UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEGREE PROGRAMME IN ELECTRONICS AND COMMUNICATIONS ENGINEERING

MASTER'S THESIS

INTEGRATION AND VERIFICATION OF PARAMETERIZED REGISTER INTERFACES

Author	Sami Huhtamäki
Supervisor	Jukka Lahti
Second reviewer	Juha Häkkinen
Technical advisor	Juuso Rantanen

2022

Huhtamäki S. (2022) Integration and Verification of Parameterized Register Interfaces. University of Oulu, Faculty of Information Technology and Electrical Engineering, Degree Programme in Electronics and Communications Engineering. Master's Thesis, 54 p.

ABSTRACT

This thesis takes an in-depth look on parameterized register models, their generation and use. The aim is to discover improvements to the current method of generating parameterized register models. The thesis is divided into two halves: a practical section that consists of a study on the generation of parameterized register models, and a theory section that supports the topics gone over in the practical section.

The practical section studied the generation flow and tools currently used at Nordic Semiconductor. The flow was analyzed to discover changes that would enable the generation of more flexible parameterized register models. The suggested changes were then used to generate a dynamic register model for a highly configurable intellectual property (IP) core. The register model was validated using a register test sequence and functional tests. Finally, the functionality of the generated register model was compared to a manually implemented model.

In the end, the test sequences and functional tests passed without errors. The generated register model could be configured directly from the testbench without editing the model manually. This also meant that the applied configurations would not be lost even if the register model were to be regenerated. The resulting register model was significantly more flexible than the previous generated models.

Keywords: parameterized register model, register model generation, register verification, UVM.

Huhtamäki S. (2022) Parametrisoitujen rekisterirajapintojen integrointi ja verifiointi. Oulun yliopisto, tieto- ja sähkötekniikan tiedekunta, elektroniikan ja tietoliikennetekniikan tutkinto-ohjelma. Diplomityö, 54 s.

TIIVISTELMÄ

Tässä opinnäytetyössä tutustutaan parametrisoituihin rekisterimalleihin, niiden generointiin, ja niiden käyttöön. Tavoitteena on löytää parannuksia nykyiseen parametrisoitujen rekisterimallien generointitapaan. Opinnäytetyö on jaettu kahteen puoliskoon: käytännön osuuteen, joka koostuu parametrisoitujen rekisterimallien tutkimuksesta, ja teoreettisesta osuudesta, joka tukee käytännön osuudessa käsiteltyjä aiheita.

Käytännön osuus tutki Nordic Semiconductorilla tällä hetkellä rekisterimallin generointiin käytettyjä prosesseja ja työkaluja. Niitä analysoimalla pyrittiin löytämään muutoksia, joiden avulla voisi generoida joustavampia parametrisoituja rekisterimalleja. Kyseisten muutosten avulla generoitiin sitten dynaaminen rekisterimalli IP lohkolle, joka sisältää paljon konfiguroitavia parametrejä. Generoitu malli varmennettiin rekisterien testisekvenssillä ja toiminnallisilla testeillä. Lopuksi rekisterimallin toiminnallisuutta verrattiin käsin kirjoitetun rekisterimallin toiminnallisuuteen.

Testisekvenssi ja toiminnalliset testit läpäistiin simuloinnissa lopulta ilman virheitä. Generoitu rekisterimalli oli konfiguroitavissa suoraan testipenkistä, eikä sitä tarvinnut muokata manuaalisesti. Tämä tarkoitti myös sitä, että testipenkissä asetettuja konfiguraatioita ei menetetä, jos rekisterimalli generoidaan uudelleen. Lopullinen rekisterimalli oli merkittävästi joustavampi kuin aikaisemmat generoidut mallit.

Avainsanat: parametrisoitu rekisterimalli, rekisterimallin generointi, rekisterivarmennus, UVM.

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION 9
2. SOC DESIGN AND VERIFICATION 10
2.1. SoC design flow 10
2.2. Levels of SoC verification 11
2.2.1. IP level verification 11
2.2.2. Sub-system level verification 12
2.2.3. Top level verification 12
2.3. Determining verification quality 13
3. REUSABILITY IN SOC DESIGN 14
3.1. IP-based design methodology and reusability 14
3.1.1. IP core reuse 15
3.1.2. Soft, firm, and hard IP cores 16
3.2. IP configuration using parameters 16
3.3. Reusable verification components 17
4. REGISTERS 19
4.1. Introduction to registers 19
4.2. Implementation of registers in an SoC environment 19
4.2.1. IP registers 20
4.2.2. Register banks 21
4.3. Register description standards 22
4.3.1. XML 22
4.3.2. IP-XACT 23
4.3.3. Register description using IP-XACT elements 24
4.4. Registers in SoC verification 26
4.4.1. UVM 27
4.4.2. UVM Register models 28
4.4.3. Register verification using UVM register models 32
5. PARAMETERIZED REGISTER MODEL IMPLEMENTATION 34
5.1. Register models and parameters 34

5.2. Approaches to parameterized register models 34
5.2.1. Static register models 34
5.2.2. Dynamic register models 35
5.3. Register model generation flow 36
5.3.1. Register descriptions 36
5.3.2. Current flow 38
5.3.3. Improved flow 40
5.3.4. Other possible changes 43
5.4. Dynamic register model experiment 44
5.4.1. The Chosen IP block 44
5.4.2. Generating the dynamic register model 45
5.4.3. Running structural tests 46
5.4.4. Running functional tests 47
6. DISCUSSION 48
7. SUMMARY 50
8. REFERENCES 52

FOREWORD

The goal of this thesis is to discover and validate ways to improve the way parameterized register models are implemented, reducing the amount of manual tinkering of generated register models. The study was done by analyzing the register description formats, tools, and scripts used at Nordic Semiconductor. This thesis was conducted at Nordic Semiconductor Finland Oy from March 2022 to December 2022.

I would like to thank Pekka Kotila for the opportunity to work on this study, and Juuso Rantanen for providing the topic and valuable guidance during the project. I also want to express my gratitude to all my colleagues at Nordic Semiconductor who assisted me throughout the project, especially Veli-Heikki Määttä, who was a great help when implementing the changes to the generation tool. Lastly, I want to thank Jukka Lahti of Oulu University for supervising this thesis.

Oulu, November 10, 2022

Sami Huhtamäki

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
C#	C Sharp
CPU	Central Processing Unit
DMA	Direct Memory Access
DUT	Design Under Test
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuit
IF	Interface
IP	Intellectual Property
RAL	Register Abstraction Layer
RC	Read Clears All
RO	Read Only
RS	Read Sets All
RTL	Register-Transfer Level
RW	Read, Write
SoC	System-on-Chip
SRAM	Static Random-Access Memory
TGI	Tight Generator Interface
TLM	Transaction Level Modeling
USB	Universal Serial Bus
UVC	Universal Verification Component
UVM	Universal Verification Methodology
VHDL	Very High-Speed Integrated Circuit Hardware Description Language
VIF	Virtual Interface
WC	Write Clears All
WCRS	Write Clears All, Read Sets All
WO	Write Only
WOC	Write Only Clears All
WOS	Write Only Sets All
WO1	Write Only Once
WRC	Write, Read Clears All
WRS	Write, Read Sets All
WS	Write Sets All
WSRC	Write Sets All, Read Clears All
W0C	Write 0 to Clear
W0CRS	Write 0 to Clear, Read Sets All
W0S	Write 0 to Set
W0SRC	Write 0 to Set, Read Clears All
W0T	Write 0 to Toggle
W1	Write Once
W1C	Write 1 to Clear
W1CRS	Write 1 to Clear, Read Sets All
W1S	Write 1 to Set
W1SRC	Write 1 to Set, Read Clears All

W1T	Write 1 to Toggle
XML	Extensible Markup Language

1. INTRODUCTION

Registers are an integral part of integrated circuits. They are used to store data, control and observe the operation of various systems and sub-systems, and to act as an interface between software and hardware. Registers are widely used in today's integrated circuit (IC) designs, and each design typically contains a large number of registers for different purposes.

As the number of registers contained in a design keeps growing larger and larger, the complexity of register verification starts to become an issue. To meet the required production times, the industry has adopted various tools and standards to increase the efficiency of verification. One of these tools is the Universal Verification Methodology (UVM) register model [13].

UVM register models provide an efficient way of accessing and observing design registers. They consist of a set of base classes that model the corresponding design registers. Register models are typically generated from a textual register description, which describes the properties of each register and the block they are contained in. The UVM register models can also be written manually, but the process can be very tedious and prone to errors due to the number of registers.

Another common method the industry has adopted to increase productivity is the use of reusable design and verification components. To improve the efficiency of verification, verification components are implemented with reusability in mind, allowing them to be used again in other projects or even in the same project. One aspect of reusable design is the use of configurable parameters. They increase the flexibility of the component, so that it can operate in different environments and situations.

Challenges arise when these two tools, register models and configurable parameters, are used in conjunction with each other. Main issue is the handling of register parameters when the register model is generated from a text-based register description. In the case of complex designs that contain numerous parameters, the generated register model usually has to be edited manually to contain the required parameters. When the register model must be regenerated, the manual editing process must also be repeated, significantly lessening the productivity gains received from automatic register model generation.

The goal of this thesis is to discover and validate ways to improve the way parameterized register models are implemented, reducing the amount of manual tinkering of generated register models. This thesis is done from Nordic Semiconductor point of view, meaning this thesis focuses mainly on parameterized register model related issues present at Nordic Semiconductor.

Chapter 2 provides a brief overview on the System-on-Chip (SoC) design flow. It goes into more detail on SoC verification and its different levels. Additionally, it describes how the quality of the verification is determined.

Chapter 3 discusses the basic IP-based design methodology and explains the concept of reusability. It also goes into more detail on how configurable parameters are used to increase the flexibility of design and verification elements.

Chapter 4 of this thesis provides an overview of registers. It includes some common information on registers, how registers are implemented in SoC environments, how they are described using register description standards, and how they are verified. The chapter also provides a more in-depth view on UVM register models.

Chapter 5 is the main practical section of the thesis. It describes the found issues related to parameterized register models, and documents the possible improvements, their implementation and validation.

Chapter 6 discusses the results of this thesis, alternative solutions, and possible future work.

2. SOC DESIGN AND VERIFICATION

As circuits are becoming increasingly complex and expensive to produce, the industry has begun to embrace new design and reuse methodologies that are collectively called system-on-chip (SoC) design methodologies [22]. SoCs are typically implemented using IP cores, also referred to as IP blocks, and IP sub-systems, which are designed to handle specific functions using several IP cores.

IP cores include both custom-made, and pre-designed and pre-verified third-party cores. They may include embedded processors, memory blocks, interface blocks, analog blocks, and blocks that are designed to handle specific functions [22]. IP cores are integrated to the SoC by defining the connections between them, implementing various design-for-test (DFT) techniques, and by verifying and validating the system-level design as a whole.

This chapter describes the design flow of IP-based designs and explains the different levels of integration. Additionally, methods for determining the quality of the verification are gone over briefly.

2.1. SoC design flow

The classic top-down SoC design flow begins with specification and division of the design into IP blocks, and ends with integration and verification. First step is to write complete specifications for the system or sub-system being designed. Next, its architecture and algorithms are refined, and the architecture is decomposed into well-defined IP blocks. The IP blocks are then designed and verified or selected from pre-designed blocks [24].

When the IP blocks are finished and their test coverage is sufficient, they are integrated into the next level of integration, for example, sub-system or top. Next, the system/sub-system level verification is performed, and the timing is checked. The designed system or sub-system is then delivered to the next higher level of integration; on the highest integration level this is the tape-out of the design. Finally, all the aspects of the design, including functionality and timing, are verified as a whole [24].

The top-down methodology assumes that the lowest level specified blocks can be implemented, but it can often turn out to be unfeasible, in which case the whole specification process needs to be repeated [24]. Hence, the design methodology is a combination of bottom-up and top-down philosophies. The methodology is based on hardware-software codevelopment while simultaneously considering physical design and performance [28]. An illustration of this flow can be seen in figure 1.

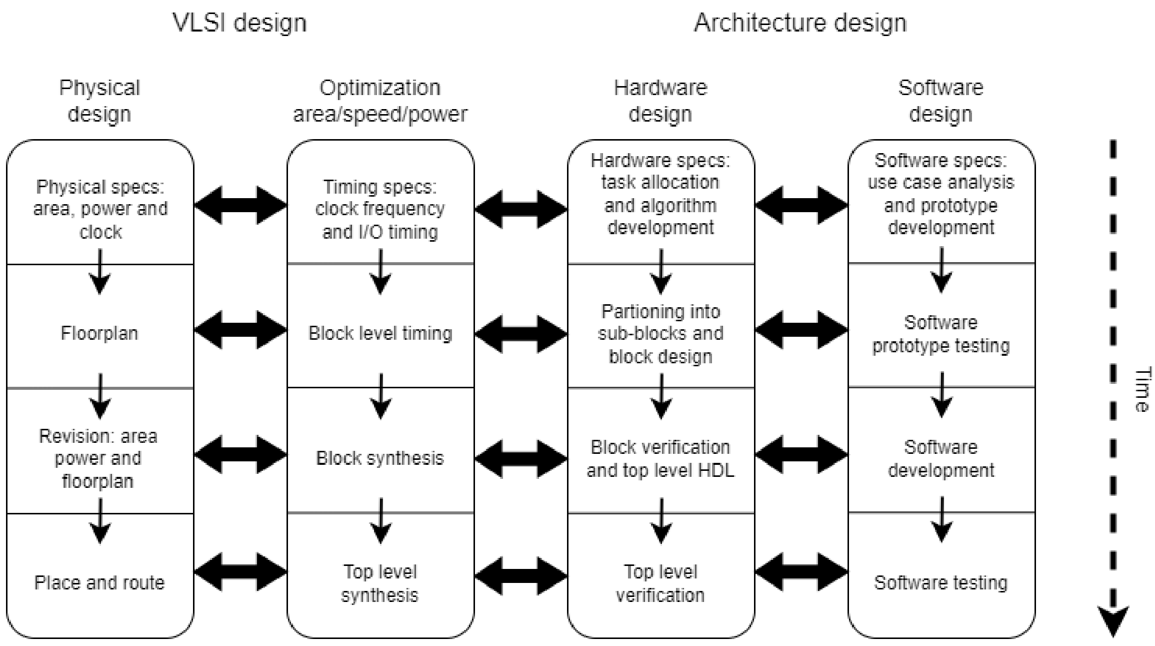

VLSI design
Architecture design
Physical design
Optimization area/speed/power
Hardware design
Software design
Physical specs: area, power and clock
Floorplan
Revision: area power and floorplan
Place and route
Timing specs: clock frequency and I/O timing
Block level timing
Block synthesis
Top level synthesis
Hardware specs: task allocation and algorithm development
Partioning into sub-blocks and block design
Block verification and top level HDL
Top level verification
Software specs: use case analysis and prototype development
Software prototype testing
Software development
Software testing
Time

Figure 1. Interlaced codevelopment design methodology.

2.2. Levels of SoC verification

One basic practice with solving large problems is to break them into smaller, more manageable challenges. In SoC verification this is done by using the existing hierarchical structure of the design. Instead of verifying the whole design at once, the verification work is done in parts, starting from the smaller building blocks of the system before moving up to the larger structures [30].

SoC verification can generally be divided into three levels of integration: IP level, sub-system level and system/top level. First the functionality of the individual IP cores is verified. When the IP core functionality is verified with a sufficient test coverage, testing can be started on the sub-system level. On the sub-system level, the interfaces and transactions between IPs and the overall functionality of the sub-system are verified. Top, or system, verification is the highest level in the verification hierarchy, and it comprises the verification of the entire design. The following sections describe the verification process on the different levels of abstraction.

2.2.1. IP level verification

The goal of IP level verification is to ensure that the IP core is correct in its functionality and timing. In comparison to single use IP cores, the verification of reusable IP cores presents additional challenges. The goal of the verification should be to have zero defects, because the reusable IP can have a wide range of different applications from consumer electronics to mission-critical aerospace applications. It should cover all legal configurations of the IP, and all legal values of its parameters. Additionally, to increase the efficiency of the verification process, the verification components should be made reusable, so they can be used again in sub-system and top level verification environments. The reusability also allows the components to be used again if the IP is substantially redesigned in the future [24].

Achieving a high level of test coverage can be a daunting task, and it requires a significant investment in developing a robust verification environment and a complete set of tests. A wide range of tools and techniques are used in IP verification: verification IPs, high-level verification languages, automated regression tests, tools for checking code coverage, emulators, and others [24].

2.2.2. Sub-system level verification

When the IP cores have been tested with sufficient coverage, they can be integrated into the higher level of hierarchy in the design [24]. Sub-systems are typically formed by several IP cores that are focused to perform a specific function of the overall design. Sub-system verification focuses on testing the interfaces between the IP cores, and the overall functionality of the sub-system.

Interface verification comprises of transaction verification and behavioral verification. Interface verification begins by checking all the transaction types that can occur at each interface. Then the behavior of the IP cores is tested to different data values in the transactions. Thus, it is preferable to use a common communication architecture to minimize the complexity of interface verification [24].

2.2.3. Top level verification

Top, or system, level verification is the final level of verification before tape-out. After the functionality of individual sub-systems and IP cores is verified, their interfaces and interconnectivity are tested on the top level. Top level verification also consists of the functional testing of the entire system and software testing [24]. Top level testing requires a lot of time and effort, which is why it should be started as soon as possible.

Top level interface verification is like the interface verification on sub-system level but is more complicated due to the larger scale of the verification environment. Once the basic functionality of the system has been verified by checking the transactions and interconnections between sub-systems and IP cores, system verification consists of exercising the entire design. The goal is to test the system as it is intended to be used. This means running software applications on the design to get as accurate representation of the design's final operation as possible [24].

Application code-based verification is essential for guaranteeing the quality of the design, but it also presents major challenges for verification teams. It is not feasible for conventional simulation to execute the millions of vectors required to run even the smallest fragments of application code, let alone to boot an operating system. This problem can be addressed by increasing the level of abstraction so that software simulators can run faster, or by using specialized hardware for performing verification, such as emulators or prototyping systems [24].

2.3. Determining verification quality

The estimation of verification quality is very important when integrating IP cores to higher design hierarchies, preparing a design for tape-out, and determining when functional verification is complete. There are several common methods used to estimate the quality of functional verification. They each use different approaches to achieve the best possible overall measurements. Five common methods are: code coverage, functional coverage, fault insertion coverage, bug tracking, and design and test plan reviews. Code coverage is a simulation-based method that in its simplest form counts which lines of register transfer layer (RTL) have been executed during verification. Functional coverage checks that all valid states of a specific block of code have been verified. Fault insertion coverage inserts a bug and checks if the test catches it. It uses a set of bugs to measure test case quality. Bug tracking is used to count all found bugs, and to correlate them to the RTL. Design and test plan reviewing is used to compare the verification test plan to the architectural and implementation specifications. It checks that all components of the specifications have a measurable verification item, such as a test or an assertion [37].

The use of configurable parameters increases the complexity of verification significantly. Ideally, each functional coverage item would be tested against each specific parameterization. As the coverage space grows exponentially, and additional parameters are added, this comprehensive method quickly becomes unfeasible [34]. Crossing two design parameters, each with two different values, would lead to four different parameterizations, which means four times the amount of functional coverage space.

A couple of options exist to ensure adequate functional coverage in various parameterizations. The first step is to cover each parameter setting without crossing them. This verifies that all parameter values have been tested. Then, only important parameters, or parameters that have a direct effect on another should be crossed. Finally, specific functional coverage items should be crossed with parameters that affect that functionality [34]. Parameters and configurable IP blocks are gone over in more detail in the following chapters.

3. REUSABILITY IN SOC DESIGN

This chapter examines the challenges caused by the increasing complexity of SoC designs, and explores different methods of increasing designer productivity, such as IP reusability and parameterization.

Reusability is a key aspect of today's SoC designs. Solutions are required to increase designer productivity and to keep development times and costs under control. When IP cores are built with reusability in mind, integration engineers can work more efficiently. To make IP cores usable for a broad range of different applications, they are designed to be configurable, for example, using parameterized registers.

3.1. IP-based design methodology and reusability

Integrating more and more functionality on a chip has been a constant trend, as predicted by the Moore's Law. It predicts that the number of transistors on a chip will double every 18-24 months. One of the key challenges in SoC development is to increase designer productivity to keep up with the complexity increase predicted by the Moore's Law. For that reason, today's notion of SoCs is defined in terms of overall productivity gains through reusable design and integration of components [22][5]. Figure 2 illustrates the annual growth rate estimates of chip complexity (58%) and designer productivity (21%). The increasing separation between the two lines is referred to as the productivity gap.

Three approaches have been proposed to solve the productivity gap problem: platforms, synthesis, and reuse. In the platform approach, semiconductor vendors provide a universal pre-designed SoC platform that contains one or more processors and several peripherals for handling different input/output (I/O) protocols and encoding/decoding functions. In the synthesis approach, the SoC chip is synthesized from a high-level functional description using a common language such as C. Finally, in the reuse approach, the design is assembled using many different IP cores from internal and/or external sources [27].

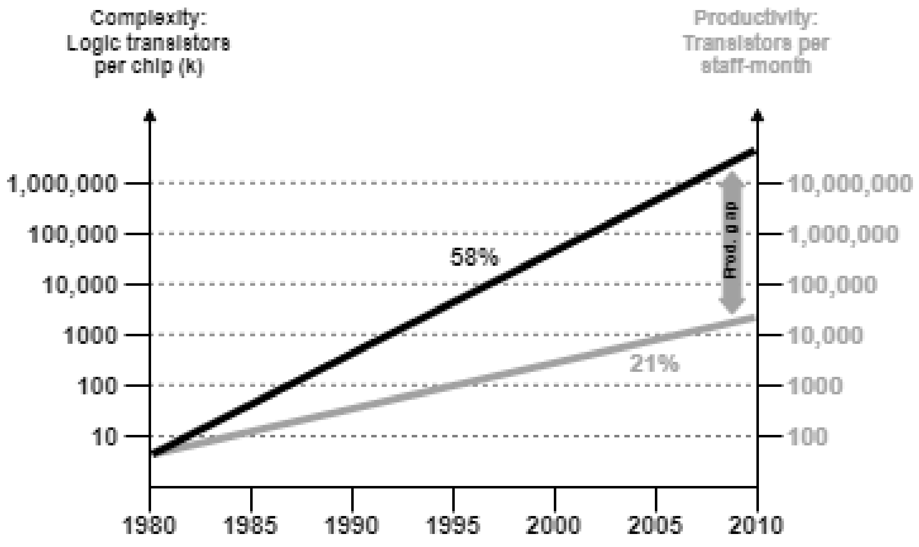


Figure 2. Productivity gap.

3.1.1. IP core reuse

Reusability is a central aspect of today's chip design. It can significantly increase designer productivity by enabling the use of previously designed and verified components. Hence, the use of reusable IP cores is a necessity for efficient implementation of SoC designs [23]. Reusable IP cores allow the designer to optimize the design for a specific application by mixing and matching different IPs. The development of reusable IP cores, however, typically takes more time and effort compared to a single-use IP.

IP core reuse comprises both, design reuse and verification reuse. Design reuse is the use of previously designed functional components for different projects, or different roles of the same project. Verification reuse means the use of previously implemented verification environments and components.

In the case of IP reuse, the terms "vertical" and "horizontal" are used to describe the context into which the IP is reused. In the case of vertical reuse, the IP is used again on a different level of design hierarchy. One example of this is the use of verification elements from IP level to sub-system and system levels. Horizontal reuse can be used by including pre-designed and pre-verified IP cores in other projects in roughly the same level of abstraction, but it can also be used to implement several different instances of the same IP in the same project [31] [30].

To support as many applications as possible, and provide the highest reuse benefits, IP cores should include a few key features: they should be configurable to meet the requirements of many different designs, and they should be compliant with defensive design practices to facilitate timing closure and functional correctness. Additionally, they should have standard interfaces and a complete set of deliverables to facilitate integration into a chip design [24].

IP cores have to be configurable to meet the needs of various applications. If their possible applications are too limited, it is usually not worth making the investment to make them reusable. For example, interface blocks like Universal Serial Bus (USB) may support multiple configurations and multiple interfaces for different physical layer interfaces [24]. IP core configuration includes the use of parameterized registers to customize the IP operation. Configurability is important for the usability of an IP, but it also poses additional challenges since it makes the core harder to verify [24].

Defensive design practices are design and coding practices for making the later parts of the design flow easier, including timing closure, verification, and packaging for reuse. These practices ultimately come down to one key principle of good engineering: keep the design as simple as possible [24].

Reusable IP cores should adopt industry standard interfaces rather than unique or core specific interfaces whenever possible. This enables the integration of various cores without having to build custom interfaces between them and the rest of the chip [24].

A complete set of deliverables includes the synthesizable RTL, the verification IP for verifying the core individually and for top-level verification, the synthesis scripts, and the documentation of the IP. They enable smoother integration IP cores, and significantly increase their reusability [24].

Reusability is the most effective when the reused IP core is pre-designed and pre-verified, and the integration engineer can integrate it into the SoC without large modifications. However, in practice, IP reuse usually is not as effective as intended. Integrating reusable IP cores often requires additional work from the integration team because of issues surrounding IP quality.

This is because the cost and time required for designing a "completely reusable" IP is too high. The IP cores are usually designed for a specific product, and due to schedule pressure, the designer has to optimize it targeting the chip specific goals, sacrificing reusability. Hence, "complete reusability" is rarely achieved. [26].

3.1.2. Soft, firm, and hard IP cores

There are three main forms of reusable IP architecture in terms of flexibility and silicon optimization: soft, firm, and hard. These forms and their relations are shown in figure 3. Soft IP blocks are specified using RTL or higher level descriptions. This form is flexible, portable, and reusable, but it does not have guaranteed timing or power characteristics because the implementation in different processes and applications produces variations in performance [22].

Hard IP blocks have fixed layouts and are highly optimized for a specific application. The advantage of hard IP blocks is that they have predictable performance. Their downsides are the higher cost and effort, and the lack of flexibility, portability, and reusability. This form of IP is usually prequalified, meaning it has already been tested by the provider [22].

Firm IP cores are provided as parameterized circuit descriptions so that they can be optimized for various design needs. Firm IP offers a compromise between soft and hard IPs, being more predictable than a soft IP and more flexible than a hard IP [22].

Today, memory cells are designed at the transistor level, and memory arrays are tiled from these cells using a compiler. Analog blocks, such as digital-to-analog and analog-to-digital converters, are designed at least partially at the transistor level. For the most part all other digital designs, however, start out as soft IP, and the RTL is considered as the golden reference. Synthesis, placement, and routing are then used to map the RTL to gates [24].

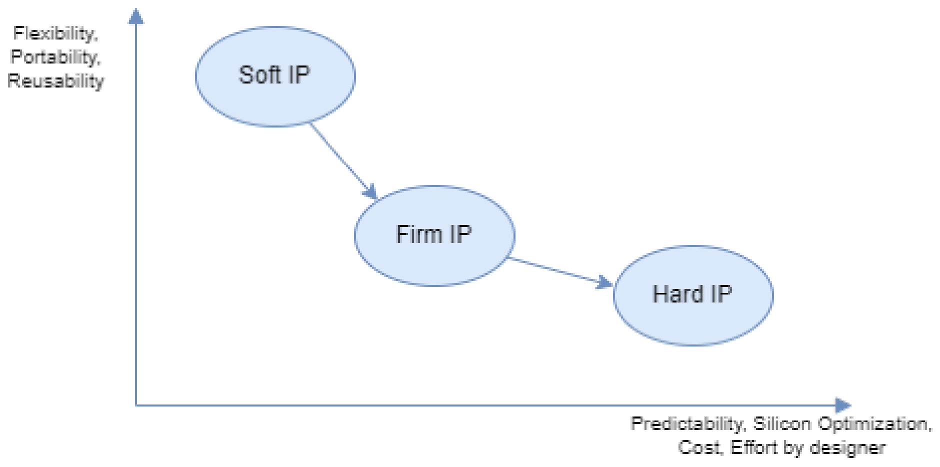

Flexibility, Portability, Reusability
Soft IP
Firm IP
Hard IP
Predictability, Silicon Optimization, Cost, Effort by designer

Figure 3. Different types of IP blocks.

3.2. IP configuration using parameters

The RTL descriptions of reusable IP cores are typically configurable, meaning that they contain user-definable parameters to tailor the core to the requirements of different applications. In the case of a processor core, the configurable parameters may include, for example, the bus width, number of registers, cache sizes, and instruction set. Such flexibility provides improved performance, and lower area and power, since unneeded features can be removed [22]. Today's IPs are often highly parameterized; large SoCs may contain hundreds of thousands of configurable parameters.

There are two main types of parameters: static and dynamic parameters. A static parameter is set before instantiating the IP core, meaning they will be fixed in the fabricated SoC. They are used to specify various structural or behavioral aspects of an IP, for example, bus widths, register field widths, or timing properties.

A dynamic parameter is set in an already instantiated IP core, and it is typically implemented as a part of the IP. Since the parameters are set after IP core instantiation, they are also run-time configurable [25]. They are typically configured by software but may also be handled by other hardware. Dynamic parameters are used to configure the functional operation of the IP core. Configuration registers are an example of dynamic parameters.

Configuration registers are typically IP specific and affect its operation in various ways. They can be accessed during run-time to, for example, start and stop processes, set operation modes, and enable interrupts. The number and properties of configuration registers can also be parameterized using static parameters, in which case there are two levels of parameterization. The parameterizable register properties include, for example, access policies, addresses, field sizes, and reset values [36].

Additionally, parameters can be classified by their level of abstraction, independent of whether they are static or dynamic. The three levels are circuit, architecture, and application. Circuit level parameters keep the same general logic structure when set but make small modifications to how the information bits are stored and transferred. Architecture level parameters can significantly reconfigure the architecture of the system. Application level parameters can change the operation of the system in a non-essential way [25].

Each of the parameters of an IP should be defined in the functional description of the IP. The description should include the parameter name, legal values, default value, and a description of its function. In addition, any dependencies to other parameters should be described [24].

3.3. Reusable verification components

Verification process is typically the most difficult and time-consuming part of the overall design [24]. In many cases, between 40% and 70% of the entire effort of a project is spent on verification. This high level of required effort indicates the possible gains to be made with successful reuse are significant. Reuse may occur between projects or within the same project [37]. Vertical reuse of verification components is a common example of reuse within the same project.

Without careful planning, the reused verification components are not likely to fit together well in the higher-level verification environments. To reuse verification components between the different hierarchy levels, it is important to plan the structure of each environment so that components can be shared. This means that the verification environment structure of each level, IP, sub-system, and top, must be designed so that there is a high degree of overlap of testbenches and test cases. Component modularity and high abstraction are also key aspects of successful reuse of verification components [37].

In the case of top level verification, the verification requirements cannot be fulfilled by looking only at the external pins in a complex top level chip. This means that verification teams need to measure coverage and check operation of critical functional data paths inside the DUT including validating relationships between IPs while running top level test cases. An additional advantage of reusing the IP level verification components in the top level is that it can test the operation of the IP more accurately. Often a core that is comprehensively validated in a stand-alone system finds itself under different and unexpected conditions in the top level environment. Thus, vertical reuse can sometimes lead to improvements in the IP level verification [31].

When testing IP cores in sub-system or top levels, the reused verification components are typically passive, meaning they themselves do not generate any stimulus. In those cases, verification stimulus is provided by active components of the verification environment outside the scope of the reused IP level environment [31]. A passive IP level verification environment is illustrated in figure 4, and a top level environment can be seen in figure 5. In figure 5, the functionality of IP block A is verified using the reused verification environment shown in figure 4.

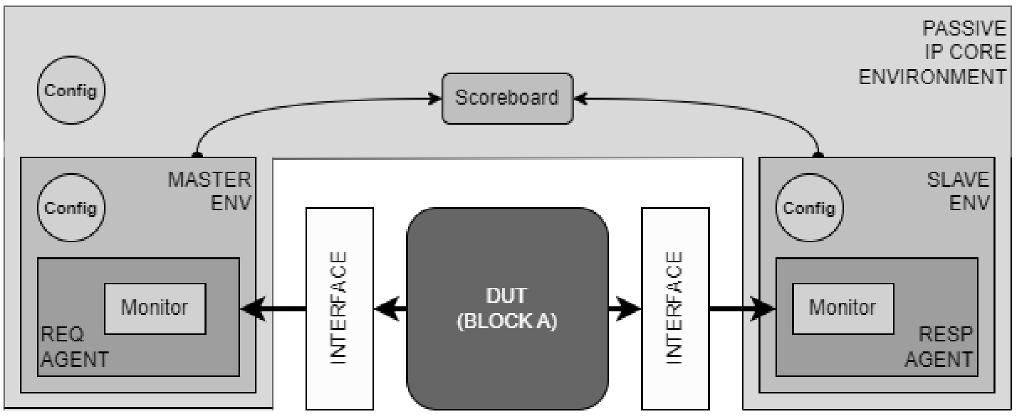


Figure 4. Passive IP level test environment.

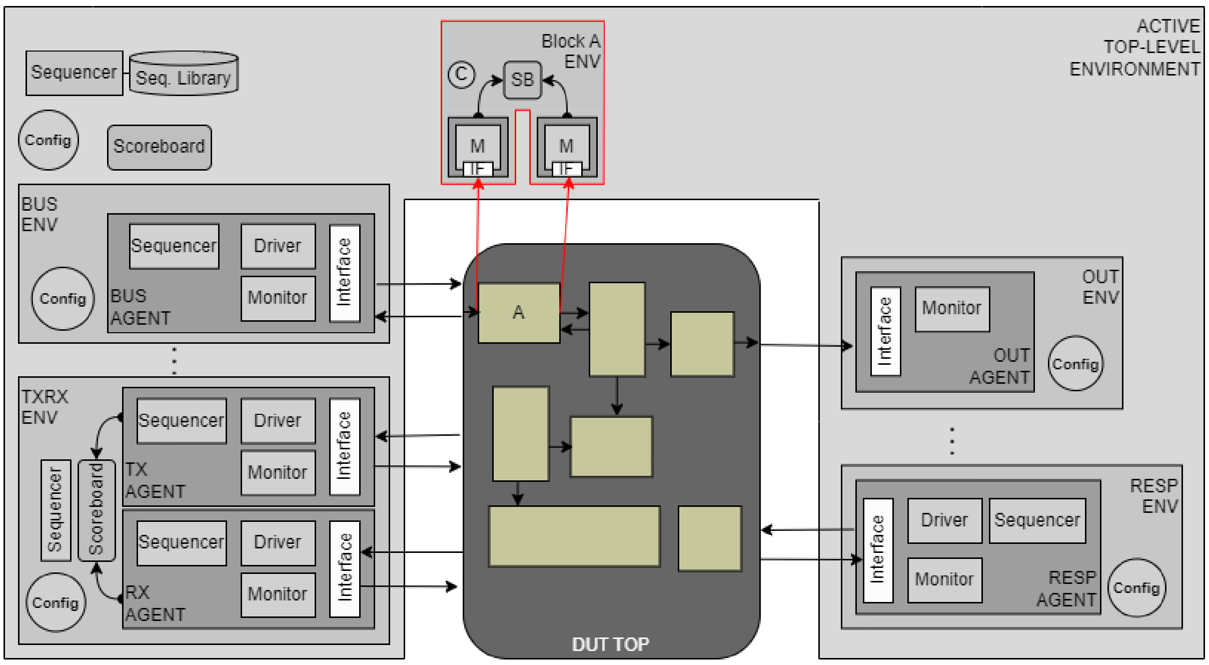


Figure 5. A reused IP level environment in top level environment.

4. REGISTERS

This chapter provides a general description of registers, and the ways they are used in SoC designs. Section 4.1 gives an overview on registers and section 4.2 describes more specifically the key registers and their properties from SoC development point of view. Section 4.3 provides an explanation on register description standards and gives a more in-depth view on IP-XACT. It also gives a general explanation of how registers are described using IP-XACT elements. Finally, section 4.4 introduces UVM and register models, and describes how they are utilized in verification. It also describes register verification using register models.

4.1. Introduction to registers

Digital designs contain many functional blocks for performing specific operations. If the operations are performed once a specific command or a clock pulse is given, the blocks are called synchronous, or clocked. Registers are an example of a clocked functional block [1].

Registers are an important component of digital systems. They are primarily responsible for momentary storage of small amounts of data and can be implemented in wide variety of ways. Registers can be implemented using individual flip-flops, standard static random-access memory (SRAM), register files, or high-speed core memory. Depending on the register, data can be transferred in and out of the register in parallel or in series. In the parallel case each of the data bits is transferred at once, and in the series case, the data is transferred one bit at a time.

There are several different types of registers, which can be divided to processor registers and hardware registers. Processor registers can be further classified according to their content or instructions that operate on them. Some of the most common user accessible registers are data registers, address registers, and general-purpose registers. Processors also contain internal registers that are no accessible by instructions, such as instruction registers, memory buffer registers, and memory address registers.

Hardware registers occur outside central processing units (CPU), and they often have similar characteristics as memory, such as the ability to read and write multiple bits at a time, and the use of addresses to select particular registers. They are usually implemented as components of peripheral devices that are connected to the CPU using a bus. The most common uses for hardware registers include configuration and start-up of different features, data buffering, status reporting, and functioning as different kinds of inputs and outputs.

4.2. Implementation of registers in an SoC environment

Hardware registers have a great importance in SoC designs. They are used to configure and control various features that are separate from the CPU. The registers work as an interface between software and hardware. Software is able to perform write and read accesses to them, transferring data between the peripheral devices and the CPU.

To enable this communication, IPs are connected to a bus, and the internal register banks and registers are assigned addresses in the bus address space. Register banks group together a number of registers that are referred to by adding an address offset to the base address of the register bank.

4.2.1. IP registers

An SoC design can contain several different IP cores and memory blocks, and the CPU requires a way to access and control them. A common way to interconnect IP cores and memory blocks with a processor is to use buses.

A bus is a digital communication mechanism that allows two or more functional units to transfer control signals and data [3]. Typically, a bus consists of a number of connections running together, called bus lines. These lines can be grouped together for carrying different signals, such as address, data, and control signals [4]. Figure 6 shows an example, where two IP cores and one memory block are connected to the CPU using a bus. Because buses are typically shared by several IPs, buses also have to be defined an access protocol. The access protocol specifies how an IP can determine whether the bus is available or is in use, and how the attached units take turns using the bus [3].

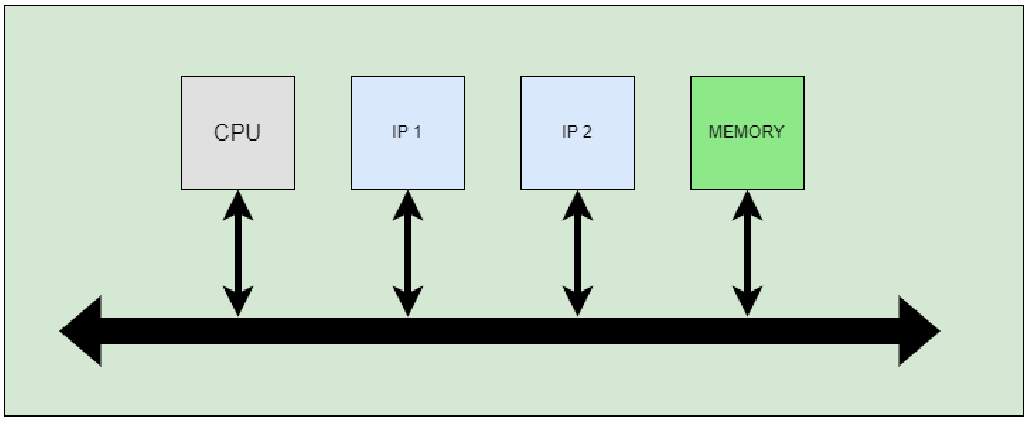


Figure 6. Bus example.

After connecting the processor to IPs through a bus, it can then control them by dealing with their internal registers as if they are regular memory locations. Each of the IPs and their registers are assigned addresses, after which the bus masters, such as the CPU or a direct memory access (DMA), can configure them by performing write and read accesses to those addresses [4].

As can be seen from figure 7, IPs require additional logic to enable the transactions between the bus master and the IP. Data, address information, and control signals must first pass through the bus interface. Next, the signals are passed to the decoder that decodes the address and selects the matching register. It also analyzes the control signals to determine if the incoming transaction is a write or read transaction. If the transaction coming from the bus is a write transaction, the received data is written to the selected register, which is then available to the IP logic. In the case of the read transaction, the read data multiplexer reads the data from the selected register or from the IP logic, and it is then transmitted to the bus through the bus interface [7].

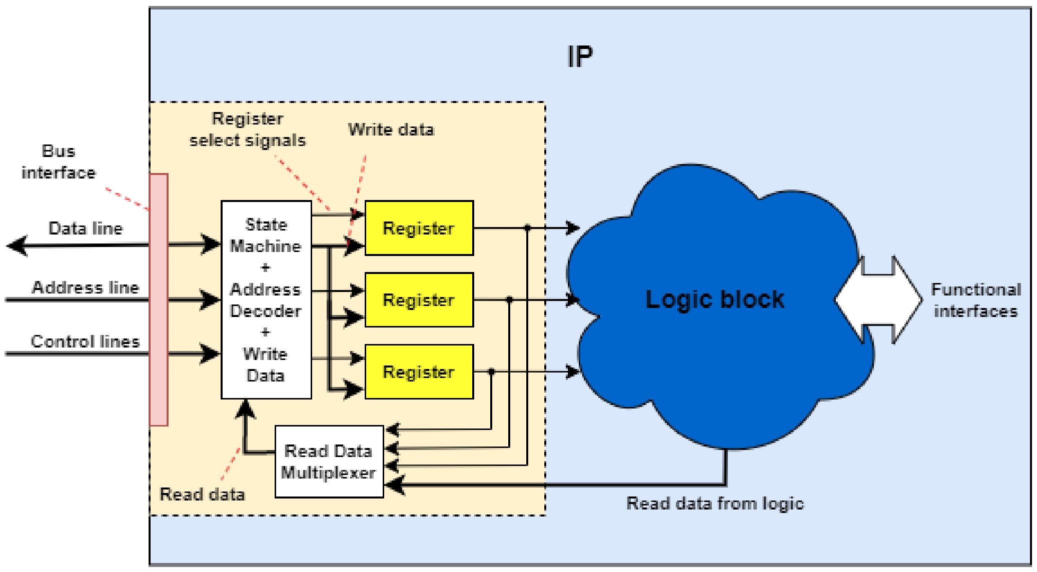


Figure 7. Typical IP core structure.

4.2.2. Register banks

The registers that have been mapped to the peripheral bus address space can be grouped together to form register banks. Register banks function as models for programming the operation of IPs. They define how the peripheral bus masters control and supervise the peripheral. IP register banks usually contain several types of registers. Two common types are configuration registers and status registers. By writing into configuration registers, CPU can configure, start, or stop features of the IP. By reading from status registers, CPU can check whether a certain event has occurred in the IP.

Typically, when registers are mapped to the addresses in the bus address space, the register bank is appointed a base address. The registers inside of the register bank are then referred to by adding a corresponding address offset to the base address. Similarly, the fields inside of the registers are referred to by adding a bit offset to the address of the register.

Depending on the application, registers and register fields can have specific pre-defined access policies. The access policies are a part of the decoding logic of IPs and define how the registers and register fields are handled when they are accessed using read and write operations. Registers have only three possible access policies: read-write (RW), read-only (RO), and write-only (WO). The policy is configured when the register is mapped to a particular address, and the register can have different access policies when it is added to different address maps.

Register fields have more options when it comes to access policies. The UVM provides a comprehensive pre-defined set of field access policies. The access policies are examined in more detail in section 4.4.

The previously mentioned register properties are depicted in figure 8. Register fields also store pre-defined reset values that are applied to the fields during reset. Additionally, various properties of register blocks, registers, and register fields can be parameterized. These properties include the number of register instances, reset values, access policies, and the width of register fields.

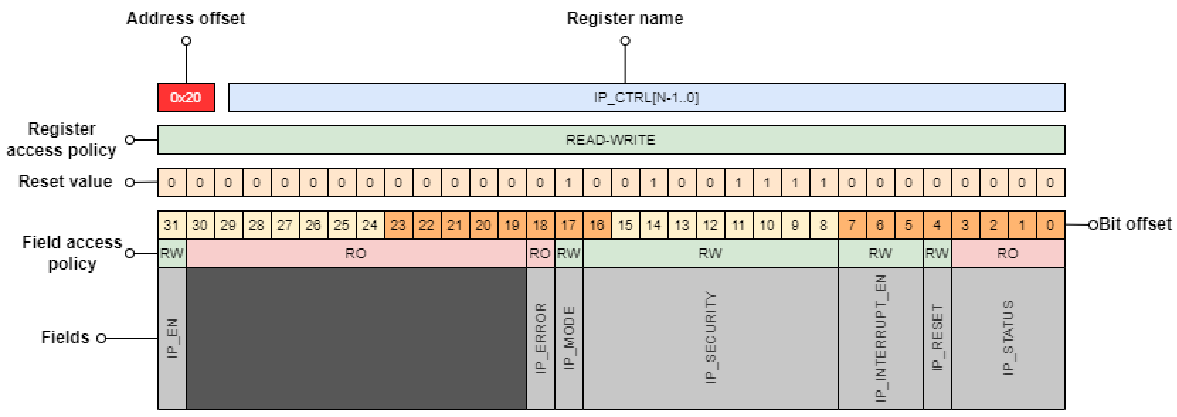


Figure 8. Typical register in a register bank.

4.3. Register description standards

Standards are typically used to provide consistent means of defining information in a specific domain. Standards such as IP-XACT and systemRDL are no different. Both define a standard way of describing key details about an IP in a way that both the users and tools are able to access the information in a consistent and potentially automated fashion [8].

Registers and memory elements form a large part of today's large and complex designs. The continuously increasing number of registers makes documentation, implementation, and maintenance a growing challenge. Additionally, changing specifications during the design cycle require repeated updates to documentation, design, test bench, and to register test cases. Manually managing these components can be tedious and increases the probability of introducing errors in the process [9].

Typically registers have a regular structure that is defined by their field attributes. This makes it possible to define a flow where the register architecture is defined using register description standards, such as IP-XACT and systemRDL. The register description files can then be used to generate the design, verification, and documentation components. This way the efficiency of the flow is increased, and the amount of error-prone register management is reduced [9].

Machine readable standards, usually based on the Extensible Markup Language (XML), are commonly used in the industry to promote IP reuse and automate parts of the flow. IP-XACT is one commonly used standard, but companies have also developed their own standards for describing the relevant parts of peripheral IPs.

4.3.1. XML

XML is a markup language that was created to structure, store, and transport information. It does not do anything by itself. XML data is stored in a text format which is software and hardware independent. This enables the transporting of data between incompatible systems and applications with varying formats. XML is also readable to both users and tools. Because XML data is in plain text, the only requirement for applications handling XML is text processing [11].

In addition to text, XML files contain tags. The tags implement the structure of the data. The text in the file is surrounded by these tags, which adhere to specific syntax guidelines. All tags are user-defined, meaning that XML has no pre-defined tags. This means that XML is highly customizable [10][11].

XML schema is used to define constraints of an XML document. It defines the elements and attributes that can appear in the document, their child elements and their order, the data types for elements and attributes, and the default and fixed values for elements and attributes. Additionally, it defines whether an element is empty or can contain text. IP-XACT, which is also based on XML, provides XML schemas for different types of XML documents [11].

4.3.2. IP-XACT

IP-XACT is an industry-wide initiative to systematize the creation of vendor-neutral machine-readable IP descriptions. The IP descriptions adhere to a specific set of syntax and sematic rules, which means that tools and scripts can utilize these "electronic databooks" for documentation generation, test generation, or any other operation that require the knowledge about the details defining IP components. The IP-XACT standard also defines an Application Programming Interface (API) called Tight Generator Interface (TGI), which the different generator scripts can use to access the IP data. The IP-XACT standard was originally developed by the SPIRIT consortium. It enables a productivity boost in design, transfer, validation, and documentation workflows [6][8].

The IP-XACT standard provides XML schemas for several different XML documents. The document types are component, design, design configuration, bus definition, abstraction definition, abstractor, generator chain, and catalog. Figure 9 depicts the IP-XACT design environment.

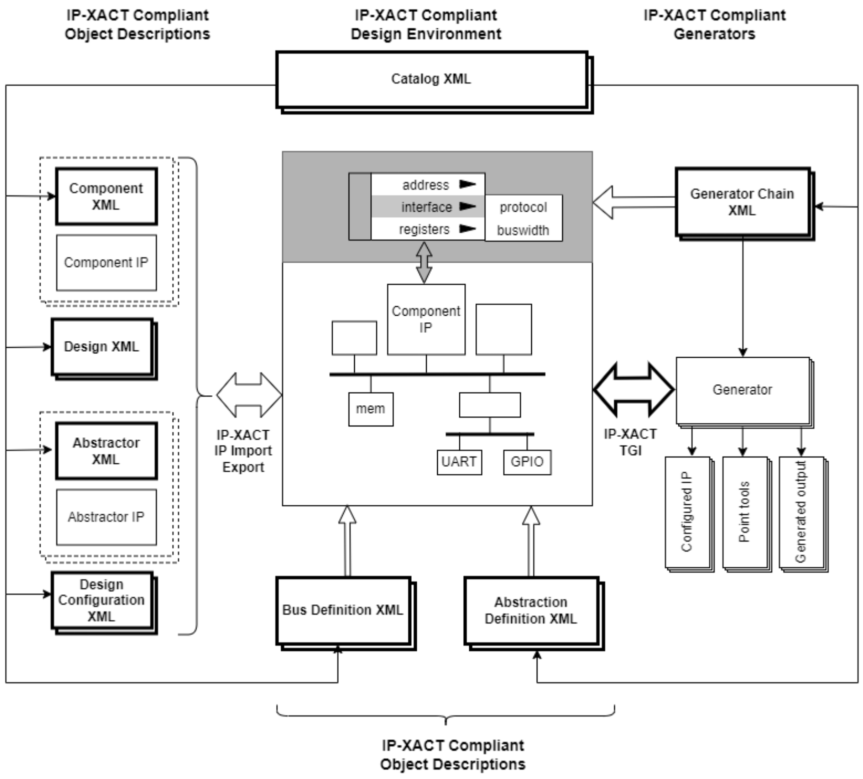


Figure 9. IP-XACT design environment. The objects shown in bold are included in the IP-XACT schema.

The purpose of IP-XACT component file is to enable the use of an IP without the need for information from the implementation files. To achieve this, component type documents the interfaces of an IP, such as parameters, registers, ports, and grouping of ports into bus interfaces. It also documents the views of an IP, such as RTL and transaction level modeling (TLM) descriptions, and the files implementing each view, such as Verilog, VHDL, and SystemC files.

The design schema contains the description of a system or sub-system and includes all component instantiations and connections between them. The interconnections may also be between interfaces or between ports on a component. Design description is comparable to a schematic of components.

An IP-XACT design configuration is a placeholder for additional configuration information of a design or generator chain description. Design configuration information is used to support configuration of hierarchical designs, for transferring designs between design environments and automating generator chain execution for a design, by storing information that would otherwise have to be re-entered by the designer.

The interface definition descriptions bus definition and abstraction definition are used to describe groups of ports that together perform a function. These two descriptions are referenced by components or abstractors in their bus or abstractor interfaces. The bus definition description has the high-level attributes of the interface, including items such as the connection method and indication of addressing. The abstraction definition contains the low-level attributes of the interface, such as the name, direction, and port widths.

Designs may contain interconnections between components that have different abstractions of the same bus type. The IP-XACT abstractor describes how such interconnections are made. An abstractor contains only two interfaces, which shall be of the same bus definition and different abstraction definitions.

The purpose of a generator chain is to describe flows that are enabled by IP-XACT, and it documents a sequence of generators. A generator is typically a script or an executable that implements a flow step by using, creating, or manipulating information described in IP-XACT files [8].

The Catalog is used to manage collections of IP-XACT files by documenting the file locations and the identifiers of the elements documented in those files [8]. It provides design environments a standard means of locating IP-XACT documents [12].

IP-XACT generators are invoked from within design environments to perform an operation required by the user of the environment. Generators can be provided to, for example, generate address maps, and verify the configuration of a subsystem. To perform their operations, most generators require an access to the IP-XACT meta-data describing subsystems. The TGI defines how the design environment and generator cooperate to perform the required operation. It defines the method of communication between the design environment and the generator, the method for invoking the generator, and the actual API that can be used to read and write the IP-XACT meta-data.

4.3.3. Register description using IP-XACT elements

An IP-XACT component file describes the meta-data associated with any IP that can be instantiated in a design. It can describe IP such as cores, peripherals, buses, or any other IP block that can be instantiated in a design. A component can be either static or configurable. Static means that the design environment cannot change it, and configurable means that the component has configurable elements, parameters for example, that can be configured by the design environment, and these elements may also configure the RTL or TLM model [12].

Figure 10 shows a rough depiction of the hierarchy used to describe registers in IP-XACT component file. Each of the slave interfaces of a component can be assigned a memory map. These memoryMaps are grouped together under a memoryMaps element. They contain addressBlock elements that each describe a single, contiguous block of memory inside the memory map [12]. Address blocks can contain either individual register elements or registerFile elements that are used to group together several register elements [8].

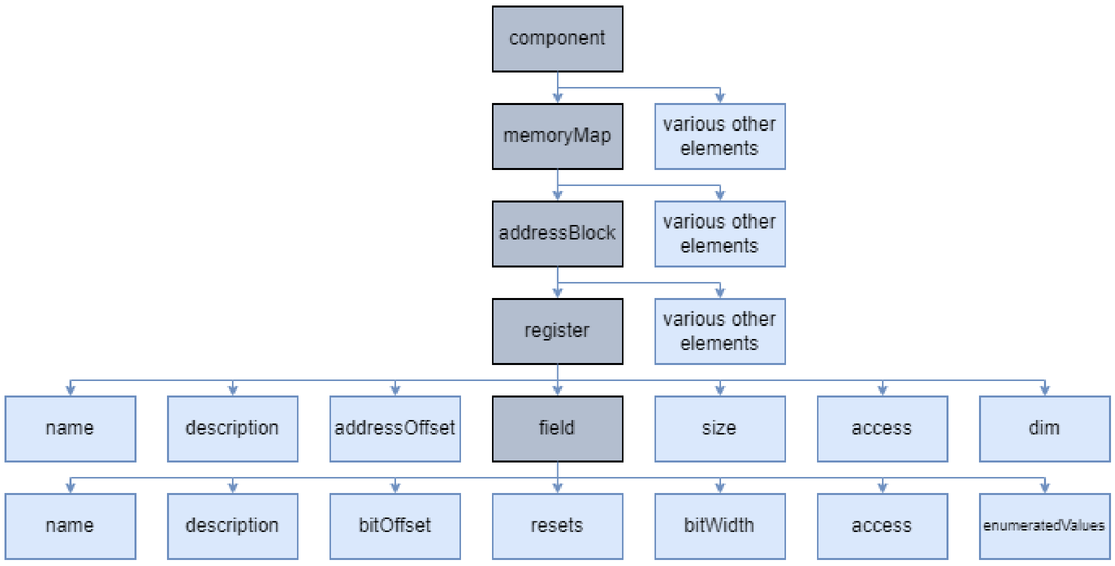
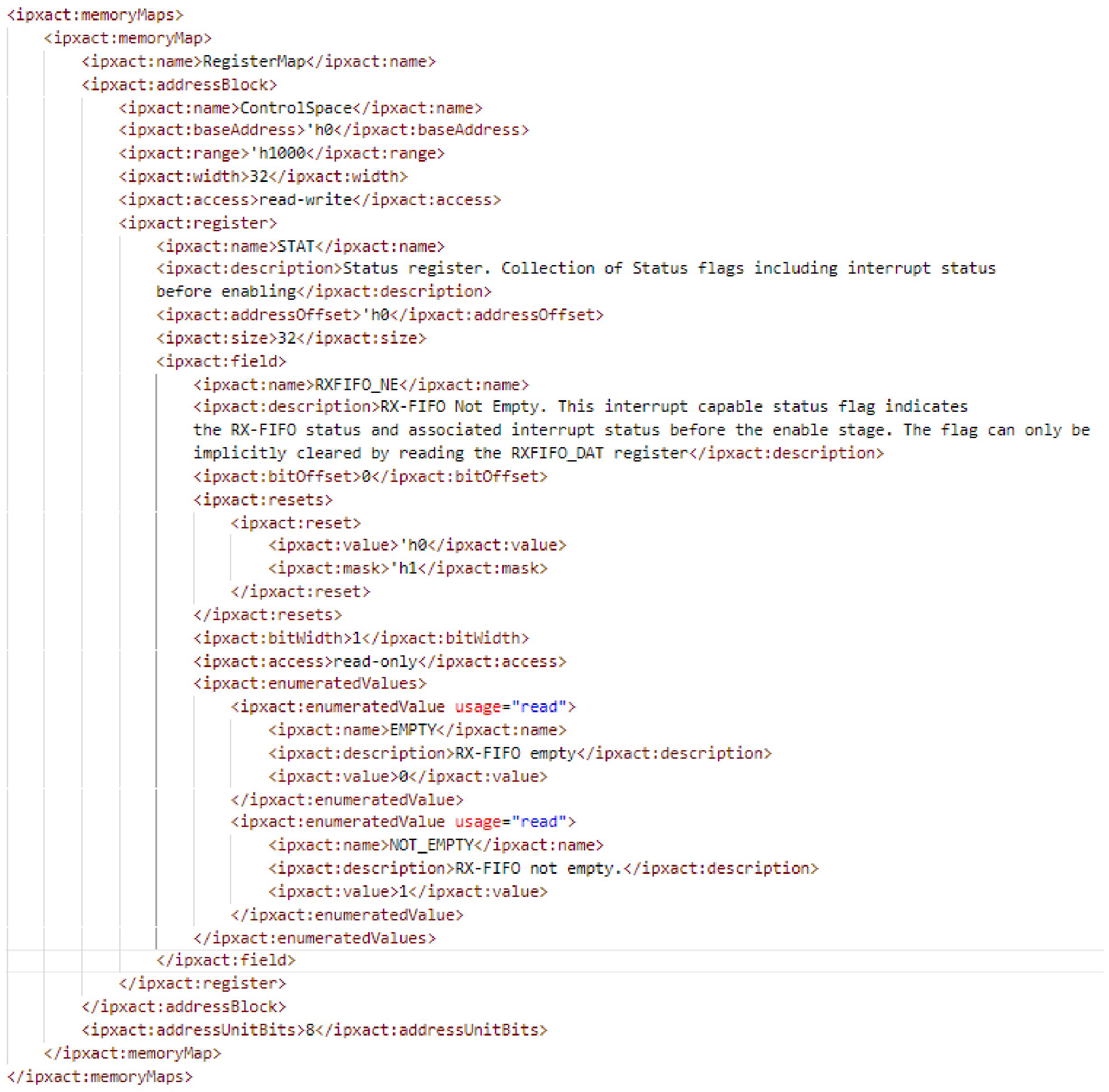


Figure 10. Hierarchical view of register related elements in IP-XACT component file.

A register element describes the software interface to a register. It contains elements for defining its name, size, and its location in the address block address space (addressOffset). It can also contain other elements, namely a description, dim, access, and fields. The description element can be used to add a user readable description of the register. The dim can be used to describe the dimension of the register. The access element describes the access policy of the register [8].

A field element describes one or more bits of a register. It contains elements for describing its name, its starting bit (bitOffset), and the number of bits it contains (bitWidth). Additionally, it can contain other elements, including a description, a resets element containing multiple reset elements, an access element, and an enumeratedValues element. Each reset element has a value describing the reset value of the bits in the field and a mask describing which bits in the field haver a defined reset value. The access elements describe the access policy of the field, similar to the access element of a register. The enumeratedValues element is a container element for enumeratedValue elements which describe a value of the field [8]. The different elements contained in register and field elements can be seen in figure 10. Figure 11 shows an example of a register description made using IP-XACT.



```xml
<ipxact:memoryMaps>
    <ipxact:memoryMap>
        <ipxact:name>RegisterMap</ipxact:name>
        <ipxact:addressBlock>
            <ipxact:name>ControlSpace</ipxact:name>
            <ipxact:baseAddress>'h0</ipxact:baseAddress>
            <ipxact:range>'h1000</ipxact:range>
            <ipxact:width>32</ipxact:width>
            <ipxact:access>read-write</ipxact:access>
            <ipxact:register>
                <ipxact:name>STAT</ipxact:name>
                <ipxact:description>Status register. Collection of Status flags including interrupt status
                before enabling</ipxact:description>
                <ipxact:addressOffset>'h0</ipxact:addressOffset>
                <ipxact:size>32</ipxact:size>
                <ipxact:field>
                    <ipxact:name>RXFIFO_NE</ipxact:name>
                    <ipxact:description>RX-FIFO Not Empty. This interrupt capable status flag indicates
                    the RX-FIFO status and associated interrupt status before the enable stage. The flag can only be
                    implicitly cleared by reading the RXFIFO_DAT register</ipxact:description>
                    <ipxact:bitOffset>0</ipxact:bitOffset>
                    <ipxact:resets>
                        <ipxact:reset>
                            <ipxact:value>'h0</ipxact:value>
                            <ipxact:mask>'h1</ipxact:mask>
                        </ipxact:reset>
                    </ipxact:resets>
                    <ipxact:bitWidth>1</ipxact:bitWidth>
                    <ipxact:access>read-only</ipxact:access>
                    <ipxact:enumeratedValues>
                        <ipxact:enumeratedValue usage="read">
                            <ipxact:name>EMPTY</ipxact:name>
                            <ipxact:description>RX-FIFO empty</ipxact:description>
                            <ipxact:value>0</ipxact:value>
                        </ipxact:enumeratedValue>
                        <ipxact:enumeratedValue usage="read">
                            <ipxact:name>NOT_EMPTY</ipxact:name>
                            <ipxact:description>RX-FIFO not empty.</ipxact:description>
                            <ipxact:value>1</ipxact:value>
                        </ipxact:enumeratedValue>
                    </ipxact:enumeratedValues>
                </ipxact:field>
            </ipxact:register>
        </ipxact:addressBlock>
        <ipxact:addressUnitBits>8</ipxact:addressUnitBits>
    </ipxact:memoryMap>
</ipxact:memoryMaps>
```

Figure 11. Example IP-XACT register description.

4.4. Registers in SoC verification

Traditional verification methods require the verification engineers to manually write many test sets to verify all registers in Design Under Test (DUT). With the increasing scale and complexity of the chips, the number of needed registers will also increase. If each register is tested by manually writing a test set for it, a large amount of the chip's research and development time will be used up [14].

One method of verifying the memories and registers more effectively is to model them using register models, which are a part of the UVM. The register models allow for easier stimulus generation and functional checking. The models consist of a set of register definitions, a list of register instances that form register blocks and their address mappings into DUT's address space. The register models can be implemented manually by the verification engineer, or preferably by generating them from a textual register description such as an IP-XACT description [13].

4.4.1. UVM

The UVM is a complete methodology and class library that codifies the best practices for efficient and exhaustive verification. It is focused on the verification of small designs and large-gate-count, IP-based SoC designs [16].

One of the key aspects of UVM is to implement reusable verification components, called UVM Verification Components (UVC). A UVM-compliant UVC is an encapsulated, ready-to-use and configurable verification environment designed for an interface protocol, a design sub-module, or even for software verification. Each UVC follows a consistent architecture, and contain elements for sending stimulus, as well as checking and collecting coverage information for a specific protocol or design [16].

The UVM Testbench typically instantiates the DUT and the UVM Test class and configures the connections between them. The test class is instantiated dynamically, allowing the testbench to be compiled once and run with several different tests [20]. An example component hierarchy of a UVM Testbench can be seen in figure 12.

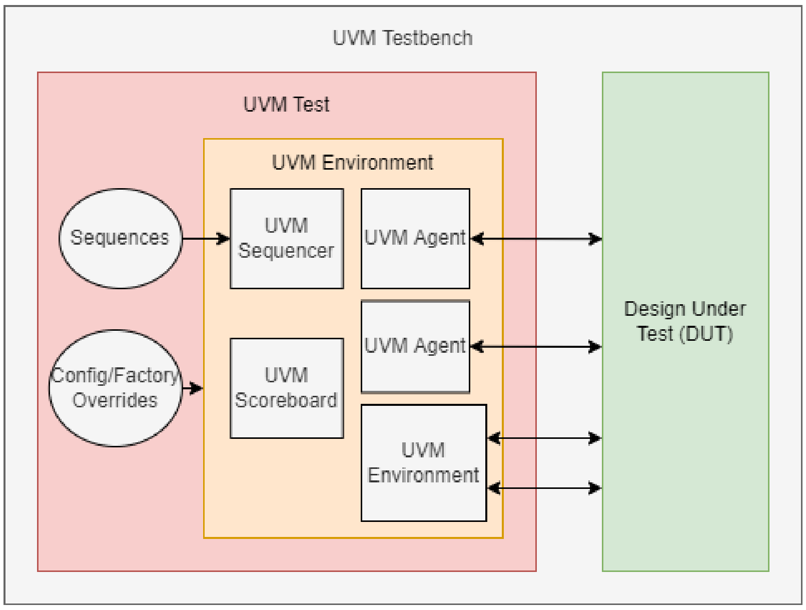


Figure 12. An example UVM testbench.

The UVM Test component has typically three main functions: instantiation of the environment, configuration of the environment, and stimulus generation by invoking UVM Sequences through the environment to the DUT. There is usually one base UVM Test component that contains the UVM Environment instantiation and other common items, which is extended by other tests. The other tests can then, for example configure the environment differently or run different sequences [20].

The UVM Environment component groups together other verification components that are interrelated. Some components that are instantiated inside the UVM Environment include UVM Agents, UVM Scoreboards, UVM Register models, or other environments.

The main function of the UVM Scoreboard component is to check the behavior of the DUT. It usually receives transactions containing input and output data of the DUT via the UVM Agent analysis ports and compares the transactions to predicted results from a reference model (predictor) [20].

The UVM Agent component groups together other verification components that are used to handle specific DUT interfaces. A typical UVM Agent contains a UVM sequencer to manage stimulus flow, a UVM driver to apply the stimulus to the DUT interface, and a UVM monitor to monitor the DUT interface [20]. The typical structure of a UVM Agent component can be seen in figure 13. UVM Driver and Monitor components are connected to the DUT interface using a Virtual Interface (VIF).

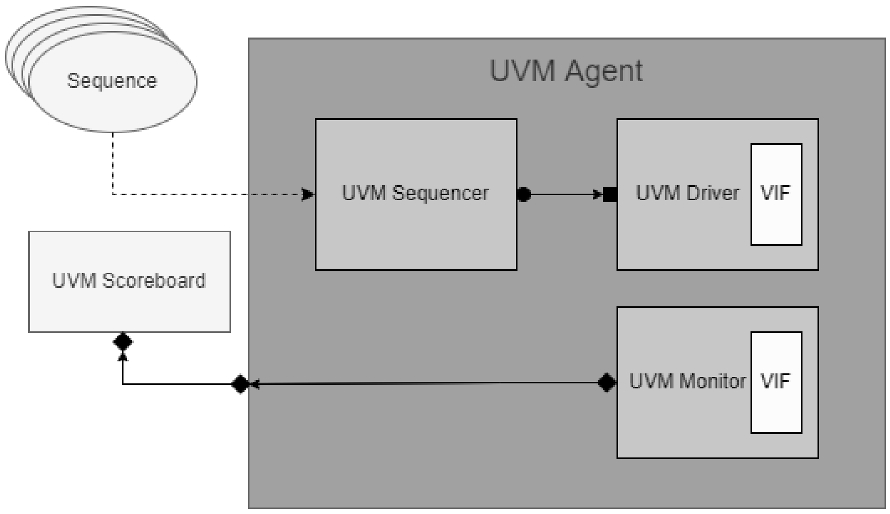


Figure 13. A typical UVM Agent structure.

UVM Class Library contains a set of base classes, utilities and macros that enable the implementation of well-constructed, reusable SystemVerilog based verification environments [19]. One of the advantages of UVM Class Library is that it provides many essential features for verification, such as complete implementation of printing, copying, test phases, factory methods, and others. Another advantage is the increased readability brought about by the hierarchical components that are extended from the base UVM Class Library components. Additionally, the UVM Class Library provides many utilities to simplify the development and use of verification environments, including utilities that provide a standard resource sharing database, user-controllable messaging for various reporting purposes, and a standard communication infrastructure between verification components (TLM). Finally, UVM Class Library also provides macros for allowing more compact coding styles [20].

4.4.2. UVM Register models

In the context of verification, a register model, or a Register Abstraction Layer (RAL), is a set of classes that model the operation of memory-mapped registers and memories in the DUT to facilitate stimulus generation and functional checking. The UVM provides a set of base classes that can be extended to implement comprehensive register modeling [17].

A register model is an instance of a register block, and they are instantiated in the UVM Environment component. Figure 14 illustrates the fundamental structure of a register block. Register blocks may contain any number of registers, register groups, memories, and other blocks. Each register contains any number of fields, which mirror the values of the corresponding fields in the DUT. The elements that form the register model are all DUT specific and are implemented by extending the base UVM classes [20].

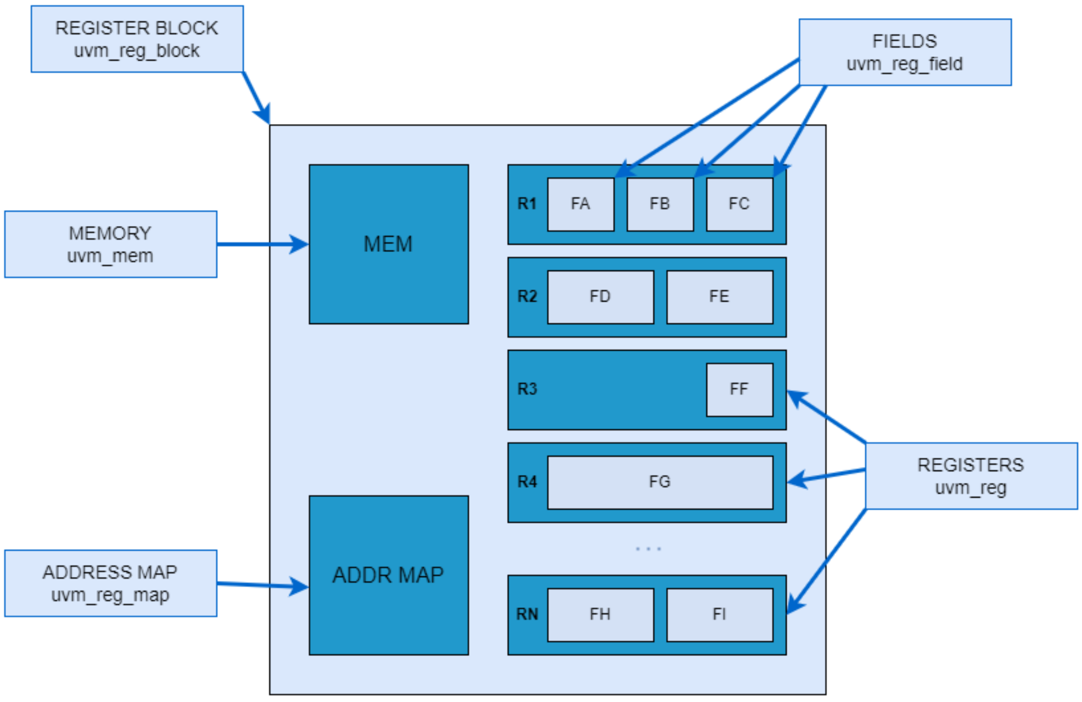


Figure 14. Typical register model structure.

The lowest register abstraction layer is the uvm_reg_field, and it represents the bits of a register. It uses several properties to store a variety of register-field values: m_reset, m_mirrored, m_desired, and value. The m_reset property stores the reset value and m_mirrored stores the value that the register model thinks is in the corresponding DUT register [18]. For non-volatile register fields, the mirrored value provides the current state of the DUT register based on all active and passive bus operations. Volatile fields require additional modeling to maintain the state of the mirrored value based on other non-bus related application-specific operations [17]. The m_desired property stores the desired mirrored value, and the value property stores the value to be sampled in a functional coverage, or the value to be constrained when the field is randomized [18].

UVM register models have built in methods for accessing registers and register fields. These methods are called register access methods. Access methods can either use front-door access or back-door access to DUT registers. The front-door access involves using the bus interface and is clocked. Back-door access operates by directly accessing the simulation constructs that implement the register model through a hierarchical path within the design hierarchy. Additionally, back-door accesses are performed instantly without using any simulation time.

Register access methods can be divided to several groups. Active operations which update both the mirrored and DUT register values include write, poke, set-update, and randomize-update operations. Active operations which update the mirrored value based on the DUT register values include read, peek and mirror operations. Finally, passive operations which update the mirrored value independently from the DUT include operations such as reset and predict. UVM also provides a configure method which is used to configure the register fields. The register access methods are described in table 1. Their operation generally revolves around moving data between the previously mentioned properties and the DUT register fields.

Table 1. A description of register model access methods

Access method	Description
write/read (front-door)	Writes/reads DUT registers using the bus UVC. Updates the mirrored value to reflect the expected value in DUT.
write/read (back-door)	Writes/reads DUT registers via the back-door mechanism, bypassing the physical interface. The behavior of the registers is mimicked as much as possible. The mirrored value is updated to reflect the actual sampled or deposited value.
peek/poke	Reads/writes DUT registers directly, bypassing the physical interface. The mirrored value is updated to reflect the actual sampled or deposited value.
get/set	Reads/writes directly to the desired value without accessing the DUT. The desired value can then be uploaded to the DUT using the update method.
randomize	Copies the randomized value from the value property to the desired value. The desired value can then be uploaded to the DUT using the update method.
update	Invokes the write method if the desired value differs from the mirrored value.
mirror	Invokes the read method to update the mirrored value based on the readback value. Can also compare the readback value with the current mirrored value before updating it.
configure	Configures the register field by setting parameters, including the reset value and the access policy.
reset	Resets the properties of the register field by writing the pre-defined reset values to the mirrored value. Does not affect DUT field values.
predict	Updates the mirrored value. Updates also the m_desired and value properties.

The UVM provides a set of pre-defined policies for accessing register fields, and they are listed in table 2. The field access policy is usually set using the configure method. An important aspect of field access policies is that their behavior can be explained in term of register operation in isolation from other fields and registers. In addition to the field's configured access policy, whether a field can be read or written depends also on the register's access policy (RW, RO, WO) in the used address map. These access policies are used together with observed read and write operations to determine the expected value of a field. Most fields fall within one of the pre-defined access policies, but it is also possible to design a field that behaves predictably but differently from the pre-defined ones.

Table 2. Predefined UVM register field access policies

Access policy	No write	Write value	Write to clear	Write to set	Write to toggle	Write once
No read	-	WO	WOC	WOS	-	WO1
Read value	RO	RW	WC W1C W0C	WS W1S W0S	W1T W0T	W1
Read to clear	RC	WRC	-	WSRC W1SRC W0SRC	-	-
Read to set	RS	WRS	WCRS W1CRS W0CRS	-	-	-

A register model can be implemented using a generator or it can be written by hand. Writing a register model with more than a few registers by hand is typically tedious and prone to errors, so it is often preferrable to generate it from a register description file such as an IP-XACT description. Generators are also helpful because they allow the use of a common register specification by the design, verification, and software teams, and it allows the register definitions for different IP cores to be merged together into an overall register description on sub-system and top levels [35].

The finished register model is instantiated in the verification environment together with other verification components, as shown in figure 15. The register model interacts with the rest of the verification environment via adapter and predictor components. The adapter converts between register model read and write methods and the interface-specific transactions. The predictor component updates the register model based on observed transactions published by a monitor [17].

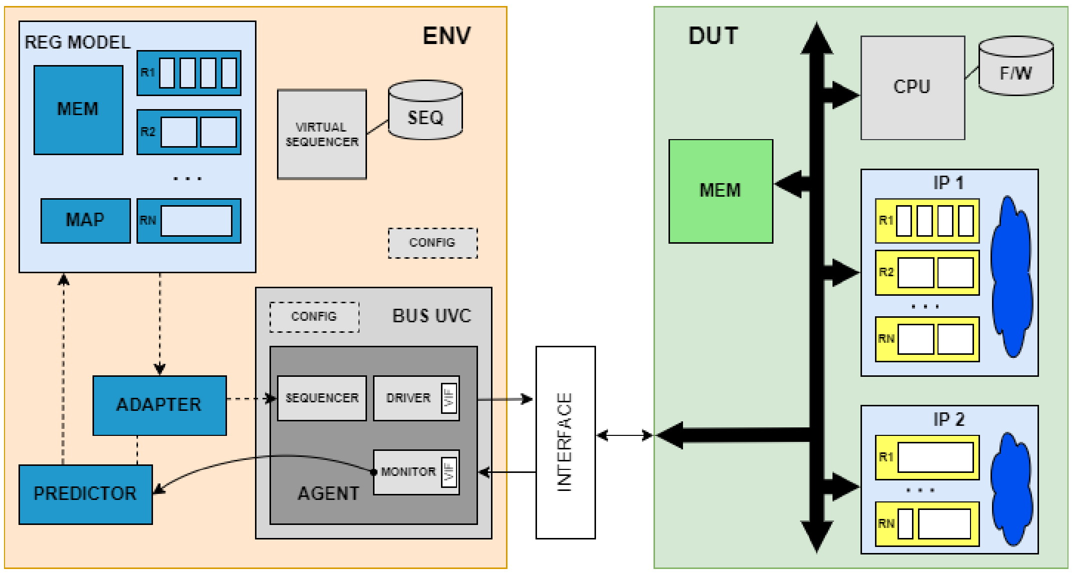


Figure 15. Environment with register model.

The UVM register model supports active and passive modes of operation and is often implemented in environments where both modes are used. In the case of active modes of operation, register operations are done using the register model methods, such as read and write. The read and write method calls get converted to sequence items via the adapter and are then passed on to an interface verification component (path 1 in figure 16). Passive operations refer to register operations which do not use the register model access methods directly. One example of a passive operation is when an interface component sequence is executed directly by the virtual sequencer without calling the register model access methods (path 2 in figure 16). Another example is when the content of a register is changed by another stimulus source, such as an embedded CPU (path 3 in figure 16).

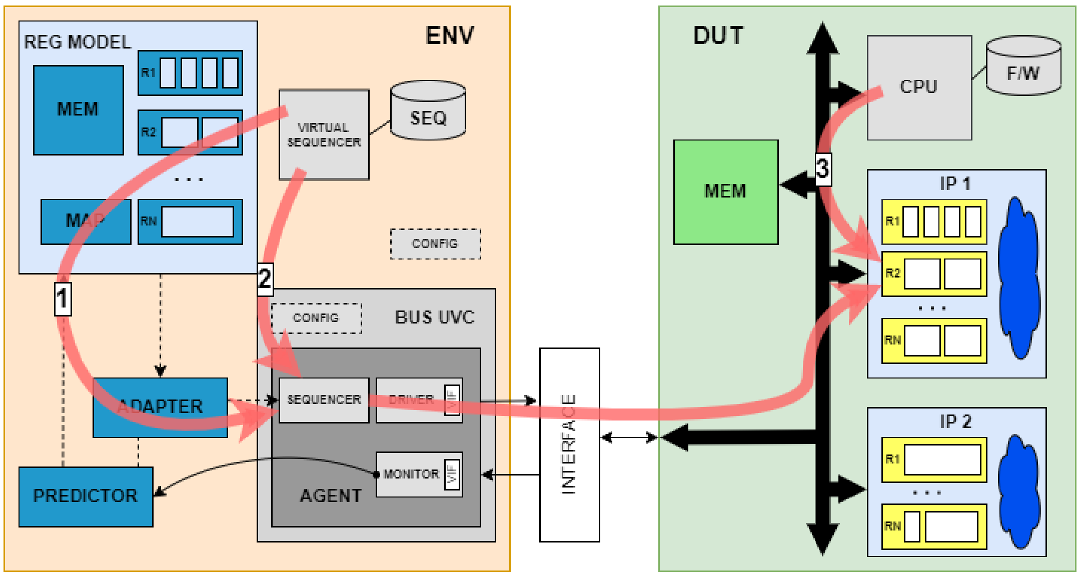


Figure 16. Register model active and passive operation paths.

4.4.3. Register verification using UVM register models

Register verification is a significant part of the verification process. Registers are one of the first aspects of the design to be tested because they are used to configure the IP cores and are the basis of the hardware-software interface. Hence, the rest of the design's functionality testing depends on the accuracy of the register implementation [32]. The implementation complexity of registers is relatively low in comparison to other design elements. The challenge of register verification is that each design contains excessive number of memory mapped registers with varying access policies and other properties. This makes manual register verification by directed or lightly randomized methods tedious and prone to errors [33].

The verification of IP core registers should be considered from two angles: register structure and functional requirements. The register structure point of view focuses on the correctness of the implementation of the register structure. This involves checking the access policies of registers and register fields, register address space accessibility, the operation of the access methods such as read and write, and the overall register operation. The functional requirements point of view focuses on the correctness of the functionality provided by the register. This includes checking whether the values of the register match the expected reset values after reset and whether all values of the configuration registers are having the desired effect on the IP core operation. It also includes checking that status registers are updated when the corresponding event is triggered, and whether they are reflecting it correctly.

The UVM provides a library of automatic test sequences for testing register structure correctness, such as reset values and read-write method operation [35]. Checking the functional correctness of registers is more challenging, since it includes checking how the combinations of each of the values of each of the different registers affects the overall operation of the IP. UVM RAL is a commonly used for verifying the functionality provided by registers more efficiently.

5. PARAMETERIZED REGISTER MODEL IMPLEMENTATION

This chapter covers a case study on the implementation of parameterized register models. Section 5.1 gives a general overview on parameterized register models. Section 5.2 explains the differences between static and dynamic register models. Section 5.3 introduces the register description formats that are used at Nordic Semiconductor and goes through the currently used processes for generating register models. Additionally, it proposes changes for enabling the generation of dynamic register models, and some other possible changes for increasing the flexibility of the generated register model. Section 5.4 documents an experiment in which the proposed changes are used to implement a dynamic register model for a highly configurable IP core. The section also describes the methods used to verify the dynamic register model.

5.1. Register models and parameters

Two commonly used methods of increasing the efficiency of register verification are the use of register models and elements that enable reusability, such as parameters. Register models provide an effective method of accessing design register values, and parameterized register models increase the flexibility of the verification environment.

Due to the vast number of registers, the register models are typically generated automatically using generator tools. The addition of parameterized registers makes the process of generating register models more complicated. Ideally, the automatically generated register model would use the parameters defined in the verification environment without any manual tinkering. This would make the register model flexible, reducing the need to regenerate the register model each time the parameter configuration is changed.

Currently, however, the register models are generated without parameters, or they are configured before model generation. The generated models are then edited manually to make them more flexible, or to set certain properties that could not be set during generation. This manual configuring process takes away the efficiency benefits gained from automatically generating the register model. The issue is even worse when taken into account that this manual process must be repeated each time the register model has to be generated again.

5.2. Approaches to parameterized register models

Register models can be divided into two groups depending on which point of the register model implementation the values of the parameters are configured. These two groups can be referred to as static and dynamic register models.

5.2.1. Static register models

In the case of static register models, the values of the parameters are configured before or during the model generation. The resulting register model is then included in the verification environment and can be used to verify that specific parameter configuration. The concept of static register models is illustrated in figure 17.

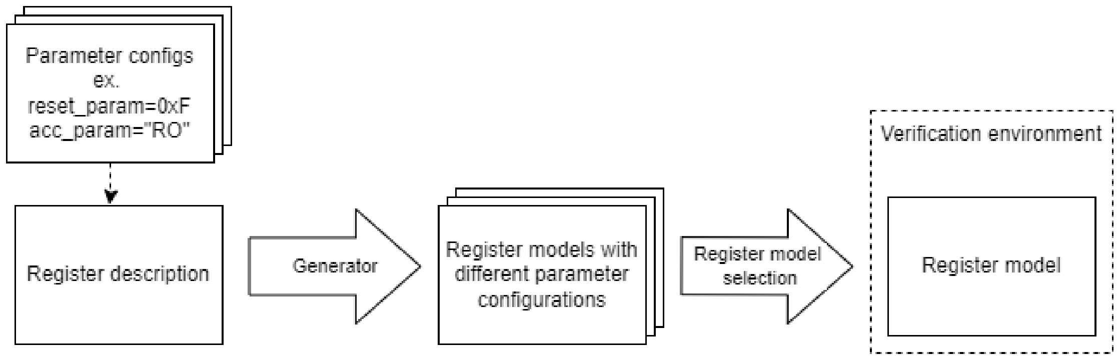


Figure 17. Static register model.

Comprehensive verification of all parameter combinations would require the verification engineer to generate several register models with different parameter configurations. This method could be feasible with a small number of parameters, but as the number of parameters increases, the number of different configurations increases exponentially. Since generating large numbers of register models is not ideal, verification engineers often opt to manually edit the generated register models to make them configurable in the verification environment.

5.2.2. Dynamic register models

Parameterized verification environments typically contain VHDL or SystemVerilog package constructs for storing the parameter values. Verification components are then configured using those values. Hence, the parameter configuration of the entire verification environment can be changed at once by changing the values in the package.

Register models that use the parameter values defined in the verification environment can be considered dynamic. This means that the parameter configuration of the register model is linked to the verification environment, and the parameterized properties of the register model can be configured without having to generate it again. The idea behind dynamic register models is shown in figure 18.

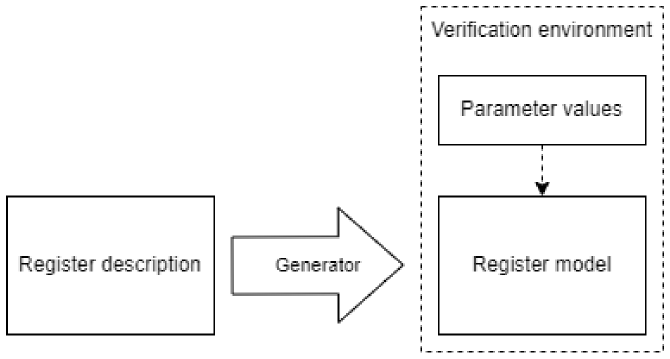


Figure 18. Dynamic register model.

This type of flexibility allows the same generated register model to be reused in various environments that contain different parameter configurations. It also improves the efficiency of IP level verification, since testing the different parameter configurations will require less effort.

A key challenge in the implementation of dynamic register models is the application of the verification environment parameter values. They should be applied in a way which is not affected by the regeneration of the register model. This could be achieved by, for example, linking the parameters to the values of the verification environment during generation, or by applying the parameter values in the verification environment as the register model is instantiated.

5.3. Register model generation flow

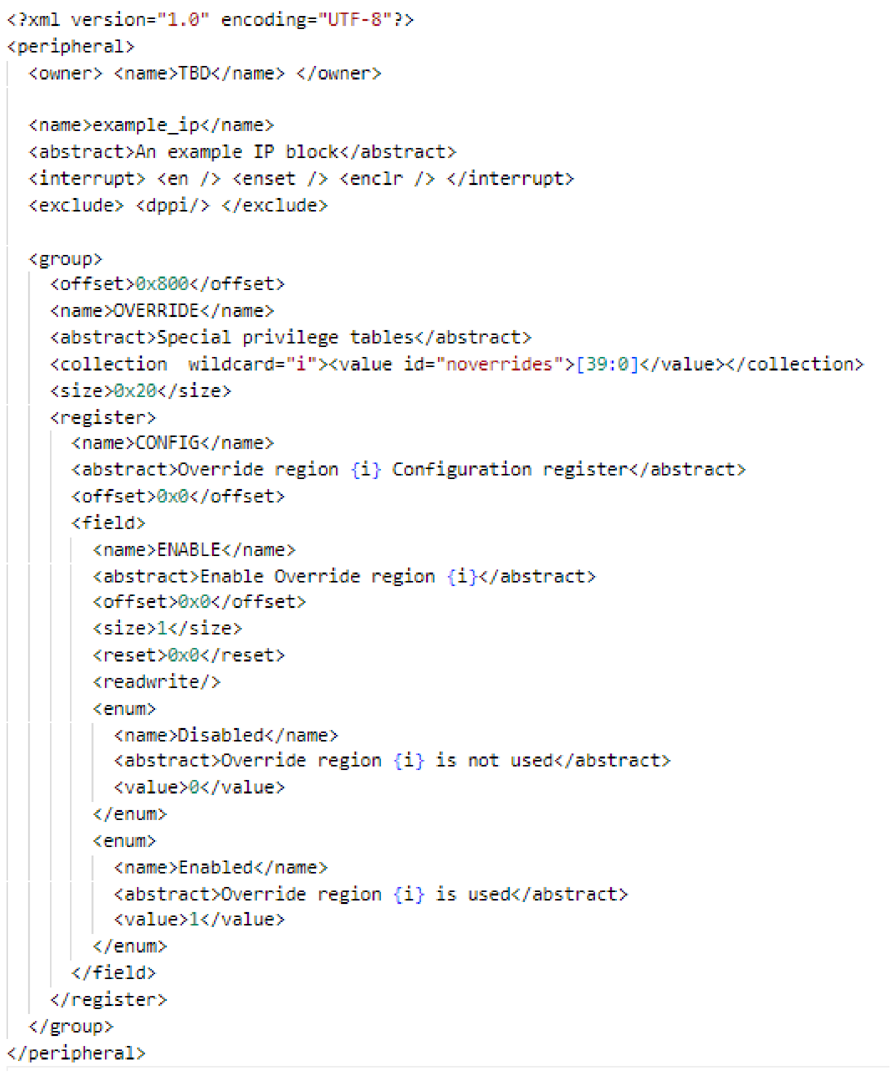
The following sections describe the two different register description formats and the currently used generation flows used at Nordic Semiconductor. The last two propose an improved flow for generating dynamic register models and list some other possible changes to increase the flexibility of the generated register model.

5.3.1. Register descriptions

There are two XML based register description formats in use at Nordic Semiconductor. They are influenced by the IP-XACT standard but are less complex and provide a narrower scope of the device. The first format is developed in house, and the second is a third-party format.

The first register description format (register description A) is focused completely on the description of registers. It is used to define the registers for the company internal product specifications, which specify the functionality of the IP blocks, sub-systems, and systems of the design.

In practice, it is the main description format at Nordic Semiconductor. It is used as a base to generate various other files and documents. An example of the description format can be seen in figure 19. It describes a group of registers, each of them containing a single field with two enumerated values.



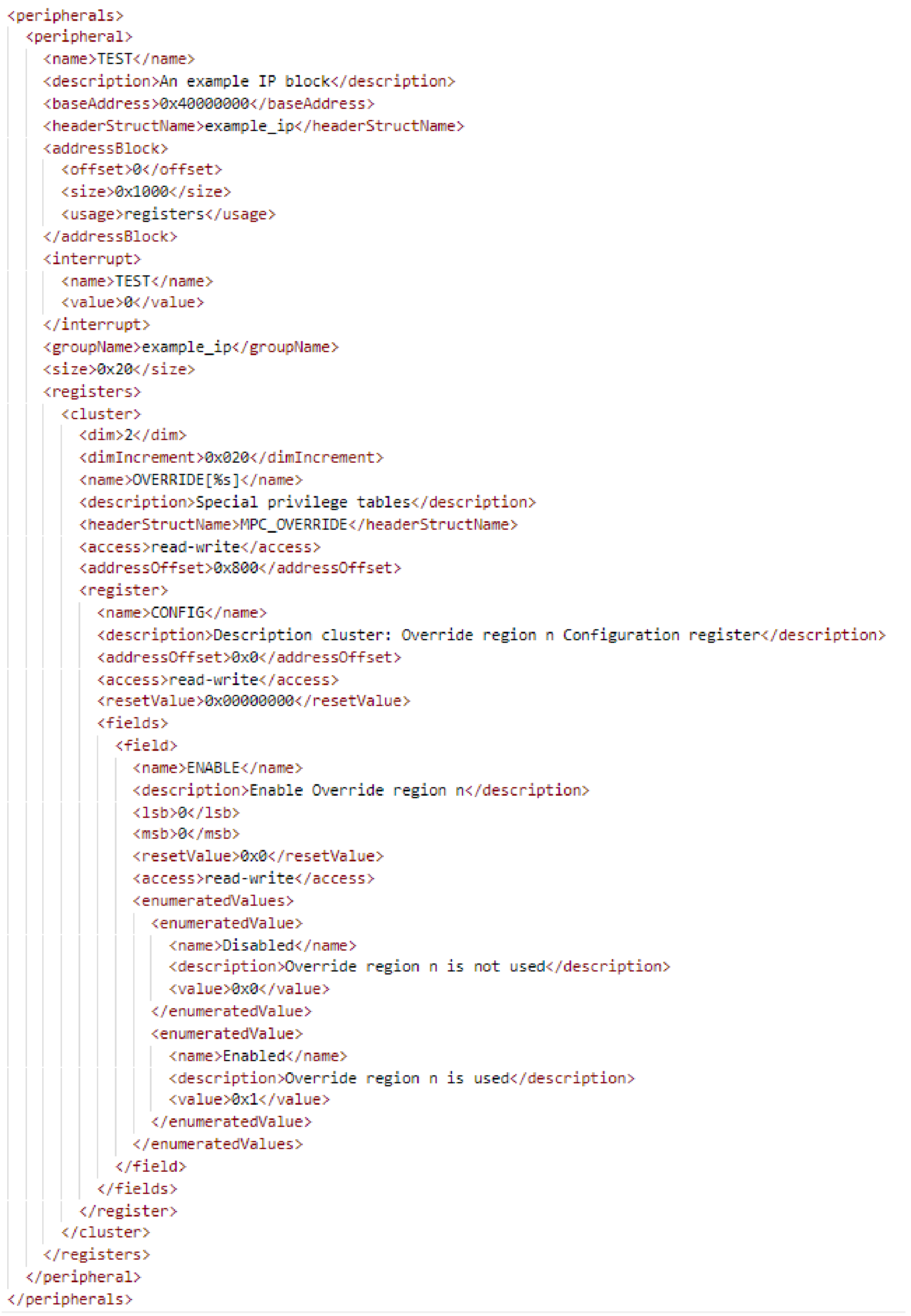
```
<?xml version="1.0" encoding="UTF-8"?>
<peripheral>
  <owner> <name>TBD</name> </owner>

  <name>example_ip</name>
  <abstract>An example IP block</abstract>
  <interrupt> <en /> <enset /> <enclr /> </interrupt>
  <exclude> <dppi/> </exclude>

  <group>
    <offset>0x800</offset>
    <name>OVERRIDE</name>
    <abstract>Special privilege tables</abstract>
    <collection  wildcard="i"><value id="noverrides">[39:0]</value></collection>
    <size>0x20</size>
    <register>
      <name>CONFIG</name>
      <abstract>Override region {i} Configuration register</abstract>
      <offset>0x0</offset>
      <field>
        <name>ENABLE</name>
        <abstract>Enable Override region {i}</abstract>
        <offset>0x0</offset>
        <size>1</size>
        <reset>0x0</reset>
        <readwrite/>
        <enum>
          <name>Disabled</name>
          <abstract>Override region {i} is not used</abstract>
          <value>0</value>
        </enum>
        <enum>
          <name>Enabled</name>
          <abstract>Override region {i} is used</abstract>
          <value>1</value>
        </enum>
      </field>
    </register>
  </group>
</peripheral>
```

Figure 19. Example of register description A.

The second format (register description B) is tailored towards the description of the programmer’s view of a device. The format is mostly intended to provide register information for software verification and provides CPU and overall device information in addition to the register information. The register sections of the formats contain roughly the same information fields with different names. An example of the register information section of the description can be seen from figure 20.



```xml
<peripherals>
  <peripheral>
    <name>TEST</name>
    <description>An example IP block</description>
    <baseAddress>0x40000000</baseAddress>
    <headerStructName>example_ip</headerStructName>
    <addressBlock>
      <offset>0</offset>
      <size>0x1000</size>
      <usage>registers</usage>
    </addressBlock>
    <interrupt>
      <name>TEST</name>
      <value>0</value>
    </interrupt>
    <groupName>example_ip</groupName>
    <size>0x20</size>
    <registers>
      <cluster>
        <dim>2</dim>
        <dimIncrement>0x020</dimIncrement>
        <name>OVERRIDE[%s]</name>
        <description>Special privilege tables</description>
        <headerStructName>MPC_OVERRIDE</headerStructName>
        <access>read-write</access>
        <addressOffset>0x800</addressOffset>
        <register>
          <name>CONFIG</name>
          <description>Description cluster: Override region n Configuration register</description>
          <addressOffset>0x0</addressOffset>
          <access>read-write</access>
          <resetValue>0x00000000</resetValue>
          <fields>
            <field>
              <name>ENABLE</name>
              <description>Enable Override region n</description>
              <lsb>0</lsb>
              <msb>0</msb>
              <resetValue>0x0</resetValue>
              <access>read-write</access>
              <enumeratedValues>
                <enumeratedValue>
                  <name>Disabled</name>
                  <description>Override region n is not used</description>
                  <value>0x0</value>
                </enumeratedValue>
                <enumeratedValue>
                  <name>Enabled</name>
                  <description>Override region n is used</description>
                  <value>0x1</value>
                </enumeratedValue>
              </enumeratedValues>
            </field>
          </fields>
        </register>
      </cluster>
    </registers>
  </peripheral>
</peripherals>
```

Figure 20. Example of the register section of register description B.

5.3.2. Current flow

Register models can be generated from either of the description formats, but each method uses separate scripts and generation tools. Figure 21 illustrates the register model generation flows used at Nordic. The green blocks in the figure represent internal, and the yellow block third-party tools. The two methods are represented by the red arrows.

The first register model generation method involves parsing all of the necessary register information from the register description and listing it in data tables. The tables are then given to a third-party tool that generates the register model. This is currently the typically used method for generating IP level register models.

The second method involves converting the register information first into the second register description format. The register description is then generated into a register model using a python script. The conversion tool is developed by Nordic, and it is also used to generate design specifications.

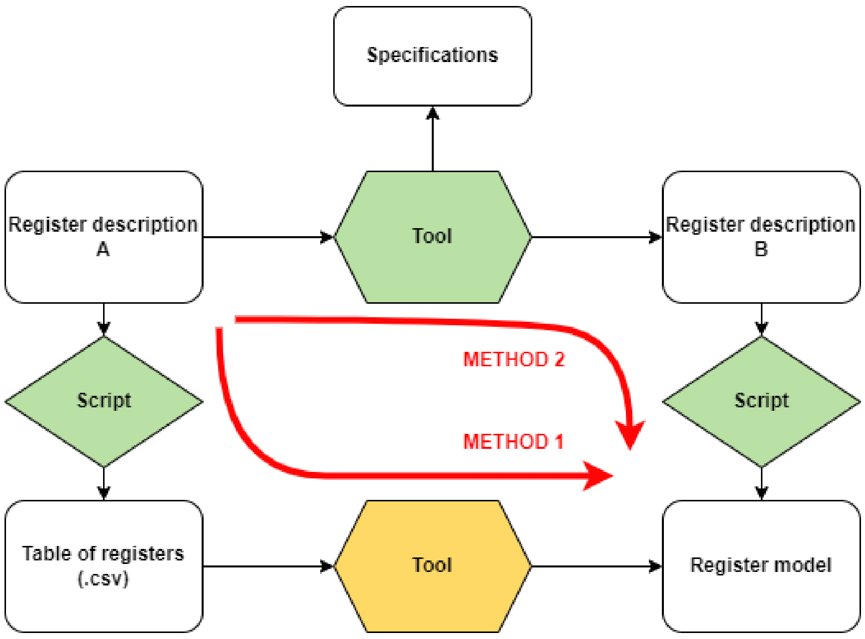


Figure 21. Register model generation flow.

These two generation flows currently only support static register models. The scripts and tools accept only specific types of data, and there are no methods to link the parameters to values defined in the verification environment. Hence, the parameter values are configured before register model generation, and the resulting register models have little flexibility. The inflexibility of the register model makes the verification of the possible parameter configurations significantly more difficult.

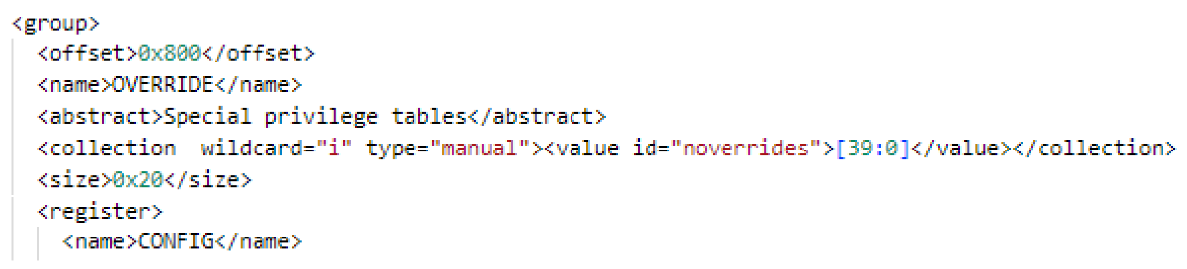
Additionally, the current flows cannot generate register models with all the required features. One example is the implementation of complex register groups. As can be seen from figures 19 and 20, in the register descriptions, groups of registers can be provided dimensions using data fields such as “collection” and “dim”. These fields make the definition of register groups with several instances significantly less tedious, but on the other hand all the instances will always be identical. However, in some cases, the individual registers they contain should have individual parameter values for properties such as reset and access. Therefore, verification engineers are currently required to manually add the missing features to the generated register model.

5.3.3. Improved flow

Instead of static register models, the generation of dynamic register models would be preferrable. To achieve this, one way would be to somehow link the register parameters, such as reset values, access policies, and the number of register instances, to the values stored in the verification environment. This requires making changes to the register descriptions, tools, and scripts of the generation flow. Comparing the two generation methods described earlier, the second would be more suitable because the used tools and scripts are developed inside Nordic and are easier to make changes to. Additionally, the conversion tool used in the second method is also already used to generate design specifications.

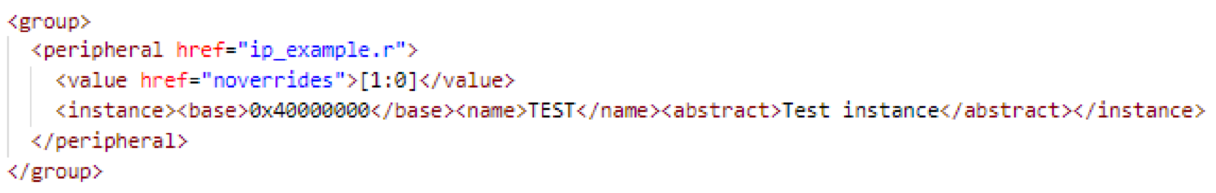
Register description files are provided to the conversion tool by including them in a separate input file. This input file is used to define some basic properties of the IPs, such as their base address, and it can include the register descriptions of several different IPs. All included register descriptions are combined in the generated register model.

Register description format A has a feature to set IDs to various values, such as reset values and register group sizes. When the description is included in the input file, those values can be substituted with other values. Figure 22 illustrates the value ID feature used on a collection element in the register description. Figure 23 shows an example of how the description is included in the tool input file, and how the value with the ID of "noverrides" is changed from [39:0] to [1:0]. The biggest advantage of this feature is that it makes the configuration of the register description a lot simpler.



```
<group>
  <offset>0x800</offset>
  <name>OVERRIDE</name>
  <abstract>Special privilege tables</abstract>
  <collection  wildcard="i" type="manual"><value id="noverrides">[39:0]</value></collection>
  <size>0x20</size>
  <register>
    <name>CONFIG</name>
```

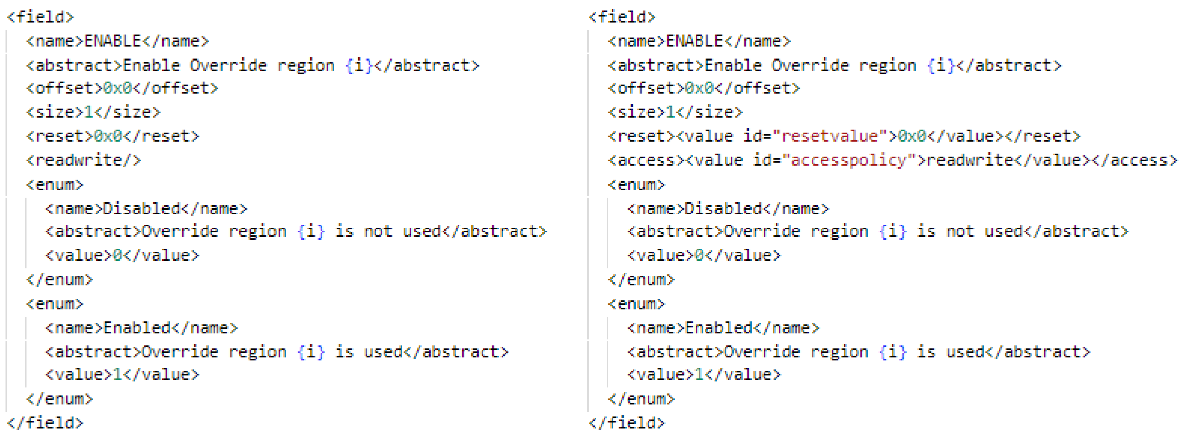
Figure 22. Register description A value IDs.



```
<group>
  <peripheral href="ip_example.r">
    <value href="noverrides">[1:0]</value>
    <instance><base>0x40000000</base><name>TEST</name><abstract>Test instance</abstract></instance>
  </peripheral>
</group>
```

Figure 23. Value substitution.

Currently, these value IDs are only used to substitute the values for the number of registers. They should also be added to all other parameterized register properties, namely reset and access values. However, because of the way access policies are defined in the register description, the value IDs cannot be used to substitute their values. A solution would be to change the description format so that the access policies are defined the same way as other parameterized properties. Figure 24 shows a side-by-side comparison of the original register field element and the new element with added value IDs and a changed access definition.



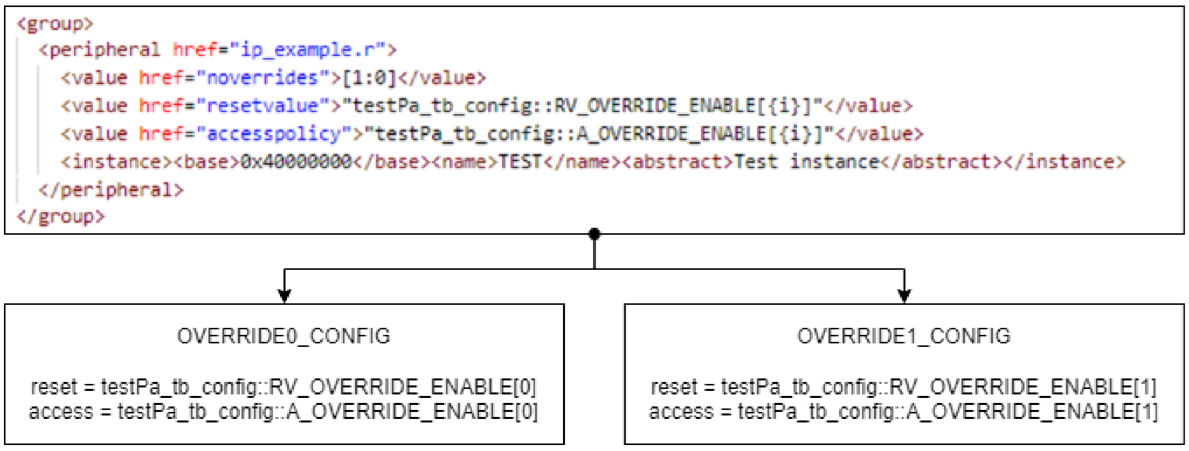
```
<field>
  <name>ENABLE</name>
  <abstract>Enable Override region {i}</abstract>
  <offset>0x0</offset>
  <size>1</size>
  <reset>0x0</reset>
  <readwrite/>
  <enum>
    <name>Disabled</name>
    <abstract>Override region {i} is not used</abstract>
    <value>0</value>
  </enum>
  <enum>
    <name>Enabled</name>
    <abstract>Override region {i} is used</abstract>
    <value>1</value>
  </enum>
</field>
```

```
<field>
  <name>ENABLE</name>
  <abstract>Enable Override region {i}</abstract>
  <offset>0x0</offset>
  <size>1</size>
  <reset><value id="resetvalue">0x0</value></reset>
  <access><value id="accesspolicy">readwrite</value></access>
  <enum>
    <name>Disabled</name>
    <abstract>Override region {i} is not used</abstract>
    <value>0</value>
  </enum>
  <enum>
    <name>Enabled</name>
    <abstract>Override region {i} is used</abstract>
    <value>1</value>
  </enum>
</field>
```

Figure 24. Added value IDs and changed access definition.

A dynamic register model should be configurable after it is generated and included in a testbench. Therefore, the dynamic parameters should point to values stored in an external package file. To link register model parameters to the values of the testbench, the parameters could be substituted with paths pointing to the corresponding values, as shown in figure 25. The figure presents a practical example of how parameter values are fetched from a package. The conversion tool currently accepts only specific types of data, for example, only numerical reset values are accepted. The tool would need to be edited so that it can handle, in addition to specific values, paths to values located in the verification environment.

Register groups provide an additional challenge in terms of linking values to the parameters. Because of how groups of registers are defined together in the collection element, the current value ID substitution cannot be used to set different values to specific registers of the group. To bypass this issue, the path used in the substitution should include an element that is replaced by the current register index, as illustrated in figure 25. This same feature is already used for description and abstract elements defined inside collection elements. An example of it can be seen from figure 19.

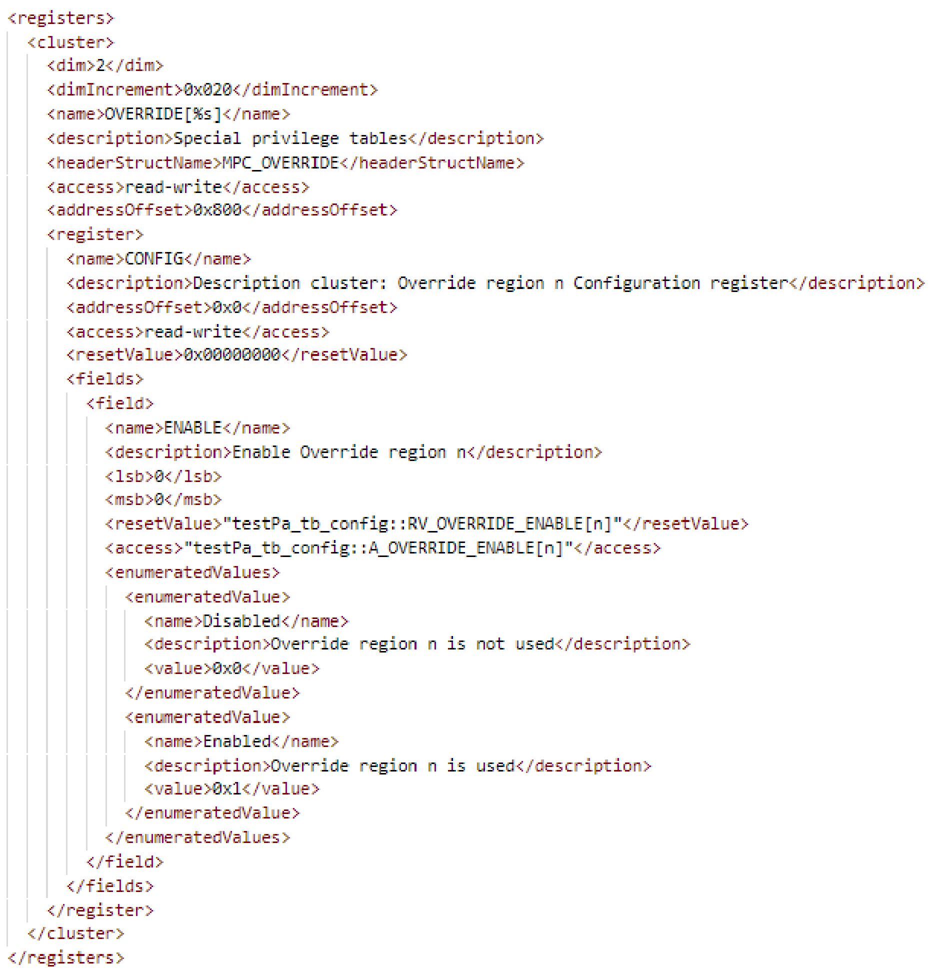


```
<group>
  <peripheral href="ip_example.r">
    <value href="noverrides">[1:0]</value>
    <value href="resetvalue">"testPa_tb_config::RV_OVERRIDE_ENABLE[{i}]"</value>
    <value href="accesspolicy">"testPa_tb_config::A_OVERRIDE_ENABLE[{i}]"</value>
    <instance><base>0x40000000</base><name>TEST</name><abstract>Test instance</abstract></instance>
  </peripheral>
</group>
```

OVERRIDE0_CONFIG
reset = testPa_tb_config::RV_OVERRIDE_ENABLE[0]
access = testPa_tb_config::A_OVERRIDE_ENABLE[0]
OVERRIDE1_CONFIG
reset = testPa_tb_config::RV_OVERRIDE_ENABLE[1]
access = testPa_tb_config::A_OVERRIDE_ENABLE[1]

Figure 25. Value substitution using paths to verification environment values.

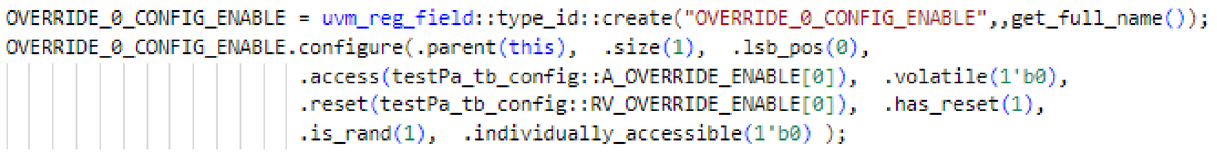
Figure 26 shows the register section of the converted description. To generate the register model with the added paths, the generator script will also need to be edited to handle them with the replaceable elements.



```xml
<registers>
  <cluster>
    <dim>2</dim>
    <dimIncrement>0x020</dimIncrement>
    <name>OVERRIDE[%s]</name>
    <description>Special privilege tables</description>
    <headerStructName>MPC_OVERRIDE</headerStructName>
    <access>read-write</access>
    <addressOffset>0x800</addressOffset>
    <register>
      <name>CONFIG</name>
      <description>Description cluster: Override region n Configuration register</description>
      <addressOffset>0x0</addressOffset>
      <access>read-write</access>
      <resetValue>0x00000000</resetValue>
      <fields>
        <field>
          <name>ENABLE</name>
          <description>Enable Override region n</description>
          <lsb>0</lsb>
          <msb>0</msb>
          <resetValue>"testPa_tb_config::RV_OVERRIDE_ENABLE[n]"</resetValue>
          <access>"testPa_tb_config::A_OVERRIDE_ENABLE[n]"</access>
          <enumeratedValues>
            <enumeratedValue>
              <name>Disabled</name>
              <description>Override region n is not used</description>
              <value>0x0</value>
            </enumeratedValue>
            <enumeratedValue>
              <name>Enabled</name>
              <description>Override region n is used</description>
              <value>0x1</value>
            </enumeratedValue>
          </enumeratedValues>
        </field>
      </fields>
    </register>
  </cluster>
</registers>
```

Figure 26. The converted register section.

Figure 27 shows an example how the register fields are configured in the finished register model. The arguments corresponding to the parameterized properties (.access and .reset) are now configured using the external parameter values.



```
OVERRIDE_0_CONFIG_ENABLE = uvm_reg_field::type_id::create("OVERRIDE_0_CONFIG_ENABLE",,get_full_name());
OVERRIDE_0_CONFIG_ENABLE.configure(.parent(this),  .size(1),  .lsb_pos(0),
                                   .access(testPa_tb_config::A_OVERRIDE_ENABLE[0]),  .volatile(1'b0),
                                   .reset(testPa_tb_config::RV_OVERRIDE_ENABLE[0]),  .has_reset(1),
                                   .is_rand(1),  .individually_accessible(1'b0) );
```

Figure 27. Register field configuration in the generated register model.

As the generated register model and the tests contained in the verification environment use the same parameter values, they are more tightly linked together. This way the properties of the register model can be configured together with verification environment properties, making the verification process more efficient.

5.3.4. Other possible changes

The changes proposed in the earlier section would enable the generation of dynamic register models. In addition to those changes, there are other changes to register description formats and generation tools that should be taken into consideration to increase the efficiency of the register models.

Neither of the used register description formats currently allow the generation of register models that support backdoor accesses. They lack the data structures for defining the HDL paths of the registers. This leads to situations where the verification engineers have to manually edit the generated register models to add in the missing paths if they are required.

The IP-XACT format uses specific elements, called accessHandles, to store the HDL paths of all memory mapped components. Each of the elements stores a portion of the path, and each component combines the elements of the parent objects to form the complete path [12]. A similar solution could be used to define HDL paths in the register description formats used at Nordic Semiconductor.

Another issue is regarding the way multidimensional register groups are implemented in the generated register model. In the register description formats, there are two methods for defining register groups. They can either be defined individually or as arrays. A register group (cluster) that has been defined as an array can be seen from figure 26. Currently the generation script implements each register in the generated register model individually, regardless of the method that is used to define them in the register description. This means that the number of registers is always defined during register model generation, meaning it is always static.

Since register groups can already be defined separately or as arrays in the descriptions, a similar option should be added to the generation script. If they can be generated as arrays, they could then be made dynamic the same way as reset and access parameters, by substituting the group dimension value with a path to a value stored in the verification environment.

Even after the additions considered earlier, the generation flow is still quite unrefined. From the perspective of IP level register model generation, the step of converting the register information from one description format to another is unnecessary. Since all required information is already provided to the conversion tool, it could be extended to generate the register model. The tool consists of a frontend component and various backend components for different functions, such as generating specifications and register description B files. The extension would entail the addition of a backend component for generating register models. Figure 28 illustrates the register model generation flow after the conversion tool additions.

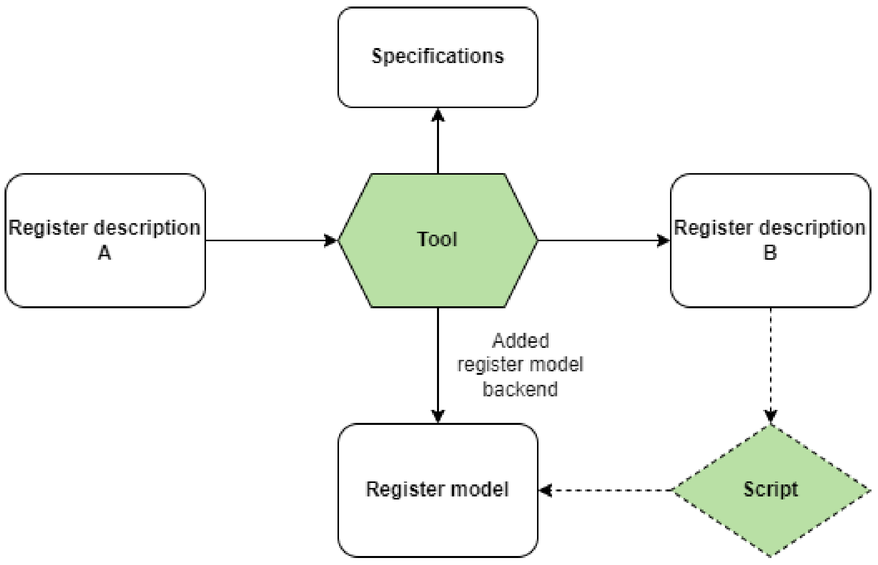


Figure 28. Generating a register model directly using the conversion tool.

5.4. Dynamic register model experiment

This part provides an overview on the practical implementation process for dynamic register models. The improvements proposed in the previous part are validated, and the methods and results are documented. A dynamic register model of an example IP block is generated using the improved flow to see that the proposed changes are reasonable. Next, the register model is tested using a UVM register test sequence. Finally, the functionality of the register model is tested by simulation. The results are compared to those from a manually implemented register model.

5.4.1. The Chosen IP block

The IP block chosen for this experiment was decided on because of its complexity in terms of parameterization. Figure 29 illustrates the way registers are implemented in the IP block. It contains a few individual registers and several register groups, each containing a set of registers.

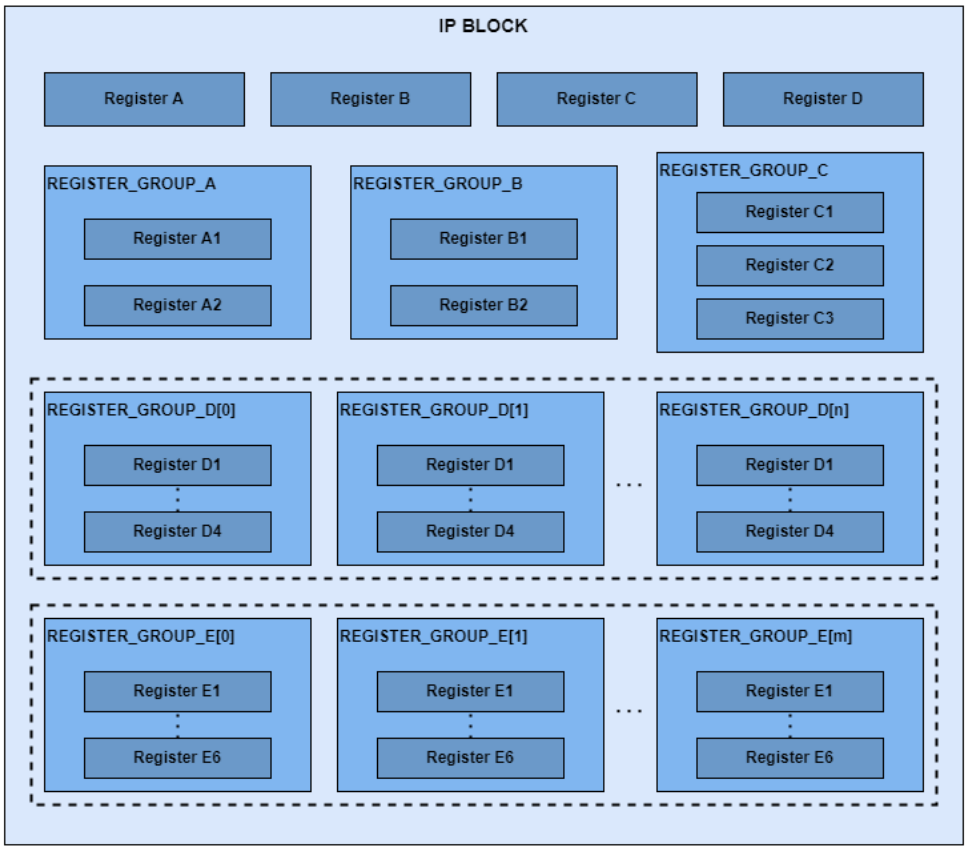


Figure 29. The registers contained in the chosen IP block.

Most of the registers contain fields with parameterized reset and access values. Additionally, the total numbers of register groups, registers, and fields are parameterized. The IP register description utilizes data fields called "collections" that specify the number of instances of the structure it is included in. These structures include register groups, registers, and fields. As shown in figure 29, the chosen IP block contains two register groups, which number of instances can be configured using these collection elements. Furthermore, some registers and fields inside the groups also have configurable number of instances, making the parameterization of the IP multidimensional.

The IP verification environment contains a package that stores the parameter values, a previously made register model and test cases. The values contained in the package are referenced in the verification environment, allowing the properties of the environment to be configured more efficiently. The register model is mostly hand coded, and it is also configured using the parameter values stored in the package. Since it cannot be generated, it is currently also kept in sync with the register description manually.

5.4.2. Generating the dynamic register model

To implement the flow for generating dynamic register models proposed in part 5.3, changes needed to be made to the tools, scripts and register descriptions. The generator tool was extended to support the flow illustrated earlier in figure 28. The register description was edited to allow dynamic access values. Additionally, the testbench and test needed to be edited so that the register names matched the ones in the generated register model. Finally, all additional parameters needed to be defined in the parameter package.

The implementation of backdoor accesses and dynamic collection sizes was left out for now to limit the scope of this experiment. Generation of functional backdoor accesses is valuable for dynamic register models but is not as essential as the other features. Dynamic collections, on the other hand, are problematic to implement when each field of each register can have individual parameter values. If implemented correctly, they would greatly increase the efficiency of the generated register model and decrease the size of the generated file. For the time being, however, they were left static.

The generator tool consists of roughly two sections: a section that parses the register description and extracts all the necessary information, and a template engine that combines the information with a template file to generate the register model. The parsing section needed to be able to handle the new access definitions and other features such as replacing access and reset values with paths to parameter values. To limit the workload, some features of the generator tool were not implemented for this project. For example, the way registers A, B, C, and D shown in figure 29 are defined in the register description differs from other registers, so they are not currently handled. Additionally, as the parsing section was coded using the C# language, the actual implementation of the changes was left to a colleague, who was more familiar with the language.

The template engine takes in the parsed data and a template file, which determines the formatting of the generated register model. For the purposes of this experiment, the template had to be coded basically from the ground up. A template from a separate tool was used as a base, but because it used a different templating language, it had to be edited and expanded to a great extent. The final template was written in the Scriban templating language.

Next, the register description and the generator input file required changes. To enable dynamic reset and access values, access definitions were changed to resemble reset definitions, and each of the parameter values were assigned an ID. Each of the parameter values were then substituted in the input file with a corresponding parameter path.

The parameter package already contained parameters for the verification environment and the old hand coded register model. Most of them could also be used for the register model, but some register model specific ones had to be added. The reset parameters could be used for the dynamic register model as is, but the access parameters had to be redefined.

Next, when the other changes were finished and a dynamic register model had been generated, several testbench files, such as test cases, needed to be updated because of naming differences between the new and old register models.

5.4.3. Running structural tests

To make sure the registers in the dynamic register model have the correct properties, the operation of each register needed to be checked. The structure could be checked using a pre-defined UVM test sequence, and the uvm_reg_bit_bash_seq sequence was the most suited for this purpose. The test sequence goes through each register of the model and sequentially writes 1's and 0's in each bit of the register. It then checks that the bit is set or cleared as intended, based on the field access policy specified for the field it is contained in.

The aim was to use the sequence to check each register and their fields to verify that the dynamic access values use the intended parameter values from the package file. If the values did not match the corresponding design values, and the read and write accesses did not operate as expected, simulation gave errors. In the end, the UVM sequence ran on the dynamic register model as expected, and simulation finished with no errors.

5.4.4. Running functional tests

In addition to the operation of the registers, the functionality provided by the registers should also be verified. This can be done by writing and simulating test cases and checking that the results correspond to the intended operation of the IP. In the scope of this thesis, the functionality of the whole IP block was not tested. Instead, the functional verification focused on testing a few aspects of the IP block. The design functionality had already been verified using the handmade register model, therefore errors during simulation would indicate issues with the generated model or the parameter values. If there were no errors, it could be determined that the functionality of the generated dynamic register model matched the handmade register model.

While running the tests, it became clear that the registers of the register description did not completely match the design registers or the handmade register model. This caused errors during simulation because the structure of the generated dynamic register model follows the description closely. For example, some registers contained different numbers of fields. For this experiment it was better to make a few changes to the register description, in stead of the RTL of the IP.

After making sure the register description and design registers matched, and the reset parameter values in the package matched the corresponding design parameters, the test simulated without errors.

6. DISCUSSION

The central aim of this thesis was to analyze the currently used processes and tools used at Nordic Semiconductor to discover improvements that would improve the implementation process of parameterized register models. A way to generate dynamic register models was found, making it possible to generate significantly more flexible register models for highly configurable IP cores. The validity of the new generation process was verified using functional and structural register verification. Therefore, the aim of this thesis was reached.

Parameterized properties of the registers, such as reset and access values, were configurable using values stored in a separate package file. The functionality of the register model and testbench could also be linked together by configuring the register model using the same parameter values. Additionally, if the register model had to be generated again, the configurations applied in the verification environment would not be lost.

The resulting dynamic register model is significantly more flexible than the previously generated static models. The increased flexibility leads to more efficient IP verification. A single dynamic register model can be used to verify each of the IP configurations, reducing the work required to generate new register models and to manually tinker static register models.

Because changes had to be made to the register description format in 5.3.3, some issues may arise if the process is to be implemented more widely at Nordic Semiconductor. The changes would either have to be left as register model generation specific, or the changes would also have to be taken into account everywhere else the format is used, such as specifications generation. The latter option would require a vast amount of work.

The sizable workload of generating static register models and manually keeping them up to date with the parameterized register description is a serious problem. In addition to the solution proposed in this thesis, other possible methods for implementing parameterized register models have been proposed before. One of those methods is to generate a static register model, and the parameters are then added in the verification environment using a UVM configuration object [36]. This method works similarly to the method used in this thesis: the parameter values are stored separately from the generated register model, and they are fetched and inserted to the parameterized properties in the configuration sections of the model. In this case however, it was suggested that the values are stored in a configuration data base instead of a parameter package.

For this thesis, it was better to use a package to store the parameter values, because the testbench variables were already stored the same way. This reduced the number of different files that had to be handled when changing configurations.

The generation process implemented in this thesis closer to a proof of concept rather than a finished flow. To limit the scope of this thesis, the generation tool cannot yet handle all data structures defined in the register description. This means that some special registers still must be added to the generated register model manually. Additionally, the process can currently handle only dynamic access and reset parameters. It cannot generate register groups, registers, or fields with dynamic dimension parameters, or dynamic address values, so those parameters are currently left static. The dynamic dimension parameters are especially difficult to implement when, in extreme cases, each field of each register in every register group can have individual parameter values.

The generator currently generates individual classes for each register listed in the register description. For most IP cores this is not a problem, but in the case of IP cores with large number of registers, such as the IP described in 5.4.1, the generated models can become inconveniently large. For example, the register model generated during the experiment contained over 105000 lines of code. To increase the efficiency of the model, future work should include implementing a feature which would allow similar types of registers to use a common base class. As mentioned in the section above, each register may have individual parameter values, which should also be considered.

This thesis focused mainly on IP level register models. On sub-system and top levels, the issues with parameterized register models are as severe, if not more. When the flow proposed in this thesis has been made more robust, further study could be done to see, if it could also be utilized on the higher levels of verification. The input file discussed in 5.3.3 can include register descriptions of many different IP cores into a single register model. In theory, it could be used to generate register models for groups of IPs, and the parameterized properties could be replaced with parameter paths the same way as when generating IP level register models. However, this has not been tested in this thesis.

7. SUMMARY

This thesis aims to improve the implementation process of parameterized register models. Differences between static and dynamic register models are discussed, and the different processes and tools for generating register models at Nordic Semiconductor are analyzed. The thesis proposes changes needed for generating dynamic register models and improving the flexibility of the model. The proposed changes are tested by generating a dynamic register model for a highly configurable IP block. Finally, the validity of the model is checked by running a UVM test sequence and functional tests.

As the complexity of chip design keeps increasing over time, the required workload also becomes more and more difficult to handle. New methods have to be adopted so that the productivity of developers can keep up with the increasingly challenging production goals. Several methods have been developed to increase productivity. One of the key methods is the use of reusable design and verification components. For example, reusable IP cores are commonly used in systems and sub-systems containing several IP cores. A common aspect of reusable IP cores is their configurability. To make an IP usable in various scenarios, it should be designed containing configurable parameters. As registers are one of the key elements for controlling the functionality of an IP core, they are also often parameterized.

However, parameterization makes the verification process significantly more complex. As more parameters are added to an IP, the number of parameter configurations that have to be verified grows exponentially. The verification of parameterized registers is especially challenging, as an IP can contain hundreds of registers. Therefore, efficient methods are required to effectively verify the different configurations.

Register models provided by the UVM are an effective way to verify the registers of an IP design by enabling efficient stimulus generation and functional checking. There are two ways for implementing them: coding them manually by hand, or by generating them from a register description. Since coding them manually can be very tedious and prone to errors, the generation approach is often the preferrable option. However, problems arise when trying to generate register models for highly parameterized IP cores.

This thesis analyzed the tools and processes currently used to generate register models at Nordic Semiconductor to discover methods implement parameterized register models more efficiently. The current generation process cannot handle IP cores with many parameterized registers, so the generated model must be manually edited to match the register description. The analysis found that the register description and the generation tool can be edited and extended so that the generated register model is able to fetch parameter values from a separate package file in the verification environment. The changes and other possible improvements to the generation flow were presented.

The discovered improvements and the new generation process were validated by generating a dynamic register model for a highly configurable IP core. The register correctness of the generated register model was tested using the uvm_reg_bit_bash_seq sequence provided by the UVM. Next, it was checked that the IP functioned correctly with the dynamic register model using functional tests. After making sure the register description registers matched the design registers, and the parameter values in the package file were correct, the test sequence and functional tests ran without errors.

The generated register model could be configured directly from the testbench without editing the model manually or having to generate it again. This also meant that the applied configurations would not be lost even if the register model were to be regenerated. The resulting register model was significantly more flexible than the previous generated models.

8. REFERENCES

[1] Patrick D. R. A., Fardo S. W. C. & Chandra V. C. (2007) Electronic Digital System Fundamentals. Fairmont Press Incorporated, 340 p.

[2] Registers in Computer Architecture (read 10.06.2022). URL: https://www.studytonight.com/computer-architecture/registers

[3] Comer D. (2017). Essentials of computer architecture (Second edition.). CRC Press, Taylor & Francis Group, 511 p.

[4] Abd-El-Barr M. & El-Rewini H. (2005) Fundamentals of Computer Organization and Architecture. Wiley, 289 p.

[5] Mathaikutty D. A., Shukla S., & Shukla S. K. (2009) Metamodeling-driven IP reuse for SoC integration and microprocessor design. Artech House, 310 p.

[6] Encinas W. S., Romulo da Silva Araújo F., & Abrahim H. (2016) Infrastructure for formal and dynamic verification of peripheral programming model. In: 17th Latin-American Test Symposium (LATS), April 6-8, Foz do Iguacu, Brazil, pp. 165-170.

[7] Accelera. (Accessed 23.5.2022) IP-XACT seminar at DAC 2011. URL: http://www.accellera.org/images/activities/committees/ip-xact/IP-XACT-at-DAC2011-06072011.pdf

[8] Accellera IP-XACT Working Group. (2018). IP-XACT User Guide. Accellera Systems Initiative. URL: https://www.accellera.org/images/downloads/standards/ip-xact/IP-XACT_User_Guide_2018-02-16.pdf

[9] Banerjee B., Rajan S., & Naidu S. (2011) Automated approach to Register Design and Verification of complex SOC. In: Proc. Design and Verification Conference & Exhibition (DVCon).

[10] Indeed Editorial Team. (read 28.4.2022) What is an XML file and How Do I Open One? URL: https://www.indeed.com/career-advice/career-development/xml-file#:~:text=An%20XML%20file%20is%20an,adhere%20to%20specific%20syntax%20guidelines.

[11] Accelera. (Accessed 23.5.2022) IP-XACT seminar at DAC 2011. URL: http://www.accellera.org/images/activities/committees/ip-xact/IP-XACT-at-DAC2011-06072011.pdf

[12] IEEE Standards Association. (2014) IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows. In: IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009), September 12, pp.1-510.

[13] El-Ashry S. & Adel A. (2018) Efficient Methodology of Sampling UVM RAL During Simulation for SoC Functional Coverage. In: 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2018, pp. 61-66, Austin, Texas, USA.

[14] Zhou S., Geng S., Peng X., Zhang M., Chu M., Li P., Lu H., & Zhu R. (2020) The Design Of UVM Verification Platform Based On Data Comparison. In: Proceedings of the 2020 4th International Conference on Electronic Information Technology and Computer Engineering, pp.1080–1085.

[15] Yun Y., Kim J., Kim N., and Min B., (2011) Beyond UVM for practical SoC verification, In: International SoC Design Conference, November 17-18, Jeju, South Korea, pp. 158-162.

[16] Rosenberg S. & Meade K. A. (2010) A Practical Guide to Adopting the Universal Verification Methodology (UVM). Cadence Design Systems Inc, 344 p.

[17] Litterick M. & Harnisch M. (2014) Advanced UVM register modeling. In: Design and Verification Conference & Exhibition Europe.

[18] ClueLogic. (Accessed 19.7.2022) UVM Tutorial for Candy Lovers - 16. Register Access Methods. URL: http://cluelogic.com/2013/02/uvm-tutorial-for-candy-lovers-register-access-methods/

[19] Verification Guide. (Accessed 19.7.2022) Introduction to UVM. URL: https://verificationguide.com/uvm/

[20] Accellera. (Accessed 19.7.2022) Universal Verification Methodology (UVM) 1.2 User's Guide. URL: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf

[21] Pasricha S. & Dutt N. (2008) On-chip communication architectures: System on chip interconnect (1st edition.). Elsevier / Morgan Kaufmann Publishers, 541 p.

[22] Saleh R., Mirabbasi S., & Greenstreet M., Lemieux G., Pande P. P., Grecu C. & Ivanov A. (2022) System-on-Chip: Reuse and Integration. In: Proceedings of the IEEE, vol. 94, no. 6, IEEE, pp 1050-1069.

[23] Mathaikutty D. A., Shukla S., & Shukla S. K. (2009) Metamodeling-driven IP reuse for SoC integration and microprocessor design. Artech House, 310 p.

[24] Keating, M. & Bricaud, P. (2002) Reuse Methodology Manual for System-on-a-Chip Designs: For System-on-a-chip Designs, Third Edition. Springer Science & Business Media, 291 p.

[25] Junchao Z., Weiliang C. & Shaojun W. (2001) Parameterized IP core design. In: ASICON 2001. 2001 4th International Conference on ASIC Proceedings, October 6, Shanghai, China, pp. 744-747.

[26] Sarkar S., Chanclar G. S. & S. Shinde. (2005) Effective IP reuse for high quality SOC design. In: Proceedings 2005 IEEE International SOC Conference, September 25-28, Herndon, VA, USA, pp. 217-224.

[27] Gajski D. D., Wu A. C. -H., Chaiyakul V., Mori S., Nukiyama T. & Bricaud P. (2000) Essential issues for IP reuse. Proceedings 2000. Design Automation Conference, January 25-28, Yokohama, Japan, pp. 37-42.

[28] Rajsuman R. (2006) System-on-a-chip : Design and Test. Artech House, 289 p.

[29] Givargis T. D. & Vahid F. (2000) Parameterized System Design. In: CODES00: 8th International Workshop on Hardware/Software Codesign, May 1, San Diego, CA, USA, pp. 98-102.

[30] Wile B., Goss J. & Roesner W. (2005) Comprehensive functional verification: The complete industry cycle. Morgan Kaufmann, 704 p.

[31] Litterick M. & Munich V. (2013) Pragmatic Verification Reuse in a Vertical World. In: DVCon 2013.

[32] Maxfield M. (Read 3.6.2022) New Tool Automates Register Verification Process for FPGA, SoC & IP designs. URL: https://www.eetimes.com/new-tool-automates-register-verification-process-for-fpga-soc-ip-designs/

[33] Chen W., Ray S., Bhadra J., Abadir M. & Wang, L. C. (2017) Challenges and trends in modern SoC design verification. In: IEEE Design & Test 34(5), pp. 7-22.

[34] Lovett C., Ramirez B., Secatch S. & Horn M. (2012) Relieving the Parameterized Coverage Headache. DVCon 2012.

[35] Verification Academy. (Accessed 8.6.2022) UVM Cookbook. URL: https://verificationacademy.com/cookbook/uvm

[36] Rosselli S. M. & Falconeri G. (Accessed 19.7.2022) A single generated UVM Register Model to handle multiple DUT configurations. DVCon 2020.

[37] Meyer A. (2003) Principles of functional verification. Newnes, 217 p.