Politecnico di Torino

Master's Degree Thesis
in Electronic Engineering

UVM environment for I3C Target Device

Supervisors

Prof. Maurizio MARTINA

Prof. Guido MASERA

Ing. Sandro SARTONI

Candidate

Federica BONGO

July 2023

Abstract

I3C Basic is a scaled-down and less complex version of the powerful, flexible and efficient I3C interface, suitable for a wide range of device connectivity applications, including sensor and memory interfaces. The I3C interface was developed by the MIPI Alliance and is designed to overcome the limitations of the I2C interface, while maintaining backward compatibility.
Similar to I2C, devices on the I3C bus communicate in a controller/target environment, where both the controller and target device can initiate communication.
In this project, the I3C target device used is the I3CS IP, which supports functions based on the MIPI I3C v.1.1.1. Due to the size and complexity of the project, functional verification becomes a challenge throughout the design flow. For this reason, a suitable verification environment must be developed to accelerate the verification phase. Starting from the architecture study of the target, the objective of this work, performed in collaboration with TDK InvenSense, is to develop a verification environment that can be used to test the correct functioning of the main operations of the I3CS IP target.
The testbench environment consists of the VIP block, which simulates the controller, the DUT, i.e. the design under test (IP I3CS), the Register File, which serves as the reference model; and a Scoreboard, which compares the actual values sent by the VIP with the values expected by the Register File. An interface must be defined between the VIP and DUT blocks so that the Controller (VIP) can communicate with the I3CS IP Target (DUT) via two buses, Serial Data (SDA) and Serial Clock (SCL). In this way, the controller can generate stimuli for the DUT to verify its correct behaviour.
The verification environment is based on the Universal Verification Methodology (UVM), with the UVM class library adding many automation functions such as sequences and data automation to the System Verilog language. In the UVM environment there are several components that together are responsible for driving the input tests to the Design Under Test (DUT), collecting the output transactions and finally comparing the actual results with the expected ones. To improve the verification of the I3CS IP target, it is necessary to test its functionality outside normal operating conditions by performing appropriate tests, including illegal frames.

Table of Contents

List of Tables

List of Figures

Acronyms

VIP
Verification IP

DUT
Design Under Test

UVC
Universal Verification Component

UVM
Universal Verification Methodology

OVM
Open Verification Methodology

SCL
Serial Clock

SDA
Serial Data

I2C
Inter Integrated Circuit

I3C
Improved Inter Integrated Circuit

BCR
Bus Characteristics Register

CCC

Common Command Code

DCR

Device Characteristics Register

SDR

Single Data Rate

HDR

High Data Rate

ENTDAA

Enter Dynamic Address Assignment

RTL

Register Transfer Level

SV

System Verilog

OOP

Object Oriented Programming

TLM

Transaction-Level Modeling

ACK

Acknowledgment

NACK

Negative Acknowledgment

Chapter 1

Introduction

1.1 Goal of the thesis and structure

The aim of the thesis activity is to verify the correct operation of the I3CS IP Target device using the UVM (Universal Verification Methodology). This activity is carried out by the Verification team at TDK Invensense, a company based in Milan.

The primary goal is to perform top-level verification of the I3C Target device architecture, identify and rectify any potential design errors. As digital systems become increasingly complex, ensuring comprehensive coverage of all scenarios and working conditions becomes more challenging. The UVM methodology is employed to address this challenge by providing tools for creating a verification environment capable of covering all cases.

The UVM methodology offers advantages such as reduced verification time through randomization and parallel simulation, and automatic generation of test scenarios.

By leveraging the UVM methodology, the thesis work aims to achieve thorough verification of the I3C Target device and ensure its compliance with the specifications outlined by the I3C protocol.

The topics covered in the various chapters of the thesis are outlined below.

In **chapter 2** the I3C protocol and UVM methodology are introduced in general. The parts that make up the I3C protocol, Controller and Target, are described at a high level, and particular attention is paid to the interfacing between these two blocks. Next, the UVM standard used for verification is described at a high level, focusing on the description of the fundamental aspects that make it efficient compared to other standards, namely the use of uvm_components and uvm_phases. The verification flow pursued in the thesis work will be detailed in the following chapters, providing a step-by-step description of the methodology used to verify

the I3CS IP Target device.

chapter 3 focuses on the analysis of the architecture of the Target I3CS IP device provided by the company. The features of the Target device are described and a particular emphasis is made on the modes it supports. These modes include SDR mode, Legacy I2C mode, and HDR mode. The chapter provides an overview of each mode and explains their significance in the context of the I3C protocol.

Additionally, the chapter introduces and explores various features of the I3C protocol that are supported by the Target device. This includes an overview of CCC (Common Command Code), which are standardized commands used by the Controller to communicate with the Target. The hot-join mechanism is also discussed, which allows the Target to join the I3C bus after it has been configured.

The communication frame between the Controller and Target is analyzed in detail. The frame consists of various components, such as the START condition, Header, Data, and STOP condition. Each component is explained, highlighting its role in the communication process between the Controller and Target.

In **chapter 4**, the focus is on the detailed structure of the UVM verification environment developed for the verification of the Target I3CS IP device. The discussion starts by delineating the reference test-bench that served as the foundation for developing the entire environment. The main blocks of the test-bench are identified, and their development is described.
The chapter provides insights into the thought process and decision-making involved in developing the sequences and tasks within the driver. It highlights the importance of considering various scenarios and working conditions to ensure thorough verification coverage of the Target device.

In **chapter 5**, the focus is on the analysis of the tests implemented during the thesis activity. The chapter begins by providing a list of the different test cases that are designed and executed. These test cases cover various scenarios and aspects of the Target I3CS IP device, including the main read and write operations.
Furthermore, the results generated by the executed tests are represented. The chapter includes information on the success or failure of each test case, any errors or issues encountered during the verification process, and any deviations from the expected behavior. The results provide valuable insights into the correctness and functionality of the Target device, as well as the effectiveness of the UVM test environment in capturing and detecting errors.

chapter 6 deals with conclusions highlighting the aims achieved in terms of completeness of the functional verification and effectiveness of the tests, moreover the skills acquired and the portability of the tests devised are emphasised.

Chapter 2

Overview of I3C Protocol and UVM Environment

2.1 Introduction to I3C Protocol

I3C is a two-wire bidirectional serial Bus, optimized for multiple sensor target devices and controlled by only one I3C controller device at a time. I3C is backward compatible with many Legacy I2C Devices, but I3C Devices support higher speeds up to a maximum of 12.5 MHz, new communication Modes, and new Device roles.

In the I3C protocol, communication between the Controller and the Target is facilitated through a two-wire interface consisting of the SDA (Serial Data) line and the SCL (Serial Clock) line.

The **SDA** line is responsible for carrying the actual data during the transfer. It is bidirectional, allowing both the Controller and the Target to transmit and receive data. The Controller initiates the communication by driving the SDA line to transmit data, while the Target responds by either driving the line to transmit data back or releasing it to allow the Controller to transmit.

The **SCL** line serves as a clock signal that synchronizes the communication between the Controller and the Target. It provides timing information to ensure that data is transferred at the correct rate. The SCL line is typically driven by the Controller, and both the Controller and the Target read and write data on the SDA line in synchronization with the clock pulses on the SCL line.

By using this two-wire interface, the I3C protocol enables communication between the Controller and the Target, allowing them to exchange data and synchronize their actions during the transfer. This approach ensures efficient and reliable data transfer in a simplified manner, as compared to protocols with additional communication lines.

fig:Communication Flow illustrates how I3C communication is initiated.

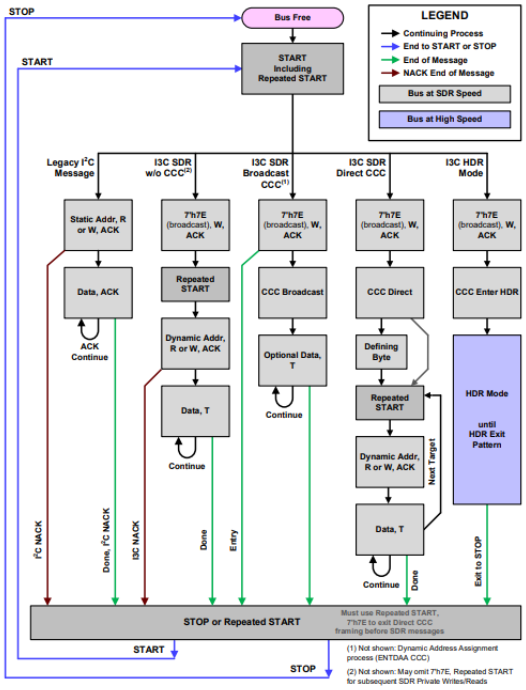

Figure 2.1: I3C Communication Flow

- **SDA** is a bidirectional data pin
- **SCL**

An I3C Bus supports the mixing of various Message types:

1. I2C-like SDR Messages, with SCL clock speeds up to 12.5 MHz
2. Broadcast and Direct CCC Messages that allow the Controller to communicate to all or one of the Targets on the I3C Bus, respectively
3. I2C messages to Legacy I2C Targets

2.1.1 I3C Controller Device

In the I3C protocol, the Active Controller refers to the I3C device that currently has control over the bus and is actively initiating and controlling the communication. The Active Controller can be considered as the "master" device in the I3C bus system.

Typically, the Active Controller is the device that sends the majority of the I3C Commands (CCC) on the bus. These commands can be either Broadcast CCCs, which are intended for all targets on the bus, or Directed CCCs, which

are specific to individual targets. The Active Controller uses these commands to initiate various operations and control the behavior of the targets on the bus.

Moreover, the Active Controller is the only device on the I3C bus allowed to send I2C Messages. This means that it can communicate with I2C devices that are present on the bus, providing backward compatibility with the legacy I2C protocol.

By being the Active Controller, a device assumes the role of actively controlling the communication on the I3C bus, sending commands and interacting with the targets. This distinction helps in managing the bus and coordinating the communication between different devices effectively.

In addition to send I3C Commands and I2C Messages, an I3C Controller Device also performs several other important functions in the I3C bus system. Some of these functions include:

- **Dynamic Address Assignment**: The I3C Controller Device is responsible for assigning unique Dynamic Addresses to the Target devices connected to the bus. It initiates the Dynamic Address Assignment procedure by broadcasting the ENTDAA (Enter Dynamic Address Assignment) command.

- **Arbitration**: The I3C Controller Device participates in address arbitration when multiple devices on the bus attempt to drive an address simultaneously. It follows the Open Drain approach and competes with other devices, including other Controllers and Targets, to determine the device that successfully drives the address onto the bus.

- **Clock Generation**: The I3C Controller Device generates and controls the clock signal (SCL) on the bus. It ensures that the clock signal is synchronized with the data signal (SDA) for reliable and accurate communication between the devices.

- **Bus Management**: The I3C Controller Device manages the overall operation and behavior of the I3C bus. It controls bus transactions, timing, and protocols to ensure proper data transfer and synchronization between the devices.

- **Power Management**: The I3C Controller Device may also have the capability to manage power-related functions on the bus. It can control power modes, perform power management operations, and coordinate power-related activities with the connected devices.

- **Error Handling**: The I3C Controller Device monitors and handles errors that may occur during communication on the bus. It detects and manages bus errors, data integrity issues, and other error conditions to ensure reliable and robust communication.

Overall, the I3C Controller Device plays a crucial role in managing and controlling the I3C bus system. It is responsible for initiating communication, assigning addresses, coordinating transactions, generating clocks, and ensuring smooth and efficient operation of the bus.

2.1.2 I3C Target Device

An I3C Target Device primarily listens to the I3C Bus for relevant I3C Commands sent by the Active Controller and responds accordingly. It acts as a "slave" device in the communication process. Additionally, an I3C Target Device always supports I3C SDR Mode, which is the basic mode of operation for I3C.

Unlike the I3C Controller Device, an I3C Target Device does not generate the bus clock (SCL). It relies on the clock generated by the Active Controller for synchronization during data transfer. The Target Device follows the timing and synchronization provided by the Active Controller.

For addressing, the I3C Target Device supports the Dynamic Address Assignment method, known as ENTDAA. This method allows the Target Device to participate in the Dynamic Address Assignment procedure initiated by the Active Controller. Through this procedure, the Target Device can obtain a unique Dynamic Address assigned by the Controller, which it uses for subsequent communication on the I3C Bus.

In summary, an I3C Target Device listens to commands, supports SDR Mode, follows the bus clock generated by the Active Controller, and participates in Dynamic Address Assignment for proper addressing on the bus. In addition the I3C Target Device can optionally:

- Generate Hot-Join events
- Request to become Active Controller, if the I3C Target Device also has I3C Controller Device capability.

[1] [2]

2.2 Introduction to Verification

The development of a verification environment for the I3C Target Device is a crucial aspect of the thesis, and traditional testbenches may not be sufficient for verifying the functionality of a large-scale design like the I3CS IP Target. The UVM (Universal Verification Methodology) methodology is chosen to address these challenges.

UVM is a SystemVerilog-based verification methodology that provides a standardized framework for developing modular, reusable, and robust verification

environments. It is built upon the OVM (Open Verification Methodology) version 2.1.1, which was created by Accellera. The UVM Class Library offers pre-defined building blocks and guidelines that aid in the development of well-structured and reusable verification components and test environments.

The primary goal of the verification process is to compare the RTL (Register Transfer Level) implementation of the design with its intended functionality. This involves defining the expected behavior of the DUT and comparing it against the observed behavior during simulation. The discrepancies or logic errors observed between the expected and observed behavior highlight potential functional logic errors in the design.

These logic errors can occur due to different factors and the verification process helps to uncover these errors and ensure that the design functions as intended according to the specifications.

By using the UVM methodology and developing a comprehensive verification environment, the thesis aims to enhance the verification process, improve test coverage, and detect any functional logic errors in the I3C Target Device design.

2.2.1 Verification Plan

The three key aspects of a verification plan in hardware design are:

- Coverage Measurements;
- Stimulus Generation;
- Response Checking.

Coverage Measurement This section is the one in which the verification scopes are described. This section determines if all bugs have been found .

Stimulus generation This section is responsible to generate the input test vector required to test the whole behaviour of a design. This part is crucial, since it generates not only valid test vectors but also invalid test vectors to drive the device outside of normal operating parameter in order to check the error detection logic of DUT. The goal is to generate test-vectors that allow reaching an high coverage level.

Response Checking The response checking section is responsible to verify the DUT responses conform to the specifications regarding the Reference model. One of the approaches used for this section is based on a Register File and Scoreboard in a structure like the one shown in Figure 2.2. For example, written and read values from Register File should match. When a written operation is performed to

the design, the Scoreboard receives the packet with output value from VIP and it should be the actual value. After that, the same Register File is read back from the design and the data is the expected value which is always sent to UVM Scoreboard. At this point the Scoreboard can compare the expected and the actual values to check if they match.

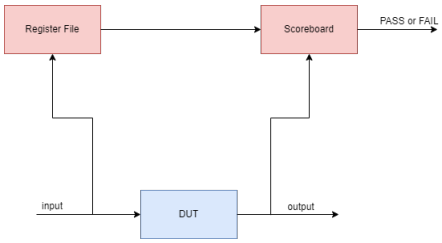

Figure 2.2: Scoreboard approach

2.3 The Universal Verification Methodology

The Universal Verification Methodology is an IEEE standard and the power of this methodology consists mainly in three aspects.

- The drastic reduction in the cost of verification: the verification engineer is assisted by the tools available within the UVM class library, thanks to which there is an increase in productivity.
- Reusability: it is possible to make use of previously created VIPs and configure them for new DUT with the same interfaces.
- Interoperability: the UVM standard makes it possible to standardise the construction of verification environments, allowing easy communication between different verification teams.

UVM is based on SV (System Verilog), a hardware description and verification language. It is open-source and compatible with all the major commercial simulator like Cadence, Mentor Graphics, Aldec and Synopsys. The base classes in the UVM hierarchy largely fall into three distinct categories:

- **UVM Components**: These classes represent the fundamental building blocks of a verification environment. They encapsulate reusable functionality and provide a modular structure for organizing the verification environment. Examples of UVM components include the *uvm_component* class, which serves as the base class for all UVM components, and the *uvm_test* class, which is used to define the top-level test sequences and scenarios.

- **UVM Phases**: Phases are used to orchestrate the execution of different tasks within the verification environment. They provide a structured flow for initializing, configuring, running, and shutting down the verification components. The UVM phases allow for better control and synchronization of the verification process. Some of the key UVM phases include the *build_phase*, *connect_phase*, *run_phase*, and *shutdown_phase*.

- **UVM Transactions**: Transactions represent the communication between the testbench and the DUT. They encapsulate the data and control information exchanged between the verification components and the DUT. UVM transactions provide a standardized format for stimulus generation, response checking, and functional coverage collection. The *uvm_transaction* class is a base class for creating custom transaction classes tailored to specific communication protocols or interfaces.

These base classes, along with other supporting classes and utilities, form the foundation of the UVM methodology. They promote **reusability**, **scalability**, and **maintainability** in the verification process by providing a standardized framework for developing verification environments. By leveraging these classes, verification engineers can focus on developing the specific test scenarios and sequences needed to thoroughly verify the functionality of the design.

In UVM, the following phases are commonly used to synchronize and coordinate the activities of different components within the test-bench:

Build Phase

- **uvm_build_phase**: This phase is responsible for constructing and initializing the components of the testbench. It is typically used to create and configure the necessary objects, set up connections, and allocate resources required for the testbench.

- **uvm_connect_phase**: In this phase, the connections between the different components of the testbench are established. It ensures that the signals and interfaces are correctly connected and ready for communication.

- **uvm_end_of_elaboration_phase**: This phase occurs after the elaboration of the design and testbench is complete. It is often used for performing any final configuration or initialization tasks before the simulation starts.

- **uvm_start_of_simulation_phase**: This phase marks the beginning of the simulation and is typically used to set up initial stimulus or prepare the testbench for simulation.

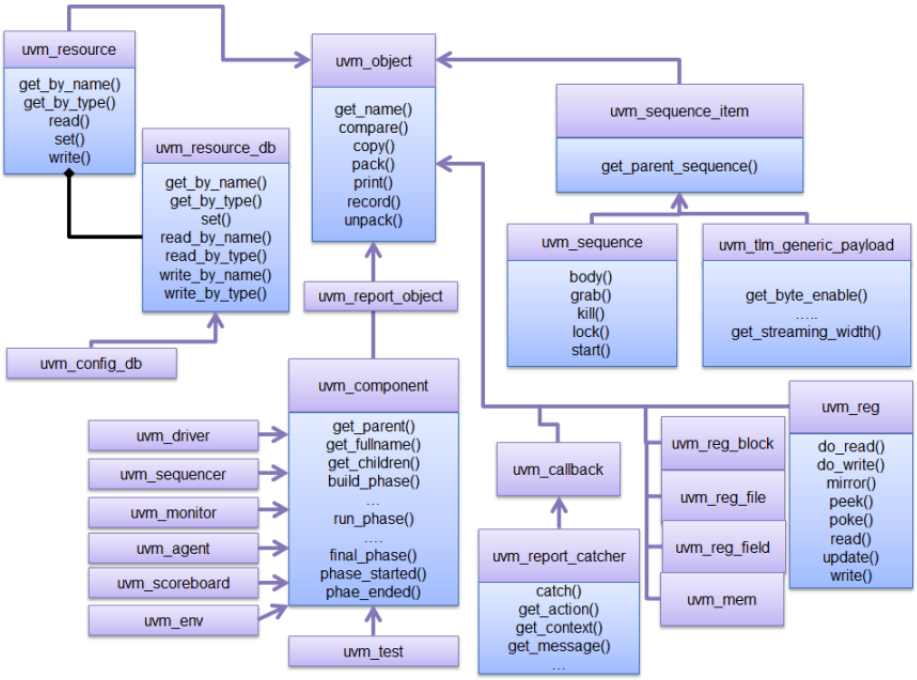

Figure 2.3: UVM Class Diagram

Run phase

- **uvm_run_phase**: The run phase is the main phase where the testbench executes the test scenarios and sequences. It includes generating stimulus, driving the signals, and verifying the expected behavior of the design.

Clean up phase

- **uvm_extract_phase**: This phase is used for collecting and analyzing coverage data, functional coverage, and other metrics related to the verification progress.
- **uvm_check_phase**: It checks for any unexpected conditions in the verification environment.
- **uvm_report_phase**: It reports results of the test.
- **uvm_final_phase**: It ends the simulation.

These phases provide a structured and synchronized flow for the testbench components, ensuring that they execute their tasks in the appropriate order. By

using these phases, the testbench components can collaborate effectively and exchange information at the right stages of the verification process.

2.3.1 The UVM Components

UVM is based on OOP, this allows to increase the redeployment. UVM Library provides a set of useful class from which deriving object and components, each class contains methods to deal with common operations.
The UVCs present in the verification environment are developed to communicate with the Target I3CS IP, and they behave as a Controller because:

- They send I3C commands: the UVCs generate and send I3C commands to the Target I3CS IP. These commands can be broadcast commands that target all devices on the bus or directed commands that address specific devices. By sending these commands, the UVCs emulate the behavior of an I3C Controller.
- They control the bus clock: the UVCs are responsible for generating and controlling the bus clock (SCL) signal during communication with the Target I3CS IP. They synchronize the data transfer and ensure that the communication occurs at the correct timing and frequency.
- They handle the communication protocol: the UVCs implement the I3C communication protocol, including the formatting of messages, addressing, and data transfer. They follow the rules and specifications defined by the I3C protocol to ensure proper communication with the Target I3CS IP.

By behaving as a Controller, the UVCs simulate the actions and behavior of an I3C Controller device, allowing for the verification of the Target I3CS IP's response and behavior in various scenarios and test cases.

Moreover,UVM uses TLM (Transaction-Level Modeling) APIs to facilitate communication between UVM components. TLM provides an abstraction layer that allows components to exchange transactions, which are packets of data representing specific actions or events in the DUT.

Sequences are used to define a sequence of transactions or actions to be performed by a UVM component. These sequences can be customized and extended to represent various scenarios and test cases. The sequences generate transactions that carry the necessary information and commands for the DUT.

Methods such as **put** and **get** are used to send or receive transactions between components. The **put** method is used to send a transaction from one component to another, while the **get** method is used to receive a transaction. These methods provide a standardized interface for communication and ensure that transactions are correctly passed between components.

By using TLM APIs and combining sequences and methods, UVM components can effectively exchange transactions and communicate with each other, enabling the coordinated and synchronized execution of test scenarios and the verification of the DUT's behavior.

Design Under Test It is basically the RTL description in the designing language. It describes the features and functions of the design.

Sequencer It is the entity on which the sequences will run. In order to test DUT behavior, sequence of transaction needs to be applied. Sequencer runs stimulus generation code and sends sequence items down to driver whenever driver demands by it.

Driver It acts as an active component in the verification environment.

It receives sequence items from the sequencer, which encapsulate the necessary data and information about the desired transactions. The driver then translates these sequence items into appropriate signal values on the interface of the DUT. It uses the appropriate protocol or interface-specific mechanisms to send the transactions to the DUT.

Monitor It plays a crucial role in capturing and analyzing the behavior of the DUT, enabling the verification environment to collect coverage information, perform checks, and ensure the correctness of the design under verification.

Agent It serves as a bridge between the testbench and the DUT. It manages the communication and data flow between the testbench and the DUT through its driver and monitor. The sequencer within the agent controls the generation and sequencing of transactions or stimulus that are sent to the DUT.

The agent can operate in two modes: passive and active.

Scoreboard It is responsible for verifying the correctness of the DUT's behavior by comparing its output signals, registers, or other relevant data with the expected values. It ensures that the DUT is producing the expected results according to the specified functionality or requirements.

Environment It is a higher-level structure that assembles and manages various components of the verification environment. It provides a modular and organized approach to building the testbench for verifying the design under test.

Test It is the top-level component in the verification environment hierarchy. It is represented by a class that is derived from the uvm_test base class provided by the UVM library.

The test class serves as a container for configuring the testbench and coordinating the overall verification process. It allows you to control the dynamic behavior of the

testbench components, such as the DUT (Design Under Test), VIP (Verification IP), and other UVM components, by utilizing sequences.

Sequence items They are the necessary data objects that are passed at an abstract level between the verification components. Sequence items can contain fields or properties that represent different aspects of the transaction, such as addresses, data values, control signals, or any other relevant information. These fields are typically defined as class variables within the sequence item class.

Sequences They are responsible for generating a set of transactions or stimuli to be applied to the design under test (DUT). Sequences gather sequence items, which encapsulate the necessary data for each transaction, and combine them to create a coherent set of inputs.

Sequences can be either randomized or pre-determined, depending on the specific testing requirements.

[3]

Chapter 3

Target I3CS IP architecture

Prior to delving into the implementation of the UVM framework, it is important to provide an introduction to the device under verification and its features.

The DUT, in this case, is the I3CS Target IP, which is designed to adhere to the specifications outlined in the MIPI I3C Standard v1.1.1.

The I3CS supports the following features:

- I2C Private Read and Private Write,
- I3C SDR Private Read and Private Write,
- I3C CCC Broadcast and Direct commands,
- I3C Hot Join request,
- I3C ENTDAA procedure.

3.1 Bus Configuration

An I3C bus can have the following compatible devices connected to it:

- I3C Primary Controller,
- I3C Secondary Controller,
- I3C Target,
- I2C Target.

In the context of I3C, a pure bus refers to a scenario where there are no I2C devices present on the bus, and all devices connected to the bus are I3C devices.

On the other hand, a mixed bus refers to a scenario where both I3C devices and I2C devices are present on the bus.

When an I3C bus is initially configured, it is done so in SDR mode by the I3C Primary Controller. The Primary Controller is responsible for controlling and managing the bus operations. If there is an I3C Secondary Controller present on the bus, it operates as a Target to the Primary Controller. The Secondary Controller follows the instructions and commands issued by the Primary Controller and participates in the bus transactions accordingly.

It's worth nothing that I3C is designed to be backward compatible with I2C, allowing I2C devices to coexist on the same bus. However, in a pure I3C bus configuration, there won't be any I2C devices present, and all devices on the bus will adhere to the I3C protocol.

A typical bus typology is reported in Figure 3.1.

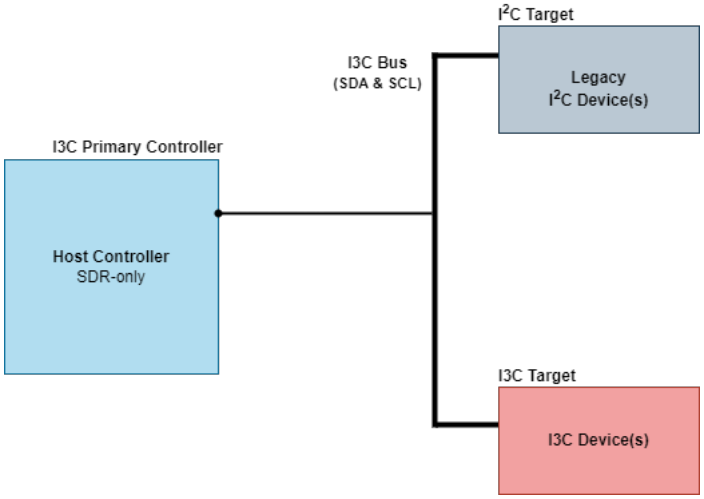

Figure 3.1: Typical I3C Bus Configuration

[1]

3.2 Start, Stop and Restart Conditions

I3C uses a Frame encapsulation method. Each I3C Frame consists of the START, Header, Data, and STOP, which are always present.

Every transaction on the I3C bus begins with a START condition. This condition occurs when the SDA line transitions from high to low while the SCL line remains consistently high. The Target detects the Start condition on the bus, while the Controller initiates it.

Conversely, all transactions on the I3C bus conclude with a STOP condition asserted by the Controller. A STOP condition is triggered by a low-to-high transition on the SDA line while the SCL line remains consistently high. The Target detects the STOP condition on the bus, while the Controller asserts it.

Instead of relying solely on the STOP condition, the I3C protocol also introduces a RESTART condition. This allows for the transmission of multiple messages within the same frame without requiring a stop and start signal between each message.

It is worth noting that the Start, Restart, and Stop conditions in the I3C Protocol are identical to those found in the I2C protocol. The behaviour of the SDA and SCL signals during the START, STOP and RESTART condition are shown in Figure 3.2, Figure 3.3 and Figure 3.4 respectively

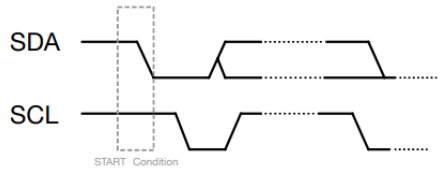

Figure 3.2: START Condition

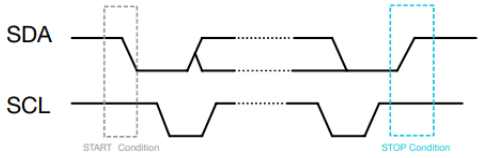

Figure 3.3: STOP Condition

3.2.1 I3C Address Header

After initiating a START, the Header in I3C serves the purpose of Bus Arbitration. The Controller utilizes the Header to address the Target Device.

An I3C Bus consists of one Controller and one or more Targets. A device that possesses both I3C Controller and I3C Target capabilities cannot simultaneously function as both a Controller and a Target. Instead, it must be configured to operate either as an I3C Target Device or as an I3C Controller Device.

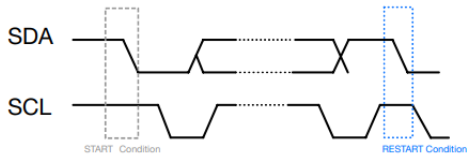

Figure 3.4: RESTART Condition

Within the I3C protocol, various Controller and Target Device Roles are defined to represent the functional capabilities of each respective I3C device. Each I3C Device must support at least one Device Role, although it can be designed to support multiple Device Roles. The supported Device Roles of an I3C Device are exposed through its BCR (Bus Characteristics Register).

During a data transfer, the bus Controller initiates the process. It begins by issuing a START signal to all connected devices. Subsequently, the Controller sends the ADDRESS of the specific Target device it intends to communicate with. The Read or Write operation signal bit is also transmitted alongside the ADDRESS bits. All devices connected to the bus compare the received address bits with their own address. If there is no match, they simply wait until the bus is released. Conversely, if the address matches, the chip generates an ACK (Acknowledgment) signal in response.

Upon receiving an acknowledgement, the Controller proceeds with the transmission of DATA. Each data byte comprises 8 bits, and an acknowledgement or transition bit follows each transferred byte. Once the transmission is complete, the Controller issues the STOP signal.

3.2.2 I3C Address Arbitration

An Address Header that follows a START condition, excluding a Repeated START, is susceptible to Arbitration. This means that both the Controller and one or more Targets may contend for control of the Bus and attempt to drive their respective Addresses onto the Bus using the SDA line. The Arbitration model employed in this scenario adheres to the standard Open Drain approach.

In this model, both the Controller and the Target(s) that are transmitting an Address must adhere to the same rule:

1. If the current bit to transmit is a 0, then the Device shall drive SDA Low after the falling edge of SCL and hold Low until the next falling edge of SCL.

2. If the current bit to transmit is a 1, then the Device shall not drive SDA, but rather shall High-Z SDA on the falling edge of SCL.

3.3 I3C SDR Data Words

An I3C Message is considered an SDR Message if it adheres to the specifications and requirements of the SDR mode in the I3C protocol. Here are the conditions that define an SDR Message in I3C:

- The Address in the Address Header could be 7'h7E, that is the I3C Broadcast Address. All I3C Targets shall match Address value 7'h7E. While, no I2C Target will match the Broadcast Address because this value is reserved and unused in I2C.

- The Address in the Address Header matches the Target's Dynamic Address. All I3C Targets shall match their own Dynamic Address.

In summary, an I3C Message is classified as an SDR Message when it follows the specifications and requirements of the SDR mode in terms of mode of operation, data rate, frame format, signal levels, and device support.

The I3C SDR mode is backward compatible with I2C protocol and conditions, which allows for legacy I2C Target Devices to coexist with I3C devices on the same I3C bus. The MIPI I3C Specification strongly suggests that legacy I2C devices incorporate 50 ns spike filters on the SDA and SCL pads to make it possible for them to ignore the I3C traffic higher speeds. With spike filters implemented for all I2C targets on the bus, the I3C bus can operate at the maximum rated clock frequency. The I3C Target module on this device can be used in I2C Target mode until it is assigned a Dynamic Address.

In Figure 3.5 is reported the frame about I3C SDR mode and about the I2C Legacy mode.

In I3C SDR, the Data Words match I2C only in the sense that they are both 9 bits long. I3C SDR Data Words differ from I2C in two ways.

- **Ninth Bit of SDR Controller Written Data as Parity :** In I2C, the ninth Data bit written by the Controller serves as an ACK from the Target. However, in I3C, the ninth Data bit written by the Controller represents the Parity of the preceding eight Data bits. As a result, in I3C, the Target is not required to drive the SDA line for Data written by the Controller in SDR mode.

  In the context of SDR, the ninth bit of Write data in I3C is commonly referred to as the Transition Bit (T-bit).

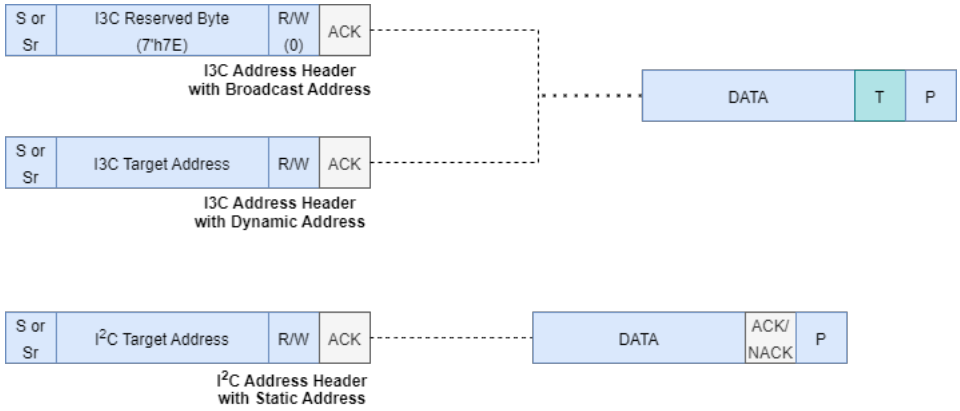

Figure 3.5: Address Header Comparison

- **Ninth Bit of SDR Target Read Data as End-of-Data:** In I2C, the ninth Data bit transmitted from the Target to the Controller serves as an ACK by the Controller. However, in I3C, this ninth Data bit has a different purpose. It allows the Target to terminate a Read operation, and it enables the Controller to abort a Read operation.In the context of SDR mode in I3C, the ninth bit of Read data is referred to as the T-Bit. It has specific significance in determining the end of a Read operation. To facilitate this, a Target should incorporate an SDA Read detector. This detector monitors whether the SCL clock has remained unchanged for a duration of 100 microseconds (us) or more. If such a condition is detected, the Target can abort the ongoing Read operation by switching SDA to a High-Z state (high impedance) and waiting for a Repeated START or STOP condition.

Ninth Bit of SDR Controller Written Data as Parity bit In I3C, the ninth data bit of each SDR Data Word written by the Controller is a Parity Bit, which is calculated using odd parity. The value of this Parity bit is determined by performing an XOR operation on the 8 Data bits along with a binary 1, as follows: XOR(Data[7:0], 1).

During the SCL High period, Parity bit writes should be maintained valid. This means that the Parity bit should remain stable and not change during this period. When the T-Bit represents the last data byte being transmitted, the write of the Parity bit is kept valid through the SCL High period. Once the SCL transitions to Low, the Controller can then proceed to either change the SDA line or leave it unchanged in preparation for the Repeated START or STOP condition that follows.

Ninth Bit of SDR Target Read Data as End of Data In I2C, the Read operation from the Target is controlled by the Controller only, which means that the Target has no control over the amount of data it returns. However, in I3C SDR, the Target has the ability to control the number of data words it returns, providing more flexibility. Additionally, the I3C Controller also has the ability to abort the Read operation prematurely if necessary.

The mechanism that controls the Read operation in I3C SDR is based on the ninth (T) data bit of each data word returned by the Target. The Target can return the ninth bit in one of three ways:

- No Change: the Target keeps the ninth bit the same as the previous data word, indicating that there is more data to be read.
- Change to 1: the Target changes the ninth bit to 1, indicating that it has finished returning data and there is no more data to be read.
- Change to 0: the Target changes the ninth bit to 0, indicating an abort condition. This allows the Controller to prematurely abort the Read operation if necessary.

By manipulating the ninth bit in this manner, the Target and Controller can effectively control the amount of data transferred during a Read operation in I3C SDR.

- The I3C Target returns the ninth bit as 0 (SDA Low) to end the Message:
  - The Target shall set SDA Low on the falling edge of SCL.
  - On the following rising edge of SCL, the Target shall set SDA to High-Z.
  - The I3C Controller shall drive SDA Low on the rising edge of SCL.
  - The I3C Controller then shall issue either a STOP, or a Repeated START.
- The I3C Target returns the ninth bit as 1 (SDA High) to continue the Message:
  - The Target shall set SDA High on the falling edge of SCL.
  - On the following rising edge of SCL, the Target shall set SDA to High-Z.
- The Target shall monitor the SDA on the falling edge of SCL:
  - If SDA is High, then the Target shall continue with the next value.
  - If SDA is Low and if there has been a Repeated START, then the Message has been aborted, and the Target shall not drive SDA after that.

3.3.1 Transition from Address ACK to SDR Controller Write Data

The end of any Address Header is an ACK or NACK by the one or more addressed Targets, using Open Drain on SDA:

- If 7'h7E, then it is the ACK of all I3C Targets on the Bus.
- If a single Target Address, then it is the ACK (or NACK) of the addressed Target, or a NACK if no such Target is on the Bus.

When the Address Header results in an ACK, I3C SDR specifies how the handoff is to occur.

3.4 Legacy I2C Transaction on I3C Bus

Until the Dynamic Address is assigned, the I3C Target operates in I2C Target Mode and uses the Static Address to represent itself on the bus. When in this mode, the Controller can use an **I2C Write** Transfer to write data to the Target directly and an **I2C Read** Transfer to read data from the Target directly. It is possible for an I2C Transaction to take place even when the bus is configured to operate in I3C SDR Mode or when an I3C SDR Transaction is in progress. For instance, the Controller can choose to transmit I3C Broadcast Address 7'h7E/W followed by a Restart and I2C Static Address to begin an I2C Transaction while in I3C SDR Mode.

3.4.1 I2C Private Read and Private Write

I3C Targets are capable of acting as standard I2C Targets as long as they have an I2C Static Address. The I3CS IP supports two types of I2C transfers:

1. Legacy I2C transfer starting with the Static Target address
2. Legacy I2C transfer starting with the 7'h7E broadcast address

Both types of transfer support Read and Write access types.

In legacy I2C and I3C SDR the Controller must use the following protocol to transmit the register offset then the data to the I3CS IP.

The ADR field is the Target Static or Dynamic address, OFFSET is the register offset. DATA is an unbounded series of bytes, The first phase is always a Write transaction to set the OFFSET, the second phase can be a Read or Write transaction.

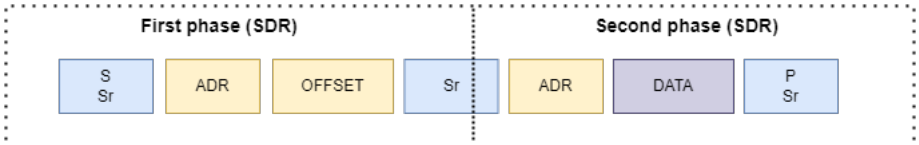

Figure 3.6: Legacy I2C and I3C SDR protocol

In **Read mode**, the first DATA byte will come from a register read access at address OFFSET. Then each subsequent byte will come from a read access at the previous address incremented by one if ena_autoinc is set to one.
In **Write mode**, the first DATA byte will be sent to register write access at address OFFSET. Then each subsequent byte will be sent to a write access at previous address incremented by one through ena_autoinc signal.

- The I3C START and STOP are identical to the I2C START and STOP in their signaling, but they vary in their timing. In I3C SDR, a STOP or Repeated START is tolerated any time that SCL is high while the Controller checks SDA or SDA is Open-Drain. This is unlike I2C, which wants STOP or Repeated START only after a NACK of an address, or after ACK/NACK of data.

- The I3C Address Header is identical to the I2C Address Header in bit form and in signaling, but it may vary from I2C in its timing.

- The Data 9-bit Words use the same bit count as I2C, but differ in the ninth bit where the acknowledge is substituted with the Transition Bit.

3.5 Dynamic Address Assignment Mode

The Active Controller assumes the responsibility of conducting the Dynamic Address Assignment procedure, which aims to assign a unique Dynamic Address to every I3C Device connected to the Bus. Once a Target device receives a Dynamic Address, it will use this address for all subsequent transactions on the I3C Bus, unless the Controller decides to change it, if applicable.

The Dynamic Address Assignment procedure involves an Address Arbitration process that shares similarities with I2C. However, it diverges from I2C by incorporating the concatenated values of the 48-bit Provisioned ID, BCR (Bus Characteristics Register), and DCR (Device Characteristics Register). In each Arbitration round, the Device on the I3C Bus with the lowest concatenated value emerges as the winner, and the Controller assigns a unique Dynamic Address to each victorious Device.

3.5.1 Dynamic Address Assignment Procedure

The Dynamic Address Assignment procedure in I3C is initiated by the Active Controller through broadcasting the ENTDAA CCC. If a Target device does not have a Dynamic Address already assigned, it automatically participates in this procedure.

During the Dynamic Address Assignment, the Target device sends its own 48-bit Provisional ID, BCR, and DCR in Open Drain mode to engage in Arbitration. The Target device will win the Arbitration if it has the lowest concatenated value of the Provisional ID, BCR, and DCR.

Once the Target device wins the Arbitration, the Active Controller transfers a 7-bit Dynamic Address to the Target device, followed by the Parity T-bit. If the parity is valid, the Target device acknowledges (ACK)s the Active Controller, stores the Dynamic Address, switches to I3C SDR mode, and updates its Dynamic Address accordingly.

However, if the parity is invalid, indicating an error in the data transmission, the Target device passively NACKs (negative acknowledgement) the Active Controller and waits for the next Arbitration round to retry the Dynamic Address Assignment procedure. Figure 5.16 shows the frame format for a typical Dynamic Address Assignment procedure.

3.5.2 Target Device48-bit Provision ID

A Device that supports the Broadcast Command Code **Enter Dynamic Address Assignment** shall have a 48-bit Provisioned ID. The Controller shall use this 48-bit Provisioned ID, unless the Device has a Static Address and the Controller uses the Static Address.

The 48-bit Provisioned ID is composed of three parts:

1. **Bits[47:33]:MIPI Manufacturer ID** (15 bits) only the 15 Least Significant Bits are used.

2. **Bit[32]:Provisioned ID Type Selector**(One Bit, 1'b1:Random Value, 1'b0:Vendor Fixed Value)

3. **Bits[31:0]**:32 bits containing either a Vendor Fixed Value or a Random Value, depending on the value of Bit[32]. If the value of **Bit[32]** is **1'b0** : Vendor Fixed Value

3.5.3 CCC (Common Command Code)

Common Command Codes are standardized commands that are universally supported and can be transmitted by the Controller in the I3C bus. These commands

can either be directed to a specific target or broadcasted to all targets simultaneously. The CCC protocol is formatted using I3C SDR mode and always commences with the I3C Broadcast Address (7'h7E/W). This specific address is recognized by all I3C targets present on the bus. However, any I2C target on the bus will NACK the request as 7'h7E is a reserved address in the I2C protocol.
Each CCC is assigned a unique 8-bit command code. The command code space is divided into two categories: Broadcast CCCs and Direct CCCs.

Broadcast CCCs encompass command codes ranging from 0x00 to 0x7F. These codes are used for commands that are broadcasted to all targets on the bus simultaneously.

On the other hand, Direct CCCs include command codes from 0x80 to 0xFE. These codes are utilized for commands directed specifically to a particular target.

To differentiate between the two types of CCCs, targets can examine the Most Significant bit (bit 7) of the command code. If bit 7 is set to 0, it indicates a Broadcast CCC, whereas if it is set to 1, it signifies a Direct CCC.

For a comprehensive list of the command codes associated with all supported CCCs in this Target module, please refer to table 3.1.

All the Broadcast CCCs share the same general frame format and has the following sequence:

- Start or Restart, followed by the Broadcast Address,
- Broadcast CCC value, followed by any required defining byte or data,
- End of command.

All the Direct CCCs share the same general frame format and has the following sequence:

- Start or Restart, followed by the Broadcast Address,
- Direct CCC value, followed by any required defining byte or data,
- Restart, followed by the address of the targeted Target, followed by any required defining byte or data,
- Repeat step 3 if the Controller wants to address multiple targets in the same CCC transaction,
- End of command.

In this thesis activity, the tested CCC is the ENTDAA command. However, it's important to note that the Target module also supports other commands listed

in table 3.1. The Controller employs the Broadcast ENTDAA CCC to signal all I3C devices on the bus to initiate the Dynamic Address Assignment procedure, as outlined in the Dynamic Address Assignment section. The Target module will participate in this procedure unless it already has a Dynamic Address assigned. In such cases, the Target will NACK the ENTDAA command and wait for the next Start condition.

It's important to note that the ENTDAA CCC always concludes with a Stop condition (not a Restart).

Table 3.1: CCCs Table

Common Command Code (CCC)	**Type**		**Value**	**Brief Description**
ENEC	Enable Events Command	Broadcast Write	0x00	Enable Target events
		Direct Write	0x80	such as Hot-Join
DISEC	Disable Events Command	Broadcast Write	0x01	Disable Target event
		Direct Write	0x81	such as Hot-join
ENTDAA	Enter Dynamic Address Assignment	Broadcast Write	0x07	Enter Controller initiation of Dynamic Address Assignment Procedure
RSTDAA	Reset Dynamic Address Assignment	Broadcast Write	0x06	Discard Current
		Direct Write	0x86	Dynamic Address
SETNEWDA	Set New Dynamic Address	Direct Write	0x88	Controller assigns new Dynamic Address to a Target
GETPID	Get provisional ID	Direct Read	0x8D	Controller queries Target's Provisional ID
GETDCR	Get Device Characteristics Register	Direct Read	0x8F	Controller queries Target's Device Characteristics Register
GETBCR	Get Bus Characteristics Device	Direct Read	0x8E	Controller queries Target's Bus Characteristics Register
RSTACT	Target Reset Action	Broadcast Write	0x2A	Controller configures and/or queries Target Reset action and timing
		Direct Write and Read	0x9A	
SETMRL	Set Maximum Read Length	Broadcast Write	0x0A	Controller sets
		Direct Write	0x8A	maximum read length and IBI payload size
SETMWL	Set Maximum Write Length	Broadcast Write	0x09	Controller sets
		Direct Write	0x89	maximum write length
GETMRL	Get Mximum Read Length	Direct Read	0x8C	Controller queries Target's maximum possible read length and IBI payload size
GETMWL	Get Maximum Write Length	Direct Read	0x88	Controller queries Target's maximum possible write length
GETMXDS	Gets Maximum Data Speed	Direct Read	0x94	Controller queries Target's maximum read and write data speeds and maximum read turnaround time
SETBUSCON	Set Bus Context	Broadcast Write	0x0C	Controller specifies a higher-level protocol and/or I3C specification version
GETSTATUS	Get Device Status	Direct Read	0x90	Controller queries Target's operating status

3.6 Hot-Join Mechanism

The Hot-Join mechanism in I3C allows a Target to join the I3C bus even after the bus has been configured according to the I3C Bus Configuration. Hot-Join is typically used in scenarios where the Target remains depowered until needed or when the Target is physically inserted into the I3C bus without disrupting the SDA and SCL lines.

To initiate a Hot-Join request, the Target must meet the following conditions:

- The Target is Hot-Join capable.
- The Target does not have a Dynamic Address already assigned.
- Hot-Join is enabled on the bus by the Controller.

The Target can initiate a Hot-Join request and once the request has been made, the Target will wait for the Bus Idle condition before proceeding. In the standard Hot-Join process, the Target will issue a Start on the bus by pulling the SDA line low. The Active Controller acknowledges the Start condition by sending clocks on the SCL line, marking the beginning of the Arbitrable Address Header. During this phase, the Target transmits the 7'h02/W Hot-Join Address on the bus.

However, it is not always necessary for the Target to wait for the Bus Idle condition to occur. If another device on the bus issues a Start signal before the Bus Idle condition, the Target can still participate in the Address Arbitration process. In this case, the Target will passively engage in the arbitration by transmitting the 7'h02/W Hot-Join address on the bus. After the Target successfully wins the address arbitration and the Controller acknowledges the Hot-Join request, the Controller proceeds with sending the Broadcast ENTDAA CCC on the bus. This initiates the Dynamic Address Assignment procedure for the Target, following the guidelines outlined in the Dynamic Address Assignment section.

In the event that the Hot-Join request is unsuccessful, either due to the Controller NACKing the request or the Target losing arbitration, the Target will continue to attempt the Hot-Join request in subsequent opportunities. In the case of standard Hot-Join, the Target will retry at the next Bus Idle condition. In the case of passive Hot-Join, the Target will retry at the next Start condition on the bus. The Target will make further attempts until the Hot-Join request is successfully acknowledged. The frame format of a successful Hot-Join transaction is shown in

3.7 HDR Mode

The I3C HDR (High Data Rate) modes are specifically designed to enable the transfer of larger amounts of data while maintaining the same bus frequency.

However, it's important to note that the Target module on this particular device does not support the HDR modes specified in the MIPI I3C Specification.

Despite not supporting the HDR modes, the Target module is still capable of detecting HDR Enter and Exit Patterns. This allows the module to appropriately respond to bus traffic involving HDR operations, ensuring compatibility and seamless communication within the I3C bus.

HDR Enter Pattern To initiate an HDR mode, the Controller would typically broadcast an Enter HDR Mode CCC, using the corresponding CCCs (ENTHDR0 through ENTHDR7). However, in the case of the Target module being discussed, it does not support HDR modes or their corresponding CCCs.

Nevertheless, the Target module is still capable of detecting when the Controller is attempting to enter an HDR mode. In such situations, the Target module will disregard all bus traffic until it detects an HDR Exit Pattern. This allows the Target module to appropriately handle the situation and maintain proper communication on the bus, despite not supporting the HDR modes.

HDR Exit Pattern Once an HDR mode is entered, the Controller has the ability to exit the mode by using the HDR Exit Pattern. It is important to note that whenever the Controller exits an HDR mode, it always transitions back to SDR mode. The same HDR Exit Pattern is employed to exit any HDR mode.

When the Target module detects the HDR Exit Pattern, it resumes monitoring SDR traffic on the I3C bus. This allows the Target module to re-engage with the bus and continue normal communication in SDR mode.

Additionally, there is an alternative pattern called the HDR Restart Pattern, which enables the Controller to send multiple messages within HDR mode without the need to transition back to SDR mode between messages. However, it's worth mentioning that the Target module on this particular device does not support any HDR modes, and therefore cannot detect or utilize the HDR Restart Pattern.

Chapter 4

UVM Environment

The crucial point of this thesis is the realization of the UVM Environment. According to the redeployment system of UVM, all the components are derived from the UVM base classes exploiting inheritance.

4.1 Test-Bench scheme

In Figure 4.1 is reported the reference framework structure with the main components.

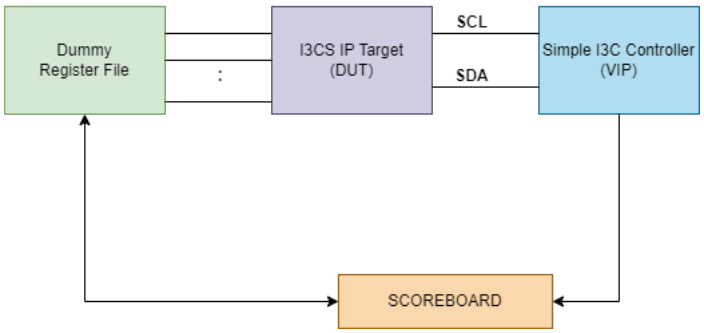

Figure 4.1: Test-Bench Scheme

The verification environment follows a typical structure in UVM methodology with some modifications to accommodate the specific requirements of the design.

The VIP serves as the Controller and communicates with the DUT through the SDA and SCL buses. It is responsible for generating the necessary signals and transactions to interact with the DUT.

The DUT, in this case, is the I3CS IP Target, which is the block being verified. It receives commands and messages from the VIP and responds accordingly.

The Register File block acts as a reference model, containing the expected values for comparison. It provides a benchmark against which the actual values generated by the VIP can be compared.

The Scoreboard component plays a crucial role in the verification environment. It compares the expected values from the Register File with the actual values from the VIP and determines whether they match. It keeps track of the correctness of the behavior and provides information on the number of matches and mismatches.

The overall verification environment is developed using the UVM methodology, which provides a standardized approach for creating modular and reusable testbenches.

There are some modifications made to the typical UVM structure and these modifications could include customization in the sequence of operations, handling of interfaces, or other aspects of the testbench architecture. Adapting the UVM methodology to fit the requirements of the design is common practice to ensure efficient and effective verification.

4.2 UVM Structure

A schematic view of the typical UVM framework structure is reported in Figure 4.2 with the main components and their role in the verification process.

In the modified structure (Figure 4.3) of the verification environment for the thesis project, the decision is made to exclude the monitor component. The monitor is typically a passive component responsible for capturing DUT signals using a virtual interface and translating them into sequence items or transactions. These transactions are then transmitted to other components such as the UVM scoreboard using a TLM analysis port.

However, in this thesis project, the monitor component is not included, and instead, the DUT is connected directly to the scoreboard. This modification is made to simplify the structure of the environment. This flexibility to customize the structure is one of the advantages provided by the UVM standard.

The two main components in the modified structure are "TB_TOP_UVM" and "UVM_TEST".

- "TB_TOP_UVM" is a component that combines the DUT (Design Under Test) and the Register File. This component represents the top-level of your testbench hierarchy and includes the actual DUT implementation as well as the reference model provided by the Register File.

- "UVM_TEST" describes the UVM environment and serves as the test class derived from the UVM base class. It encompasses two main blocks or components:

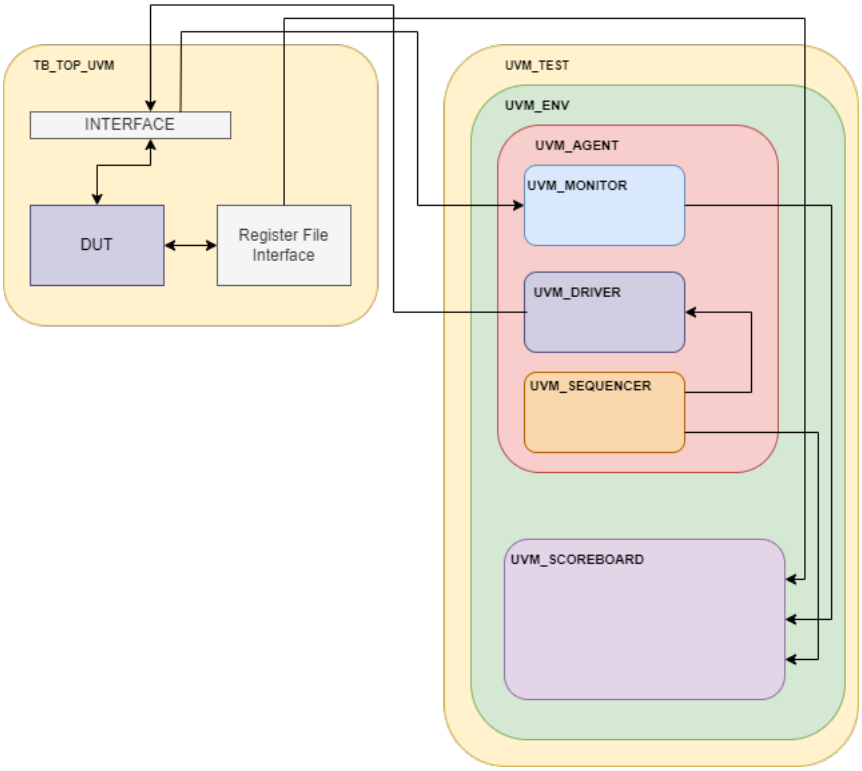

Figure 4.2: Typical UVM Framework Structure

1. **UVM_AGENT**;
2. **UVM_SCOREBOARD**;

By customizing the structure and excluding the monitor component, it has tailored the verification environment to meet the specific requirements of the project while leveraging the flexibility and reusability provided by the UVM standard.

The UVM_AGENT in the verification environment is an active component that is responsible for both driving and capturing signals. It consists of three main components:

- UVM_SEQUENCER: The sequencer is tasked with generating random test vectors or sequences. It uses the test scenarios and constraints to create input transactions that will be applied to the DUT. These sequences are designed to thoroughly test the functionality of the DUT.
- UVM_DRIVER: The driver is responsible for receiving the input transactions from the sequencer and sending them to the DUT. The driver interfaces with the DUT's input ports and follows the specified communication protocol to ensure proper interaction with the DUT.

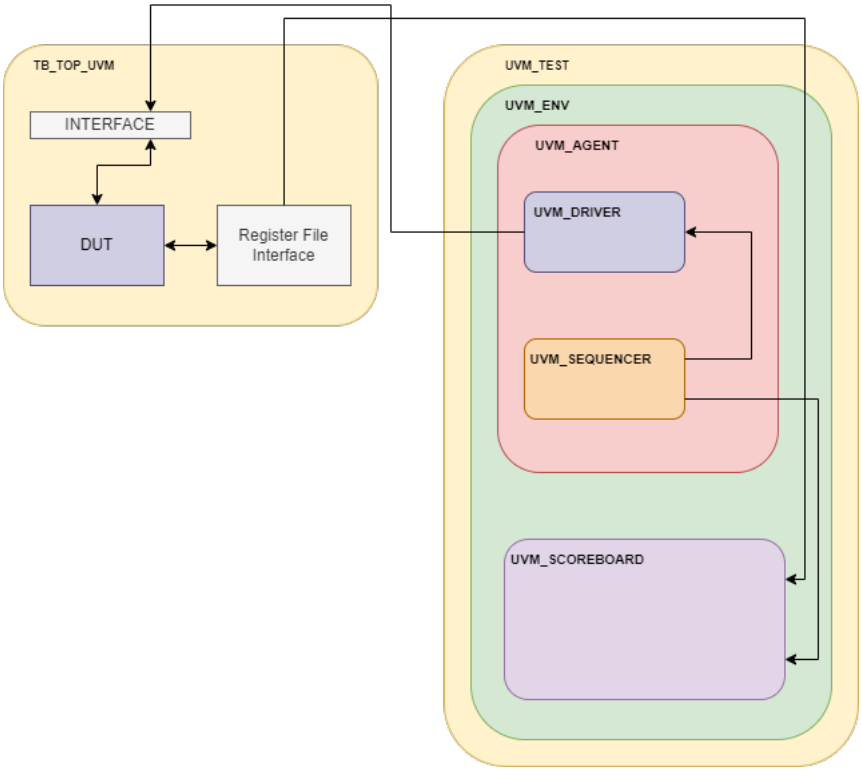

Figure 4.3: UVM Framework Structure

- UVM_SCOREBOARD: The scoreboard receives the expected data from the Register File and the actual data from either the VIP or the DUT. It performs a comparison between the expected and actual data to determine if they match or not. Based on this comparison, the scoreboard provides a pass or fail indication, showing whether the DUT behaves correctly or not.

Overall, the UVM_AGENT actively drives the verification process by generating test vectors through the sequencer, sends the input transactions to the DUT using the driver, and captures and verifies the results through the scoreboard. This active agent plays a critical role in the verification flow and ensures the correctness of the DUT's behavior based on the comparison between expected and actual data.

4.2.1 Top

The operations performed in the described section of the UVM hierarchy, within the tdk_i3c_uni_hvl_top, are as follows:

1. Clock Generation: This operation involves generating the system clock (mclk) that will be used for driving the DUT and synchronizing the UVM components.

The clock generation ensures that all the components operate in sync and at the desired frequency.

2. Reset De-assertion: After a certain number of cycles, the reset signal (rst_b) is de-asserted. The reset signal is initially asserted to ensure a known state for the DUT, and it is de-asserted after a specific duration to start the functional operation of the DUT.

3. Interface Assignments: In this operation, the necessary interfaces between the DUT and the UVM framework are defined. The UVM interfaces provide a connection between the DUT and the UVM components, allowing them to capture and analyze signals from the DUT. The signals required for verification purposes are hierarchically assigned to the interfaces, making them accessible within the UVM framework.

4. Declaration of the Design (I3CS IP): The design under test, which in this case is the I3CS IP, is declared within the UVM hierarchy. This declaration establishes the DUT as an instance within the UVM environment, enabling communication and interaction between the DUT and the UVM components.

5. Test Run: Once the necessary configurations and connections are established, the UVM test is launched using the run_test() function. This initiates the execution of the UVM testbench, including the generation of test sequences, driving of inputs to the DUT, capturing of outputs, and verification using the UVM components.

The described operations in this section of the UVM hierarchy set up the necessary infrastructure for the UVM testbench to interact with the DUT. It ensures proper clocking, assigns interfaces for signal capture, declares the DUT within the UVM environment, and initiates the execution of the UVM test.

Interface

In UVM, communication between the testbench and the DUT is facilitated using virtual interfaces. A virtual interface represents a collection of signals that are used to drive the DUT from the testbench.

In order to access the signals of the DUT through the virtual interface, each UVM component that needs to drive interface signals must declare a virtual interface instance of the corresponding interface. The reference to the interface is obtained from the UVM configuration database.

The interface used in the *tdk_i3c_uni_controller_interface.sv* file contains a set of signals, such as SCL and SDA, which need to be driven to send input vectors to the DUT at each clock cycle.

The signals in the virtual interface represent the buses used for communication, where SCL is the clock signal and SDA is the data signal.

By using virtual interfaces, the testbench can interact with the DUT and drive the necessary signals for testing, while the DUT can also provide outputs through the virtual interface for analysis and verification by the testbench.

4.2.2 Sequences

The *sequencer* is a verification component that allows the execution of sequences of instructions to drive the device, on which the control of test progress depends. There is a hierarchical organisation, according to which the highest-level component called the Virtual Sequencer, synchronises the lower-level sequencers integrated in the UVCs. The latter in turn route instruction packets to the drivers via the TLM ports. In general, a UVM sequence must fulfil the following tasks:

- sending the START
- waiting for the *trigger events*
- command of write and read operations, thanks to which UVCs, move the lowest level lines, using the implemented protocol to perform writes or reads to the received addresses;
- sending the STOP
- sending read data on the lines to the scoreboard

In the UVM framework, the *uvm_sequence_item* class is used as a base class to define and declare data items or transactions. These transactions represent the information that needs to be exchanged or captured between the testbench and the DUT. The *uvm_sequence_item* class provides a set of useful methods to randomize the transaction fields, compare or print transaction objects, and perform other operations. These methods help in generating randomized test vectors and manipulating the transaction data.

In the specific UVM framework developed for the Target device under verification, there are two different sequence objects: one for input transactions and another for output transactions. These sequence objects are responsible for generating the appropriate input or output stimuli and coordinating the flow of transactions between the testbench components. Respectively, the definition of the two sequence object is present in

- *tdk_i3c_uni_controller_sequence_item.sv*

- *tdk_i3c_uni_controller_packet.sv*

The behaviour of the signals on the interface pads corresponds to that described in the protocol being used, whereby the lines that are moved by the master UVC are consistent with the *timing specification* reported in Table 4.1. The creation of custom sequences is described in detail in the dedicated paragraphs of the test chapters.

Furthermore, Figure 4.4 and Figure 4.5 show the reference diagram of the sequence hierarchy levels implemented to generate the frame for the write operation and the frame for the read operation with the Static Address respectively. Instead, Figure 4.6 and Figure 4.7 show the reference diagram of the sequence hierarchy levels implemented to generate the frame for the write operation and the frame for the read operation with the Broadcast Address respectively.

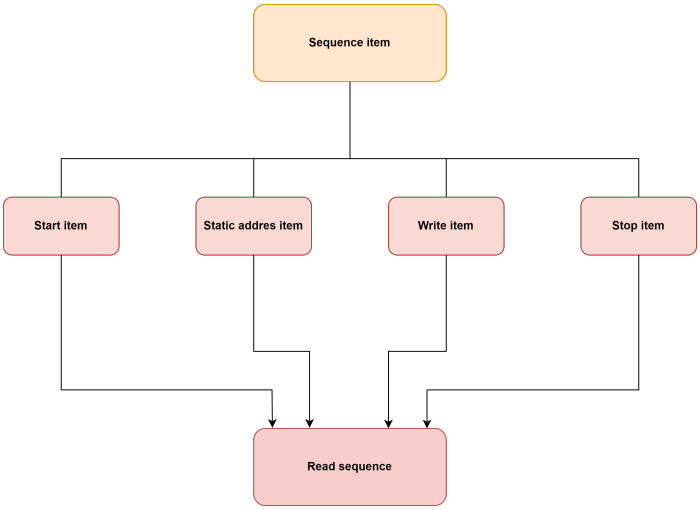

Figure 4.4: Reference diagram of the sequence hierarchy levels for write operation with Static Address

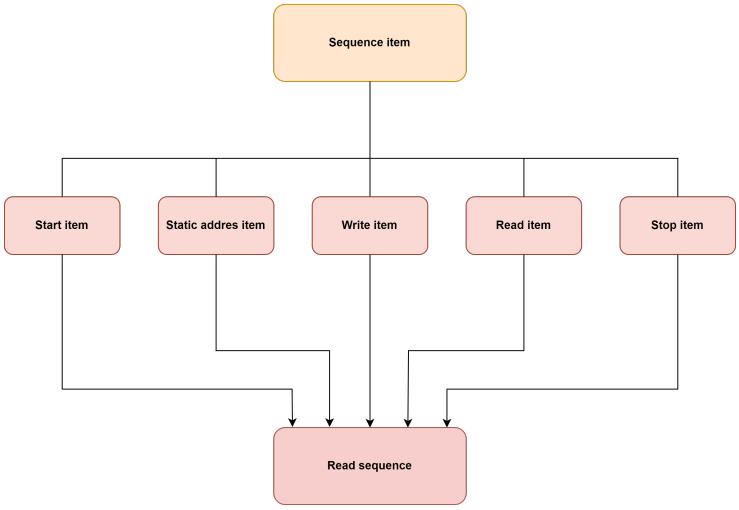

Figure 4.5: Reference diagram of the sequence hierarchy levels for read operation with Static Address

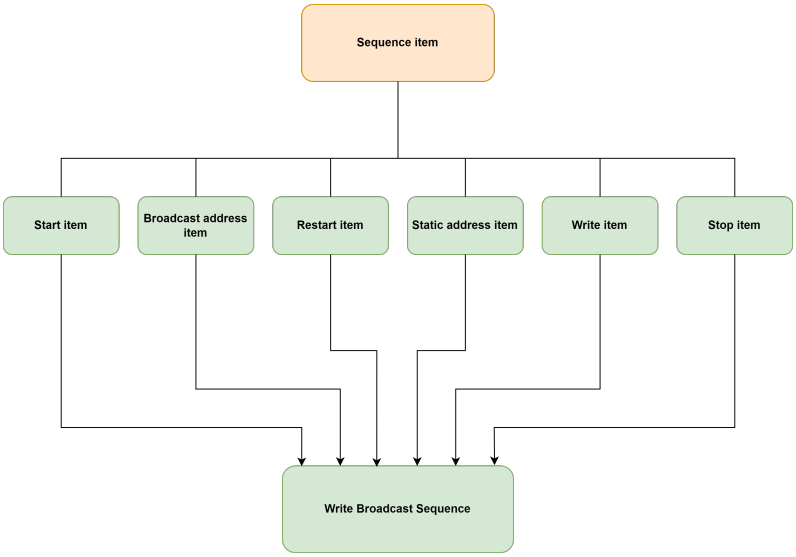

Figure 4.6: Reference diagram of the sequence hierarchy levels for write operation with Broadcast Address

tdk_i3c_uni_controller_sequence_item In this class the signals included are shown in the code file reported below. Some of them are defined as *rand* to randomize input values and create input vectors. Then there is the usage of utility and field macros, because the UVM uses the concept of a *factory* where all objects are registered with it so that it can return an object of the requested type when

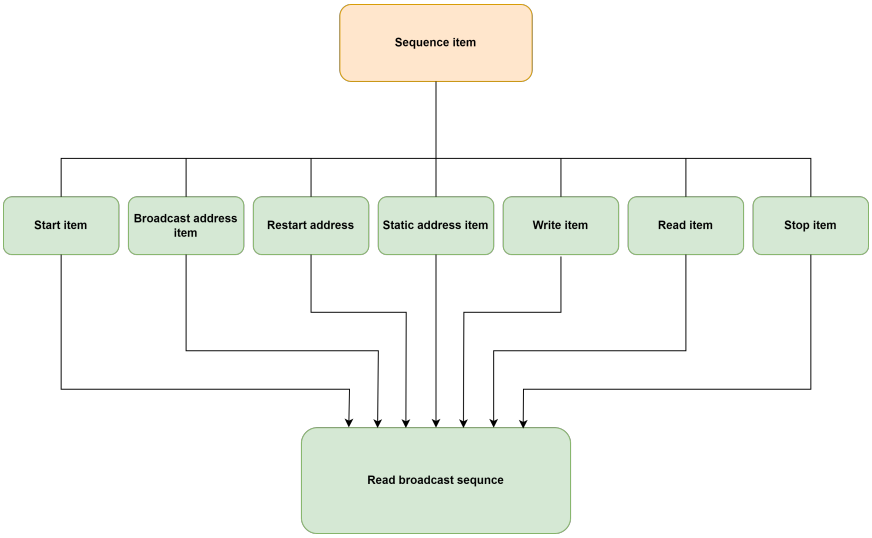

Figure 4.7: Reference diagram of the sequence hierarchy levels for read operation with Broadcast Address

required. The utility macros help to register each object with the factory. The *utils* macro is mandatory for the *new* function to be explicitly defined for every class.

Listing 4.1: tdk_i3c_uni_controller_sequence_item code

```
`ifndef _TDK_I3C_UNI_CONTROLLER_SEQUENCE_ITEM_SV
`define _TDK_I3C_UNI_CONTROLLER_SEQUENCE_ITEM_SV

class tdk_i3c_uni_controller_sequence_item extends uvm_sequence_item;
//declaration of data

  rand i3c_direction_t                  direction;
  rand i3c_initialization_t             initialization;
  rand i3c_address_acknowledge_t        address_acknowledge;
  rand i3c_acknowledge_t                acknowledge;
  rand i3c_address_t                    is_address;
  rand i3c_rw_t                         is_rw;
  rand i3c_stop_t                       is_stop;
  rand i3c_parity_bit_t                 is_parity_bit;
  rand i3c_T_bit_t                      is_T_bit;
  rand i3c_data_t                       is_data;
  rand i3c_par_bit_address_t            is_par_bit_addr;
  rand i3c_transmit_ack_t               is_transmit_ack;

  rand logic    [6:0]           i3c_addr;
  rand logic    [7:0]           data;
  rand int unsigned             N_bit;

  `uvm_object_utils_begin (tdk_i3c_uni_controller_sequence_item)
  `uvm_field_enum(              i3c_direction_t, direction,UVM_ALL_ON)
  `uvm_field_enum(              i3c_initialization_t, initialization,
   UVM_ALL_ON)
  `uvm_field_enum(              i3c_address_acknowledge_t,
   address_acknowledge,UVM_ALL_ON)
  `uvm_field_enum(              i3c_acknowledge_t, acknowledge,
   UVM_ALL_ON)
  `uvm_field_enum(              i3c_address_t, is_address,UVM_ALL_ON)
  `uvm_field_enum(              i3c_rw_t, is_rw,UVM_ALL_ON)
  `uvm_field_enum(              i3c_stop_t, is_stop,UVM_ALL_ON)
  `uvm_field_enum(              i3c_T_bit_t, is_T_bit,UVM_ALL_ON)
  `uvm_field_enum(              i3c_parity_bit_t, is_parity_bit,
   UVM_ALL_ON)
  `uvm_field_enum(              i3c_par_bit_address_t, is_par_bit_addr
   ,UVM_ALL_ON)
  `uvm_field_enum(              i3c_data_t, is_data,UVM_ALL_ON)
  `uvm_field_enum(              i3c_transmit_ack_t, is_transmit_ack,
   UVM_ALL_ON)
  `uvm_field_int (              i3c_addr,                   UVM_ALL_ON)
  `uvm_field_int (              data,                       UVM_ALL_ON)
  `uvm_field_int (              N_bit,                      UVM_ALL_ON)
 `uvm_object_utils_end

//macros fields
```

```
43    function new (string name = "tdk_i3c_uni_controller_sequence_item")
        ;
44      super.new(name);
45    endfunction
46
47    function void post_randomize();
48      `uvm_info(get_type_name(), $sformatf("\n%s" ,this.sprint()),
       UVM_NONE)
49    endfunction
50  endclass
51
52  `endif //_TDK_I3C_UNI_CONTROLLER_SEQUENCE_ITEM_SV
```

4.2.3 Sequencer Operations

In order to verify the main read and write operations of the Target device, several sequence classes are used, which put together form the complete protocol frame. In this way, there are several sequences that perform different tasks to verify different aspects of the design. Going to consider a generic write operation frame there are:

- Start
- Static Address Target/Broadcast Address with RW and ACK
- Write Data bytes
- Stop or Repeated Start

A sequence class is associated to each frame. So, there is:

- *tdk_i3c_uni_controller_start_sequence.sv*,
- *tdk_i3c_uni_controller_address_sequence.sv*,
- *tdk_i3c_uni_controller_repeated_start_sequence.sv*,
- *tdk_i3c_uni_controller_write_sequence.sv*,
- *tdk_i3c_uni_controller_stop_sequence.sv*

Every sequence class are an extension of *tdk_i3c_uni_controller_sequence_item* where in the body the operations and constraints are defined and they are the stimulus for DUT. Combining these existing sequences, it is possible to create new ones, in order to perform Start sequence followed by Static address sequence followed by Restart sequence and go

on until the stop sequence. In this way it is possible to create frames for each mode of the protocol, for example I2C mode, SDR mode or Dynamic Address mode.

This structure turns out to be very flexible and makes it easy to verify the behavior of the Target.

The same sequences are used for the read operation, but *tdk_i3c_uni_controller_write_sequence.sv* is substituted with *tdk_i3c_uni_controller_read_sequence.sv*.

4.2.4 Packet sequence

Packet Sequence is the container of the output transaction and it is an extension of the uvm_sequence_item base class. The signals captured from the DUT are shown in the following piece of code.

Listing 4.2: Packet sequence code

```
1  `ifndef _TDK_I3C_UNI_CONTROLLER_PACKET_SV
2  `define _TDK_I3C_UNI_CONTROLLER_PACKET_SV
3
4
5
6  class tdk_i3c_uni_packet extends uvm_object;
7
8    rand bit [7:0] i3c_address;
9    rand bit [7:0] i3c_data ;
10
11     `uvm_object_utils_begin(tdk_i3c_uni_packet)
12         `uvm_field_int(i3c_address, UVM_ALL_ON)
13         `uvm_field_int(i3c_data, UVM_ALL_ON)
14     `uvm_object_utils_end
15
16     function new(string name="tdk_i3c_uni_packet");
17         super.new(name);
18     endfunction: new
19
20
21  endclass
22
23 `endif //_TDK_I3C_UNI_CONTROLLER_PACKET_SV
```

4.3 Environment

The environment is a container class, in general, it can contains one or more agents and other components such as the scoreboard. The environment is defined in *tdk_i3c_uni_sve.sv* and it contains the implementation of:

- Agent
- Scoreboard

The environment operations are concentrated only during the build and connect phases. After the declaration of the UVC's during the build phase, the creator function is called to create each of them. During the connect phase, the analysis port of Driver and Scoreboard are connected. This is an important step as analysis ports are the way transactions move throughout the UVM environment. A scheme of the connection is shown in Figure 4.8.

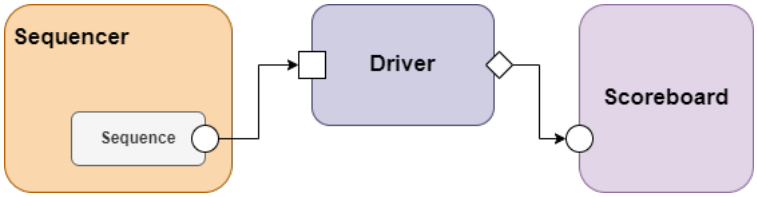

Figure 4.8: Types of Driver connection with Scoreboard and Sequence

It is important to note that the dots on the Scoreboard represent the implementation of the analysis ports while the diamond represents the analysis port.

4.3.1 Agent

The UVM framework requires an Agent that is an active component capable driving and capturing the transaction. Typically, an Agent contains a Driver and a Sequencer. The Agent is described in *tdk_i3c_uni_controller_agent.sv*.

tdk_i3c_uni_controller_agent It is responsible for the creation of the required UVC's during the build phase and for the connection of the sequencer export port with the driver port during the connect phase.

4.4 Driver

The driver class derives from *uvm_driver* base class and it is responsible to send the input vectors received from the sequencer to the DUT. The signals required are: SCL and SDA.

During the build phase, the Driver to Scoreboard port is created. The driver operations are executed during the *run_phase* task. In particular, if the request signal is high, then the driver gets a new transaction object from the sequencer using

get_next_item() method and associate the signals restrained in the transaction to the DUT signals. Once the signal has been driven to the DUT the transaction is sent to the scoreboard through the Driver to Scoreboard port using the *write()* method.

In general, the Driver's run phase is divided into several **case** loops based on desired sequences that are sent. Each case invokes tasks that have always been implemented in the driver and will be executed using the features or constraints of the selected sequence.

In this way, it is possible to choose how to compose the protocol frame.

4.4.1 Tasks of Driver

In the tasks, the main signals are driven: SCL and SDA so as to comply with the specifications of the I3C protocol.

The tasks implemented are reported below:

- start_condition();
- bit_construction_write();
- bit_construction_read();
- transmit_address();
- i3c_direction();
- get_slave_ack();
- ack_without_handoff();
- nack_without_handoff();
- write_data();
- read_data();
- transmit_ack();
- repeated_start_condition();
- stop_condition().

start_condition() To implement the Start condition in I3C, the SDA and SCL signals need to be driven following the specified timing requirements. The Start condition involves a high-to-low transition on the SDA line while the SCL line remains constant high. To achieve this in the testbench, the SDA and SCL signals are driven accordingly with the times specified in the I3C protocol manual MIPI v1.1.1 and reported also in Table 4.1.

bit_construction_write() To reconstruct the writing of a bit in I3C, where the bit changes on each falling edge of the SCL signal, it is necessary to drive the SDA and SCL signals accordingly with the specification time reported on the table Table 4.1.

bit_construction_read() This task drives SDA and SCL signals in order to reconstruct the reading of a bit from the Target. The latter must occur at each fall edge of SCL. In order to avoid conflicts on the bus SDA between Controller and Target, when occurs a Read operation the Controller should let Target drive the bus by going to leave SDA in high impedance.

transmit_address() This task handles the transmission of the address consisting of 7 bits. Within a **for** loop the task is called the task *bit_construction_write()*. Furthermore, within this task is also implemented the calculation of the parity bit (XOR negated between the address bit and 0) that must follow the 7 address bits in the Dynamic Address mode sequence.

i3c_direction() This task simply writes on SDA bus the direction of the Frame sent. In the case it writes a 0 bit, it means that a writing will occur, otherwise a reading will occur. Also in this case is evoked the *bit_construction_write* task.

get_slave_ack() This task simply, at the falling edge of SCL, reads the value of the Acknowledge on SDA bus. When ACK is 0 means that the Target has received the address, otherwise not. In this task is the Target to drive the SDA bus and so the Controller leaves the bus in high impedance.

ack_wihtout_handoff() and nack_wihtout_handoff() These two tasks are used in the case of the read or write operation when neither the controller nor the target sends the ACK bit. These are used mainly in the Dynamic Address mode, and ACK is represented with the bit 0, while NACK with the bit 1.

write_data() This task exploits the *bit_construction_write* task in a **for** loop to write the 8 bits of data. Also in this task there is the calculation of parity bit, named also Transition bit that is the XOR between every bit of data with 1. The Transition bit is used when the sequence sent implements the frame in SDR mode.

read_data() This task exploits the *bit_construction_write* task in a for loop, in order to read the 8 bits of data sent by Target. In addition, within this task there is a **case** loop about the Transition bit in the case the sequence sent implements the frame in SDR mode.

transmit_ack() This task is used in the case of a Read operation, when the Controller should communicate to the Target if it has read the data (bit 0) or if it has not read the data correctly (bit 1).

repeated_start_condition() In this task, the Controller drives the SCL and SDA bus in order to create the Repeated Start Condition, may be used to end one message and begin another within a single bus transaction. The Controller can generate a **Repeated START (Sr)** condition, setting the SDA line to one during the LOW phase of the SCL line.
The Repeated start condition is used in the following situations:

- To continue transmission with the same Target device in the opposite direction. After a Repeated START condition, the Controller sends the same Target device address followed by another direction bit.
- To start transmission to or from another Target device. After the repeated START condition, the Controller sends another Target address.

stop_condition() In this task, the Controller drives the SCL and SDA lines to generate the **STOP (P)** condition. The Controller changes the SDA line from zero to one while the SCL line is high.

4.5 Scoreboard

The Scoreboard is the most complex component in the UVM environment. It is an extension of the *uvm_scoreboard* base class. It verifies that everything is worked as expected by looking at input and output transaction. It has seen that transaction objects have been moved throughout the UVM framework using analysis port. Most of them are directed to the scoreboard where their implementation is defined. Analysis port works like a callback, each time that UVC's write a transaction to

the analysis port, the correspondent callback task is executed. It is a non-blocking mechanism that avoids time-delay in the verification environment.
The scoreboard is responsible for decoding the Write and Read operation's values from VIP and from Register file in order to compare them and provide a PASS or FAIL signal.
The operation of Read or Write is triggered by the use of *uvm_event* in the Sequences where is implemented the operation, while the Scoreboard waits for the events. In the run phase there is a *fork* which executes two processes in parallel. The first one is referred to the Write operation where the scoreboard waits for the write event from the Register file and the VIP, while the second one is referred to the Read operation, where the scoreboard waits for the read event always from the Register file and the VIP.
Following both processes, through the use of an **if** loop, there is a check on the VIP and Register File data to verify if they match. Once the packets are synchronized and compared, two *uvm_info* are executed. The first one is used to print out on the console the actual content of the packet of the DUT. The second one is executed after the comparison of data to show on the console the result of the comparison.

4.5.1 uvm_event

In the described project, the *uvm_event* is used for synchronization between processes in different components. The *uvm_event_pool* is a pool that stores *uvm_event* objects when the processes triggering and waiting for events are running in different components and need to share the event handle.

The *uvm_event_base* class provides several methods, two of which are used in this thesis:

1. **ev.trigger()**: This method is used to trigger the event. In the context of this thesis, the event is triggered by the first component (equivalent to the Sequence class) and is passed a packet as a parameter. The packet contains the address of the Register File and the data byte sent by the serial interface.

2. **ev.wait_ptrigger()** :This method is used to wait for the event to be triggered. In this thesis, the second component (the Scoreboard) uses this method to wait for the event triggered by the first component. The value of the packet is then analyzed by the Scoreboard.

The **ev.trigger(Packet)** method is used in the respective sequences such as

- *tdk_i3c_uni_I2C_write_sequence.sv*
- *tdk_i3c_uni_SDR_write_sequence.sv*

- *tdk_i3c_uni_I2C_read_sequence.sv*
- *tdk_i3c_uni_SDR_read_sequence.sv*

This method is used to detect write or read events. The **ev.wait_ptrigger(Packet)** method is used in the Scoreboard to synchronize with the triggered events. Additionally, **ev.trigger()** is also used to detect write or read events in the Register file, which is always waited for in the Scoreboard.

By using events and the provided methods, the different components in the project can synchronize and communicate with each other effectively.

4.6 Register File

The register file is developed in parallel with the implementation of the UVM environment. The Register File in question is designed in System Verilog language and it is integrated within the entire verification environment as an interface to the DUT as reported in Figure 4.9.
In the UVM environment, in particular, it is decided to implement the Register File not as a *module* but directly as an *interface*. This made it possible to simplify the connections of the Register File with the Scoreboard and the Top of the UVM environment (tdk_i3c_uni_hvl_top) by using *virtual interface* directly.

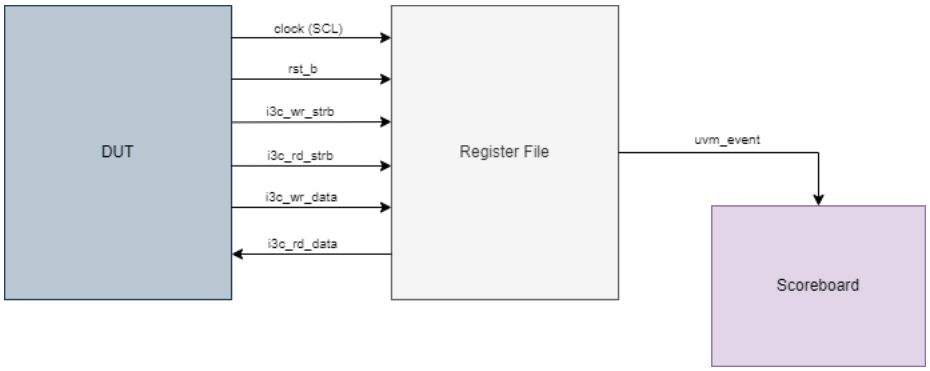

Figure 4.9: Register File connections with the DUT and Scoreboard

It is implemented according to the specifications of a generic Register File that has both write and read access. It consists of 256 addressable locations where each location is addressed by the signal *i3c_addr* that is the direct address received from the serial interface *pad_sdai* line (DUT).
The clock of the Register File is managed by the SCL line, the same of the I3C protocol, while the reset of the Register File is the same of the whole system *rst_b* that is driven in the top file code of the UVM environment (tdk_i3c_uni_hvl_top).

This architecture is designed to verify the absence of anomalies during writing and reading by design. For this reason, the Register File communicates with the Scoreboard, which compares the current values with the expected values, via **uvm_event**. Specifically, each time the serial interface sends a data, a location in the Register File pointed by the address *i3c_addr* is filled, which coincides with the first data byte sent by the serial interface SDA, and subsequent accesses to the locations are pointed by the increment of *i3c_addr* by one.
There are control signals that indicate whether a read or write access occurs, which in this architecture are: *i3c_wr_strb* and *i3c_rd_strb*. When a write occurs the *i3c_wr_strb* signal is raised, then the transcription of the byte sent to the location pointed to by *i3c_addr* takes place and afterwards the trigger of the write event in the Register File takes place. When a read occurs, on the other hand, the signal *i3c_rd_strb* rises, then the current value of the data in the location pointed to by *i3c_addr* is returned as an output, and the trigger of the read event in the Register File takes place afterwards.
Both the write event and the read event after they are triggered in the Register file, these are waited for, with the use of EV.PTRIGGER_WAIT(), by the Scoreboard in order for the data comparison to take place.

4.7 Timing Specification

Table 4.1, which provides reference timing requirements for Legacy I2C Mode, outlines the timing characteristics that must be adhered to in order to maintain compatibility with I2C devices. These timing requirements ensure reliable communication and interoperability between I2C and I3C devices when operating in Legacy Mode.

In Legacy Mode, the timing diagram showcases the timing relationships between various signals during an I2C communication transaction. The timing parameters specified in Table 4.1 define the specific timing requirements for different signals involved in the I2C communication.

The timing diagram typically includes the following signals:

- SCL: This signal represents the clock line of the I2C bus. It indicates the timing of data transfer.
- SDA: This signal represents the data line of the I2C bus. It carries the actual data being transmitted or received.
- START: This signal indicates the start of a communication transaction.
- STOP: This signal indicates the end of a communication transaction.

- ACK: This signal is used for acknowledgement of data transmission.

The timing parameters in Table 4.1 specify the minimum and maximum duration for various events, such as the setup and hold times for data, the rise and fall times of the clock and data signals, and the setup and hold times for the start and stop conditions. These parameters ensure proper timing synchronization between the communicating devices in order to adhere to the I3C protocol.The timing parameters specified in Table 4.1 and in Figure 4.10 serve as a reference for managing the SDA and SCL signals in the driver tasks.

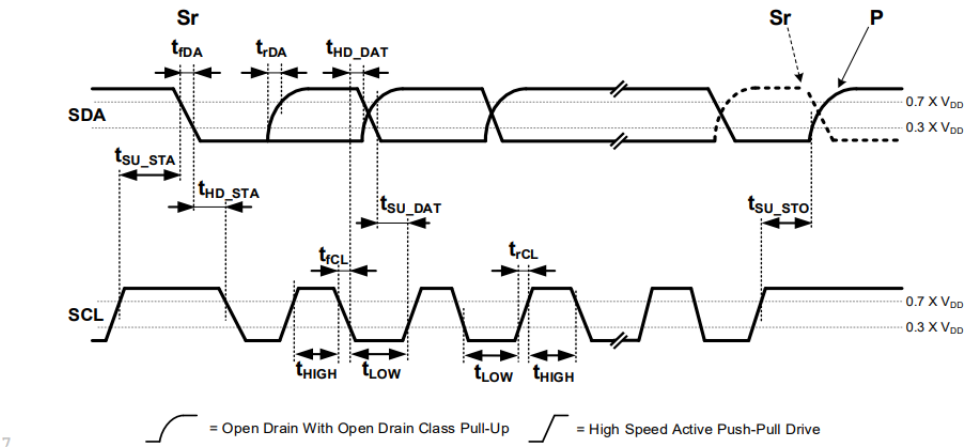

Figure 4.10: I3C Legacy Mode Timing

[3] [4] [5]

Table 4.1: I3C Timing Requirements When Communicating With I2C Legacy Devices

Parameter	Symbol	Min	Max	Units
SCL Clock Frequency	f_{SCL}	0	1.0	MHz
SCL Clock Low Period	t_{LOW}	500	-	ns
SCL Clock High Period (for Mixed Bus)	t_{HIGH}	260	-	ns
SCL Clock Rise Time	t_{rCL}	-	120	ns
SCL Clock Fall Time	t_{fCL}	-	120	ns
Setup Time for a (Repeated) START	t_{SU_STA}	260	-	ns
Hold Time for a (Repeated) START	t_{HD_STA}	260	-	ns
Data Setup Time	t_{SU_DAT}	50	-	ns
Data Hold Time	t_{HD_DAT}	-	-	ns
SDA Signal Rise Time	t_{rDA}	-	120	ns
SDA Signal Fall Time	t_{fDA}	-	120	ns
Setup Time for STOP	t_{SU_STO}	260	-	ns

Chapter 5

Test and Results

In this chapter, the tests are analyzed and the results of the verification procedure are shown and described. In the section 5.1, the packets sent to the design are described by proposing reference timings representing what we would expect as a result of the simulations on the Cadence tool and reference frames representing the highest-level sequence encompassing the other sequences used to create the I3C protocol frame. It is worth noting that in the frame images, each part of the frame is associated with a colour that indicates how the communication between Controller and Target takes place and it is shown in Figure 5.1

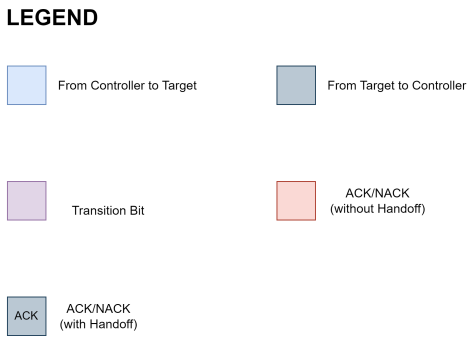

Figure 5.1: Frame images Legend

5.1 Tests

The UVM environment just described is generated from a higher-level class level, referred to as *test*, that extends directly from the *uvm_test* class of the *UVM Standard Library*. Other classes can be defined that overcome the base class; each of them is associated with a specific test and allows the use of the components

defined specifically for those tests. In this way, it is possible to call up:

- randomisation class used
- the specific scoreboard class implemented to control the feature to be tested
- the sequence responsible for sending writes and reads through the interface and, more in general, of the evolution of test operations.

These tests cover different test cases and a particular test case can be selected and execute by providing the UVM_TESTNAME command line argument as reported in Figure 5.2. This method allows more flexibility to choose different tests without modifying the testbench top every time it is necessary to run a different test.

```
#!/bin/csh -e

if ( $#argv > 0 ) then
  set testname = $1
else
  set testname = "tdk_i3c_uni_base_test"
endif
set run_dir = /server/scratch/${USER}/tdk_i3c_uni/sim/${testname}
rm  -rf $run_dir
mkdir -p $run_dir
cd run_dir
xrun \
  -uvm \
  -uvmhome CDNS-1.2 \
  -messages \
  -gui \
  -access +rwc \
  -top tdk_i3c_uni_hvl_top \
  -linedebug \
  +UVM_TESTNAME=${testname} \
  -F ${DV_TOP_DIR}/ver/sim.fof \
  +UVM_VERBOSITY=UVM_HIGH #\
```

Figure 5.2: File Code used to run the tests

5.1.1 UVM Base Test

It is a custom test called *base_test* that inherits from *uvm_test* and it is declared and registered with the factory.
Testbench environment component called *tdk_i3c_uni_sve* and its configuration object is created during the **build_phase**. It is then placed into the configuration database using *uvm_config_db* so that other testbench components within this environment can access the object and configure sub components accordingly.
A test sequence object is built and started on the environment virtual sequencer using its **start** method.

The Base Test can help in the setup of all basic environment parameters and configurations. Instead, in this project the other tests, useful to verify the Read

and Write operations in the different modes of the I3C protocol, are simply an extension of the *uvm_test* where different sequences are executed according to the test case under consideration.

The test cases considered are listed below:

- I2C Private Write with Static Target Address
- I2C Private Read with Static Target Address
- I2C Private Write with Broadcast Address
- I2C Private Read with Broadcast Address
- I3C Private Write
- I3C Private Read
- Dynamic Address assignment: I3C ENTDAA Procedure

The tests in question are the core of the thesis activity. In the follow sections there is a detailed description of the test classes implemented for testing the I3C Target Device, while the file codes in System Verilog of working tests are reported in appendix A.

5.1.2 I2C Private Write with Static Target Address Test

This test is executed by the *tdk_i3c_uni_I2C_write_test.sv*. The code is reported in appendix A.

The test in question runs the sequence *tdk_i3c_uni_controller_I2C_write_sequence* where is implemented the frame reported in Figure 5.4.

The operations executed by the sequence are:

- start condition;
- Static Target Address with 7 bit (7'h68);
- '0' to indicate the write operation (RW bit);
- ACK from the Target following the recognition of the register address;
- writing data bytes into the Target (4 bytes);
- ACK to indicate the successful reading by the Target for every data byte;
- stop condition.

The timing of reference is represented in Figure 5.3 and the frame of reference is reported in Figure 5.4.

5.1.3 I2C Private Read with Static Target Address

The I2C Read operation with Static Target Address is executed by the *tdk_i3c_uni_I2C_read_test.sv*. This test calls the sequence *tdk_i3c_uni_I2C_controller_read_sequence.sv*. The sequence file invokes other sequences that all together execute the following operations:

- Start Condition;
- Static Target Address with 7 bits (7'h68);
- '1' to indicate the Read Operation (RW bit);
- ACK from the Target following the recognition of the register address;
- writing one data byte to indicate the address of Register File;
- reading data bytes from the Target;
- ACK to indicate the successful reading by the Controller for every data byte;
- Stop Condition.

The timing of reference is reported in Figure 5.5 and the frame of reference is shown in Figure 5.6.

5.1.4 I2C Private Write with Broadcast Address

The test is executed by the *tdk_i3c_uni_I2C_write_bd_test.sv*. This Test runs the sequence implemented in *tdk_i3c_uni_I2C_controller_write_bd_sequence.sv*.

The sequence includes other sequences that all together execute the following operations:

- Start condition;
- Broadcast Address with 7 bit (7'h7E);
- '0' to indicate the write operation (RW bit);
- ACK from the Targets following the recognition of the register address;
- Restart condition;
- Static Target Address with 7 bits (7'h68);
- '0' to indicate the write operation (RW bit);

- ACK from the Target following the recognition of the register address;
- writing data bytes into the Target (4 bytes);
- ACK of the Target following the correct reception of data for every data;
- Stop condition.

The timing of reference is reported in Figure 5.7 and the frame of reference is reported in Figure 5.8.

5.1.5 I2C Private Read with Broadcast Address

The I2C Read operation with Broadcast Address is executed by the *tdk_i3c_uni_I2C_Broadcast_read_test.sv*. This test invokes the sequence that is implemented in
tdk_i3c_uni_I2C_controller_Broadcast_read_sequence.sv.

The sequence file code implements different sequences that together execute the following operations:

- Start condition;
- Broadcast address with 7 bits (7'h7E);
- '0' to indicate the write operation (RW);
- ACK from the Targets following the recognition of the register address;
- Restart condition;
- Static Target Address with 7 bits (7'h68);
- '1'to indicate the Read Operation;
- ACK from the Target to indicate the successful reading of address;
- writing one data byte to provide the address of the Register File;
- reading of data bytes from Controller;
- ACK for every data byte to indicate the successful reading by the Controller;
- Stop condition.

The timing of reference is reported in Figure 5.9 and the frame of reference is reported in Figure 5.10.

5.1.6 I3C Private Write

The Write operation in SDR mode is executed by the *tdk_i3c_uni_SDR_write_test.sv*. This test invokes the sequence implemented in the file code *tdk_i3c_uni_SDR_controller_write_sequence.sv*. The sequence file code put together other sub-sequences that execute the following operations:

- Start condition;
- Broadcast Address with 7 bits (7'h7e);
- '0' to indicate the write operation (RW);
- ACK from the Targets following the recognition of the register address;
- Restart condition;
- Static Target Address with 7 bits (7'h68);
- '0' to indicate the write operation (RW);
- ACK from the Target following the recognition of the register address;
- writing data bytes into the Target;
- Transition Bit following each data byte (T-bit);
- Stop condition.

The timing of reference is reported in Figure 5.11 and the frame of reference is shown in Figure 5.12.

5.1.7 I3C Private Read

The Read operation in SDR mode is implemented in the *tdk_i3c_uni_SDR_read_test.sv*. This test invokes the read sequence *tdk_i3c_uni_SDR_controller_read_sequence.sv* that put together other sequences.

The sequence file code executes the following operations:

- Start condition;
- Broadcast Address with 7 bits (7'h7e);
- '0' to indicate the write operation (RW);
- ACK from the Targets following the recognition of the register address;

- Restart condition;
- Static Target Address with 7 bits (7'h68);
- '1' to indicate the read operation (RW);
- ACK from the Target following the recognition of the register address;
- writing data bytes into the Target to indicate the address of Register File;
- reading data bytes;
- Transition Bit following each data byte (T-bit) that represents the parity bit;
- Stop condition.

The timing of reference is reported in Figure 5.13 and the frame of reference is shown in Figure 5.14.

5.1.8 Dynamic Address assignment: I3C ENTDAA Procedure

The test
tdk_i3c_uni_SDR_dynamic_address_test.sv invokes the sequence
tdk_i3c_uni_SDR_controller_dynamic_address_sequence.sv that implements through other sub-sequences the Dynamic Address assignment using the CCC command **ENTDAA** that is represented by the value 8'h07.

The sequence file code executes the following operations:

- Start condition;
- Broadcast Address with 7 bits (7'h7E);
- '0' to indicate the write operation (RW);
- ACK from the Targets following the recognition of the register address;
- I3C Modal Broadcast CCC: ENTDAA (8'h07);
- Transition bit that represents the parity bit of the Address;
- Restart condition;
- I3C Broadcast Address with 7 bits (7'h7E);
- '1' to indicate the Read operation in order to read the following 64 bits;

- ACK from the Targets following the recognition of the register address;
- Read Data 8 bytes: 48-bit Unique ID, BCR, DCR;
- Assign 7-bit Dynamic Address to the Target;
- Parity bit that is the XOR of '1' with the bits of the Dynamic Address;
- ACK without Handoff;
- Repeated Start;
- I3C Broadcast Address with 7 bits (7'h7E);
- '1' to indicate the Read Operation;
- NACK without Handoff;
- Stop condition.

The reference frame is reported in Figure 5.16 and the reference timing is shown in Figure 5.15 .

5.2 Results and analysis

In this section, the results of the tests conducted in the previous section are presented. It is unfortunate to note that due to internal company problems related to internal deadlines, the tests involving reading could not be fully tested. However, the test environment is properly created for reading operations as well. The simulations that focused on writing operations, especially the compatibility with the I2C protocol, generated correct results. These tests are successfully executed, and the expected results are obtained. This indicates that the verification environment developed for the I3C Target Device is capable of validating the write functionality and ensuring compatibility with the I2C protocol. In addition to the complete results related to the write operation, partial results were also analyzed to verify a specific aspect of the I3C protocol. Despite the limitations and the inability to fully test the read operation, these partial results focus on a specific aspect of the protocol to gain some insights.

The partial results obtained indicate that the verification environment is capable of handling the selected aspect of the I3C protocol correctly. Although it provides only a limited scope of validation, it demonstrates its potential to accurately verify the I3CS IP Target design, provided that the necessary testing of the read operation is carried out successfully in the future.

5.2.1 Simulation Results

The proposed architecture I3CS IP is simulated and verified for accuracy and functionality using **Cadence**. Naturally, testing procedure is based on the System Verilog code.
In the investigation of the results, the focus was primarily on evaluating the correctness of the behavior of the main driven signals, namely SDA and SCL, to ensure they comply with the protocol specifications. These signals play a crucial role in the communication between the I3C Target device and the Controller.

Furthermore, the correctness of the data reported by the VIP was also examined. The VIP is responsible for capturing and reporting the data exchanged between the I3C Target device and the Controller. The reported data was compared with the expected data sent by the Register File, which represents the reference model of the expected behavior.

To facilitate this comparison, a Scoreboard component is used. The Scoreboard checks whether the reported data matches the expected data, ensuring that the communication between the Target device and the Controller is working correctly. Any discrepancies or mismatches detected by the Scoreboard indicate potential errors or inconsistencies in the implementation.

The wave-forms provided in the following paragraphs offer a visual representation of the main signals involved in the write operation.

The start, stop, and repeated start conditions, crucial for initiating and terminating the communication, are highlighted in **blue**. These conditions indicate the beginning and end of a transaction and are essential for synchronization between the Controller and the Target.

The interface signals, SDA and SCL, which carry the data and synchronize the communication, are depicted in **yellow**. These signals indicate the state of the communication line at each clock cycle, enabling data transfer between the devices.

The *pad_sda_oe* signal, shown in a **darker shade of yellow**, represents the output enable signal. It controls the driving capability of the SDA line, allowing the Target to indicate its acknowledgment or intention to communicate. By observing the changes in the output enable signal, it is possible to determine when the Target is responding or initiating a communication.

The interface signals with the Register File, such as *i3c_wr_data*, *i3c_rd_data*, *i3c_wr_strb*, and *i3c_rd_strb*, are highlighted in **orange**. These signals facilitate the transfer of data between the I3C Target device and the Register File, providing a means for data storage or retrieval.

Finally, the *sdr_wr_data* signal, highlighted in **light blue**, represents the data being written by the Controller to the SDA line at each clock cycle. This signal captures the actual value of the data being transmitted and it is essential for verifying the correctness of the write operation.

By analyzing these wave-forms and observing the behavior of the various signals, it becomes possible to validate the proper functioning of the write operation and ensure that the signals conform to the expected protocol specifications.

Results of I2C Private Write with Static Target Address Test The result of the simulation I2C Private Write with Static Target Address Test provided by Cadence is shown in Figure 5.17

While in Figure 5.18, is reported the comparison provided by the Scoreboard between serial interface and Register File.

The simulation accurately represents the behavior of the I3C protocol during a write operation with I2C compatibility.

The simulation shows that when the static address of the I3C Target matches the static address of the I2C Target (7'h68), the Target switches to I2C mode. As a result, the subsequent communication follows the specifications of the I2C protocol. This can be observed in the waveform where, instead of the Transition bit that is specific to the I3C protocol, the Target sends an acknowledgement at the end of each address and each byte of data.

These results indicate that the I3C Target device successfully transitions to I2C mode and operates according to the I2C protocol when the compatibility condition is met. This demonstrates the expected behavior of the I3C protocol for write operations with I2C compatibility.

The Figure 5.18 reports additional information about the console output on Cadence and the matching of data between the Controller, the Register File, and the simulation waveform.

It is evident that the data written by the Controller during the simulation matches the data reported by the *i3c_wr_data* signal and the packet received by the Scoreboard. This confirms the correctness of the write operation and the accurate transmission of data within the verification environment.

Furthermore, the first byte of data, which is 2B, corresponds to the address of the Register File. This aligns with the expected behavior, as the address is taken from the *i3c_addr* signal and incremented by 1 for each subsequent byte.

The rising edge of the *i3c_wr_strb* signal at the end of each byte's writing indicates that the data has been successfully transferred to the Register File, triggering the appropriate events in the Register File.

Overall, these observations and the matching of data between the Controller, Register File, and simulation results reinforce the correctness and functionality of the write operation in the verification environment.

Results of I2C Private Write with Broadcast Address Test The result of the simulation I2C Private Write with Static Target Address Test provided by Cadence is shown in Figure 5.19

While in Figure 5.20, is reported the comparison provided by the Scoreboard between serial interface and Register File.

In this test case, the same data as the previous simulation is used, but with a different approach to addressing the Target device. Instead of directly sending the static address of the Target, the broadcast address (7'h7E) is sent to all Target devices on the bus first. Then, a repeated start condition is sent, followed by the static address of the Target I2C.

The simulation results Figure 5.21 show the sequence of events in detail, starting with the broadcast address, followed by the repeated start condition (*i3c_restart*), and finally the static address of the Target I2C. This sequence is in accordance with the I3C protocol specifications for addressing the Target device.

From this point onward, the behavior and results are expected to be the same as explained in the previous paragraph, as the Target device switches to I2C mode and follows the requirements of the I2C protocol. Also in this case the scoreboard reports the match between the data written by the Controller and that detected by the Register File.

Overall, the simulation demonstrates the correct handling of the broadcast address, repeated start condition, and addressing of the Target I2C, leading to the expected behavior in accordance with the I3C and I2C protocols.

Partial results In this paragraph, the results at the interface between the VIP and the DUT are presented to demonstrate the correct functioning of the driver tasks that are developed. However, due to the limitations mentioned earlier, it was not possible to fully verify their validity in communication with the Target device.

Reading operations are indeed crucial, even in simple packet transmission in SDR mode, because the SDR mode utilizes Dynamic Addressing. After sending the ENTDAA command, the Controller is expected to read 8 bytes (Provisional ID, BCR and DCR) from the Target device to obtain the Dynamic Address.

In the SDR mode of I3C, the transition bit is used at the end of each byte instead of the ACK signal that is used in the I2C mode. The transition bit serves as an indicator to the Controller that the Target device has received the byte of data successfully.

In the results at the interface between the VIP and the DUT reported in Figure 5.22, it is demonstrated that the transition bit works properly in the SDR mode. While the specific details of the results are not provided, the focus remains on validating the correct behavior of the transition bit in the SDR mode. This ensures that the SDR mode, with its use of the transition bit, is properly implemented and functioning in accordance with the I3C protocol specifications.

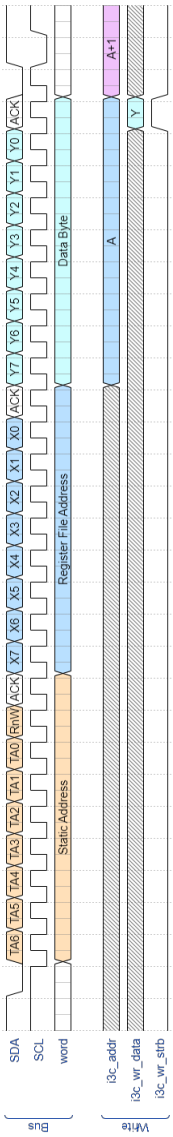

Figure 5.3: Reference timing of I2C Write Operation with Static Target Address

Figure 5.4: Reference frame of I2C Write Operation with Static Target Address

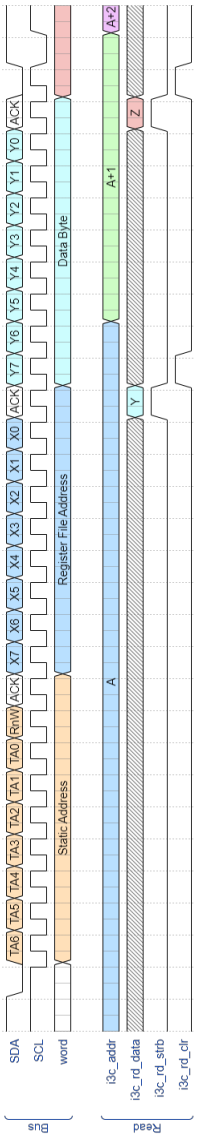

Figure 5.5: Reference timing of I2C Read Operation with Static Target Address

Figure 5.6: Reference frame of I2C Read Operation with Static Target Address

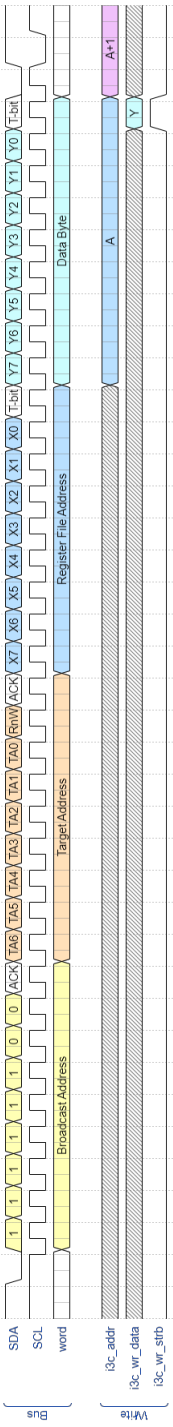

Figure 5.7: Reference timing of I2C Write Operation with Broadcast Address

Figure 5.8: Reference frame of I2C Write Operation with Broadcast Address

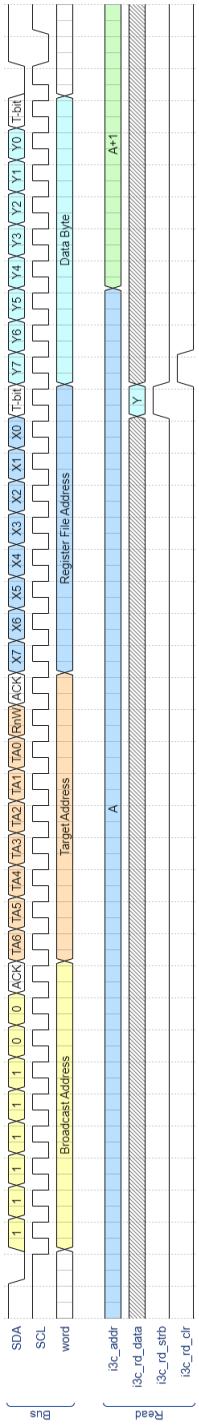

Figure 5.9: Reference timing of I2C Read Operation with Broadcast Address

Figure 5.10: Reference frame of I2C Read Operation with Broadcast Address

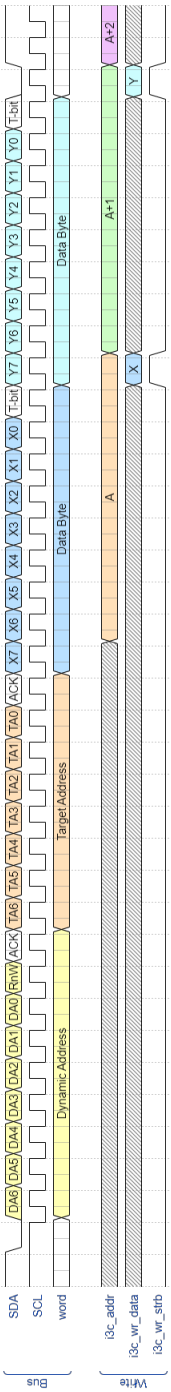

Figure 5.11: Reference timing of SDR Write Operation with Broadcast Address

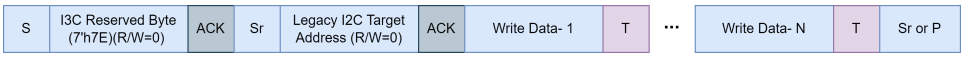

Figure 5.12: Reference frame of SDR Write Operation with Broadcast Address

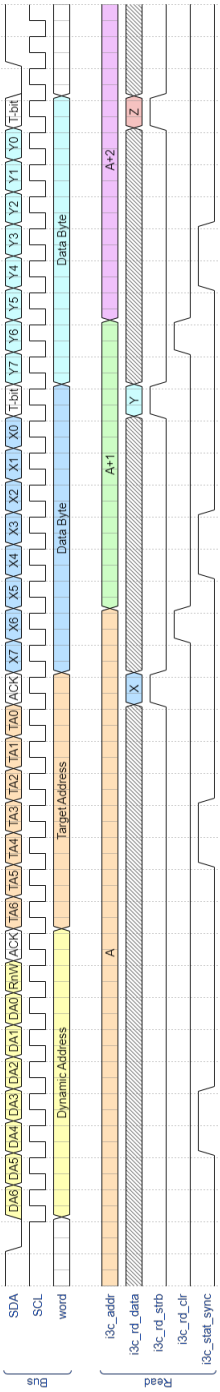

Figure 5.13: Reference timing of SDR Read Operation with Broadcast Address

Figure 5.14: Reference frame of SDR Read Operation with Broadcast Address

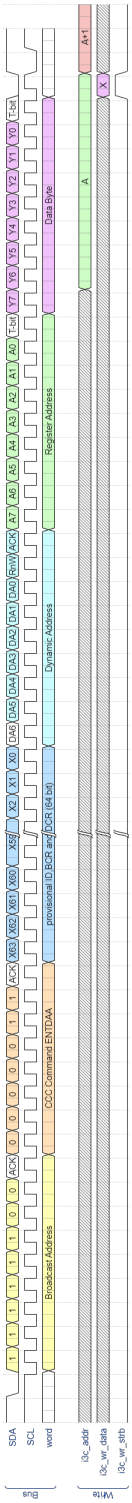

Figure 5.15: Reference timing of ENTDAA procedure

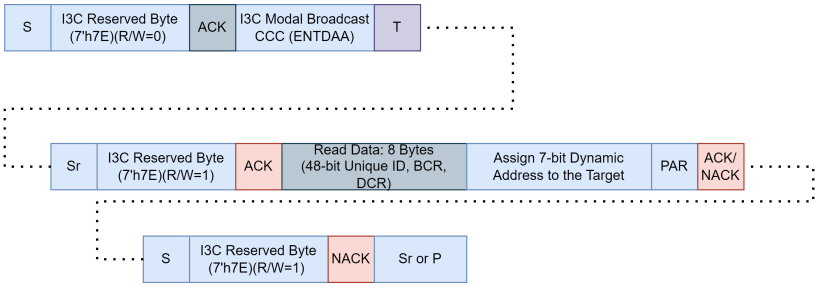

Figure 5.16: Reference frame of ENTDAA Procedure

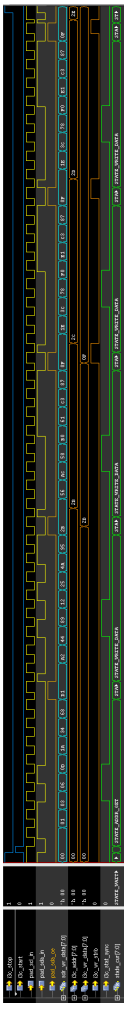

Figure 5.17: Simulation of an I2C Write Operation using Static Address

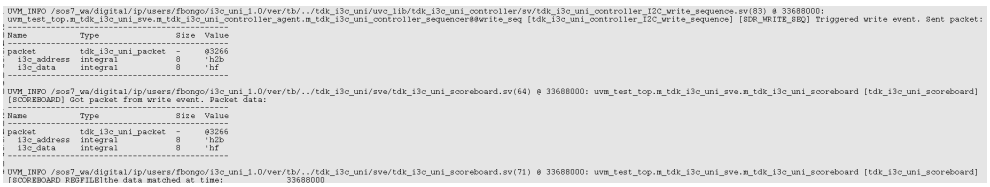

Figure 5.18: Result of the comparison provided by the Scoreboard

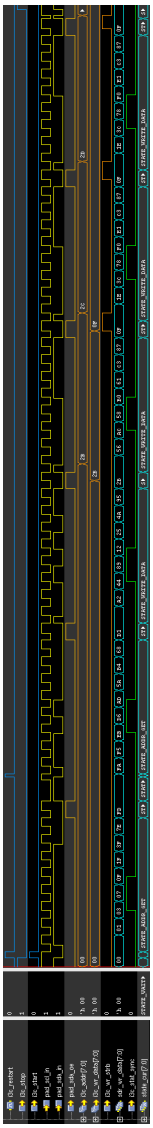

Figure 5.19: Simulation of an I2C Write Operation using Broadcast Address

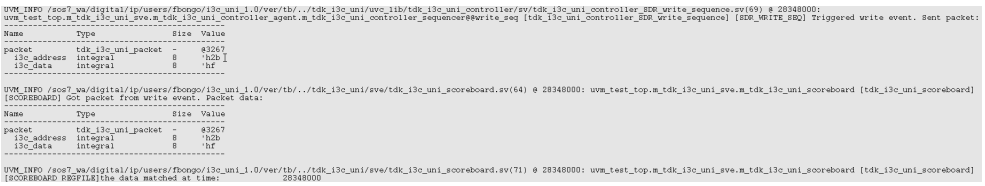

Figure 5.20: Result of the comparison provided by the Scoreboard

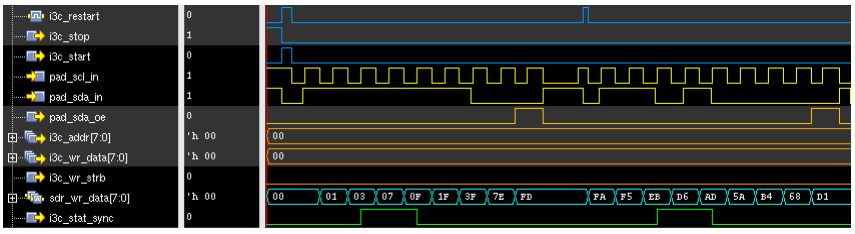

Figure 5.21: Detail of the simulation regarding the sequence Broadcast Address 7'h7E, Repeated Start and Static Address 7'h68

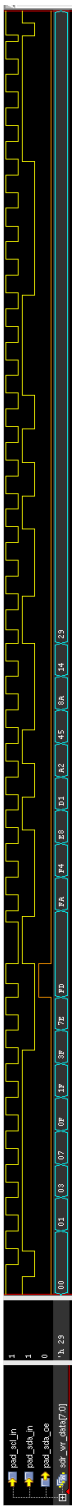

Figure 5.22: Detail of the simulation regarding the Transition Bit at the end of each Data Byte

Chapter 6

Conclusion and Future works

The thesis work focused on implementing functional tests to verify the correct operation of the Target I3CS IP. These tests were designed to detect any discrepancies or issues with the design of the Target. Although the tests were able to identify some problems, due to internal company deadlines, there was not enough time to make improvements to the design and complete the verification process.

The developed UVM environment served as a solid foundation for testing both the read and write operations of the Target. However, due to the time constraints, the read operation could not be thoroughly tested. The UVM environment itself is designed to be simple yet comprehensive, providing a good starting point for future testing.

One notable aspect of the UVM environment is the use of sub-sequences, which adds flexibility and reusability to the implementation. This allows for easy extension and improvement of the environment to test the complete design with all its associated features in the future.

While the verification process could not be fully completed within the given timeframe, the work done lays the groundwork for further testing and verification efforts to ensure the robustness and functionality of the Target I3CS IP.

In addition to verifying the Target I3CS IP, the thesis activity also offered valuable learning opportunities. It involved gaining proficiency in the usage of the UVM environment, which is a widely adopted methodology for hardware verification. This included becoming familiar with the hardware verification language System Verilog, which is commonly used in conjunction with UVM.

Working within the UVM environment required the implementation of new classes for each test scenario. These classes were responsible for managing configuration parameters, defining timing sequences to interact with the DUT, and

constructing methods to capture and analyze results for eventual comparison.

Through these tasks, the thesis activity provided hands-on experience in working with complex verification frameworks, developing test strategies, and utilizing various UVM features to achieve effective verification of the Target IP. These skills are valuable in the field of hardware design and verification and can be applied to future projects and professional endeavors.

Appendix A

Working Test Codes

A.1 tdk_i3c_uni_base_test.sv

Listing A.1: Base Test Code

```
1
2
3  `include "tdk_i3c_uni_sve.sv"
4
5
6  class tdk_i3c_uni_base_test extends uvm_test;
7        `uvm_component_utils(tdk_i3c_uni_base_test)
8
9         tdk_i3c_uni_sve m_tdk_i3c_uni_sve;
10        tdk_i3c_uni_controller_dummy_seq item;
11
12       function new(string name ="tdk_i3c_uni_base_test",
     uvm_component parent = null);
13
14       super.new (name,parent);
15       endfunction
16
17
18       virtual function void build_phase (uvm_phase phase);
19         super.build_phase(phase);
20
21     m_tdk_i3c_uni_sve = tdk_i3c_uni_sve::type_id::create ("
     m_tdk_i3c_uni_sve", this);
22
23       endfunction
24
25
26       virtual task run_phase(uvm_phase phase);
27         super.run_phase(phase);
```

```
28
29      phase.raise_objection(this);
30
31      item= tdk_i3c_uni_controller_dummy_seq::type_id::create("item");
32      item.start(m_tdk_i3c_uni_sve.m_tdk_i3c_uni_controller_agent.
        m_tdk_i3c_uni_controller_sequencer);
33
34          phase.drop_objection(this);
35        endtask
36
37  endclass
```

A.2 tdk_i3c_uni_I2C_write_test.sv

Listing A.2: Legacy I2C Write test code

```
1   `include "tdk_i3c_uni_sve.sv"
2
3
4   class tdk_i3c_uni_I2C_write_test extends uvm_test;
5
6         `uvm_component_utils(tdk_i3c_uni_I2C_write_test)
7
8          tdk_i3c_uni_sve m_tdk_i3c_uni_sve;
9
10         tdk_i3c_uni_controller_I2C_write_sequence write_seq;
11
12        function new(string name ="tdk_i3c_uni_I2C_write_test",
        uvm_component parent = null);
13
14        super.new (name,parent);
15        endfunction
16
17
18        virtual function void build_phase (uvm_phase phase);
19          super.build_phase(phase);
20
21      m_tdk_i3c_uni_sve = tdk_i3c_uni_sve::type_id::create ("
        m_tdk_i3c_uni_sve", this);
22
23        endfunction
24
25
26        virtual task run_phase(uvm_phase phase);
27          super.run_phase(phase);
28
29      phase.raise_objection(this);
```

```
30
31
32      write_seq=tdk_i3c_uni_controller_I2C_write_sequence::type_id::
        create("write_seq");
33      void'(write_seq.randomize());
34      write_seq.start(m_tdk_i3c_uni_sve.m_tdk_i3c_uni_controller_agent.
        m_tdk_i3c_uni_controller_sequencer);
35
36
37          phase.drop_objection(this);
38        endtask
39
40  endclass
```

Where the **body** task of the tdk_i3c_uni_controller_I2C_write_sequence consists of other sequences as reported below

Listing A.3: Task body code of Legacy I2C Write test

```
1       virtual task body();
2   `uvm_info(get_type_name(), $sformatf("ADDRESS: 0x%0h , DATA: 0x%0h"
      , packet.i3c_address, packet.i3c_data), UVM_HIGH)
3
4
5     `uvm_do       (start_seq)
6
7
8   `uvm_do_with (address_seq, {address_seq.address_t == 7'h68;
9               address_seq.rw_t  == RW;
10                  address_seq.address_acknowledge_t == ADDRESS_ACK;
11                  address_seq.acknowledge_t == NO_ACK;
12                     })
13
14
15    `uvm_do_with (write_seq, {
16                  write_seq.data_t          == packet.i3c_address;
17                  write_seq.is_address_ack == NO_ADDRESS_ACK;
18                  write_seq.acknowledge_t  == NO_ACK;
19                  write_seq.is_parity_bit  == NO_parity_bit;
20                    })
21
22    `uvm_do_with (write_seq, {
23                    write_seq.data_t         == packet.i3c_data;
24                    write_seq.is_address_ack == ADDRESS_ACK;
25                    write_seq.acknowledge_t == NO_ACK;
26                  write_seq.is_parity_bit == NO_parity_bit;
27                    })
28
29    `uvm_do_with (write_seq,      {
30                    write_seq.data_t          == packet.i3c_data;
```

```
                    write_seq.is_address_ack == ADDRESS_ACK;
                    write_seq.acknowledge_t  == NO_ACK;
                   write_seq.is_parity_bit   == NO_parity_bit;

                    })

    `uvm_do_with (write_seq,      {
                    write_seq.data_t         == packet.i3c_data;
                    write_seq.is_address_ack == ADDRESS_ACK;
                    write_seq.acknowledge_t == NO_ACK;
                   write_seq.is_parity_bit == NO_parity_bit;
                    })

    `uvm_do       (stop_seq)



    write_ev = uvm_event_pool::get_global("write_ev");
    write_ev.trigger(packet);
    `uvm_info(get_type_name(), $sformatf("[SDR_WRITE_SEQ] Triggered
    write event. Sent packet:\n%s", packet.sprint()), UVM_HIGH)


  endtask
```

A.3 tdk_i3c_uni_I2C_write_bd_test.sv

Listing A.4: Legacy I2C Write test code with Broadcast Address

```
`include "tdk_i3c_uni_sve.sv"


class tdk_i3c_uni_I2C_write_bd_test extends uvm_test;

      `uvm_component_utils(tdk_i3c_uni_I2C_write_bd_test)

       tdk_i3c_uni_sve m_tdk_i3c_uni_sve;

       tdk_i3c_uni_controller_I2C_write_bd_sequence write_seq;

      function new(string name ="tdk_i3c_uni_I2C_write_bd_test",
    uvm_component parent = null);

      super.new (name,parent);
      endfunction
```

```
16
17
18        virtual function void build_phase (uvm_phase phase);
19          super.build_phase(phase);
20
21      m_tdk_i3c_uni_sve = tdk_i3c_uni_sve::type_id::create ("
        m_tdk_i3c_uni_sve", this);
22
23        endfunction
24
25
26        virtual task run_phase(uvm_phase phase);
27          super.run_phase(phase);
28
29      phase.raise_objection(this);
30
31
32      write_seq=tdk_i3c_uni_controller_I2C_write_bd_sequence::type_id::
        create("write_seq");
33      void'(write_seq.randomize());
34      write_seq.start(m_tdk_i3c_uni_sve.m_tdk_i3c_uni_controller_agent.
        m_tdk_i3c_uni_controller_sequencer);
35
36
37        phase.drop_objection(this);
38        endtask
39
40  endclass
```

Where the **body** task of the tdk_i3c_uni_controller_I2C_write_sequence consists of other sequences as reported below.

Listing A.5: Task body code of Legacy I2C Write test with Broadcast Address

```
1
2       virtual task body();
3     `uvm_info(get_type_name(), $sformatf("ADDRESS: 0x%0h , DATA: 0x%0h"
        , packet.i3c_address, packet.i3c_data), UVM_HIGH)
4
5
6       `uvm_do        (start_seq)
7
8       `uvm_do_with (address_seq, {
9                      address_seq.address_t == 7'h7E;
10                     address_seq.rw_t   == RW;
11                     address_seq.address_acknowledge_t == ADDRESS_ACK;
12                     address_seq.acknowledge_t == NO_ACK;
13                       })
14
15
```

```
16      `uvm_do       (restart_seq)
17
18    `uvm_do_with (address_seq, {address_seq.address_t == 7'h68;
19              address_seq.rw_t  == RW;
20                  address_seq.address_acknowledge_t == ADDRESS_ACK;
21                  address_seq.acknowledge_t == NO_ACK;
22                     })
23
24
25    `uvm_do_with (write_seq, {
26                  write_seq.data_t          == packet.i3c_address;
27                  write_seq.is_address_ack == NO_ADDRESS_ACK;
28                  write_seq.acknowledge_t   == NO_ACK;
29                  write_seq.is_parity_bit   == NO_parity_bit;
30                    })
31
32    `uvm_do_with (write_seq, {
33                  write_seq.data_t         == packet.i3c_data;
34                  write_seq.is_address_ack == ADDRESS_ACK;
35                  write_seq.acknowledge_t == NO_ACK;
36                  write_seq.is_parity_bit == NO_parity_bit;
37                    })
38
39      `uvm_do_with (write_seq,      {
40                    write_seq.data_t          == packet.i3c_data;
41                    write_seq.is_address_ack == ADDRESS_ACK;
42                    write_seq.acknowledge_t   == NO_ACK;
43                   write_seq.is_parity_bit   == NO_parity_bit;
44
45                    })
46
47      `uvm_do_with (write_seq,      {
48                    write_seq.data_t         == packet.i3c_data;
49                    write_seq.is_address_ack == ADDRESS_ACK;
50                    write_seq.acknowledge_t == NO_ACK;
51                   write_seq.is_parity_bit == NO_parity_bit;
52                    })
53
54      `uvm_do       (stop_seq)
55
56
57
58
59      write_ev = uvm_event_pool::get_global("write_ev");
60      write_ev.trigger(packet);
61      `uvm_info(get_type_name(), $sformatf("[SDR_WRITE_SEQ] Triggered
       write event. Sent packet:\n%s", packet.sprint()), UVM_HIGH)
62
63
```

```
64
65    endtask
```

Bibliography

[1] *MIPI I3C Basic Specification*. MIPI Alliance. 2021 (cit. on pp. 6, 16).

[2] Microchip. *I3C Peripheral*. Available on line. 2001. URL: https://www.microchip.com/en-us/products/microcontrollers-and-microprocessors/8-bit-mcus/core-independent-and-analog-peripherals/communication-connectivity-peripherals/i3c (cit. on p. 6).

[3] *Universal Verification Methodology (UVM) 1.2 User's Guide*. Accellera. October, 2015 (cit. on pp. 14, 48).

[4] ChipVerify. *UVM*. Available on line. URL: https://www.chipverify.com/uvm/uvm-tutorial (cit. on p. 48).

[5] Verification Academy. *UVM Basics*. Available on line. URL: https://verificationacademy.com/courses/uvm-basics (cit. on p. 48).