named in-order and out-of-order checking. With in-order checking we compare the transactions we receive in the order that we receive them by usually implementing a FIFO queue. The expected result is also calculated using current data. In some cases however we need to either compare or use a previous transaction to verify the correctness of a current transaction. This leads to the usage of arrays to write and then read data when it is needed. Both of these methods will be used in this project and are shown[11] in Figure 3.5 and Figure 3.6 respectively. The needed comparisons are executed in the `run_phase` task in several

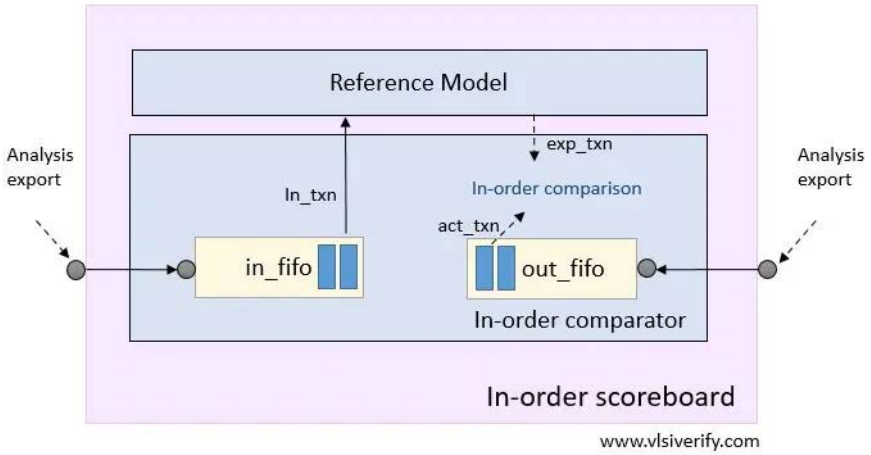

Figure 3.5: In-Order Scoreboard

tasks. The first important check we implement, is comparing the expected APB output with the actual output in the hndshk interface. The task named `calc_apb` uses the received spi transaction from a FIFO and computes the expected `apb_seq_item`. We then compare the two items on the `compare_apb` task. We also compare `sys_stat_i` with the status field from the MISO frame and the condition that needs to be fulfilled is:

```
spi_in_item.status[5:0] == sideband_item.sys_stat_i
```

We also check the bit in position 22 that indicates an error response. If our request was acknowledged successfully then its value should be 1 otherwise it must be 0. There are two out

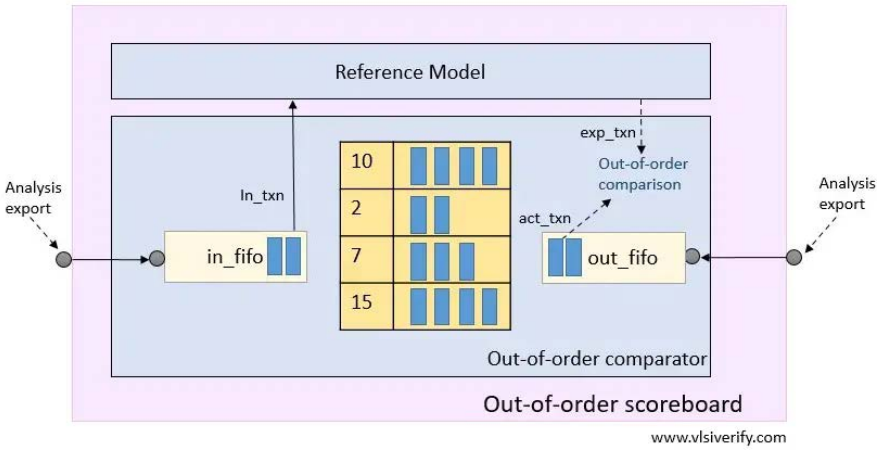

Figure 3.6: Out-Of-Order Scoreboard

of order checks that are imperative to our simulation and are related to the CRC field and the MISO response to our MOSI frame. After the first transaction is complete, we push incoming SPI and APB packets on the back of two arrays. We then pop items from the front when needed. For the CRC check, on the `calc_crc` task (Appendix F), depending on the CRC mode that we have set through `crcmd_i`, we calculate the expected `miso_crc` field. For the second out-of-order check we compare the previous APB item's `rdata` field with the current MISO item's `read_data` field, expecting them to be equal. Finally, we calculate the expected `spi_crc_mode` signal from the `tm_en_ctrl` interface based on the related sideband signal and the MOSI CRC field. The comparison with the actual value happens on the `compare_tm_en_ctrl` task. We also use three integers in order to know the number of transactions we have, how many passed and how many of them failed (`transactions`, `pass_count` and `error_count`). The entire `run_phase` task can be read at Appendix G. A very significant part of a scoreboard is the creation of covergroups that pertain to the given requirements. We need to cover the address of the HNDSHK interface as well as the `write_addr` field of the SPI item. We must also cover the `crcmd_i` and cross cover it with the CRC field of MOSI frames. Finally we cover the signals related to acknowledgment in the APB interface and their possible combinations. All created covergroups are shown in Appendix H and are sampled in the write function in Appendix E. The metrics from those covergroups are printed in the report phase, with the number of transactions and a PASS or FAIL message (shown in Appendix I).

3.9 The Testbench

The testbench consists of three very important components. The virtual sequencer who contains handles for all the sequencers of the aforementioned agents, the testbench environment (shown in Figure 3.7), that creates the scoreboard, the virtual sequencer and all the

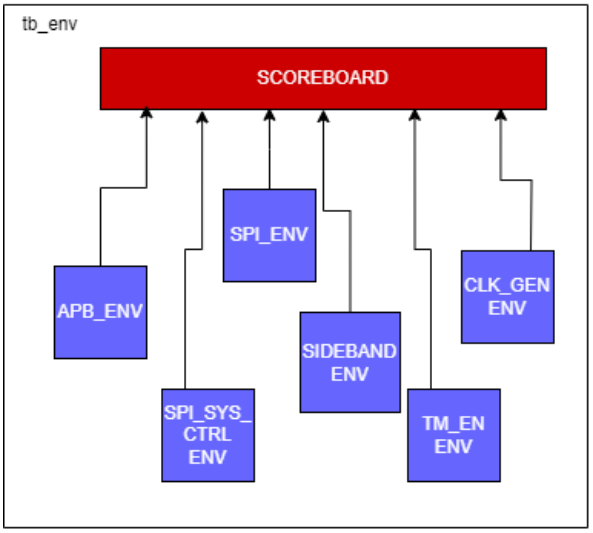

Figure 3.7: SPI Testbench Environment

smaller environments and connects them with each other, and the `spi_tb_top` module that instantiates and connects the DUT with all environment interfaces:

```
d_spi_v2 dut
  (
    .spi(spi_wire),
    .hndshk_m(apb_wire),
    .tm_en_ctrl(tm_en_ctrl_wire),
    .spi_sys_ctrl(spi_sys_ctrl_wire),
    .clk_i(clk_wire.clk_i),
    .reset_n_i(sideband_wire.reset_n_i),
    .scan_mode_i(sideband_wire.scan_mode_i),
    .crcmd_i(sideband_wire.crcmd_i),
    .sys_stat_i(sideband_wire.sys_stat_i),
    .tm_entry_key_init_i(sideband_wire.tm_entry_key_init_i),
    .tm_entry_key_finish_i(sideband_wire.tm_entry_key_finish_i),
    .default_reply_i(sideband_wire.default_reply_i));
```

In the build phase of our testbench environment we can also make configurations for any of our agents through the configuration object that is created for every environment. In our instance we want the `spi_sys_ctrl` agent to be passive and set it with:

```
cfg.spi_sys_ctrl_inst_cfg.is_active = UVM_PASSIVE;
```

Chapter 4

Simulation Results

Before we showcase some simulation results and the metrics that arise from them, we should talk about the tests that we have created and what functionality they are meant to cover. In Table 4.1 all tests that we are able to execute are shown.

Test Name	Purpose
test_spi_mult_reads	Send 1000 valid read transactions with CRC-24 mode
test_spi_mult_writes	Send 1000 valid write transactions with CRC-24 mode
test_spi_mult_reads_*	Send 1000 valid read transactions with any CRC mode on *
test_spi_mult_writes_*	Send 1000 valid write transactions with any CRC mode on *
test_apb_err	Send error responses through the HNDSHK interface
test_apb_nack	Send nack responses through the HNDSHK interface

Table 4.1: Tests to be ran

For running a simulation we use the XCELIUM tool by Cadence Design Systems, Inc. We can launch a test using the `xrun` command by using the `+UVM_TESTNAME` argument and, if we want, enable coverage by writing `-coverage all`. XCELIUM then opens a SimVision window that is used for debugging our simulation. Aside from the expected Waveform and Console viewer SimVision offers the ability to view our UVM environment hierarchy (shown in Figure 4.1) and thus verifying its correctness by checking all the created components. We can control some UVM functionalities (e.g. UVM Verbosity) and view related information through some useful UVM buttons (shown in Figure 4.2). We can also proceed with the simulation phase by phase, through the usage of the third button shown in the previously mentioned figure. After clicking the play button the simulation will begin and all messages

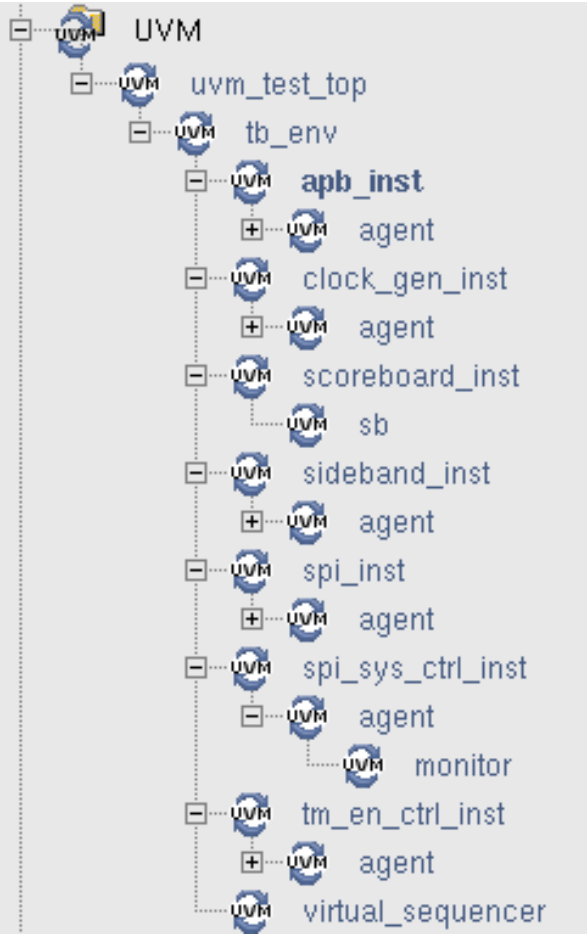

Figure 4.1: UVM Hierarchy Viewer

Figure 4.2: UVM Related Buttons

that match the given verbosity will be printed in the console window. For a successful run the message that we want to be printed out is shown in Figure 4.3. Normally the driver also prints out every sequence item sent, but a more convenient way to see sequences and all sent sequence items lies in the UVM Sequence Viewer window (Figure 4.4.

Through the Waveform window (Figure 4.5) we are able to see how the hndshk interface sends a request based on the MOSI frame we have sent.

Despite our ability to print the coverage score in the console window this does not prove to be enough for our purposes. Imagine that we have a specific address field in a sequence item that we decide to cover, like the APB address here. Along with how many values we have actually covered, we might also need to see which specific values have been invoked by our test. The results of coverage, that we saw, were also related to only one test that can't

```
*** TEST PASSED - 1000 transactions ran, 1000 transactions passed ***

APB Address Coverage = 84.96 %
APB Acknowledgment Coverage = 45.83 %
SPI Write Address Coverage = 0.79 %
SPI CRC Mode Coverage = 50.00 %
```

Figure 4.3: Successful Simulation Message

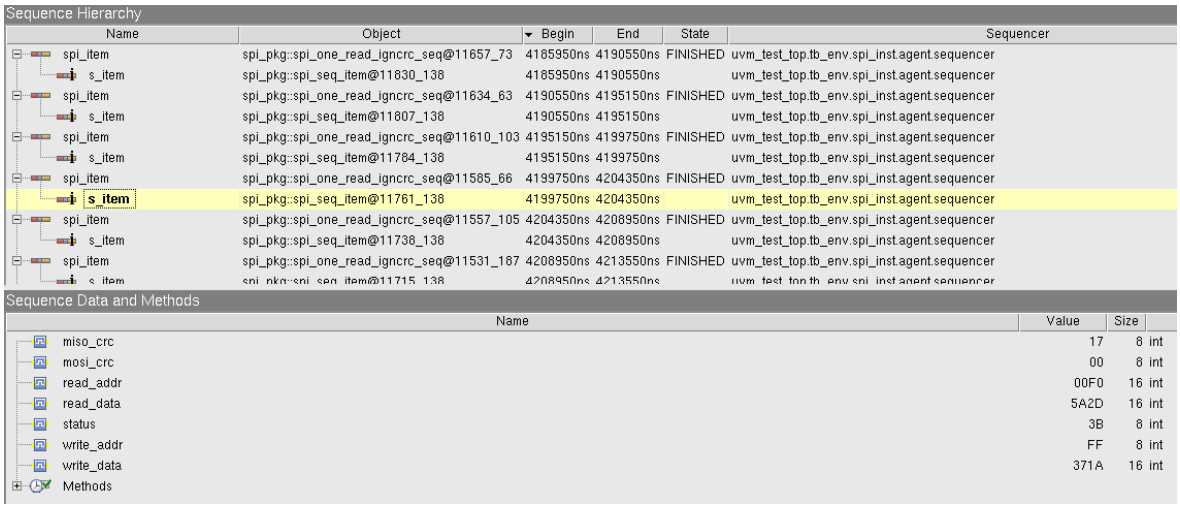

Figure 4.4: Sequence Viewer Window

cover all possible functions of a DUT. Therefore, we need a tool that is able to show detailed coverage metrics of user defined coverpoints, show code coverage metrics and show all these results accumulated from multiple tests. VManager by Cadence Design Systems, Inc. fulfills all these demands by being able to launch a Regression through a .vsif (verification session input format) file. In this file we can name the tests we desire to run and how many times we want them to be executed, as shown below:

```
test test_spi_mult_reads {
    count       :  3;
};
test test_spi_mult_reads_crc40 {
    count       :  3;
};
test test_spi_mult_reads_nocrc {
    count       :  3;
};
test test_spi_mult_reads_igncrc {
```

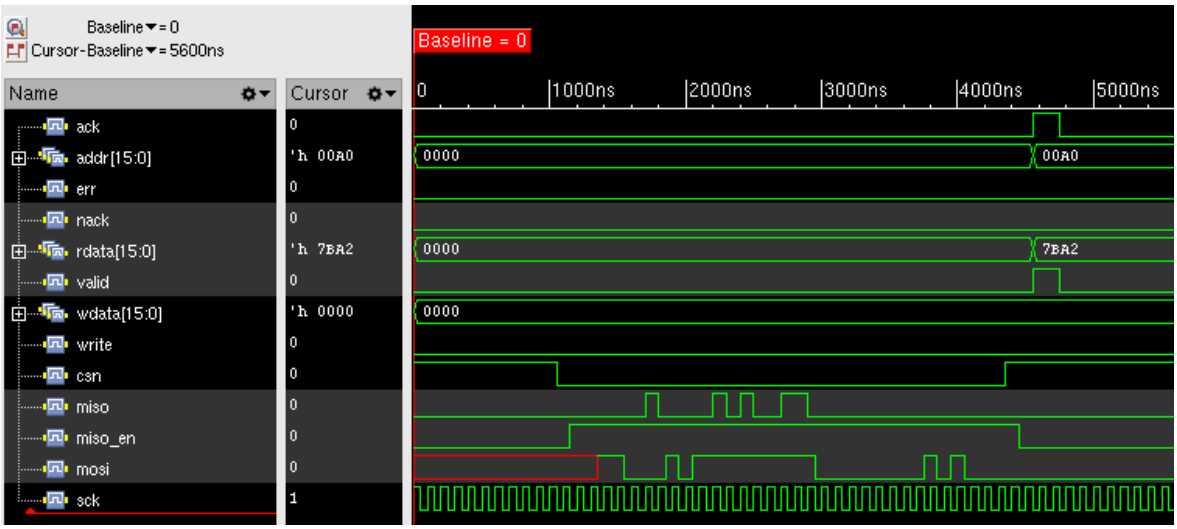

Figure 4.5: Waveform Window

```
    count       :  3;
};
test test_spi_mult_writes {
    count       :  3;
};
test test_apb_nack {
    count       :  1;
};
test test_apb_err {
    count       :  1;
};
```

After running a regression we can view the tests that were ran and whether they were successful or not (Figure 4.6). Then by clicking the Metrics button we can view the coverage

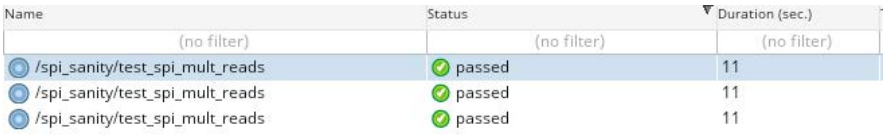

Figure 4.6: VManager Regression Tests

hierarchy, where the percentages for each component, interface and DUT can be viewed. By clicling the DUT we can view the code coverage metrics in the bottom right of the window. Coverage is calculated separately for different parts of the code as shown in Figure 4.7. By

Name	Overall Average Grade	Overall Covered
Overall	62.62%	2772 / 4828 (57.42%)
Code	65.53%	2249 / 4038 (55.7%)
Block	91.07%	514 / 674 (76.26%)
Statement	93.64%	501 / 631 (79.4%)
Expression	73.57%	421 / 648 (64.97%)
Toggle	62.88%	1314 / 2716 (48.38%)

Figure 4.7: Code Coverage Window

navigating to the scoreboard component in the hierarchy we are able to see separate metrics for each covergroup, coverpoint and bin. As an example, in Figure 4.8 we can view the metrics of three bins and their cross product for covergroup `apb_addr_space`. Our goal is to

Name	Overall Average Grade	Overall Covered
(no filter)	(no filter)	(no filter)
apb_ack	50%	1 / 2 (50%)
apb_nack	50%	1 / 2 (50%)
apb_err	50%	1 / 2 (50%)
AxB request_states	33.33%	1 / 3 (33.33%)

Figure 4.8: Covergroup Metrics Example

achieve 100% coverage on the user defined covergroups that showcase the requirements that we have managed to verify. By adding different tests we see a plotting of the coverage metrics in Figure 4.9. We notice a sudden increase in coverage after the (18, 75.2) point compared to

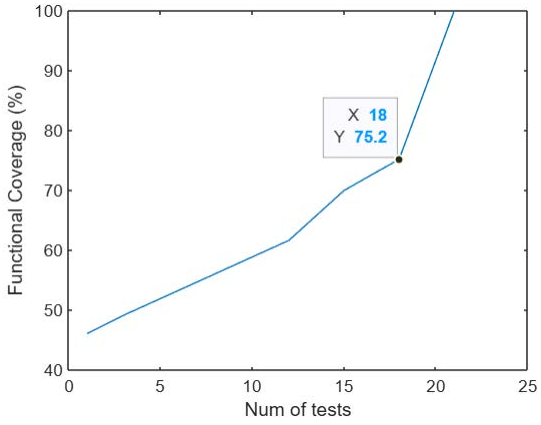

Figure 4.9: Scoreboard Coverage Plot

the previous trend of our line. This is due to the number of bins covered by the test we added on our vsif file. The number of values that a test covers is more important than the number of tests we choose to run for a given regression. Code coverage is more difficult to raise, especially after a certain point, as shown in Figure 4.10 where we only manage to cover 70% with 21 tests. In order to completely cover the DUT code we need fine grained coverage with

a variety of tests that cover incredibly specific cases. The requirements that we wanted and

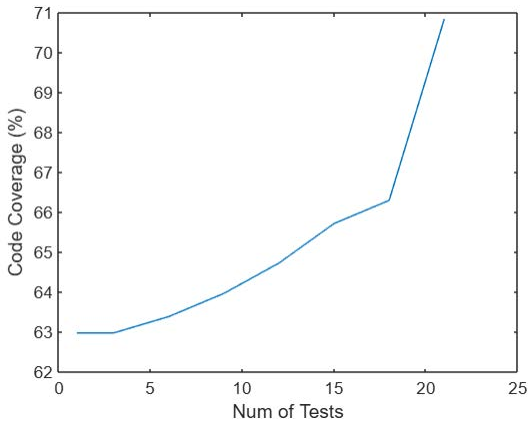

Figure 4.10: DUT Code Coverage Plot

managed to cover with our tests are:

- Successful transmission of 32-bit read frame through SPI
- Successful transmission of 32-bit write frame through SPI
- Successful operation under CRC-24 mode
- Successful operation under CRC-40 mode
- Successful operation under NO-CRC mode
- Successful operation under IGN-CRC mode
- The data field in the MISO frame should be the same as the read data field in APB
- The response status field must be equal to the status sideband signal
- In case of an error the response frame should have a specific form
- The response CRC field must be in adherence with the given CRC mode
- Through the HNDSHK interface we can acknowledge a request by only asserting the `ack` signal
- Through the HNDSHK interface we can signal a failure to acknowledge a request by only asserting `nack`

- Thorugh the HNDSHK interface we can indicate an error during processing by asserting the `ack` and `err` signals
- In SPI `read_addr` has a range of $[0'h0000 : 0'h001F]$ and `write_addr` has a range of $[0'h00 : 0'hFC]$
- The CRC signal that is returned by the `tm_en_ctrl` interface should be in tandem with the MOSI CRC field and the `crcmd_i` sideband signal
- The `en_com_clk` signal should be asserted when a `csn` falling edge is detected

In conclusion, we have managed to test and verify a big and significant part of the SPI module by fully covering the requirements regarding the HNDSHK, SPI_SYS_CTRL and SPI interfaces. As it was our goal, UVM has allowed us to create a changeable and easily configurable environment comprised of completely reusable parts and to view and present the accumulated results of multiple tests in an organized and understandable way.

Bibliography

[1] Accelera. *SystemVerilog 3.1: Accelera's Extensions to Verilog*, April 2003. Available at https://www.eda-twiki.org/sv-ec/SystemVerilog_3.1_final.pdf.

[2] https://web.archive.org/web/20110927101523/http://www.accellera.org/activities/vip/VIP-TC_standard_effort_update_Jan_2010.pdf. VIP-TSC Standardization Update.

[3] https://www.chipverify.com/systemverilog/systemverilog-interface. SystemVerilog Interfaces.

[4] https://vlsiverify.com/system-verilog/randomization-in-systemverilog/. SystemVerilog Randomization.

[5] https://www.chipverify.com/systemverilog/systemverilog-functional-coverage. SystemVerilog Functional Coverage.

[6] https://www.chipverify.com/uvm/uvm-testbench-example-1. Testbench Top Example.

[7] https://www.chipverify.com/uvm/uvm-phases. UVM Phases.

[8] https://www.chipverify.com/uvm/uvm-utility-field-macros#field-macros. UVM Macros.

[9] https://www.chipverify.com/uvm/uvm-tlm-analysis-port. UVM TLM Analysis Port.

[10] Motorola Inc. *SPI Block Guide V03.06*, February 2003. Available at https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf.

[11] https://vlsiverify.com/uvm/uvm-scoreboard/. UVM Scoreboard.

APPENDICES

Appendix A

CRC Calculation Functions

```
function logic[7:0] crc24_calc(input logic[7:0] crcIn,
                               input logic[23:0] data);
  begin
    crc24_calc[0] = crcIn[0] ^ crcIn[1] ^ data[0] ^ data[4] ^
                    data[5] ^ data[6] ^ data[10] ^ data[13] ^
                    data[15] ^ data[16] ^ data[17];
    crc24_calc[1] = crcIn[0] ^ crcIn[1] ^ crcIn[2] ^ data[1] ^
                    data[5] ^ data[6] ^ data[7] ^ data[11] ^
                    data[14] ^ data[16] ^ data[17] ^ data[18];
    crc24_calc[2] = crcIn[0] ^ crcIn[2] ^ crcIn[3] ^ data[0] ^
                    data[2] ^ data[4] ^ data[5] ^ data[7] ^
                    data[8] ^ data[10] ^ data[12] ^ data[13] ^
                    data[16] ^ data[18] ^ data[19];
    crc24_calc[3] = crcIn[0] ^ crcIn[3] ^ crcIn[4] ^ data[0] ^
                    data[1] ^ data[3] ^ data[4] ^ data[8] ^
                    data[9] ^ data[10] ^ data[11] ^ data[14] ^
                    data[15] ^ data[16] ^ data[19] ^ data[20];
    crc24_calc[4] = crcIn[4] ^ crcIn[5] ^ data[0] ^ data[1] ^
                    data[2] ^ data[6] ^ data[9] ^ data[11] ^
                    data[12] ^ data[13] ^ data[20] ^ data[21];
    crc24_calc[5] = crcIn[5] ^ crcIn[6] ^ data[1] ^ data[2] ^
                    data[3] ^ data[7] ^ data[10] ^ data[12] ^
```

```
                    data[13] ^ data[14] ^ data[21] ^ data[22];
    crc24_calc[6] = crcIn[6] ^ crcIn[7] ^ data[2] ^ data[3] ^
                    data[4] ^ data[8] ^ data[11] ^ data[13] ^
                    data[14] ^ data[15] ^ data[22] ^ data[23];
    crc24_calc[7] = crcIn[0] ^ crcIn[7] ^ data[3] ^ data[4] ^
                    data[5] ^ data[9] ^ data[12] ^ data[14] ^
                    data[15] ^ data[16] ^ data[23];
  end
endfunction

function logic[7:0] crc40_calc(input [7:0] crcIn,
                               input [39:0] data);
  begin
    crc40_calc[0] = crcIn[2] ^ crcIn[3] ^ crcIn[5] ^ crcIn[6] ^
                    crcIn[7] ^ data[0] ^ data[4] ^ data[5] ^
                    data[6] ^ data[10] ^ data[13] ^ data[15] ^
                    data[16] ^ data[17] ^ data[24] ^ data[25] ^
                    data[28] ^ data[31] ^ data[34] ^ data[35] ^
                    data[37] ^ data[38] ^ data[39];
    crc40_calc[1] = crcIn[0] ^ crcIn[3] ^ crcIn[4] ^ crcIn[6] ^
                    crcIn[7] ^ data[1] ^ data[5] ^ data[6] ^
                    data[7] ^ data[11] ^ data[14] ^ data[16] ^
                    data[17] ^ data[18] ^ data[25] ^ data[26] ^
                    data[29] ^ data[32] ^ data[35] ^ data[36] ^
                    data[38] ^ data[39];
    crc40_calc[2] = crcIn[1] ^ crcIn[2] ^ crcIn[3] ^ crcIn[4] ^
                    crcIn[6] ^ data[0] ^ data[2] ^ data[4] ^
                    data[5] ^ data[7] ^ data[8] ^ data[10] ^
                    data[12] ^ data[13] ^ data[16] ^ data[18] ^
                    data[19] ^ data[24] ^ data[25] ^ data[26] ^
                    data[27] ^ data[28] ^ data[30] ^ data[31] ^
                    data[33] ^ data[34] ^ data[35] ^ data[36] ^
```

```
                    data[38];
    crc40_calc[3] = crcIn[0] ^ crcIn[4] ^ crcIn[6] ^ data[0] ^
                    data[1] ^ data[3] ^ data[4] ^ data[8] ^
                    data[9] ^ data[10] ^ data[11] ^ data[14] ^
                    data[15] ^ data[16] ^ data[19] ^ data[20] ^
                    data[24] ^ data[26] ^ data[27] ^ data[29] ^
                    data[32] ^ data[36] ^ data[38];
    crc40_calc[4] = crcIn[1] ^ crcIn[2] ^ crcIn[3] ^ crcIn[6] ^
                    data[0] ^ data[1] ^ data[2] ^ data[6] ^
                    data[9] ^ data[11] ^ data[12] ^ data[13] ^
                    data[20] ^ data[21] ^ data[24] ^ data[27] ^
                    data[30] ^ data[31] ^ data[33] ^ data[34] ^
                    data[35] ^ data[38];
    crc40_calc[5] = crcIn[0] ^ crcIn[2] ^ crcIn[3] ^ crcIn[4] ^
                    crcIn[7] ^ data[1] ^ data[2] ^ data[3] ^
                    data[7] ^ data[10] ^ data[12] ^ data[13] ^
                    data[14] ^ data[21] ^ data[22] ^ data[25] ^
                    data[28] ^ data[31] ^ data[32] ^ data[34] ^
                    data[35] ^ data[36] ^ data[39];
    crc40_calc[6] = crcIn[0] ^ crcIn[1] ^ crcIn[3] ^ crcIn[4] ^
                    crcIn[5] ^ data[2] ^ data[3] ^ data[4] ^
                    data[8] ^ data[11] ^ data[13] ^ data[14] ^
                    data[15] ^ data[22] ^ data[23] ^ data[26] ^
                    data[29] ^ data[32] ^ data[33] ^ data[35] ^
                    data[36] ^ data[37];
    crc40_calc[7] = crcIn[1] ^ crcIn[2] ^ crcIn[4] ^ crcIn[5] ^
                    crcIn[6] ^ data[3] ^ data[4] ^ data[5] ^
                    data[9] ^ data[12] ^ data[14] ^ data[15] ^
                    data[16] ^ data[23] ^ data[24] ^ data[27] ^
                    data[30] ^ data[33] ^ data[34] ^ data[36] ^
                    data[37] ^ data[38];
  end
```

```
endfunction
```

Appendix B

SPI drive task

```
task spi_driver::drive(REQ req, RSP rsp);
  logic [31:0] frame;
  logic [7:0] crc;
  `uvm_info("DEBUG", {"\n", req.sprint()}, UVM_FULL)

  //- runs transaction cycle on interface
  `uvm_info("ITEM", {"\n", req.sprint()}, UVM_NONE)
  // set signals according to 'req'

  // Change the frame to be sent according to write address value
  // We either send a read or a write
  if (req.write_addr == 8'b11111111) begin
    frame = {req.mosi_crc, req.write_addr, req.read_addr};
  end
  else begin
    frame = {req.mosi_crc, req.write_addr, req.write_data};
  end

  // If we aren't in no_crc or ign_crc mode we calculate
  // the crc field to be sent according to the given requirements
  if ((req.mosi_crc != 8'hf1) && (req.mosi_crc != 8'h00)) begin
    crc = crc_calc(8'hFF, {frame[7:0], frame[15:8], frame[23:16]});
```

```
    frame[31:24] = crc ^ 8'hff;
    `uvm_info("DEBUG", {"\n", $sformatf(" %x", crc)}, UVM_NONE)
  end

  `uvm_info("DEBUG", {"\n", $sformatf(" %b", frame)}, UVM_FULL)

  // We send the frame one bit in every negative clock edge
  // We begin from 45 to wait for the csn signal to drop
  // The csn signal is raised one clock cycle earlier due to
  // internal synchronization
  for(int i = 45; i > -1; i--) begin
    if ((i >= 0) && (i < 32)) begin
      vif.mosi = frame[i];
      if (i == 1) begin
        vif.csn = 1;
      end
    end
    @(negedge vif.sck);
  end
endtask: drive
```

Appendix C

SPI collect task

```
task spi_monitor::collect(input int count);
  // collect a single transaction

  // trace signals and translate to transactions
  logic [31:0] frame_miso;
  logic [31:0] frame_mosi;

  // When we notice the csn dropping we begin capturing the bits
  // Count showcases the packet number. In the first packet we begin
  // sampling after 2 clock cycles. In subsequent packets we sample
  // after 1 cycle. Sampling happens on a negative clock edge.
  @(negedge vif.csn)
  for(int i = 35; i >= -1; i--) begin
    if (count == 0) begin
      if ((i >= 0) && (i < 32)) begin
        frame_mosi[i] = vif.mosi;
      end
      if ((i >= 1) && (i < 33)) begin
        frame_miso[i-1] = vif.miso;
      end
    end
    else begin
```

```
      if ((i >= -1) && (i < 31)) begin
        frame_mosi[i+1] = vif.mosi;
      end
      if ((i >= 0) && (i < 32)) begin
        frame_miso[i] = vif.miso;
      end
    end
    @(negedge vif.sck);
  end
  `uvm_info("DEBUG", {"\n", $sformatf(" %b", frame_mosi)},
            UVM_MEDIUM)
  `uvm_info("DEBUG", {"\n", $sformatf(" %b", frame_miso)},
            UVM_MEDIUM)

  // After capturing the mosi and miso frames we fill the sequence
  // items accordingly. We have two separate crc fields for checking
  // purposes.
  item.mosi_crc = frame_mosi[31:24];
  item.miso_crc = frame_miso[31:24];
  item.write_addr = frame_mosi[23:16];
  if (frame_mosi[23:16] == 8'b11111111) begin
    item.read_addr = frame_mosi[15:0];
  end
  else begin
    item.write_data = frame_mosi[15:0];
  end
  item.status = frame_miso[23:16];
  item.read_data = frame_miso[15:0];
  `uvm_info("DEBUG", {"\n", item.sprint()}, UVM_MEDIUM)

endtask: collect
```

Appendix D

APB drive task

```
task apb_driver::drive(REQ req, RSP rsp);
  `uvm_info("DEBUG", {"\n", req.sprint()}, UVM_FULL)

  //- runs transaction cycle on interface
  // TODO: replace the following by protocol specific code
  `uvm_info("ITEM", {"\n", req.sprint()}, UVM_NONE)
  // set signals according to 'req'
  @ (posedge vif.valid)

  // When the valid signal is asserted by the DUT we check
  // if we have a write or a read request. When we have a write
  // we save the data in a parameter and pass it in the next read
  // request (there is always a read after the write)
  if ((req.nack == 0) && (req.err == 0)) begin
    if (vif.write == 1) begin
      vif.ack = 1;
      data = vif.wdata;
    end
    else begin
      vif.ack = 1;
      if (data != 16'd0) begin
        vif.rdata = data;
```

```
      end
      else begin
        vif.rdata = req.rdata;
      end
    end
  end
  else if (req.nack == 1) begin
    vif.nack = 1;
  end
  else if (req.err == 1) begin
    vif.ack = 1;
    vif.err = 1;
  end
  // capture any output (including input) in 'rsp'
  // #100ns; // avoid infinite loop for template
endtask: drive

task apb_driver::drive_ack(uvm_phase phase);
  // When valid drops we must also drop ack, err and nack
  forever begin
    @(negedge vif.valid)
    vif.ack = 0;
    vif.nack = 0;
    vif.err = 0;
  end
endtask
```

Appendix E

Implementation of write functions

```
function void spi_scb_scoreboard::write_spi_port(
  spi_seq_item item   // received transaction
);
  //-START-USER-REGION: "IMPLEMENTATION-WRITE_SPI_PORT"
  `uvm_info("SPI_PORT", {"\n", item.sprint()}, UVM_FULL);
  void'(spi_infifo.try_put(item));
  spi_cov_item.copy(item);
  spi_q.push_back(item);
  spi_crc_cov.sample();
  spi_write_addr_cov.sample();

  //-END-USER-REGION: "IMPLEMENTATION-WRITE_SPI_PORT"
endfunction: write_spi_port


//-----------------------------------------------------------------
function void spi_scb_scoreboard::write_apb_port(
  apb_seq_item item   // received transaction
);
  //-START-USER-REGION: "IMPLEMENTATION-WRITE_APB_PORT"
  `uvm_info("APB_PORT", {"\n", item.sprint()}, UVM_FULL);
```

```
  apb_cov_item.copy(item);
  apb_ack_cov.sample();
  void'(apb_actfifo.try_put(item));

  // We are only interested in read commands for the spi2apb
  // comparison
  if (item.write == 0) begin
    apb_addr_cov.sample();
    apb_q.push_back(item);
  end

  //-END-USER-REGION: "IMPLEMENTATION-WRITE_APB_PORT"
endfunction: write_apb_port


//---------------------------------------------------------------
function void spi_scb_scoreboard::write_sideband_port(
  sideband_seq_item item   // received transaction
);
  //-START-USER-REGION: "IMPLEMENTATION-WRITE_SIDEBAND_PORT"
  `uvm_info("SIDEBAND_PORT", {"\n", item.sprint()}, UVM_FULL);
  void'(sideband_fifo.try_put(item));
  sideband_cov_item.copy(item);
  //-END-USER-REGION: "IMPLEMENTATION-WRITE_SIDEBAND_PORT"
endfunction: write_sideband_port


//---------------------------------------------------------------
function void spi_scb_scoreboard::write_tm_en_ctrl_port(
  tm_en_ctrl_seq_item item   // received transaction
);
  //-START-USER-REGION: "IMPLEMENTATION-WRITE_TM_EN_CTRL_PORT"
```

```
  `uvm_info("TM_EN_CTRL_PORT", {"\n", item.sprint()}, UVM_FULL);
  void'(tm_en_ctrl_fifo.try_put(item));
  //-END-USER-REGION: "IMPLEMENTATION-WRITE_TM_EN_CTRL_PORT"
endfunction: write_tm_en_ctrl_port
```

Appendix F

CRC Calculation Scoreboard Task

```
task calc_crc(input sideband_seq_item sideband_item,
input spi_seq_item spi_in_item, input apb_seq_item apb_act_item,
output logic[7:0] exp_crc_field);
  logic [23:0] in_data;
  logic [39:0] big_in_data;

  case (sideband_item.crcmd_i)
    // CRC-24 Mode (24-bit word)
    2'd0: begin
      in_data = {apb_act_item.rdata[7:0], apb_act_item.rdata[15:8],
      spi_in_item.status};
      exp_crc_field = crc24_calc(8'hFF, in_data);
      exp_crc_field = exp_crc_field ^ 8'hff;
    end
    // CRC-40 Mode (40-bit word)
    2'd1: begin
      if (spi_in_item.write_addr == 8'hff) begin
        big_in_data = {spi_in_item.read_addr[7:0],
        spi_in_item.read_addr[15:8], apb_act_item.rdata[7:0],
        apb_act_item.rdata[15:8], spi_in_item.status};
      end
      else begin
```

```
        big_in_data = {spi_in_item.write_addr, 8'h00,
        apb_act_item.rdata[7:0], apb_act_item.rdata[15:8],
        spi_in_item.status};
      end
      exp_crc_field = crc40_calc(8'hFF, big_in_data);
      exp_crc_field = exp_crc_field ^ 8'hff;
    end
    // NO-CRC Mode (Fixed Value)
    2'd2: begin
      exp_crc_field = 8'hf1;
    end
    // IGN-CRC Mode (24-bit word)
    2'd3: begin
      in_data = {apb_act_item.rdata[7:0], apb_act_item.rdata[15:8],
      spi_in_item.status};
      exp_crc_field = crc24_calc(8'hFF, in_data);
      exp_crc_field = exp_crc_field ^ 8'hff;
    end
  endcase
endtask
```

Appendix G

Scoreboard Run Phase

```
task spi_scb_scoreboard::run_phase(uvm_phase phase);
  logic [7:0] exp_crc;
  spi_seq_item spi_in_item;
  spi_seq_item prev_spi;
  apb_seq_item apb_act_item;
  apb_seq_item prev_apb;
  apb_seq_item apb_exp_item = new("apb_exp_item");
  sideband_seq_item sideband_item;
  tm_en_ctrl_seq_item tm_en_ctrl_act_item;
  tm_en_ctrl_seq_item tm_en_ctrl_exp_item =
                                    new("tm_en_ctrl_exp_item");

  forever begin
    spi_infifo.get(spi_in_item);
    apb_actfifo.get(apb_act_item);
    sideband_fifo.get(sideband_item);
    tm_en_ctrl_fifo.get(tm_en_ctrl_act_item);

    transactions++;

    // Compare the expected apb to the actual apb.
    if (!spi_infifo.size()) begin
```

```
    compare_status(spi_in_item, sideband_item);
    calc_apb(spi_in_item, apb_act_item, apb_exp_item);
    compare_apb(apb_act_item, apb_exp_item);
  end

  // After the first transaction we compare the spi miso_crc
  // to the expected crc and the spi miso read_data to the
  // apb_data.
  if ((transactions > 1)) begin
    if ((!apb_act_item.err) && (!apb_act_item.nack)) begin
      prev_apb = apb_q.pop_front();
      prev_spi = spi_q.pop_front();

      calc_crc(sideband_item, prev_spi, prev_apb, exp_crc);

      compare_crc_field(spi_in_item, exp_crc);
      compare_spi_2_apb(spi_in_item, prev_apb);
    end
    else begin
      if (spi_in_item.status[6] == 1) begin
        error_count++;
        `uvm_error("ERROR", $sformatf("Actual=1 Expected=0 \n"))
      end
      else begin
        `uvm_info("PASS", $sformatf("Actual=0 Expected=0 \n"),
                  UVM_HIGH)
      end
    end
  end

  // According to the crc mode that is set we calculate
  // the expected tm_en_ctrl crc value and compare it to
```

```
    // the actual value
    if (!tm_en_ctrl_fifo.size()) begin
      calc_tm_en_ctrl(spi_in_item, sideband_item, tm_en_ctrl_exp_item);
      compare_tm_en_ctrl(tm_en_ctrl_act_item, tm_en_ctrl_exp_item);
    end
    pass_count++;
    `uvm_info("DEBUG", {"DONE COMPARING! PASSED!"}, UVM_HIGH);
  end
endtask
```

Appendix H

Scoreboard Covergroups

```
covergroup apb_addr_cov;
  apb_addr_space: coverpoint apb_cov_item.addr {
    bins valid_read [] = {[16'h0000:16'h01ff]};
  }
endgroup

covergroup apb_ack_cov;
  apb_ack: coverpoint apb_cov_item.ack {
    bins ack_req = {1'b1};
    bins ack_zero = {1'b0};
  }
  apb_nack: coverpoint apb_cov_item.nack {
    bins nack_req = {1'b1};
    bins nack_zero = {1'b0};
  }
  apb_err: coverpoint apb_cov_item.err {
    bins err_req = {1'b1};
    bins err_zero = {1'b0};
  }
  // We ignore all unwanted combinations
  request_states: cross apb_ack, apb_nack, apb_err {
    ignore_bins ack = binsof(apb_ack.ack_req) &&
```

```
                        binsof(apb_nack.nack_req);
    ignore_bins nack = binsof(apb_nack.nack_req) &&
                        binsof(apb_err.err_req);
    ignore_bins general = binsof(apb_ack.ack_zero) &&
                          binsof(apb_nack.nack_zero);
  }
endgroup

covergroup spi_write_addr_cov;
  write_addr_space: coverpoint spi_cov_item.write_addr {
    bins read = {8'hff};
    bins write [] = {[8'h00:8'hfe]};
  }
endgroup

covergroup spi_crc_cov;
  spi_crc_mode: coverpoint spi_cov_item.mosi_crc {
    bins ign_crc = {8'h00};
    bins no_crc = {8'hf1};
    bins other = {[8'h00:8'hf0], [8'hf2:8'hff]};
  }
  sideband_crc_mode: coverpoint sideband_cov_item.crcmd_i {
    bins crc24 = {2'b00};
    bins crc40 = {2'b01};
    bins no_crc = {2'b10};
    bins ign_crc = {2'b11};
  }
  // We ignore all unwanted combinations
  crc_of_operation: cross spi_crc_mode, sideband_crc_mode {
    ignore_bins crc24_mode = binsof(sideband_crc_mode.crc24) &&
                             !binsof(spi_crc_mode.other);
    ignore_bins crc40_mode = binsof(sideband_crc_mode.crc40) &&
```

```
                                !binsof(spi_crc_mode.other);
    ignore_bins no_crc_mode = binsof(sideband_crc_mode.no_crc) &&
                                 !binsof(spi_crc_mode.no_crc);
    ignore_bins ign_crc_mode = binsof(sideband_crc_mode.ign_crc) &&
                                  !binsof(spi_crc_mode.ign_crc);
  }
endgroup

covergroup spi_sys_ctrl_cov;
  en_com_clk: coverpoint spi_sys_ctrl_cov_item.en_com_clk {
    bins active = {1};
    bins passive = default;
  }
endgroup
```

Appendix I

Scoreboard Report Phase

```
function void spi_scb_scoreboard::report_phase(uvm_phase phase);
  super.report_phase(phase);

  if (transactions && !error_count) begin
    `uvm_info("PASSED",
    $sformatf("\n\n\n*** TEST PASSED - %0d transactions ran,
              %0d transactions passed ***\n",
              transactions, pass_count), UVM_LOW)
  end
  else begin
    `uvm_error("FAILED",
    $sformatf("\n\n\n*** TEST FAILED - %0d transactions ran,
              %0d transactions passed,
              %0d transactions failed ***\n",
              transactions, pass_count, error_count))
  end

  $display ("APB Address Coverage = %0.2f %%",
            apb_addr_cov.get_inst_coverage());
  $display ("APB Acknowledgment Coverage = %0.2f %%",
            apb_ack_cov.get_inst_coverage());
  $display ("SPI Write Address Coverage = %0.2f %%",
```

```
            spi_write_addr_cov.get_inst_coverage());
  $display ("SPI CRC Mode Coverage = %0.2f %%",
            spi_crc_cov.get_inst_coverage());
endfunction
```