ILMENAU UNIVERSITY OF TECHNOLOGY

Faculty of Computer Science and Automation
Integrated HW/SW Systems Group
Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel

Master Thesis

A Generic Debug Interface for IP-Integrated Assertions

Christoph Kuznik

This master thesis was submitted in fulfillment
of the requirements for the degree

DIPLOMINGENIEUR
(Dipl.-Ing.)

Tutor	Dr.-Ing. Volkan Esen
Professor in charge	Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel
University adviser	Dr.-Ing. Dieter Wuttke
Submission	Ilmenau, 9. December 2008
Inventory no.	2008-12-09/161/II03/2235

Integrated HW/SW Systems Group

Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel

Faculty of Computer Science and Automation

Ilmenau University of Technology

in Cooperation with

Infineon Technologies AG Munich

IFAG ETS DMI AFS SFV

Prof. Dr.-Ing. Wolfgang Ecker

Dr.-Ing. Volkan Esen

Declarations

Herewith I declare, that I have made the presented master thesis myself and solely with the aid of the means permitted by the examination regulations of the Ilmenau University of Technology. The literature used is indicated in the bibliography. I have indicated literally or correspondingly assumed contents as such.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, diese Diplomarbeit eigenhändig und selbständig verfasst, keine als die angegebenen Hilfsmittel und Quellen verwendet und direkt oder indirekt übernommene Gedanken als solche gekennzeichnet zu haben. Diese Arbeit wurde bisher in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt noch anderweitig veröffentlicht.

Ilmenau, 9. December 2008

Christoph Kuznik

Theses

1. Verification of complex heterogeneous electronic systems like System-on-Chip is the bottleneck of every design flow. Due to the design-verification gap and in general a methodology gap, it is unlikely that future designs can be verified to a large extend pre-silicon while keeping track in the time to market.
2. Moreover, assertion-based verification methodology enables early error discovery and information on where and when assertions fail. This information is an important aid in the debugging process and the fundamental reasoning behind the ABV methodology.
3. Therefore, in-system, at-speed silicon validation using assertion checkers will be the next step for successful electronic systems design flows.
4. As electronic system design moves towards software engineering, there is emerging interest for model based approaches to cope with the growing complexity. A promising approach is the usage of meta modeling and the concept of domain-specific code-generation, for example using templates.
5. Using template based code-generation an abstract modeled functionality can be generated for a chosen architecture in an ideal way for every target language.
6. The model representation for a debuggable assertion interface makes usage of these two approaches and generates an interface to access on-silicon assertion. This allows the designer to optionally include silicon checkers within the design.
7. The capture of the debuggable assertion interface within a meta model allows implementation independent specification. Besides, the assertion interface allows the consistent modeling of assertion across multiple abstraction levels.
8. With help of the debuggable assertion interface correct module integration can be verified. Moreover, the monitoring of IP-internal assertion checker significantly improves the observability of internal signals, which may escaped previous pre-silicon verification and simulation steps.

Ilmenau, 9. December 2008

Christoph Kuznik

Acknowledgments

I am delighted to thank Prof. Dr.-Ing. habil. Andreas Mitschele-Thiel from the Faculty of Computer Science and Automation at the Ilmenau University of Technology for the opportunity to write a diploma thesis at Infineon Technologies AG Munich and Dr.-Ing. Heinz-Dietrich Wuttke from Ilmenau University of Technology for his kindly willingness to supervise it.

Besides, I also sincerely thank Prof. Dr.-Ing. Wolfgang Ecker of Infineon Technologies AG for giving me the possibility to process a research thesis topic within the System Design Methodologies department at Infineon. Moreover, I would like to express my deep gratitude to my tutor Dr.-Ing. Volkan Esen for his invaluable suggestions and support. I also want to thank my Infineon colleagues Dipl.-Ing. Michael Velten and Dipl.-Ing. Thomas Steininger for their help, support and encouragement.

Finally I thank the Infineon students of the department and my girlfriend Nicole for the enjoyable time in Munich as well as my family which always supported me during my studies.

Abstract

Nowadays electronic systems design requires fast time to market and solid verification throughout the entire design flow. Many concepts have been researched to raise the level of abstraction during the design entry phase, whereas model-based design is the most promising one. Assertion-based verification enables the developer to specify properties of the design and to get report if these are violated. Assertions are common during development and simulation of electronic products but often are not included in the final silicon. In this thesis an UML-based model defined at Infineon Technologies for capturing design specification information and to generate code automatically using templates, will be extended to allow the description of an abstract debuggable assertion interface for silicon assertions. With help of the assertion interface it shall be possible to verify the correct module integration and to monitor IP-internal assertion checker results. Besides, the code-generation templates for the assertion interface model will be described. To demonstrate the usability of the developed concepts an example system will be introduced to validate the approach.

Kurzfassung

Der Entwurf von Hardware/Software Systemen ist auf eine solide Verifikationsmethodik angewiesen, die den ganzen Design Flow durchzieht. Viele Konzepte haben eine Erhöhung des Abstraktionsniveaus bei der Entwurfseingabe gemeinsam, wobei der modell-basierte Hardware-Entwurf einen vielversprechenden und sich verbreitenenden Ansatz darstellt. Assertion basierte Verifikation ermöglicht dem Entwickler die Spezifikation von Eigenschaften des Entwurfes und die Aufdeckung von Fällen, in denen diese verletzt werden. Während Assertions in Entwurfs- und Simulationsstadien weit verbreitet sind, ist der Ansatz, diese mit auf dem integrierten Schaltkreis (IC) zu fertigen, neuartig. In dieser Diplomarbeit soll ein von Infineon Technologies entwickeltes, auf UML basierendes Datenmodell, welches zur Erfassung von Entwurfsspezifikation und zur automatischen Code-Generierung genutzt wird dahingehend erweitert werden, die Beschreibung für im IC integrierte Assertions zu ermöglichen. Für diese Zwecke wird ein abstraktes Datenmodell beschrieben werden. Das Assertion Interface soll die spezifikationsgetreue Modellintegration gewährleisten, sowie IC interne Assertionresultate dem umgebenen System über das Interface zugänglich machen und damit zum Debugging während der Laufzeit ermöglichen. Ferner werden die Codegenerierungs Templates erläutert und ein Beispielsystem eingeführt, um die beschriebenden Konzepte zu validieren.

Nomenclature

ABV	Assertion-based Verification
AHB	Advanced High-performance Bus
AMBA	Advanced Microprocessor Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
ECSI	European Electronic Chips & Systems design Initiative
EDA	Electronic Design Automation
FSM	Finite State Machine
HTML	Hypertext Markup Language
IC	Integrated circuit
IP	Intellectual Property, in this context hardware or software modules
IP-XACT	The new name of the SPIRIT schema, which will be standardized as IEEE 1685
ITRS	International Technology Roadmap for Semiconductors
MARTE	Modeling and Analysis of Real-Time Embedded Systems
MOF	Meta-Object Facility
OMG	Object Management Group
OVL	Open Verification Library
PSL	Property Specification Language
RTL	Register Transfer Language
SIF	Serial InterFace
SoC	System-on-Chip

SPRINT ..... Open SoC Design Platform for Reuse and Integration of IPs

SVA ..... SystemVerilog Assertions

TLM ..... Transaction Level Modeling

UART ..... Universal Asynchronous Receiver Transmitter

UML ..... Unified Markup Language

VCD ..... Value Change Dump

VHDL ..... Very High Speed Integrated Circuit Hardware Description Language

W3C ..... World Wide Web Consortium

WLF ..... Wave Log Format

XMI ..... XML Metadata Interchange

XML ..... Extensible Markup Language

Contents

List of Figures

List of Tables

List of Algorithms

1. Introduction

1.1. Motivation

The famous Moore's law describes a long-term trend in the history of computing hardware. Since the invention of the *Integrated Circuit* (IC) in 1958, the number of transistors that can be placed inexpensively on an integrated circuit has increased exponentially, doubling approximately every two years. This trend has continued for almost half of a century and is not expected to stop for another decade at least and perhaps much longer.[59] Therefore, more and more heterogeneous parts of an electronic system can be integrated on a single IC. This *System-on-a-chip* (SoC) may contain digital, analog, mixed-signal or radio-frequency functions – all on one chip. Unfortunately, this situation leads to more complex electronic systems design, as designers face a combination of various disciplines, the coexistence of multiple design languages, and several abstraction levels. For example, it is now common that software is an integral part of semiconductor products. The customer expects to have a high-level access to the product. Otherwise the necessary low level design of the system would take too much time. Examples are protocol stacks for communication ICs or software APIs.[33] In this multi-disciplinary context, several gaps in a joint unified software and hardware design flow can be identified. For example, traditional HDL-based designs do no longer suffice to leverage the logical resources on improved manufacturing within the shortening time-to-market windows. Moreover, it is indeed also much more unlikely that such large designs can be verified to a large extend while keeping track in the time to market. Therefore, nowadays verification is the bottleneck and the most costly stage within the entire design flow. Figure 1.1 illustrates this so called design-manufacturing gap. In fact, one can also speak from a methodology gap. Therefore, new methodologies for development and verification are necessary for example starting the implementation from higher level modeling. As electronic system design moves towards software engineering, there is emerging interest for model based approaches within the hardware community. For example, different *Unified Modeling Language* (UML) diagrams and their variations found their application in requirements specification, testbenches, architectural descriptions, and behavioral modeling of electronic systems. Moreover, extensive IP Reuse and early verification must be supported by emerging standards like SPIRIT IP-XACT.[52, 51] The *International Technology Roadmap for Semiconductor* (ITRS) report from 2007[1] affirms these requirements for future verification and design flows. Since perfect logic and timing verification of a complex SoC is practically impossible pre-silicon, postsilicon validation

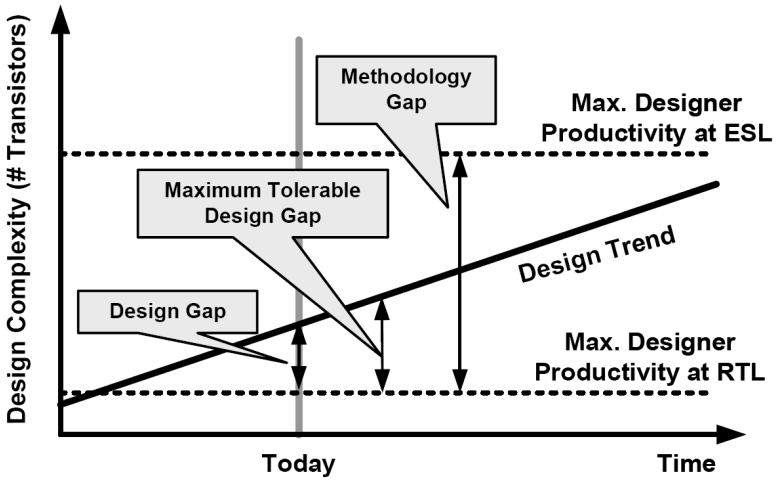

Figure 1.1: Design-Verification Gap[14, P. 3]

has become an essential step in the design implementation methodology. Therefore, in this thesis a meta-model based approach to describe hardware and to generate code automatically using templates, will be extended to allow the description of an abstract assertion interface. This assertion interface will allow to specify properties of a digital design and shall also be expressed via a data model. With help of the assertion interface it shall be possible to verify the correct module integration and to monitor IP-internal assertion checker results. To demonstrate the usability of the developed concepts a widely used bus architecture was implemented in an existing Infineon example system. For the RTL level code generation work VHDL has been chosen.

This thesis was the result of work in the **IFAG ETS DMI AFS SFV** department at Infineon Technologies, headquartered in Munich. The department is dealing with System Design Methodologies including System Verification, TLM Modeling and XML Single Source Methodology.

1.2. Chapter Overview

Chapter 2 briefly describes related concepts and related work of this thesis topic. In chapter 2.1 the aspects of assertion-based verification will be analyzed. Moreover, this chapter will describe the concept and advantages of in-system validation and debug. Apart from that, chapter 2.2 will refer to similar approaches of this thesis work.

In chapter 3 the underlying Infineon design flow for this thesis work will be explained. First

the related fundamentals will be briefly described. Apart from that, an overview about the SPIRIT IP-XACT[51, 52] oriented Infineon Essence meta model, the fundamental concept of the flow, will be illustrated in sub-chapters 3.2 and 3.3.

In chapter 4 the Infineon SPINNI example system for the proof-of-concept will be described. Moreover, the system elements and architecture will be explained. Besides, the additions and enhancements to the example system are introduced in chapter 4.2.

In chapter 5 a requirement analysis for the to be modeled assertion interface meta-model will take place. After the requirement collection chapter 5.2 will present the proposed data model solution. Thereafter, chapter 5.3 describes the target code architecture. Besides, chapter 5.4 will validate the proposed solution against the requirements.

In chapter 6 the template generation will be introduced. Therefore, section 6.1 describes challenges on template development. Moreover, the final templates architecture and the resulting VHDL code will be explained. Besides, section 6.2 will summarize the created plugins and templates during the thesis work.

In chapter 7 the simulation and verification flow for the assertion interface development is introduced. Moreover, the general validation approach and application results of the assertion interface generation are described.

Chapter 8 summarizes the thesis development and gives a short outlook for the possible usage of this approach in the near future.

2. Related Work

This thesis work deals about the development of a hardware debug interface for silicon assertions using a meta model approach. Moreover, in the implementation stage a proof-of-concept example system will be generated. The various related concepts to this overall approach will be briefly described in the following sections. Besides, an overview about existing similar approaches and the differences to this thesis shall be given.

2.1. Assertion based Verification

Assertion based design and more specifically *Assertion based Verification* (ABV) is gaining wide acceptance in the design community. It has been identified as a modern, powerful verification paradigm that can assure enhanced productivity, higher design quality and, ultimately, faster time to market and higher value to engineers and end-users of electronics products.

2.1.1. What is an Assertion

In general, an assertion is an expression for an intended behavior, also called property which can be understood as *"descriptions of valid temporal state"*.[54] This means that assertions can verify combinatorial as well as temporal properties. The assertion statement itself does not contribute any functionality to the element it is being used with. The purpose of an assertion is to ensure the consistency between specification and what is actually created.[20] Assertions are specified using a variety of Boolean expressions as primitives, along with regular expressions and numerous temporal operators. This concept has been used in software engineering for decades. But as the systems to be designed became more and more complex there was the demand to cope with the growing verification problem. So the concept of assertions also was introduced in the field of hardware or hardware-software co-design. If a specified property is violated then the assertion fires. In case, there is a scenario in which the design is *not* working correctly. Information on where and when assertions fail is an important aid in the debugging process, and is the fundamental reasoning behind the ABV methodology.

Assertions for hardware are typically written in a verification language such as PSL (Property Specification Language, IEEE 1850 standard) or SVA (SystemVerilog Assertions), but also HDL have assertion functions which are inner part of the language, but with much less functionality. Another approach is the use of assertion libraries

that can be invoked with common languages like VHDL or Verilog and provide in-built assertion functionality. An example is the *Open Verification Library* (OVL) maintained by Accellera[3] which provides checkers whose functionality can be even modified by adjusting checker parameters. The benefits of use of assertion can be summarized to:[20]

- **error detection**
  Despite the output in case of a black box testing is correct, there are maybe design flaws that affect internal signals in certain conditions. The black box testing approach will not reveal these possible malfunctions. These internal violations can only be discovered with help of assertions.
- **Improving observability, error isolation**
  With help of assertions the source of the bug can be caught closer then before. Even the basic assertion functionality of HDL supports the output of a text message along with the information that the assertion failed. In an ABV enabled design flow tools help to keep track of assertions results and the overall coverage. With all these benefits proper or unexpected behavior of the design can be isolated and fixed.
- **error notification**
  If an assertion modeled property is violated, the assertion fires at once. If used in simulation the simulation kernel will halt and give report to the designer. For silicon-integrated assertion (see chapter 2.1.2) the automatic throws of IRQ are imaginable. In a design flow without assertion it is more difficult to find out where an error is originating from.
- **correct usage checking of interfaces**
  More and more designs make use of third-party IP. With assertions the external vendor of IP cores can take care of the correct usage of the interface to the IP. Using assertions the interface protocols are monitored (mostly only during simulation) and validated using modeled properties.
- **reduced debug time**
  It is easy to see that with help of ABV the overall design productivity is enhanced. Especially in case of errors and debug activities the needed amount of time decreases.
- **faster time to market**

Assertions are used at various stages of the design process. They are mostly used for pre-silicon simulation and verification. In design-entry and simulation tools such as Mentor Graphics Questa[41] the assertions are monitored by the simulation kernel when

a circuit is simulated. If designs are to be emulated in hardware, assertions can not be directly mapped into the hardware because they are written in a higher-level language that is not necessarily amenable to synthesis (see chapter 2.1.2).

Assertion-Based Design practices also advocate the use of assertions as part of the design effort, when used as a formal specification (describing designer intent).[9] The difference between assertions and formal verification is that an assertion models a property which has to be tested with a lot of testbench input to gain acceptable coverage. If the assertion never fails the assumption is that the design is stable and bug free. In formal verification it is calculated if the modeled property will *ever* be false.

2.1.2. Assertions in Silicon Debug

In this thesis work the interface for (silicon debug) assertion shall be developed and modeled via the Essence meta model. Therefore, the benefits of silicon-debug shall be discussed first. For an in detail view about assertion checkers and checker generators please refer to [8]. Detailed resources for in-system debug can be found in [9, 2].

Normally assertions are only used during simulation and pre-silicon verification of a newly developed design. If an assertion fails then the designer knows that the property which is modeled by the assertion is violated. If so, a scenario has been revealed in which the design is not working correctly. So assertions help to discover potential problems within the design. But complete system-level verification of a complex SoC is not feasible pre-silicon, therefore in-system, at-speed silicon validation has become an essential step in the design implementation methodology. This is why assertion checker are useful and also necessary, because the common verification methods

- simulation
- emulation
- FPGA prototyping
- timing analysis
- and formal verification

will not reveal deepsubmicron problems that only occur in the actual device in the desired technology. So special corner cases and analog influences in the real system maybe stay undiscovered. Moreover, just a few minutes of real-time system operation generate more stimuli for an assertion *"then it would get in weeks of simulation or days of emulation,*

thus extending the coverage and the usage of assertion for in-system validation."[2] That is because at-speed, in-system usage under stress conditions introduces many new functional patterns and explores deep states and corner cases not previously encountered, thereby exposing errors that escaped pre-silicon verification.[2]

But how can assertion statements be put on silicon? When the power of assertions is to be used in hardware, a checker generator is used to automatically produce monitoring circuits (also called assertion checkers), from the given assertion statements. The resulting assertion checker is a circuit that captures the behavior of the source assertion. At the moment there are only two established checker generators existing, MBAC[40] from the McGill University Montreal and FoCs[24] from IBM.[1] With help of a checker generator assertions can be applied beyond design and verification. Therefore, the checker generator produces permanent circuitry, e.g. HDL code that can be synthesized and incorporated into the design. This allows the observability of internal signals without requiring new pin and leads to enhanced debug functionality. For example a firing assertion could serve as a trigger to stop the recording in a trace buffer. Using this concept the designer can investigate the history of the input code.[2] These concepts of self-test, on-line silicon monitoring and diagnosis assistance can be used during the *complete lifespan* of the IC.

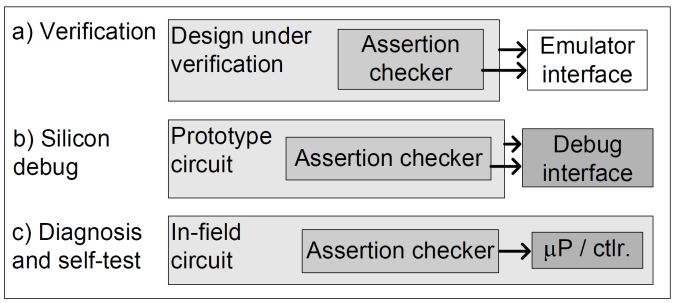

Figure 2.1: Usage scenarios for hardware assertion checkers[9]

There are three scenarios for the usage of assertion checkers (figure 2.1):

- Verification
  The checkers are used only in the verification stage when the design is to be simulated, emulated in hardware (other technology) or executed on a simulation accelerator. Therefore, it will not be included in the final silicon.
- Silicon Debug
  The assertion checkers are used to perform post-fabrication in-circuit debugging

[1] As per November, 2008

for prototypes. The results of the checker can only be validated using external debug hardware. The design is implemented in its intended technology, as opposed to being implemented in reprogrammable logic during hardware emulation. This allows at-speed debugging under expected operating conditions.

- Diagnosis and self test
  The assertion checkers are incorporated in final silicon and can be queried from the system. This allows the design to assess its operating conditions inline in real time. Besides, this concept allows in-field in-chip diagnosis and self test of the design and is from big interest for harsh environments where systems need to check themselves and, in case of failure, make use of redundant parts or perform error correction.

As mentioned above incorporated checkers can test for functional faults and timing issues which can not be fully tested pre-fabrication. By connecting the checker outputs to the proper external equipment or on-board read-back circuits, the user can get immediate feedback on assertion failures in order to start the debugging process.[9]

2.2. Similar Approaches

Within the electronic design community several approaches are existing to either support in-silicon debug or code generation. A public available approach for model based design and code generation is SPIRIT IP-XACT. The complete standard manual can be found in [51, 52]. By now the model-based hardware flow is prototyped at various companies [35] and is being researched within the european SPRINT project.[17, 16] The underlying concept of the Infineon flow, the Essence data model, can be compared to IP-XACT.

An advanced verification approach can be found in [12]. The patented DAFCA ClearBlue technology can be seen as comprehensive approach to in-system silicon validation and debug. Using a software the user can insert reconfigurable instruments for validation. This insertion is done at RTL and the instrumented SoC is processed by standard synthesis-based design flows. The approach supports many validation paradigms such as on-chip signal capture and logic analysis, assertions, stimulate-and-capture, performance monitoring and fault and error injection. Moreover, in case of assertions they can be used in conjunction with embedded logic analyzers to aid the debugging process. For example, an assertion firing can be employed as a trigger to stop recording in the trace buffer; the recorded signals provide a window into activity preceding the malfunction detected by the assertion.[12] Despite this approach is really advanced, comprehensive and powerful, the explanations concerning the underlying generation concept are vague.

Moreover, the DAFCA technology seems to be only target at RTL implementations, whereas assertion can also be used in higher abstraction levels like TLM. Moreover, the main focus is about the checker functionality not the description of a general underlying interface. Two examples for the definition of transaction-level assertions are [22, 15]. But again the concepts are not combined with interface code generation. In general it can be stated that various approaches like [34] introduce the silicon debug assertion functionality, but do more focus on assertion checker generation. Therefore, synthesis of assertion languages is the main objective, not the modeling of an universal interface to them. In this thesis work the main focus is the template based interface generation and the modeling of the interface in an abstract, implementation-independent manner.

3. Infineon XChange Flow

The XML-Methodology project XChange aims to provide an Infineon-wide solution for a single-source design of electronic systems. The most important goals are: [28, P. 12]

- Provision of single source design methodology for whole Infineon
- an unified and consistent data model for Infineon product design data
- Unification of Infineon code generation approaches
- I/O with the SPIRIT Consortium IP-XACT standard[52, 51] to ensure interoperability with third party IP

The proposed approach by Infineon, called Essence, is based on a meta model which can be compared to the public SPIRIT IP-XACT[6] standard. This new design flow concept has several benefits. The single source for component design data, which is reused at different design stages in different groups, allows better consistency of designs and specifications and reduces the error rate of the design process. Moreover, automated code generation reduces design effort and time as well as eases functional redesigns. The design information within the XChange flow is expressed via the Essence meta data model and represented via the use of XMI and XML. Figure 3.1 gives an overview about the database architecture concept. The Essence and IP-XACT meta models are familiar but differ in their main architecture concept. IP-XACT focus is to describe IP, however, Essence has the concept to *generate* IP and documentation. Moreover, Essence supports more features, but is proprietary and thus only used at Infineon Technologies. Within the Essence meta model an information will not appear twice. The redundancy within the meta model is to be minimized. This has a lot of advantages when it comes to data model consistency and effectiveness. On the contrary, it is possible that within an IP-XACT meta data description the same information appears several times. In both meta models *Intellectual Property* (IP) is specified and documented using meta-data. In the XChange Flow it is possible to import SPIRIT IP-XACT descriptions and convert them to the Essence meta model. The Infineon Essence meta model consists of nine UML diagrams[2] whereas each of them models different aspects of the overall system. The UML models provide the elements and rules to describe functionality and architecture in a very abstract and object-oriented way. Specific systems whose models are built to the Infineon meta model standard are represented via a XMI scheme inside a XML data file.

[2] As per November 2008

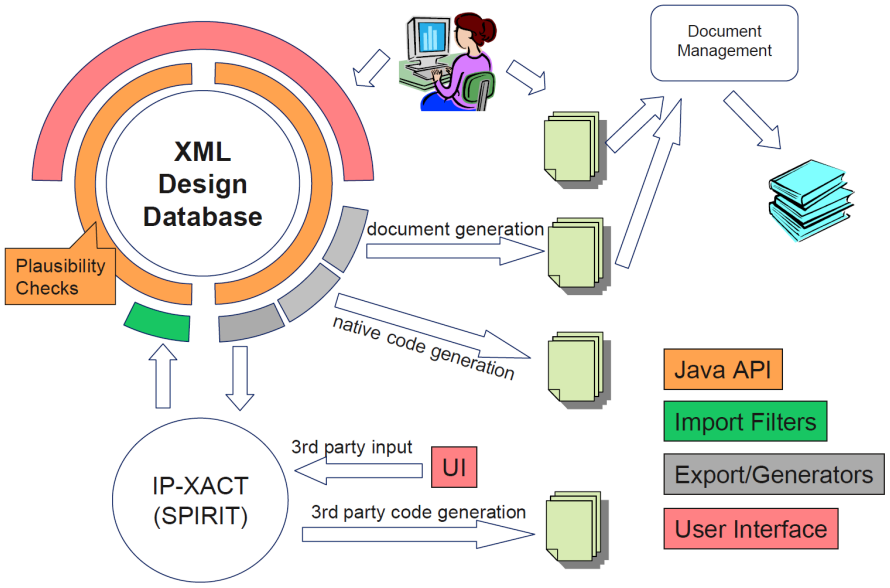

Figure 3.1: Infineon XChange, based on a single-source XML-Methodology[30, P. 9]

It is important to understand the architecture and concept of the XChange flow because requirements for the SPRINT project are proved and tested with help of Essence. If the proof-of-concept is successfully the technical expertise then can be used to issue suggestions for the further development of SPIRIT IP-XACT.

3.1. Fundamentals

In the following sub-chapters the fundamentals and underlying concepts of the XChange and the Essence meta model flow will be discussed. The experienced reader may skip this introductory sections and continues at chapter 3.2.

In chapter 3.1.1 the concept of meta data and meta modeling will be explained. Chapters 3.1.2 and 3.1.3 will provide additional information about the project. Chapter 3.1.4 will give a briefly overview about the *Unified Modeling Language* (UML). Moreover, chapters 3.1.6 to 3.1.8 will give an overview about the tool flow and the underlying concepts of Essence as well as the principle of code generation using templates. Because it was decided to implement a public used bus standard chapter 3.1.9 will give an overview about the AMBA bus protocol family. Moreover, chapter 3.1.10 will focus on the AMBA APB standard which was chosen to be implemented in the example system.

3.1.1. Meta data and meta models

Meta data (metadata, or sometimes metainformation) is ”data about data”, of any sort in any media. An item of meta data may describe an individual date, or content item, or a collection of data including multiple content items and hierarchical levels, for example a database schema. This hierarchy is used in XML files (see chapter 3.1.5). In data processing, meta data is definitional data that provides (additional) information about or documentation of other data managed within an application or environment.[58]

In general a meta model sets foundations on how to build a model. This includes the definition of means of modeling as well as constraints and assertions to define the allowed expression. This definition is the core functionality of the meta model and therefore also the core part of tools. A concrete model itself is based on a meta model and build with the allowed means for modeling. Classes get instantiated and attributes get filled with real-world content. So the concrete model is the desired output of a meta model tool. For a summary please refer to table 3.1.

Meta model	**model**
foundation for a model	based on meta model
model on how to build a model	model to a concrete system
definition of the means for the modeling	modeled with defined means
description of meta classes	instantiation of classes
core of the tool	results of the tools use

Table 3.1: Differences between *meta model* and *model* [21]

An easy example is a dictionary where the model of a language (with its defined semantic and syntax rules) is explained with the means of the same model. The *Object Management Group* (OMG) defines four layers of meta-modeling. Each level of modeling is validated by the next layer. The Level M0 is the runtime level. It contains instances of objects, for example a record. Level M1 defines the model and schema. Level M2 defines the meta model, for example the *Unified Modeling Language* (UML). The most abstract level M3 defines the meta-metamodel, a model that defines a metamodel. More abstractions are imaginable but do not lead to better strength of expression.

The Level M3 is used by the *Meta-Object Facility* (MOF), This concept is applied by XMI (see chapter 3.1.6). MOF can be considered as a standard to write meta models, for example in order to model the abstract syntax of Domain Specific Languages. For more information about MOF please refer to [44, 47]. An example for the MOF hierarchy is

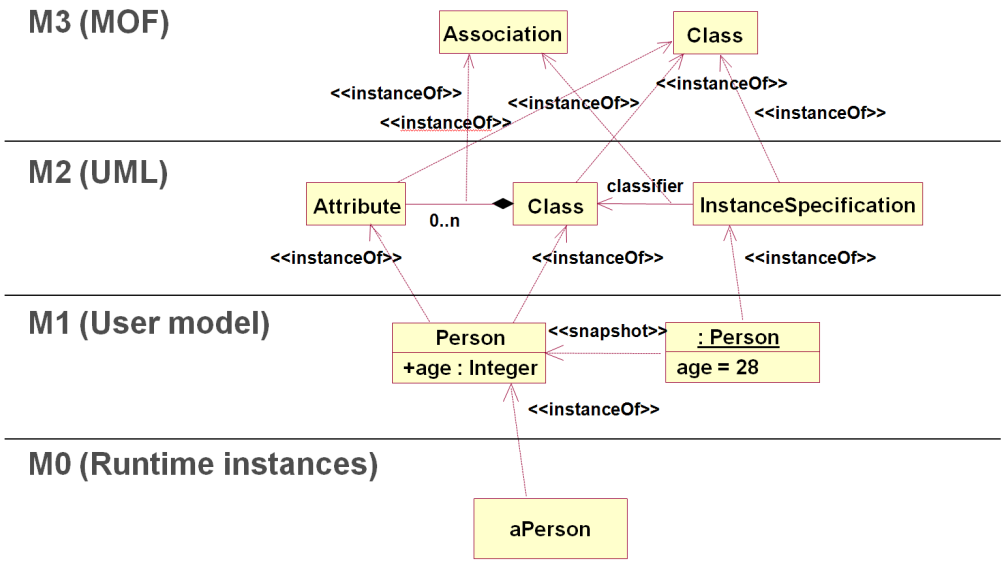

Figure 3.2: Example for the MOF architecture[49]

shown in figure 3.2.

So in the Essence flow, an UML tool is used to capture the meta model as a class diagram. After this java code is generated out of the UML meta model as well as an API for filling the model with information or respectively retrieving the information. The resulting java classes build the foundations for the usage of the Essence meta model. The Java part of Essence is bound with the later explained Python scripting programming language (see chapter 3.1.7).

3.1.2. SPIRIT

The "*Structure for Packaging, Integration and Reuse IP within Tool-Flows*" (SPIRIT) consortium consists of numerous semiconductor and EDA companies as well as fab-less IP creators. The objective was to develop a standard for more efficient integration of IP in System-on-Chip (SoC). To ensure consistency and efficiency on integration of third-party IP there had to be a widely adopted standard existing. Therefore, the Spirit Consortium is developing the meta model based IP-XACT specification within the IEEE P1685 SPIRIT Standardization Working Group.[51] All components of an IP library are incorporated and specified using human-legible text information in XML files which are built in accordance to a specific XML schema. The Essence meta model schema differs from IP-XACT but the general approach to group information into meta data is the

same. Meanwhile numerous companies do rely on IP-XACT, with positive results. In the next versions of IP-XACT more advanced modeling shall be supported. Some of the proposed improvements are currently prototyped in the european SPRINT project.[35]

3.1.3. SPRINT

The SPRINT project[3] is funded by the European Commission's 6th Framework Program under the IST (Information Science Technology) priority which started at the 1st of February 2006.[55] The consortium is supported by the major european semiconductor companies Infineon Technologies and Philips Semiconductors (including research groups at Philips Research) as well as ST Microelectronics. Moreover, several IP Vendors are participating. Among them are ARM, Evatronix S.A. and Syosil. Also EDA vendors and universities do engage in the development. Examples are research groups at Paderborn University, TIMA[53] and the Royal Institute of Technology Sweden (KTH). The global objective is to "*enable Europe to be the leader in design productivity and quality in Systems-on-Chip (SoC) design, by mastering the SoC design complexity with effective standards and design technology for reuse and integration of IP*".[17] Recent global standards such as SystemC/TLM and SPIRIT IP-XACT, which have been driven successfully by SPRINT partners, will be taken as starting point.[18, P. 7] Building on this existing infrastructures, the SPRINT Project will develop new methodologies and standards for interoperability and integration of the high-level IP modules from which modern SoC designs are assembled.[19]

Summarized the key SPRINT objectives are: [18]

- Techniques and standards for IP module modeling that allow the fast simulations required for architecture exploration and early software development, as well as provision of reference models in SystemC/TLM for hardware functional verification
- Definition of standard communication interfaces that simplify the integration of IP modules while also resolving Quality-of-Service (QoS) issues
- Creation of an open SoC design platform that is based upon these standards
- Development of a SoC design methodology, with matching tools and IP modules to automate SoC design, verification and debug
- To enable European companies to be the first in the world to demonstrate and subsequently exploit the new standards-based SoC design environments in an

[3] SPRINT: Open SoC Design Platform for Reuse and Integration of IPs (IST-2004-027580)

interoperable way in order to improve on design productivity and the quality in SoC design

More information can be found on the *European Electronic Chips & Systems design Initiative* (ECSI) website.[16]

3.1.4. Unified Modeling Language

The *Unified Modeling Language* (UML) is a standardized language for modeling software and other systems. It is being developed and standardized by the *Object Management Group* (OMG). UML is a very expressive language and allows in detail modeling of systems without setting constraints about a possible implementation. The information about a system is expressed with the help of several diagram types. Using these UML diagrams the relations and actions in a system can be modeled in an abstract way. To compare it to electronics design, a design can be expressed in a very high level. The different diagram types express different views on the same problem. So the complexity of a given system is reduced.

UML is based on principles of the object technology. The object technology reflects real world objects and offers elements, methods and functions to describe these objects and systems as a bunch of cooperating elements and relations between them. The basic principles of the object technology are

- classification vs. instantiation
- composition vs. decomposition
- generalization vs. specialization
- aggregation
- inheritance

The figure 3.3 shows an example diagram. It contains two classes with their corresponding attributes. The class `address` stays in a "is part of" association with class `person`.

In general the use of UML is not restricted to modeling of software systems. Besides, all kinds of relationships between objects can be modeled due to the generous approach of UML. So for example UML can also be used to model biological, electronic (hardware) or hydraulic systems. Since electronic systems design moved towards software engineering, there is emerging interest to use UML for the hardware engineering. Different UML diagrams and their variations found their application in requirements specification,

testbenches, architectural descriptions, and behavioral modeling.[42] These descriptions are platform independent and therefore offer an ideal start for platform-specific code generation.

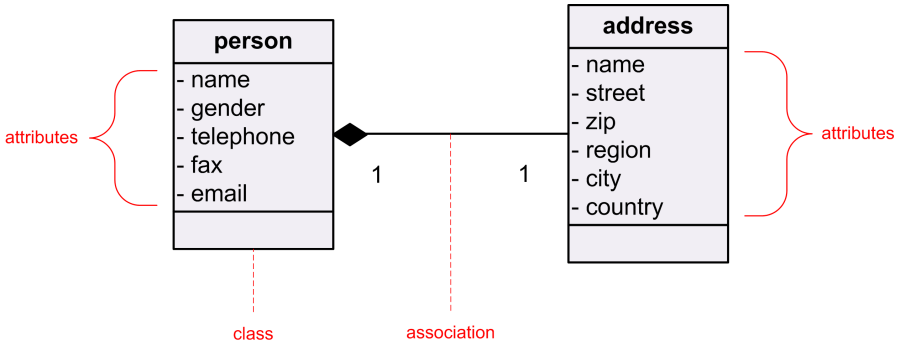

Figure 3.3: Example UML diagram[39]

One of the of the key strengths of UML 2.0[48] is the flexible foundation for customization and extensions of the modeling elements. In detail UML provides a meta model of the modeling elements. So different application domains can be make accessible via so-called UML profiles, which currently receives increasing tool support and give UML great potential to complement current C++-oriented languages for ESL design. In this context

- SysML [45]
- UML for SoC extension [46]
- MARTE [43]
- UML 2.0 Profile for SystemC (ST Microelectronics)

are already available as OMG profiles for Systems Engineering and SoC application and several proprietary profiles are under development.[7] Regarding model exchange between tools, the UML-related XMI (XML Metadata Interchange) format and its relationship to SPIRIT IP-XACT, the emerging IEEE standard, are of additional particular interest. Partial overlaps can be identified and are currently under investigations by some projects, like SPRINT.

Within the Essence flow mainly UML class diagrams are used to define the meta model. For an in detail view about all UML diagram types please refer to [37, 56]. For a comprehensive overview about the usage of UML for electronic system design please refer to [63].

3.1.5. Extensible Markup Language

The *Extensible Markup Language* (XML) is a human-legible specification for creation of custom markup languages. A markup language is an artificial language using a set of annotations to text that give instructions regarding the structure of text or how it is to be displayed.[61] Thus, data is structured in a hierarchical way. XML was developed by the *World Wide Web Consortium* (W3C) and allows the specification of custom languages using XML-schemata. A XML-schema sets semantic constraints, like specifying the hierarchical structure and the allowed content in fields. This benefited the existence of many custom XML languages for information interchange based on meta-date, for example HTML. With help of tools it is possible to check if a XML file complies to a given schema.

Algorithm 3.1 Example for a well-formed XML

```
1  <?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
2  <thesis>
3      <title>example_thesis</titel>
4      <content>
5          <focus>
6              <name>Engineering</name>
7              <picture>verification_gap</picture>
8          </focus>
9          <focus>
10             <name>Related_work</name>
11             <picture>IEEE</picture>
12         </focus>
13     </content>
14 </thesis>
```

A XML file is called well-formed if it fulfills all XML rules. The most important rules are

- one element, the root node, contains all other elements; there is only one root node
- all elements have a begin and end markup
- one element must not have several attributes with identical name

The example shown in algorithm 3.1 is well-formed.

3.1.6. XML Metadata Interchange

The *XML Metadata Interchange* (XMI) is an OMG standard for exchanging metadata information via XML.[47, 62] It can be used for *any* metadata whose meta model can

be expressed in *Meta-Object Facility* (MOF). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels).

XMI integrates four industry standards:

- XML - eXtensible Markup Language, a W3C standard.
- UML - Unified Modeling Language
- MOF - Meta Object Facility, an OMG language for specifying metamodels
- MOF Mapping to XMI

The integration of these four standards into XMI allows tool developers of distributed systems to share object models and other metadata. The XMI source code in listing 3.2 corresponds to the class `address` of the example UML diagram in figure 3.3 on page 16. It is easy to see the even a simple and small "piece of UML" results in many lines of XML syntax with XMI.

Algorithm 3.2 XMI description of the `address` UML class (figure 3.3)[39]

```
1  <?xml version="1.0"?>
2  <XMI xmi.version="1.2" xmlns:UML="org.omg/UML/1.4">
3     <XMI.header>
4        <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
5     </XMI.header>
6     <XMI.content>
7        <UML:Model xmi.id="M.1" name="address" visibility="public"
8           isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
9           <UML:Namespace.ownedElement>
10             <UML:Class xmi.id="C.1" name="address" visibility="public"
11                isSpecification="false" namespace="M.1" isRoot="true" isLeaf="true"
12                isAbstract="false" isActive="false">
13                <UML:Classifier.feature>
14                   <UML:Attribute xmi.id="A.1" name="name" visibility="private"
15                      isSpecification="false" ownerScope="instance"/>
16                   <UML:Attribute xmi.id="A.2" name="street" visibility="private"
17                      isSpecification="false" ownerScope="instance"/>
18                   <UML:Attribute xmi.id="A.3" name="zip" visibility="private"
19                      isSpecification="false" ownerScope="instance"/>
20                   <UML:Attribute xmi.id="A.4" name="region" visibility="private"
21                      isSpecification="false" ownerScope="instance"/>
22                   <UML:Attribute xmi.id="A.5" name="city" visibility="private"
23                      isSpecification="false" ownerScope="instance"/>
24                   <UML:Attribute xmi.id="A.6" name="country" visibility="private"
25                      isSpecification="false" ownerScope="instance"/>
26                </UML:Classifier.feature>
27             </UML:Class>
28          </UML:Namespace.ownedElement>
29       </UML:Model>
30    </XMI.content>
31 </XMI>
```

3.1.7. Python

Python is a general-purpose, high-level programming language which was intended to be highly readable. It aims toward an uncluttered visual layout for example its use of whitespace as block delimiters is unusual among popular programming languages. This design philosophy emphasizes programmer productivity and code readability.[60] Python's core syntax and semantics are minimalistic, while the standard library is large and comprehensive. This large standard library, one of Python's greatest strengths, providing pre-written tools suited to many tasks. Python allows binding and can be a powerful glue language between other programming languages and tools. Algorithm example 3.3 shows the typical whitespace indentation, rather than curly braces or keywords, to delimit statement blocks. An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block.

Algorithm 3.3 Example for Python Code

```
1  # Including functions of library sys
2  import sys
3
4  # definition of a function
5  def run_code():
6      print "Example Code"
7
8  # Example for Indentation
9  arg = len(sys.argv)
10 if arg == 1:
11     print sys.argv[0]
12 else:
13     for i in len(sys.argv):
14         print sys.argv[i-1]
15
16 if sys.platform.lower() == "win32":
17     run_code()
18 else:
19     pass
20
21 sys.exit(0)
```

Python allows to split programs into modules that can be reused in other Python programs. Furthermore, Python is an interpreted language, which can save considerable amount of time during code development because no compilation and linking is necessary. This allows the Python programs to be always read-able and platform independent. The interpreter can be used interactively, which makes it easy to experiment with features of the language, to write throw-away programs, or to test functions during bottom-up

program development.

Python has numerous advantages in comparison to other high-level languages like C++.[36]

- clean syntax due to indentation concept, no variable or argument declarations are necessary
- less coding necessary due to high level data types that allow to express complex operations in a single statement
- object-oriented programming
- support for scientific computing (for example numeric)
- binding to other programming languages, C++, Java well supported

One of the most important modules to ease usage of Essence is GNOSIS.[13] It contains several Python modules for XML processing, plus other generally useful tools. An example is `xml.objectify` which turns arbitrary XML documents into Python objects. So the objects of the Essence meta model that are expressed via a XML schema can be handled the same way like objects from the Essence java class (through Java binding). With help of these two concepts an universal flow of IP is possible. First IP can be captured using the meta model and exported as XML file. Later on the same source can be read in using templates which make use of the GNOSIS module.[31, P. 11]

3.1.8. Templating with MAKO

Mako is a template library written in Python which provides a familiar non-XML syntax. For maximum performance MAKO compiles into Python modules. The syntax and API borrows from the best ideas of many other template engines, including Django templates, Cheetah, Myghty, and Genshi.[5]

In the concept of templating a template engine combines one ore more templates with a data model. The template consists of placeholders where information from the data model can be inserted. Moreover, its possible to have more advanced templates. For example, the template can further contain Mako-specific directives which represent variable and/or expression substitutions, control structures (i.e. conditionals and loops), server-side comments, full blocks of Python code, as well as various tags that offer additional functionality. All of these constructs compile into real Python code. This means that you can leverage the full of power of Python in almost every aspect of a Mako

template. Therefore, the kind output (Text, HDL-Code, HTML, XML etc.) depends on the template itself. The usage of a template engine has benefits:

- enhances productivity by reducing unnecessary reproduction of effort
- enhances teamwork by allowing separation of work (sub-templates)

Conceptually, MAKO is an embedded Python (i.e. Python Server Page) language. This allows the developer to make use of a template engine and Python at the same time.

In the implementation stage of this thesis work the code-generation templates were all written in MAKO. Therefore, its necessary to have fundamental knowledge about syntax and semantics of the language. For a complete reference please refer to the MAKO Documentation.[5]

Syntax

The simplest expression is just a variable substitution using the `${}` construct. In order to work, x has to be known before the substitution. For instance it can be derived from the data model first. Within the substitution all of the functionality provided by Python can be used. Algorithm 3.4 illustrates an easy example.

Algorithm 3.4 Simple Text substitution using MAKO template

```
1 ## this is a MAKO comment
2
3 the value of x: ${x}
4 pythagorean theorem:  ${pow(x,2) + pow(y,2)}
```

The python function `pow(x,y)` returns `x` to the power of `y`. The algorithm 3.5 shows the generated "code" of the template in the case of `x = 10` and `y = 5`.

Algorithm 3.5 Output of algorithm for `x=10`, `y=5`

```
1 the value of x: 10
2 pythagorean theorem:  125
```

Furthermore, the MAKO template language allows the use of control structures and inline python blocks. So the code-generation process can be done in a very flexible way and with help of the comprehensive functions of Python. Algorithm 3.6 shows an example. In contrast to Python where comments are indicated using a single `#` character, MAKO

comments are specified using two `##` characters. Inline Python blocks start with the `<%` and end with the `%>` marker. MAKO control structures are identified using the `%` marker. A noticeable difference between the Python coding style and the MAKO coding style is that indentation is only necessary for Python. In MAKO its not a must and sometimes unrequested. The reason is that every space or tabulator inside the template will also appear in the output, because all content of the input file that is not prefixed with markers or is commented will just appear in the output file. This impedes the generation of output code where positions of the output do matter. Moreover, it makes this kind of MAKO templates harder to read, because there is no indentation used. Fortunately recent design entry tools, in the easiest case text editors, support the developer with hierarchy folding and therefore significant ease template development.

Algorithm 3.6 MAKO Control structures

```
1  ## the next line will appear as it is in the output
2  this is a template
3
4  <%
5      # This is a Python block which calls a somewhere
6      # else defined function that returns a list
7
8      x = get_x_list_fct()
9  %>
10
11 % if len(x) > 1:
12 % for elem in x:
13     element: ${elem}
14 % endfor
15 % else:
16     only one element ${x}
17 % endif
```

A template can be composed of sub-functions and sub-templates. Inside the template a function is declared using the `<%def>` and `</def%>` markers. It exists within generated Python as a callable function.

Algorithm 3.7 MAKO `def` statement

```
1 <%def name="myfunc(x)">
2  this is myfunc, x is ${x}
3 </%def>
4
5 ${myfunc(7)}
```

3.1.9. AMBA Overview

The *Advanced Microprocessor Bus Architecture* (AMBA) is a widely used open standard for an on-chip bus system which is defined by ARM Limited. The AMBA standard was introduced in 1996 and aims to ease the component design, by allowing the combination of interchangeable components in the SoC design. It promotes the reuse of intellectual property components and is the de-facto standard for 32-bit embedded processors because it is well documented and can be used without royalties.[57, 23] Therefore, many third party IP providers use an AMBA bus for interfacing to their IP. Within the SPINNI System it shall be possible to include these IPs, assumed an Essence (or at least IP-XACT) description is existent. The AMBA standard defines different groups of buses

- Advanced eXtensible Interface (AXI)
  all from below plus more abstract, channels etc.
- Advanced High-performance Bus (AHB)
  high performance, pipelined transfers, burst transfer, split transfers, multiple bus master, bus arbiter
- Advanced System Bus (ASB)
  multiple bus masters, high performance
- Advanced Peripheral Bus (APB)
  one bus master, simple interface, limited functionality, designed for low power

which are typically used in a hierarchical fashion. The figure 3.4 shows an example. The AXI standard is the most powerful bus with elaborated functions. The APB is the most basic bus for peripheral components that do not require massive data throughput. The AMBA specification is technology independent. So physical implementation details like voltage levels are not dictated by the specifications. In fact, it gives an overview about the architecture, the signal count, their names, transfer timing diagrams as well as basic state machines.

For the proof-of-concept work both APB and AHB shall be bridged within the SPINNI system. But the functionality of the AHB shall be reduced significantly. First of all some concepts of the AHB, like pipelined transfers and multi-master, are not meaningful within the SPINNI system because then the peripheral bus would be much more powerful than the central system bus, called `SimpleBus`. The SPINNI Example system is described in chapter 4. So as long as the `SimpleBus` is a "functionality bottleneck" inside the SPINNI system the AHB bridge shall have the same functionality as the APB bridge. The AHB

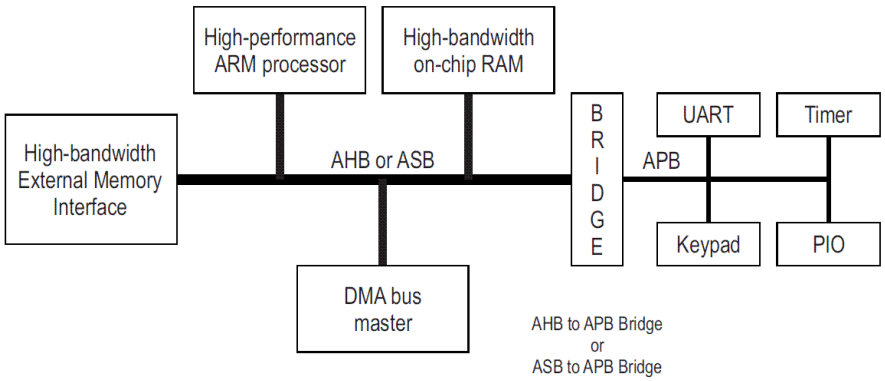

Figure 3.4: A typical AMBA enabled system[38]

components are very similar to the APB ones, but differ in the used interface definition. Therefore, the signal names will differ. For this reasons only the APB protocol will be explained in detail.

3.1.10. AMBA APB

The APB is part of the AMBA 3 protocol family and is optimized for minimal power consumption and reduced interface complexity. The APB has an unpipelined protocol where every transfer takes at least two clock cycles. Furthermore, transfers can be extended via low on the `PREADY` signal. All signal transitions are only related to the rising edge of the clock to enable the integration of APB peripherals easily into any design flow.[38]

The figure 3.5 shows the state diagram of the operating states of the AMBA APB protocol. If no transfer is requested the FSM remains in the `IDLE` state. The bus slave select signal `PSELx` stays low. Therefore, no slave is selected. The signal is named `PSELx` to indicate that there are actually several select signals, one for every APB slave. If there is a transfer requested the FSM changes the state from `IDLE` to `SETUP`. The select signal of the desired APB slave will get `HIGH`. The AMBA APB specification gives no proposal how this slave is determined. In the SPINNI Example System several bits of the address signal are used for an address decoder.

In the second cycle the FSM will always move to the `ACCESS` state. The enable strobe signal `PENABLE` will get `HIGH`. Moreover, the address, write, select and write data signals must remain stable during the transition to the `ACCESS` state.

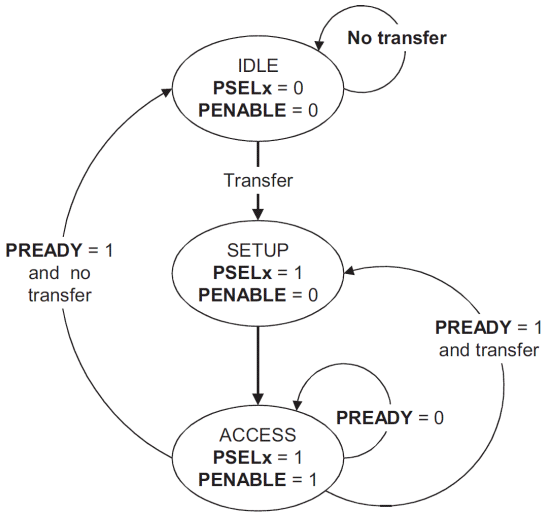

Figure 3.5: APB Bus Transaction State diagram[38]

Now the data to be written is read on the selected slave or in the other case, the to read data is put on the bus by the selected slave. If the two cycle flow of the APB can be met, the `PREADY` signal will be `HIGH`. If the data phase takes more than one cycle the `PREADY` signal will stay `LOW`. When the transfer is completed the next action depends if there is a new transfer request. If so, the FSM will go directly to the `SETUP` state and save one cycle. If not it will go to the `IDLE` state again. In order to reduce power consumption the address signal and the write signal will not change after a transfer until the next access occurs.[38, P. 5-5]

More information about the APB protocol, for example timing diagrams for read or write transfers, can be found in the AMBA APB specification.[38] In chapter 4.2 a `SimpleBus`-to-APB bridge will be implemented for the SPINNI Example System.

3.2. The Essence Meta Model

In the following the Essence data models will be briefly described. Despite the additional SPINNI system enhancements are expressed by means of the core Essence data models, the additional `ModelConfig` data model (see figure A.1 on page 85) is the most important one for this thesis work. Therefore, the explanations will not focus in the other data model but will still try to give an overview about the architecture. The Essence data model is intellectual property of Infineon Technologies. Therefore, the core data models are not shown within the thesis or appendix.

The Essence Meta Model contains nine basic model types. All of these models make usage of the principles of the object technology. So inheritance and composition are widely used. If an object is inherited from an existing one, it inherits all attributes and methods from the existing one. Apart from that, it can contain additional attributes and methods. This concept is called specialism. In nearly every Essence model all objects inherit attributes from the class `SingleSourceNode`. This class provides basic attributes like `Identifier`, `Name`, `Description`. The `ID` is of particular interest, because each element within a single Essence model XML file representation has an unique `ID`. These basic attributes will then be inherited to derived classes. The composition is a part-whole relation. In a graphical UML notation the rhomb points to the whole, the opposite side to the part. The existence of the part objects depends on the existence of the whole objects. In Essence the method of composition is used to declare which elements can contain one-to-many other elements. For instance in the UML example in figure 3.3 on page 16 a person has exactly one address. So the resulting XML structure of the datamodel is declared implicit. The meta model itself was designed and modeled with help of Enterprise Architect[50], an UML tool capable of generating Java source code. The documentation for the Essence meta model is also generated automatically. It is derived directly from the Java source code with help of javadoc. Javadoc is a documentation generator from Sun Microsystems for generating API documentation in HTML format. The work of this thesis is especially related to the `ModelConfig` data model because the functionality of the debuggable assertion interface was incorporated into it.

3.2.1. Component Data Model

Components are the only design elements that can contain collections of registers or memories. These internal structures can be modeled in a very detailed manner. Therefore, Component Data Model XMLs are often very large. Moreover, Components can have several Interfaces.[26] For each Interface the `Role`, i.e. Master or Slave, is specified. The properties of the Interface are modeled in the Interface Definition data model. With help of the `Role` information a code generator then can determine what ports to generate.

3.2.2. Interface Definition Data Model

The *Interface Definition* (IFD) data model defines the interface of a component/system. So a component XML and the interface definition XML are in close relation to each other. Using an IFD it is possible to declare interface protocols once, and reuse them

later in other components. This strategy speeds up design and makes coding less error-prone. Moreover, the interface can be modeled as seen from different views/abstractions called, `InterfaceDefView`. Examples are RTL and TLM level. In TLM the protocol is described very abstract via function calls and not via physical signals. Within the `InterfaceDefView` all signals of the interface. e.g. the protocol are specified. Each signal has a `Name`, `DataType` and unique `ID`. Protocols often have Master and Slave Roles. This information is specified in the `InterfaceDefRole`. For each Role, for example Master, the used signals of the protocol, and their directions are specified. With all these model elements it is possible to express very complex and different protocols within the Essence meta model. The references to elements in the data model and the relation between different data models are expressed via the XRef data model which will be declared next.

In the implementation stage of this thesis work the proof-of-concept system was modeled in the RTL level, because VHDL code-generation was projected. The figure 4.5 on page 41 illustrates the architecture of a resulting example `InterfaceDefinition` XML file.

3.2.3. XRef Data Model

The XRef data model contains all references from one element to other ones. The target identifier for a reference is always the `ID`. Note that the IDs are only unique per XML file. For each `XRef` there is a corresponding `ExtVLNV` field in the superior hierarchy. Therefore, a references needs a pair of `ExtVLNV` and `XRef`. An example can be seen within the ModelConfig data model on figure A.1 on page 85 and in the developed assertion interface model illustrated in figure 5.3 on page 51. So using the `ExtVLNV` the corresponding XML file can be determined and opened. The next step would be to search for the declared ID within that file. Fortunately, there are several convenience functions, like `getXRefTarget()`, available withing the Essence API that ease the reference retrieval. The data mode itself offers varies types of references. Examples are `XRefSlaveInterface`, `XRefInterfacePort` or `XRefSignal`. `VLNV` has the meaning of `Vendor`, `Library`, `Name`, `Version` and can be seen as the identifier of the XML file. An Essence XML file must be named like the `VLNV` of the `RootNode`.

For example, the XRef data model is used for assigning signals to an interface definition. Moreover, it is also of use within an abstract assertion interface data model to associate signals to an assertion.

3.2.4. Bus Data Model

The bus data model only consists of a `SingleSourceNode` and `ModelRoot` pair as well as the object `Bus`. Using attributes of the object `Bus` the minimum and maximum participants of the to be designed bus can be specified. With use of the `ExtVLNV` field this model points to an external Interface Definition that declares all types of signals and ports of that bus.

Within a system, such a bus component can be instantiated and other elements can be connected to it.

3.2.5. System Data Model

The system data model is properly the most complex essence data model. It is the only model that allows instances of other data models to be instantiated and connected.[26] Consequently within the system, there can be instances of components, buses and subsystems whose interfaces are might connected. So an Essence system data model can be compared to a VHDL top level entity, in which VHDL entities get instantiated and wired. Furthermore, the system can have Interfaces too. Using the `LocalInterfaceMap` the System's interfaces can be mapped to internal instances of objects using the `Connection`. A connection is a 1-to-1 mapping relation. For example the `clk` port of a system interface can be mapped to a component instances interface that is instantiated within the system.

3.2.6. ModelConfig Data model

The Essence data model holds on implementation independent information. However, it is also necessary to provide implementation and flow data as meta data as well. Therefore, the XChange Flow has an additional data model called `ModelConfiguration`. The figure A.1 on page 85 shows the data model architecture. Its purpose is to store all language dependent and flow specific information. For example special tool flows (the generator environment) and batch command flows that are necessary for a generation can be put into that model. Other examples are language dependent port types like `std_logic` and `bit_vector` for VHDL respectively. Despite not being language dependent the `AssertionInterface` model shall also be included in the `ModelConfiguration` data model because at the moment it is the only allowed place to add extensions to the Essence data model.[4] The `ModelConfiguration` model is not mandatory for an Essence

[4] As per November 2008

design. But it is possible for a design to have *one* corresponding `ModelConfiguration` data model. Using a naming convention rule it is possible to check for the presence of a `ModelConfiguration` data model file for a given Essence model. Within the ModelConfig XML the link to its corresponding Essence Object is done via the `VLNV` and the `RefXMLType` field. The figure 3.6 depicts this relation.

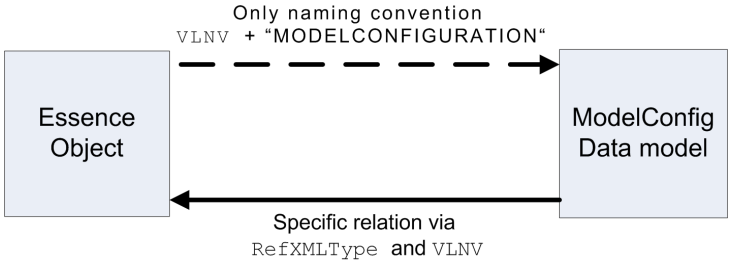

Figure 3.6: Relation of Essence model and ModelConfig model

So the generator template can check for an optional ModelConfig XML of the current Essence object and consider its information during the code-generation process. Because the assertion interface will be placed inside the ModelConfig data model, this concept allows the *optional* generation of the assertion interface.

3.2.7. Other Data Models

Apart from the already mentioned data models there exist some models which declare the elements used by the other data models:

- DataType Data model
  defines the data type definitions
- Variables Data model
  constants, parameters, generics
- Miscellaneous Data Model
  enumeration types
- Connectivity Data Model
  all elements for interconnection between objects

It is not necessary to introduce these models in detail for purposes of this thesis work. For a detailed description please refer to the Essence documentation [26] or the Essence class diagrams themselves.

3.3. Toolchain

To build the enhancements for the example system and to model the generic assertion interface data model the Infineon Essence tool flow had to be used. Using the toolchain it is possible to build up new Essence data models or perform code-generation with help of templates. Therefore, all written templates and python code are used along with this toolchain. Consequently it is meaningful to give an overview about the tools and their underlying concepts.

The Infineon Essence flow is based upon several independent tools and sub-flows. The main end user tools to create and work with Essence XML files are `essimport` and `essemplate`. In general the aim is to "*ease the consistent generation of TLM and RTL register interfaces and HW/SW interfaces using Essence XML descriptions as a single source.*"[31, P. 1] Both tools are written in the Python programming language. This has several benefits. First of all python is platform independent and offers good binding functionality. Moreover, its an object-oriented approach. The initial idea behind this approach was to find a flexible template engine which supports binding to C++ and Java. This template engine should be *the* template language/engine for Infineon. The binding to Java was necessary to interact with the compiled Essence Meta Model.[32, P. 4]

In a nutshell the functionality of the tools can be described as the follows

essimport enables the Python based creation of Essence XML files, the conversion of XML files from one Essence version to another, and an import of an arbitrary XML format into an Essence XML description

essemplate generates arbitrary target code from an Essence XML file with help of templates (generators)

Sections 3.3.1 and 3.3.2 will give a detailed view about the architecture and principles of both tools.

3.3.1. Essimport

Figure 3.7 shows the two possible ways how to build an Essence-conform XML. The tool `essimport` is used to create a XML file which complies to the Essence Metadata model, in most cases with the help of a plugin. The plugin itself is written in Python and makes extensive use of the Essence API.

The Essence API provides convenience functions to create and instantiate the elements for the desired data model, for example a component datamodel. Furthermore, these

newly created objects can then be filled or set to valid content. The output file of this flow is an Essence XML file. In this case, there is no other input XML file needed. The desired content for the output Essence XML is specified within the plugin. This file can later be used for code generation with help of `essemplate`. The essimport API is accessible through the `api` constructor argument of the plugin.

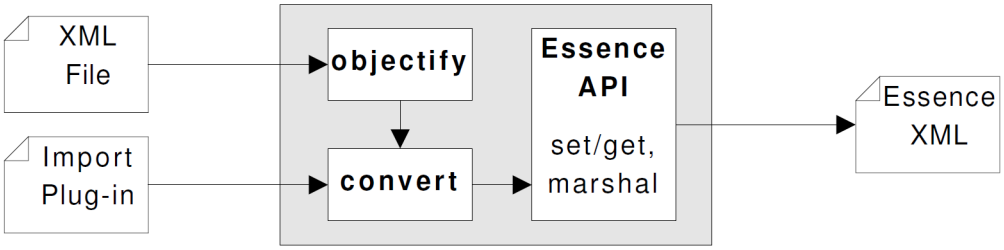

Figure 3.7: `Essimport` Flow[31, P. 7]

It is also possible to convert between different Essence XML data model versions. A Python conversion plug-in and an optional arbitrary XML file are the input and the Essence XML is the output. Within the tool the XML input is objectified using the Python Gnosis library. The obtained object structure is iterated with the conversion plug-in. [31, P. 7]

Algorithm 3.8 shows a simple example for the usage of essimport. The plugin file `simple_component.py` includes a class with two functions. The `__init__` function is a python statement and is automatically called when the `simple_component.py` is loaded. With the help of the Essence API a component is created. Within the `executePlugin()` function the Name attribute of the newly created component is set. If this plugin is used along with `essimport` the output will be a Essence Component XML file.

Algorithm 3.8 Example of an `essimport` plugin with set of an attribute(written in Python)[31, P. 12]

```
1 class simple_component:
2     def __init__ (self, api)
3         self.api = api
4         self.Component = api.Essence.createComponent()
5         self.executePlugin()
6
7     def executePlugin(self):
8         self.Component.setName(simple component)
```

3.3.2. Essemplate

With the `essemplate` tool it is possible to generate arbitrary target code from an Essence XML file using templates. This provides the possibility of easy code generation for different target applications (e.g., the register interface of RTL or TLM models or firmware header files). Therefore, the Essence meta model API is linked with a template engine to allow the access to the XML data directly from within the template. The principle of using templates allows the separation of model and view. In this case the data provided by the Essence XML equals the model and is separated from the the target code to be generated, which equals the view.[29, P. 16]

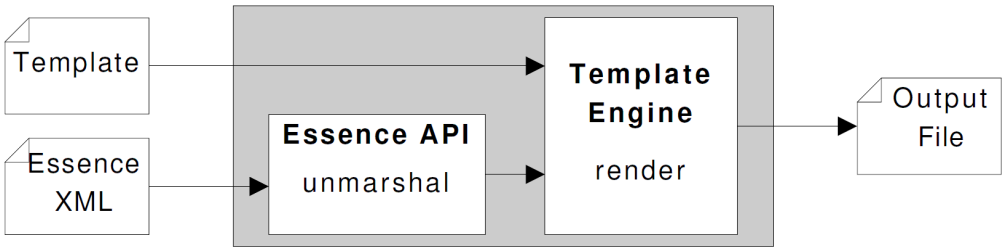

Figure 3.8: `Essemplate` Flow[31, P. 15]

As template language the Infineon developers have chosen MAKO. The MAKO template engine supports the complete Python scripting functionality. So inside the template blocks of arbitrary python code can be placed. The template itself can be composed of hierarchical templates. Hence, it is possible to make reuse of sub-templates where ever possible. Figure 3.8 shows the basic flow for the `essemplate` tool. The output file type depends on the template. So a template could generate VHDL, SystemC or just outputs text after applying tests to the Essence XML. Moreover, on each essemplate run only a single output file can be generated.[5] If it is intended to generate multiple files out of the same source multiple tool calls are necessary.

[5] As per November 2008

4. Infineon SPINNI Example System

For the validation and testing of the Essence meta model approach an Infineon example system called `SPINNI System` was existing. In order to demonstrate the usability of the assertion interface concept it was decided to implement a widely used bus architecture within the system. Using an advanced bus the functionality of the assertion interface could be exhausted to a greater extend. Therefore, the AMBA protocol family, especially APB and AHB[6], were chosen to be represented within the Essence meta model. However, such a subsystem was not available and thus it was developed within this thesis as well. For this approach several preparations where necessary. First the protocols had to be expressed with means of Essence, in particular the `Component`, `Bus` and `InterfaceDefinition` data model. After this an extended Essence System had to be created. Moreover, code-generation out of the data model should be performed along with the development. As target RTL language VHDL was chosen.

The chapter 4.1 explains the existing SPINNI example systems architecture and components. In the following chapter 4.2 the improvements and additionally modeled elements will be discussed.

4.1. Existing System

4.1.1. Architecture

The Infineon SPINNI system is an example system for proof-of-concept of the XChange flow as well as the prototyped work from the SPRINT project. The figure 4.1 gives an overview about the architecture. The whole system is using the single-source XML-based flow and is getting generated out of the Essence data model using templates. The main system consists of the open source MLITE CPU and a very basic system bus called `SimpleBus`. The RAM is the only directly connected component on the `SimpleBus`. Besides, several bus bridges translate the `SimpleBus` to other bus standards. The *External Bus* (XBUS) and *Peripheral Bus* (PBUS) are "down-sized" versions of the `SimpleBus`, and therefore very similar to it. This is because the address line is scaled down due to the address decoder within the simple bus. Typical peripheral components on the PBUS are the UART and GPIO. For the test of these components *Testbench Elements* (TBE) are used which model the exterior interaction with the peripheral components.

[6] in fact only a subset of AHB

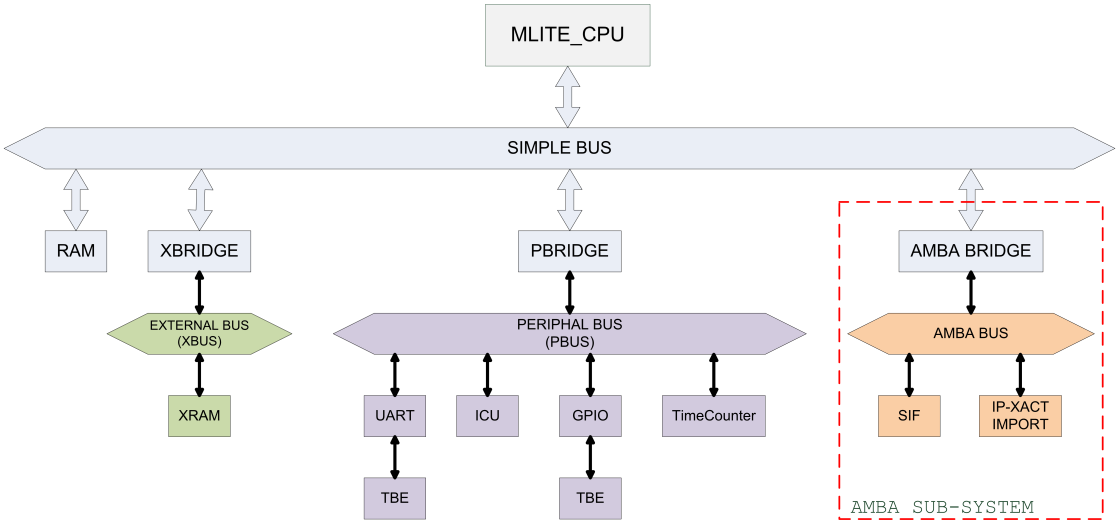

Figure 4.1: Infineon SPINNI System with intended enhancements

The implementation of the `AssertionInterface` shall be done within an AMBA subsystem. To interface the subsystem with the existing SPINNI system it was necessary to develop a `SimpleBus` to AMBA Bridge and the corresponding Essence XML files. Therefore, chapter 4.1.2 will summarize the concept and signals of the `SimpleBus`.

4.1.2. Simple Bus Specification

The `SimpleBus` is the central system bus of the SPINNI example system. It connects the CPU, the RAM, and several bus bridges within the SPINNI system. The naming convention for the `SimpleBus` signals is

`S_<signal_name><width>_<direction>`.[27]

If a signal is not a vector the width is skipped. The data and control signals from the CPU are switched to the `SimpleBus` slaves. On the other hand, data from the RAM and the bus bridges is OR-ed and transferred to the CPU. Slave selection is done via an address decoder. The figure 4.2 shows the interface specification of a general `SimpleBus` slave.

The slave selection is indicated via logical one on signal `S_AccEn_i`. The `SimpleBus` is an one-phase protocol. Therefore, once a slave is selected all other signals also hold their intended values. The signal `S_Wr_i` indicates read or write access. The datain and dataout signals are named `S_Data32_i` and `S_Data32_o` respectively. The two bit signal

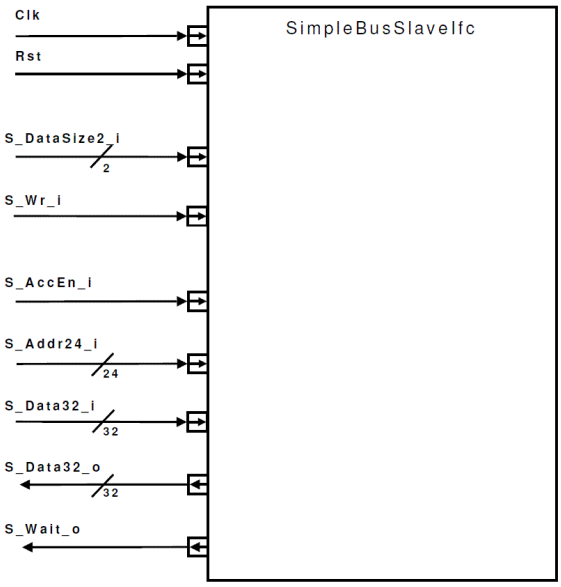

Figure 4.2: Interfaces of a `SimpleBus` slave[27]

`S_DataSize2_i` indicates byte, half-word or word access. The address signal has only 24-bits of former 32-bits on the CPU side left due to the previous address decoder in the bus itself. Furthermore, the bus wait signal `S_Wait_o` allows to extend a SimpleBus transaction. If the transfer cannot be to accomplished within one cycle, `S_Wait_o` goes to one. Then the CPU is waiting for the bus transaction to end before issuing new requests on the `SimpleBus`.

4.2. Extensions to the SPINNI System

In the following sub-chapters the developed extensions for the SPINNI System are explained. Within the data model all new components were implemented in a way that allows RTL level code-generation, because the usage of VHDL was projected. The overall work was developed in a divide-and-conquer approach. First the development concentrated on the single elements of the AMBA Subsystem (see figure 4.1) and then the sub-system itself. After this the assertion functionality was prototyped on the system. In the last stage an integration test along with real HDL Code simulation was performed. During the thesis work APB and AHB components and systems were developed. Because the AHB components only differ in the signal names and do not offer more functionality then the APB, the following sub-chapters will focus on the APB components only.

4.2.1. APB Bridge Architecture

For the development of the SimpleBus to APB bridge it is advisable to know about the signaling of both SimpleBus and AMBA APB. Chapter 4.1.2 on page 34 gives an overview about the SimpleBus, chapter 3.1.9 on page 23 about APB. The bridge itself is slave on the SimpleBus. The AMBA specification gives an example of an APB Bridge, because the APB is often used with AHB altogether (see figure 3.4 on page 24). The bridge has to fulfill the following requirements:

- protocol translation and synchronization of SimpleBus and APB cycle-accurate signaling in accordance to the respective bus protocol
- propagation of wait signals in case of read or extended write transfers
- support for consecutive APB transfers, word/half-word/byte read and write decoding

In figure 3.5 on page 25 the 2-phase APB protocol is illustrated using a FSM. Usually a n-state state machine is implemented using parallel VHDL processes. Each process is clock independent but shifts its output to the next process on clock. This is called phase-oriented design approach. It is very common for large pipeline designs, like the n-stage pipelining inside processor cores. The APB consists of only two phases, the setup and data phase. This results in a quite easy FSM. Therefore, the VHDL implementation is realized in style of the AMBA APB state machine from figure 3.5. This eases the understanding of the VHDL code and is not too ineffective due to the low phase count.

The general interface of the SimpleBus-to-APB bridge is shown in figure 4.3. To speak in the Essence philosophy, the APB bridge has a SimpleBus interface of role Slave and an APB interface with role Master. Within the bridge byte, half-word or word access to the data signal is decoded via 2-bits of the SimpleBus address signal. Apart from this the bridge implements the APB FSM which will be discussed in detail in chapter 4.2.2. The APB slave selection has been put into the APB according to the Essence philosophy to divide component and bus functionality. Moreover, demuxing the different APB slaves output signals is done in the APB too. The APB address signal PADDR width has been chosen to 32-bit in spite of only 20-bits have meaningful values because of previous address decoders. The APB protocol does not specify the data signal width, but to use common numbers 32-bits were chosen.

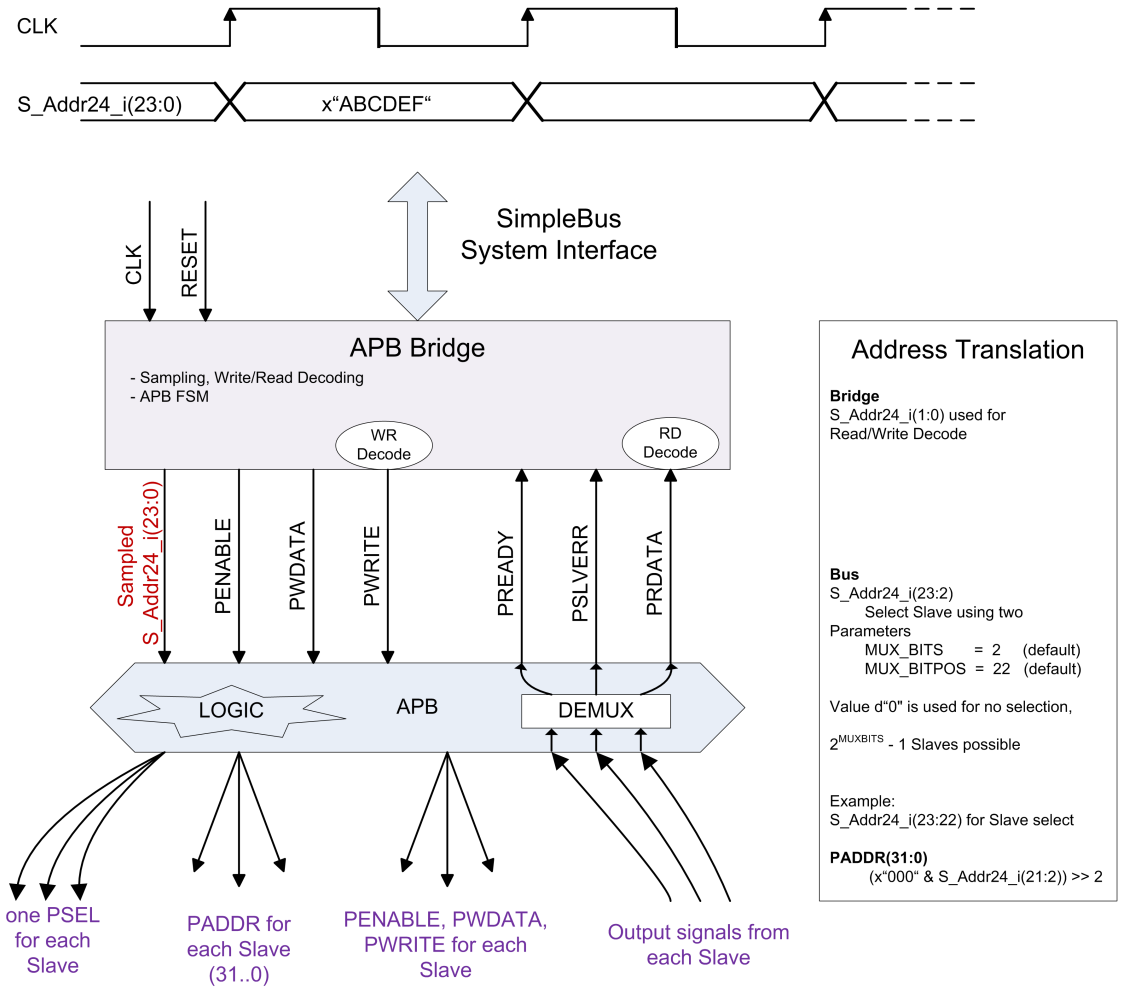

Figure 4.3: APB Bridge Overview

4.2.2. APB Bridge Behavior

Within the `SimpleBus` to `APB` bridge a FSM is translating requests from the `SimpleBus` side to peripherals on the APB. The state machine consists of three states. Inside the target-code language VHDL they are labeled `S_<state>` to ease the readability of the text. The figure 4.4 illustrates the FSM states and their transitions. In the following each state shall be discussed in detail.

An example code-generation of the APB bridge can be found in the appendix A.2. It is quite helpful to comprehend the following explanations.

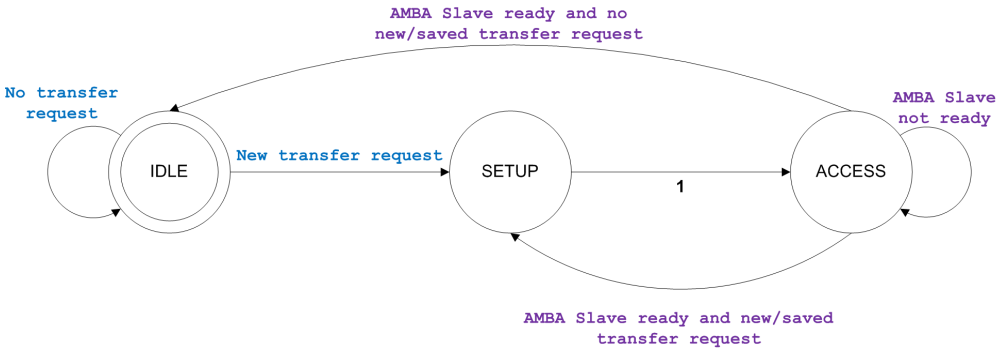

Figure 4.4: `SimpleBus` to `APB` finite state machine for bridge

IDLE State

If the `S_AccEn_i` signal goes to `HIGH` the bridge is selected on the `SimpleBus`. The `SimpleBus` other signals indicate what action is requested on the Slave. To be on the safe side the bridge samples these signals first using several VHDL variables all prefixed with `sample_<signal_name>`. The signal `sample_S_Wr_i` is now used to determine the requested transfer type.

On `HIGH` a write transfer shall be performed. Therefore, the `sample_S_DATASIZE2_i` signal is used to determine what part of the `sample_S_DATA32_i` shall be put on the APB. Options are the whole word, half-word or a byte. This function has been swapped out into a VHDL package. The `SimpleBus` protocol specifies a mapping table using two bits of the address signal `sample_S_ADDR24_i`. Because the outer system does not need to wait for the bridge to continue it is not necessary to set `S_Wait_o` to `HIGH`.

On `LOW` a read transfer is requested. This operation takes at least 2 cycles. Therefore, the `SimpleBus` wait signal `S_Wait_o` has to be set `HIGH`.

In case of a transfer request the next state will always be `S_SETUP`, else the FSM remains in `S_IDLE`. To stick to the Infineon Essence philosophy and data model style, the slave selection will be performed within the APB.

SETUP State

First the following FSM state is assigned, which is always `S_ACCESS`. Inside the APB the selected slaves APB `PSEL` signal is set to `HIGH`. The slave is selected using an address decoder. In the `S_SETUP` state the APB signals `PENABLE` and `PWRITE` get assigned with the pre-sampled `sample_S_Wr_i`. If the current transfer is a read request the `S_Wait_o`

signal is held LOW. It is tried to assign every output signal in each state to avoid synthesis problems.

If the current transfer is a WRITE transfer the S_IDLE state has not put the S_Wait_o signal to HIGH. Therefore, it could be that the system queries another transfer in the S_SETUP state. If so, the transfer is sampled in a second set of VHDL variables prefixed with pipe_sample_<signal_name>. It must be stated that APB is not a pipelined protocol. The bridge implementation allows right-after-another transfer on the APB bus only.

ACCESS State

The S_ACCESS state contains the most VHDL code because the transfer-after-transfer cases needs to be implemented here. Moreover, it is allowed to extend an APB transfer in AMBA protocol version 3. For example a read request on peripheral components can take longer then one cycle of the system clock. So at first it is checked if PREADY equals HIGH, which means the APB slave is ready.

If so the PENABLE signal is set to LOW again. The other APB signals stay as they are to save energy. If the ongoing action is a read transfer then the bridge decodes the APB data in signal PRDATA. This is done the same way like the write decoding, using an outsourced mapping function which returns word, half-word or a byte. The SimpleBus wait signal S_Wait_o is set to LOW again so that in the next state S_IDLE an eventually new transfer can be requested from the outside. It must be noted that a pipelined read transfer is not possible due to the specific way of the SimpleBus wait generation. This action would result in an infinity process activation loop. This issue will might get revised in the next SimpleBus version.[7]

If the ongoing action is a write transfer, then all related signals are held during the S_ACCESS state. Only in the write transfer case it is checked for an already sampled subsequent transfer, or a new transfer request in the S_ACCESS state. If there is a saved transfer request from the S_SETUP state the pipe_sample<> signals are assigned to the sample_<> signals and the same actions are taken as in the S_IDLE state. Moreover, the next state will be the S_SETUP state because the actions from the S_IDLE state were already taken. This saves one cycle and allows consecutive transfers.

If PREADY is not HIGH then the next state will always be S_ACCESS until it is. Moreover, it is checked if another transfer request is issued to the APB bridge. If so, it is sampled

[7] As per November 2008

and can be processed once the previous transfer has completed and PREADY changes to HIGH.

4.2.3. APB

In order to create an Essence representation for the AMBA APB it was first necessary to create an APB InterfaceDefinition. Therefore, a python plugin was written. With help of essimport a corresponding Essence InterfaceDefinition XML was created. All signals of the APB were declared. The datawidth of the datain / dataout signals was chosen to 32-bit. Because the essimport plugin and the resulting XML are very bulky they are not illustrated here. Instead, the figure 4.5 gives a brief overview about the resulting XML file.

4.2.4. APB Peripheral

To test and develop the APB as well as the SimpleBus-to-APB bridge a simple APB slave was written. This slave is a simple Essence component with an interface of role slave of the APB interface definition. The slave component contains a number of registers. Their amount is specified via a parameter within the XML file. The template is using this information to generate an array of integer within the VHDL file. An example code generation of the APB Slave can be found in the appendix A.3. The register read or write functionality can be implemented without a state machine. Using an asynchronous reset process it is checked if PSEL equals HIGH. In case, some bits of the address line are used for the register selection. Depending on the PWRITE signal the datain signal is assigned to that register or the dataout signal is assigned with the registers content. PREADY and PSLVERR were not used in this first simple slave. Therefore, they are bound to 1 or 0 respectively. So the transfer will be finished after one cycle, and PSEL will become low. When this happens the dataout signal will go to x"0000_0000" again to allow simple OR-decoding within the bus.

In addition to this first easy implementation another component was written that could be instantiated within the simple slave. This additional component also contained an interface to the superior components registers. So in the target language VHDL, if data was written to the APB slave, the incorporated component could access it. This additional component contained a simple *Serial Interface* (SIF) functionality, realised using a FSM. A SIF shifts out parallel data in a serial way. The register's information was again used to specify various test options, like

- back/forward shift
- invert bits
- start/end pattern.

Moreover, using the SIF the byte, half-word and word access as well as the wait generation and propagation over the bridge could be verified. Besides, this functionality made the overall sub system more complex and therefore more realistic to make use of the assertion interface later on.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<InterfaceDefinition>
    <Custom></Custom>
    <Hidden>false</Hidden>
    <ID>1</ID>
    <LongDescription>
    <Name>spinni_apb_bridge_ifd</Name>
    <ShortDescription>IFD APB Bridge, APB Side</ShortDescription>
    <ConstDefBlock>
    <EssenceVersion>210</EssenceVersion>
    <GenericDeclBlock>
    <ParamDeclBlock>
    <VLNV>
    <AddressUnit>32</AddressUnit>
    <DataUnit>32</DataUnit>
    <InterfaceDefView>
        <Custom></Custom>
        <Hidden>false</Hidden>
        <ID>7</ID>
        <LongDescription>
        <Name>RTL</Name>
        <ShortDescription></ShortDescription>
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <Custom></Custom>
            <Hidden>false</Hidden>
            <ID>8</ID>
            <LongDescription>
            <Name>PSEL</Name>
            <ShortDescription>Indicates that this certain APB Slave is selected</ShortDescription>
            <DataType>
                <SignInterpretation>unsigned</SignInterpretation>
                <ObjectType>dig</ObjectType>
            </DataType>
        </Signal>
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Signal xsi:type="WireSignal" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <InterfaceDefRole>
        <InterfaceDefRole>
    </InterfaceDefView>
</InterfaceDefinition>
```

Figure 4.5: APB `InterfaceDefinition` XML snapshot

4.2.5. APB Subsystem

After the individual components had been implemented and tested, the entire sub-system was set up. Therefore, it was first necessary to write an `essimport` plugin to generate the Essence system data model XML. Within the plugin references the used Essence objects `Component` and `Bus` where specified. During the development more and more components were instantiated and connected. As a reminder, the optional assertion interface was implemented for the AMBA Bridge and subsystem. The final subsystem is depicted in figure 4.6.

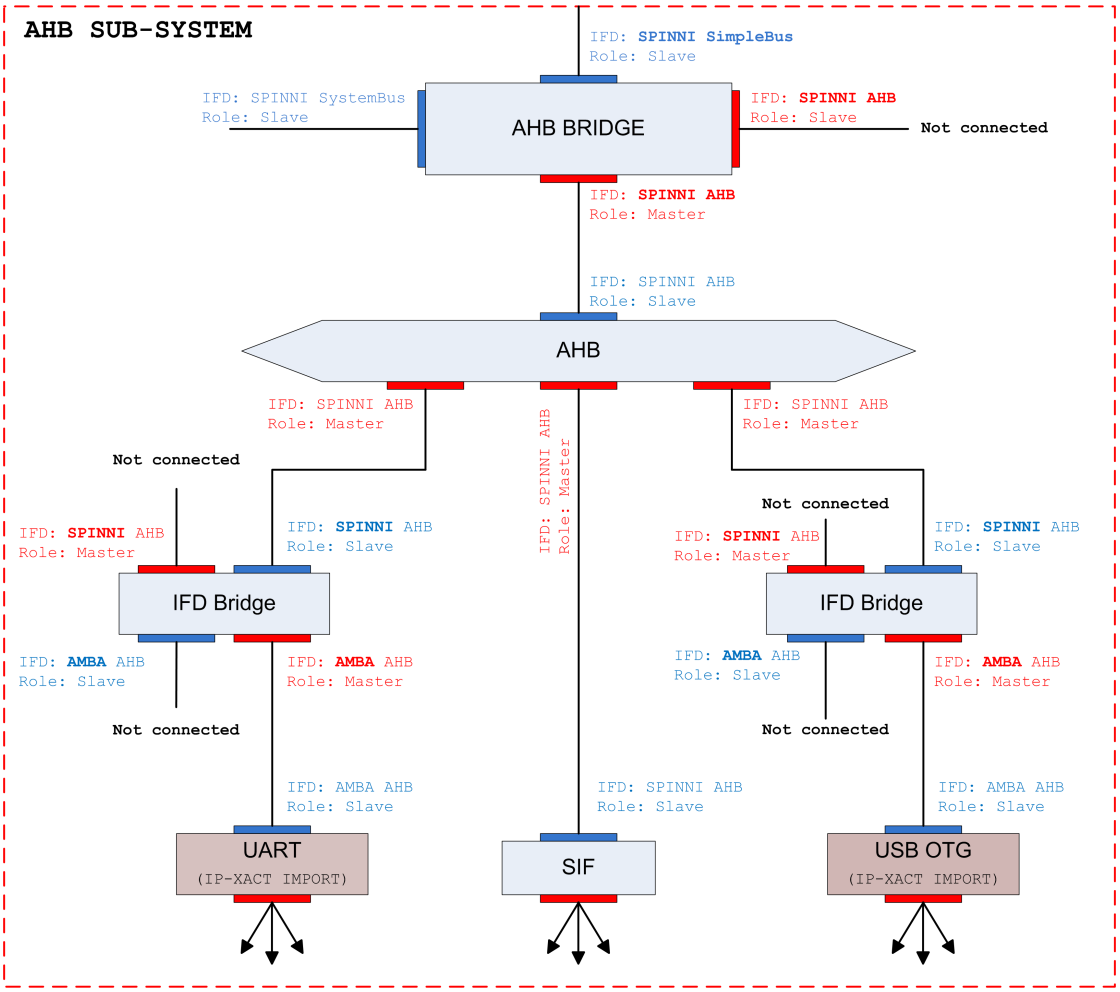

Figure 4.6: The AMBA subsystem in detail

So in the end the overall SPINNI example system contained several components, the `SimpleBus` and the AMBA subsystem. Despite not related to the assertion topic, it

was tried to connect IP-XACT components in the test system within the final stages. During this process several problems occurred because the used `InterfaceDefinition` for the AHB protocol in the SPINNI system were not the same as in the imported components. The background is that the SPINNI system only contains a downgraded version of the AHB, for example without multi-master capabilities, therefore, some signals were not necessary at all. To solve this problem an `InterfaceDefinition` bridge was implemented. This bridge is a simple component with all roles from the to be bridged interfaces. Not used interfaces remain unconnected.

For the superior system please refer to figure 4.1 on page 34.

4.2.6. Interface Mapping

The assertion interface data model shall model assertions and involved signals. Moreover, it shall provide the access to the assertion registers. But in general the interface of the component and the interface of the registers may differ. In an abstract view the interface and the core logic of a component can be seen separated. So within the Essence flow a component can be generated using another referenced `InterfaceDefinition`. This results in different ports names, width and count. So in general there has to be a mapping of the interface side of a component and the internal registers interface (if present). Moreover, the Essence component model allows very detailed descriptions of internal registers, for example their length and bitfields. Each of these capillary register elements can have individual read or write enables, depending what actions are allowed for the specific register area. This allows a lot of possible options. For example, the component interface could only offer a 8-bit `datain` signal. On the contrary the register itself could be 32-bit wide. In this case there are several options on how the `datain` data can be put into the register. Within the Essence example system there was no general mapping mechanism existing yet.[8] Therefore, it was necessary to develop a mapping template first. Within the Essence data model interfaces are specified using three parameters. Table 4.1 gives a briefly overview.

[8] As per November 2008

Parameter	Component Interface	Register Interface
Data Width (DW)	width of signal	number of bits in a register
Address Unit (AU)	an address increments by 1 shifts, the access point increments by AU-bits	an address increment by 1 shifts the access point within the register by AU-bits, therefore allows to address every AU-bits of the DataUnit separately
Data Unit (DU)	smallest payload width	smallest unit in bits with independent access enable

Table 4.1: Essence Interface Parameters

If we consider only the values 32-bit, 16-bit and 8-bit for the three parameters of each interface this results in many possible options. Three parameters for both interfaces result in $3^6 = 729$ cases. To ease the research if a generous approach is feasible a Python script was written that performs the following options:

1. Build up a table of all 3^6cases
2. Apply Filters #1
   only integer multiplies of parameters are allowed, for example

   $$DataWidth_{Interface} = n \cdot DataWidth_{Register}$$

   whereas `n` is integer. This condition must be fulfilled for the other parameters as well.
3. Apply Filters #2
   Because there were many options left, it was tried to focus on simple and common cases first. Therefore, with help of example cases some additional constraints where defined that would allow a moderate mapping effort. For example, the condition

   $$DataWidth_{Interface} \leq DataWidth_{Register}$$

   must be fulfilled. Otherwise the write case to the register would cause trouble. Moreover, the condition

   $$AddressUnit_{Interface} \geq AddressUnit_{Register}$$

   was defined, else it would not be possible to access every actually individual addressable part of the register.

Apart from that, several other constraints where defined. These additional constraints will not be discussed in detail at this point, because the main thesis focus is about the assertion interface generation and not on the interface mapping. The remaining simple subset of mapping cases was implemented within a mapping template. The performed actions within the template can be summarized to

- duplication of the read or write enable signal depending on relation of the `DataUnit` of both interfaces
- sliced `DataWidth` mapping of the register content with the equal or smaller interface content, the remaining bits within the register get set to zero
- simple address translation, calculates the access point depending on `AddressUnit` of both interfaces

Figure 4.7 shows an example where the address translation constraint is violated. The interface `DataUnit` is 32-bit, the `AddressUnit` is 8-bit. This allows to "address" every 8-bit of the 32-bit independently. In contrast the register has `DataUnit` 32-bit and `AddressUnit` 16-bit. Therefore, within the register every 16-bit can be accessed independently. These addressable parts result in offset addresses (purple numbers). In this example, the address translation can not be performed for all offsets.

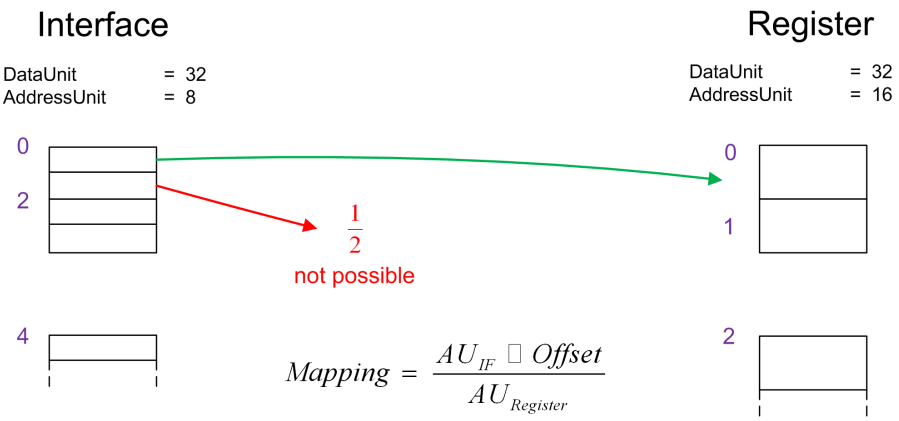

Figure 4.7: Constraint violation on Interface Mapping

5. Debuggable Assertion Interface

This chapter introduces the key concepts of the debuggable assertion interface which was developed within this thesis work. The interface exposes IP-internal assertions to the bus, such that it is possible to retrieve assertion information and results, as well as controlling assertions, i.e., enabling and disabling. This interface supports also interaction with on-chip debug components which are utilized by hardware debuggers.

The overall approach consists of collecting requirements for the assertion interface, defining the structure of this interface, and to capture it within a the Essence meta-model, in order to utilize code-generators. As a reminder the model will not contain the real synthesized assertion functionality itself, but will provide a generative interface to allow access to the results and config registers of every assertion. Moreover, the model shall be as abstract as possible and shall not imply a certain target-code or target-architecture. The `AssertionInterface` data model will model the concept and the assertions interfaces in a high-level and language-independent, tool-independent way. Moreover, it was decided that the Essence component and System data models should be supported. Therefore, for these two models it shall be possible to optionally define an assertion interface model.

First the requirements for the future `AssertionInterface` data model will be gathered in chapter 5.1. Moreover, an architecture decision, benefits and tradeoffs will be discussed. Chapter 5.2 will introduce the developed data model. In chapter 5.3 the target code architecture will be described. Chapter 5.4 summarizes how the previous collected requirements are fulfilled by the proposed solution.

5.1. Requirement Analysis

For the development of a general model representation for a debuggable assertion interface a lot of consideration has to be done. These requirements shall be axiomatic and independent from implementation. But during the research activities one has to keep in mind that the proof-of-concept code-generation in the implementation stage will be based on VHDL. Therefore, the axiomatic requirements have to be checked against the needs of RTL-level code generation, in specific VHDL. This is because if we have a well-formed data model but it is not possible to allow code-generation for register-transfer level HDL, the model is of no use. Therefore, the research process can be seen as a meet-in-the-middle approach. Besides a paper research about assertions and their functionality has been done (see [9, 2]). In the following sub-chapters the summarized requirements are presented.

5.1.1. Hardware

The assertion data model has to be consistent through multiple abstraction levels. So the models information must be of use for electronic system level (ESL) design approaches like TLM but on the other hand it must be also possible to generate register-transfer level HDL code. Furthermore, the assertion interface shall be independent of the property implementation. Therefore, the model shall have no impact on design criteria and independent from implementation. Above this, it shall be possible to include an assertion interface optionally. This means in one case it must be supported to generate a system out of the model which has just the core logic, in the other case also the in-built assertion interface has to be included. Besides the assertion interface shall support on-chip debug support (OCDS). This means that the result of assertions can be read from within the system. For example, a firing assertions could write an error code to a register or cause a hardware interrupt. Then the surrounding system can query the assertions status via the assertion interface and react on the property violation. These hardware requirements for the assertion interface can be summarized to:

Req.1 no impact on design criteria, independent from implementation

Req.2 consistent through model abstraction (TLM, RTL)

Req.3 not necessarily included in final silicon

Req.4 Option to interrupt on assertion failures, OCDS

5.1.2. Architecture

There are numerous architecture requirements for the assertion interface. The architecture defines how the assertion interface is build, how the flow of information is organized. These requirements do not define the later code-architecture in detail, but also give an idea what has to be supported by the target-code. For example, it must be possible to access every single assertion within the interface, in the easiest case to enable or disable it. Moreover, it shall be also possible to group similar assertions, or families of assertions, into scopes. Besides, it shall be possible to perform a broadcast to all assertions within all scopes. So all assertions could be configured simultaneously. Furthermore, advanced debug operations shall be supported. An example is the reset of the assertion in its initial stage. A system could also perform polling upon the assertion and access coverage data in real-time. An enhanced option would be to make use of a hardware interrupt.

These architecture requirements can be summarized to:

Req.5 Enabling / Disabling of every IP Assertions

Req.6 Reset Assertion, Reset results or failures

Req.7 Grouping the Assertions to scopes

Req.8 Broadcast to all assertions

5.1.3. Bus Interfacing

In difference to the architecture requirements which specify how the assertion is built, the interface consideration is about how the assertion can be accessed. As mentioned in chapter 5 the assertion interface shall be used for Essence component and system data models. If we think of such a data model, e.g. a component, how shall the assertion functionality be accessed from the outside? There are three different possible implementations. The figure 5.1 shows the available options.

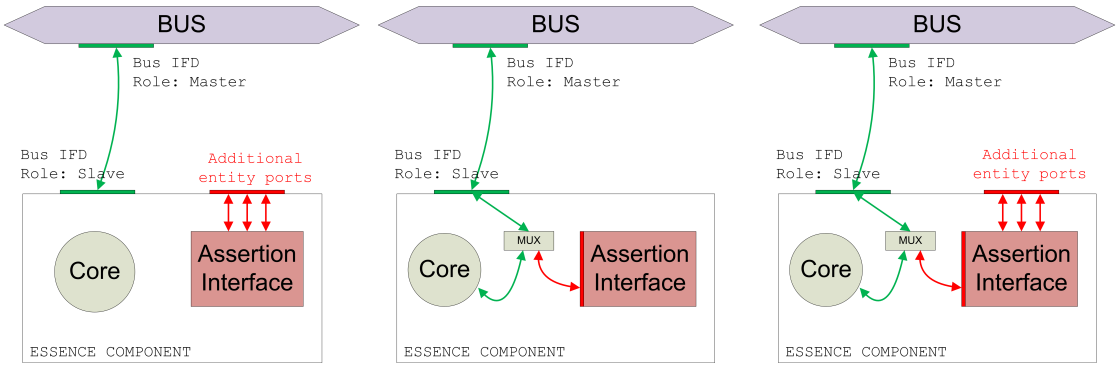

Figure 5.1: Possible ways to access the Assertion Interface

1. The optional assertion interface results in additional ports on the component. If we think in VHDL, then additional ports need to be generated for the entity part. Moreover, the system generator then needs to connect these new ports to *something*. If the component is a bus slave, then the assertion could act as another slave of the same bus. This is not trivial for complex bus systems. Another possibility is to connect the assertion interface directly to another components assertion-specific interface. This connection has to be specified within the system data model XML file - therefore its not influencing the assertion data model.

2. The optional assertion interface of the component is not visible to the outside. Instead of providing additional ports, interfaces are muxed within the component.

Using one bit of the component's address line the muxer can differentiate if the core logic shall be accessed or the assertion interface. But if the assertion interface shall provide support for interrupt propagation throughout the system, then at least one additional port has to be routed to an interrupt controller. Therefore - in case of interrupt - an additional port has to be generated too.

3. The most complex solution would be to provide both options simultaneously. This also would result in two rival slave interfaces for the same assertion interface.

Proposal one would require a lot of changes in the system interconnect generator, which is not part of this thesis and actually was *being developed* at the same time. Consequently solution three is also not feasible. Therefore, concerning a future implementation the option two was preferred and can be defined as additional requirement:

Req.9 Access to the Assertion Interface on a components / systems slave interface

Moreover this approach is independent from other parts of the Essence flow - hence a good starting point for the assertion interface generation.

5.1.4. Multiple Assertion Interfaces per component

If we think of a single assertion checker one can state that the assertion itself is monitoring a set of signals. Taking VHDL as example these signals could then appear in the `SensitivityList` of a VHDL process. If the process is activated due to a signal change the checker functionality evaluates if the property is violated or not. So an assertion is working on one or several signals. In an Essence object each signal belongs to an Interface.

So the data model must capture this possible case and shall allow the definition of as many assertion interfaces as slave interfaces of the Essence object. If an Essence object has several slave interfaces it must be specified which interface is used to access the assertions. Figure 5.2 shows the possible options. The green and blue colors indicate allowed accesses. So it is allowed that an Essence object has more then one assertion interface. But all assertion interfaces must be located on different slave interfaces, otherwise this leads to collision during code generation. Moreover, it shall be permitted that an assertion interface is configured via one slave interface but read using another one. This case seems very unlikely and would result in difficult dependencies during code generation. Moreover the actually independent assertion interfaces would need to exchange data. Therefore, this case is forbidden. The collected additional requirements can be summarized to:

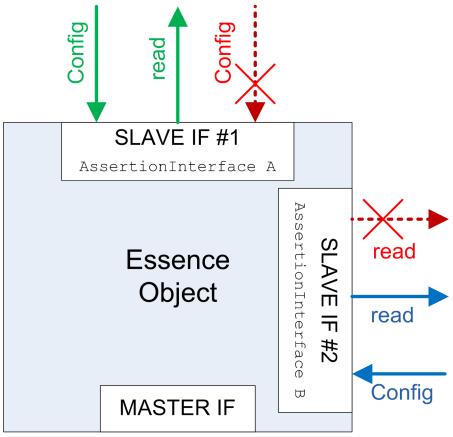

Figure 5.2: Allowed accesses to the `AssertionInterface`

Req.10 support for as many assertion interface as slave interface on an Essence component / system

Req.11 support for monitoring of several interfaces per assertion

5.2. Assertion Interface Meta Model

After the requirements had been identified the implementation of the data model was done. The `AssertionInterface` meta model was build as an extension to the `ModelConfiguration` meta model. As design entry tool Enterprise Architect from Sparx Enterprises[50] was used. It is fully integrated within the Infineon Essence flow and allows Java code generation. Therefore, new models will be compiled and integrated into the Essence data model. The figure 5.3 shows the final implementation. During the modeling it was tried to use existing Essence elements wherever possible but build new elements where necessary. The following sections shall give a detailed description of the data model elements.

The `AssertionInterface` is the only data model with nested `ExtVLNV - XRef` pairs within Essence. Therefore, a "missing feature" of Essence was discovered during the development of the model which resulted in wrong return objects on the usage of the API convenience functions. The required tree traversal of the `ExtVLNV - XRef` pair will be added in future versions.[9]

An example ModelConfiguration XML with an incorporated assertion interface is shown in the appendix A.4.

[9] As per November 2008

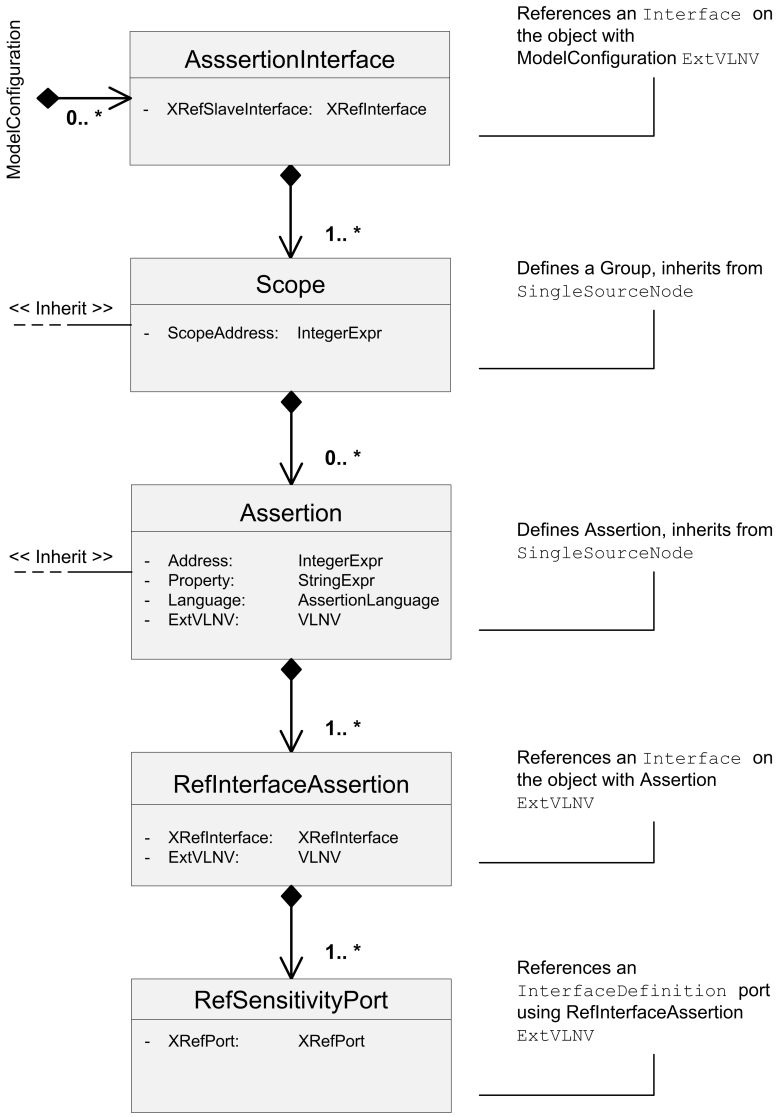

Figure 5.3: `AssertionInterface` data model

5.2.1. AssertionInterface

`AssertionInterface` is the root object for the assertion part of the `ModelConfiguration` data model, which can be found in the appendix A.1. One `ModelConfiguration` data model can have zero to infinity `AssertionInterfaces`. Each `AssertionInterface` has an attribute called `XRefSlaveInterface`. This attribute is a XRef (reference) to an interface of the component or system using the `ModelConfiguration` attribute `ExtVLNV`. This attribute type is defined within the Essence XRef data model. With help of it

each AssertionInterface, which may work on multiple signals from different interfaces, has one certain slave interface. The AssertionInterface can only be accessed via that interface. If there is more than one AssertionInterface in a single ModelConfiguration data model, then all AssertionInterface must have different XRefSlaveInterface, else there is a collision during code generation. It is important to distinguish the XRef-SlaveInterface from the assertions XRefInterface. One single AssertionInterface can only be read and configured from one XRefSlaveInterface. An assertion can work on many different slave and also master interfaces of a component. But the assertion is only monitoring. The Assertion Interface is receiving and submitting data via its XRefSlaveInterface.

5.2.2. Scope

Each AssertionInterface can have one to infinity Scope. The sense of scopes is to group similar assertions. So for example there could be a number of assertion which monitor the system role signals, and another group which monitors the bus transactions. The Scope inherits the attributes from the SingleSourceNode of the ModelConfiguration data model. So for example each Scope is further defined by Name, ID and ShortDescription.

5.2.3. Assertion

Each of the Scope can have zero to infinity Assertion. If a Scope has no Assertion specified, it is a temporary placeholder and will not get generated.

Each Assertion has several attributes:

Property specifies the actual assertion, this field has been included for future work packages which may include assertion checker synthesis

Language specifies the used assertion language; this value is an enumeration type; predefined values are for example SVA, OVL and PSL

ExtVLNV this field is necessary for Ref2Interface, it specifies the VLNV of the Essence object where the referenced interface is specified

Moreover, Assertion also inherits the attributes from the SingleSourceNode of the ModelConfiguration data model.

5.2.4. RefInterfaceAssertion

Each `Assertion` is working on one to infinity Interfaces. Because a RTL code assertion is working on certain ports, the interfaces of these ports must be identified here. Using the Essence XRef model the selected Interface of the Essence object are referenced. This is done via the `ID` of the to be referenced interface. For details please refer to the chapter 3.2.3 on page 27. It is mandatory that each assertion contains at least one reference to a system role interface. This is because in general an assertion statement shall be related to the rising edge of the clock.

5.2.5. RefSensitivityPort

From each Interface the assertion is working on, only some ports are really monitored. In a generated VHDL file those ports would appear in the sensitivity list of a process statement. Inside the data model they are referenced via `RefSensitivityPort`. Again the XRef model is used to point to the selected port.

5.3. Target Code Architecture

The figure 5.4 shows the proposed `AssertionInterface` architecture for any target code implementation. So within the real system the `AssertionInterface` shall consists of four bus-accessible registers. The `AddressRegister` is used to select the scope and assertion. The width of the register shall be 32-bit whereas four bits are used for scope definition. If one scope encoding is used to define the broadcast case, this allows $2^4 - 1 = 15$ possible scopes. Moreover, this allows the definition of 2^{28} unique assertion addresses. It is unlikely that this amount will ever by used at all.

Depending on the `AddressRegister` the assertions internal registers are mapped to the global ones, which are visible on the bus interface. So using the `AddressRegister` a mapping to the virtual registers `ConfigReg` and `StatusReg` is done. Using the registers an interface to advanced debugging features can be provided. If the `ConfigReg` is one-hot encoded this results in 31 possible options that can be enabled or disabled simultaneously. So the assertion could be configured to count the number of failures in an internal register. Another configuration might puts this value into the `StatusReg` where it can be read back from the surrounding system. Furthermore, the interface provides the propagation of a property violation signal. The assertion-failed signals from all assertions are OR-ed. The result will appear in the bus-accessible `FailureReg`.

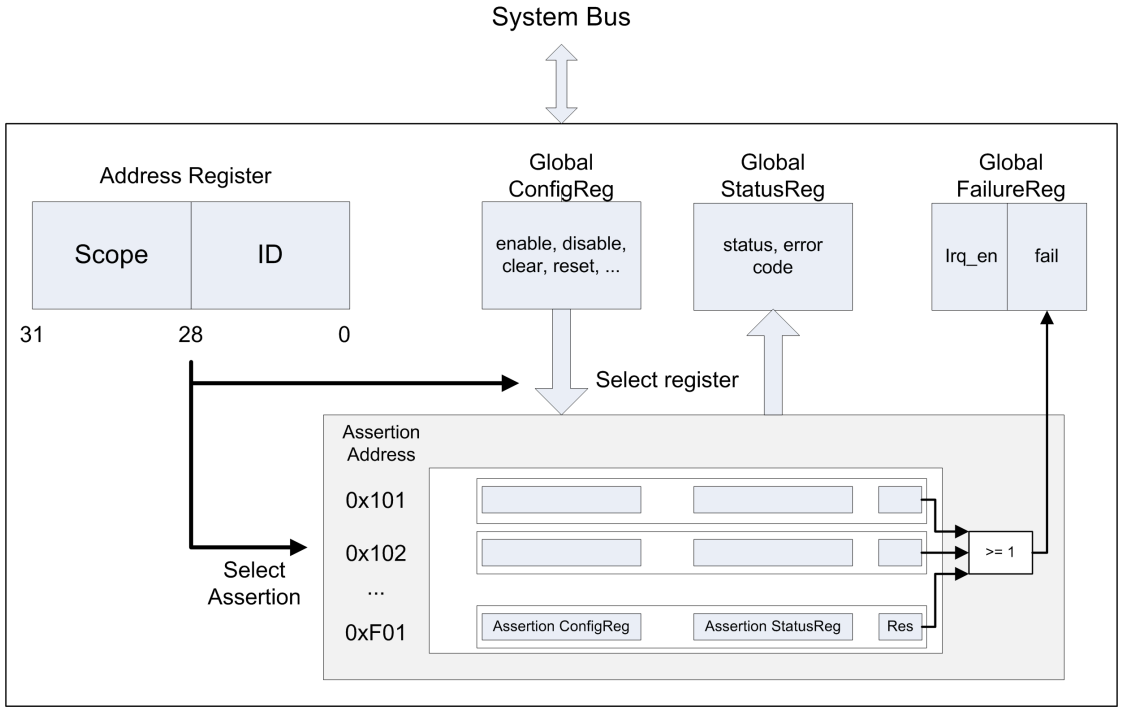

Figure 5.4: The `AssertionInterface` architecture

The register information encoding has not been put into the data model. First of all if we think of the variant to have multiple options enabled at the same time, only an one-hot encoding can fulfill this requirement anyway. Moreover, the intended information representation of register content may varies from case to case, so it tends to be implementation specific. The assertion interface is the outer shell to the real assertion checker functionality. So if the checker part has to be implemented by the developer, it is a meaningful approach to leave a greater variability on how the information of the interface is used to him too.

5.3.1. Assertion Register Constraints

For safety issues the real assertion registers shall have certain constraints. The Essence Component data model offers different concepts to separate the allowed access to registers. So for example a register can be marked read-only or write-only for a certain access type. Using it one can distinguish between the access to the register from within the component and from the component's interface. A re-implementation of a the AccessType and Bitfield concept inside the `AssertionInterface` data model does not seem reasonable.

First of all it violates the general Essence approach - no redundancy within the design. Moreover, the assertion interface architecture will be the same for every generation process. So in every generation for any kind of supported Essence components the amount, size and architecture of the registers will be the same. Therefore, it is not necessary for the registers to be configurable at the moment. This status might be revised in the future development of Essence.[10]

These additional requirements to the registers can be summarized to:

Req.12 The `ConfigRegister` shall only be writeable from the bus side, the assertion part can only read it

Req.13 The `StatusRegister` shall only be readable from the bus side, the assertion part can write to it

Req.14 The `ResultRegister` shall not be writeable from the bus side, the output of all `ResultRegister` is OR-ed and put into the global `FailureReg` register.

5.4. Requirement fulfillment

Some of the previously collected requirements were fulfilled within the assertion interface data model, for others it was advantageous to implement them using the code-generation template. It must be kept in mind that it is advisable not to put any implementation detail into a data model itself. This means in case of this thesis, that the data model shall only contain the information about the assertion, the related signals, but will not give a fixed definition on how to implement the data model in a certain target code. For example, the grouping of assertion should be accomplished within the data model. The in detail architecture, for example the register implementation and how the multiplexing is done is implemented using the template. This has the benefit that the assertion interface can be generated in an optimal way for different target-code languages.

The proposed solution fulfills all previous collected requirements. The table 5.1 comments why a certain requirement is fulfilled. Requirement 1 and 2 are already fulfilled using the meta model approach. With help of it, the assertion interface is expressed in a very abstract and implementation-independent way. The target code will be generated later on using implementation-specific templates. Therefore, this template can generate TLM or RTL code. Requirement 3 is fulfilled using the `ModelConfiguration` data model of Essence. A template can decide to make use of the optional modeled assertion

[10] As per November 2008

	Solution	Implemented
Req.1	fulfilled by usage of the meta model approach itself	meta model
Req.2	fulfilled by usage of the meta model approach itself	meta model
Req.3	fulfilled by usage of the ModelConfiguration data model + templating	data model
Req.4	fulfilled by the assertion interface architecture template (`FailureRegister`)	template
Req.5	fulfilled by the assertion interface architecture template (`ConfigRegister`)	template
Req.6	fulfilled by the assertion interface architecture template (`ConfigRegister`)	template
Req.7	fulfilled by the assertion interface data model	data model
Req.8	fulfilled by the assertion interface architecture template	template
Req.9	fulfilled by the data models `XRefSlaveInterface` reference, component wrapper + multiplexer which depends on one address bit	various
Req.10	fulfilled by the assertion interface data model	data model
Req.11	fulfilled by the assertion interface data model	data model
Req.12	fulfilled by the assertion interface architecture template	template
Req.13	fulfilled by the assertion interface architecture template	template
Req.14	fulfilled by the assertion interface architecture template	template

Table 5.1: Fulfillment of the `AssertionInterface` requirements

interface or not. Requirement 4, 5 and 6 are fulfilled due to the various registers in the assertion interface architecture that provide the possibility to define options for every assertion. Requirement 7 is fulfilled by the means of the assertion interface data model. Using the `Scope` element, assertions can be grouped. The requirement 8 is fulfilled using the code-generation within the template and the assumption that the `Scope` address x"0" will broadcast to all assertions in all scopes. The access to the assertion interface from the outside, requirement 9, is achieved using a wrapper around the desired component as well a multiplexer structure that distinguishes between the assertion interface and the core IP. This solution is independent from other parts of the Essence flow. Requirement 10 and 11 are both fulfilled within the assertion interface data model. Using the `XRefSlaveInterface` attribute in the highest hierarchy of the data model, the slave interface for bus access is specified. Moreover, using the `RefInterfaceAssertion` attribute multiple interfaces and their signals, can be referenced per assertion. Requirements 12, 13 and 14 are fulfilled using the the code-generation template. The generated assertion interface architecture considers the allowed options during write and read access to the registers.

6. Assertion Interface Generation

For the code generation of the optional assertion interface several MAKO templates had to be written. Moreover, it had to be decided what VHDL code architecture should be generated. For the efficient development a generation flow was developed. The figure 6.1 on page 59 illustrates the generation steps. Within the implementation the flow is represented via several shell scripts (`.csh` files) which perform the necessary actions and invoke `essemplate` with code-generation templates. The design flow is organized as follows:

1. Before a template can perform any code generation, the XML files need to get generated first using `essimport` and corresponding plugins. The result is the `ModelConfiguration` XML which also defines one or several `AssertionInterface`. During the `essimport` generation simple checks are performed within the plugin. For example, if the developer wants to add an Assertion to a Scope he has to specify all its attributes. Some of them have to be unique. Before the creation the API checks if the new objects have unique identifiers where it is necessary. These checks are useful for every-day usage and cannot be performed within a semantic rule check of the XML against the schema. The schema only defines the structure and data types of attributes but sets no further constraints.

2. In step two the `ModelConfiguration` is first validated against the schema for the data model. This allows easy discovery of flawed or outdated data models. After this a semantic check using a dummy template is performed. This means that the template will not generate any output but uses the python functionality of MAKO. Therefore, a basic semantic checker was written to check for the correctness of the `ModelConfiguration` XML and allows advanced verification in contrast to simple schema validation. First it is checked if an `AssertionInterface` is present at all. Moreover, the existence of the `RefSlaveInterface` is examined. After this another function iterates over all `Scope` and checks for unique addresses. Thereafter, the same check is performed on the assertions. At last all references are looked up to their sources.

3. Once the verification steps are finished the actual generation is performed. First the VHDL wrapper is build for the Essence Component or Essence System. It instantiates the real component/system and includes the demuxer to access either the IP core or the assertion interface. Moreover, all necessary signals, variables and VHDL type definitions are generated for the given amount of assertion interfaces.

4. In the next step a `.csh` script is generated which performs all commands to generate an assertion. This approach was necessary because `essemplate` can only generate one file per session. Therefore, it is not possible to generate the numerous assertions `.vhd` files in a single run. Using an `essemplate` tool option a directive can be passed into the runtime environment which then can be evaluated within the template. The script generator template makes usage of this concept and builds as many `essemplate` tool calls as assertion. For the directive option the unique assertion `ID` is used.
5. In the last step this just generated script is executed. Now a single .vhd file is generated for every specified assertion within the data model. The VHDL files names follow the naming convention:

   `<Essence_ObjectType>_<Name>`__AIF_ID`<Value>`_ScopeAdr`<Value>`_AssAdr`<Value>`.vhd

Using this approach the actual wrapper and assertion interface are separated from the assertions themselves. This eases file handling and also reduces the assertions designs file size. The real assertion checker part can be added within the assertions `.vhd` files. After this generation process the output files can be used as starting point for the simulation flow (see figure 7.1 on page 68).

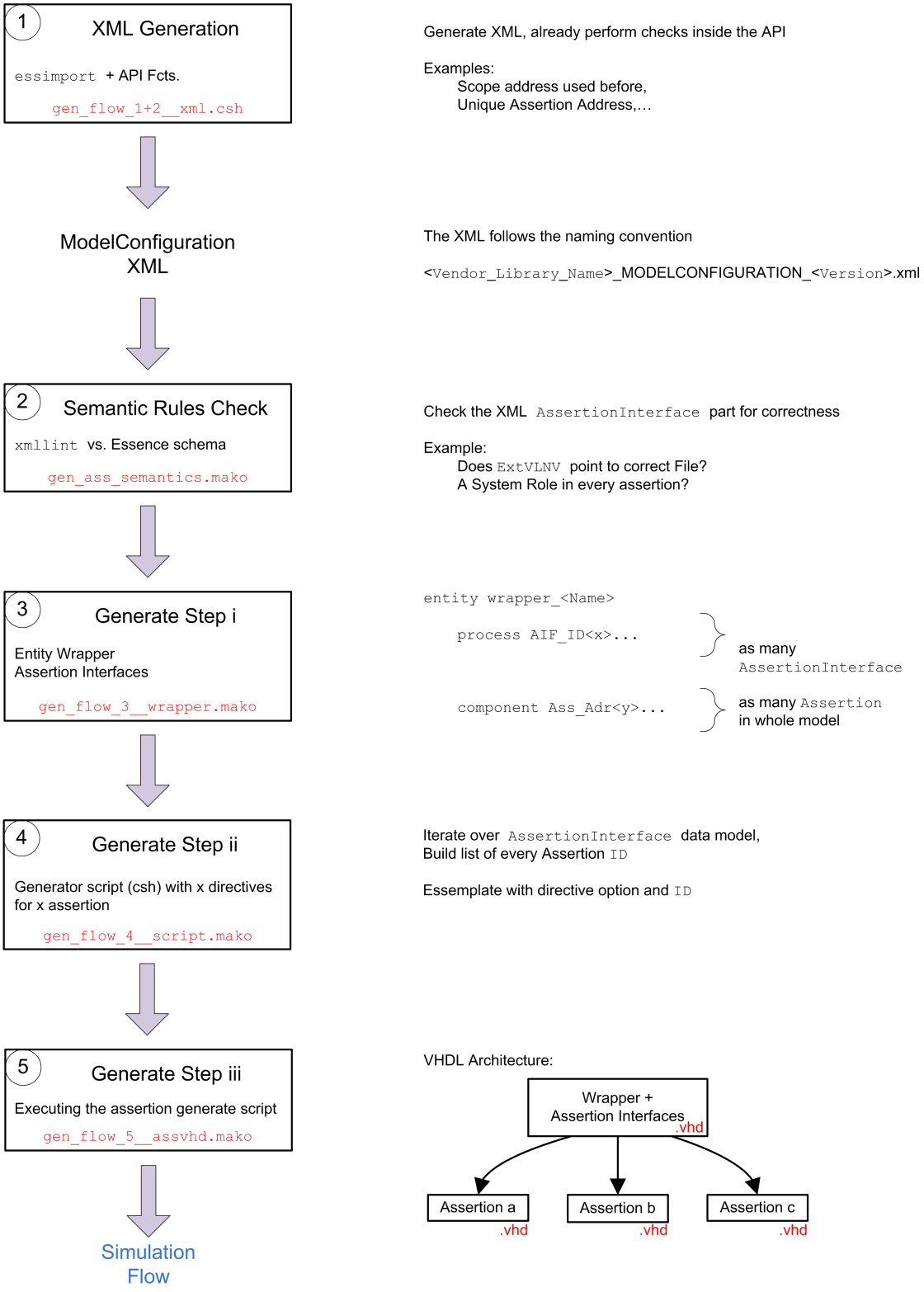

Figure 6.1: `AssertionInterface` generation flow

6.1. Challenges on Template development

During the template development some work-a-arounds and challenges had to be solved. In the following sub-chapters the requirements and resulting architecture of the chosen target implementation language VHDL are presented. During the code generation it was tried to use the highest level of available abstractions of VHDL to reduce the need for generation and keep the `.vhd` files generic. The recently published VHDL-2008 enhancements[4] were omitted due to the lack of tool support, especially in the open-source field. Apart from that all efforts have been taken to reduce the need for generation with the help of VHDL features while keeping the code synthesizable. For example, attributes like `<signal>'lenght` and `<signal>'range` are used to prevent unnecessary code generation. The remaining non-static information will get generated out of the data model. Moreover, it is tried to reduce the amount of repetitive structures using VHDL `for loop` constructs. Where not avoidable the information will also get generated out of the data model. The source [11] gives an overview about available constructs of the VHDL language. For a detailed description please refer to the IEEE 1076 language reference manual [25].

6.1.1. General Coding Style

Inputs for the assertion interface generation flow are Essence component and system data models. Therefore, it was tried to make the templates as universal as possible and allow code generation of different source models within a single template. When `essemplate` is invoked with an Essence object the Python `context` command is used to determine the model type of the object. Within the Python code a variable is assigned with the actual model type. In the left over source code only this variable is used. Depending on the input Essence object the templates perform different actions. If the input is a `ModelConfiguration` data model then the corresponding Essence object is searched via the `ModelConfiguration` root node attributes `ExtVLNV` and `RefXMLType`. If the input is another Essence data model it is checked if a `ModelConfiguration` XML is existing via the naming convention rule. In case, it is furthermore checked if the `ModelConfiguration` points the the same data model that was passed as argument to `essemplate`. This approach allows very variable template invoking. Moreover, the Python `Logger` object is used to provide debug information and to display internal templates decision or assumptions. So if referenced files are missing or entries seem to be flawed an error message appears in the terminal and gets recorded in the log file. In case

of errors this eases the template debugging for the developer. Moreover, if certain corner cases are not supported, for example within the mapping template (see chapter 4.2.6), the template will stop and report via the `Logger` object which constraint is violated. Furthermore, it was tried to harden the templates so that they will not crash in case of errors. Apart from that the template hierarchy outsources often used functions in sub-templates. The relation of the main assertion interface templates and the defined sub-templates is shown in figure 6.2.

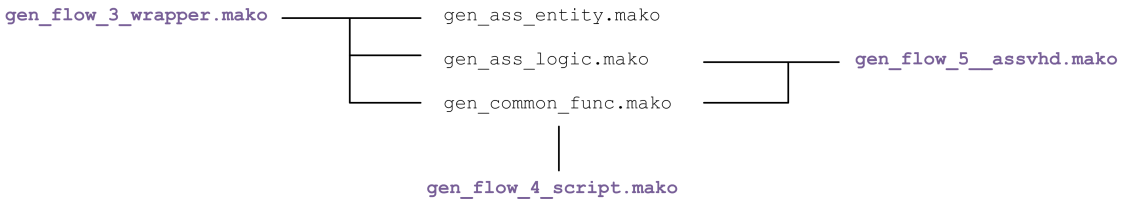

Figure 6.2: Main and sub templates

Another important task was to take care of the data structure handling within Python. The mostly used Python programming language elements are dictionary, tuple and list, whereas n-tuple are a special case of list. Despite these constructs are all used to simple save data in a structured way, they have an important difference concerning code generation. Within a list each element has a fixed position, on every run of the template. The list is build up by adding elements to the list or purging them. A dictionary is a key-to-item relation. Therefore, there is no fixed position of data within the dictionary.[11] In a code generation iterator loop it is important to consider if the generated code has to be in a certain order or not. If it does not matter the use of dictionary causes no problems. In the other case the generation order might be mutated from run to run. This may result in flawed code. Even if the code is mutated but still is correct this can cause problems. Within a verification flow it is common to check the actual build against a golden reference. So it might happen that both files contain the same information, but differ in their internal composition. In case, the verification might states a failure where actually none is present. In this thesis work it was tried to avoid these problems. Moreover, the verification flow (see figure 7.1 on page 68) is based upon the comparison of simulation waveforms. These simulation results will be the same even if the internal file constructs are mutated.

[11] In fact, there are several options how to iterate over a dictionary, some of them do not result in mutated code.

The generated port names get extracted from the data model. In detail the port name retrieval is outsourced into `gen_common_fct.mako`. So the structure, a python dictionary where the port names are stored, stays the same for every bus protocol. Moreover, the keys for `clock`, `reset`, `datain`, `dataout` and `address` signal are identical for every dictionary. This allows the template to be mostly generic concerning the dictionary indexing. Every additional bus specific port then has an unique key.

6.1.2. Wrapper Part

The pure wrapping part is realized within `gen_flow_3__wrapper.mako` and is quite manageable. An example code generation can be found in the appendix A.5. Moreover, this example also contains the assertion interface part. As naming convention the wrapper VHDL entity will have the identifier `WRAPPER_<original_name>`. The port names can have the same identifiers like on the real IP because this leads to no conflict during mapping. Within the architecture part which is also prefixed with `WRAPPER_<architecture>` the real VHDL object is declared using a component declaration. Within the body region of the VHDL architecture an instance of the declared component is instantiated and mapped to the wrapper entity ports. These generation steps can be summarized to:

- declare the real component
- instantiate and port map the real component
- propagates generic parameters

It should be noted that not every signal is just mapped to the entity's one. Under certain conditions of the assertion they have to be assigned to a temporary signal first. The next sub-chapter will explain this in detail.

6.1.3. AssertionInterface Part

An Essence component or system with several slave interfaces can also have several assertion interfaces. Therefore, the code generation needs to provide an unique representation for every assertion interface. This is achieved using the unique attribute `ID` of the assertion interface `XRefSlaveInterface TargetID` because there can be only one `AssertionInterface` on a slave interface. The target code is oriented on the general architecture proposal depicted in 5.4 on page 54. An example code generation can be found in the appendix A.5. The resulting VHDL code for every `AssertionInterface` consists of two processes. The main process functionality is separated into three parts:

1. Interface decoder part
   This part determines if the assertion interface is selected or not. Therefore, a single bit position of the address line is used. This bit is specified using a `ParameterDecl` within the `ModelConfiguration` data model. Moreover, it demuxes the dataout signal using the same information. If the assertion interface is selected the data out signal of the wrapper is assigned to the assertion interface. Now status information about the assertion could be read back. In the other case the wrapper dataout signal is just assigned with real IP ones.

2. Register write part
   If the assertion interface is selected this part determines the desired action using a VHDL `case` statement. The architecture of the assertion interface consists of four bus accessible register (see figure 5.4 on page 54). Hence, two bits need to be used to distinguish the action to be performed. Within the VHDL code the two leftmost bits of the address line are used. These positions can be specified within the XML using a `ParameterDecl`. Within each `when` branch of the `case` statement it is checked if a read or write action to the register shall be performed. In case of read the register data is put into a temporary signal which gets assigned to the dataout signal within the interface decoder part. In case of write the datain signal is assigned to the register. Moreover, the constraints of the registers are considered. Therefore, writing is not allowed for only bus-readable register and vice versa.

3. Virtual assertion register mapping part
   This process selects the desired assertion based upon the address register. The value `x"0"` leads to a broadcast to all assertions. In case the VHDL `for loop` construct is used, which is synthesizable, else the selection is implemented via a nested `case` statement. The physical address is build up of the scope address and the assertion address. If a certain assertion is selected its config register is assigned with the global config register. So using this process a certain assertion's config register gets actually written. In the broadcast case all assertions can be written at once. This allows the initial "arming" of the assertions in a system. Moreover, this part is independent from an actually assertion interface selection. If a specific assertion is addressed the global status register will always be updated with the assertion's one. If the status register is read it will always show the most recent value.

The second process just contains one functionality:

1. Assertion result demultiplexer
   Using a simple VHDL `for loop` construct all single bit result fields of the assertions are OR-ed. The result is written to the global result register. Using a polling mechanism the surrounding system can react on the failure. It would not have been advantageous to specify an information which assertion failed inside the failure register. If we assume 1-bit for the result information, there would be 31-bits left to display an assertion failure, or multiple assertion fails. In the second case, one bit would correspond to one assertion (one-hot encoding). Therefore, this modeling would limit the assertion count inside the `AssertionInterface` to 30, despite the assertion address register would allow $2^{32-Scopes\,Bits} = 2^{28}$ assertions. Another reason why the one-hot coding would not be expedient is that this register encoding would result in hardware depending software. The register content only states assertion number `x` and `y` failed. But what is the assertions address? Moreover, if the position gets generated out of the data model it might happen that the generation sequence is interchanged. This is related to the handling of python data types. For all this reasons this feature was not implemented.

The `dataout` signal of the core entity is the best example for the problems that occur on the generation of assertions that operate on output signals. Actually an assertion is only monitoring a group of signals. If we think of a VHDL interface those signals could have the direction `in`, `out`, or `inout`. The propagation of signals with direction `in` causes no problems. The signal from the wrapper is assigned to the assertion and the real IP at the same time. If an `out` or `inout` signal is within the assertion's signal list the real components output signals must be assigned to an temporary signal first. After that, the signal can be used within the VHDL assertion process sensitivity list and also can be assigned to the wrapper output port. This case had to be taken care of within the template. Moreover, the special corner case of the dataout signal in the sensitivtiy list of an assertion had to be considered to avoid double signal declaration.

6.1.4. Assertion Part

The template for the assertion part generates a VHDL entity with just one process inside the architecture body. As a reminder, the assertion template generates an universal interface and provides basic command aliases. This means the template generates an "*outer shell*" for every assertion. The real assertion itself, the assertion checker part, has to be entered manually or could get generated with help of the checker generators

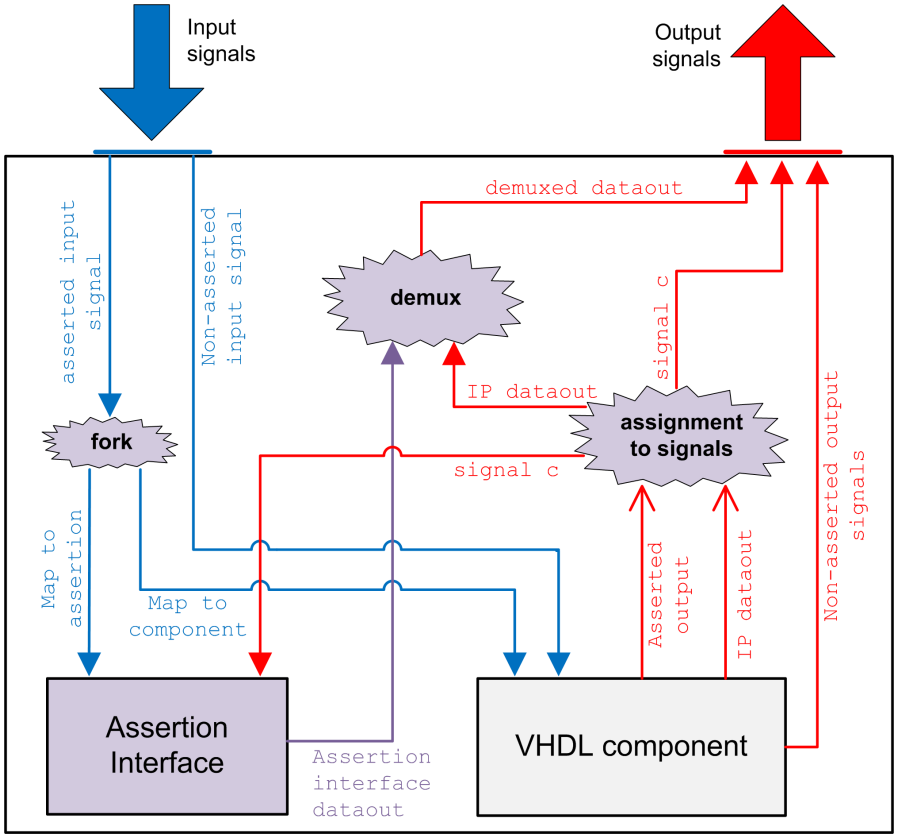

Figure 6.3: Temporary signal mapping on asserted `out` / `inout` signals

presented in chapter 2.1.2. Each assertion entity has three static ports whereas each is connected to the assertion interface. An example code generation can be found in the appendix A.6. The `CFG_Reg` port is used for the virtual port-map of the config register in case of a write request to it. The `Status_Reg` is the 32-bit output port of the assertion. For example the assertion could be configured to report the failure count or the coverage count. The corresponding value would be put into the `Status_Reg`. Using the demultiplexing within the assertion interface this value can be read back using the bus interface of the component. The `Res_Reg` is a single bit output port that propagates the assertion results to the assertion interface.

The assertion process contains all referenced ports from data model part of the specific assertion. Note that the VHDL implementation only allows assertions with at least one interface with role system. With help of it a clocked process with asynchronous reset is generated. Moreover, within the process *static* alias identifier are used to index certain bit positions of the `CFG_Reg`. This concept was implemented in a first stage of

Static Ports	Direction	Port type	Length
CFG_REG	in	bit_vector	(31 downto 0)
STAT_REG	out	bit_vector	(31 downto 0)
RES_REG	out	bit	

Table 6.1: Static Assertion VHDL ports

the assertion interface generator. In that past stage the wrapper, the assertion interface, and all assertions, were located within the same VHDL file. Therefore, the `CFG_Reg` had to be unique and contained several IDs to distinguish them for every assertion. During the later development this concept was taken over as it eases the checker development because the register alias to be queried are the same for every assertion. Moreover, if the bit positions of the config register may be put in the data model in a later revision too, the manual code does not need to be updated. It would be sufficient to generate only the alias definition where these certain bits positions are specified.

6.2. Plugin and Template overview

During the data model development and implementation phases various plugins and templates had to be written. As reminder, plugins build up the Essence XMLs and templates generate target-code out of the Essence XMLs. The table 6.2 summarizes all developed plugins. All plugins prefixed with `spinni_` generate the XML files for the example system. The remaining `gen_ass_4_bridge.py` and `gen_ass_4_system.py` define example assertion interfaces within the `ModelConfiguration` data model.

An example output of a the `gen_ass_4_bridge.py` is shown in the appendix A.4. It illustrates a `ModelConfiguration` XML with only one `AssertionInterface` specified. Because the Essence meta model system, which is intellectual property of Infineon, is necessary to execute all plugins and templates, not all developed files are listed in appendix.

The table 6.3 summarizes all developed templates.

Name	Description	Essence Type
spinni_apb_bus.py	APB Bus, refers to IFD	Bus
spinni_apb_bus_ifd.py	APB Protocol definition, signals and directions	InterfaceDefinition
spinni_apb_bridge.py	defines component with SimpleBus slave, and APB master interface plus system interface	Component
spinni_sub_system.py	defines system with APB/AHB components	System
spinni_ahb_bus.py	AHB Bus, refers to IFD	Bus
spinni_ahb_bus_ifd.py	defines the AHB signals, actually only a subset is defined	InterfaceDefinition
spinni_ahb_bridge.py	AHB Master interface	Component
gen_ass_4_bridge.py	defines example assertion for the APB/AHB bridge	ModelConfig
gen_ass_4_system.py	defines example assertion for the APB/AHB system	ModelConfig
if-calc.py	calculates possible mapping cases, no Essence object creation	-

Table 6.2: Overview about developed plugins

No.	Name	Generates
1	gen_apb_bridge.mako	APB Bridge
2	gen_apb_bus_entity.mako	APB Bus (multiplexing, demultiplexing), Slave assignment, address decoder
3	gen_apb_slave.mako	APB Test Slave, incorporates simple registers to test bus transactions
4	gen_apb_sif.mako	Simple SIF FSM to test the wait generation with the APB Test Slave
5	gen_common_fct.mako	universal library module
6	gen_testbench.mako	generates testbench
7	gen_ahb_bridge.mako	AHB Bridge, very similar to #1
8	gen_ahb_bus_entity.mako	AHB Bus Entity, uses AHB port dict, nearly identical to #2
9	gen_ahb_slave.mako	same as #3 for AHB
10	gen_ahb_sif.mako	same as #4 for AHB
11	gen_constraints.mako	nothing - checks for slave amount and muxer bits relation
12	gen_flow_3__wrapper.mako	Assertion Wrapper
13	gen_flow_4__script.mako	.csh file for Assertion generation
14	gen_flow_5__assvhd.mako	Assertion
15	gen_ass_entity.mako	outsourced functions for assertion interface entity
16	gen_ass_logic.mako	outsourced functions for assertion interface architecture
17	gen_ass_semantic.mako	nothing - performs semantic check on the data model
18	t1_mapping.mako	interface - register parameter mapping

Table 6.3: Overview about developed templates

7. Simulation

During the development and implementation stage enduring testing was performed to validate the assertion interface concept. First the validation focused on the extended SPINNI system components, like the `SimpleBus-to-APB` bridge and the `APB` generation itself. Thereafter, the assertion interface was generated for the example system. Within the first stages of the development the verification was still performed manually till a prototype of the system was working as intended. This first "golden reference" was not meant for a future usage. For example the used VHDL constructs in the code were not optimized yet and some of them even not amenable to synthesis. This status of the project, including the used Essence build, was frozen and used as the golden reference for the current build of the thesis work. Based upon this golden reference a design flow was build to enable early error discovery. The figure 7.1 depicts the used approach. Starting

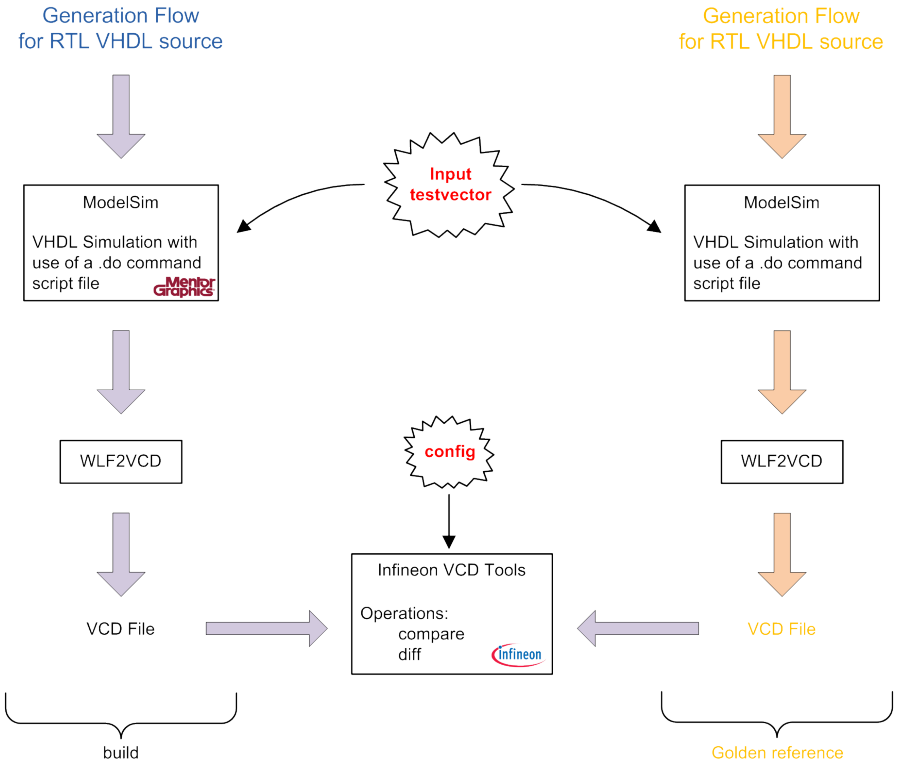

Figure 7.1: Simulation and Verification Flow

point is the RTL VHDL source code from the generation flow. The difference between the build and golden reference generation is, that the plugins, xml files and all necessary tools are frozen in the reference. After this RTL Code simulation was performed using Mentor

Graphics Questa AFV[41]. The used tools allow the specification of the commands in script files. So a script file was created to load interesting signals of the design and to run the simulation for a certain time. Moreover, the waveforms were saved in the proprietary WLF file format. In the next step the simulation result was converted into the *Value Change Dump* (VCD) file format which is defined within the IEEE Verilog Standard (IEEE 1364-2001). With help of a VCD Tools package from Infineon the build and reference VCD files can be compared.

On Essence data model updates, which sometimes lead to plugin and template code revision, the reference model was unaffected. Therefore, it was easy to track down errors that originated from the update. At certain steps of the development, for example when the build version introduced a new "feature" that could not be tested with the existing reference, the present build version was used as the future golden reference. In general, extensive usage of the UNIX file concepts was made (for example symbolic links) to reduce redundancy and ensure that both parts of the flow work on the same file where intended.

The following sub-chapters will explain the general validation approach and the obtained simulation results of the assertion interface enabled SPINNI System in-detail.

7.1. Enhanced SPINNI System

To verify the developed APB/AHB bridge a testbench and an APB/AHB slave component were written. Because both systems are very similar only the APB system will be explained in detail. The testbench uses the VHDL `textio package` and reads an input text file. The text file contains vectors of binary or hexadecimal values which are separated by tab. Moreover, the character `#` has been defined as comment symbol within the testbench. This allows the documentation of the input vectors and eases understanding of the test files. The vector information is read and the individual parts are assigned to the system signals. Figure 7.2 shows an example input file.

The input file contains `SimpleBus` input data, like it would arrive from the CPU on the bridge. These predefined `SimpleBus` queries are put on the `SimpleBus` Interface of the bridge. The APB slave component has several registers inside. The testbench input data was designed to write and read to this registers and moreover make use of all possible state transitions in the FSM.

The basic testbench just reads the vector information and assigns it to the system. During development an advanced testbench was introduced which really behaved like

```
1  # Structure of testbench input file
2  #
3  #
4  # SLV_SEL       : selects one of the available APB Slaves
5  # SLV_REG_SEL   : LOWEST NIBBLE DETERMES THE TARGET SLAVE REGISTER
6  #
7  # BYTEADR: With ByteAdress and Datasize, the data part is selected
8  #                and the to be written part is further specified (if < 32bit)
9  #
10 #                        THESE PARTS FORM THE SB_ADDR(23...0)
11 #                        |-2 bits-------------20 bits-hex----------2-bits-|    --bits--          --hex---
12 #ACC_EN       WRITE     SLV_SEL         SLV_REG_SEL     BYTEADR   DATASIZE        DATA_IN
13      0         0         00              00000           00        00              00000000
14      1         1         01              00001           00        10              ABCDEF77     #WRITE DATA 1
15      0         0         00              00000           00        00              00000000
16      0         0         00              00000           00        00              00000000
17      0         0         00              00000           00        00              00000000
18      1         0         01              00001           00        10              00000000     #READ DATA 1
19      0         0         00              00000           00        00              00000000
20      0         0         00              00000           00        00              00000000
21      0         0         00              00000           00        00              00000000     #3 zero lines necessary
22      1         0         10              00002           00        10              22222222     #READ DATA 2
23      0         0         00              00000           00        00              00000000
24 #    1         0         01              00001           00        10              33333333   #READ DATA 3, ACCESS -> SETUP not possible!!!
25 #    0         0         00              00000           00        00              00000000   #not possible because of S_WAIT is 1
26      0         0         00              00000           00        00              00000000
27      0         0         00              00000           00        00              00000000
28      0         0         00              00000           00        00              00000000
29      1         1         10              00001           00        10              FFFFFFFF     #WRITE DATA
```

Figure 7.2: Example `input file` for simulation

the `SimpleBus`. So the input file does not to contain `S_AccEn_i` signals, because the testbench hold these signals in accordance to the `SimpleBus S_Wait_o` signal. The bridge functionality was tested with both approaches, but in general the old testbench concept was preferred because the bridge and bus components could be hardened against imaginable wrong `SimpleBus` signal transitions. These transitions, for example hazards, could might occur within a real silicon implementation of the system.

Apart from this, the templates were modified to generated VHDL `assert` blocks that print out status messages during the simulation, for example from a state transition of the `S_ACCESS` to the `S_SETUP` state. To ensure an error free design the input data was chosen to give a good code coverage on all branches of the FSM. The gained coverage was evaluated with help of Mentor Questa AFV[41]. Moreover, to be precisely all simulation and verification tasks have been performed using the latter. In doing so, the tool evaluates if all possible transitions and signal changes, for example 1 to 0, occurred during the simulation. So the input testbench code could be optimized to test all transitions of the finite state machines.

7.2. Mixed Language Simulation

After the individual components where validated using the testbench approach it was decided to furthermore validate the system integration. For the SPINNI system all `.vhd` sources were present. Moreover, with help of a SystemC/TLM based MIPS CPU emulator it would be possible to execute real C-Code within the simulation. For this approach a mixed language simulation of SystemC and the generated VHDL sources was projected. This simulation should consists of a SystemC top which instantiates the surrounding system. Therefore, the system module was replaced by the corresponding VHDL module using the SystemC `foreign` construct.[10, P.68] After this a simple C-Code with

- Write a value to a `SimpleBus` address
- Read a value from a `SimpleBus` address

commands was written. In both cases this address is actually bridged to the APB. Algorithm 7.1 shows an easy example. The simulation revealed that this approach would not allow to test all transitions of the APB Bridge. The reason is, that the MIPS CPU needed intermediate clock cycles between the transfers. An assumption is that this behavior is related to the used compiler, which may generated inefficient MIPS commands. Another reason could be the internal architecture of the MIPS CPU. This issue was not further investigated because the interfacing validation was successful.

Algorithm 7.1 Example validation code for SPINNI system

```
1  #define MemRead(A)(*(volatile unsigned int*)(A))
2  #define MemWrite(A,V) *(volatile unsigned int*)(A)=(V)
3
4  int main()
5  {
6      unsigned temp;
7      MemWrite(0x02000004, 0xAAAAAAAA);
8      MemWrite(0x02400004, 0xBBBBBBBB);
9      MemWrite(0x02400008, 0xCCCCCCCC);
10     MemWrite(0x0240000C, 0xDDDDDDDD);
11
12     temp = MemRead (0x0240000C);
13     MemWrite(0x02000004, temp);
14     temp = MemRead (0x02400008);
15     MemWrite(0x02400004, temp);
16
17     return 0;
18 }
```

The mixed language approach was not applied for the assertion interface validation but with the utmost probability there will be no problems. This can be expected because in the generated code the assertion interface is a wrapper around the actual component, which is slave on the APB bus. Therefore, the validation of the interfacing of the CPU with the `SimpleBus`-to-`APB` bus is sufficient.

7.3. Assertion Interface

In order to test the assertion interface the testbench input file was modified. In addition to the `SimpleBus` commands for the bridge validation, additional bus transactions were added. First the individual assertion interfaces where addressed, indicated via a certain bit of the address line to be `HIGH`. As reminder, this information was stored using a `ParamterDecl` within the XML files. The various code parts of the assertion interface contain VHDL `report` commands. Therefore, it is easy to follow the requests on the assertion interface during simulation. The input code performs the following actions:

- access every single assertion
- try to access a non valid assertion, scope
- write data to assertion register, perform broadcast
- read data back from assertion
- mix these actions with commands for the real IP functionality, for example SIF
- config assertion, select to be read information
- read back status register

Using this general testing approach several design flaws could be identified within the first generation stages. For validating the interface itself this testing was adequate. In order to have a more realistic validation of the concept, it was decided to implement a basic checker functionality. This implementation could consist of a false-positive checker, therefore a property that is violated if the signals behave the way it is actually intended. In the example, a special value on `datain` was checked. Moreover, two custom variables were added to the main assertion process. The first variable `var_coverage` counted the successful validation of the property. Because the process will only be activated if one of the signal gets an assignment, the coverage counter also will only increment on each actual necessary property validation. In the same way a property violation variable `var_failures` was defined. With help of the testbench input file this false-positive

property assertion was armed within the first cycles. After this normal `SimpleBus` transactions were perform that were targeted at the APB Slave's real functionality. In some cases, the `datain` signal contained the violation pattern. So both variables increased. Thereafter, the assertion was configured to either put the coverage or failure value into the status register. So the assertion status could be read back from assertion interface, over the bridge component, into the main SPINNI system. At the end, several similar assertion checkers were implemented in different .vhd files (different assertion, that monitor different signals). With this approach it should be tested that all actions are working as intended in a concurrently operating system.

7.4. Application Results

During the development of this thesis work greater experience with the Essence meta modeling concept was gained. Therefore, the necessary effort to build an assertion interface model shall be assessed within this chapter. Assuming an existing system that is expressed in Essence, the time to add an assertion interface model of course varies if the developer is experienced with the underlying concepts or not. Apart from that, the plugin development to build an assertion interface within a `ModelConfiguration` can be considered as the most error prone task. Within the plugin all assertions, their monitored interfaces and the actually monitored signals on these interfaces are specified. When writing the template it is necessary to open all the related XML file using python code or convenience functions. Therefore, the retrieval of an object that shall be added takes much more lines of code than the main functionality itself. This coding style of a plugin is an error prone task. Despite checks within the "add to data model" functions of the assertion interface model were implemented, this task could be significantly improved if a more graphical environment would be available. With help of a graphical tool that can "browse" through an Essence design, the related signals could be collected in a more efficient way. In fact, this would mean that the generation of the plugin is template based itself - from a graphical source. So the generation of an assertion interface can be considered as moderated in the first stages, but eases significantly if only revisions are necessary.

After code-generation the usage of the assertion interface is quite easy. All interface related parts will get generated. It is only necessary to know about the address decoding flow within the target system. In the used testbench approach the corresponding `SimpleBus` commands where entered manually within the input file. A more efficient way would be to also generate the testbench data for a certain system. This approach was not

tackled within this thesis work. The addition of the actual assertion checker part (`.vhd`) has to be done manually. The attributes `Property` and `AssertionLanguage` have been implemented in the assertion data model anyway. In future revisions, a checker generation could be invoked within the template to generate target code. Once an assertion checker is implemented it's result can be read back over the interface. For example, during the development a simple design flaw could be identified with help of the assertions which were configured to simply fire on access. In that case, the assertion selection was not working correctly. Therefore, the debuggable assertion interface approach has great benefits during the systems design.

8. Summary and Outlook

In the described thesis work a data model for the representation of a debuggable assertion interface was created. Based on the Infineon Essence flow an example system was extended to validate the overall approach in more realistic scenario. The introduced data model allows the abstract definition of assertions and their related signals in a flexible, efficient and compact way. The described code generation templates allow the rapid creation of VHDL target code. Moreover, the described design flow incorporates a lot of mechanism to perform error checking in early generation stages. Examples are the inbuilt API checks during the `AssertionInterface` XML generation, the semantic checker which validates if all referenced signals are existing and valid and the waveform based simulation flow. Future extensions of the concept are imaginable. For example, the generation process could be extended to support the generation of an API or firmware header files to allow software the access to the assertion functionality. Moreover, additional code-generation templates could be written. Another test system using the mixed language simulation approach could be set up, to make use of the interface in a real-world product. Consequently, real silicon validation could be tried, first starting with programmable logic devices. The effort to convert the VHDL templates to other RTL level languages like Verilog can be seen as little. Another interesting approach could be to write ESL languages templates. The developed functionality provides the access to IP-internal register and can be used to implement in-system silicon checkers. Therefore, the described concept can be of great advantage within the Infineon Essence model based design flow. Beginning from the first target-code generations these checkers can be included to aid the debugging and validation of the system. Assertion enabled components, and more general Assertion-based Verification, then can lead to faster and less error prone designs. Moreover, within the overall design flow the (assertion based) verification can start beginning from the first code-generation. Consequently, the introduced approach is a step in the right direction to cope with the verification and methodology gap.

References

[1] *The International Technology Roadmap for semiconductors - Design*. http://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_Design.pdf. Version: 2007 1

[2] Abramovici, Miron: In-System Silicon Validation and Debug. Los Alamitos, CA, USA : IEEE Computer Society Press, 2008. – ISSN 0740–7475, S. 216–223 6, 7, 46

[3] Accellera Organization: *Open Verification Library (OVL)*. http://www.accellera.org/activities/ovl/ 5

[4] Ashenden, Peter J. ; Lewis, Jim: *VHDL-2008: Just the New Stuff*. Morgan Kaufmann, 2008 http://www.ashenden.com.au/new-stuff/ 60

[5] Bayer, Michael: *Mako Documentation*. http://www.makotemplates.org/docs/documentation.html. Version: September 2008. – Version 0.2.2 20, 21

[6] Berman, Victor: Standards: The P1685 IP-XACT IP Metadata Standard. In: *IEEE Design and Test of Computers* 23 (2006), Nr. 4, S. 316–317. – ISSN 0740–7475 10

[7] Bjerge, Kim: *System and Architecture Modeling for Real-time systems UML*. Briefing: Modeling of Digital Designs now and in the future. http://www.teknologisk.dk/_root/media/30358_SystemandArchitectureModelingforReal-timesystemsUMLpart1.pdf. Version: May 2008. – Powerpoint Presentation, [Online; accessed 9-November-2008] 16

[8] Boule, M. ; Zilic, Z.: *Generating Hardware Assertion Checkers for Hardwar Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. ISBN: 978-1-4020-8585-7. Springer, 2008. – 280 S. 6

[9] Boule, Marc ; Chenard, Jean-Samuel ; Zilic, Zeljko: Assertion Checkers in Verification, Silicon Debug and In-Field Diagnosis. In: *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2795–7, S. 613–620 xiv, 6, 7, 8, 46

[10] Chinnusamy, Mahendran: *Analysis of Crossing Language Boundaries Using Different Commercial Simulators*, Darmstadt University of Applied Sciences, Diplomarbeit, 2007 71

[11] College of New Jersey, Orlando H.: *VHDL Quick Reference Card*. http://www.tcnj.edu/~hernande/r/VHDL_QRC__01.pdf. Version: 2004 60

[12] DAFCA, Inc.: Miron Abramovici, Paul B.: *A New Approach to In-System Silicon Validation and Debug*. Whitepaper. http://www.dafca.com/downloads/DAFCA_WhitePaper6.pdf. Version: May 2007 8

[13] David Mertz, Frank M.: *Gnosis Utils*. public domain. http://freshmeat.net/projects/gnosisxml/. Version: August 2007 20

[14] Densmore, Douglas: *A Design Flow for the Development, Characterization, and Refinement of System Level Architectural Services*, University of California, Diss., 2007. http://ddensmore.net/dza/wp-content/uploads/2007/06/densmore_phd_thesis.pdf xiv, 2

[15] Ecker, Wolfgang ; Esen, Volkan ; Steininger, Thomas ; Velten, Michael: Requirements and Concepts for Transaction Level Assertion Refinement. In: *IESS*, 2007, S. 1–14 9

[16] ECSI Association: *European Electronic Chips & Systems design Initiative*. http://www.ecsi-association.org 8, 15

[17] ECSI Association: *SPRINT: Open SoC Design Platform for Reuse and Integration of IPs*. http://www.sprint-project.net/. – [Online; accessed 8-November-2008] 8, 14

[18] ECSI Association: *Open SoC Design Platform for Reuse and Integration of IPs*. October 2005. – Annex 1 - Description of Work, version 3.1 14

[19] ECSI Association: *SPRINT Press Release*. http://www.ecsi-association.org/ecsi/PR/SPRINT_Press_Release_Aug4-2006.pdf. Version: August 2006 14

[20] Foster, Harry ; Lacey, David ; Krolnik, Adam: *Assertion-Based Design*. Norwell, MA, USA : Kluwer Academic Publishers, 2003. – ISBN 1402074980 4, 5

[21] Geburzi, Alexander: *Projectgroup RECLIPSE - Tutorial on the UML meta model*. http://wwwcs.uni-paderborn.de/cs/ag-schaefer/PG/Reclipse/seminare/Metamodelle.pdf. Version: November 2004 xv, 12

[22] Gharehbaghi, M. Hessabi S. A.M. Babagoli B. A.M. Babagoli: Assertion-based debug infrastructure for SoC designs. In: *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, IEEE, 2007. – ISBN 1–59593–381–6, S. 137–140 9

[23] Gunar Schirner, Rainer D.: System Level Modeling of an AMBA Bus / Center for Embedded Computer Systems University of California, Irvine. Version: April 2005. http://www.cecs.uci.edu/. 2005 (CECS-05-03). – Technical Report 23

[24] IBM Haifa Research lab: *FoCs*. http://www.haifa.ibm.com/projects/verification/focs/index.html. – [Online; accessed 15-November-2008] 7

[25] IEEE: *VHDL Language Reference Manual*. Institute of Electrical and Electronics Engineers, Inc., 2002. http://ieeexplore.ieee.org/servlet/opac?punumber=7863 60

[26] Infineon Technologies AG: *Essence Datamodel Documentation*. 2008 26, 28, 29

[27] Infineon Technologies AG: *SPINNI System - Simple Bus Specification*. 2008 xiv, 34, 35

[28] Infineon Technologies AG - Bauer, Matthias: *Department: System Design Methodology - Overview*. Powerpoint Presentation, April 2008 10

[29] Infineon Technologies AG - Ecker, Wolfgang ; Esen, Volkan ; Nageldinger, Ulrich ; Steininger, Thomas ; Velten, Michael: UML based Code Generation for the HW/SW Interface. In: *5th International UML-SoC Workshop, Anaheim*, June 8, 2008 32

[30] Infineon Technologies AG - Nageldinger, Ulrich ; Steininger, Thomas: *Overview: XML Methodology - Xchange*. Powerpoint Presentation, January 2008 xiv, 11

[31] Infineon Technologies AG - Nageldinger, Ulrich ; Steininger, Thomas ; Velten, Michael: *ESSEMPLATE User Manual*. v2.1.1. August 2008 xiv, xvi, 20, 30, 31, 32

[32] Infineon Technologies AG - Velten, Michael: *ESSEMPLATE v2.1.0 Powerpoint Presentation*. August 2008 30

[33] J.A. Carballo, A.B. Kahng H. Kashiwagi S. Rawat W. Rosenstiel G. S. T. Hiwatashi H. T. Hiwatashi: *Presentations from the 2008 ITRS Conference on 16 July 2008 - System Design*. http://www.itrs.net/Links/2008Summer/PublicPDF. – [Online; accessed 16-November-2008] 1

[34] Kakoee, Mohammad R. ; Riazati, Mohammad ; Mohammadi, Siamak: Enhancing the Testability of RTL Designs Using Efficiently Synthesized Assertions. Los Alamitos, CA, USA : IEEE Computer Society, 2008, S. 230–235 9

[35] Kruijtzer, Wido ; Wolf, Pieter van d. ; Kock, Erwin de ; Stuyt, Jan ; Ecker, Wolfgang ; Mayer, Albrecht ; Hustin, Serge ; Amerijckx, Christophe ; Paoli, Serge de ; Vaumorin, Emmanuel: Industrial IP integration flows based on IP-XACT^{TM}standards. In: *DATE '08: Proceedings of the conference on Design,*

automation and test in Europe. New York, NY, USA : ACM, 2008. – ISBN 978–3–9810801–3–1, S. 32–37 8, 14

[36] Langtangen, Hans P.: *Python Scripting for Computational Science (Texts in Computational Science and Engineering)*. 3rd ed. Springer, 2008. – ISBN 3540739157 20

[37] Lano, Kevin: *Advanced Systems Design with Java, UML and MDA*. Newton, MA, USA : Butterworth-Heinemann, 2005. – ISBN 0750664967 16

[38] Limited, ARM: *AMBA Specification*. http://www.arm.com/products/solutions/axi_spec.html. Version: May 1999. – Revision 2.0 xiv, 24, 25

[39] Marchal, Benoit: *UML, XMI, and code generation, Part 1-4*. http://www.ibm.com/developerworks/xml/library/x-wxxm23/. Version: March 2004 xiv, xvi, 16, 18

[40] McGill University, Integrated Microsystems L.: *MBAC*. http://www.techtransfer.mcgill.ca/technologies/assertion.php 7

[41] Mentor Graphics Corp.: *Questa Advanced Functional Verification*. http://www.mentor.com/products/fv/abv/questa_afv/ 5, 69, 70

[42] Mueller, Wolfgang ; Vanderperren, Yves: UML and model-driven development for SoC design. In: *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–370–0, S. 1–1 16

[43] Object Management Group: *MARTE - Modeling and Analysis of Real-time and Embedded systems*. http://www.omgmarte.org/. – [Online; accessed 8-November-2008] 16

[44] Object Management Group: *Meta-Object Facility*. http://www.omg.org/mof/. – [Online; accessed 8-November-2008] 12

[45] Object Management Group: *Systems Modeling Language (SysML)*. http://www.sysml.org/. – [Online; accessed 7-November-2008] 16

[46] Object Management Group: *UML Profile for System on a Chip (SoC)*. http://www.omg.org/docs/formal/06-08-01.pdf. Version: August 2006. – v1.0.1, [Online; accessed 13-November-2008] 16

[47] Object Management Group: *MOF 2.0 / XMI Mapping Specification, v2.1.1*. http://www.omg.org/technology/documents/formal/xmi.htm. Version: December 2007. – [Online; accessed 7-November-2008] 12, 17

[48] Object Management Group: *Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*. http://www.omg.org/docs/formal/07-11-04.pdf. Version: December 2007. – [Online; accessed 7-November-2008] 16

[49] Sinderen, M. van ; Pires, L. F.: *Model-Driven Architecture: making metamodels - Part II*. SIKS basic course on Systems modelling, Vught, 31 May 2006. https://doc.telin.nl/dsweb/Get/File-64485. Version: May 2006. – [Online; accessed 9-November-2008] xiv, 13

[50] SparxSystems Software GmbH: *Enterprise Architect v7.1*. http://www.sparxsystems.de/. Version: 2008 26, 50

[51] SPIRIT Consortium: *IEEE P1685 SPIRIT Standardization Working Group*. http://www.spiritconsortium.org/tech/p1685/. – [Online; accessed 12-November-2008] 1, 3, 8, 10, 13

[52] SPIRIT Consortium: *IP-XACT v1.4 specification*. http://www.spiritconsortium.org/tech/docs. – [Online; accessed 12-November-2008] 1, 3, 8, 10

[53] TIMA Laboratory Grenoble: *Website*. http://tima.imag.fr/. – [Online; accessed 7-November-2008] 14

[54] Turumella, Babu ; Sharma, Mukesh: Assertion-based verification of a 32 thread SPARC^{TM}CMT microprocessor. In: *DAC '08: Proceedings of the 45th annual conference on Design automation*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–115–6, S. 256–261 4

[55] Vaumorin, Emmanuel ; Stuyt, Jan ; Kilic, Fatih: SPIRIT IP-XACT Extensions and Exploitation for Verification Software Methodology / Magillem Design Services, NXP. 2008. – Forschungsbericht 14

[56] Weilkiens, Tim: *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, 2008 http://www.system-modeling.com/ 16

[57] Wikipedia: *AMBA specification — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=AMBA_specification&oldid=250908286. Version: 2008. – [Online; accessed 19-November-2008] 23

[58] Wikipedia: *Metadata — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Metadata&oldid=250231077. Version: 2008. – [Online; accessed 7-November-2008] 12

[59] Wikipedia: *Moore's law — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Moore. Version: 2008. – [Online; accessed 2-December-2008] 1

[60] Wikipedia: *Python (programming language) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=250222348. Version: 2008. – [Online; accessed 9-November-2008] 19

[61] Wikipedia: *XML — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=XML&oldid=250206050, 2008. – [Online; accessed 7-November-2008] 17

[62] Wikipedia: *XML Metadata Interchange — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=XML_Metadata_Interchange&oldid=231448414. Version: 2008. – [Online; accessed 7-November-2008] 17

[63] Yves Vanderperren, Wim D. Wolfgang Mueller M. Wolfgang Mueller: *UML for Electronic Systems Design: A Comprehensive Overview*. http://jerry.c-lab.de/~wolfgang/sjes08.pdf. – [Online; accessed 17-November-2008] 16

Index

A. Appendix

A.1. Essence ModelConfiguration data model

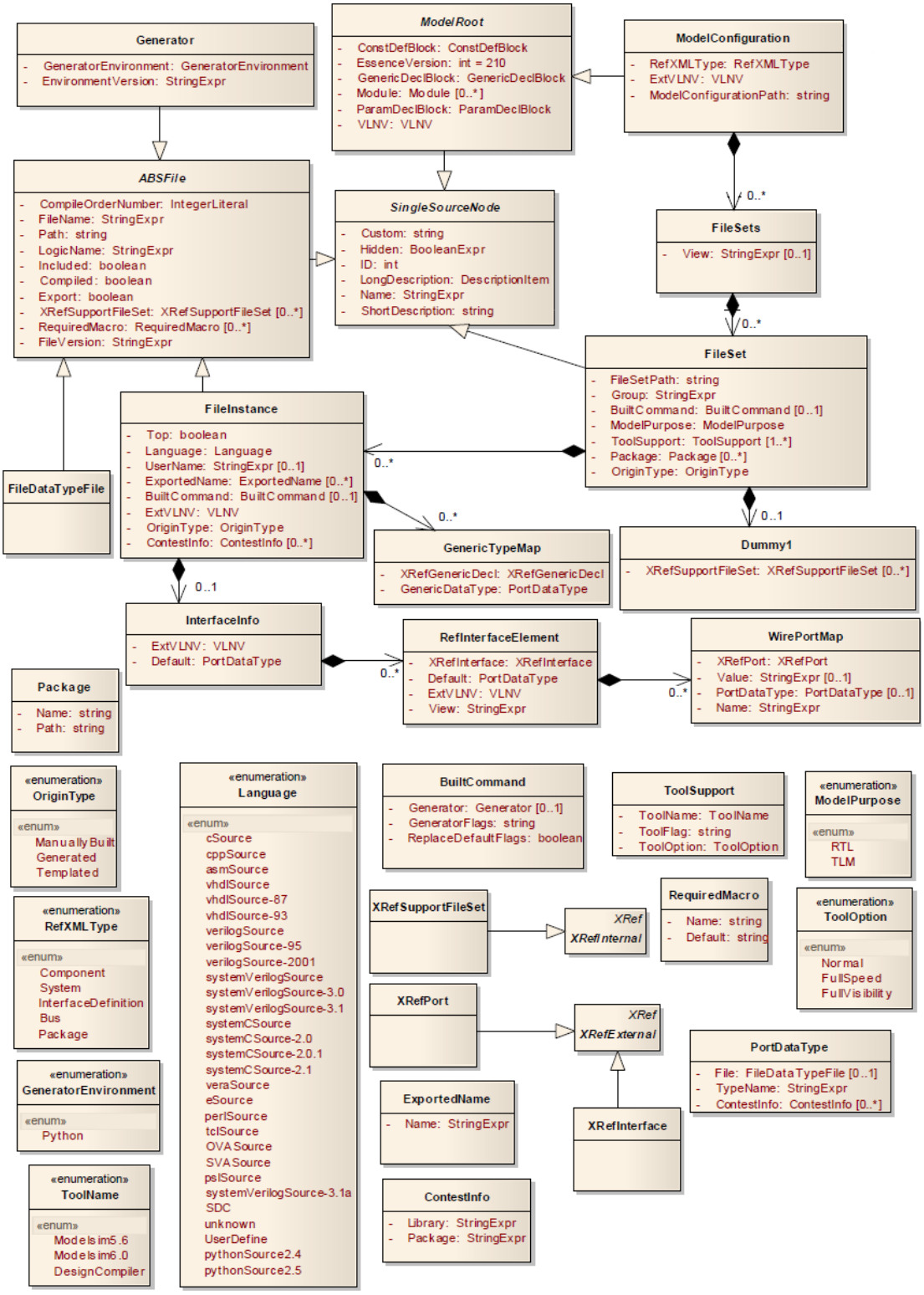

Figure A.1: Infineon Essence ModelConfiguration data model

A.2. AMBA APB Bridge Example

```vhdl

library ieee;
use work.my_func.all; -- WRITE DECODE, READ DECODE

entity apb_bridge is

    port(
    -- the ports here are from interface APB_BRIDGE_MASTER_PORT
    APB_BRIDGE_MASTER_PORT_PWRITE: out bit;
    APB_BRIDGE_MASTER_PORT_PENABLE: out bit;
    APB_BRIDGE_MASTER_PORT_PADDR: out bit_vector( 23 downto 0 );
    APB_BRIDGE_MASTER_PORT_PWDATA: out bit_vector( 31 downto 0 );
    APB_BRIDGE_MASTER_PORT_PRDATA: in bit_vector( 31 downto 0 );
    APB_BRIDGE_MASTER_PORT_PREADY: in bit;
    APB_BRIDGE_MASTER_PORT_PSLVERR: in bit;
    -- the ports here are from interface APB_BRIDGE_SLAVE_PORT
    APB_BRIDGE_SLAVE_PORT_S_DataSize2_i: in bit_vector( 1 downto 0 );
    APB_BRIDGE_SLAVE_PORT_S_Wr_i: in bit;
    APB_BRIDGE_SLAVE_PORT_S_AccEn_i: in bit;
    APB_BRIDGE_SLAVE_PORT_S_Addr24_i: in bit_vector( 23 downto 0 );
    APB_BRIDGE_SLAVE_PORT_S_Data32_i: in bit_vector( 31 downto 0 );
    APB_BRIDGE_SLAVE_PORT_S_Data32_o: out bit_vector( 31 downto 0 );
    APB_BRIDGE_SLAVE_PORT_S_Wait_o: out bit;
    -- the ports here are from interface APB_BRIDGE_SYSTEM_PORT
    APB_BRIDGE_SYSTEM_PORT_Clk: in bit;
    APB_BRIDGE_SYSTEM_PORT_Rst: in bit);

end apb_bridge;

architecture Arch_apb_bridge of apb_bridge is

    type    STATES is (S_IDLE, S_SETUP, S_ACCESS);
    signal  STATE   :   STATES  :=  S_IDLE;
    signal  ASYNC_WAIT  :   bit := '0';
    signal  PERI_WAIT   :   bit := '0';

begin

ASYNC_WAIT <= '0';
APB_BRIDGE_SLAVE_PORT_S_Wait_o <= ASYNC_WAIT or peri_wait;


P_STATES: process (APB_BRIDGE_SYSTEM_PORT_Clk, APB_BRIDGE_SYSTEM_PORT_Rst)

 variable sample_S_DATA32_i     :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_Data32_i'range);
 variable sample_S_ADDR24_i     :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_Addr24_i'range);
 variable sample_S_DATASIZE2_i  :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_DataSize2_i'range);
 variable sample_S_WR_i         :   bit;
 variable sample_PRDATA         :   bit_vector(APB_BRIDGE_MASTER_PORT_PRDATA'range);
 variable tmp_S_DATA32_o        :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_Data32_o'range) := (others => '0');

 variable pipe_sample_S_DATA32_i    :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_Data32_i'range);
 variable pipe_sample_S_ADDR24_i    :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_Addr24_i'range);
 variable pipe_sample_S_DATASIZE2_i :   bit_vector(APB_BRIDGE_SLAVE_PORT_S_DataSize2_i'range);
 variable pipe_sample_S_WR_i        :   bit;

  -- indicates that an pipelined transfer request occured
 variable pipe_request          :   boolean :=  false;
 variable second_run            :   boolean :=  false;
 variable second_run_finished   :   boolean :=  false;


begin

    if APB_BRIDGE_SYSTEM_PORT_Rst = '0' then            -- asynchronous reset

            pipe_request := false;
            second_run := false;
            second_run_finished := false;

            peri_wait   <=  '0';

            STATE   <=  S_IDLE;
```

```
74
75              --PADDR    <=    (others => '0');
76              APB_BRIDGE_MASTER_PORT_PWDATA    <=  (others => '0');
77              APB_BRIDGE_MASTER_PORT_PENABLE   <=  '0';
78              APB_BRIDGE_MASTER_PORT_PWRITE    <=  '0';
79
80
81              --PSELX    <=    (others => '0');
82              tmp_S_DATA32_o       :=  (others => '0');
83              sample_S_ADDR24_i    :=  (others => '0');
84
85              APB_BRIDGE_SLAVE_PORT_S_Data32_o  <=  tmp_S_DATA32_o;
86
87      elsif APB_BRIDGE_SYSTEM_PORT_Clk = '1' and APB_BRIDGE_SYSTEM_PORT_Clk'event then
88
89          case STATE is
90
91              when S_IDLE   =>
92
93                      APB_BRIDGE_MASTER_PORT_PENABLE   <=  '0';
94                      pipe_request        := false;
95                      second_run          := false;
96                      second_run_finished := false;
97
98                      if APB_BRIDGE_SLAVE_PORT_S_AccEn_i = '1' then
99
100                         -- IDLE ===> SETUP
101
102                         STATE   <=  S_SETUP;
103                         --- SAMPLE ALL INPUTS
104                         sample_S_ADDR24_i       :=  APB_BRIDGE_SLAVE_PORT_S_Addr24_i;
105                         sample_S_DATA32_i       :=  APB_BRIDGE_SLAVE_PORT_S_Data32_i;
106                         sample_S_DATASIZE2_i    :=  APB_BRIDGE_SLAVE_PORT_S_DataSize2_i;
107                         sample_S_WR_i           :=  APB_BRIDGE_SLAVE_PORT_S_Wr_i;
108
109                         APB_BRIDGE_MASTER_PORT_PWRITE   <=  sample_S_WR_i;
110
111                         if sample_S_WR_i = '1' then
112                             APB_BRIDGE_MASTER_PORT_PWDATA <= WRITE_DECODE(sample_S_ADDR24_i(1 downto 0), ↵
    sample_S_DATASIZE2_i, sample_S_DATA32_i);
113                             peri_wait   <=  '0';
114
115                         else
116                             -- READ CASE, do always THROW WAIT
117                             peri_wait   <=  '1';
118                         end if;
119
120
121                         -- ALL PSEL ASSIGNMENTS ARE TAKEN IN THE BUS
122
123                     else
124                         -- IDLE ===> IDLE
125                         sample_S_ADDR24_i := (others => '0');
126                         peri_wait   <=  '0';
127
128                         STATE   <=  S_IDLE;
129
130                     end if;
131
132
133
134
135             when S_SETUP    =>  -- SETUP ===> ACCESS
136                                 STATE   <=  S_ACCESS;
137
138                                 -- PSEL ASSIGNED IN BUS
139                                 APB_BRIDGE_MASTER_PORT_PENABLE  <=  '1';
140                                 APB_BRIDGE_MASTER_PORT_PWRITE   <=  sample_S_WR_i;
141
142                                 if sample_S_WR_i = '0' then
143                                     peri_wait   <=  '1';
144
145                                 elsif second_run then
```

```
146                                     pipe_request := false;
147                                     second_run :=  false;
148                                     second_run_finished := true;
149                                     peri_wait   <=  '1';
150 
151                                     PIPELINED_WRITE_3:
152                                     REPORT "(WRITE CASE) Access->Setup with previous saved INPUT DATA!"
153                                     severity NOTE;
154 
155                                 elsif APB_BRIDGE_SLAVE_PORT_S_AccEn_i = '1' then
156                                     peri_wait   <=  '1';
157                                     pipe_request := true;
158 
159                                     -- save new data
160 
161                                     pipe_sample_S_ADDR24_i      :=  APB_BRIDGE_SLAVE_PORT_S_Addr24_i;
162                                     pipe_sample_S_DATA32_i      :=  APB_BRIDGE_SLAVE_PORT_S_Data32_i;
163                                     pipe_sample_S_DATASIZE2_i   :=  APB_BRIDGE_SLAVE_PORT_S_DataSize2_i;
164                                     pipe_sample_S_WR_i          :=  APB_BRIDGE_SLAVE_PORT_S_Wr_i;
165 
166 
167                                     PIPELINED_WRITE_1:
168                                     REPORT "(WRITE CASE) New Data in Setup Phase!"
169                                     severity WARNING;
170 
171                                 else
172                                     peri_wait   <=  '0';
173                                 end if;
174 
175 
176 
177 
178             when S_ACCESS   =>
179 
180                     second_run := false;
181 
182                     if APB_BRIDGE_MASTER_PORT_PREADY = '1' then
183 
184                         APB_BRIDGE_MASTER_PORT_PENABLE   <=  '0';
185                         -- Outputs not driven until the S_ACCESS State!
186 
187                         if sample_S_WR_i = '0' then
188                             tmp_S_DATA32_o  :=  READ_DECODE(sample_S_ADDR24_i(1 downto 0), ↵
    sample_S_DATASIZE2_i, APB_BRIDGE_MASTER_PORT_PRDATA);
189 
190                             peri_wait   <=  '0';
191                             sample_S_ADDR24_i := (others => '0');
192                             --!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
193                             -- ACCESS->STATE is not possible in the READ CASE
194                             -- REASON: SIMPLE BUS WAIT GENERATION!
195                             STATE   <=  S_IDLE;
196                             --!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
197 
198 
199                         elsif (APB_BRIDGE_SLAVE_PORT_S_AccEn_i = '1' and not second_run_finished) or ↵
    pipe_request then
200 
201                             -- ACCESS ===> SETUP
202                             STATE   <=  S_SETUP;
203 
204                             SAVE_CYCLE:
205                             REPORT "(ACCESS=>SETUP) SAVING ONE CLK CYCLE."
206                             severity NOTE;
207 
208 
209                             if pipe_request then
210                                 second_run := true;
211                                 peri_wait   <=  '1';
212 
213                                 sample_S_ADDR24_i       :=  pipe_sample_S_ADDR24_i;
214                                 sample_S_DATA32_i       :=  pipe_sample_S_DATA32_i;
215                                 sample_S_DATASIZE2_i    :=  pipe_sample_S_DATASIZE2_i;
216                                 sample_S_WR_i           :=  pipe_sample_S_WR_i;
```

```
217
218                             else
219                             --- SAMPLE ALL INPUTS
220                                 sample_S_ADDR24_i       :=  APB_BRIDGE_SLAVE_PORT_S_Addr24_i;
221                                 sample_S_DATA32_i       :=  APB_BRIDGE_SLAVE_PORT_S_Data32_i;
222                                 sample_S_DATASIZE2_i    :=  APB_BRIDGE_SLAVE_PORT_S_DataSize2_i;
223                                 sample_S_WR_i           :=  APB_BRIDGE_SLAVE_PORT_S_Wr_i;
224                             end if;
225
226                             APB_BRIDGE_MASTER_PORT_PWRITE   <=  sample_S_WR_i;
227                             -- PADDR and PSEL assignments are taken in the BUS
228
229                             if sample_S_WR_i = '1' then
230                                 APB_BRIDGE_MASTER_PORT_PWDATA <= WRITE_DECODE(sample_S_ADDR24_i(1 downto 0), ↲
    sample_S_DATASIZE2_i, sample_S_DATA32_i);
231                             end if;
232
233
234                         else -- NO SB_TRANSFER_REQ
235                             -- ACCESS ===> IDLE
236                             STATE   <=  S_IDLE;
237
238                             peri_wait   <=  '0';
239                             APB_BRIDGE_MASTER_PORT_PWDATA   <=  (others => '0');
240                             sample_S_ADDR24_i := (others => '0');
241
242                             -- dont touch PADDR, PWRITE to save energy (V2-5.2.2)
243
244                         end if;
245
246                     else -- PREADY not 1
247                         -- ACCESS ===> ACCESS
248                         STATE   <=  S_ACCESS;
249
250                         -- dont touch:
251                         -- PSEL, PENABLE, PWDATA, PADDR, PWRITE
252
253
254                         if sample_S_WR_i = '0' then
255
256                             tmp_S_DATA32_o  :=  READ_DECODE(sample_S_ADDR24_i(1 downto 0), ↲
    sample_S_DATASIZE2_i, APB_BRIDGE_MASTER_PORT_PRDATA);
257                             peri_wait   <=  '1';
258
259                         elsif APB_BRIDGE_SLAVE_PORT_S_AccEn_i = '1' then
260                             -- Not Ready, WRITE Case, New Access (can be only Write)
261
262                                 if not pipe_request then
263
264
265                                     PIPELINE_WRITE_2:
266                                     REPORT "(WRITE CASE) New Data in ACCESS Phase, while PREADY not 1!"
267                                     severity WARNING;
268
269                                     pipe_sample_S_ADDR24_i      :=  APB_BRIDGE_SLAVE_PORT_S_Addr24_i;
270                                     pipe_sample_S_DATA32_i      :=  APB_BRIDGE_SLAVE_PORT_S_Data32_i;
271                                     pipe_sample_S_DATASIZE2_i   :=  APB_BRIDGE_SLAVE_PORT_S_DataSize2_i;
272                                     pipe_sample_S_WR_i          :=  APB_BRIDGE_SLAVE_PORT_S_Wr_i;
273
274                                     -- SAVE DATA
275
276                                     pipe_request := true;
277                                 end if;
278
279
280                                 peri_wait   <=  '1';
281
282                         else -- Not Ready, Write Case, No new Access
283                             peri_wait   <=  '0';
284
285                         end if;
286
287                     end if;
```

```
288
289             when others  =>     pipe_request := false;
290
291                                 STATE   <=  S_IDLE;
292
293                                 APB_BRIDGE_MASTER_PORT_PWDATA   <=  (others => '0');
294                                 APB_BRIDGE_MASTER_PORT_PENABLE  <=  '0';
295                                 APB_BRIDGE_MASTER_PORT_PWRITE   <=  '0';
296                                 peri_wait   <=  '0';
297
298                                 tmp_S_DATA32_o  :=  (others => '0');
299                                 APB_BRIDGE_SLAVE_PORT_S_Data32_o  <=  tmp_S_DATA32_o;
300
301         end case;
302
303
304         -- DO ALWAYS ASSIGN IF   CLK = 1 AND APB_BRIDGE_SYSTEM_PORT_Clk'EVENT
305         APB_BRIDGE_SLAVE_PORT_S_Data32_o    <=  tmp_S_DATA32_o;
306         APB_BRIDGE_MASTER_PORT_PADDR(sample_S_ADDR24_i'range)   <=  sample_S_ADDR24_i;
307
308         end if;
309
310
311
312     end process P_STATES;
313
314     assert not (APB_BRIDGE_MASTER_PORT_PREADY = '0' AND APB_BRIDGE_SLAVE_PORT_S_AccEn_i = '1' AND STATE = ↵
    S_IDLE)
315     report "STATE: IDLE,  PREADY is still 0 ?!,  S_ACCEN_i gets 1 --> New Action Request"
316     severity error;
317
318
319 end Arch_apb_bridge;
```

A.3. AMBA APB Slave Example

```
architecture Arch_sif_apb of sif_apb is

 constant   REG_CNT : integer range 0 to 31 := 3;
 subtype    REG_WIDTH   is  bit_vector(31 downto 0);
 type       REGISTERS   is  array (integer range 0 to REG_CNT) of REG_WIDTH;

 -- signal TL_REG : REGISTERS;
 -- signal TL_PREADY : bit;

 -- component SIF_TEST
  -- port ( CLK    :      in     bit;
          -- RST           :      in     bit;
          -- CONTROL       :      in     bit_vector(31 downto 0);
          -- DATAOUT       :      out    bit;
          -- READY    :      out    bit;
          -- DATAIN :      in     bit_vector(31 downto 0)
          -- );
 -- end component;

begin

     -- SIF_0:   SIF_TEST port map
              -- (   CLK             =>    CLK,
                  -- RST             =>    RST,
                  -- CONTROL         =>    TL_REG(0),
                  -- DATAOUT         =>    open,
                  -- READY       =>    TL_PREADY,
                  -- DATAIN      =>    TL_REG(1)
              -- );

DUMMY: process (SIF_APB_SYSTEM_PORT_Clk, SIF_APB_SYSTEM_PORT_Rst)
    variable REG : REGISTERS;
    variable REG_SEL: integer range 0 to REG_CNT;

    begin

        if SIF_APB_SYSTEM_PORT_Rst = '0' then

            for I in 0 to REG_CNT loop
                REG(I)      :=  x"0000_0000";
                -- TL_REG(I)    <=   REG(I);
            end loop;
            SIF_APB_SLAVE_PORT_PRDATA   <=  (others => '0');

        elsif SIF_APB_SYSTEM_PORT_Clk = '1' and SIF_APB_SYSTEM_PORT_Clk'event then

            if  SIF_APB_SLAVE_PORT_PSEL =   '1'  then

                REG_SEL := VEC2INT(SIF_APB_SLAVE_PORT_PADDR( 3 downto 0) );

                if  SIF_APB_SLAVE_PORT_PWRITE   =   '0' then
                    SIF_APB_SLAVE_PORT_PRDATA   <=  REG(REG_SEL);
                else
                    REG(REG_SEL)    :=  SIF_APB_SLAVE_PORT_PWDATA;
                end if;

                -- for I in 0 to REG_CNT loop
                    -- TL_REG(I)    <=   REG(I);
                -- end loop;

            else
                SIF_APB_SLAVE_PORT_PRDATA   <=  (others => '0');
            end if;
        end if;

    end process DUMMY;

    --PREADY  <= TL_READY;
    SIF_APB_SLAVE_PORT_PREADY <= '1';
    SIF_APB_SLAVE_PORT_PSLVERR <= '0';  -- not used in this SLAVE

end Arch_sif_apb;
```

A.4. AssertionInterface XML Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ModelConfiguration>
    <Custom></Custom>
    <Hidden>false</Hidden>
    <ID>1</ID>
    <LongDescription>
        <html></html>
    </LongDescription>
    <Name>AHB_BRIDGE_MODELCONFIGURATION</Name>
    <ShortDescription>Language Specific and optional Assertion Interface Declaration</ShortDescription>
    <ConstDefBlock>
        <Custom></Custom>
        <Hidden>false</Hidden>
        <ID>2</ID>
        <LongDescription>
            <html></html>
        </LongDescription>
        <Name>DefaultName</Name>
        <ShortDescription></ShortDescription>
    </ConstDefBlock>
    <EssenceVersion>210</EssenceVersion>
    <GenericDeclBlock>
        <Custom></Custom>
        <Hidden>false</Hidden>
        <ID>3</ID>
        <LongDescription>
            <html></html>
        </LongDescription>
        <Name>DefaultName</Name>
        <ShortDescription></ShortDescription>
    </GenericDeclBlock>
    <ParamDeclBlock>
        <Custom></Custom>
        <Hidden>false</Hidden>
        <ID>4</ID>
        <LongDescription>
            <html></html>
        </LongDescription>
        <Name>DefaultName</Name>
        <ShortDescription></ShortDescription>
        <ParamDecl xsi:type="IntegerDecl" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <Custom></Custom>
            <Hidden>false</Hidden>
            <ID>5</ID>
            <LongDescription>
                <html></html>
            </LongDescription>
            <Name>MUX_BIT</Name>
            <ShortDescription>One bit is needed to distinguish between Assertion IF and IP Core
</ShortDescription>
            <DefaultValue>23</DefaultValue>
        </ParamDecl>
    </ParamDeclBlock>
    <VLNV>
        <Vendor>IFX</Vendor>
        <Library>SPINNI</Library>
        <Name>AHB_BRIDGE_MODELCONFIGURATION</Name>
        <Version>100</Version>
    </VLNV>
    <RefXMLType>Component</RefXMLType>
    <ExtVLNV>
        <Vendor>IFX</Vendor>
        <Library>SPINNI</Library>
        <Name>AHB_BRIDGE</Name>
        <Version>100</Version>
    </ExtVLNV>
    <ModelConfigurationPath>not used</ModelConfigurationPath>
    <AssertionInterface>
        <XRefSlaveInterface>
            <XRefTargetID>12</XRefTargetID>
        </XRefSlaveInterface>
        <Scope>
            <Custom></Custom>
```

```
            <Hidden>false</Hidden>
            <ID>6</ID>
            <LongDescription>
                <html></html>
            </LongDescription>
            <Name>Periphal_Assertions_Group</Name>
            <ShortDescription></ShortDescription>
            <ScopeAddress>5</ScopeAddress>
            <Assertion>
                <Custom></Custom>
                <Hidden>false</Hidden>
                <ID>8</ID>
                <LongDescription>
                    <html></html>
                </LongDescription>
                <Name>RST_Assertion</Name>
                <ShortDescription>This Assertions checks the Reset Signal</ShortDescription>
                <Address>77</Address>
                <Property></Property>
                <Language>UAL</Language>
                <ExtVLNV>
                    <Vendor>IFX</Vendor>
                    <Library>SPINNI</Library>
                    <Name>AHB_BRIDGE</Name>
                    <Version>100</Version>
                </ExtVLNV>
                <RefInterfaceAssertion>
                    <XRefInterface>
                        <XRefTargetID>12</XRefTargetID>
                    </XRefInterface>
                    <ExtVLNV>
                        <Vendor>IFX</Vendor>
                        <Library>SPINNI</Library>
                        <Name>SIMPLE_BUS_IFD</Name>
                        <Version>100</Version>
                    </ExtVLNV>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>21</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>22</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>23</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>24</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>25</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>26</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>27</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                </RefInterfaceAssertion>
                <RefInterfaceAssertion>
```

```
                    <XRefInterface>
                        <XRefTargetID>13</XRefTargetID>
                    </XRefInterface>
                    <ExtVLNV>
                        <Vendor>IFX</Vendor>
                        <Library>SPINNI</Library>
                        <Name>SYSTEM_IFD</Name>
                        <Version>100</Version>
                    </ExtVLNV>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>9</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                    <RefSensitivityPort>
                        <XRefPort>
                            <XRefTargetID>10</XRefTargetID>
                        </XRefPort>
                    </RefSensitivityPort>
                </RefInterfaceAssertion>
            </Assertion>
        </Scope>
    </AssertionInterface>
</ModelConfiguration>

```

A.5. Assertion Wrapper Example

```
1
2   library ieee;
3   use work.my_func.all;
4
5   entity WRAPPER_ahb_bridge is
6
7   generic(
8       TEST_GEN_1: integer := 22;
9       TEST_GEN_2: integer := 2);
10
11      port(
12      -- the ports here are from interface AHB_BRIDGE_MASTER_PORT
13      AHB_BRIDGE_MASTER_PORT_HTRANS: out bit_vector( 1 downto 0 );
14      AHB_BRIDGE_MASTER_PORT_HADDR: out bit_vector( 23 downto 0 );
15      AHB_BRIDGE_MASTER_PORT_HWRITE: out bit;
16      AHB_BRIDGE_MASTER_PORT_HSIZE: out bit_vector( 2 downto 0 );
17      AHB_BRIDGE_MASTER_PORT_HBURST: out bit_vector( 2 downto 0 );
18      AHB_BRIDGE_MASTER_PORT_HWDATA: out bit_vector( 31 downto 0 );
19      AHB_BRIDGE_MASTER_PORT_HRDATA: in bit_vector( 31 downto 0 );
20      AHB_BRIDGE_MASTER_PORT_HREADY: in bit;
21      AHB_BRIDGE_MASTER_PORT_HRESP: in bit_vector( 1 downto 0 );
22      -- the ports here are from interface AHB_BRIDGE_SB_SLAVE_PORT
23      AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i: in bit_vector( 1 downto 0 );
24      AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i: in bit;
25      AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i: in bit;
26      AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i: in bit_vector( 23 downto 0 );
27      AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i: in bit_vector( 31 downto 0 );
28      AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o: out bit_vector( 31 downto 0 );
29      AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o: out bit;
30      -- the ports here are from interface AHB_BRIDGE_AHB_SLAVE_PORT
31      AHB_BRIDGE_AHB_SLAVE_PORT_HSEL: in bit;
32      AHB_BRIDGE_AHB_SLAVE_PORT_HADDR: in bit_vector( 31 downto 0 );
33      AHB_BRIDGE_AHB_SLAVE_PORT_HWRITE: in bit;
34      AHB_BRIDGE_AHB_SLAVE_PORT_HTRANS: in bit_vector( 1 downto 0 );
35      AHB_BRIDGE_AHB_SLAVE_PORT_HSIZE: in bit_vector( 2 downto 0 );
36      AHB_BRIDGE_AHB_SLAVE_PORT_HBURST: in bit_vector( 2 downto 0 );
37      AHB_BRIDGE_AHB_SLAVE_PORT_HWDATA: in bit_vector( 31 downto 0 );
38      AHB_BRIDGE_AHB_SLAVE_PORT_HRDATA: out bit_vector( 31 downto 0 );
39      AHB_BRIDGE_AHB_SLAVE_PORT_HREADY: out bit;
40      AHB_BRIDGE_AHB_SLAVE_PORT_HRESP: out bit_vector( 1 downto 0 );
41      -- the ports here are from interface AHB_BRIDGE_SYSTEM_PORT
42      AHB_BRIDGE_SYSTEM_PORT_Clk: in bit;
43      AHB_BRIDGE_SYSTEM_PORT_Rst: in bit);
44
45  end WRAPPER_ahb_bridge;
46
47
48  architecture Wrapper_Arch_ahb_bridge of WRAPPER_ahb_bridge is
49
50  component ahb_bridge is
51
52  generic(
53      TEST_GEN_1: integer := 22;
54      TEST_GEN_2: integer := 2);
55
56
57      port(
58      -- the ports here are from interface AHB_BRIDGE_MASTER_PORT
59      AHB_BRIDGE_MASTER_PORT_HTRANS: out bit_vector( 1 downto 0 );
60      AHB_BRIDGE_MASTER_PORT_HADDR: out bit_vector( 23 downto 0 );
61      AHB_BRIDGE_MASTER_PORT_HWRITE: out bit;
62      AHB_BRIDGE_MASTER_PORT_HSIZE: out bit_vector( 2 downto 0 );
63      AHB_BRIDGE_MASTER_PORT_HBURST: out bit_vector( 2 downto 0 );
64      AHB_BRIDGE_MASTER_PORT_HWDATA: out bit_vector( 31 downto 0 );
65      AHB_BRIDGE_MASTER_PORT_HRDATA: in bit_vector( 31 downto 0 );
66      AHB_BRIDGE_MASTER_PORT_HREADY: in bit;
67      AHB_BRIDGE_MASTER_PORT_HRESP: in bit_vector( 1 downto 0 );
68      -- the ports here are from interface AHB_BRIDGE_SB_SLAVE_PORT
69      AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i: in bit_vector( 1 downto 0 );
70      AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i: in bit;
71      AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i: in bit;
72      AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i: in bit_vector( 23 downto 0 );
73      AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i: in bit_vector( 31 downto 0 );
```

```
74        AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o: out bit_vector( 31 downto 0 );
75        AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o: out bit;
76        -- the ports here are from interface AHB_BRIDGE_AHB_SLAVE_PORT
77        AHB_BRIDGE_AHB_SLAVE_PORT_HSEL: in bit;
78        AHB_BRIDGE_AHB_SLAVE_PORT_HADDR: in bit_vector( 31 downto 0 );
79        AHB_BRIDGE_AHB_SLAVE_PORT_HWRITE: in bit;
80        AHB_BRIDGE_AHB_SLAVE_PORT_HTRANS: in bit_vector( 1 downto 0 );
81        AHB_BRIDGE_AHB_SLAVE_PORT_HSIZE: in bit_vector( 2 downto 0 );
82        AHB_BRIDGE_AHB_SLAVE_PORT_HBURST: in bit_vector( 2 downto 0 );
83        AHB_BRIDGE_AHB_SLAVE_PORT_HWDATA: in bit_vector( 31 downto 0 );
84        AHB_BRIDGE_AHB_SLAVE_PORT_HRDATA: out bit_vector( 31 downto 0 );
85        AHB_BRIDGE_AHB_SLAVE_PORT_HREADY: out bit;
86        AHB_BRIDGE_AHB_SLAVE_PORT_HRESP: out bit_vector( 1 downto 0 );
87        -- the ports here are from interface AHB_BRIDGE_SYSTEM_PORT
88        AHB_BRIDGE_SYSTEM_PORT_Clk: in bit;
89        AHB_BRIDGE_SYSTEM_PORT_Rst: in bit);
90
91    end component;
92
93
94    -- ### ### ### ### ### ###
95    -- constants, signals, ... for the Assertion IF
96    constant AIF_ID12_AssCnt : integer range 1 to 2 := 2;
97
98
99    --- STATIC
100   signal  AIF_ID12_rd_data    : bit_vector(31 downto 0);
101
102
103   --STATIC 1-time Register for ASS_IF
104   signal AIF_ID12_Ass_Addr_Reg        :   bit_vector(31 downto 0);
105   signal AIF_ID12_Ass_CFG_Reg     :   bit_vector(31 downto 0);
106   signal AIF_ID12_Ass_STAT_Reg        :   bit_vector(31 downto 0);
107   signal AIF_ID12_Ass_ErrInd_Reg  :   bit_vector(1 downto 0);
108
109
110   -- STATIC, btw this construct is synthesizable! no problem at all!
111   subtype     AIF_ID12_REG_WIDTH  is  bit_vector(31 downto 0);
112   type        AIF_ID12_ASS_REG        is  array (integer range 0 to AIF_ID12_AssCnt-1)    of  AIF_ID12_REG_WIDTH;
113   type        AIF_ID12_RES_REG        is  array (integer range 0 to AIF_ID12_AssCnt-1)    of  bit;
114
115
116   -- also static
117   signal AIF_ID12_CFG_REG_x   :   AIF_ID12_ASS_REG;
118   signal AIF_ID12_STAT_REG_x  :   AIF_ID12_ASS_REG;
119   signal AIF_ID12_RES_REG_x   :   AIF_ID12_RES_REG;
120   signal AIF_ID12_RES :   bit;
121
122
123   -- DYNAMIC, DO GENERATE FOR EVERY ASSERTION WHICH
124   -- performs check on output signals
125   signal tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o  :   bit_vector( 31 downto 0 );
126   signal tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o    :   bit;
127
128
129   -- DUMMY SIGNAL
130   signal dummy_signal : bit;
131
132
133
134   component AIF_ID12_ScopeAdr5_AssAdr77 is
135
136       port (
137       -- These ports are for the Assertion IF
138       CFG_REG:    in  bit_vector(31 downto 0);
139       STAT_REG:   out bit_vector(31 downto 0);
140       RES_REG:    out bit;
141       -- These ports are for the assertion itself
142       AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i: in  bit_vector( 1 downto 0 );
143       AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i:    in  bit;
144       AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i: in  bit;
145       AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i:    in  bit_vector( 23 downto 0 );
146       AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i:    in  bit_vector( 31 downto 0 );
```

```
147       AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o:    in  bit_vector( 31 downto 0 );
148       AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o:  in  bit;
149       AHB_BRIDGE_SYSTEM_PORT_Clk: in  bit;
150       AHB_BRIDGE_SYSTEM_PORT_Rst: in  bit);
151   end component;
152
153
154   component AIF_ID12_ScopeAdr5_AssAdr76 is
155
156       port (
157       -- These ports are for the Assertion IF
158       CFG_REG:    in  bit_vector(31 downto 0);
159       STAT_REG:   out bit_vector(31 downto 0);
160       RES_REG:    out bit;
161       -- These ports are for the assertion itself
162       AHB_BRIDGE_SYSTEM_PORT_Clk: in  bit;
163       AHB_BRIDGE_SYSTEM_PORT_Rst: in  bit);
164   end component;
165
166
167
168
169
170   ---------------------------------------------
171   -- ARCHITECTURE CODE PART
172   begin
173
174   -- Map the Wrapper to the real model
175   -- placeholder
176
177   AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o <= tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o;
178
179
180   Instance_ahb_bridge: ahb_bridge
181   generic map(
182       TEST_GEN_1 => TEST_GEN_1,
183       TEST_GEN_2 => TEST_GEN_2)
184
185   port map(
186
187       AHB_BRIDGE_AHB_SLAVE_PORT_HSEL  =>  AHB_BRIDGE_AHB_SLAVE_PORT_HSEL,
188       AHB_BRIDGE_AHB_SLAVE_PORT_HADDR =>  AHB_BRIDGE_AHB_SLAVE_PORT_HADDR,
189       AHB_BRIDGE_AHB_SLAVE_PORT_HWRITE    =>  AHB_BRIDGE_AHB_SLAVE_PORT_HWRITE,
190       AHB_BRIDGE_AHB_SLAVE_PORT_HTRANS    =>  AHB_BRIDGE_AHB_SLAVE_PORT_HTRANS,
191       AHB_BRIDGE_AHB_SLAVE_PORT_HSIZE =>  AHB_BRIDGE_AHB_SLAVE_PORT_HSIZE,
192       AHB_BRIDGE_AHB_SLAVE_PORT_HBURST    =>  AHB_BRIDGE_AHB_SLAVE_PORT_HBURST,
193       AHB_BRIDGE_AHB_SLAVE_PORT_HWDATA    =>  AHB_BRIDGE_AHB_SLAVE_PORT_HWDATA,
194       AHB_BRIDGE_AHB_SLAVE_PORT_HRDATA    =>  AHB_BRIDGE_AHB_SLAVE_PORT_HRDATA,
195       AHB_BRIDGE_MASTER_PORT_HRDATA   =>  AHB_BRIDGE_MASTER_PORT_HRDATA,
196       AHB_BRIDGE_MASTER_PORT_HREADY   =>  AHB_BRIDGE_MASTER_PORT_HREADY,
197       AHB_BRIDGE_MASTER_PORT_HRESP    =>  AHB_BRIDGE_MASTER_PORT_HRESP,
198       AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i  =>  AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i,
199       AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i,
200       AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i  =>  AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i,
201       AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i,
202       AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i,
203       AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o =>  tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o,
204       AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o   =>  tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o,
205       AHB_BRIDGE_SYSTEM_PORT_Clk  =>  AHB_BRIDGE_SYSTEM_PORT_Clk,
206       AHB_BRIDGE_SYSTEM_PORT_Rst  =>  AHB_BRIDGE_SYSTEM_PORT_Rst );
207
208
209   Instance_Ass_ID12_ScopeAdr5_AssAdr77: AIF_ID12_ScopeAdr5_AssAdr77
210   port map(
211           CFG_REG =>  AIF_ID12_CFG_REG_x(0),
212           STAT_REG    =>  AIF_ID12_STAT_REG_x(0),
213           RES_REG =>  AIF_ID12_RES_REG_x(0),
214           AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i  =>  AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i,
215           AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i,
216           AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i  =>  AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i,
217           AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i,
218           AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i,
219           AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o =>  tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o,
```

```
220           AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o    =>  tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o,
221           AHB_BRIDGE_SYSTEM_PORT_Clk   =>  AHB_BRIDGE_SYSTEM_PORT_Clk,
222           AHB_BRIDGE_SYSTEM_PORT_Rst   =>  AHB_BRIDGE_SYSTEM_PORT_Rst);
223
224
225   Instance_Ass_ID12_ScopeAdr5_AssAdr76: AIF_ID12_ScopeAdr5_AssAdr76
226   port map(
227           CFG_REG =>  AIF_ID12_CFG_REG_x(1),
228           STAT_REG    =>  AIF_ID12_STAT_REG_x(1),
229           RES_REG =>  AIF_ID12_RES_REG_x(1),
230           AHB_BRIDGE_SYSTEM_PORT_Clk   =>  AHB_BRIDGE_SYSTEM_PORT_Clk,
231           AHB_BRIDGE_SYSTEM_PORT_Rst   =>  AHB_BRIDGE_SYSTEM_PORT_Rst);
232
233
234
235
236
237   -- processes for the Assertion IF (IF-Demux, IF-Register R/W, Virtual Reg Demux, Result-Demux)
238   --#############################################################################
239
240
241   --decoder part
242   --select either the real IP or the Assertion IF
243   --    AND
244   --READ_DATA demulitplexer
245   DECODE_IF_ID_12:
246   process (AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i, AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i,
      tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o, AIF_ID12_rd_data, AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i,
      AIF_ID12_Ass_STAT_Reg, AIF_ID12_Ass_CFG_Reg, AIF_ID12_Ass_ErrInd_Reg, AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i,
      AIF_ID12_Ass_Addr_Reg, AIF_ID12_STAT_REG_x)
247
248   variable tmp_stat : bit_vector(31 downto 0);
249   variable AIF_ID12_enable : bit;
250
251   begin
252
253       if AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i = '1' then
254
255           AIF_ID12_enable := '0';
256
257           --AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i( <<ASS_IF_MUX_BIT_OF_ADD>> )
258           case AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i(23) is
259               when '0' => AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o <=  tempout_AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o;
260
261               when '1' => AIF_ID12_enable := '1';
262
263                           ASSIF_SELECT:
264                           REPORT "ADDR(23) = 1, ACCESSING ASSERTION IF ID12!"
265                           severity NOTE;
266
267                           AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o <=  AIF_ID12_rd_data;
268
269               when others =>  AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o <= (others => '0');
270
271           end case;
272
273       else
274
275           AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o <= (others => '0');
276           AIF_ID12_enable := '0';
277
278       end if;
279
280
281
282
283   --#############################################################################
284   --write register part(bus specific!)
285   -- check if ass_if is enabled and perform write/read action of 1of4 register
286
287
288       if AIF_ID12_enable = '1' then
289
```

```
290             --use leftmost 2 bits of the (22:0) remaining bits
291             case AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i(1 downto 0) is
292                 when "00"   =>  AIF_ID12_ASS_ADR_SELECT:
293                                 REPORT "AIF_ID12: Assertion Address Register selected"
294                                 severity NOTE;
295
296                                 if  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =   '0' then
297                                     AIF_ID12_rd_data    <=  AIF_ID12_Ass_Addr_Reg;
298                                 else
299                                     AIF_ID12_Ass_Addr_Reg <=    AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i;
300                                 end if;
301
302
303
304                 when "01"   =>  AIF_ID12_ASS_CFG_SELECT:
305                                 -- READ and WRITE is allowed
306
307                                 REPORT "AIF_ID12: Assertion Config Register selected"
308                                 severity NOTE;
309
310                                 if  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =   '0' then
311                                     AIF_ID12_rd_data    <=  AIF_ID12_Ass_CFG_Reg;
312                                 else
313                                     AIF_ID12_Ass_CFG_Reg <= AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i;
314                                 end if;
315
316
317
318                 when "10"   =>  AIF_ID12_ASS_STAT_SELECT:
319                                 -- ONLY READ IS ALLOWED
320                                 REPORT "AIF_ID12: Assertion Status Register selected"
321                                 severity NOTE;
322
323                                 if  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =   '0' then
324                                     AIF_ID12_rd_data    <=  AIF_ID12_Ass_STAT_Reg;
325                                 else
326                                     REPORT "AIF_ID12: Assertion Status Register CANNOT be written from Bus Side"
327                                     severity WARNING;
328                                 end if;
329
330
331
332                 when "11"   =>  AIF_ID12_ERR_IND_SELECT:
333                                 REPORT "AIF_ID12: Error Indicating Register selected"
334                                 severity NOTE;
335
336                                 if  AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i =   '0' then
337
338                                     AIF_ID12_rd_data    <=  (others => '0');
339                                     AIF_ID12_rd_data(1 downto 0)    <=  AIF_ID12_Ass_ErrInd_Reg;
340                                 else
341
342
343                                     --ONLY BIT 1 can be written by Software, else
344                                     AIF_ID12_Ass_ErrInd_Reg(1)  <=  AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i(1);
345                                 end if;
346
347
348                 when others =>  AIF_ID12_rd_data    <= (others => '0');
349
350             end case;
351
352         end if;
353
354
355
356
357     --#############################################################################
358     -- real register to virtual register map
359     -- NEEDS TO GET GENERATED
360
361         case AIF_ID12_Ass_Addr_Reg(31 downto 28) is
362
```

```
363           when x"0"    =>  tmp_stat     :=  (others => '0');
364
365                            for I in 0 to (AIF_ID12_AssCnt-1) loop
366                                AIF_ID12_CFG_REG_x(I)    <=  AIF_ID12_Ass_CFG_Reg;
367                                tmp_stat     :=  tmp_stat OR AIF_ID12_STAT_REG_x(I);
368                            end loop;
369
370
371                            AIF_ID12_BROADCAST:
372                            REPORT "AIF_ID12: SCOPE = 0, Broadcast to all Assertions!"
373                            severity NOTE;
374
375
376           when others =>
377
378           --#####################################################################
379           -- THE REAL "SCOPE & ASSID" to internal register mapping!
380           -- NEEDS TO GET GENERATED
381
382
383                            case AIF_ID12_Ass_Addr_Reg  is
384
385
386
387                            when x"5_000004d"    =>   --SCOPE No. 0, ASSERTION No. 0
388                                                      AIF_ID12_CFG_REG_x(0)    <=  AIF_ID12_Ass_CFG_Reg;
389                                                      tmp_stat     :=  AIF_ID12_STAT_REG_x(0);
390
391                            when x"5_000004c"    =>   --SCOPE No. 0, ASSERTION No. 1
392                                                      AIF_ID12_CFG_REG_x(1)    <=  AIF_ID12_Ass_CFG_Reg;
393                                                      tmp_stat     :=  AIF_ID12_STAT_REG_x(1);
394
395
396                            when others      =>  REPORT "There is no Assertion with that Address = SCOPE & ASS_ID"
397                                                 severity WARNING;
398
399                                                 tmp_stat := (others => '0');
400
401                            end case;
402
403       end case;
404
405
406       AIF_ID12_Ass_STAT_Reg    <=  tmp_stat;
407
408
409   end process;
410
411
412
413   -- Assertion result demuxer
414   -- STATIC
415   AIF_ID12_RES_REG_x_or_12:
416       process (AIF_ID12_RES_REG_x)
417           variable tmp_res : bit;
418
419       begin
420
421           tmp_RES :=  '0';
422           for I in 0 to (AIF_ID12_RES_REG_x'high) loop
423               tmp_RES    :=    tmp_RES    OR    AIF_ID12_RES_REG_x(i);
424           end loop;
425
426           AIF_ID12_RES <= tmp_RES;
427
428       end process;
429
430   -- processes for the Assertion IF (IF-Demux, IF-Register R/W, Virtual Reg Demux, Result-Demux)
431
432
433   end Wrapper_Arch_ahb_bridge;
434
435
```

A.6. Assertion Interface Example

```
1  library ieee;
2  entity COMPONENT_ahb_bridge_AIF_ID12_ScopeAdr5_AssAdr77 is
3
4      port(
5      CFG_REG:    in  bit_vector(31 downto 0);
6      STAT_REG:   out bit_vector(31 downto 0);
7      RES_REG:    out bit;
8      AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i: in bit_vector( 1 downto 0 );
9      AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i: in bit;
10     AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i: in bit;
11     AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i: in bit_vector( 23 downto 0 );
12     AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i: in bit_vector( 31 downto 0 );
13     AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o: in bit_vector( 31 downto 0 );
14     AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o: in bit;
15     AHB_BRIDGE_SYSTEM_PORT_Clk: in bit;
16     AHB_BRIDGE_SYSTEM_PORT_Rst: in bit);
17
18 end COMPONENT_ahb_bridge_AIF_ID12_ScopeAdr5_AssAdr77;
19
20
21 architecture COMPONENT_ahb_bridge_AIF_ID12_ScopeAdr5_AssAdr77_Arch of
   COMPONENT_ahb_bridge_AIF_ID12_ScopeAdr5_AssAdr77 is
22 begin
23
24 -- Assertion: RST_Assertion, Scope: Periphal_Assertions_Group
25 Process_AIF_ID12_ScopeAdr5_AssAdr77:
26     process (AHB_BRIDGE_SB_SLAVE_PORT_S_DataSize2_i, AHB_BRIDGE_SB_SLAVE_PORT_S_Wr_i,
   AHB_BRIDGE_SB_SLAVE_PORT_S_AccEn_i, AHB_BRIDGE_SB_SLAVE_PORT_S_Addr24_i, AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_i,
   AHB_BRIDGE_SB_SLAVE_PORT_S_Data32_o, AHB_BRIDGE_SB_SLAVE_PORT_S_Wait_o, AHB_BRIDGE_SYSTEM_PORT_Clk,
   AHB_BRIDGE_SYSTEM_PORT_Rst)
27
28         --#################################################
29         --DO ALWAYS GENERATE THE ALIAS below
30
31         -- MANAGING ASSERTION
32         alias CFG_Enable        : bit is CFG_REG(0);
33         alias CFG_ClearRes      : bit is CFG_REG(1);
34
35         -- EXECUTION SEMANTICS
36         alias CFG_RestartOnEC   : bit is CFG_REG(2);
37         alias CFG_Threading     : bit is CFG_REG(3);
38
39         -- RESET
40         alias CFG_Reset     : bit is CFG_REG(4);
41         alias CFG_Failures  : bit is CFG_REG(5);
42         alias CFG_Coverage  : bit is CFG_REG(6);
43
44         -- QueryDebug Informations
45         alias CFG_getAssertionFail      : bit is CFG_REG(7);
46         alias CFG_getAssertionCover     : bit is CFG_REG(8);
47         alias CFG_getAssertionThreads   : bit is CFG_REG(9);
48         alias CFG_getErrorID            : bit is CFG_REG(10);
49
50                 -- ALIAS FOR REGISTERS
51         alias RESULT                    : bit is RES_REG;
52         alias STATUS                    : bit_vector is STAT_REG;
53
54         --#################################################
55         -- PLACE CUSTOM variables in here (fail_cnt, cover_cnt, ...)
56
57     begin
58     -- This Assertion models the property:
59
60     if AHB_BRIDGE_SYSTEM_PORT_Rst = '0' then
61             RESULT <= '0';
62     elsif AHB_BRIDGE_SYSTEM_PORT_Clk = '1' and AHB_BRIDGE_SYSTEM_PORT_Clk'event then
63
64     -- insert checker functionality here
65
66     end if;
67     end process;
68
69 end COMPONENT_ahb_bridge_AIF_ID12_ScopeAdr5_AssAdr77_Arch;
```