FH JOANNEUM Elektronik und Computer Engineering

dialog SEMICONDUCTOR

Digital Verification using UVM and System Verilog

in cooperation with Dialog Semiconductors

Thesis 2 / 2

created by:

Roman Winkler

at the bachelor degree programme Elektronik und Computer Engineering
der FH JOANNEUM – University of Applied Sciences, Austria

supervised by:

Dipl-Ing Dr. Netzberger

external supervisor:

Joao Martins, MSc

Graz, July 31, 2018

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Bachelorarbeit 2 selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle ausgedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet. Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

31. Juli 2018

Graz, am

[illegible]

(Unterschrift)

Kurzfassung

Pre-silicon verification – die Verifizierung schon vor der Fabrizierung – ist ein wichtiger Bestandteil im Entwicklungsprozess, um eine hohe Funktionsgüte und Qualität des Produktes zu sichern. Um ein digitales, event-basiertes SystemVerilog Low Dropout Voltage Regulator (LDO) Steuerungsmodul zu verifizieren, welches später synthetisiert wird und in einem von Dialog Semiconductors Power Management Integrated Circuits (PMICs) Anwendung findet, wird ein Universal Verification Component (UVC) des LDO mit der Nutzung von Universal Verification Methodology (UVM) erstellt. Dieses interagiert mit dem Modul und bietet die Möglichkeit, dessen Funktionalität während direkten und randomisierten Tests zu verifizieren. Das UVC implementiert außerdem die Grundsätze von vertikaler und horizontaler Wiederverwendung und kann daher in verschiedenen Projekten sowie in jeweils Top- und Modullevelverifikation zum Einsatz gebracht werden. Zu Beginn wird das digitale, event-basierte LDO Modell erstellt, welches alle Funktionalitäten von verschiedensten LDO-Typen in einem, konfigurierbaren SystemVerilog Modul repräsentiert – um dieses Modell herum wird die UVM Umgebung entwickelt, welche das Verhalten des Modells selbst, sowie das des Steuermoduls verifiziert. Danach wird das LDO Modell mit dem Steuerungsmodul verbunden und auf Modulebene verifiziert. Schlussendlich wird das LDO - UVC in die bereits existierende PMIC Toplevel-Umgebung eingebettet, um das Steuerungsmodul über vertikale Wiederverwendung auf Chiplevel zu verifizieren. Zusätzlich dazu geben Funktions- und Codeanalysen Aufschluss über die Verifikationsqualität während und nach den Testdurchgängen. Diese Schlüsselpunkte bieten dem Testingenieur eine einfach handzuhabende und vielfach einsetzbare Verifikationslösung für diesen bestimmten Systemblock, aber auch für alle anderen, die vom Vorhandensein eines generischen und konfigurierbaren LDO Modells profitieren.

Abstract

To ensure high-quality and error-less digital products even before production, pre-silicon verification plays a vital part in the development process.

To verify a digital event-based SystemVerilog LDO controller module, which will later be synthesized and used inside a Dialog Semiconductor's PMIC, an LDO - UVC with the use of UVM is created. It interacts with this module and has the abilities to verify its behavior during directed and constrained random test scenarios. The UVC implements the ideas of horizontal and vertical re-use so it can be configured for different projects and used in both top and module level verification. Firstly, the digital event-based LDO model is created to feature all possible functionalities of several LDO types inside one configurable SystemVerilog module. Secondly, the UVM environment around it is developed to verify the models behavior. After that, the main LDO controller module is attached and verified in module level. Finally, the LDO - UVC is embedded into the existing PMIC top level environment via vertical re-use and the controller is verified on chip level. Additionally, functional and code coverage is implemented to assess the verification quality.

These key features provide the engineer with a straightforward out-of-the-box solution to verify both this specific block and all others, that might take advantage of having a generic LDO model attached.

List of Abbreviations

ADC	Analog - Digital Converter
AHB	Advanced High Performance Bus
ASIC	Application Specific Integrated Circuit
DAC	Digital - Analog Converter
DfT	Design for Test
DUT	Design under Test
DUV	Design under Verification
DVC	Dynamic Voltage Control
FET	Field Effect Transistor
HDL	Hardware Description Language
HDVL	Hardware Description and Verification Language
LDO	Low Dropout Voltage Regulator
OOP	Object Oriented Programming
PMIC	Power Management Integrated Circuit
SPMI	System Power Management Interface
TO	Tape Out
UVC	Universal Verification Component
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Description Language

Contents

List of Figures

Listings

1 Introduction

This thesis shall introduce the creation of a UVC, which incorporates the aspects of re-usability, scalability and stability in order to be used in later designs and further projects. The UVC includes a model for event based simulation of an analog functionality as well as an environment connected to it, which provides all abilities needed to correctly interact with the digital pendant in the same way as one would with the analog device. This specific UVC shall represent the functionality of LDOs, specifically those used in Dialog Semiconductors PMICs.

1.1 Motivation

The development or advancement of an electronic device of such a high level class brings together many different teams, time-lines and deliverables. To deliver a fully functional and verified design at Tape Out (TO), early debugging and verification has to be done. In order to accomplish this with various stages of finished and in-progress blocks and to highly minimize simulation time, analog circuits tend to be represented as digital models in Universal Verification Components - mostly for module level verification but also for the use in top level verifications via vertical reuse. Those representations have to implement all functionalities with the same in- and output signals given by the real device (pin accurate), which means that no or only minor adaptations have to be done in the existing digital layers that would or will later connect to the intra-chip analog interfaces. When developing such UVCs, the designer has to choose a suitable level of abstraction for the model and provide source code as generic as possible, so the UVC can be used in many different flavors.

1.2 Objectives

The main objective for this thesis is to create a digital, event based representation of LDOs used in PMIC currently in development, implementing all the functionalities given in the scope of that specific PMIC as well as reasonable features that might be needed in future or similar projects. Further, a UVM environment shall be created to interact with the created model - this leads to reusable verification methods and easier integration when used in future projects to simulate analog behavior. After the finishing and verifying the UVC and testbench, the UVC shall be embedded into the Dialog Semiconductor work-flow for later reuse.

1.3 Outline

This thesis is divided into three parts: First, a background in digital development, as well as the history of Hardware Description Languages (HDLs) and the further progress towards reusable verification methods such as UVM is given. Secondly, it is shown how the analog functionality of LDOs is evaluated, transformed into digital blocks and implemented as a single SystemVerilog module. It will be described how this module is further embedded into the UVM environment and verified using various test-scenarios together with the rest of the digital environment. After the successful verification, the coverage report and the integration of the UVC into the existing work-flow will be discussed. The last part draws a conclusion on the progress and outlines possible further developments in this area.

2 Background

2.1 Development and Verification of Digital Designs

In the last years digital hardware has drastically improved and grown, leading to a highly increased number of transistors, a smaller footprint and faster transmission rates. To implement and verify such an extensive digital design one has to consider multiple layers of abstraction to progressively create a design that is - or can be - verified at various levels of complexity to shield the verification engineer from low level implementations when high level problems shall be solved. This way, the amount of data or available information can be broken down and restricted to manageable sizes, providing only the very vital details needed.

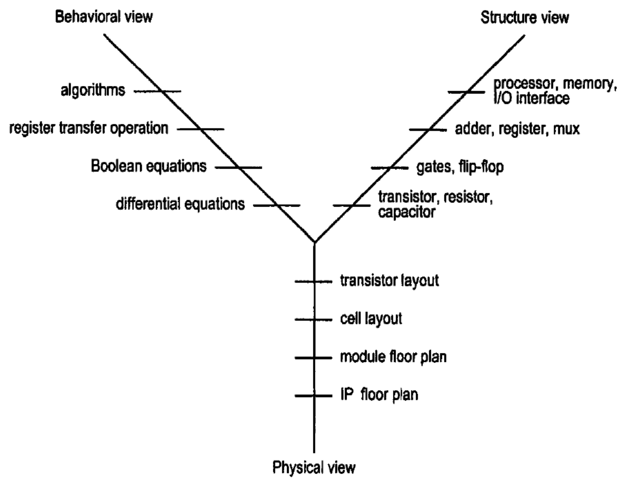
According to [2], a system can be divided into three perspectives: *Behavioral View*, *Structural View* and *Physical View*. The behavioral view describes the functionality of the system as a black box model, ignoring the internal implementation. A structural view further describes the internal implementation (white box model), adding a netlist or schematic structure. Physical view adds real life characteristics like size of components or their location.

The low-level implementation represents the least abstracted view and can be further parted into:

- Transistor level
- Gate Level
- Register Transfer Level
- Processor level

	Typical blocks	**Signal representation**	**Time representation**	**Behavioral description**	**Physical description**
Transistor	transistor, resistor	voltage	continuous function	differential equation	transistor layout
Gate	and, or, xor, flip-flop	logic 0 or 1	propagation delay	Boolean equation	cell layout
RT	adder, mux, register	integer, system state	clock tick	extended FSM	RT-level floor plan
Processor	processor, memory	abstract data type	event sequence	algorithm in C	IP-level floor plan

(a) Characteristics of each abstraction level [2, p. 13]



(b) Y-chart connecting the different views and abstraction layers [2, p.10]

Figure 2.1: Lowest level of abstraction

Figure 2.1 combines these high detail levels and the methods used to describe them.

The scope of this thesis will mainly stay in the areas of behavioral and structural view on gate, register transfer and processor level - all in a digital point of view. Keeping these separations in mind, a clean and structured system can be established and maintained.

2.2 System Verilog

System Verilog, officially the IEEE Std $1800 - 2005^{TM}$ *standard, is a set of extensions to the IEEE Std* $1364 - 2005^{TM}$ *Verilog Standard (commonly referred to as Verilog 2005). These extensions provide new and powerful language constructs for modeling and verifying the behavior of designs that are aver increasing in size and complexity. The SystemVerilog extensions to Verilog can be generalized into two primary categories:*

- *Enhancements primarily addressing the needs of hardware modeling, both in terms of overall efficiency and abstraction levels*
- *Verification enhancements and assertion for writing efficient, race-free testbenches for very large, complex designs.* [3]

The mentioned IEEE $1800 - 2005^{TM}$ standard has then evolved over IEEE $1800 - 2009^{TM}$ and IEEE $1800 - 2012^{TM}$ to the latest version of IEEE $1800 - 2017^{TM}$.

At its core, Verilog is a classical HDL, providing a common framework for designers and various software tools to implement the crucial functionality, that classical programming languages like C or Java miss, which is true concurrent operations, the concept of propagation and delay or the connection of parts.[2] SystemVerilog further integrates concepts of C, C++ and Very High Speed Integrated Circuit Hardware Description Language (VHDL) along with assertions used in the verification part, which makes it a Hardware Description and Verification Language (HDVL).[3]

The key features, according to [3], are listed below:

- A unified assertion language for both simulation and formal verification
- Object oriented C++ like classes, with encapsulation, inheritance, and polymorphism
- Interfaces to encapsulate communication and protocol checking within a design
- Special program blocks and clocking domains for defining race free test programs
- Constrained random number generation
- C like data types, such as int
- User-defined types, using the C typedef
- Enumerated types

- Type casting
- Structures and unions, as in C
- Strings, dynamic arrays, associative arrays and lists
- External compilation-unit scope declarations
- ++, –, += and other assignment operators
- Pass by reference to tasks, functions and modules
- Semaphore and mailbox inter-process communication and synchronization
- A Direct Programming Interface (DPI) to allow SystemVerilog to directly call C functions, and for C functions to directly call Verilog functions, without the complex Verilog Programming Language Interface (PLI).

Please beware, that not all listed features are synthesizable - some are only intended in verification scopes.

2.3 Universal Verification Methodology

Implementing the ideas of Object Oriented Programming (OOP), SystemVerilog gives the ability to define often used algorithms and concepts inside of classes for improved re-usability by extending from those for every possible test application.
The UVM Class Library provides generic utilities, such as component hierarchy, transaction library model (TLM), configuration database, etc., which enable the user to create virtually any structure he/she wants for the testbench.[1]
This leads to a faster implementation of basic structures often or always used in a test bench design, again giving some kind of abstraction and shielding from low level implementations.

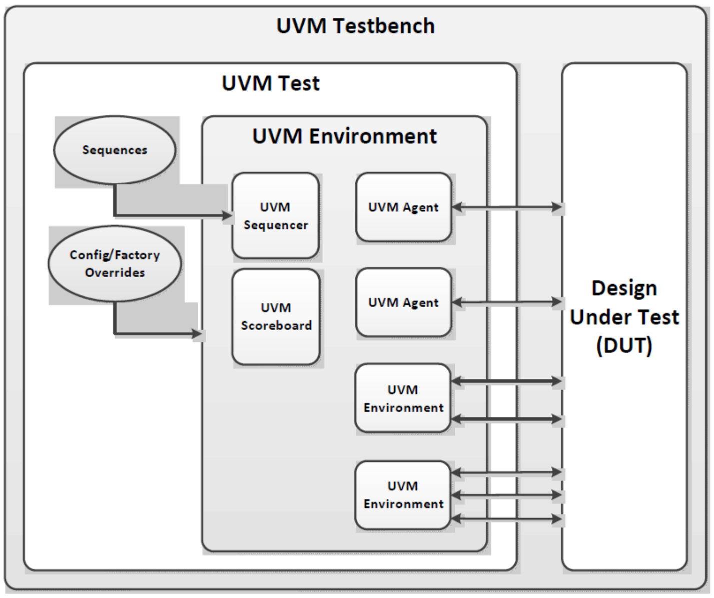


Figure 2.2: Principle structure of a UVM testbench [1]

The most common structure of such a test bench is displayed in figure 2.2, which shows the division into the test bench wrapper (UVM Testbench), the unique test file (UVM Test) and the environment needed to implement agents, drivers, monitors, subscribers, scoreboards and more to interact with the Design under Test (DUT). The UVM library, currently in version *1.2*, provides an inheritance structure and base or template classes (seen in figure 2.3) for each component needed. The root base class is represented by *uvm_object*, from which each class used in a UVM project should derive from at least.

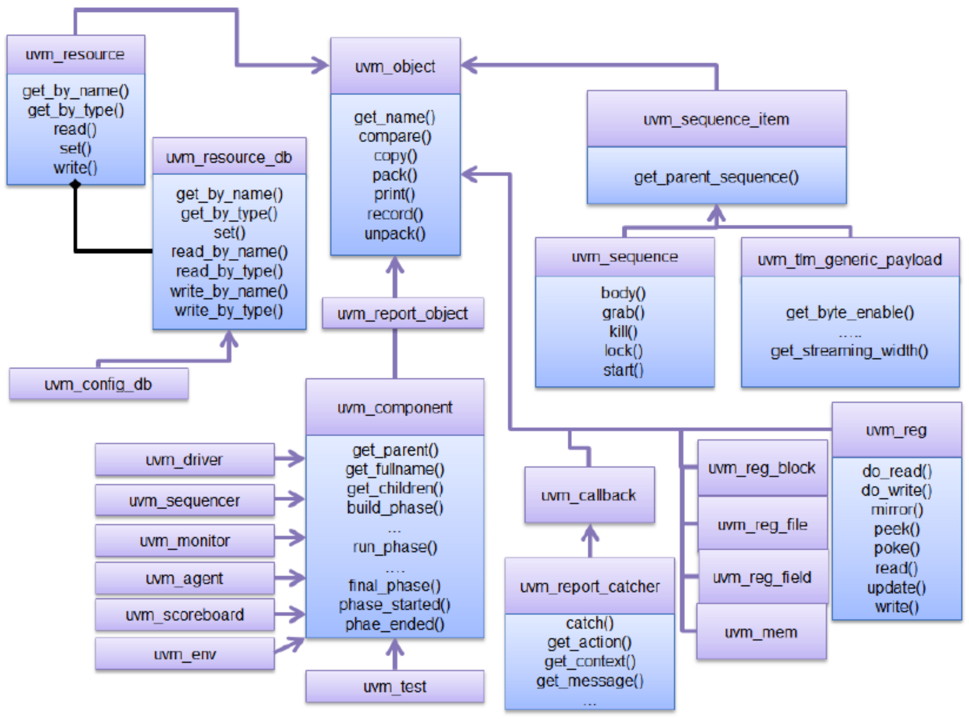


Figure 2.3: UVM base class structure [1, p.4]

The following pages describe the use of the mentioned components.

2.3.1 UVM Testbench

The testbench usually connects to the DUT or Design under Verification (DUV) via an interface, provides a handle to it over a database which can be accessed by the UVM environment and starts the test. The classical understanding of a test bench (often just one file, directly manipulating the ports of the DUT) is replaced by the idea of a multi class, hierarchically built implementation with (randomly created) transactions pushed to the DUT by a driver instead. Code snippet 2.1 show a standard minimal example of such a testbench file.

Listing 2.1: Minimal testbench example

```
module tb_top;
  my_interface if();

  initial begin
    uvm_config_db #(virtual my_interface)::set(null, "*my_agent_h*", "if", if)
        ;
```

```
6        run_test("my_test");
7      end
8
9      my_dut dut(.intf(if));
10
11   endmodule
```

Line 2 shows the instantiation of an interface to connect (in this case) the DUT with the rest of the environment (done with line 9). The next clause is represented by an `initial begin end` block, which takes the handle to this interface and forwards it to the environments agent *my_agent* over the UVM configuration database. Line 6 finally calls the UVM test - which proceeds with configuration, creation of the needed environment and stimulus creation via sequences. Optimally, the testbench only connects components, forwards interfaces and calls the test. By calling the test by name, it is easy to change the testfile to be used for a specific simulation from the outside either by providing a string or via the implemented method $+UVM\_TEST = my\_test$ on the simulator's command line.

2.3.2 Interface

See [3, p. 225] for reference.

Interfaces in SystemVerilog are static constructs that wrap inputs and outputs to and from a DUT. The single interface variables get connected to modules which are instantiated inside the testbench and then forwarded as handles to the UVM configuration database. From there, every class with the correct naming/scope combination can access this handle and its ports. An interface could solely contain in- and outputs, but also implement checkers or protocol specific mechanisms. In the given example at 2.2, two so called modports are added to define the driving direction of the ports according to the usage of the interface. The usage as `my_interface_h.master.input_val2` prevents a driver from wrongly driving a pin that might actually be an output pin in this context.

Listing 2.2: Minimal UVM interface example

```
interface my_interface;
  logic input_val1;
  logic input_val2;
  logic output_val1;

  modport slave{input input_val1, input_val2, output output_val1;}
  modport master{output input_val1, input_val2, input output_val1;}
endinterface
```

2.3.3 UVM Test

See [4, p. 359] for reference.

This class represents the highest UVM hierarchy, which instantiates the top level environment, invokes the creation of transaction sequences and optionally does further configuration. It also raises the main objection of the simulation and therefore highly contributes to the simulations runtime. Although it is a class, it doesn't have to be created at runtime by the user. This is done automatically via the UVM registry, as soon as the name of the class registered over the '*uvm_component_utils*()-macro is called via *run_test*() in the testbench. This makes it easier to run simulations with multiple different test classes without having to change the source code (in certain ways even without recompiling). Each class should at least register with the registry, have a constructor and extend from *uvm_object* in some way. Further class objects, parameters, functions and uvm phase methods can be implemented as needed. The class in listing 2.3 only implements the constructor, instantiates the environment object and registers with the *uvm_component_utils* macro. Further, two UVM phase methods are implemented - those are called automatically once for each object for each simulation run according to the phase the simulator is in. The method called at build time is *build_phase*, which creates the environment object over a UVM registry method. The task *run_phase* is called at run time and raises the objection for this test for 100 μseconds. Each UVM component can raise an objection, which keeps the simulator from going into the next phase as long as not all objections have been dropped. To keep things simple, it is advised to only raise the objection once inside the UVM test object.

Listing 2.3: Minimal UVM test example

```
class my_test extends uvm_test;
  uvm_component_utils(my_test)

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

  my_environment my_environment_h;

  function void build_phase(uvm_phase phase);
    my_environment_h = my_environment::type_id::create("my_environment_h",
        this);
  endfunction

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #100us
```

```
      phase.raise_objection(this);
    endtask
  endclass
```

The commonly used simulation phases are defined in [4, p.148] as uvm_build_phase, uvm_connect_phase, uvm_end_of_elaboration_phase, uvm_start_of_simulation_phase, uvm_run_phase, uvm_extract_phase, uvm_check_phase, uvm_report_phase and uvm_final_phase. *The common phases are the set of function and task phases that all uvm_components execute together. All uvm_components are always synchronized with respect to the common phases. The names of the UVM phases (which will be returned by get_name() for a phase instance) match the class names specified below with the "uvm_" and "_phase" removed.* [4, p.148]
In build_phase, most objects should be constructed - therefore the build_methods are called in a top down manner to hierarchically build the environment. After the tests build method returns, all the build methods of the just created objects are called which leads to a recursive process.

2.3.4 UVM Environment

See [4, p. 361] for reference.
The environment is instantiated by the test file and may contain handles to agents, register models, sequencers, analysis ports and more. There is no limit on how many environments can be instantiated in one test or whether an agent has to be inside an environment - it depends on the use case. In principle, listing 2.4 shows the same basic steps taken inside an environment class like before in the test class: Registry macro, constructor, objects and phase methods. In addition to that, the monitors analysis port gets connected to the scoreboards' analysis export inside the connect_phase. This is also included in the explanation of UVM agent.

Listing 2.4: Minimal UVM environment example

```
class my_environment extends uvm_environment;
  `uvm_component_utils(my_environment)

  my_agent my_agent_h;
  my_scoreboard my_scoreboard_h;

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

  function void build_phase(uvm_phase phase);
    my_agent_h = my_agent::type_id::create("my_agent_h", this);
    my_scoreboard_h = my_scoreboard::type_id::create("my_scoreboard_h", this);
    endfunction

    function void connect_phase(uvm_phase phase);
      my_agent_h.my_monitor_h.aport.connect(my_scoreboard_h.my_export_h);
    endfunction

    task run_phase(uvm_phase phase);

    endtask
  endclass
```

2.3.5 UVM Agent

See [4, p. 362] for reference.

The UVM Agent unites a sequencer managing the sequences of transactions, a driver pushing those transactions to the DUT and a monitor collecting the transactions. The general rule of thumb is to provide one agent for each connected interface. In addition to the already introduced concepts and phases, this agent implements the connect_phase method, which gets executed after build_phase. This method creates all needed connections between instantiated objects - mostly over the construct of ports. In this use case, the sequences containing sequence items are transferred to the driver via the sequence-item port (this connection is seen in listing 2.5, line 22). In addition to the sequence-item port, analysis ports exist that may connect a monitor to specific subscribers like scoreboards or coverage collectors to transfer sequence items (as seen in listing 2.4). Usually, an agent has an active and passive mode implemented, where the active one initializes driver, monitor, sequencer and connects interfaces as well as the driver to the sequencer (as seen in listing 2.5). The passive mode on the other hand solely focuses on monitoring the interface without actively driving or manipulating anything. Therefore, only a handle to the interface and the monitor object is needed in this case. To choose between modes before entering the agents build process, a variable of type *uvm_active_passive_enum* with the options of $UVM\_PASSIVE$ and UVM_ACTIVE is commonly passed to the agent. Lines 14 and 15 further show the usage of the UVM configuration database, which can hold handles to uvm_objects as well as variables of all types. The main use in uvm is to share interface handles from the testbench to all classes below, but also parameterization or global values can be provided over this database. It is used with the shown getter and setter methods, where the scope, the name inside the database and the type can be configured at function call.

Listing 2.5: Minimal UVM agent example

```
class my_agent extends uvm_agent;
  `uvm_component_utils(my_agent)

  my_driver my_driver_h;
  my_monitor my_monitor_h;
  my_sequencer my_sequencer_h;
  virtual my_interface my_interface_h;

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

  function void build_phase(uvm_phase phase);
    uvm_config_db#(virtual my_interface)::get(this,"","if", my_interface_h)
    uvm_config_db#(virtual my_interface)::set(this,"*","if", my_interface_h);
    my_driver_h = my_driver::type_id::create("my_driver_h", this);
    my_monitor_h = my_monitor::type_id::create("my_monitor_h", this);
    my_sequencer_h = my_sequencer::type_id::create("my_sequencer_h", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    my_driver_h.seq_item_port.connect(my_sequencer_h.seq_item_export);
  endfunction

  task run_phase(uvm_phase phase);

  endtask
endclass
```

2.3.6 UVM Sequencer

See [4, p. 401] for reference.

Most of the time sequencers don't have to be additionally configured to work properly. In most cases, it is sufficient to define a new object *my_sequencer* of type *uvm_sequencer* parametrized with the corresponding sequence item (listing 2.6), which is needed to configure the sequence-item port later. As shown before, the sequencer is instantiated inside the agent and its sequence_item_export is connected to the drivers sequence_item_port to communicate.

Listing 2.6: Minimal UVM sequencer example

```
typedef uvm_sequencer #(my_sequence_item) my_sequencer;
```

2.3.7 UVM Sequence

See [4, p. 362] for reference.

For the sequencer to provide transactions for the driver, a sequence of transactions is implemented (listing 2.7). This class creates objects of type *my_sequence_item*, randomizes (line 13) and provides them to the connected sequencer (line 14). As *uvm_sequence* extends directly from *uvm_object* instead of *uvm_component*, the constructor as well as the registry macro are slightly different. The sequence either consists of a hard coded chain of sequence items that are pushed sequentially to the sequencer, or only provides one single sequence item per call. This process is usually driven from the outside via a virtual sequencer and virtual sequences. These two constructs wrap all the agents' sequencers and sequences that are available inside the environment - this gives the convenient possibility to control all sequences inside one sequence object.

Listing 2.7: Minimal UVM sequence example

```
1  class my_sequence extends  uvm_sequence #(my_sequence_item);
2    `uvm_object_utils(my_sequence)
3
4    my_sequence_item tx;
5
6    function new(string name = "");
7      super.new(name);
8    endfunction
9
10   task body();
11     tx = my_sequence_item::type_id::create("tx");
12     start_item(tx);
13     assert(tx.randomize());
14     finish_item(tx);
15   endtask
16 endclass
```

2.3.8 UVM Sequence Item

See [4, p. 407] for reference.

The sequence item holds at least one fitting variable for each interface port to store either the next value for the driver or the recorded value of the monitor (as seen in listing 2.8). As *uvm_sequence_item* extends directly from *uvm_object* instead of *uvm_component*, the constructor as well as the registry macro are slightly different. All variables inside this class should be of type *rand* to enable randomization inside the sequences.

Listing 2.8: Minimal UVM sequence item example

```
  class my_sequence_item extends uvm_sequence_item;
    `uvm_object_utils(my_sequence_item)

    function new(string name);
      super.new(name);
    endfunction

    logic input_val1;
    logic input_val2;
    logic output_val1;

  endclass
```

2.3.9 UVM Driver

See [4, p. 366] for reference.
When provided with a sequencer, the driver's only purpose is to repeatedly request sequence_items over the sequence_item_port and drive the corresponding interface ports to the received values. It would also be possible to drive the ports to hard coded values with various constructs, but would defeat the purpose of an agile, scalable and reusable component. The minimal example 2.9, line 4 shows again the usage of the interface like seen in the testbench example, but with the preceding keyword `virtual`. This is always needed when transitioning from the static world (testbench) to the class based (drivers, monitors,..) - this keyword makes it possible to access the physical interface instantiated and connected in the testbench from inside a dynamic class via a reference or handle. This keeps the designer from referencing to the interface signals by name or path and thus creating a way more generic and stable environment. *The most common use for a virtual interface is to allow objects in a testbench to refer to items in a replicated interface using a generic handle rather than the actual name. Virtual interfaces are the only mechanism that can bridge the dynamic world of objects with the static world of modules and interfaces.*[5, p.279]

Listing 2.9: Minimal UVM driver example

```
1  class my_driver extends uvm_driver #(my_sequence_item);
2    `uvm_component_utils(my_driver)
3
4    virtual my_interface my_interface_h;
5
6    function new(string name, uvm_component parent);
7      super.new(name, parent);
8    endfunction
9
10   function void build_phase(uvm_phase phase);
11     uvm_config_db#(virtual my_interface)::get(this,"","if", my_interface_h)
12   endfunction
```

```

  task run_phase(uvm_phase phase);
    forever begin
      my_sequence_item tx;
      seq_item_port.get_next_item(tx);
      my_interface_h.slave.input_val1 = tx.input_val1;
      my_interface_h.slave.input_val2 = tx.input_val2;
      seq_item_port.item_done();
    end
  endtask

endclass
```

2.3.10 UVM Monitor

See [4, p. 364] for reference.

A monitor usually includes handles to the virtual interface it covers, as well as a *uvm_analysis_port* to transmit the recorded transactions to its connected subscribers. The connection of the monitor's *port* and the subscriber's *export* is usually done inside the environment class during connect_phase via the connect method (see listing 2.4). In the most basic way, a transaction is triggered as soon as one port of the connected interface toggles. More advanced monitors are written specifically for protocols or special operations. In this case, the monitor would look for combinations of signals on the interface that, for example, form a valid Advanced High Performance Bus (AHB) or System Power Management Interface (SPMI) transaction. If all the needed signal properties are found, the most vital information (e.g. address, data, flags,..) is bound to a sequence item and forwarded to a subscriber like a scoreboard or a register model adapter.

Listing 2.10: Minimal UVM monitor example

```
class my_monitor extends uvm_monitor;
  `uvm_component_utils(my_monitor)

  uvm_analysis_port #(my_sequence_item) aport;
  virtual my_interface my_interface_h;

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

  function void build_phase(uvm_phase phase);
    aport = new("aport", this);
    uvm_config_db#(virtual my_interface)::get(this,"","if", my_interface_h)
  endfunction

```

```
16   task run_phase(uvm_phase phase);
17     forever begin
18       tx = my_sequence_item::type_id::create("tx");
19       search_fork:
20       fork
21         @my_interface_h.input_val1;
22         @my_interface_h.input_val2;
23         @my_interface_h.output_val1;
24       join_any
25       begin_tr(tx, get_name());
26       disable search_fork;
27       #(sample_delay*1ns)
28       tx.input_val1 = my_interface_h.input_val1;
29       tx.input_val2 = my_interface_h.input_val2;
30       tx.output_val1 = my_interface_h.output_val1;
31       aport.write(tx);
32       end_tr(tx);
33     end
34   endtask
35 endclass
```

2.3.11 UVM Scoreboard

See [4, p. 365] for reference.

A scoreboard receives sequence items over the uvm_analysis_port (recognized at the DUT pin-level by a monitor) and covers their correctness, often by comparing the real data to a reference model (golden model). In reality, several agents will provide such items with various different variables and parameter. The scoreboards task is to gather all this information in the correct timing context and bring it to a common denominator to enable a prediction and comparison of the bigger picture. To ease this process, it is important that the monitors already bring the information of the transactions to a higher level of abstraction if possible in order to omit low level implementations in the scoreboard. The actual development of a scoreboard is highly dependent on the given DUT and agents implemented. For the bachelors project, a register model also had to be added and crosschecked with every possible register setting. The first line in listing 2.11 shows another usage of a uvm_macro: *uvm_analysis_imp_decl* declares an additional uvm_analysis_port to be used inside the scoreboard. This way one port for each connected agent can be created, parametrized with the corresponding sequence item. The subscriber, in this case the scoreboard itself, further has to implement the analysis_ports *write* method - as seen in line 21, listing 2.11. This method gets called each time the monitor has finished a transaction, as seen in line 31 of example 2.10. The input parameter to the function must be named *t* in order for the function call to work properly. It is up to the designer, how the process shall proceed from this point on. For the sake of simplicity, example 2.11 calls a function to check the correctness of the transaction immediately afterwards. A second way would be to dissociate the checking from the simulation progress by storing the transactions into first in - first out (FIFO) structures and process them independently to the simulation progress. The verification demands guide the way how the scoreboard is implemented.

Listing 2.11: Minimal UVM scoreboard example

```
`uvm_analysis_imp_decl(_my_port)

class my_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(my_scoreboard)

  uvm_analysis_imp_my_port #(my_sequence_item, my_scoreboard) my_export_h;
  my_sequence_item my_sequence_item_h;

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

  function void build_phase(uvm_phase);
    my_export_h = new("my_export_h", this);
  endfunction

  task run_phase(uvm_phase phase);

  endtask

  function void write_my_port(my_sequence_item t);
    my_sequence_item_h = t;
    my_check_function(my_sequence_item);
  endfunction;

  function void my_check_function(my_sequence_item tx);
    assert(tx.input_val1 & tx.input_val2 == tx.output_val1)
    else
      `uvm_error(get_type_name(), "DUT output does not match DUT inputs")
  endfunction
endclass
```

3 Universal Verification Component

To provide an understanding of the existing chip environment and still respect confidential and disclosure agreements, the following figures are highly abstracted representations of schematics and digital layouts.

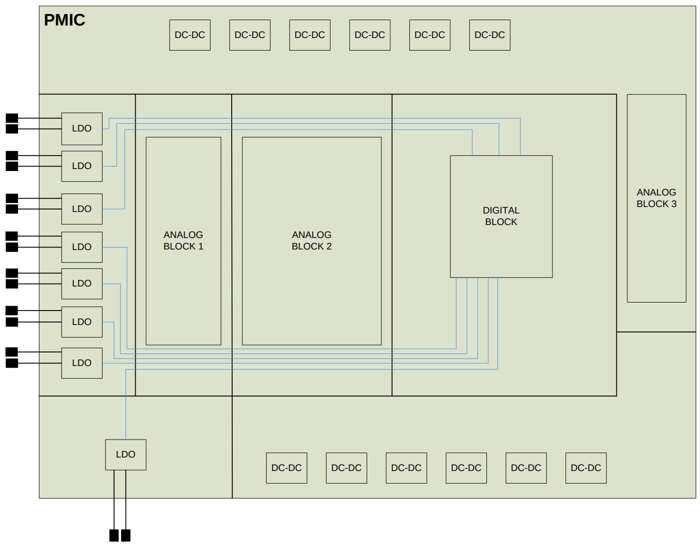


Figure 3.1: PMIC block diagram, top

The ongoing project at Dialog Semiconductor, that this bachelor's thesis was attached to, is a state of the art PMIC with several DC-DC converters, serial communication, LDOs, safety measures as well as I/Os - developed as an Application Specific Integrated Circuit (ASIC) for integration in mobile systems.
In the most basic form, figure 3.1 shows the top level block diagram of this PMIC. The chip is

split into analog components like LDOs, buck converters, Digital - Analog Converters (DACs), Analog - Digital Converters (ADCs) and sensors as well as a digital part responsible for controlling analog, communication, storage and more. The scope of this thesis is to create a digital module that represents the analog LDO's behavior and gives the ability to verify digital blocks with their interaction easier and faster. More precisely, the digital block on the chip includes a module that drives the analog components and their functionalities according to the current operating power state, register setting and digital inputs. Together with the LDO block, a verification environment for this digital module shall be created for module-, and in a further step integrated into top-level verification.

3.1 Analog LDO

To model the behavior of an LDO, the analog principles have to be understood. The following pages will introduce the general concept of low-dropout linear voltage regulators and the additional functionalities for this PMIC.

3.1.1 General LDO Functionality

An electronic device might need several different stable input voltages within certain ranges or configurable voltage sources for specific blocks. These sources are provided and managed by a PMIC, which gives control over the power distribution on such a platform. To generate different voltage levels from one main power supply, in the case of mobile devices mostly a battery, converters or regulators are used. The general rule of thumb is to use switching converters when the difference between supply and voltage output (dropout voltage) is high, and Low Dropout Voltage Regulators (LDOs), when the difference is small and a stable output voltage is needed. This is due to the way, how line regulators work:
LDOs give a simple possibility to regulate an output voltage from a higher input voltage. The difference between input and output is called dropout voltage and is mainly a consequence of the pass device's power consumption. For example, a 3.3V LDO with a dropout voltage of 300mV requires an input voltage of at least 3.6V to provide a stable output. Figure 3.2 shows a simplified diagram of an LDO, with a Field Effect Transistor (FET) used as pass device instead of a Darlington transistor stage, which leads to a much lower voltage loss.
The input voltage is applied to a pass element, which is typically an N-channel or P-channel FET, but can also be an NPN or PNP transistor. The pass element operates in the linear

region to drop the input voltage down to the desired output voltage. The resulting output voltage is sensed by the error amplifier and compared to a reference voltage.The error amplifier drives the pass element's gate to the appropriate operating point to ensure that the output is at the correct voltage. As the operating current or input voltage changes, the error amplifier modulates the pass element to maintain a constant output voltage. Under steady state operating conditions, an LDO behaves like a simple resistor.[6]
This means that with a higher difference in input to output voltage, the mentioned resistor/pass element needs to consume more power (thermal energy) and therefore decreases the efficiency of the LDO. Due to this, LDOs should be used for input-output differences near their minimal dropout voltage to provide the highest possible efficiency.

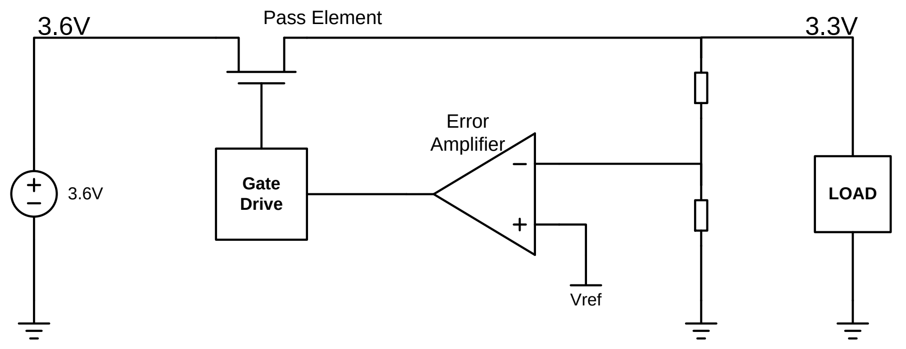


Figure 3.2: LDO principle functionality

3.1.2 Added Functionalities

In addition to the core principles, further features are added to implement either Design for Test (DfT) concepts or application specific requirements. A small subset of those will be presented here: Dynamic Voltage Control (DVC) and Bypass functionality. Generally, nearly all LDOs in the given project have a programmable output voltage that is configured by former mentioned digital block. The corresponding value is set via register access over a serial bus. This value is adapted for each LDO and passed as a digital signal to the analog component, which drives a variable resistor network in pull-down mode to the output voltage of the LDO (see figure 3.3). Most often, such a change in the voltage selection results in a step or step-like function of the output voltage. To avoid such behavior, DVC functionality creates a linear ramp between start and endpoint by stepping up the value at a choose-able rate. Most LDOs feature software bypass mode, which simply puts the LDO in dropout mode. In

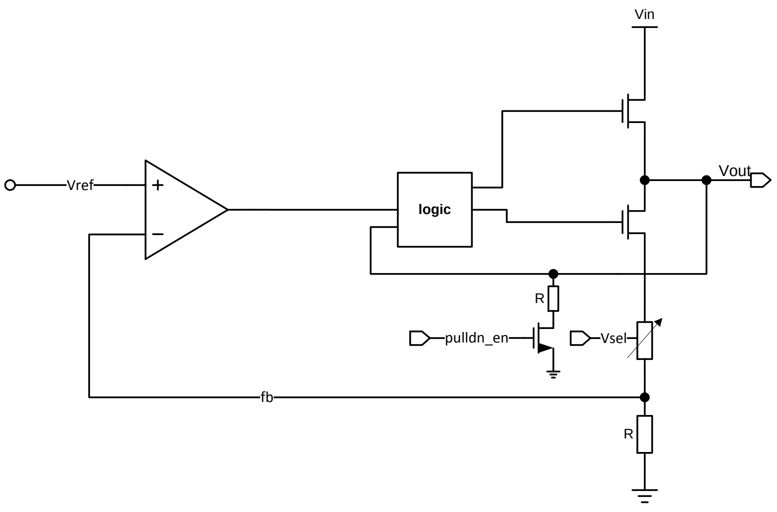


Figure 3.3: LDO project functionality

addition, some on the chip have hardware bypass options, where a switching process toggles between LDO mode and a fixed bypass input voltage. This further results in different current limit settings and safety concerns, as the uncontrolled, maybe noisy bypass voltage could cause damage to the consecutive components. Diagram 3.4 shows the basic implementation of such functionality.

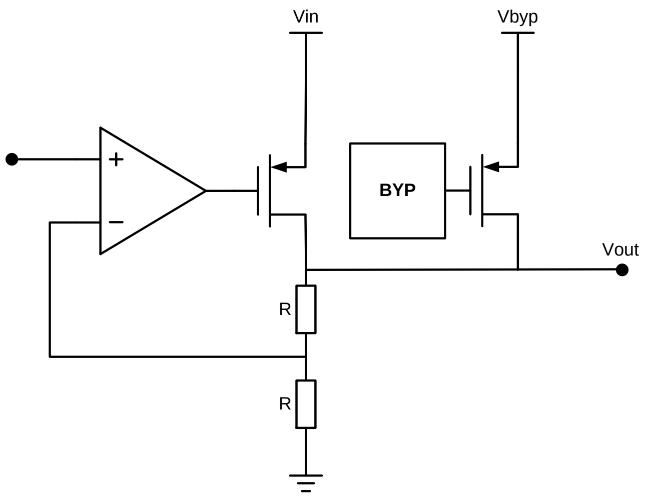


Figure 3.4: Bypass diagram

3.2 Digital Implementation

All given and possible functionalities of LDOs in the PMIC were implemented into one configurable SystemVerilog module. For each connected LDO in analog, one fully configured module instance is created inside the testbench and connected to the digital block driving the LDOs. To group all the signals or ports to the LDOs and back to digital, three interfaces per LDO module have been created:

Listing 3.1: LDO module header

```
module ldo_module (

ldo_dig_intf dig_intf,
ldo_config_intf config_intf,
ldo_ana_intf ana_intf

);
```

Digital Interface: All signals going to the LDOs or going from there to the digital block of the chip are captured here as SystemVerilog logics. This way, either stimuli directly from the digital module (connected inside the testbench), or from sequence generated items can be used to control the LDO module.

Analog Interface: This interface groups input, output and bypass voltage, attached output load and resulting output current as integers or reals. This way, the supply voltages as well as the attached load can be provided as sequence item stimuli during the test, which then influences the virtual output current of the LDO.

Configuration Interface: At project start, it was decided to use an interface for the module's configuration instead of input parameters. This way, the module doesn't have to be configured inside the testbench (module parameters have to be set at module instantiation, which happens inside the testbench file) but rather inside the test, which is way more flexible when it comes to different test scenarios or test runs with varying functionalities of the LDOs since the testbench usually doesn't change, but the individual tests do. Still, this configuration can only be set once per testrun and will be locked for the rest of simulation - similar to how an analog component would behave in real life.

Inside the module, the behavior is mainly described using *always*, *initial*, *always_comb*, *always_ff*, *always_latch* and *assign* blocks, where each keyword represents individual functionalities with defined boundaries and implications. The most important difference is between *initial* and *always*: The *initial* block is called only once at simulation time zero and its content is executed sequentially, where the execution can take as much time as required. In the case of the first block in listing 3.2, it is active as long as the signal *config_intf.setting_done* doesn't assert. As soon as this signal is triggered, the rest of the block executes in zero simulation time and the block becomes inactive until a new simulation is started. In contrast to this, the *always* block gets called instantly again as soon as it finishes, which makes it necessary to include a process that controls the execution of this block, like triggering or waiting for an event or a simple time delay. Without this control in execution, the simulation would be stuck inside this loop like it happens with an empty *while*() function in other programming languages.

To better specify the designer's intent of an *always* block, especially when modeling hardware, SystemVerilog introduced additionally the *always_comb*, *always_ff* and *always_latch* blocks. *Software tools do not need to infer from context what the designer intended, as must be done with the general purpose always procedural block. If the content of a specialized procedural block does not match the rules for that type of logic, software tools can issue warning messages. By using always_comb, always_latch, and always_ff procedural blocks, the engineer's intent is clearly documented for both software tools and for other engineers who review or maintain the model. Note, however, that SystemVerilog does not require software tools to verify that a procedural block's contents match the type of logic specified with the specific type of always procedural block.*[3, p.108]

Listing 3.2 further shows the initialization process happening inside the LDO module. Both *initial* blocks start executing at the same time - simulation time zero. The second block however is only executing to line 14, where it waits for the trigger config_intf . config_locked coming from the first *initial* block. The first block waits until the signal setting_done on the configuration interface asserts and then creates a local copy from the chosen settings to be independent of changing configuration inputs during simulation. After the success of this operation config_intf . config_locked =1 is set, which enables the further execution of the second block, which proceeds with “booting”the LDO and bringing it into idle mode. The signal config_done is further used throughout the design as an enable signal for various functionalities.

Listing 3.2: LDO initialization

```
1   initial begin
2     wait(config_intf.setting_done);
3     config_t = config_intf.config_t_i;
4     assert(!$isunknown(config_t));
5     config_intf.config_locked = 1; //locally copied - cannot be changed anymore
          from outside
```

```
  `uvm_info($sformatf("LDO_%02d", config_t.reference_number), "Configuration locked. ", UVM_HIGH)
end

int boot_time = 10; //ns

initial begin
  config_intf.config_done = 0;
  config_done = 0;
  @config_intf.config_locked;
  `uvm_info($sformatf("LDO_%02d", config_t.reference_number), "LDO booting ", UVM_HIGH)
  #(boot_time*1ns)
  config_intf.config_done = 1;
  config_done = 1;
  `uvm_info($sformatf("LDO_%02d", config_t.reference_number), "LDO running ", UVM_HIGH)
end
```

To show the implementation of an LDO functionality, the block for driving the output voltage of the LDO is displayed at codesnippet 3.3. Here, a combination of *always_comb* and *always*@(∗) had to be used. In fact, this is the only occasion inside the module, where another block than *always_comb* or *assign* has been used (where *assign* implements most of *always_comb*s functionalities and intents). This is due to the fact, that the *always_comb* represents purely combinational logic without any timing information. Therefore, no wait or trigger procedures can happen inside such a block and every signal can only be driven from one *always_comb* block which makes the code much cleaner and reduces errors by multiple drivers. The combination @(∗) after the *always* keyword in line 25 of listing 3.3 indicates an implicit sensitivity list for this block. The idea of sensitivity lists comes from former HDLs like Verilog or VHDL, where all signals included in this list will trigger a new execution of this block. If no trigger asserts, the block also doesn't execute, which is why no additional wait state or delay is needed inside such a block. An implicit sensitivity list takes all signals included in a block and triggers on them automatically. This is also what happens inside the expanded *always* blocks (like *always_comb*) without having to provide this additional option. The first block in snippet 3.3 assigns its output (`V_out`) immediately to changes of its inputs (`ana_intf.V_bypass, config_t.Vout_max, config_t.Vout_min, voltage_step, vsel, config_t.V_drop, config_t. bypass_enable_vsel_value , config_t. bypass_enable_vsel_value`). It is again to highlight on this example that the signal `V_out` cannot be written anywhere else outside this block and that all the mentioned inputs are automatically part of the blocks implicit sensitivity list. While the *always_comb* block just outputs the voltage variable based on its inputs, the second *always* block drives this value to the analog interface output, if no error conditions are met. In case of an error or fault condition, the output voltage is either immediately set to zero (appropriating the behavior of an attached pull-down resistor at the output) or slowly decreased over time (imitating voltage loss by leakage only). The second case and its time exposure is the reason why this block couldn't be realized with *always_comb*, which would force an execution in zero simulation time by definition.

Listing 3.3: LDO output voltage

```
//==========================CALCULATING OUTPUT==========
always_comb begin
  if(bypass_enabled === 1) begin //only ldo6
    V_out = ana_intf.V_bypass;  //v_bypass can be v_supply or V_HI
    if(V_out <= 0)
      V_out = 0;
    if(V_out > config_t.Vout_max) //to compensate lack of internal current
        sink
      V_out = config_t.Vout_max;
  end

  else begin
    V_out = config_t.Vout_min + voltage_step*vsel - config_t.V_drop; //check
        if that is correct
    if(V_out > config_t.Vout_max) //not needed with correct configuration,
        might stay tough
      V_out = config_t.Vout_max;
    if(V_out < config_t.Vout_min)
      V_out = config_t.Vout_min;
    if( vsel == config_t.bypass_enable_vsel_value) //bypass enable for ldo2,
        10 and 15
      V_out = config_t.Vout_max - config_t.V_drop;
  end
end


//=========================DRIVING OUTPUT===============
real ldo_ramp_down_slope = 0.9;
always @(*) begin //PULLDOWN LOGIC
  if(error_signal & bypass_enabled & (!dig_intf.sel_disable_pulldown_i) &
      dig_intf.en_active_discharge_i === 1 & config_t.has_pulldown === 1)
      begin //disabling with pulldown, only possible if byp enabled and fault
    ana_intf.V_out = 0; //do pulldown
    `uvm_info($sformatf("LDO_%02d", config_t.reference_number), "Shutting down
        with pulldown", UVM_HIGH)
  end
  else if(error_signal) begin
    `uvm_info($sformatf("LDO_%02d", config_t.reference_number), "Shutting down
        with leakage ", UVM_HIGH)
    repeat(1000) begin
      if(ana_intf.V_out < 30) begin
      ana_intf.V_out = 0;
      break;
```

```
        end
        #1ns ana_intf.V_out = ana_intf.V_out * ldo_ramp_down_slope;
        end
    end
    else
      ana_intf.V_out = V_out;
  end
```

All other given functionalities for LDOs are implemented in a similar way, trying to keep the logic as simple as possible while still featuring all possibilities. In the end, the *ldo_module* is able to run with a large number of different, feature based configurations to represent every different type of LDO placed on this PMIC within a single model. This module is the core of the LDO-UVC, which shall be the projects outcome.

3.3 Environment

As mentioned before, the module features three interfaces, which demands three agents with different functionalities for each. Those three are the only agents solely needed for the LDO module - to interact with the existing digital module, five more agents needed to be created. All agents get handles to their interfaces from the testbench via the configuration database and an additional configuration from each specific test. According to this information, the subcomponents, like monitors, drivers or sequencers of the agents are created and connected. In active mode, the agents' drivers stimulate the interfaces' ports according to the values received from transaction items via sequences through their sequencers. In passive mode, the drivers don't get instantiated at all and only monitoring is executed. This mode is an important vertical reuse feature from module to top level verification, where stimuli to a module won't come from verification agents anymore but directly from other modules connected. This way, only passive monitoring components are needed inside the agents.
Each monitor creates packages of information of the interface-values and transfers them to its analysis port, where a subscriber might be connected on the outside. This subscriber could be a coverage collector or scoreboard. In the case of this UVC, all monitors of all instantiated agents connect to a scoreboard, where their information packages (sequence items) are collected, analyzed and evaluated. Further, the register model, which is a representation of a subset of the registers available on the chip, is continuously updated by the information the parallel on-chip bus agents' monitor captures from connected bus interface. All those components are instantiated and connected inside the environment of the test, which also acts

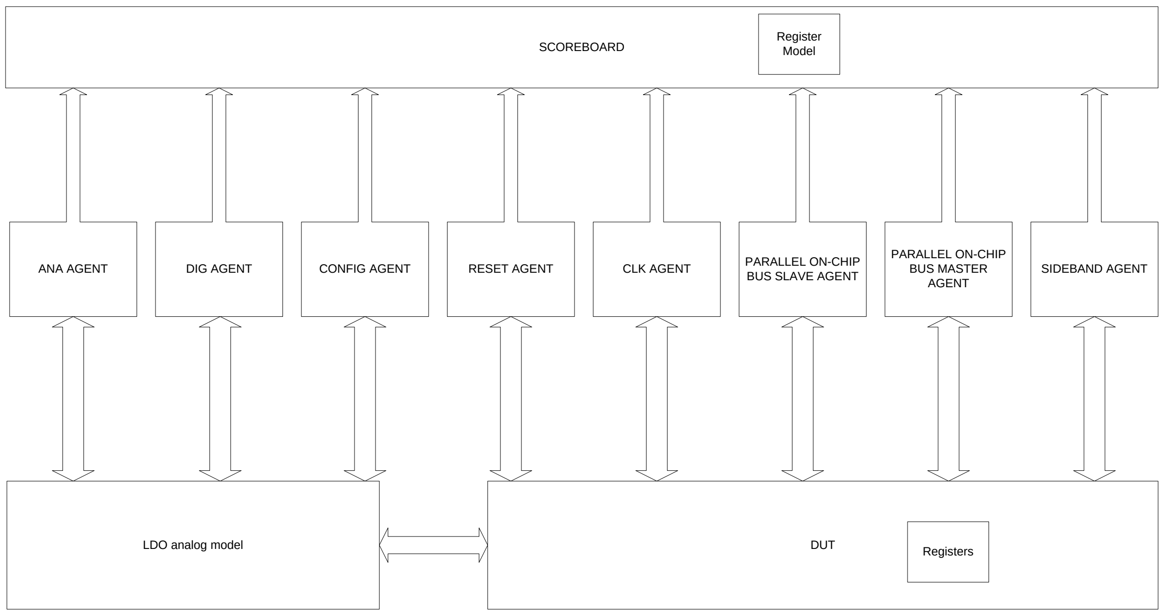


Figure 3.5: Agent-Scoreboard setup

according to the configurations coming from the test. This way, each test can run with different configurations of environment, agents or sequences to test the DUT in as many ways as possible.
In addition to the mentioned agents of the LDO module, a clock-agent, reset-agent, sideband-agent,parallel on-chip master-agent and parallel on-chip slave-agent contribute to each scoreboard of each LDO. The idea of having one scoreboard per instantiated LDO might not seem useful at first, but makes sense considering that the LDO modules don't interfere or interact with each other and that this way the amount of information per scoreboard is really limited to what it shall observe. Figure 3.5 shows exactly this implementation - again, it is to mention that only one instantiation for reset-, clock-, sideband-, parallel on-chip bus agents and register model exists per test environment but several of LDO module, LDO agents and scoreboards.

3.4 Verification

A first verification and validation of the created LDO module already happened when the newly created UVC was attached to the existing digital module, which actually should be verified in the end. At this point it already showed, which signals were implemented correctly

and which would cause errors or bugs. After a first smoke test and some test runs, the created module could be validated and the actual verification of the digital module (DUT/DUV) according to the existing specification was started. In order to do so, a generic scoreboard class, which fits each LDO configuration, was created and instantiated for each LDO module. Due to a lot of prerequisites for most functionalities, directed tests are the method of choice for a first verification. This way, each functionality block of every connected LDO is driven one by one by setting the according register settings and evaluated automatically by the connected scoreboard. Further, all listed functionalities of the DUV are driven at least once to show their correct behavior. In the end, error injection is done to assess the DUV's behavior when unpredictable or unlikely input values are supplied to the DUV's inputs.
To monitor these efforts taken, functional and code coverage is added to help understand which signals have already been triggered, which code blocks or functionalities might have been missed by a test and how efficiently the code is written.
As Dijkstra said: *"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."[7]*
It is a general dilemma of verification and testing, that there is no way to prove a hundred percent working and error free system - but with these measures taken, it is very likely that the created LDO-UVC is a good starting point for verifying this specific digital module, that is used throughout several projects in Dialog Semiconductors.

4 Conclusion

The UVC created using SystemVerilog and UVM is fully functional and ready to be used for module level verification of this specific digital module in several environments and use-cases. By providing the correct configuration, the ldo_module with its three core agents can further be used for various scenarios whenever a digital LDO model is needed, showing the implementation of horizontal reuse in digital verification.

Due to the measures taken, a switch to top level verification was the next logical step to fully take advantage of this verification component in the full chip level environment, following the concept of vertical reuse. The benefits of such an approach showed up immediately when the created UVC was embedded into the existing top level verification environment of this PMIC, where setting the UVC mode as uvm_passive and some minor adjustments in configuration were the only changes that had to be done. This way, a broad range of already existing top level tests could be used to verify the digital control module on chip level with the benefit of now having a highly advanced LDO model and automated verification measures in place.

At the time of finishing this project, another team of Dialog Semiconductor already showed interest in its outcome and proposed additional verification features that they would like to see implemented for their further use.

This shows that Dialog Semiconductor and especially my coworkers in Graz managed to not only provide me with a highly interesting and state of the art project, but also one that will actually be used by different teams.

Therefore, I would like to thank my manager Joao Martins, my daily advisor Michael Langreiter and my colleagues Daniel Kolednik, Naren Kesireddy and Michele Spelta, who were never tired of my questions and provided guidance with extensive knowledge in their fields.

Bibliography

[1] *Universal Verification Methodology (UVM) 1.2 User's Guide*, online, Accellera Std., 2015.

[2] P. P. Chu, *RTL Hardware design using VHDL.* Wiley Interscience, 2006.

[3] S. Sutterland, S. Davidman, and P. Flake, *System Verilog for design.* Springer, 2004.

[4] *Universal Verification Methodology (UVM) 1.2 Class Reference*, online, Accellera Std., 2014.

[5] C. Spear, *System Verilog for Verification.* Springer, 2006.

[6] M. Day, "Understanding low drop out (ldo) regulators," *Texas Instruments*, 2006. [Online]. Available: http://www.ti.com/lit/ml/slup239/slup239.pdf

[7] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, 1972.