TÉCNICO LISBOA

Universal Verification Methodology for Power Management Unit

Márcio Éder Sequeira Soares

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. Marcelino Bicho dos Santos
Prof. Jorge Manuel Dos Santos Ribeiro Fernandes

Examination Committee

Chairperson: Prof. Teresa Maria Canavarro Menéres Mendes de Almeida
Supervisor: Prof. Marcelino Bicho dos Santos
Member of the Committee: Prof. Fernando Manuel Duarte Gonçalves

November 2022

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

To my parents...

Acknowledgments

First of all I would like to thank my supervisors Professor Marcelino Bicho dos Santos and Jorge Fernandes for their counselling and support but also for their availability during this dissertation.

This work is dedicated to my parents, Valdemar da Cruz Soares and Marcelina do Rosário Sequeira. Everything I've ever accomplished in my life is because of them. Their unconditional love, their guidance, sacrifice and commitment to family made me into the person I am today. I want to thank my brother and sister, Odracir Almeida and Maura Soares for their love and for always checking up on me during good and bad times. Thank you Irineu Justino Delgado, Anabela de Melo, Alberto Monteiro, Eunice Matos and Gustavo Almeida for embracing me throughout all these years.

I would like to thank all the people at SiliconGate for all the teachings and all the shared moments. To João Lucas Munhão, thank you for everything. I cannot express how deeply thankful I am for everything he's done for me. I thank him for his patience, for all the teachings, and above it all for being my friend. A special thank you for Tiago Moita, André Agostinho, Válter Sádio and Bruno Santos.

To all my friends, thank you for always making me feel at home. I wish a special thank you to Carlos Santos for being by my side throughout my journey at Instituto Superior Técnico. His confidence and commitment served as example and made me work hard every single day.

And last but certainly not least, to Dânia dos Reis, thank you for always being by my side and for always pushing me to be the best version of myself. Thank you for all the love, all the advice, all the comforting and, above it all, thank you for being my partner.

This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 and pAvIs, PENTA Project n. 20016.

Resumo

Atualmente, circuitos de sinais mistos existem em larga escala na indústria dos semicondutores. Validação de circuitos de sinais mistos introduz complexidade no processo de verificação, dificultando a validação funcional. A *Universal Verification Methodology* (UVM) representa a metodologia padrão para verificação de circuitos digitais e de sinais mistos. *Real Number Modelling* permite a descrição de circuitos mistos através duma linguagem de alto nível. Esta abordagem introduz limitações no processo de verificação mas constrói alicerces para a verificação orientada a cobertura e verificação funcional. A *Universal Verification Methodology* aplicada a circuitos baseados em *Real Number Modelling* potencia a criação de ambientes de verificação robustos diminuindo significativamente o tempo de simulação, antecipando a introdução no mercado. Neste trabalho, propõe-se a implementação de um ambiente UVM para teste e verificação de reguladores de tensão e uma unidade de gestão de energia no âmbito do projeto pAvIs. Este novo método de verificação é então integrado no processo de design e teste da SiliconGate.

Palavras Chave

Universal Verification Methodology, Unidade de Gestão de Energia, Verificação orientada a cobertura, teste de circuitos de sinais mistos, reguladores de tensão

Abstract

Nowadays, mixed signal applications are widespread in the semiconductor industry. Mixed signal validation adds complexity to the verification process, which difficults functional verification Universal Verification Methodology (UVM) is the current standard methodology for verifying digital and mixed-signal designs. Real Number Modelling allows the description of mixed-signal designs using a High-level verification language. This approach imposes limitations upon the verification process but builds the foundation for coverage-driven verification and functional verification. Universal Verification Methodology applied to models based on Real Number Modelling allows for robust verification environments while significantly reducing simulation time and time-to-market. In this work, a UVM testbench environment is proposed for voltage regulators and a power management unit verification, under the scope of pAvIs project. This new verification solution is integrated in the design and test flow of SiliconGate.

Keywords

Universal Verification Methodology, Power Management Unit, Coverage-driven Verification, mixed-signal testing, voltage regulators

Contents

List of Figures

List of Tables

List of Code Segments

Acronyms

ADC	Analog-to-Digital Converter
APC	Advanced Power Control
API	Application Programming Interface
CDV	Coverage-driven Verification
CP	Charge Pump
DCIS	Conference on Design of Circuits and Integrated Systems
DUT	Device Under Test
EDA	Electronic Design Automation
HB	H-bridge Regulator
HVL	High-level Verification Language
ILIM	Current Limiter
IO	Input-output
IP	Intellectual Property
JTM	Junction Temperature Measurement
LDO	Low-dropout Regulator
MCU	Microcontroller Unit
MISO	Master In Slave Out
MOSI	Master Out Slave In
MRI	Magnetic Resonance Imaging
pAvIs	Patient and Environment Aware Adaptive Intelligent Sensor Systems
PMU	Power Management Unit
RNM	Real Number Modelling
RTC	Real Time Clock

RTL	Register-transfer Level
SoC	System-on-a-chip
SPI	Serial Peripheral Interface
SV	System Verilog
TLM	Transaction-level Modeling
URG	Unified Report Generator
UVC	Universal Verification Component
UVM	Universal Verification Methodology

1

Introduction

Contents

1.1 Motivation

Advances in fabrication technology and increasing time to market pressure alongside physical effects of shrinking the process technology impose greater challenges in design and testing of System-on-a-chip (SoC)s. Modern SoCs encompass both digital and analog blocks and require pre-silicon verification to validate their integration [1]. This approach is mandatory as multiple issues can be detected at early stages, aiming for first-time right silicon, saving time and resources. Analog and digital simulation exist in different domains [2]. Analog verification is an ad-hoc complex procedure that performs computation of large data structures with no obvious signal flow pattern, therefore lacking of a standardized methodology [2]. On the other hand, digital verification is powered with versatile tools that allow for testbench automation, constrained-random stimulus generation, coverage collection and much more. To improve the accuracy of the analog model representation, Real Number Modelling (RNM) is used to quantify analog behavior in the relevant wires. Analog behavior is described as real data, where floating point numbers are used. This approach relies on a digital solver to achieve near-digital verification speeds and improves the pre-silicon verification time when compared to transistor-level simulation. By adopting this methodology, full-chip verification is facilitated for a large scale of mixed-signal designs and the methodologies applied for digital testing can be ported and adapted to fit a mixed-signal design. These behavioral models are also important as they are rapidly produced and independent of the process technology.

Innovation and frequent new projects that benefit from reusability of previously designed Intellectual Property (IP) generate situations where test environments can be easily adapted and ported to new projects. Universal Verification Methodology (UVM) provides the infrastructure to explore this methodology. UVM proves to be an improvement for traditional mixed-signal and digital verification methods as it provides constrained-random coverage-driven test environments. It allows for test automation granting high configurability, interoperability and Coverage-driven Verification (CDV). Directed tests can also be explored by UVM as it provides a robust methodology for test generation. Testbenches can be designed for reusability reducing the effort when migrating components for different projects.

Combining UVM and RNM enables high-performance mixed signal SoC verification. RNM is limited for analog verification, but excels when integrated in a verification environment based on functional and coverage-driven verification. In a mixed-signal design, one can use it to imitate the analog counterpart of the SoC which enables extensive testing and increased simulation speeds. This verification level would not be possible for a pure embedded analog design.

Patient and Environment Aware Adaptive Intelligent Sensor Systems (pAvIs) European project [3] aims to develop innovative approaches in improving the electronics and intelligent sensor systems for professional healthcare diagnosis. Magnetic Resonance Imaging (MRI), computed tomographic and ultrasound imaging are some of the examples. The objective of this effort is to improve the current one-

size fits all paradigm, to sensor-based systems capable of diagnosing diseases, monitoring or enabling the restoration of physiological functions, or treating adverse medical conditions. This approach provides greater adaptability to an individual patient and the operating environment, providing personalized diagnosis and treatment.

SiliconGate and Instituto Superior Técnico are designing a Power Management Unit (PMU) (Figure 1.1) of an intelligent sensor system for a MRI machine with mixed-signal processing IPs at its core for the pAvIs project. These sensors will be embedded in an environment with strong electromagnetic fields, which require specialized design and testing techniques to obtain a fully functional and robust integrated circuit.

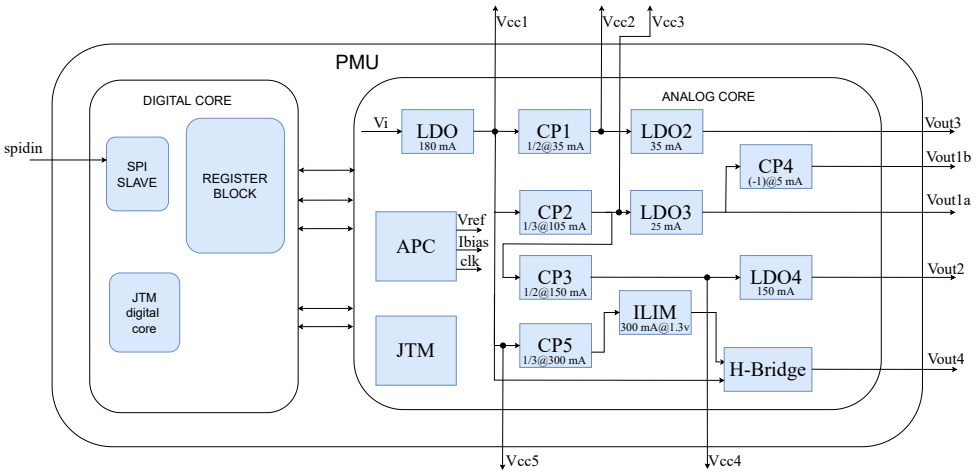


Figure 1.1: PMU architecture.

Table 1.1: PMU Specifications.

Symbol	Min	Typ	Max
V_{in}	10 V	12 V	14 V
$freq$	-	540/145 MHz	-
$out1a$	2.7 V	2.8 V	2.9 V
	0 mA	13 mA	20 mA
$out1b$	-2.9 V	-2.8 V	-2.7 V
	0 mA	1 mA	5 mA
$out2$	1.3 V	1.4 V	1.5 V
	0 mA	100 mA	150 mA
$out3$	3.2 V	3.3 V	3.4 V
	0 mA	25 mA	35 mA
$out4$[a]	-9.9 V	-11.9 V	13.9 V
	0 mA	0.1 mA	1 mA
$out4$[a]	1.3 V	2 V	2.6 V
	0 mA	160 mA	300 mA

[a]out4 output has two modes of operation.

A UVM-based architecture for a RNM model of PMU and its components is presented in this thesis using the pAvIs project as an example. This architecture is used for the validation of voltage regulators and digital core within the model and their integration to fit the specification of the PMU, described in table 1.1. The implementation, previously defined for the PMU, is presented in Figure 1.1.

1.2 Objectives and Deliverables

The purpose of this work is to implement a functional UVM environment for testing and validation of a RNM model of voltage regulators and to validate their integration in a PMU presenting, as an example, the pAvIs project.

The work developed in this thesis resulted in a paper accepted for poster presentation in the XXXVII Conference on Design of Circuits and Integrated Systems (DCIS) proceedings.

1.3 Thesis Outline

Chapter 2 introduces RNM design flow and describes the PMU architecture. RNM models are described for analog core components. Constrained-random and CDV concepts are also described. Conventional model validation approaches are assessed, showing their limitations and the advantages of adopting UVM.

Chapter 3 presents an overview of UVM and previous studies and applications. Furthermore it displays an introduction to UVM concepts and their roles in the model verification environment. Transaction-level Modeling (TLM) concepts are also introduced.

In Chapter 4, the implementation of the UVM architecture is explained for voltage regulators and digital cores. An environment to test the whole PMU is also described.

Chapter 5 shows the results and comparisons to the adopted standard System Verilog (SV) flow.

Chapter 6 presents the conclusions of this work, identifies limitations of UVM and lists encountered problems. Suggestions for future work on the matter are outlined.

2

PMU architecture

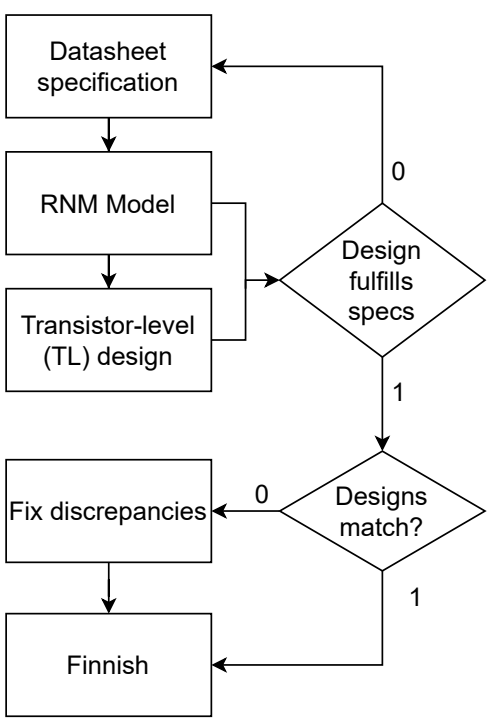
Contents

2.1 Analog SV-RNM

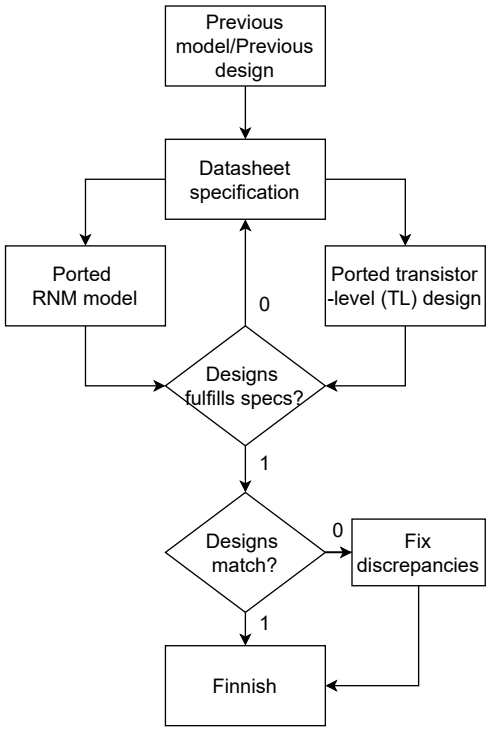
When designing mixed-signal circuits it is essential to perform a translation from the specification to the real transistor-level implementation. A verilog model of the mixed-signal circuit is also frequently designed using RNM for key parameters. SV RNM models are of extreme importance for this transition between specification and transistor-level design, as they are responsible to ensure that the design fits the specification and that the integration of digital and analog counterparts is properly implemented.

The verilog model of mixed-signal circuits can be designed in two different contexts. This first one (Figure 2.1(a)) depicts the scenario where the RNM model is built for the first time, only based on the datasheet specification. In this case the RNM model is only built to meet the specification. The analog design team develops the transistor-level circuit to be in accordance with the RNM model and, consequently, in accordance with the datasheet specification.

The second scenario (Figure 2.1(b)) is the one where the RNM models and transistor-level designs are independently ported from a previously existing project. In this scenario, design features and parameters must be updated to fit the new project's datasheet for both RNM model and transistor-level design. At the end both design must present the same behavior.



((a)) Design flow for first case.



((b)) Design flow from porting.

Figure 2.1: Design flow with RNM models.

When the transistor (or gate) level schematic already exists, the extraction of the corresponding

schematic is very useful for the verilog model design since it contains the high level structure of the design. Moreover digital cells are already automatically converted to SV logic function and need no further changes. The analog behaviour is the one to model as analog signals are described using a discrete representation, defining the analog counterpart as a signal flow event-driven model. For the scope of behavioral verification information such as supply verification and threshold values, evolution of the output voltage through an output capacitance model (when applicable) and definition of other output signals, RNM provides an accurate representation of each component functional behavior. All these features are described with SV primitives. Analog signals are driven as 64 bit floating point numbers, and converted to real numbers to perform computations ($bitstoreal and $realtobits serve as conversion tools).

As stated in section 1.1, verilog models introduce an abstraction layer in the design, loosing certain details of analog behavior and complexity. This enables extensive high-speed simulation that, when well explored, results in a more robust validation mechanism.

2.2 pAvIs PMU Architecture

A PMU is the circuit responsible for power management of an SoC. Its main roles consist of generating reference voltages, controlling power modes, battery charging, DC to DC conversion and other auxiliary functions to control the power flow. Generally, SoCs powered by a PMU require multiple voltage domains and can also be supplied from multiple power sources which define multiple requirements for operation of the PMU. The architecture of the pAvIs PMU is displayed on Figure 2.2.

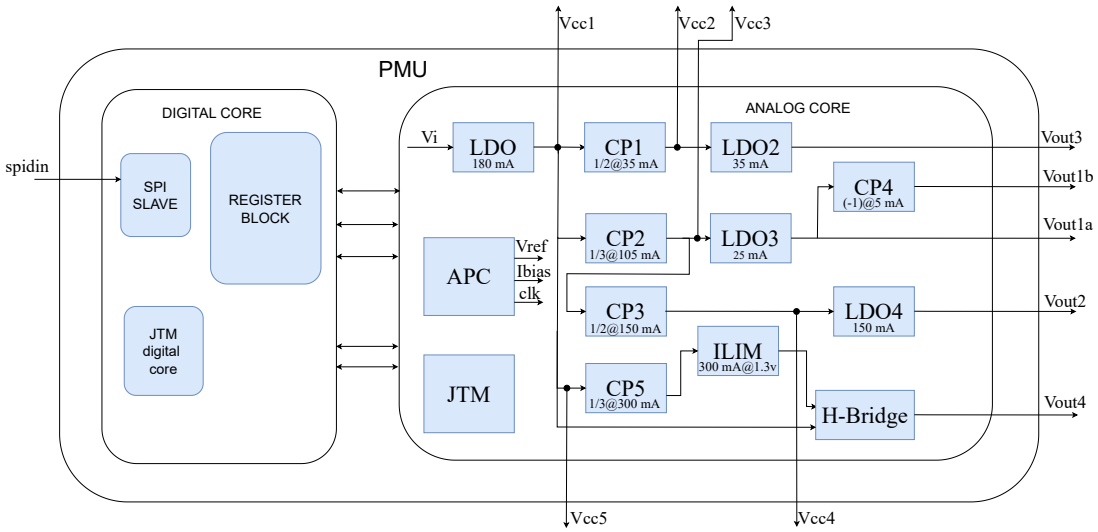


Figure 2.2: PMU architecture.

2.3 pAvIs Digital core

The pAvIs PMU possesses a digital core that encapsulates a register bank to store and configure digital control bits for the IPs inside the analog core. The digital core also receives input signals from the analog core to ensure observability and awareness of the status of the PMU internal voltages. The register bank is composed of thirteen 32-bit registers, that store configuration bits for each IP and other PMU control bits.

SPI protocol

The Serial Peripheral Interface (SPI) protocol [4] is a synchronous communication interface. It adopts a master-slave architecture, usually with one master and with one or multiple slaves. The master controls read and write operations on a register bank.

To begin communication the master defines the clock frequency, selects the slave with the chip select bit and initiates the serial communication through the Master Out Slave In (MOSI). The slave reads the bit and and sends it through the Master In Slave Out (MISO) interface resulting in a full-duplex communication. Figure 2.3 exemplifies a generic block diagram of an SPI communication protocol with one master and one slave.

The SPI protocol is used to establish the communication from the PMU interface and the top module. The PMU digital core defines a module which handles the implementation of the protocol, acting as the slave component.

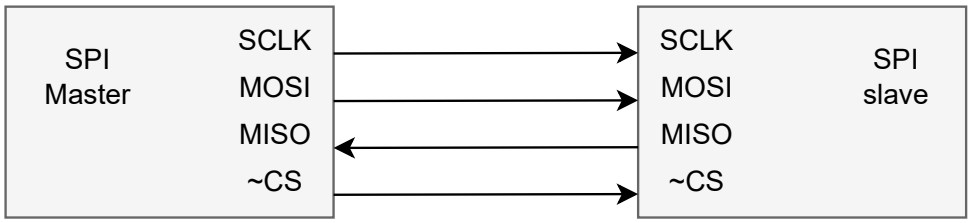


Figure 2.3: Single master to single slave SPI example communication.

The implemented SPI communication protocol for the PMU starts with the master sending an operation selector bit (read operation as logic low and write as logic high), followed by the chip select bit. Then the 6-bit address bus is driven through the MOSI interface followed by the 32-bit data bus.

2.4 pAvIs Analog core

The pAvIs analog core is composed of an Advanced Power Control (APC), four Low-dropout Regulator (LDO)s, five Charge Pump (CP)s), a Junction Temperature Measurement (JTM) regulator, a Current

Limiter (ILIM) and an H-bridge Regulator (HB).

2.4.1 APC RNM model

The APC generates control signals that coordinate the start-up sequence and overall functionality of the PMU, representing the backbone of the PMU. It generates enable signals reference voltages, bias currents and other control signals. Each regulator start-up is complete when it returns the power good (*pg*) indicator to the APC. Only after this bit is held high is the APC allowed to enable the next regulator. Generated reference voltages and currents can be digitally trimmed for increased performance.

A representative block diagram for pAvIs APC is shown in Figure 2.4.

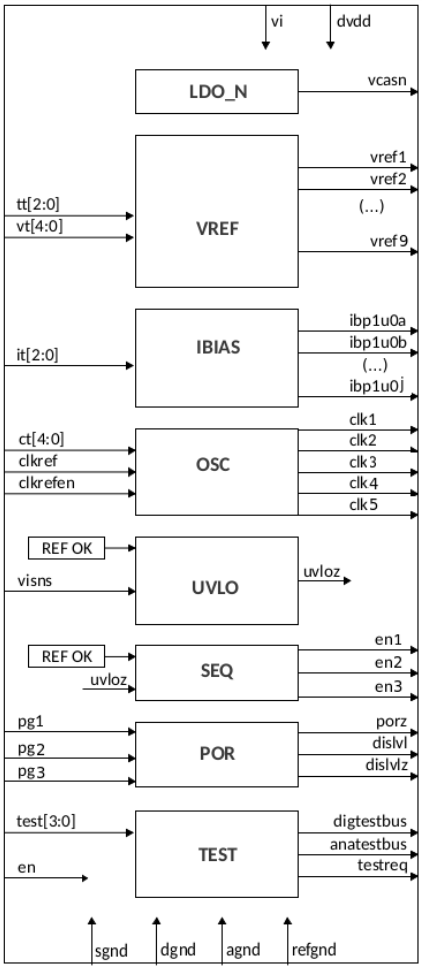


Figure 2.4: APC block diagram.

The APC model is designed as a SV behavioral Register-transfer Level (RTL) description combined with extracted views of a ring oscillator for clock generation, a bandgap circuit, and a clock division block. This inevitably results in a simplified description of the analog design. To emulate the time related with

the rising of reference voltages, debouncer circuits, or propagation of combinational logic, expected time stamps are considered, which suffices in the scope of behavioral verification. Reference voltages and currents are defined as real numbers with a 64-bit representation for each regulator with the respective trimming features. Enable output signals are generated taking into account the startup sequence, and respective power good indicators (*pg*), which culminate in the release of the *porz* (power on reset, active low) signal, which enables the system level converters. A test block is also designed with the purpose of adding controllability and observability to internal APC nets.

2.4.2 LDO RNM model

The LDO is a regulator that provides a constant output voltage, even when the supply voltage is close to the output voltage. LDO's, and voltage regulators in general, include configurable features that are controlled through a digital port. A block diagram for the LDO is displayed in Figure 2.5.

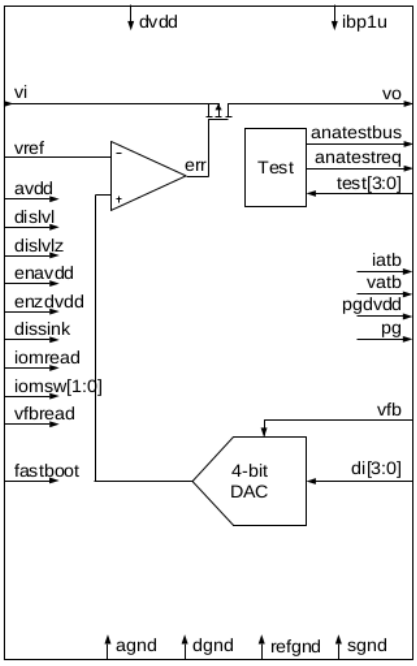


Figure 2.5: LDO block diagram.

The LDO model is a pure behavioral RTL design. The definition of the programmable output voltage is configured at the digital input di.

In table 2.1, the considered power operating modes of the LDO model are presented. The output pin *pg* is set to high when the output voltage reaches 95% of the programmed voltage. *pg* is reseted when the output voltage is lower than 90% of the programmed value.

Table 2.1: Considered power modes for the LDO.

enavdd	*enzdvdd*	*dislvl*	*pg/pgdvdd*	*power*
0	x	x	0/0	power down
1	x	1	normal/0	enabled w/def. settings
1	0	0	normal/normal	enabled
1	1	0	0/0	power down

2.4.3 CP RNM model

The charge pump is a switched capacitor circuit that regulates the output voltage using predefined ratios of the input and/or output voltage. It possesses a digital control logic that acts on switches to explore charge transfer between capacitors. A block diagram for the CP is displayed in Figure 2.6.

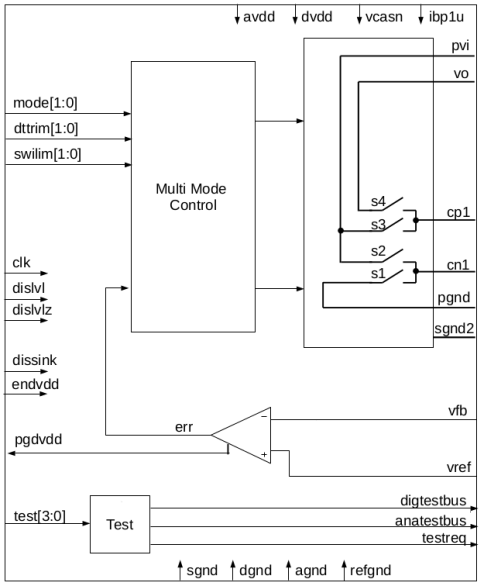


Figure 2.6: CP block diagram example.

The charge pump model top representation is extracted from the design schematic. The control block and generation of non-overlapped switching clocks consist of logic gates which are converted in SV logic representation. The power, voltage divider, and comparator blocks are replaced by simplified RNM behavioral models which implements the relevant behavior at a high level of abstraction.

The power block represents the backbone of the charge pump. For each topology, the model defines the switching configuration. For each configuration, the model explicitly computes the capacitor currents taking into account resistive losses through bond wires and switches and current limitation features. The currents serve as input of SV capacitor models (flying capacitor and output capacitor) that describe the evolution of the output voltage. The pg and pgdvdd functionality is the same as the LDO. For the different power modes *pg* and *pgdvdd* are expected to have the values represented in table 2.2. A test

block is also included in the CP.

Table 2.2: CP power modes.

endvdd	dislvl	pg/pgdvdd	power
x	1	0/0	power down
0	0	0/0	power down
1	0	normal/normal	enabled

The PMU analog core has three different charge pump topologies (see Figure 2.2). Charge pumps 2 and 5 implement a division by 3. Charge pumps 1 and 3 implement a division by 2 and charge pump 4 implements an inverter topology.

2.4.4 Remaining PMU blocks

The JTM is a high resolution Analog-to-Digital Converter (ADC) for temperature, voltage and current monitoring.

The HB specifies two modes of operation for one of the outputs of the PMU (see table 1.1). A current mode, where the model outputs a current through the pin and a voltage mode where it defines a positive or a negative voltage through the output pin.

The ILIM IP limits the current capping the output current for the HB current mode to 300 mA.

These regulators and the APC were not individually tested with UVM. Instead, they were tested with traditional SV flow.

2.5 Model validation

2.5.1 Verification techniques

The most basic SV verification environment consists of stimuli being driven to the Device Under Test (DUT) with the intention of performing a self driven verification by the engineer. The verification process relies on viewing waveforms through a wave viewer software and individually verifying every output of the design.

An improved version of this methodology explores an implementation of self-checkable processes which makes use of SV features such as the wait() statement and verification of parameter thresholds with printed messages upon failure. Currently at SiliconGate self-checkable processes are commonly used for functional verification.

Both verification approaches depend solely on the thoroughness of the verification plan and the individuals responsible for the development of the test environment. Given a certain specification for a certain IP, the test engineer generates a set of stimuli to functionally exercise the DUT.

UVM aims to consolidate the verification process, providing a robust verification environment with a well defined structure purposely built to explore extensive coverage-driven, constrained-random and self-sufficient testbenches. It possesses specialized tools to automate test generation and systematize self-checkable processes and coverage collection mechanisms.

2.5.2 SV's covergroup and coverpoint

SV defines covergroups as a user-defined type built to define the specification for a coverage model. A covergroup contains coverpoints that specify a certain statement to be covered, a set of cross coverage between coverage points, an event that defines when to sample the covergroup and other options to configure the object. When a coverpoint is defined, a set of bins is generated, which contain all possible combinations for the considered coverpoint variable. For instance, a "n" bit coverpoint variable results in 2^n automatically generated bins. It is also possible to explicitly define bins for a coverpoint. When the user crosses two coverpoints all combinations of bins for both coverpoints are generated. An example covergroup is defined in source code 2.1.

Source Code 2.1: Covergroup example.

```
covergroup cover_example;
  enable_dvdd     : coverpoint pkt.enable;
  enable_dissink  : coverpoint pkt.dissink;
  enable_mode     : coverpoint pkt.mode {
                    bins b1[]   = {[0:1]};
                  }
  cx_test         : cross enable_dvdd, enable_dissink;
  cx_test_all     : cross enable_dvdd, enable_dissink, enable_mode;
endgroup : cover_example
```

In the *cover_example* covergroup, 3 coverpoints are defined. For the enable_mode coverpoint, an explicit bin is represented, defining considered modes for the coverage model.

2.5.3 Coverage-driven constrained-random verification

Coverage-driven verification (CDV) is a verification methodology based on defining a strategy for verification. As the name implies, it is based in coverage control as building a verification plan beforehand diminishes the time needed to successfully verify a design. Figure 2.7 illustrates a coverage collection example in UVM. The monitors capture transactions from the DUT interfaces and alongside pre-defined SV coverage model, assess if all test scenarios were exercised on the DUT.

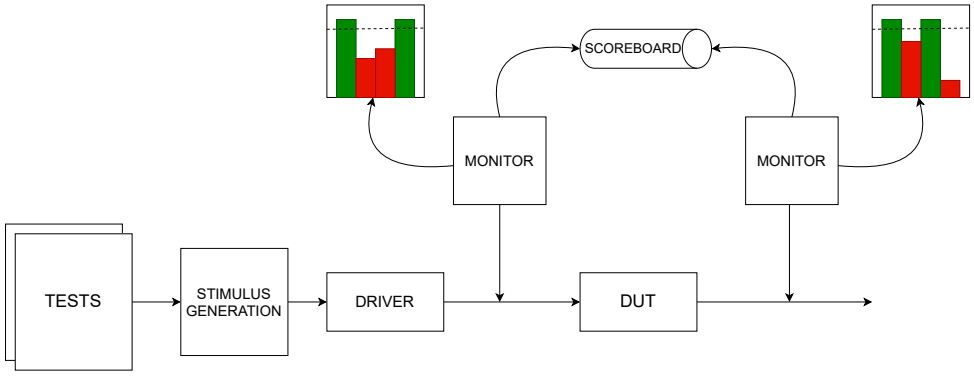


Figure 2.7: A CDV example in a UVM environment.

The goal of constrained-random is to create transactions that allow the DUT to operate in meaningful scenarios where results can be used to improve the current constraints aiming for a coverage goal and validation of the correct behavior of the DUT. CDV is useful as it thoroughly stresses the DUT and, when coupled with a scoreboard validation, results in a robust verification mechanism as all considered test scenarios are validated.

2.6 Benefits of verification with UVM

Standard SV test implementation is not the most effective, as directed tests are adopted and specifically made to validate the behaviour of design features lacking of coverage collection mechanisms and standardized ways of finding unexpected behaviour of models. UVM is introduced in chapter 3.

The verification process is not automated and porting test environments from previous projects is an exhaustive and repetitive process.

UVM provides the infrastructure to automate testbench generation, benefiting from its predefined Universal Verification Component (UVC)s. The scoreboard component alongside monitors and a reference model, validate the outputs of the DUT during run-time, generating UVM reports for miscompares and detailing the input stimuli which resulted in unexpected results. Detailed coverage reports are generated with the Unified Report Generator (URG) VCS tool, which accelerates the process of reaching coverage goals (Annex B shows an example coverage report). Some IPs are not as thoroughly tested as others, hence, this standardized building mechanism allows for conventional and robust test environments, granting the same level of verification for all DUTs.

Besides, the randomization of test sequences allows the environment to exercise the DUT with input stimuli which were not considered in the verification plan. This approach is prone to identifying defects in the model, which can be reported to the design team.

In short, UVM verification provides the following advantages when compared to traditional SV verifi-

cation:

1. **Modularity and reusability**: The methodology defines modular components enabling easier replacement policies within the components of the same type (same level of abstraction) and across projects (from single IPs to PMUs), as well as across projects.
2. **Separating tests from testbenches**: Generated sequences (stimuli set) are defined inside the encapsulated environment and can be reused across different projects.
3. **Sequence generation methodology**: This methodology provides control on how stimuli is generated. E.g., sequences can be randomized, directed, layered and included in virtual sequences.
4. **Configuration**: The UVM hierarchy is deep and well defined. The configuration mechanism provides a standardized well structured way of configuring different testbench components.

2.6.1 Verification plan

The verification plan is shown in Figure 2.8. The datasheet allows the generation of the SV model, reference model and verification plan. Based on this information the designer defines the constrained-random stimuli set and covergroups to assure that all considered test scenarios are exercised. Test sequences are generated for each input variables with aim to achieve full functional verification and 100 % code coverage. Fully pseudo-random test sequences are also described and included in the verification process. The scoreboard, reference model and monitors determine the success of the verification plan, aiming for coverage milestones and clean scoreboard UVM reports.

2.6.2 Used tools

Accelera provides an Application Programming Interface (API) standard for UVM and a reference implementation which constists of a class library defined with SV.

Cadence® Virtuoso™ and Synopsys® Custom Compiler™ Schematic Editors are tools that allow the design of integrated circuits capable of genererating SPICE netlists and extracting SV models based on pre-built libraries.

Synopsys® VCS™ functional verification solution is the engine used for compilation.

Synopsys® WaveView ™ is a waveform visualizer, useful for debugging purposes.

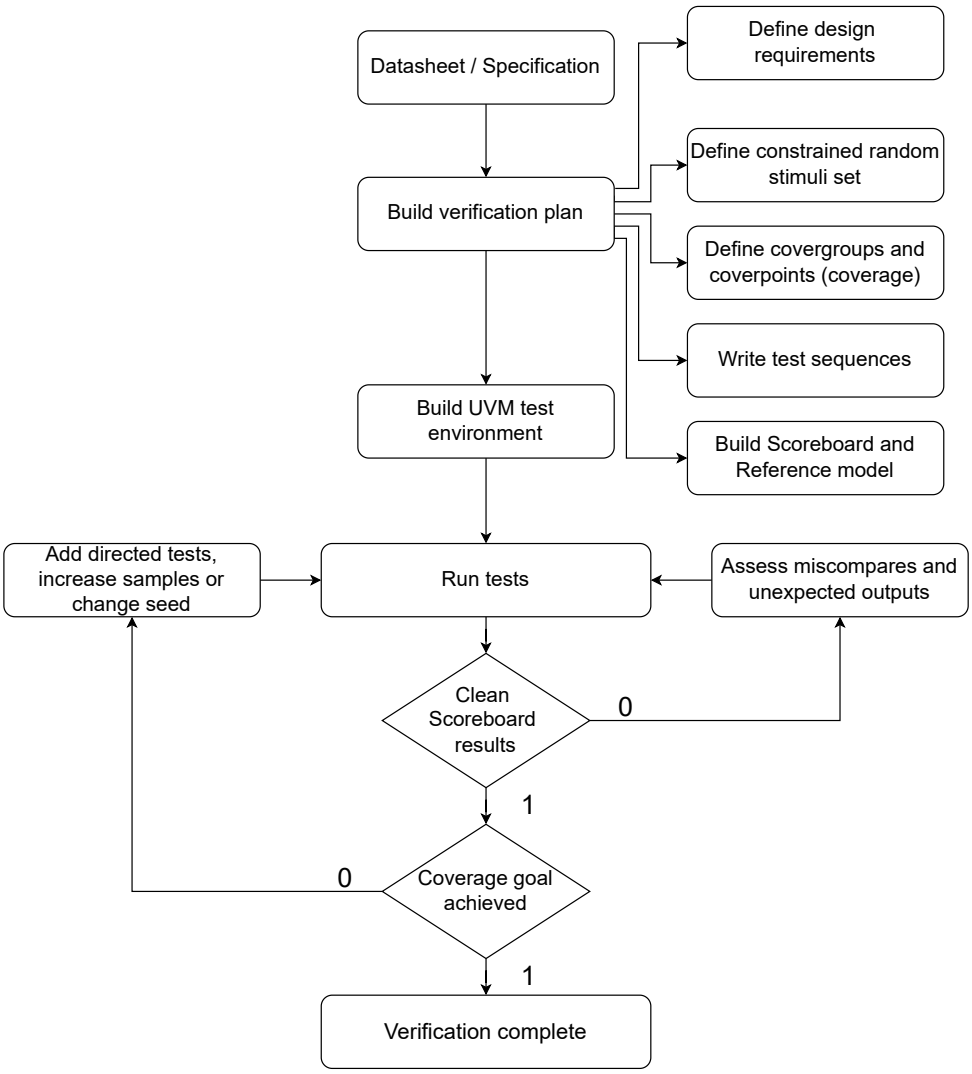


Figure 2.8: Considered UVM-based verification plan.

3

Introduction to UVM

Contents

3.1 Overview

RNM [5] allows the description of mixed-signal designs analog behaviour [6, 7], enabling signal flow event-driven models using High-level Verification Language (HVL). This approach inevitably imposes limitations, but builds the foundation for Coverage-driven Verification (CDV), [8], [9] and functional verification. Functional verification consumes a significant time of the project, [10]. Additionally, when aiming for functional verification, several test scenarios might not be considered.

UVM is the current standard methodology for verifying digital and mixed-signal designs. With its structured library classes, it allows the creation of constrained-random coverage-driven environments benefiting from SV object-oriented features, [11]. It has become the industry standard for hardware verification as it is supported by the main Electronic Design Automation (EDA) vendors and adopted worldwide for digital and mixed-signal IP testing.

An example application of UVM for a RNM design was presented for an ADC [12]. In this work, the use of UVM alongside RNM is illustrated for the validation of an ADC and its components.

UVM has been used to test the supply module of a Microcontroller Unit (MCU) inside a Real Time Clock (RTC) to enter or exit the ultra-low power modes [13] . However, in this work, only the RTC functionality was targeted with the UVM validatio, and no voltage regulation IP cores were tested. In the previous publication based on this thesis work [14], an application of UVM to test voltage regulators was implemented. The use of UVM to validate verilogAMS models was also addressed in [15] and [16]. The reliability of these approaches was also assessed [16], where advantages and disadvantages were explicitly defined.

Another relevant project explores a UVM CDV environment to test the implementation of a SPI protocol [8], as a similar approach is considered for validating the same SPI communication protocol inside the pAvIs' PMU core.

Combining RNM with UVM for mixed signal IP verification [12], [17], enables testbench automation, coverage collection and increased simulation speeds.

3.2 The systemVerilog UVM Class library

The UVM class library has several predefined classes, utilities and macros required for verification. It eases the development and reusability of verification environments [18], [19]. Components are derived from these base classes and inherit their properties, granting them great configurability and a standardized coding style.

As stated in [18], there are two great advantages when adopting the UVM class library:

- The encapsulation of important verification features such as, printing, copying, test phases, factory

methods, and more.

- The possibility to derive all components displayed in figure 3.1 from these pre-defined classes, increasing readability of the code and granting a robust and well defined hierarchical structure.

The UVM class library also provides a shared database to ease configurability (uvm_config_db), a user-controllable messaging utility for failure reporting, a standard communication infrastructure (TLM) and a flexible construction mechanism (UVM_FACTORY).

3.3 UVM testbench and environment

A typical UVM testbench architecture is displayed in Figure 3.1.

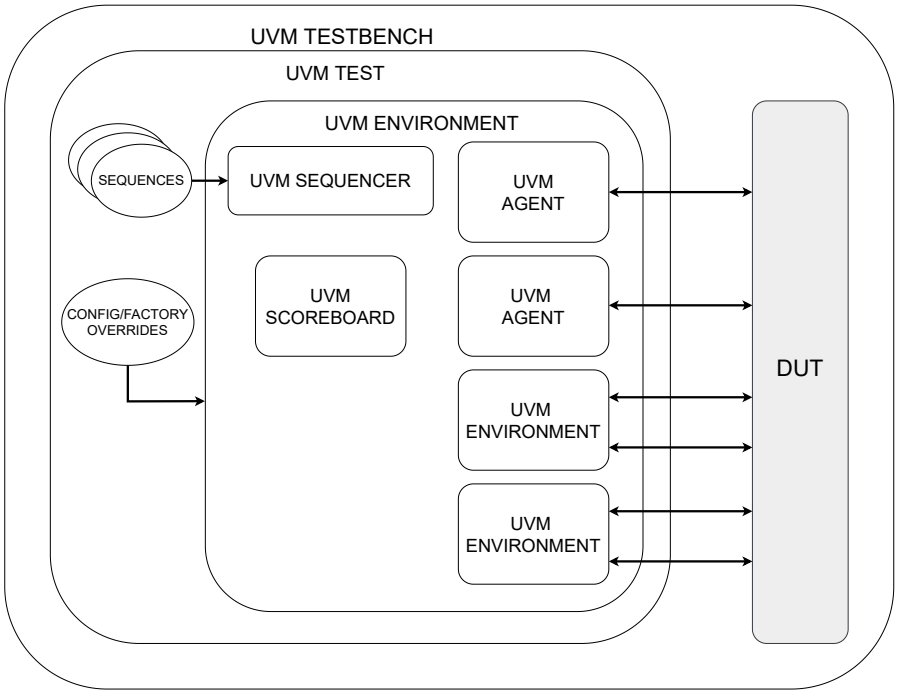


Figure 3.1: Typical UVM architecture

To briefly introduce UVM, some important concepts are explained below.

- Testbench top module - The testbench represents the top level of the hierarchy. It instantiates the UVM test environments, interfaces, DUT and other key components and coordinates the testing procedure. Annex A.1 shows examples for the considered top modules of this project.
- Test - Specifies the test scenario for the testbench. It instantiates the environment (uvm_env) and environment configuration properties [12]. An example UVM environment is shown in annex A.2. Several test environments can be instantiated for the same DUT.

- UVM environment - Derived from the base class uvm_env, it is a key building block of UVM test environment and system verification. The uvm_env provides a set of features that allow the reusability and flexibility of the environment. The environment may have multiple agents for different interfaces, a common scoreboard, a functional coverage collector and even other environments. It is responsible to integrate all the henceforth described components and coordinate sequence generation, monitoring and coverage checking of the DUT.

- UVC - A UVC represents a self contained, plug'n'play verification environment for a specific interface or a generic IP. It consists of one or more configurable agents with a predefined set of sequences that translates to test stimuli, coverage model and comprehensive failure report protocol based in UVM. Annex A.3 shows an example UVC.

- Phase - UVM controls the creation, configuration and execution of a simulation run using phases. Figure 3.2 shows a graphical representation of UVM phases. Phasing acts as a synchronization mechanism. Only after each component successfully finishes their execution in a phase, is the control flow allowed to proceed to the next phase. This creates a very structured and intuitive set of events. All phases can be grouped into three main phases: build-time phases, run-time phases and clean-up phases. The build, connect, end-of-elaboration and start-of-simulation phases are entry

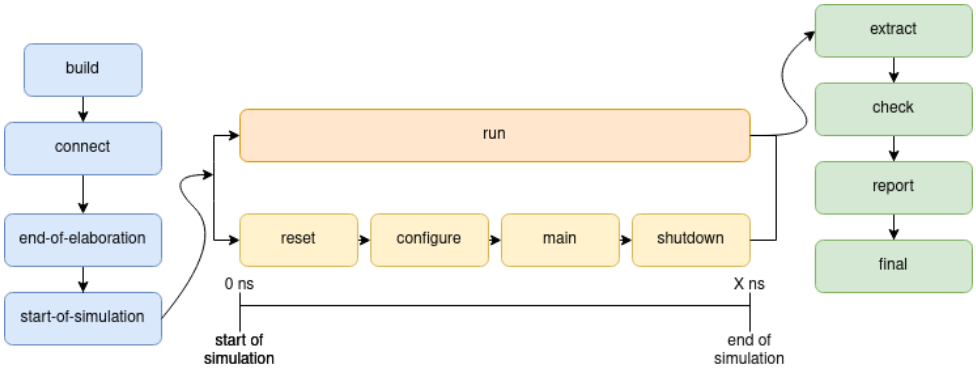


Figure 3.2: UVM phasing mechanism.

level phases that are responsible for creating, connecting and configuring the test environment. The run_phase, is the only phase that consumes simulation time, executing stimuli on the DUT. Parallel to the run-phase, UVM provides optional runtime sub-phases to give more control over stimuli during the run-phase.

The phases extract, check, report and final are exit phases which run at the end of the simulation to check scoreboards and report results.

Phasing functions have to be explicitly described under a class scope so that their execution is considered during a simulation run.

There are components in the hierarchy that concentrate their execution under specific phases, e.g. the scoreboard examines results in the check-phase and and ouputs information in the report-phase.

- Agent - The agent, derived from the uvm_agent class, encapsulates a driver, a sequencer, a monitor and a collector(when applicable). By doing so, it binds these components together to drive, monitor and collect coverage from the DUT or specific ports of the DUT. This represents an essential abstraction layer as it eases the integration of these components in the environment. An Agent can be active, if it drives signals to the DUT, or passive, if it only monitors the ports of the DUT. For this purpose the agent possesses a variable called is_active which is accessible from the UVM_environment. When set to UVM_ACTIVE the agent is set to active and is composed of a driver, sequencer and a monitor. If the variable is set to UVM_PASSIVE, the agent is set to be passive and only has a monitor. A representative block diagram for a generic active agent and its flow is displayed in Figure 3.3.

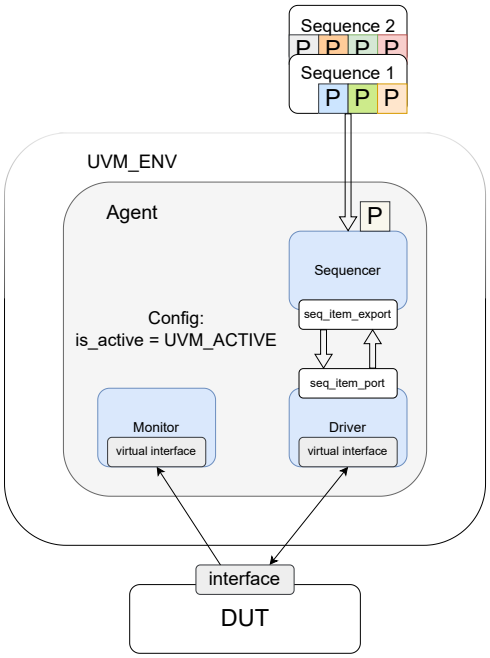


Figure 3.3: Agent block diagram.

The sequence (represented as P in Figure 3.3) exemplifies a pattern of signals (sequence-items) to be driven to the DUT. The sequencer retrieves, randomizes and sends sequence-items to the driver on demand. The driver implements the protocol to drive signals to the DUT and communicates a item_done flag to the sequencer once the data transfer is done.

The Source Code 3.1 represents an example UVM code for a generic agent.

Source Code 3.1: UVM agent example code.

```
class example_agent extends uvm_agent;
    driver    example_driver   ;//handles
    monitor   example_monitor  ;
    sequencer example_sequencer;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        monitor = output_monitor::type_id::create("monitor", this);
        if(is_active == UVM_ACTIVE) begin
            driver = output_driver::type_id::create("driver", this);
            sequencer = output_sequencer::type_id::create("sequencer", this);
        end
        `uvm_info("MSG", "Agent build_phase",UVM_HIGH)
    endfunction : build_phase
    function void connect_phase(uvm_phase phase);
        if(is_active == UVM_ACTIVE)
            driver.seq_item_port.connect(sequencer.seq_item_export);
    endfunction : connect_phase
endclass
```

The connect phase function implements the TLM connection between driver and sequencer. TLM is analyzed in detail in section 3.4.

- Sequence-item - Derived from the uvm_sequence_item base class, sequence-items represent stimuli and transactions of the UVM environment. A set of attributes, constraints and methods can be defined for the sequence-item for it to fit the needs of the DUT or the transaction type and mold the data. An example sequence-item class is presented in source code 3.2.

Source Code 3.2: UVM sequence_item example source code.

```
class example_packet extends uvm_sequence_item;
    rand int                  anatestbus                          ;
    rand bit                  anatestreq                          ;
    function new (string name = "example_packet")                 ;
        super.new(name)                                           ;
    endfunction : new
    // Enable automation of the packet's fields
    `uvm_object_utils_begin(example_packet)
        `uvm_field_int       ( anatestbus       , UVM_ALL_ON      )
```

```
        `uvm_field_int       ( anatestreq       , UVM_ALL_ON      )
    `uvm_object_utils_end
    // Define packet constraints
endclass: example_packet
```

- Sequence - A sequence, derived from the uvm_sequence class, describes a bundle of transactions (sequence-items) or other sequences. To generate sequences, UVM provides a set of macros that implement the standard flow of transaction generation. These macros encapsulate a set of methods in a single call and cover different possible scenarios for sequence generation, easing the process. Sequences are type-parameterized to a sequence_item which represents the transaction to be generated by the sequencer. Annex A.7.1 depicts an example application for virtual sequences.

- Sequencer - A sequencer, derived from the uvm_sequencer class, is responsible for sequence generation and controls the items to be sent to the driver. It generates data on-demand and returns them to the driver, as one can see in Figure 3.3). The randomization process can be controlled by setting constraints in the sequence-item model. Sequencers can exist inside agents, where their scope is local for the agent, or inside test environments as virtual sequencers. A virtual sequencer has handles to other sequencers of the environment, as shown in Figure 3.4. In this case it provides a centralized sequence generation mechanism, coordinating different elements of the environment.

- Driver - The driver, derived from the uvm_driver class, is connected to the DUT via a virtual interface and drives stimuli to its ports. Similarly to the sequencer, the driver is type-parameterized to a sequence-item that possesses the information to be driven to the DUT. During the run_phase task, the driver decodes the transaction to obtain the signals and consume simulation time to forward the data. A generic example for a driver's run_phase task is displayed in source code 3.3.

Source Code 3.3: UVM driver's run_phase task.

```
task run_phase(uvm_phase phase);
    //req is a handle which represents the transaction
    seq_item_port.get_next_item(req);//pull item from the sequencer
    send_to_dut(req);//drive function
    seq_item_port.item_done();//communicate item done to the sequencer
endtask : run_phase
```

These method calls implementation (source code 3.3) is detailed in Figure 3.5. Create_item creates the sequence-item. Wait_for_grant is a blocking method which blocks execution until it receives the

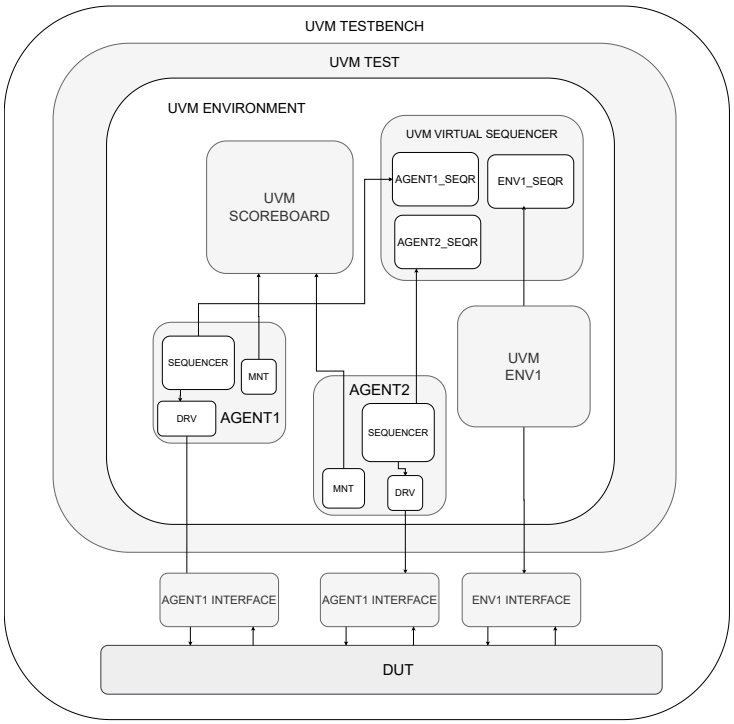


Figure 3.4: Typical UVM architecture with virtual sequencer flow.

get_next_item request from the driver. The sequence-item is then randomized and forwarded to the driver that translates to verilog synthesizable logic signals and drives them to the DUT. Finally, item_done signals the end of the transaction.

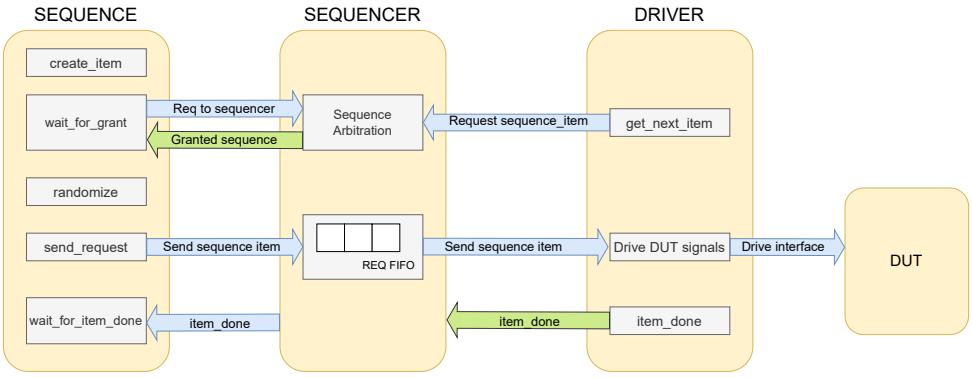


Figure 3.5: Stimulus generation and driving in the scope of UVM.

- Monitor - The monitor is a passive component in the environment. It retrieves signals from the DUT (or, in some cases, from a collector) and sends them to a scoreboard or other UVM components to perform behavioural and procedural validation. Monitors are also used to perform coverage control in order to cover several test scenarios. During the run_phase, the monitor collects data from the interface and performs housekeeping to proceed with data validation and coverage assessment.

Similarly to the driver, the connection to the DUT is made via a virtual interface.

In Figure 3.4, monitors inside active agents connect to a scoreboard component sending transactions that were previously retrieved from the virtual interface. A generic example for a monitor's run_phase task is displayed in source code 3.4.

Source Code 3.4: UVM monitor's run_phase task.

```
task run_phase(uvm_phase phase);
    collect_transaction();
    send_to_scoreboard();
    update_coverage();
endtask : run_phase
```

- Virtual Interface - The virtual interface represents an abstraction layer as it represents a pointer to an actual interface. It allows the defined classes to access the DUT ports while promoting reusability. Annex A.1.1 shows and example of a UVM top module where the connection of virtual interfaces to an actual interface are estabilished. Annex A.6.1 depicts an actual SV interface.

- Scoreboard - The scoreboard, derived from the uvm_scoreboard class, verifies if the DUT outputs the expected results. It retrieves information from monitors and, alongside a reference model, checks the correctness of the DUT outputs. The uvm_scoreboard is built with robust failure reporting features. UVM provides a set of report macros that, when correctly implemented, can extensively monitor the execution of the simulation run. This reporting mechanism takes into account severity, verbosity and simulation handling behavior, being each of them independently specified and controlled. It is intended to make the scoreboard as self-governing as possible, outputting detailed information when needed. The most commonly used macros are uvm_info, uvm_warning, uvm_error and uvm_fatal. Severity, as the name implies indicates importance, verbosity indicates filter level and simulation handling refers to the action taken by the simulator which depends on the severity being produced on the verification environment.

- Reference model - The reference model emulates the behaviour of the DUT generating the correct output for a specific input. These results are afterwards used by the scoreboard to evaluate the DUT's output.

- Factory - The UVM factory provides a standardized way of replacing an existing class by any of its inherited child classes. Upon creation, objects are registered in the factory with UVM defined macros. Once registered, the factory provides flexibility allowing the user to override certain types and instances of class objects by its child types. To enable the factory's full functionalities, the

create static method must be used on the creation of the object instance. Then the user can call one of the set_type_override macros to perform the replacement.

A typical UVM component is defined in the Source code 3.5.

Source Code 3.5: General UVM component source code.

```
class example extends uvm_component;
    `uvm_component_utils(example)//utility macro
    function new (string name, uvm_component parent);//constructor
        super.new(name,parent);
    endfunction : new
    type example_type; //handle for object creation
    function void build_phase(uvm_phase phase);//phasing
        super.build_phase(phase);
        example_type = type::type_id::create("example_type"
        , this);
        `uvm_info("MSG", "Example of how to implement a print in UVM",UVM_HIGH)
    endfunction
endclass : example
```

Classes are extended from predefined classes and inherit their features. In this example, the uvm_component_utils() macro enables automation and registers the class to the uvm_factory. All classes must have a constructor which is responsible for creating the object instance and building the hierarchical structure (new function). The user can add functions required to mold the data or collect information while performing tests.

The build_phase represents a UVM pre-defined function which is related to the phasing schema. Inside this function, there is an example of how to create a new object that complies with the uvm_factory requirements. There is also an example of a uvm_info macro that prints UVM compliant messages. The third argument of the uvm_info macro defines verbosity. Verbosity is used to filter what is printed during a run in order to only output relevant information to keep track of the execution phases or to aid in the debugging process.

3.4 Transaction-level Modelling (TLM)

UVM benefits from TLM [18], [19], as it provides an intuitive level of abstraction as stimuli is thought on the transaction-level instead of signal-level stimuli. This leads to faster simulation time as transactions simulate faster than RTL models, also benefiting from broader reusability as transactions tend to be more

flexible. Combined with UVM and its flexible, phased infrastructure, TLM promotes great interoperability between components and more coherent replacement policies.

UVM ports and exports are used to send transaction objects cross different levels of testbench hierarchy. UVM ports initiate and forward packets to upstream layers and exports accept and forward packets from top layers to destination. TLM provides a graphical notation for different types of communication (Figure 3.6). The transaction can happen on the direction producer to consumer or vice-versa. In the first case the data flow happens in the same direction of the control flow (push connection example in Figure 3.6) and in the latter, the data flows in the opposite direction of the control flow (pull connection example in Figure 3.6).

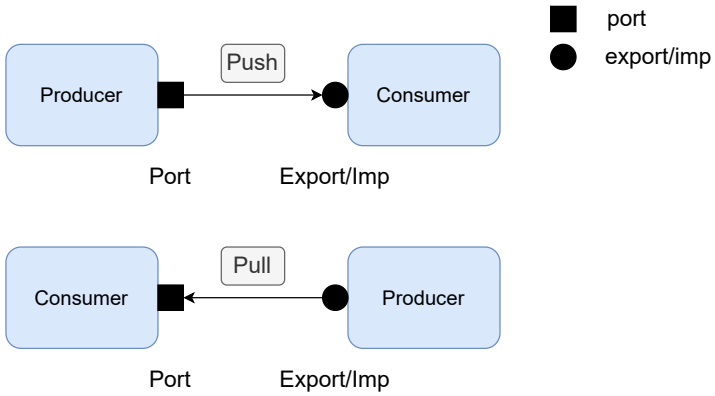


Figure 3.6: TLM graphical notations for producer and consumer

As an example, consider the Figure 3.7, that represents a canonical diagram for TLM connections. On the left hand side (Env A), ports and calls to a push connection going up the hierarchy are illustrated. On the right hand side, exports are illustrated with a push connection going down the hierarchy.

Exports differ from imps as they provide a way point in a series of TLM calls whereas the imp provides the end of the line of a TLM connection where the call will actually be implemented. This is depicted on the right hand side of Figure 3.7 (Env B). Hence, if the component implements an export, it passes the transaction to a child component. If the component has an imp it must itself provide an implementation of the correspondent task or function. Connection from environment A to B illustrate peer-to-peer connections, therefore require ports to exports.

In UVM, transactions extend from the uvm_sequence_item class. Transactions contain all the information to model a communication in the environment. Besides the information, other methods can be defined in order to to operate at the transaction level.

The connect_phase method connects the driver to the sequencer via TLM, where the driver's seq_item_port is connected to the sequencer's seq_item_export. This is shown in Figure 3.8. Seq_item_port implements a get/pull which defines the scenario where control and data flow in the opposite directions (see Figure

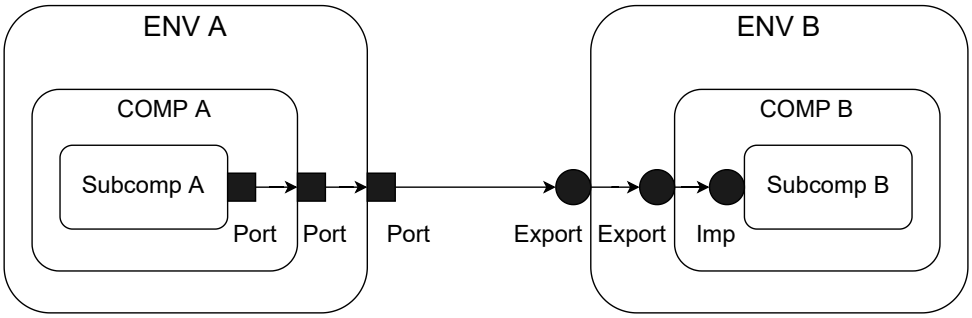


Figure 3.7: Canonical diagram for TLM connections.

3.5 where the method calls for this connection are analized in detail). This is a peer-to-peer connection as driver and sequencer exist inside the agent.

UVM also provides a uvm_analysis_port which is a specialized TLM based class that contains a list of analysis exports that are connected to it. This acts as a broadcast feature as it implements the transaction in multiples components that are connected to it. In Figure 3.8, the monitor inside the agent implements a uvm_analysys_port push/put connection defining the scenario where data and control flow in the same direction. This is a child to parent connection as the agent is the parent component of the monitor. Analysis port multiple receivers' feature is also represented as a scoreboard and a subscriber component receive the broadcasted packet from the agent. Analysis_ports do not require a receiver, and can be left unconnected.

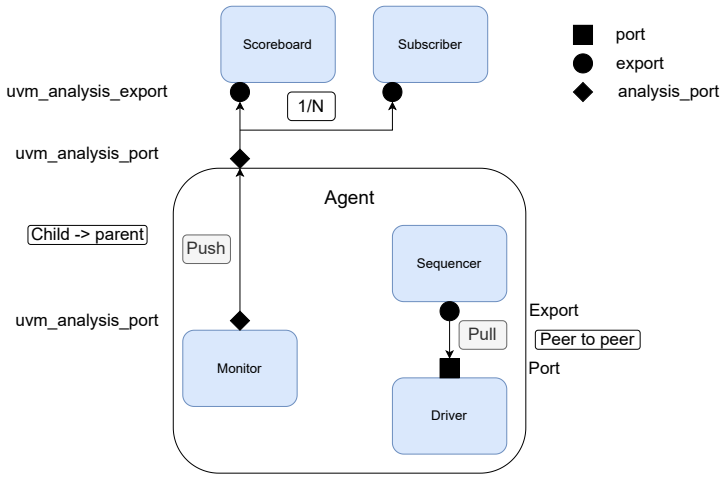


Figure 3.8: TLM connections example inside an agent.

Figure 3.9 shows multiple monitors forwarding packets through an analysis_port connected to a export/imp inside the scoreboard. The scoreboard is the final destination and consumes the transaction with a write function.

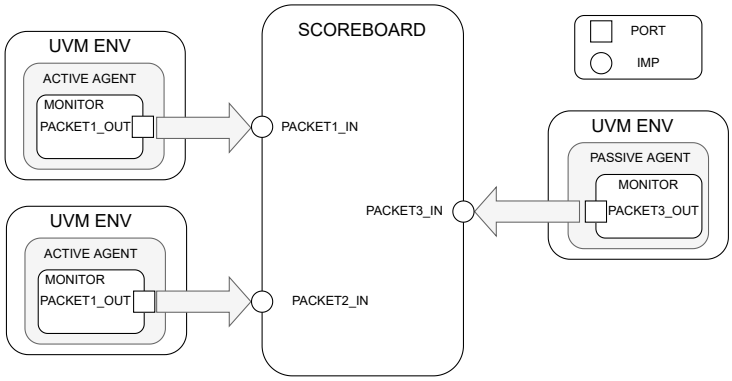


Figure 3.9: Monitor analysis port connection to scoreboard component.

4

Implementation of UVM

Contents

4.1 Reusable Universal Verification Components

First and foremost, reusable UVM enviroments are described. These will be building blocks of the IP's UVM environments. These environments are designed for reusability, aiming for validation automation. Testbenches are designed to implement a verification plan based on thoroughly stressing the DUT with directed and pseudo-random stimuli, following a set of crossed SV's coverpoints, with scoreboard validation.

All voltage regulators have an input and output voltage pin and require an input reference voltage and a dedicated input for sensing the output voltage for feedback purposes. They also possess an enable input and digital inputs to control the output voltage. They also possess a digital interface responsible for control and configuration of the IP and an analog interface responsible for providing the current and voltage references and controlling the output voltage. As these interfaces are common to the majority of the regulators (LDOs, DCDC or switch-capacitor based) it is possible to create a parameterized reusable UVM environment, that runs a pre-defined set of tests to validate their correct implementation with respect to each regulator. Power and output interfaces also possess ports that exist in all regulators making it possible to generalize a UVM environment for each.

4.1.1 Digital UVC

Voltage regulators have a digital interface ports that are grouped into a generic digital UVC. Table 4.1 shows the digital interface signals of the components of the PMU.

Table 4.1: Common digital interface signals

Digital UVC	LDO	CP	Digital core
dis	enzdvdd	disdvdd	dis
dissink	dissink	dissink	dissink
dislvl	dislvl	dislvl	-
dislvlz	dislvlz	dislvlz	-
vprog1	di	mode	vprog1
test	test	test	test
vprog2	fastboot	swilim	vprog2
vprog3	iomread	-	vprog3
vprog4	vfbread	-	vprog4
vprog5	iomsw	dttrim	vprog5

A digital UVC is generalized and encapsulated with default test sequences for each variable (Annex A.3 shows the implementation of this UVC). The agent inside this UVC can be set to active, driving stimuli to the DUT, or passive only acting as a listener of an interface. The passive implementation will be useful when testing the PMU digital core as this UVC can be replicated to sample the output of the registers for each regulator.

The sequence-item component possesses the following signals:

- *dis* - The dis pin is used to enable and disbable the regulator.
- *dislvl* - *dislvl* disables level converters and forces the regulator to ignore all driven digital signals and assume default values. It is used to disable the interface core when digital supply is not present or before the release of power-on-reset by the APC.
- *dissink* - When held high it disables the sink resistors that perform a pull down of the output voltage, if the core is disabled.
- *vprog*$_n$ - The *vprog*$_n$ bits are connected to interface bits to program certain features of the regulator. For example, vprog can be used to define the output voltage, to perform current or voltage trimming or to define operating modes of a regulator.
- *test* - The test interface represent control pins used to force testing modes for characterization and production test. Test modes can provide controllability and observability of internal signals through the *anatestbus* output pin.

Specific LDO ports are described below.

- *fastboot* - The *fastboot* pin acts on the start-up time of the regulator. When set to high it doubles the start voltage slope to reduce the start-up time.
- *iomread* and *vfbread* - When set to high, the output load internal current mirrors and the output voltage dividers are activated. The mirrored current and divided voltage can be read from the *iatb* and *vatb* output pins, respectively.
- *iomsw* - *iomsw* acts as a ratio selector for the load current mirror.
- *di* - *di* acts as a 4 bit voltage programmer. It defines the output voltage of the regulator.

Specific CP ports are described below.

- *dttrim* - The *dttrim* pin acts on the dead time between the switching phases of the regulator.
- *swilim* - *swilim* programs the current limitation on the CP switches. Input and output current limitations are closely related to switch current limitation.
- *mode* - *mode* controls the modes of operation of the CP.

To coordinate tests, functional test sequences are developed for each regulator.

Constraints are also set inside the sequence-item class, and enables CDV for a more thorough verification. These can be called from all environments that possess a digital UVC in a plug and play fashion.

4.1.2 Power UVC

Similarly to the digital interface, all regulators have a power interface that is generalized to a generic power UVC. The agent inside this UVC is set to active as it provides supply voltages for the regulators. Different sequence-item classes are generated for each regulator and can be selected on higher levels of the test environment. By taking advantage of UVM object oriented features, the user can perform constraint layering, which allows the definition of new constraints for voltage margins and other parameters. Annex A.4 shows the implementation of this UVC.

Generic power interface sequence-item signals:

- *vref* - Reference voltage.
- *avdd* - Supply for analog circuits referenced to agnd.
- *vcasn* - Supply for analog circuits referenced to agnd.
- *dvdd* - Supply for the digital cells and level converters.
- *ibp1u0* - Bias current for internal current mirrors.
- *agnd*, *dgnd* - Analog and digital ground supply.

4.1.3 Regulator UVC

All voltage regulators have an output interface that is also generalized and encapsulated inside a regulator UVC. This UVC is used for the validation of voltage regulator and digital core (see table 4.2).

Table 4.2: Common output interface signals

Regulator UVC	LDO	CP	Digital core
vo	vo	vo	vo
pg	pg	pg	pg
pgdvdd	pgdvdd	pgdvdd	pgdvdd
anatestbus	anatestbus	anatestbus	anatestbus
anatestreq	anatestreq	anatestreq	anatestreq
rl_vprog1	iatb	cp1	rl_vprog1
rl_vprog2	vatb	cn1	rl_vprog2
rl_vprog3	-	cp2	rl_vprog3
rl_vprog4	-	cn2	rl_vprog4

The correspondent agent is configured as active in the first case, and set to passive in the latter where test sequences are generated. The sequence-item component possesses the following signals:

- *vo* - Regulated output voltage.

- *pg* - Power good indicator. It is set to high when the output voltage reaches 95% of the programmed voltage.
- *pgdvdd* - *pg* converted to digital domain.
- *anatestbus* - Output test bus.
- *anatestreq* - Output bit that is held high in test modes.
- *rl_vprog*$_n$ - Adaptable output real signal.

Specific LDO ports are described below.

- *iatb* - Output for internal current mirrors.
- *vatb* - Output for internal voltage dividers.

Specific CP ports are described below.

- *cp1* - Flying capacitor positive pin.
- *cn1* - Flying capacitor negative pin.

4.2 Voltage regulators' UVM environment

The proposed architecture for a generic voltage regulator UVM environment is displayed in Figure 4.1. This environment will be used to validate the CP and the LDO.

To provide reusability, separate UVM environments to test the digital and power ports are designed and encapsulated. These can be instantiated in other UVM voltage regulator environments. Alongside these environments, a configuration environment is also considered to generate and drive specific ports, define load variables and other parameters.

The testbench has four agents, one for each environment. The agents inside the power, configuration and digital environment are set to active and drive transactions to the DUT. The agent inside the regulator environment is set to passive and only samples the output interface of the DUT.

A virtual sequencer coordinates the generation of the constrained random stimuli for the sequencers on the active agents.

During run_phase, input packets are driven to the DUT and the output port is sampled for each bundle of input signals. Each combination of input and corespondent output are orderly saved locally in queues as these will later be used to generate a reference model for comparison purposes.

Analog signals, inside the power UVC (see Annex A.4.2) are declared as 32 bit integers, as UVM does not support randomization of real values. These generated integer values are henceforth converted into a 64 bit floating point number to fit the model's real signal representation.

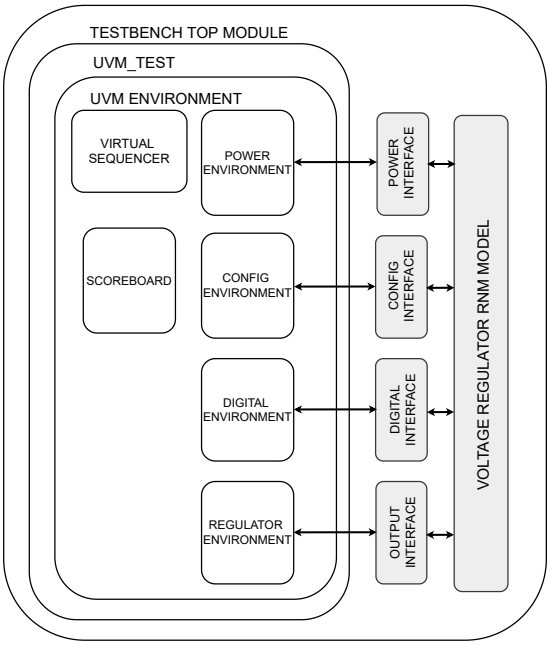


Figure 4.1: UVM environment for voltage regulator testing.

Annex A.1 show the top modules for UVM configuration and Annex A.1.2 defines the top module where the DUT and interface for each UVCs are instantiated. Annex A.2 depict the testbench UVM environment shown in figure 4.1. UVCs are declared alongside a virtual sequencer and a scoreboard component. As stated, digital and power UVCs are shown in Annexes A.3 and A.4 and run-phase calls to interface task for monitor and driver are represented. These interface calls implementation are shown in the interface source code (Annex A.6.1). The CP and LDO digital, power and output interface are shown in sections 4.1.1, 4.1.2 and 4.1.3.

4.2.1 Sequence description

For each voltage regulator, test sequences are encapsulated inside the sequences class. These sequences aim for functional verification of the voltage regulator model. Therefore directed test are considered and described for active UVCs. The digital UVC defines control signals that control the voltage regulator. The validation process results in exercising all relevant input combinations of these control bits and verify if the DUT responds accordingly. Amongst these directed oriented stimuli, pseudo-random sequences are also defined. These are only limited by the constraints explicitly defined inside the sequence-item class.

For the LDO and CP, there are independent test sequences to enable and disable the regulator, sweeps of variables such as the *test* selector bus and *di* for output voltage regulation. Once these sequences are defined, the virtual sequencer on the top level of the environment (see Figure 3.4) is

programmed to coordinate generation of stimuli for all UVC's.

This level of granularity allows the user to create complex test scenarios focusing on functional verification with directed tests, fully pseudo-random scenarios or a combination of both.

4.2.2 Scoreboards and reference models

During the check and report phases, validation metrics are generated from the UVM test environment. These contain information regarding simulation status, unmatched comparisons, coverage collection and detailed information about the test execution.

To perform validation, the queued input and output samples are sent to a reference model that generates a golden reference based on the input signals. The reference model is coded in SV. It is a representation of the data sheet acting as a look-up-table that describes the output for all input combinations. After verifying the supply, operating mode, test scenario, the programmed output voltage and other output signals it generates the correct bundle of output signals. The output is then orderly compared to this golden reference from each input packet during the check phase of the uvm_scoreboard.

After a successful comparison, these packets are removed from the queues and the next packets are assessed. Empty scoreboard queues indicates that the DUT performed as expected for all tests. A representative block diagram of this flow is displayed in Figure 4.2.

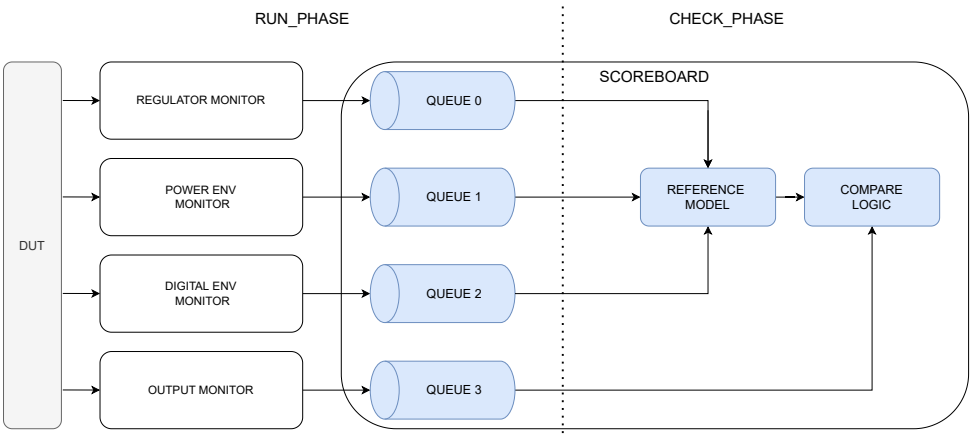


Figure 4.2: Block diagram of the validation.

4.3 Digital core UVM environment

4.3.1 Considered UVC's

4.3.1.A Digital UVC

As stated in subsection 4.1.1, the digital UVC is set to passive in this verification environment. The digital environment registers all digital control pins from all the IPs present in the analog counterpart of the design. Hence, it is straight-forward to instantiate this digital UVC for each IP, as the digital UVC is set to support interface signals of all IPs. As the agent in these UVCs are set to passive, only a monitor is present. Driver, sequencer and sequences implementation are not considered, as the goal is to sample the output of the digital core.

4.3.1.B SPI UVC

To send inputs to the digital core, a SPI UVC is designed. As displayed in Figure 2.2, the digital core possesses a *spidin* input pin which acts as as MOSI interface and a *spidout* output pin, MISO pin. It also possesses a chip select bit (*spics*) and an *spiclk* input port that serves as input clock for the SPI protocol.

To provide greater controlability on stimuli generation, the SPI sequence-item has a 32 bit data bus, a 6 bit address bus and a chip select bit. The SPI interface class implements the write and read protocols, driving the signals through the *spidin* serial input. Annex A.5 shows the implementation of the SPI UVC.

4.3.1.C JTM, APC and ILIM UVC

JTM, APC and ILIM UVCs encapsulate specific environments to test specific control bits for the current limiter IP, the JTM and the APC.

4.3.1.D General input and general output UVCs

A general input UVC and a general output UVC are also designed. They possess specific signals to the digital core that do not fit in the other UVCs encapsulation. The general input UVC encapsulates the digital core reset bit, the power on reset bit, the muxin bit and JTM digital core input configuration bits. The *muxin* allongside a *muxinaddr* 4-bit address bus define a multiplexer logic allowing the possibility to overwrite control bits with the *muxin* bit trough the digital core.

The general output UVC encapsulates the spidout serial output bit, the muxout bit, and JTM digital core output configuration bits.

4.3.1.E Regulator UVC

The analog core IPs output signals to the digital core so they can be monitored during run-time. These signals are the power good indicator (*pgdvdd*) and the test output bit (*digtestbus*). These signals exist in the regulator UVC described in subsection 4.2. Inside the digital core, these signals go through an output demultiplexer logic controlled by an 4-bit address bus (*muxoutaddr*) and an output *muxout* bit. This provides observability of these analog core output through the digital core. The regulator UVC is then set to active and instantiated once for each IP.

4.3.2 Sequence description

Test sequences for the considered active UVCs (gen_in, regulator and SPI) are randomly defined. The SPI UVC generates pseudo-random addresses(constrained for the number of addressable registers) and random data buses. Test sequences for writes and reads are implemented.

4.3.3 Scoreboard and reference model

The scoreboard implementation has a similar configuration as the one considered for the voltage regulators. During run-phase Every input and output sequence-items are inserted in queues. Output samples consist of stored digital control bits for each regulator that are sent to the analog core. The SPI databus represents the reference model. To validate writes, output samples are concatenated given an address. A mask is applied to the input SPI data bus and compared to the concatenated output bus. To validate reads the input SPI data bus is compared to the spidout serial peripheral interface. During check and report-phase, queued inputs are orderly compared with the queued outputs. After a comparison, packets are removed from the queues and the next packets are assessed. Empty scoreboard queues signal the end of the check-phase. During report-phase scoreboard statistics are displayed.

4.3.4 Proposed UVM environment architecture for the digital core

The proposed architecture is displayed in Figure 4.3.

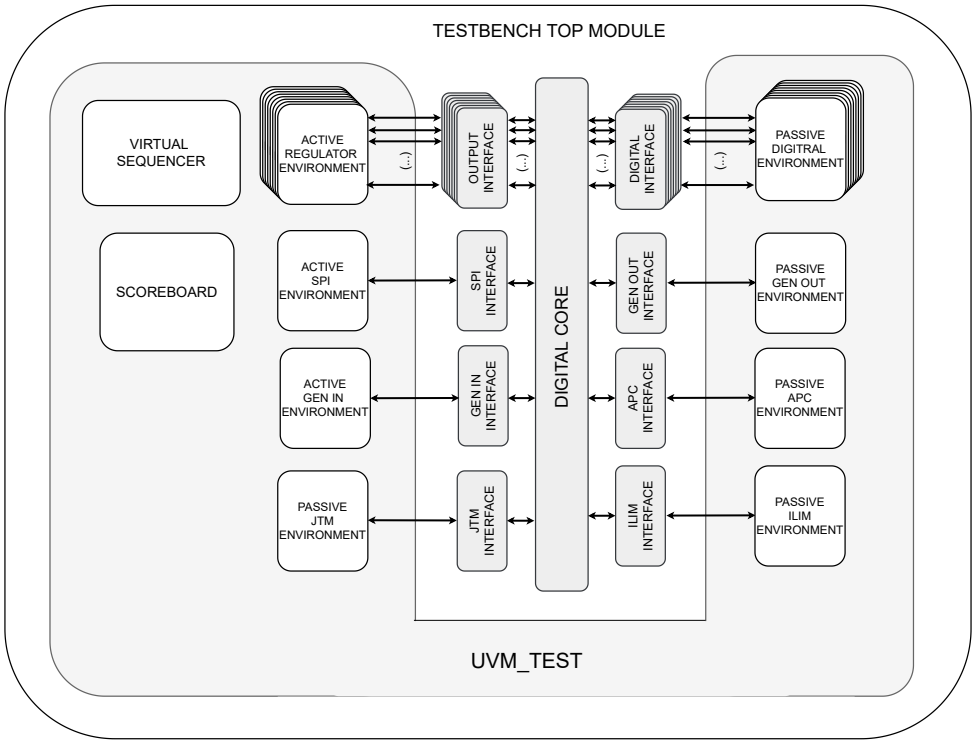


Figure 4.3: UVM environment for digital core testing.

4.4 Power management unit UVM environment

The purpose of this UVM environment is to test the start-up sequence of the PMU core and integration of the digital core with the models inside the analog core. The configuration of each validation sequence is sent through the SPI interface to program the registers and define control parameters on each regulator.

4.4.1 PMU start-up sequence

The established start-up sequence is shown in Figure 4.4. The start-up sequence is defined in order to ensure that the following conditions are met:

- Voltage regulators that define important supply domains such as digital core dvdd or Input-output (IO) voltages, power up first. This is key to the correct behavior of the PMU, as these reference voltages connect to level converters which require well defined supply domains.
- The start-up sequence is assured by an external supply. In some cases voltage regulators that generate high startup currents may lower the voltage of the external supply . Therefore the start-up of different voltage regulators is sequenced in order to limit the simultaneous current demand.

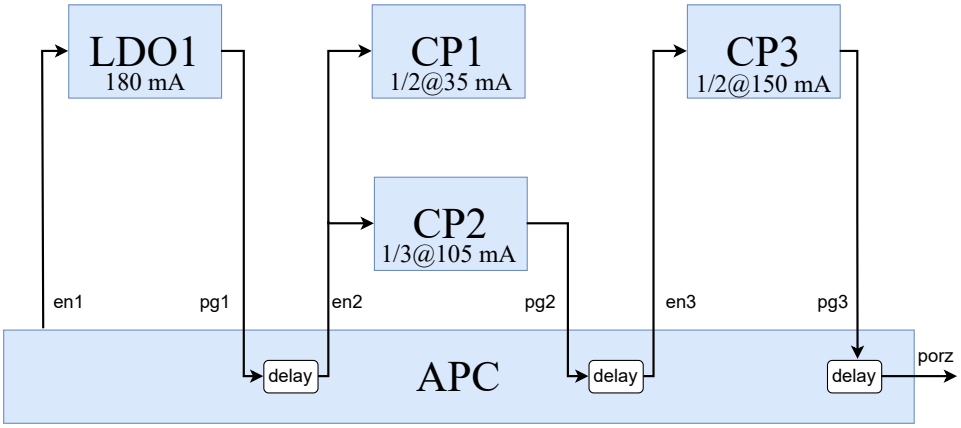


Figure 4.4: PMU's start-up sequence.

For the pAvIs PMU Voltages (see Figure 2.2) Vcc2 and Vcc4 represent the core reference voltages. LDO1 is the first regulator to be enabled as it is the first in line on the hierarchy, being connected to the input voltage (*vi*). After its *pg* indicator is released CP1 and CP2 are enabled. CP1 defines Vcc2 and CP2 generates CP3 input voltage. CP3 then defines Vcc4 which returns the last *pg* indicator. The APC then releases the *porz* indicator, which concludes the start-up sequence. The remaining regulators enable ports are connected to *dislvlz*, which represents a level converted *porz* indicator. This assures that voltage reference in all domains are defined in the right sequence.

4.4.2 Proposed environment architecture for the PMU

The proposed architecture for PMU validation is displayed in Figure 4.5.

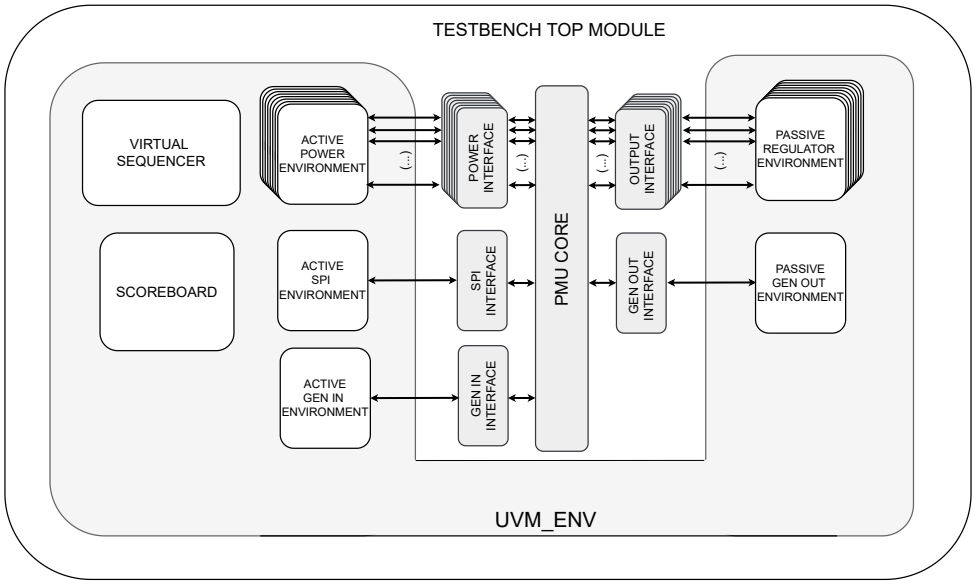


Figure 4.5: UVM environment for PMU core testing.

5

Results

Contents

5.1 Voltage regulator results

For the scope of this work, results are assessed for coverage metrics paired with scoreboard verification. Coverage reports are obtained with URG Synopsys VCS tool while the scoreboard validates the outputs given an input set.

The proposed architecture, described in section 4.2, was implemented. Directed stimuli is considered, to achieve functional coverage. Several random test scenarios are created and exercised on the DUT in order to fully cover all the possible combinations of input signals for voltage regulation. A covergroup for the digital environment was assembled. It possesses a coverpoint for each variable of the interface which were crossed to sample coverage of meaningful operating modes. The covergroup is sampled when considering the current standard SV flow and compared with the UVM flow.

5.1.1 LDO's considered coverpoints and crosses

Considered coverpoints, crosses and respective results are shown in table 5.1.

Considered crosses are explained below.

- ***cx_test_di*** - This cross is ment to test all programming codes for the output voltage (*di*) with enable, *dissink* and *dislvl* bits of the LDO. The test coverpoint is not considered.
- ***cx_test_cases*** - The test cross assures that all relevant test scenarios are considered.
- ***cx_iom_vfb*** - This crosses *iomread*, *vfbread* and *iomsw* coverpoints with a *dislvl* low coverpoint. *iatb* and *vatb* output signals are assessed for this cross.

Initially directed stimuli is considered to functionally verify the design. Therefore, a standard flow (SF) test adopted at SiliconGate is replicated in the UVM test sequences, which resulted in a coverage score of 86.58%. Crosses results are low as achieving certain input combinations require specific sequences which were not considered for functional verification. For example, a di sweep is only conducted for an enabled LDO and therefore several bins are not exercised for the *cx_test_di* cross. Unusual input combination are prone to identifying unexpected behavior. Once the several pseudo-random stimuli are introduced in the environment, 100% coverage is achieved.

For all tests, the scoreboard successfully validates the LDO RNM model behaviour.

5.1.2 CP's considered coverpoints and crosses

Considered coverpoints, crosses and initial results are shown in table 5.2. The Considered cross is explained below.

Table 5.1: Coverage results for coverpoints and crosses of the LDO.

Variable	Conditions to cover	Covered (SF)	Covered (UVM)	Cov. percentage(SF)[%]	Cov. percentage(UVM)[%]
enzdvdd (coverpoint)	2	2	2	100	100
dislvl (coverpoint)	2	1	1	100	100
dissink (coverpoint)	2	2	2	100	100
iomread (coverpoint)	2	2	2	100	100
iomsw (coverpoint)	4	4	4	100	100
vfbread (coverpoint)	2	2	2	100	100
di (coverpoint)	11	11	11	100	100
test (coverpoint)	9	9	9	100	100
cx_test_di (cross)	88	15	88	17.05	100
cx_iom_vfb (cross)	32	7	32	21.88	100
Coverage Score				86.58	100

- ***cx_test_all*** - This cross is ment to test all test scenarios for all digital interface bits. The test coverpoint is not considered.

Table 5.2: CP coverage results.

Variable	Conditions to cover	Covered (SF)	Covered (UVM)	Cov. percentage (SF)[%]	Cov. percentage (UVM)[%]
disdvdd (coverpoint)	2	2	2	100	100
dislvl (coverpoint)	2	2	2	100	100
dissink (coverpoint)	2	2	2	100	100
mode (coverpoint)	2	2	2	100	100
swilim (coverpoint)	2	2	2	100	100
dttrim (coverpoint)	4	4	4	100	100
test (coverpoint)	12	12	12	100	100
cx_test_all (cross)	32	11	32	34.38	100
Coverage Score				93.44	100

Similarly to the LDO, directed stimuli is considered to functionally verify the design, as a standard flow (SF) example testbench adopted at SiliconGate is replicated in UVM. The coverage score was 93.44%. Once again the cross presents low coverage results. Tests are conducted aiming for functional verification and unusual input combinations are not exercised as these are not considered in the verification plan. Once the several pseudo-random stimuli are introduced in the environment, 100% coverage is achieved. An issue was found in the design of the CP. *Anatestreq* output bit is set to 1 when in test mode. For a *dislvl* set to 1, level converters inside the model are expected to output default values. This means that the start-up sequence is not complete (*porz* = 0) and reference voltages in all domains are not well defined. Therefore *anatestreq* should be set to its default value of 0. This bug was corrected in the RNM model.

5.2 Digital core results

The proposed architecture, described in section 4.3, was implemented. A covergroup for the SPI environment was assembled. It possesses a coverpoint for *spics* variable and spi_address variable. Considered coverpoints, crosses and respective results are shown in table 5.3. cx_test_all crosses both presented coverpoints.

Once again the results are presented for a SV standard flow with a coverage score of 66.67%. Several random test scenarios are then created and exercised on the DUT in order to fully cover all the possible combinations of input signals for voltage regulation.

Table 5.3: Digital core coverage results

Variable	Conditions to cover	Covered (SF)	Covered (UVM)	Cov. percentage (SF)[%]	Cov. percentage (UVM)[%]
spics (coverpoint)	2	1	2	50	100
spi_address (coverpoint)	14	14	14	100	100
cx_test_all (cross)	28	14	28	50	100
Coverage Score				66.67	100

In the standard flow, tests such as writing with a high *spics* were not considered. In this scenario, the input bus should not be driven to the registers as the slave is disabled. UVM enables this tests in an extensive fashion with full coverage for the considered coverpoints. Alongside these write operations, reads are also implemented immediately after for the same address. This ensures that reads and writes are tested for all addresses for an enabled and disabled slave. The scoreboard validates the digital core for all considered inputs.

5.3 PMU core results

The PMU architecture UVM environment (Figure 4.5) is implemented. As all voltage regulators and digital core were thoroughly tested and verified, the integration and testing of the PMU is facilitated. Directed sequences are considered to test the startup sequence. For this purpose default values are written in all registers, and the APC is enabled in order to generate the enable signals for the regulators. Figure 5.1 shows the start-up sequence, depicting enable signals and their respective *pg* indicator, followed by the rise of the *porz* indicator. Voltage regulators output voltages are also shown.

5.3.1 Edge Cases

The purpose of these tests is to verify if the parameters edge cases are correctly implemented in the voltage regulators of the PMU. Therefore maximum and minimum programming codes for control bits are set to assess their impact. The LDO output voltage is programable with the *di* input digital variable.

Edge cases for *di* are assessed for LDOs that define input voltages for internal regulators (LDO1 and LDO3).

5.3.2 Edge cases results

Figure 5.2 and 5.3 displays the *vo* output voltages for both LDOs (LDO1 and LDO3) and the output voltages of the regulators connected to them.

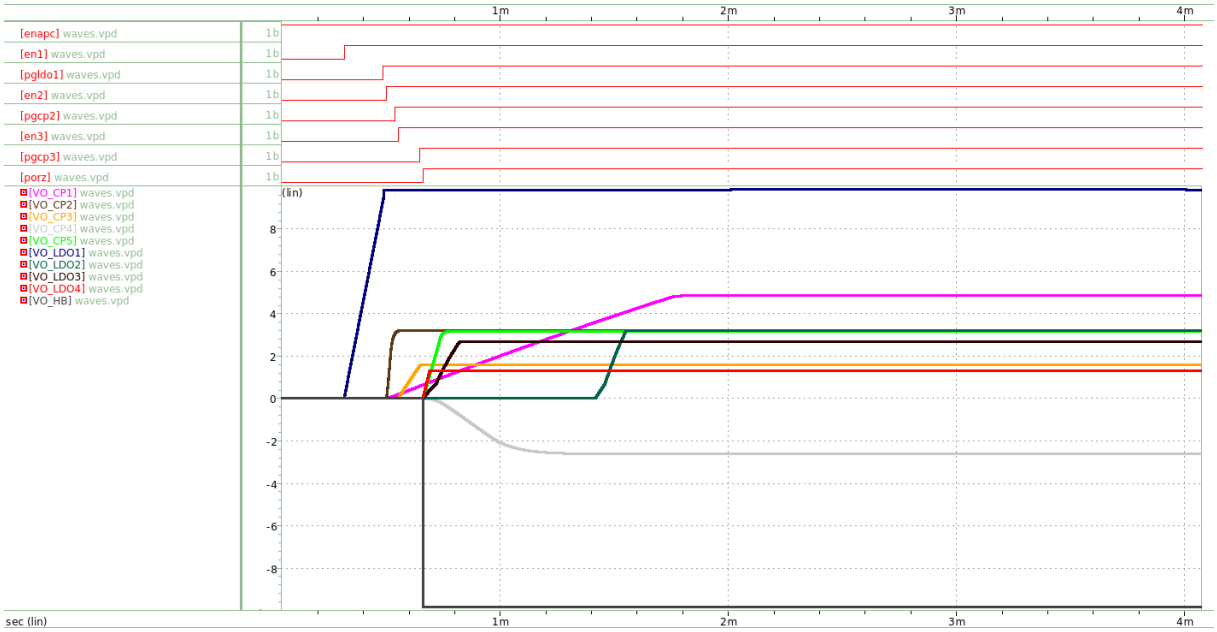


Figure 5.1: PMU's start-up sequence.

For the edge programming voltages, all subsequential regulators perceive these changes but still operate normally, as expected.

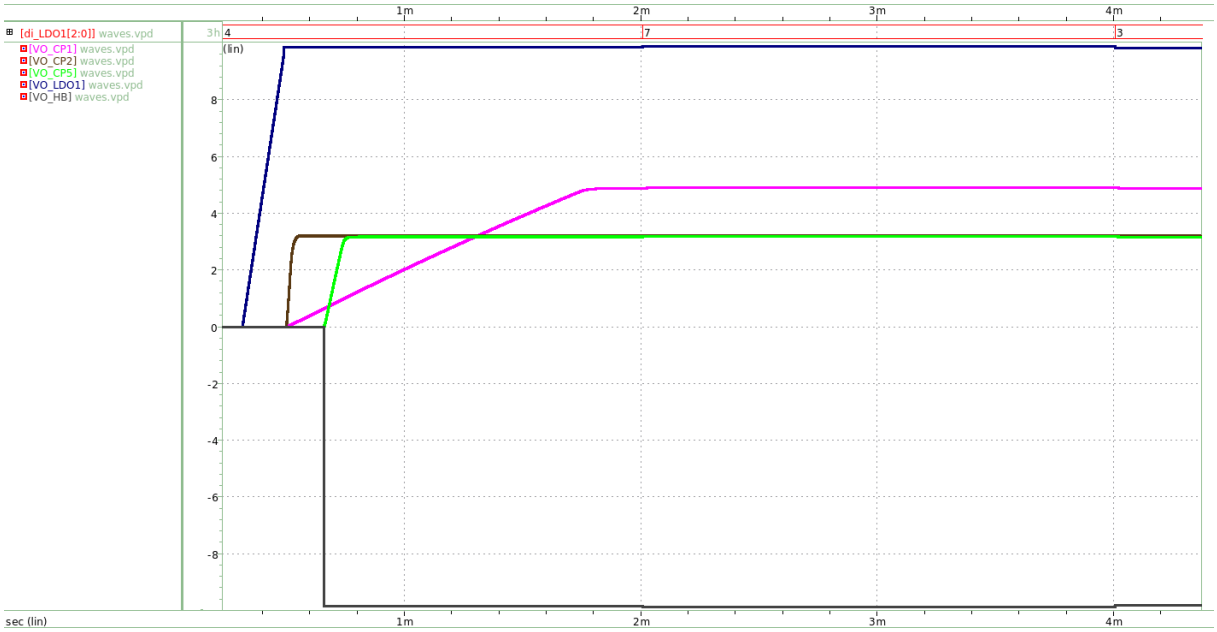


Figure 5.2: Edge cases for maximum and minimum LDO1 output voltage.

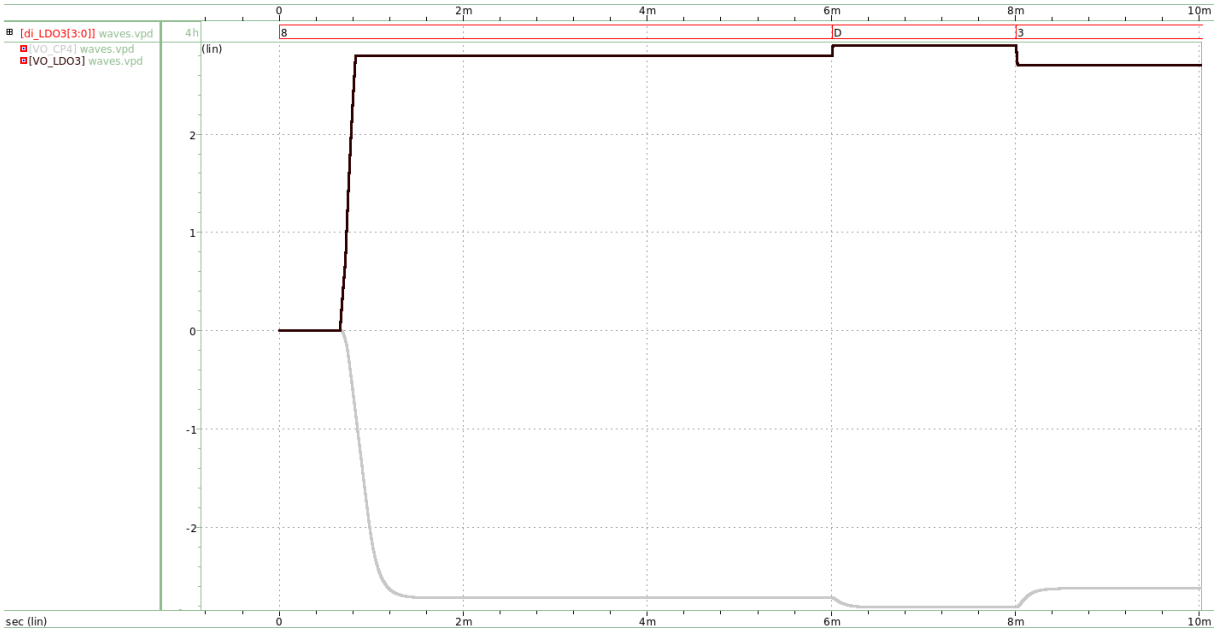


Figure 5.3: Edge cases for maximum and minimum LDO3 output voltage.

6

Conclusion

Contents

6.1 Conclusions

UVM-based mixed-signal architectures were presented for a PMU and its constituents and compared to traditional SV testing. Alongside RNM, UVM provides the infrastructure to test several scenarios in an automatic fashion while simultaneously validating the behavior of the DUT and outputting meticulous reports. Functional verification is simplified and, when aiming for coverage goals with random input stimuli, unexpected misbehavior may be detected by the scoreboard component, as shown in the CP model (section 5.1.2). This feature is much appreciated as finding these faults at earlier design stages is crucial for the success of the project. UVM was implemented in SiliconGate's test flow.

UVM UVCs were successfully reused throughout the verification process as they were intentionally built for such purpose.

Voltage regulators (mixed-signal RNM models) were validated for all considered input combinations of the digital interface ports. For the digital core, an exhaustive amount of tests were conducted. Being this design purely digital, tests simulate much faster when compared to RNM models. Coverage was also assessed to ensure that all addressable registers were exercised for random input data. The scoreboard component in these environments, alongside the reference model, play an important role as validation automation is achieved, and allows the generation of self-checkable processes. Testing of these components separately eased their integration in the scope of the PMU core. A start-up validation was performed and edge cases internal voltages were verified.

6.2 System Limitations and Future Work

The UVM class library, although very flexible and well structured, is very complex. UVM is not recommended for small projects as the overhead of building the environment is not justified by its gains. It excels for big projects where a well-planned verification environment can be implemented to speed-up the verification process. Besides, UVM does not provide a direct link from testbench sequences and code running at the SoC level [11].

One of UVM's main features is the standardized pre-built sequence generation mechanism. This allows the programmer to seamlessly implement a CDV environment. The only approach to hit coverage goals is to increase the number of samples or change the seed. A more robust sequence generation mechanism could be explored in order to reduce the occurrence of repeated sequence-items, allowing for better coverage-closure. An approach to this issue was assessed in [20].

A UVM register model will be considered as an extension of this work. The register model is a hierarchical structure of objects that contains the description of the register on the DUT. This model can store information of register fields such as address, size, reset value and access policy, that can be used to validate correct register behavior.

Master-slave topologies will be considered. This allows the user to dynamically generate UVM environments in a centralized fashion, granting an extra layer of controlability. Factory features and further configuration with a the UVM configuration database will be explored as well.

Bibliography

[1] C. Sapsanis, M. Villemur, and A. G. Andreou, "Real number modeling of a sar adc behavior using systemverilog," in *2022 18th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, 2022, pp. 1–4.

[2] S. Balasubramanian and P. Hardee, "Solutions for mixed-signal soc verification using real number models," in *Cadence, Tech. Rep.*, 2013.

[3] Penta, "pavis - penta," 2022, https://penta-eureka.eu/project-overview/penta-call-5/pavis/, Last accessed on 2022-09-16.

[4] A. N, G. Joseph, S. S. Oommen, and R. Dhanabal, "Design and implementation of a high speed serial peripheral interface," in *2014 International Conference on Advances in Electrical Engineering (ICAEE)*, 2014, pp. 1–3.

[5] M. M. Ron Vogelsong, Ahmed Hussein Osman, "Practical rnm with systemverilog," in *CDNLive2015*, 2015.

[6] N. Georgoulopoulos and A. Hatzopoulos, "Real number modeling of a flash adc using systemverilog," in *2017 Panhellenic Conference on Electronics and Telecommunications (PACET)*, 2017, pp. 1–4.

[7] N. Georgoulopoulos, A. Mekras, and A. Hatzopoulos, "Design of a systemverilog-based vco real number model," in *2019 8th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2019, pp. 1–4.

[8] B. Vineeth and B. B. Tripura Sundari, "Uvm based testbench architecture for coverage driven functional verification of spi protocol," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 2018, pp. 307–310.

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

[10] T. M. Pavithran and R. Bhakthavatchalu, “Uvm based testbench architecture for logic sub-system verification,” in *2017 International Conference on Technological Advancements in Power and Energy (TAP Energy)*, 2017, pp. 1–5.

[11] K. Salah, “A uvm-based smart functional verification platform: Concepts, pros, cons, and opportunities,” in *2014 9th International Design and Test Symposium (IDT)*, 2014, pp. 94–99.

[12] N. Georgoulopoulos, I. Giannou, and A. Hatzopoulos, “Uvm-based verification of a mixed-signal design using systemverilog,” in *2018 28th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2018, pp. 97–102.

[13] Y. Liu, N. Tan, X. Xiao, J. Xia, W. Hu, and Y. Ding, “Design and uvm verification of an rtc subsystem with temperature compensation,” in *2021 6th International Conference on Integrated Circuits and Microsystems (ICICM)*, 2021, pp. 384–389.

[14] M. Soares, M. Santos, and J. Munhão, “Universal verification methodology for voltage regulators,” accepted for poster presentation in XXXVII Conference on Design of Circuits and Integrated Systems, 2022.

[15] C. Liang, G. Zhong, S. Huang, and B. Xia, “Uvm-ams based sub-system verification of wireless power receiver soc,” in *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2014, pp. 1–3.

[16] W. Ramirez, H. Gomez, and E. Roa, “On uvm reliability in mixed-signal verification,” in *2019 IEEE 10th Latin American Symposium on Circuits & Systems (LASCAS)*, 2019, pp. 233–236.

[17] A. I. Cianga and C. Tepus, “Morphing digital functional verification to meet mixed signal challenges,” in *2014 International Semiconductor Conference (CAS)*, 2014, pp. 219–222.

[18] Accelera, “Universal verification methodology (uvm) 1.1 user’s guide,” 2011.

[19] K. M. S. Rosenberg, “A practical guide to adopting the universal verification methodology (uvm),” in *Cadence Design Systems, 2nd edition*, 2010.

[20] K. Fathy, K. Salah, and R. Guindi, “A proposed methodology to improve uvm-based test generation and coverage closure,” in *2015 10th International Design & Test Symposium (IDT)*, 2015, pp. 147–148.

A

UVM Code

A.1 Top modules

The UVM configuration top module is isolated from the hardware top module for accelaration purposes. Therefore UVM top module builds and configures the UVM environment while the hardware top module instantiates the DUT and interfaces.

A.1.1 UVM top module for voltage regulators

List of Code Segments A.1: UVM top module.

```
1  module top;
2  // import the UVM library
3      import uvm_pkg::*;
4  // include the UVM macros
5      `include "uvm_macros.svh"
6
7  // import the UVC packages
8      import cp_pkg::*;
9      import digital_pkg::*;
10     import power_pkg::*;
11     import output_pkg::*;
12
```

```
13      //Include testbench, test library file, virtual sequencer file, virtual
            sequences file and scoreboard file
14      `include "cp_mcsequencer.sv"
15      `include "cp_mcseqs.sv"
16      `include "scoreboard.sv"
17      `include "cp_tb.sv"
18      `include "cp_test_lib.sv"
19
20      initial begin
21      //Virtual interface connections to actual interfaces
22      cp_vif_config::set(
23                      null, //context is null because this is the top module
24                      "uvm_test_top.tb.cp_uvc.cp_tx_agent.*",
25                      "vif",
26                      hw_top.in0
27      );
28      output_vif_config::set(
29                      null, //context is null because this is the top module
30                      "uvm_test_top.tb.output_uvc.output_tx_agent.*",
31                      "vif",
32                      hw_top.in1
33      );
34      digital_vif_config::set(
35                      null, //context is null because this is the top module
36                      "uvm_test_top.tb.digital_uvc.digital_tx_agent.*",
37                      "vif",
38                      hw_top.in2
39      );
40      power_vif_config::set(
41                      null, //context is null because this is the top module
42                      "uvm_test_top.tb.power_uvc.power_tx_agent.*",
43                      "vif",
44                      hw_top.in3
45      );
46         run_test(); //run UVM
47      end
48  endmodule : top
```

A.1.2 Hardware top module for voltage regulators

List of Code Segments A.2: Testbench top module for charge pump.

```
1   module hw_top;
2
3     // Clock and reset signals
4     logic [31:0]  clock_period;
5     logic         run_clock;
6     logic         clock;
7
8     real         rl_iload                                          ;
9
10
11    // YAPP Interface to the DUT
12    cp_if       in0(clock);
13    output_if   in1(clock);
14    digital_if  in2(clock);
15    power_if    in3(clock);
16
17    assign in1.out_data_ready = in0.in_data_vld;
18    //  assign in2.in_data_vld = in0.in_data_vld;
19    assign in1.sample_out = in0.sample_monitor;
20
21    initial in0.in_data_vld = 0;
22
23    // CLKGEN module generates clock
24    clkgen clkgen (
25      .clock(clock),
26      .run_clock(1'b1),
27      .clock_period(32'd1000)
28    );
29
30    charge_pump dut(
31      .clk                      (       clock                   )         ,
32      .en                       (       in0.en                  )         ,
33      .vcasn                    (       in0.vcasn               )         ,
```

```
    .digtestbus             (       in1.digtestbus          )       ,
    .pgdvdd                 (       in1.pgdvdd              )       ,
    .pg                     (       in1.pg                  )       ,
    .testreq                (       in1.anatestreq          )       ,
    .anatestbus             (       in1.anatestbus          )       ,
    .cn1                    (       in1.cn1                 )       ,
    .cp1                    (       in1.cp1                 )       ,
    .vo                     (       in1.vo                  )       ,
    .vfb                    (       in1.vo                  )       ,
    .dttrim                 (       in2.vprog5              )       ,
    .disdvdd                (       in2.dis                 )       ,
    .swilim                 (       in2.vprog2              )       ,
    .dislvl                 (       in2.dislvl              )       ,
    .dislvlz                (       in2.dislvlz             )       ,
    .dissink                (       in2.dissink             )       ,
    .mode                   (       in2.vprog1              )       ,
    .test                   (       in2.test                )       ,
    .vref                   (       in3.vref                )       ,
    .dvdd                   (       in3.dvdd                )       ,
    .ibp1u                  (       in3.ibp1u               )       ,
    .avdd                   (       in3.avdd                )       ,
    .pvi                    (       in3.pvi                 )       ,
    .agnd                   (       $realtobits(0.0)        )       ,
    .dgnd                   (       $realtobits(0.0)        )       ,
    .refgnd                 (       $realtobits(0.0)        )       ,
    .pgnd                   (       $realtobits(0.0)        )       ,
    .sgnd                   (       $realtobits(0.0)        )       ,
    .sgnd2                  (       $realtobits(0.0)        )

  ) ;

endmodule
```

List of Code Segments A.3: Testbench top module for LDO.

```
module hw_top;
  // Clock and reset signals
  logic [31:0]  clock_period;
  logic         run_clock;
  logic         clock;

  ldo_if      in0(clock);
  output_if   in1(clock);
  digital_if  in2(clock);
  power_if    in3(clock);

  // CLKGEN module generates clock
  clkgen clkgen (
    .clock(clock),
    .run_clock(1'b1),
    .clock_period(32'd10)
  );

  LDO dut(
    .agnd                   (       in3.agnd                )       ,
    .avdd                   (       in3.avdd                )       ,
    .dgnd                   (       in3.dgnd                )       ,
    .dvdd                   (       in3.dvdd                )       ,
    .ibp1u                  (       in3.ibp1u               )       ,
    .sgnd                   (       in0.sgnd                )       ,
    .refgnd                 (       in0.refgnd              )       ,
    .vi                     (       in3.vi                  )       ,
    .vref                   (       in3.vref                )       ,
    .iomsw                  (       in2.vprog5              )       ,
    .iomread                (       in2.vprog3              )       ,
    .vfbread                (       in2.vprog4              )       ,
    .vfb                    (       in1.vo                  )       ,
    .dislvl                 (       in2.dislvl              )       ,
    .dislvlz                (       in2.dislvlz             )       ,
    .dissink                (       in2.dissink             )       ,
    .enavdd                 (       in2.enavdd              )       ,
    .enzdvdd                (       in2.dis                 )       ,
    .fastboot               (       in2.vprog2              )       ,
```

```
    .di                     (       in2.di                  )       ,
    .test                   (       in2.test                )       ,
    .anatestbus             (       in1.anatestbus          )       ,
    .anatestreq             (       in1.anatestreq          )       ,
    .pg                     (       in1.pg                  )       ,
    .pgdvdd                 (       in1.pgdvdd              )       ,
    .iatb                   (       in1.iatb                )       ,
    .vatb                   (       in1.vatb                )       ,
    .vo                     (       in1.vo                  )
  ) ;
endmodule
```

A.2 Testbench class for voltage regulators

List of Code Segments A.4: Testbench example for charge pump.

```
class cp_tb extends uvm_env;

   `uvm_component_utils(cp_tb)

   cp_env            cp_uvc      ;//config UVC
   power_env         power_uvc   ;//power UVC
   digital_env       digital_uvc ;//digital UVC
   output_env        output_uvc  ;//output UVC

   //cp virtual sequencer
   cp_mcsequencer   cp_mcseqr    ;//Vitual sequencer handle

   //cp scoreboard
   cp_scoreboard cp_sb           ;//Scoreboard handle

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

   //Build phase function
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      cp_uvc      =   cp_env::type_id::create( "cp_uvc"    , this);
      digital_uvc =   digital_env::type_id::create( "digital_uvc" , this);
      power_uvc   =   power_env::type_id::create( "power_uvc" , this);
      output_uvc  =   output_env::type_id::create( "output_uvc" , this);
      cp_mcseqr   =   cp_mcsequencer::type_id::create( "cp_mcseqr" , this);
      cp_sb       =   cp_scoreboard::type_id::create("cp_sb",this);

      `uvm_info("MSG", "Testbench build phase executed",UVM_HIGH)
   endfunction


   function void connect_phase(uvm_phase phase);
      //Connect the TLM ports from the cp and output UVC to the scoreboard
      cp_uvc.cp_tx_agent.monitor.item_collected_port.connect(cp_sb.sb_cp);
      output_uvc.output_tx_agent.monitor.item_collected_port.connect(
        cp_sb.sb_output);
      digital_uvc.digital_tx_agent.monitor.item_collected_port.connect(
        cp_sb.sb_digital);
      power_uvc.power_tx_agent.monitor.item_collected_port.connect(
        cp_sb.sb_power);

      //connect mc sequencer references to UVC sequencer instances
      cp_mcseqr.cp_seqr = cp_uvc.cp_tx_agent.sequencer;
      cp_mcseqr.digital_seqr = digital_uvc.digital_tx_agent.sequencer;
      cp_mcseqr.power_seqr = power_uvc.power_tx_agent.sequencer;
   endfunction : connect_phase

endclass:cp_tb
```

A.3 Digital UVC

A.3.1 Digital UVC environment class

The environment for the digital UVC is shown in the code listing below. This environment instantiates a digital agent.

List of Code Segments A.5: Digital UVC class

```
class digital_env extends uvm_env;
   //utility macro
   `uvm_component_utils(digital_env)

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

   digital_agent digital_tx_agent;//handle for agent

   //build phase function
   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      digital_tx_agent = digital_agent::type_id::create("digital_tx_agent",
          this);
      `uvm_info("MSG", "Agent build phase executed",UVM_HIGH)
   endfunction

   //start of simulation phase function
   function void start_of_simulation_phase(uvm_phase phase);
      `uvm_info( get_type_name(),{"Start of simulation for " ,
          get_full_name()} , UVM_HIGH)
   endfunction : start_of_simulation_phase

endclass : digital_env
```

A.3.2 Digital sequence-item class

The sequence-item class for the digital UVC is listed below.

List of Code Segments A.6: Digital UVC sequence-item class

```
class digital_sequence_item extends uvm_sequence_item;

   rand bit                  dislvl                        ;
   rand bit                  dislvlz                       ;
   rand bit                  dissink                       ;
   rand bit                  dis                           ;
   rand bit                  vprog2                        ;
   rand bit                  vprog3                        ;
   rand bit   [1:0]          vprog5                        ;
   rand bit                  vprog4                        ;
   rand int                  packet_delay                  ;
   rand bit   [3:0]          vprog1                        ;
   rand bit   [3:0]          test                          ;

        bit                  pop_supply                    ;
        bit                  pop_cp                        ;

   function new (string name = "digital_sequence_item");
      super.new(name);
   endfunction : new
endclass: digital_sequence_item
```

A.3.3 Digital driver run-phase task

The digital driver's run-phase task calls the task to drive DUT ports through a virtual interface. The virtual interface points to a real interface (Annex A.6.1 shows the source code of an interface).

List of Code Segments A.7: Digital driver's run phase task

```
1    task run_phase(uvm_phase phase);
2      forever begin
3          seq_item_port.get_next_item(req);
4             `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s",
                 req.sprint()), UVM_HIGH)
5              begin
6                vif.send_to_dut(
7                                req.dislvl              ,
8                                req.dislvlz             ,
9                                req.dissink             ,
10                               req.enavdd              ,
11                               req.dis                 ,
12                               req.vprog1              ,
13                               req.test                ,
14                               req.vprog3              ,
15                               req.vprog5              ,
16                               req.vprog4              ,
17                               req.vprog2              ,
18                               req.pop_supply          ,
19                               req.pop_cp              ,
20                               req.packet_delay
21                              );
22             end
23             `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s",
                  req.sprint()), UVM_LOW)
24           num_sent++;
25           // Communicate item done to the sequencer
26           seq_item_port.item_done();
27         end
28
29   endtask : run_phase
```

A.3.4 Digital monitor run-phase task

The digital monitor's run-phase task calls the task to sample DUT ports through a virtual interface.

List of Code Segments A.8: Digital monitor's run phase task

```
1  task run_phase(uvm_phase phase);
2      forever begin
3        // Create collected packet instance
4        pkt = digital_packet::type_id::create("pkt", this);
5
6          vif.collect_packet(
7                             pkt.dislvl                          ,
8                             pkt.dislvlz                         ,
9                             pkt.dissink                         ,
10                            pkt.enavdd                          ,
11                            pkt.dis                             ,
12                            pkt.vprog1                          ,
13                            pkt.test                            ,
14                            pkt.vprog3                          ,
15                            pkt.vprog5                          ,
16                            pkt.vprog4                          ,
17                            pkt.vprog2                          ,
18                            pkt.pop_supply                      ,
19                            pkt.pop_cp                          ,
20                            pkt.packet_delay
21                           );
22
```

```
23          `uvm_info(get_type_name(), $sformatf("Packet Collected :\n%s",
               pkt.sprint()), UVM_LOW)
24          num_pkt_col++;
25
26          //write call to broadcast received packet to scoreboard
27          item_collected_port.write(pkt);
28          cover_digital.sample();
29        end
30      endtask : run_phase
```

A.4 Power UVC

A.4.1 Power UVC environment class

The environment for the power UVC is shown in the code listing below. This environment instantiates a power agent.

List of Code Segments A.9: power UVC class

```
1  class power_env extends uvm_env;
2     //utility macro
3     `uvm_component_utils(power_env)
4     //Constructor
5     function new (string name, uvm_component parent);
6        super.new(name,parent);
7     endfunction : new
8
9     power_agent power_tx_agent;
10    function void build_phase(uvm_phase phase);
11       super.build_phase(phase);
12       power_tx_agent = power_agent::type_id::create("power_tx_agent", this);
13       `uvm_info("MSG", "Agent build phase executed",UVM_HIGH)
14    endfunction
15
16    function void start_of_simulation_phase(uvm_phase phase);
17       `uvm_info( get_type_name(),{"Start of simulation for " ,
            get_full_name()} , UVM_HIGH)
18    endfunction : start_of_simulation_phase
19 endclass : power_env
```

A.4.2 Power sequence-item class

The sequence-item class for the power UVC is listed below. UVM does not support the randomization of real values. Therefore variables are declared and randomized as 32-bit integers and a division converts the number into 64 bit floating point representation. A real representation of each variable is also considered for legibility. The conversion is done inside the interface model shown in Annex A.6.3.

List of Code Segments A.10: power UVC sequence-item class

```
1  class power_packet extends uvm_sequence_item;
2
3     rand int                  agnd                          ;
4     rand int                  avdd                          ;
5     rand int                  dgnd                          ;
6     rand int                  dvdd                          ;
7     rand int                  pvi                           ;
8     rand int                  vref                          ;
9     rand int                  ibp1u0                        ;
```

```
10      rand int                    packet_delay                    ;
11
12      real                        agnd_rcv                        ;
13      real                        avdd_rcv                        ;
14      real                        dgnd_rcv                        ;
15      real                        dvdd_rcv                        ;
16      real                        pvi_rcv                         ;
17      real                        vref_rcv                        ;
18      real                        ibp1u0_rcv                      ;
19
20
21      function new (string name = "power_packet");
22         super.new(name);
23      endfunction : new
24  endclass: power_packet
```

A.4.3 power driver run-phase task

The power driver's run-phase task calls the task to drive DUT ports through a virtual interface. The virtual interface points to a real interface (Annex A.6.1 shows the source code of an interface).

List of Code Segments A.11: power driver's run phase task

```
1  task run_phase(uvm_phase phase);
2      // Get new item from the sequencer
3        seq_item_port.get_next_item(req);
4          begin
5            vif.send_to_dut(req.agnd                          ,
6                            req.avdd                          ,
7                            req.dgnd                          ,
8                            req.dvdd                          ,
9                            req.pvi                           ,
10                           req.vref                          ,
11                           req.ibp1u0                        ,
12                           req.packet_delay
13                          );
14         end
15
16         `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s",
               req.sprint()), UVM_HIGH)
17       num_sent++;
18       seq_item_port.item_done();
19    endtask : run_phase
```

A.4.4 power monitor run-phase task

The power monitor's run-phase task calls the task to sample DUT ports through a virtual interface.

List of Code Segments A.12: power monitor's run phase task

```
1  task run_phase(uvm_phase phase);
2      forever begin
3        // Create collected packet instance
4        pkt = power_packet::type_id::create("pkt", this);
5
6          vif.collect_packet(pkt.agnd_rcv                         ,
7                             pkt.avdd_rcv                         ,
8                             pkt.dgnd_rcv                         ,
9                             pkt.dvdd_rcv                         ,
10                            pkt.pvi_rcv                          ,
11                            pkt.vref_rcv                         ,
12                            pkt.ibp1u0_rcv                       ,
13                            pkt.packet_delay
14                           );
```

```
15
16          `uvm_info(get_type_name(), $sformatf("Packet Collected :\n%s",
                pkt.sprint()), UVM_LOW)
17          num_pkt_col++;
18
19          //write call to broadcast received packet to scoreboard
20          item_collected_port.write(pkt);
21
22        end
23      endtask : run_phase
```

A.5 SPI UVC

A.5.1 SPI UVC environment class

The environment for the digital UVC is shown in the code listing below. This environment instantiates a SPI agent.

List of Code Segments A.13: SPI UVC class

```
1  class spi_env extends uvm_env;
2     //utility macro
3     `uvm_component_utils(spi_env)
4
5     //Constructor
6     function new (string name, uvm_component parent);
7        super.new(name,parent);
8     endfunction : new
9
10    spi_agent spi_tx_agent;
11
12    function void build_phase(uvm_phase phase);
13       super.build_phase(phase);
14       spi_tx_agent = spi_agent::type_id::create("spi_tx_agent", this);
15       `uvm_info("MSG", "Agent build phase executed",UVM_HIGH)
16    endfunction
17
18    function void start_of_simulation_phase(uvm_phase phase);
19       `uvm_info( get_type_name(),{"Start of simulation for " ,
             get_full_name()} , UVM_HIGH)
20    endfunction : start_of_simulation_phase
21 endclass : spi_env
```

A.5.2 SPI sequence-item class

The sequence-item class for the SPI UVC is listed below. It possesses send and read variables for address and data to coordinate driver and monitor execution. Hence, the driver can drive sequence-items on a positive edge of the clock while the monitor samples the interface on the positive edge.

List of Code Segments A.14: SPI UVC sequence-item class

```
1  class spi_packet extends uvm_sequence_item;
2     rand bit  [31:0]          spi_send_data               ;
3     rand bit  [6:0]           spi_send_addr               ;
4     rand bit  [31:0]          spi_read_data               ;
5     rand bit  [6:0]           spi_read_addr               ;
6     rand bit                  spics                       ;
7     rand bit                  spisi                       ;
8     rand int                  packet_delay                ;
```

```
9     function new (string name = "spi_packet");
10       super.new(name);
11    endfunction : new
12 endclass: spi_packet
```

A.5.3 SPI driver run-phase task

The digital driver's run-phase task calls the task to drive DUT ports through a virtual interface. The virtual interface points to a real interface (Annex A.6.3 shows the source code of an interface).

List of Code Segments A.15: SPI driver's run phase task

```
1    task run_phase(uvm_phase phase);
2        forever begin
3          // Get new item from the sequencer
4          seq_item_port.get_next_item(req);
5            begin
6              vif.send_to_dut(
7                              req.spi_send_data                ,
8                              req.spi_send_addr                ,
9                              req.spics                        ,
10                             req.spisi                        ,
11                             req.packet_delay
12                           );
13           end
14         `uvm_info(get_type_name(), $sformatf("Sending Packet :\n%s",
               req.sprint()), UVM_LOW)
15         seq_item_port.item_done();
16       end
17     endtask : get_and_drive
18   endtask : run_phase
```

A.5.4 SPI monitor run-phase task

The digital monitor's run-phase task calls the task to sample DUT ports through a virtual interface.

List of Code Segments A.16: SPI monitor's run phase task

```
1    task run_phase(uvm_phase phase);
2      forever begin
3        // Create collected packet instance
4        pkt = spi_packet::type_id::create("pkt", this);
5        vif.collect_packet(
6                           pkt.spi_read_data                    ,
7                           pkt.spi_read_addr                    ,
8                           pkt.spics                            ,
9                           pkt.spisi                            ,
10                          pkt.packet_delay
11                        );
12       `uvm_info(get_type_name(), $sformatf("Packet Collected :\n%s",
             pkt.sprint()), UVM_LOW)
13       //write call to broadcast received packet to scoreboard
14       item_collected_port.write(pkt);
15       cover_digital.sample();
16     end
17   endtask : run_phase
```

A.6 Example Interfaces

A.6.1 Example Interface for digital UVC

List of Code Segments A.17: Digital interface.

```
interface digital_if (input clock);
timeunit 1ns;
timeprecision 100ps;

import uvm_pkg::*;
`include "uvm_macros.svh"

import digital_pkg::*;

  // Actual Signals

  logic                       dislvl                    ;
  logic                       dislvlz                   ;
  logic                       dissink                   ;
  logic                       dis                       ;
  logic       [3:0]           vprog1                    ;
  logic       [3:0]           test                      ;
  logic                       vprog3                    ;
  logic       [1:0]           vprog5                    ;
  logic                       vprog4                    ;
  logic                       vprog2                    ;

  //Signal in_data_vld to synchronize sent and revceived packets
  logic                       in_data_vld               ;
  logic                       sample_monitor            ;

  //pop_supply and pop_cp signals to coordinate sent packets and reference
     model calculation
  logic                       pop_supply                ;
  logic                       pop_cp                    ;


  // Gets a packet and drive it into the DUT
  task send_to_dut(input
                          bit               dislvl_in             ,
                          bit               dislvlz_in            ,
                          bit               dissink_in            ,
                          bit               enavdd_in             ,
                          bit               dis_in                ,
                          bit [3:0]         di_in                 ,
                          bit [3:0]         test_in               ,
                          bit               vprog3_in             ,
                          bit [1:0]         vprog5_in             ,
                          bit               vprog4_in             ,
                          bit               vprog2_in             ,
                          bit               pop_supply_in         ,
                          bit               pop_cp_in             ,
                          int               packet_delay_in
                  );

    repeat(packet_delay_in)
      @(negedge clock);
    @(negedge clock) begin
      di                        =           di_in                 ;
      dislvl                    =           dislvl_in             ;
      dislvlz                   =           dislvlz_in            ;
      dissink                   =           dissink_in            ;
      dis                       =           dis_in                ;
      test                      =           test_in               ;
      vprog5                    =           vprog5_in             ;
      vprog4                    =           vprog4_in             ;
      vprog3                    =           vprog3_in             ;
      vprog2                    =           vprog2_in             ;
      vprog1                    =           vprog1_in             ;
      pop_supply                =           pop_supply_in         ;
      pop_cp                    =           pop_cp_in             ;

    end
```

```
68      in_data_vld <= 1'b0;
69    endtask : send_to_dut
70
71    // Collect Packets
72    task collect_packet(output
73                            bit                 dislvl_in               ,
74                            bit                 dislvlz_in              ,
75                            bit                 dissink_in              ,
76                            bit                 dis_in                  ,
77                            bit [3:0]           di_in                   ,
78                            bit [3:0]           test_in                 ,
79                            bit                 vprog3_in               ,
80                            bit [1:0]           vprog5_in               ,
81                            bit                 vprog4_in               ,
82                            bit                 vprog2_in               ,
83                            bit                 pop_supply_in           ,
84                            bit                 pop_cp_in               ,
85                            int                 packet_delay_in
86                        );
87
88      @(negedge(in_data_vld))begin
89          sample_monitor = 1'b1;
90          @(posedge clock) begin
91
92              di_in                   =       di                      ;
93              dislvl_in               =       dislvl                  ;
94              dislvlz_in              =       dislvlz                 ;
95              dissink_in              =       dissink                 ;
96              enavdd_in               =       enavdd                  ;
97              dis_in                  =       dis                     ;
98              test_in                 =       test                    ;
99              vprog5_in               =       vprog5                  ;
100             vprog4_in               =       vprog4                  ;
101             vprog3_in               =       vprog3                  ;
102             vprog2_in               =       vprog2                  ;
103             vprog1_in               =       vprog1                  ;
104             pop_supply_in           =       pop_supply              ;
105             pop_cp_in               =       pop_cp                  ;
106         end
107
108         sample_monitor = 1'b0;
109         repeat(packet_delay_in)
110           @(negedge clock);
111     end
112
113   endtask : collect_packet
114
115 endinterface : digital_if
```

A.6.2 Example Interface for power UVC

List of Code Segments A.18: Power interface.

```
1  interface power_if (input clock);
2  timeunit 1ns;
3  timeprecision 100ps;
4
5  import uvm_pkg::*;
6  `include "uvm_macros.svh"
7
8  import power_pkg::*;
9    // Actual Signals
10   logic       [63:0]          agnd                    ;
11   logic       [63:0]          avdd                    ;
12   logic       [63:0]          dgnd                    ;
13   logic       [63:0]          dvdd                    ;
14   logic       [63:0]          pvi                     ;
15   logic       [63:0]          vref                    ;
16   logic       [63:0]          ibp1u0                  ;
17
18   //Signal in_data_vld to synchronize sent and revceived packets
19   logic                       in_data_vld             ;
20   logic                       sample_monitor          ;
21
22   // Gets a packet and drive it into the DUT
23   task send_to_dut(input int    agnd_in                       ,
```

```
                          int     avdd_in                          ,
                          int     dgnd_in                          ,
                          int     dvdd_in                          ,
                          int     pvi_in                           ,
                          int     vref_in                          ,
                          int     ibp1u0_in                        ,
                          int     packet_delay_in
                    );
      repeat(packet_delay_in)
        @(negedge clock);

      in_data_vld <= 1'b1;
      @(negedge clock) begin
        agnd            =          $realtobits( agnd_in / 10)      ;
        avdd            =          $realtobits( avdd_in / 10)      ;
        dgnd            =          $realtobits( dgnd_in / 10)      ;
        dvdd            =          $realtobits( dvdd_in / 10)      ;
        pvi             =          $realtobits( pvi_in  / 10)      ;
        vref            =          $realtobits( vref_in / 10)      ;
        ibp1u0          =          $realtobits( ibp1u0_in / 10)    ;
      end
      in_data_vld <= 1'b0;
    endtask : send_to_dut

    // Collect Packets
    task collect_packet(output real    agnd_in                       ,
                        real           avdd_in                       ,
                        real           dgnd_in                       ,
                        real           dvdd_in                       ,
                        real           pvi_in                        ,
                        real           vref_in                       ,
                        real           ibp1u0_in                     ,
                        int           packet_delay_in
                    );
      @(negedge(in_data_vld))begin

      sample_monitor = 1'b1;
      // Drive remaining signals
      @(posedge clock) begin
        agnd_in         =          $bitstoreal(  agnd      )        ;
        avdd_in         =          $bitstoreal(  avdd      )        ;
        dgnd_in         =          $bitstoreal(  dgnd      )        ;
        dvdd_in         =          $bitstoreal(  dvdd      )        ;
        pvi_in          =          $bitstoreal(  pvi       )        ;
        vref_in         =          $bitstoreal(  vref      )        ;
        ibp1u0_in       =          $bitstoreal(  ibp1u0    )        ;
      end
        sample_monitor = 1'b0;
        repeat(packet_delay_in)
        @(negedge clock);
        end

    endtask : collect_packet

endinterface : power_if
```

A.6.3 Example Interface for SPI UVC

The spi_write function implements the SPI protocol and drives data through the spidin input port.

List of Code Segments A.19: Power interface.

```
interface spi_if (input clock);
timeunit 1ns;
timeprecision 100ps;

import uvm_pkg::*;
`include "uvm_macros.svh"

import spi_pkg::*;

  // Actual Signals
  wire                        spidin                  ;
  wire                        spics                   ;
  wire                        spisi                   ;
```

```
14
15    //Signal sending_spi to synchronize sent and revceived packets
16    logic                          sending_spi             ;
17    logic                          sample_monitor          ;
18    logic                          start                   ;
19
20    // Gets a packet and drive it into the DUT
21  task send_to_dut(input
22                             bit     [31:0]         spi_send_data
                                                              ,
23                             bit     [6:0]          spi_send_addr
                                                              ,
24                             bit                    spics_in
                                                                ,
25                             bit                    spisi_in
                                                                ,
26                             int                    packet_delay_in
27                    );
28
29      @(posedge clock) begin sending_spi <= 1'b1; sample_registers = 1'b0;
            addr_send = spi_send_addr; end
30      spi_write(spi_send_addr, spi_send_data,0,1);//write task with enabled
            slave
31      @(posedge clock) sample_registers = 1'b1;
32      spi_write(spi_send_addr, spi_send_data,0,0);//write task with disabled
            slave
33      @(posedge clock) sending_spi <= 1'b0;
34  endtask : send_to_dut
35
36    // Collect Packets
37  task collect_packet(output
38                             bit     [31:0]         spi_read_data
                                                              ,
39                             bit     [6:0]          spi_read_addr
                                                              ,
40                             bit                    spics_in
                                                                ,
41                             bit                    spisi_in
                                                                ,
42                             int                    packet_delay_in
43                    );
44      integer          i;
45      @(negedge(sample_registers))begin
46      sample_monitor = 1'b1;
47      @(posedge clock) begin
48         spics_in                          =          spics          ;
49         write_read                        =          spidin         ;
50      end
51      for ( i=0; i<=6; i=i+1 ) begin
52        @(posedge clock )
53           spi_read_addr[i] <=  spidin;
54      end
55      for ( i=0; i<=31; i=i+1 ) begin
56        @(posedge clock )
57         spi_read_data[i] <= spidin ;
58      end
59      @(negedge clock )
60      sample_monitor = 1'b0;
61      end
62  endtask : collect_packet
63
64  endinterface : spi_if
```

A.7 Other relevant code listings

A.7.1 Virtual Sequences class example

Virtual sequence require a declaration of a parent sequencer parameter that points to the regular sequencer (inside the agent) on which the sequence will run. Handles to local sequences are also defined and the uvm_do_on macros coordinate the sequence generation process.

List of Code Segments A.20: Virtual sequence example.

```
class cp_mcseq_run_all extends cp_mcseq_base_seq;

   //utility macro
   `uvm_object_utils(cp_mcseq_run_all)

   // Constructor
  function new(string name="cp_mcseq_run_all");
    super.new(name);
  endfunction

   //declare p_sequncer
   `uvm_declare_p_sequencer(cp_mcsequencer)
   //Handles to the UVC sequences to execute
   //cp sequences
   cp_1_seq                 cp_seq;
   cp_resistive_maxload_seq cp_res_max_load;
   cp_fixed_load_seq        cp_fix_load;
   cp_fixed_half_load_seq   cp_fix_half_load;
   cp_fixed_small_load_seq  cp_fix_small_load;
   cp_enable_low_seq        cp_enable_low;

   //digital sequences
   digital_mode_seq         digital_mode_seq;
   digital_test_seq         digital_test;
   digital_di_seq           digital_di_seq;
   digital_pop_cp           digital_pop_cp;
   //digital_disable          digital_dis;
   digital_enable           digital_en;
   digital_disable          digital_dis;
   //digital pseudo-random sequence
   digital_random           digital_ran

   //power sequences
   power_1_seq              power_seq;
  //digital random sequence
   digital_random           digital_ran;

  virtual task body();
    //Configure en
    `uvm_do_on(cp_enable, p_sequencer.cp_seqr)

    //update load with pop logic
    `uvm_do_on(digital_pop_cp, p_sequencer.digital_seqr)

    //digital disable sequence
    `uvm_do_on(digital_dis, p_sequencer.digital_seqr)

    //digital sequence for mode sweep
    `uvm_do_on(digital_mode_seq, p_sequencer.digital_seqr)

    //digital sequence for test sweep
    `uvm_do_on(digital_test, p_sequencer.digital_seqr)

    //Configure CP packet for small load
    `uvm_do_on(cp_fix_small_load, p_sequencer.cp_seqr)

    //digital sequence for di sweep
    `uvm_do_on(digital_di_seq, p_sequencer.digital_seqr)

    //digital random sequence
```

```
61      `uvm_do_on(digital_ran    , p_sequencer.digital_seqr)
62
63      //increase load to speed up discharge
64      `uvm_do_on(cp_fix_load, p_sequencer.cp_seqr)
65
66      //digital sequence to disable cp with endvdd
67      `uvm_do_on(digital_dis, p_sequencer.digital_seqr)
68
69    endtask
70
71  endclass : cp_mcseq_run_all
```

B

VCS URG coverage report

An example coverage report of a covergroup is presented. Results are generated for each coverpoint and each cross. When no bin construct is defined, there is an implicit bin for all possible values of a coverage point variable. Explicit bins (manually defined with the bin construct) define the possible scenarios to fully cover a coverage point. For each coverpoint/cross a coverage score is presented which are used to compute the overall coverage score. Covered and uncovered scenarios are detailed.

Unified Coverage Report :: Group :: digital_pkg::digital_monitor::cover_digital

<username>

Summary for Variable enable_dvdd

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	2	0	2	100.00

Automatically Generated Bins for enable_dvdd

Bins

Summary for Variable enable_dvdd1

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	1	0	1	100.00

User Defined Bins for enable_dvdd1

Bins

NAME	COUNT	AT LEAST
one	13	1

Summary for Variable enable_dsl

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	2	0	2	100.00

Automatically Generated Bins for enable_dsl

Bins

Summary for Variable enable_dsl0

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	1	0	1	100.00

User Defined Bins for enable_dsl0

Bins

NAME	COUNT	AT LEAST
one	24	1

Summary for Variable enable_dissink

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	2	0	2	100.00

Automatically Generated Bins for enable_dissink

Bins

Summary for Variable enable_mode

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	1	0	1	100.00

User Defined Bins for enable_mode

Bins

NAME	COUNT	AT LEAST
b1_0	25	1

Summary for Variable enable_swilim

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
Automatically Generated Bins	2	0	2	100.00

Automatically Generated Bins for enable_swilim

Bins

Summary for Variable enable_dttrim

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	4	0	4	100.00

User Defined Bins for enable_dttrim

Bins

NAME	COUNT	AT LEAST
b2_0	6	1
b2_1	7	1
b2_2	7	1
b2_3	6	1

Summary for Variable enable_test

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT
User Defined Bins	12	0	12	100.00

User Defined Bins for enable_test

Bins

NAME	COUNT	AT LEAST
test_valid_0	11	1
test_valid_1	1	1
test_valid_2	1	1
test_valid_3	1	1
test_valid_4	1	1
test_valid_5	1	1
test_valid_6	1	1
test_valid_7	1	1
test_valid_8	1	1
test_valid_9	1	1
test_valid_a	1	1
test_valid_b	1	1

Summary for Cross cx_test_all

Samples crossed: enable_dvdd enable_dsl enable_dttrim enable_swilim enable_mode

CATEGORY	EXPECTED	UNCOVERED	COVERED	PERCENT	MISSING
Automatically Generated Cross Bins	32	21	11	34.38	21

Automatically Generated Cross Bins for cx_test_all

Element holes

enable_dvdd	enable_dsl	enable_dttrim	enable_swilim	enable_mode	COUNT	AT LEAST	NUMBER
[auto[0]]	[auto[0]]	*	[auto[1]]	*	--	--	4
[auto[0]]	[auto[1]]	*	*	*	--	--	8
[auto[1]]	[auto[0]]	[b2_0]	[auto[1]]	*	0	1	1
[auto[1]]	[auto[0]]	[b2_2 , b2_3]	[auto[1]]	*	--	--	2
[auto[1]]	[auto[1]]	[b2_0]	*	*	--	--	2
[auto[1]]	[auto[1]]	[b2_1]	[auto[1]]	*	0	1	1
[auto[1]]	[auto[1]]	[b2_2]	*	*	--	--	2
[auto[1]]	[auto[1]]	[b2_3]	[auto[1]]	*	0	1	1

Covered bins

enable_dvdd	enable_dsl	enable_dttrim	enable_swilim	enable_mode	COUNT	AT LEAST
auto[0]	auto[0]	b2_0	auto[0]	b1_0	4	1
auto[0]	auto[0]	b2_1	auto[0]	b1_0	2	1
auto[0]	auto[0]	b2_2	auto[0]	b1_0	4	1
auto[0]	auto[0]	b2_3	auto[0]	b1_0	3	1
auto[1]	auto[0]	b2_0	auto[0]	b1_0	2	1
auto[1]	auto[0]	b2_1	auto[0]	b1_0	3	1
auto[1]	auto[0]	b2_1	auto[1]	b1_0	1	1
auto[1]	auto[0]	b2_2	auto[0]	b1_0	2	1
auto[1]	auto[0]	b2_3	auto[0]	b1_0	2	1
auto[1]	auto[1]	b2_1	auto[0]	b1_0	1	1
auto[1]	auto[1]	b2_3	auto[0]	b1_0	1	1