Informatics

Intelligentes Testen und Konfiguration von SoCs durch die Nutzung von IJTAG ergänzt durch on-chip Mikroprozessor-Zugriff

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Clemens Pircher, BSc
Matrikelnummer 01525889

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Mitwirkung: Dipl.-Ing. Heimo Hartlieb
Univ.Ass. Dipl.-Ing. Florian Ferdinand Huemer, BSc

Wien, 3. Februar 2022

Clemens Pircher   Andreas Steininger

Informatics

Smart SoC testing and remote configuration facilitated by the use of IJTAG complemented with on-chip microprocessor access

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Clemens Pircher, BSc
Registration Number 01525889

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger
Assistance: Dipl.-Ing. Heimo Hartlieb
Univ.Ass. Dipl.-Ing. Florian Ferdinand Huemer, BSc

Vienna, 3rd February, 2022

Clemens Pircher   Andreas Steininger

Erklärung zur Verfassung der Arbeit

Clemens Pircher, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Februar 2022

Clemens Pircher

Danksagung

Im Laufe dieser Arbeit habe ich viel fachliche und emotionale Unterstützung durch Personen in meinem Umfeld erfahren. Ich möchte allen von ihnen aus meinem tiefsten Herzen danken.

Zunächst möche ich Heimo Hartlieb und Wolfgang Ecker meinen Dank für die Möglichkeit an diesem spannenden Projekt zu arbeiten aussprechen. Insbesondere will ich euch auch für die erfrischenden und ergiebigen Diskussionen, sowie die ständige unbezahlbare Unterstützung danken. Ein weiteres großes Dankeschön gebührt Heimo auch dafür, dass er diese Arbeit über seine Feiertage korrekturgelesen hat.

Ich möchte mich auch bei meinem Betreuer Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger für seine Unterstützung, Flexibilität und Zuvorkommenheit im Fachlichen und Organisatorischen erkenntlich zeigen. Auch er war immer schnell zur Stelle und sehr engagiert mich in meinen Zeitplänen zu unterstützen.

Weiters möchte ich unserem Manager Gerald Derflinger dafür danken, dass er mir während einer globalen Pandemie immer den Rücken freigehalten hat.

Ich möchte auch allen Kollegen bei Infineon für ihre Hilfe, aber auch für den Teamgeist und die generell gute Zeit danken. Es hat mir viel Spaß gemacht mit euch allen zu arbeiten. Ein großes Dankeschön gebührt Keerthikumara Devarajegowda und Paritosh Sinha für ihre rasche Unterstützung bei allen Fragen und Problemen bezüglich Metagen und MetaRTL. Ich schulde auch Michael Werner meinen Dank und zumindest einen Abend voller Bier für seine Unterstützung bei allen MetaFirm-betreffenden Themen und für die schnelle Implementierung von Erweiterungen welche für diese Arbeit nötig waren. Ich möchte auch Zhao Han für unsere Diskussionen und seine Code Reviews danken. Spezieller Dank gebührt Timotei Muresan für die Verifikation meiner generierten Designs. Desweiteren möchte ich Thomas Grubelnik dafür danken, dass er bei der SoC Integration geholfen hat und mich während stundenlangen Debug-Sessions bei Laune gehalten hat. Ich danke auch Evren Kenanoglu für seine Hilfe beim Aufsetzen der Software Toolchain und der Modifikation der genutzten Linker Skripte.

Zu guter Letzt möchte ich meiner Freundin, meiner Familie und meinen Freunden meinen tiefsten Dank für ihre überwältigende Unterstützung und ständige Ermutigung während meines Studiums und dieser Arbeit aussprechen. Ohne euch hätte ich das alles nicht geschafft.

Acknowledgements

Over the course of working on this thesis, a lot of people have provided me with technical and emotional support. I want to thank all of them from the bottom of my heart.

In particular, I would like to express my gratitude to Heimo Hartlieb and Wolfgang Ecker for the opportunity to work on this exciting project. I also want to thank both of you for the refreshing and fruitful discussions that we had in the process and the constant and invaluable support that I have received from you. Moreover, a very big thanks to Heimo for proofreading this thesis during his holidays.

I am also very grateful to my supervisor Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Steininger for his support, flexibility and forthcomingness in technical and organizational matters. He was always promptly available and committed to supporting me in my schedule.

I must also thank our manager Gerald Derflinger for always having my back while working through a global pandemic.

Moreover, I want to thank all my colleagues at Infineon not only for the help they have offered but also for their good team spirit and in general the good times that we had. I had a blast working with you all. Big thanks go to Keerthikumara Devarajegowda and Paritosh Sinha for their prompt support with any Metagen- and MetaRTL-related questions and issues. I also owe gratitude and at least an evening's worth of beer to Michael Werner for his support with any MetaFirm-related topics and for providing the library extensions that were required for this thesis. I also want to thank Zhao Han for our discussion and his feedback during code reviews. Special thanks go to Timotei Muresan for spending the time to verify my generated designs. I also want to thank Thomas Grubelnik for his help with the SoC integration and for humoring me during seemingly endless hours of debugging. Furthermore, I want to thank Evren Kenanoglu for his help with the software toolchain setup and linker script modifications.

Last but not least, I want to express my deepest gratitude to my girlfriend, my family and my friends for their overwhelming support and words of encouragement throughout my studies and this thesis. I could not have done it without you.

Kurzfassung

Die ständige Zunahme an integrierten Komponenten in modernen SoCs sorgt für einen enormen Anstieg der Komplexität. Dies wirkt sich nicht nur auf die digitalen Design- und Verifikationsprozesse aus, sondern auch auf die End of Line Tests der fabrizierten ICs deren Anteil an den Produktionskosten immer signifikanter wird. Gängige Design For Test Praktiken können unter Zuhilfenahme der CPU genutzt werden um den Parallelismus von End of Line Tests zu erhöhen und die Testzeit zu verkürzen. Diese Methode ermöglicht auch die Wiederverwendung von Scanketten wodurch mit minimalem Mehraufwand Selbsttests sowie das Trimmen und die Konfiguration von analogen Bauteilen zur Laufzeit genutzt werden kann. Die vorliegende Arbeit präsentiert ein Framework aus Hardware- und Softwaregeneratoren welches in der Lage ist eine modulare on-chip Testinfrastruktur auf der Basis der etablierten JTAG und IJTAG Standards zu generieren. Aus einer Spezifikation kann mittels der Generatoren ein systemspezifisches Reconfigurable Scan Network aus unabhängigen und wiederverwendebaren Subnetzwerkdefinitionen zusammengestellt werden. Durch ein eigens entwickeltes Modul welches Datenverkehr zwischen der CPU und dem Reconfigurable Scan Network ermöglicht können softwarebasierte Selbsttest-, Selbsttrim- und Konfigurationsabläufe zur Verfügung gestellt werden. Die Generierung von spezifischen Treiber aus high-level Definitionen von Scanoperationen nahe der standardisierten Procedural Description Language unterstützt die Entwicklung der entsprechenden Software. Dadurch fördert der beschriebene Ablauf nicht nur die Wiederverwendung von Intellectual Properties und Embedded Instruments sondern auch die Wiederverwendung der relevanten Software Routinen. Schlussendlich wird die Integration eines entsprechenden Designs in einen RISC-V-basierten SoC präsentiert. Die Entwicklung einer Komparator-basierten Selbsttrim-Applikation zeigt die Flexibilität, Modularität und Produktivitätssteigerung des vorgestellten Frameworks auf.

Abstract

The ever-increasing number of embedded elements in modern SoCs is a major driver for growing design complexity. This affects not only the digital design and verification processes but also End of Line testing of manufactured ICs whose fraction of overall production cost is becoming more significant. Making Design For Test features accessible to the CPU can augment End of Line testing by increasing parallelism and reducing test time. Moreover, reusing scan chains in this way can provide valuable Built-In Self-Test, analog trimming and analog configuration mechanisms in the field with little to no additional hardware overhead. This thesis presents a hardware and firmware generator framework that is capable of generating a modular on-chip testing infrastructure based on the established JTAG and IJTAG standards. By providing a single source specification to the generators, a system-specific Reconfigurable Scan Network implementation is compiled from independent and reusable subnetwork definitions. A specially designed peripheral that manages traffic between the CPU and Reconfigurable Scan Network enables the provision of software-based self-testing, self-trimming and analog configuration. The generation of custom drivers from high-level scan operation descriptions close to the standardized Procedural Description Language further supports the development of the respective software. In doing so, the proposed flow does not only foster reuse of Intellectual Properties and Embedded Instruments but also reuse of test- and trim-related software routines. In the end, integration of a particular design in a RISC-V-based SoC is presented. The development of a comparator-based self-trim application is demonstrated to underline the flexibility, modularity and productivity that the proposed flow offers.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the past decades, the complexity of Integrated Circuits (ICs) has been increasing rapidly while manufacturing costs and feature sizes have diminished [1]. Besides the enormous design and verification effort required to keep up with these fast-paced developments, the increased risk of manufacturing-induced defects necessitates thorough End of Line (EOL) testing [1, 2]. The use of Design For Test (DFT) techniques like scan chains and Built-In Self-Test (BIST) is inevitable to uphold or even improve upon product standards [3, 1, 4, 5].

With the complexity and number of embedded elements in modern Systems On Chips (SoCs) increasing, so do the number of Embedded Instruments (EIs) required to achieve acceptable test coverage. This further adds to the test time and test costs causing their share in overall production expenses to become more significant [2, 6]. Moreover, the time required to develop the tests should not be underestimated [1]. Integration of BIST engines can help to alleviate the demands and burden on Automated Test Equipment (ATE) during EOL testing while also offering valuable in-field tests but they come at the expense of increased area overhead [1, 6, 5]. In addition, to test requirements for digital and analog Intellectual Properties (IPs), analog blocks also require trimming which takes up a large portion of test time [7, 8] and cannot be parallelized on ATE easily [8]. Moreover, the required runtime configuration can be problematic because analog IPs usually do not feature bus interfaces.

The classical scan-inspired approach as described in the IEEE 1149.1 [9], while well supported by ATE and Electronic Design Automation (EDA) tools, is becoming unwieldy given the high number of EIs which complicates the test-related aspects of IP-reuse. This is even more relevant when targeting an entire product family range with a variety of feature sets among the individual SoCs. Moreover, the EOL test features add to the overall chip area but usually do not serve any purpose for the rest of the product's lifetime.

Therefore, the (re-)use of scan chains with minimal per-IP overhead is of interest for the purpose of self-trimming, in-field testing and in-field configuration. The fact that an SoC already comes with a Central Processing Unit (CPU) suggests that enabling on-chip access to the scan chains can be of enormous help in reaching these goals in a flexible and future-proof way.

1.2 Objectives

The aim of this thesis is to develop an on-chip testing infrastructure to overcome the mentioned limitations. Specifically, it is essential to support conventional and transparent EOL testing as defined in IEEE 1149.1 combined with the additional benefits of smart software-driven self-testing, self-trimming and runtime configuration based on scan chain reuse. For this purpose, a strategy that allows interaction with scan chains via the CPU needs to be devised.

In order to ease IP and test-reuse and ensure future extensibility, recent developments with respect to flexible scan chain management [10] have to be taken into account. Moreover, the proposed architecture should be modular and flexible in its components to cover different on-chip testing requirements on a per-product basis.

Introducing an interface between the CPU and a set of scan chains also implies bridging the hardware/software boundary. Since dealing with scan chains directly is an intricate matter, having to work with a simple Hardware Abstraction Layer (HAL) is too tedious for software engineers. Therefore, productive development of the required test software must be ensured by provisioning reasonable high-level tool support.

As a demonstration of the achieved results, a practical example that illustrates the opportunities achieved by the integration of the described approach in an SoC shall be presented. Specifically, the aim is to showcase the reuse and flexibility made possible by design decisions at a hardware level as well as how a high level of abstraction in the firmware toolchain fosters software development.

CHAPTER 2

Background

Over the course of this thesis, it is essential to have a firm understanding of scan architectures. Another key requirement is a good understanding of metamodeling and how automated hardware generation is practiced at Infineon. Lastly, it is beneficial to understand the basic architecture of the targeted SoCs and how certain parts of the toolchain are involved during development. This chapter gives an overview of the necessary theoretical prerequisites for this thesis.

2.1 Scan Chain Architecture

Scan chains are without a doubt the most used concept when it comes to DFT.The key idea is to arrange Flip Flops (FFs) in a chain during testing so they act as a shift register. This can provide any level of observability and controllability of integrated circuits with a minimal interface. The goal of this section is to introduce standardized and time-proven (boundary) scan architecture concepts as well as recent developments for dealing with the ever-increasing complexity of SoCs.

2.1.1 Boundary Scan Architecture

About three decades ago, the Joint Test Action Group (JTAG) finalized the original IEEE 1149.1 standard [9] focusing on the description of a Test Access Port (TAP) and boundary scan architecture. The primary purpose was to provide a standardized approach for the testing of IC to Printed Circuit Board (PCB) interconnect as well as the IC itself. Oftentimes, the 1149.1 standard is referred to as the JTAG standard.

Figure 2.1 gives a basic overview of the scan architecture. The centerpiece is the so-called TAP controller which is driven via a TAP interface consisting of four digital signals:

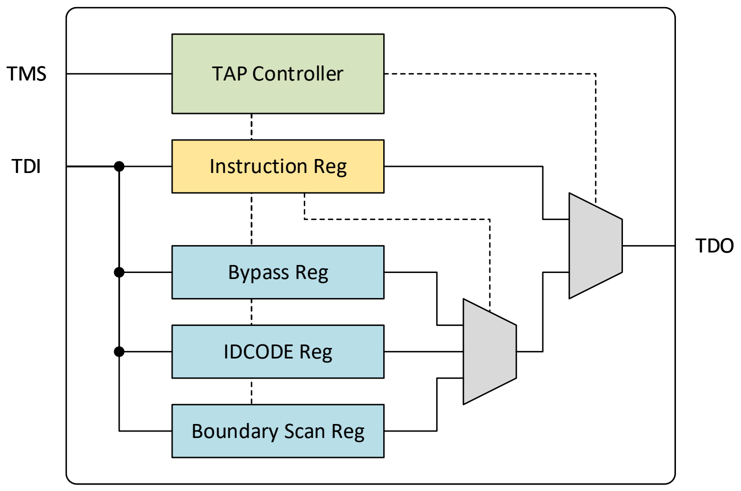


Figure 2.1: Simple overview of the boundary scan architecture as outlined in the standard. Solid lines are data signals and dashed lines are control signals.

- The Test Clock (TCK) which is responsible for clocking of the TAP controller and the scan chains.
- The Test Mode Select (TMS) is used to change the state of the TAP controller at the rising edge of TCK.
- The Test Data Input (TDI) is the serial input clocked into the system at the rising edge of TCK.
- The Test Data Output (TDO) is the serial output clocked out of the system at the falling edge of TCK.

Figure 2.2 shows how the TMS signal controls the scan architecture. Essentially, when the TAP controller is not idle, it is either modifying the Instruction Register (IR) or a Test Data Register (TDR) (e.g. *Bypass*, *IDCODE*, *Boundary Scan Register*, ...).

The content of the IR determines which TDR is currently selected and would be operated on during a data scan operation. Moreover, multiple values of the IR may select the same TDR but cause a difference in the mode of operation. A single access to a TDR (called scan operation) consists of a *capture* operation (data is loaded into the register), a series of *shift* operations through the register and an *update* of the TDR (data shifted into the register takes effect).

All TDRs are chains of so-called TDR cells. Figure 2.3 shows a schematic of a so-called capture-update cell which consists of a capture-shift register and an update-register. The former has two roles in this circuit. Firstly, it acts as a stage in a shift register chain during shift operations in that it fetches data from the serial input (SI) and provides its content to the serial output (SO). Moreover, during *capture* operations, it captures data

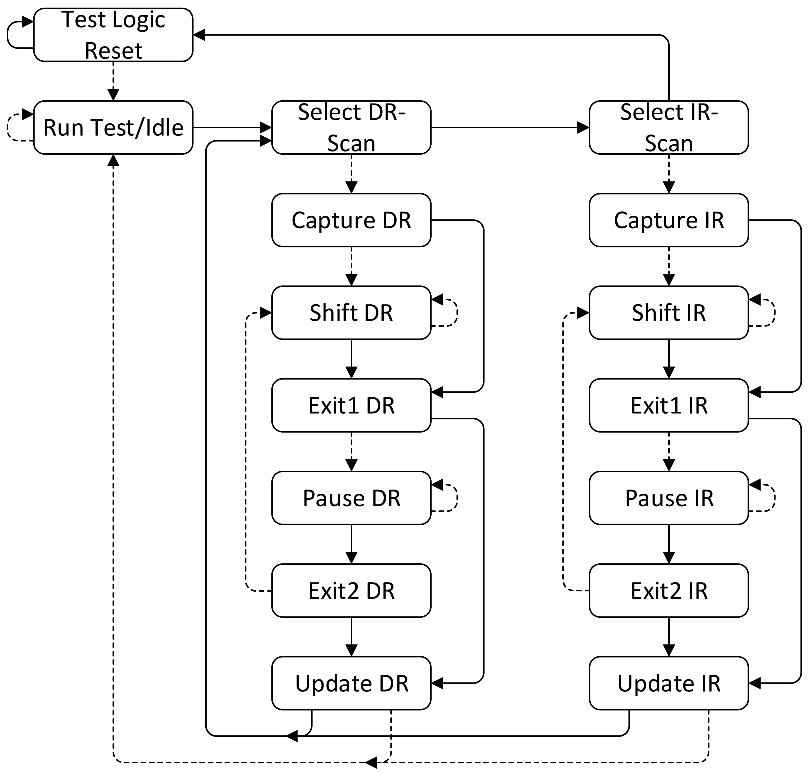


Figure 2.2: State diagram of the TAP controller as described in the standard. Solid lines are transitions taken when TMS=1 while dashed lines are transitions taken when TMS=0.

from the parallel input (i.e. observes a circuit or PCB signal via *PI*). In contrast, the update-register fetches the content of the capture-shift register during *update* operations and provides its state onto a parallel output (i.e. controls a circuit or PCB signal via *PO*). The control signals *CaptureEn*, *ShiftEn* and *UpdateEn* are generated by the TAP controller in the respective states. It is worth noting that *capture* and *shift* operations take place on the rising edge of TCK while *update* operations take place on the falling edge.

Figure 2.4 illustrates a simple scan operation on a 3-bit TDR. As specified in the standard, the Least Signigicant Bit (LSB) is always shifted into the TDR first. This means that the cell that represents the Most Signigicant Bit (MSB) is closest to TDI and farthest from TDO.

In general, depending on the specific needs, TDR cells can have varying functionality. Leaving out the update register results in a capture cell while leaving out the capture logic (*CaptureEn*-driven multiplexer) results in an update cell. Waiving both results in a shift-only cell. Missing capture functionality means that observing signals via the cell is not possible anymore. Notably, missing update functionality does not imply a

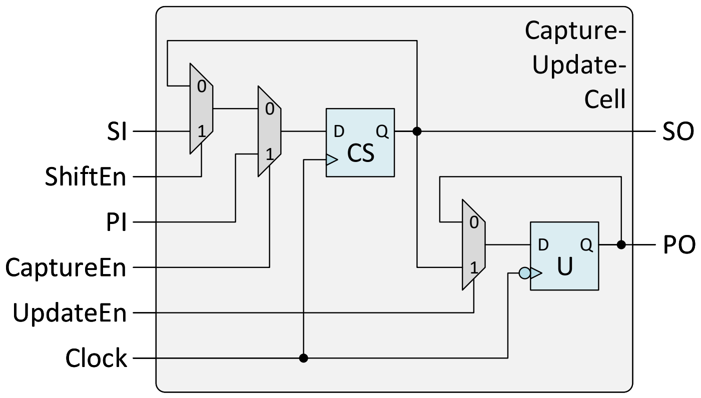


Figure 2.3: Schematic of a capture-update cell as described by the standard.

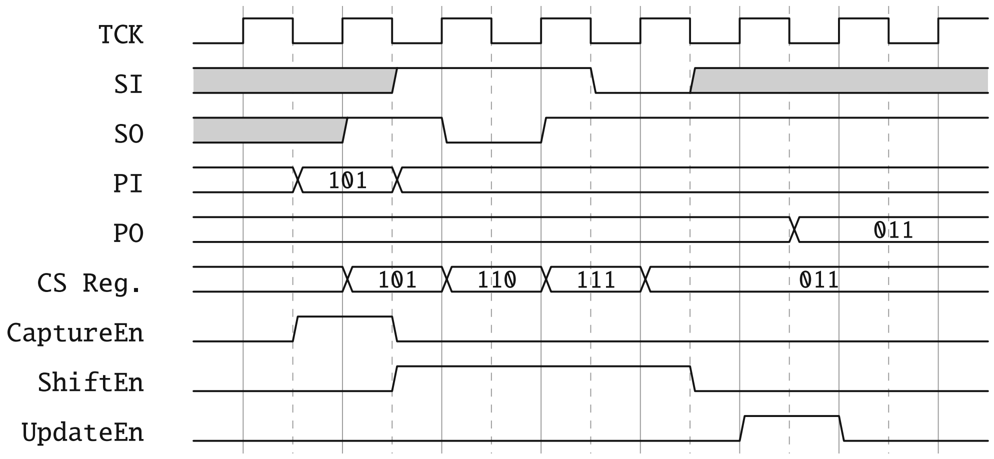


Figure 2.4: Timing diagram demonstrating a scan operation on a TDR via the signals generated by the TAP controller. In this case, `101` is captured from the parallel input and `011` is shifted in and updated to the parallel output.

loss of controllability. However, without the update register, there is no way of ensuring consistent updates and *shift* operations cause rapid signal changes at cell outputs.

The idea of boundary scan is to provide a TDR of so-called boundary scan cells around the input and output pins of an IC. This TDR which is often referred to as Boundary Scan Chain (BSC) allows the application of stimuli and observation of signals at the IC's border. A boundary scan cell is essentially a TDR cell whose parallel output is not connected directly to the circuit or PCB but instead multiplexed with a default signal, usually its parallel input, based on a mode select signal. This makes it possible to control the operation mode (i.e. test or normal operation) independently of scan operations on the BSC. Controlling the output cells and observing the input cells enables testing of interconnect. In contrast, controlling input cells and observing output cells allows testing of the IC's functionality.

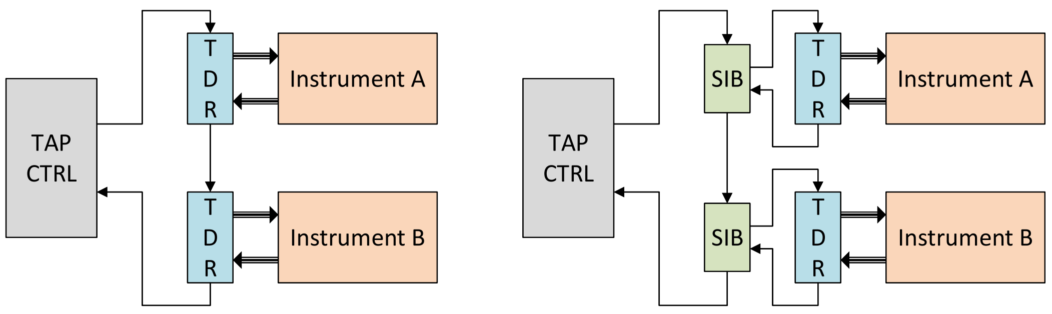
The IEEE 1149.1 standard requires implementation of at least the bypass register, a simple 1-stage shift register for when no operation is required, and the *Boundary Scan Register* for boundary scans. There are further standardized TDRs like *IDCODE* which provides a device identification. Further, it is possible to implement custom TDRs as needed (e.g. for operation of an On Chip Debug System (OCDS)).

For operation of the TDRs, the standard enforces provision of four instructions:

- A *BYPASS* instruction to select the bypass register.
- A *PRELOAD* instruction which can be used to load data into the BSC's update stage without asserting the test mode.
- A *SAMPLE* instruction which can be used to fetch data into the BSC's capture stage without asserting the test mode.
- An *EXTEST* instruction which asserts the test mode and also allows scan operations to be performed on the BSC. This is inteded for testing PCB connectivity.

It is permitted to combine instructions that do not conflict with one another meaning that implementation of a *SAMPLE_PRELOAD* instruction is also allowed. Moreover, there are several other standardized instructions like *INTEST* which is similar to *EXTEST* but targets testing of the IC itself. For that purpose, it also has to support single stepping in the TAP controller's *idle* state.

The JTAG architecture has been used for decades and is supported by practically every relevant EDA tool. Nevertheless, it does come with some problems. Making a design scannable by replacing FFs with Scannable Flip Flops (SFFs) is possible by implementing a custom TDR and making it selectable with a custom instruction. However, in modern SoCs where multiple embedded elements require a TDR each, one has to decide whether they should be arranged in series or in parallel. In the first case, accessing a part of the TDR sequence means that data has to traverse the entire scan chain which causes a noticeable increase in test time. Moreover, special care has to be taken to avoid corrupting the state of other parts of the scan chain. Putting the TDRs in parallel does solve both of the problems but also does not allow any patterns where the TDRs of multiple EIs are accessed at the same time which is another cause of increased test time. Moreover, it would require introducing a lot of instructions which ultimately causes strong coupling between the TAP controller and the feature set of a system. In a way, the described scan chain architecture is too rigid for the purposes of modern SoCs. Furthermore, retargeting of tests due to product variety and IP reuse is very tedious if not impossible. While the newest revision of the standard introduced some improvements regarding these aspects, an entirely new standard has evolved with the goal of extending IEEE 1149.1 to fix those exact problems.



(a) Scan chain of two TDRs in series. (b) Scan chain of two TDRs guarded by SIBs.

Figure 2.5: Example of how SIBs are used in the IJTAG network to make it configurable.

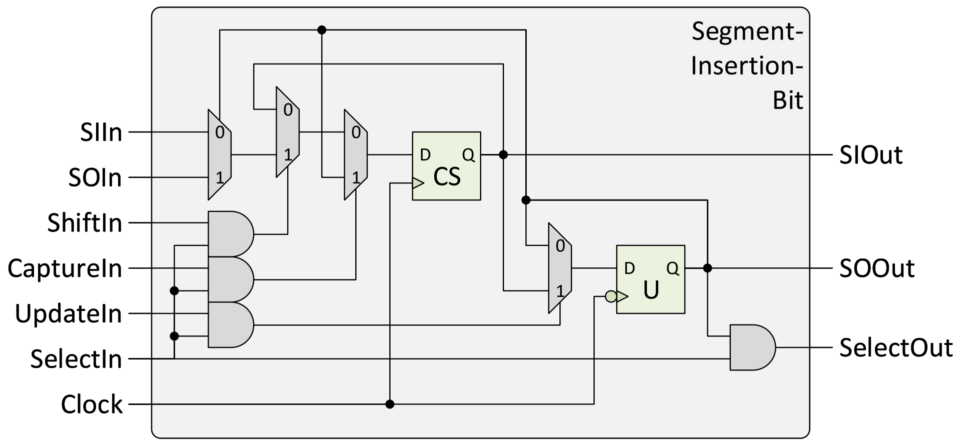


Figure 2.6: Schematic of a possible implementation of an SIB as specified in the standard.

2.1.2 A flexible and reconfigurable Scan Network

The Integrated Joint Test Action Group (IJTAG) has recognized that the TAP is used for many more purposes than just boundary scan and identified the need for an Reconfigurable Scan Network (RSN) based on scan chains for increased flexibility when accessing EIs as well as retargeting capabilities. This caused the emergence of the IEEE 1687 standard [10].

The basic idea is to allow flexible inclusion and exclusion of parts of a scan chain. An elementary building block to achieve this is the Segment Insertion Bit (SIB). It can be thought of as a single bit update cell that acts as a kind of TDR-guard. When it is active, the TDR is included in the scan chain right before the SIB's internal shift stage. Deasserting the SIB causes the TDR to be excluded from the scan chain. In this case, it essentially acts as a bypass register. Figure 2.5 outlines how SIBs can be used to guard subnetworks. Of course, a subnetwork guarded by an SIB may contain SIBs of its own which enables the construction of multi-level hierarchies.

A possible implementation of the SIB is visualized in Figure 2.6. It can be seen as a modified capture-update cell that multiplexes the input of its shift stage between the serial input of its predecessor (*SIIn*) and the serial output of the TDR it is guarding (*SOIn*). Moreover, it generates a *SelectOut* signal for the subnetwork it is guarding which can be used to mask the control signals of the subnetwork to freeze it when the SIB is de-asserted.

The SIB is an example of an in-line scan chain multiplexer meaning that the control signal is generated from the same TDR as the SIB is configuring. An alternative would be a remotely controlled multiplexer whose control signal is being generated from a different register (e.g. another TDR or a decoded instruction). The main difference between the two approaches is that the SIB allows updating the configuration for the next scan operation during the current scan operation while remotely controlled scan multiplexers may require changing the IR.

Besides providing methods to allow more efficient and flexible operation on scan chains, the standard also provides definitions for two description languages. The first one is the Instrument Connectivity Language (ICL) which makes it possible to define the structure of the RSN for a given product. This allows efficient retargeting of operations on TDRs written in the second language called Procedural Description Language (PDL). The advantage is that sequences of operations can be developed at the level of EIs. Given the ICL definition of a system (e.g. an SoC) a toolchain can then retarget a set of PDL definitions to the system level which significantly improves test development time. Moreover, the development of the operation procedures does not depend in any way on the final feature set of the system itself and can thus take place rather early or also entirely independent of the final system (e.g. third-party IP).

2.2 Automated Code Generation

The concept of automated code generation is a major driver for productivity and a valuable tool for the purpose of closing the design productivity gap [11]. When practiced correctly, it can also offer the same conveniences as IP reuse with the addition of significantly increased flexibility. For this reason, many processes at Infineon build upon a metamodeling-based code generation framework. In conjunction with the concepts of Model Driven Architecture (MDA), this has also opened up opportunities for user-friendly Register Transfer Level (RTL) generation. This section provides an introduction to the concepts of this toolchain which is used throughout this thesis.

2.2.1 Metamodeling

The MDA methodology makes use of models to capture properties of systems at different levels of abstraction. It is therefore clear that a model has to consist of a set of objects. However, a model is essentially worthless without knowledge of its respective domain. Moreover, depending on the domain, a model may have to fulfill certain constraints.

This is where the concept of metamodels comes into play. A metamodel is essentially a structural and relational description of a class of models. One could say that a metamodel

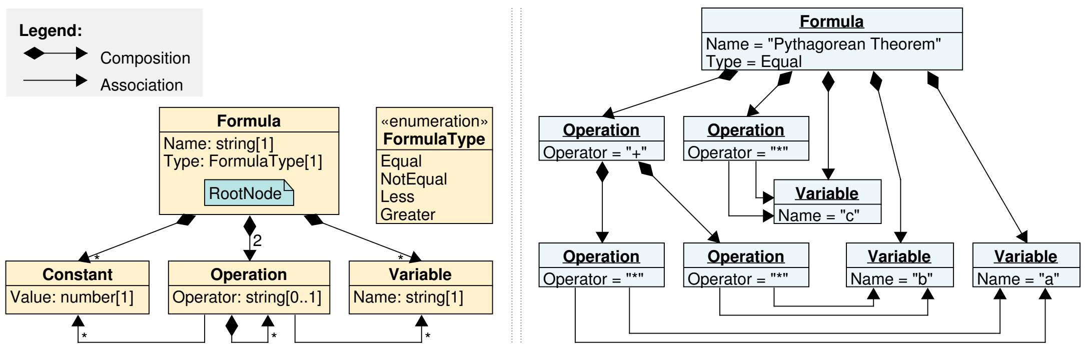


Figure 2.7: Example of a *Formula* metamodel (left) and a specific *Formula* model (right).

is itself a model of an underlying model; in a sense, it is "beyond" this described model, hence the name metamodel. A metamodel defines a set of models (i.e. a design space) by establishing certain constraints. The models in this set are said to be instances of the metamodel or conform to the metamodel. A model, in turn, characterizes a system at some level of abstraction.

Figure 2.7 illustrates a metamodel of simple mathematical formulas described in Unified Modeling Language (UML) as well as a concrete instance of the metamodel. The metamodel contains a root node which describes the attributes of a *Formula*, a *Name* and a *Type* which denotes whether the *Formula* is an equation or an inequality of some sort. Moreover, a *Formula* has multiple constituents. On the one hand, it can have an arbitrary number of *Constants* with assigned *Values* and *Variables* with assigned *Name*. In addition, it is also composed of exactly two *Operations* which basically describe the left- and right-hand sides of the formula. Each *Operation* can[1] have an *Operator* and an arbitrary number of associated *Constants* and *Variables*. Also an *Operation* can have an arbitrary number of "child"-*Operations*. As an example, a *Formula* instance representing the Pythagorean theorem $a^2 + b^2 = c^2$ is visualized.

It is apparent that the exemplary model fulfills all restrictions of the metamodel in that it has a valid structure and valid attributes. Moreover, it is certainly a valid model of the Pythagorean theorem. However, it is also noteworthy that the metamodel itself allows the construction of models that no mathematician would ever accept as valid formulas. For example, it is possible to describe an operation without any operands, a unary operation with more than one operand or a non-unary operation with just a single operand. In a practical scenario, this would have to be forbidden by either introducing proper constraints in the metamodel (e.g. via Object Constraint language (OCL) for UML 2 and above) or implementing proper constraint checking in tools that take instances of the metamodel as input.

[1]It does not *have* to have an *Operator* since it would be acceptable for it to just be a *Variable* or a *Constant*.

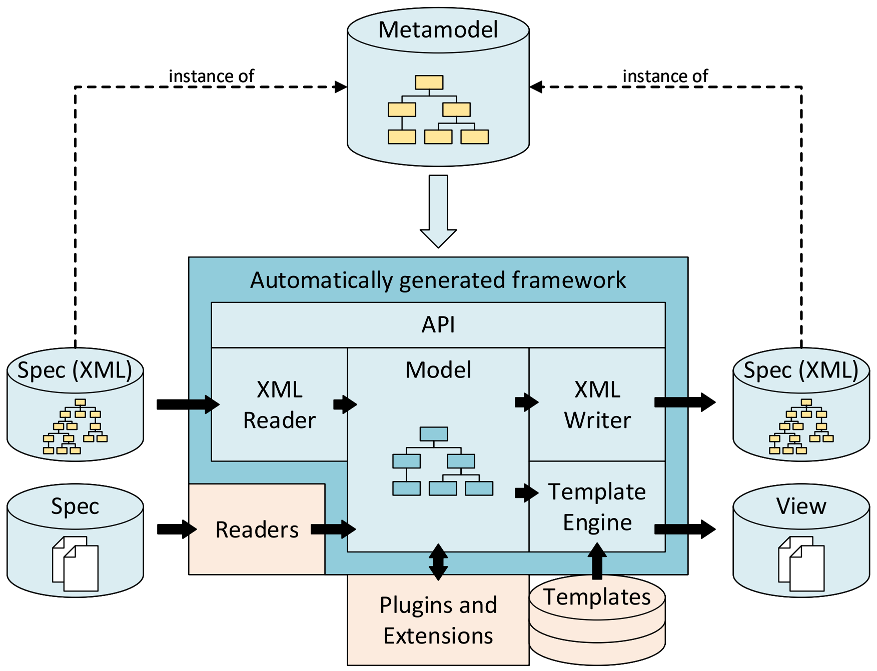


Figure 2.8: Visualization of the Metagen framework.

The goal of metamodeling is to identify and capture application-specific requirements in an abstract metamodel. This results in a formalized description of the models of interest which can be used to guide automation.

2.2.2 Metamodel-based Code Generation

Metagen is a metamodel-based code generation framework that was developed at Infineon. Given a metamodel described in UML by the user, a Python-based framework is generated (see Figure 2.8). This framework consists of the following parts:

- An Application Programming Interface (API) for construction of and interaction with models as defined by the metamodel provided by the user including respective documentation.
- A set of readers and writers for de-/serialization of models from/to predefined formats (e.g. XML).
- A Graphical User Interface (GUI) for the purpose of creation, inspection and modification of model instances.

When a model is loaded into the framework through a reader, a Python representation of the model is constructed. In addition to the automatically generated readers, the user also has the option to provide their own readers. A custom reader implementation may be useful to map data in an arbitrary input format to the Python-based model representation using the generated API. Essentially, the input data is interpreted as a specification for the resulting model.

After a model has been loaded into the framework, extensions provided by the user can be employed for model modifications. This may not always be useful or required but drastically increases the flexibility of the entire approach.

Finally, given a model representation, a default or custom writer can be invoked. The most relevant use case is the employment of custom write routines by providing so-called templates to a powerful template engine called Mako. Templates are basically a skeleton of the output format including placeholders and Python code. By combining a template with a model, a model-specific view is generated by the template engine. Multiple templates can be provided to generate multiple views from a single model (e.g. code, documentation, ...).

Simply put, Metagen is used to drastically speed up the development of code generators accepting models conforming to a metamodel. An interesting aspect of this flow is that, at a higher level, Metagen itself is a code generator accepting metamodels conforming to a meta-metamodel. This is in line with the concept of Meta-Object Facility (MOF) as defined by Object Management Group (OMG).

Metagen is used very extensively and has a high potential for reducing Non-Recurring Engineering (NRE) costs via automation and reuse. Specifically, productivity gains of up to 95% for particular tasks and effort reductions of up to 70% have been observed [12, 13].

While the advantages of the flow are evident as soon as a generator is usable, there is still one problem to be addressed during the development. When faced with the task of generating a specific type of view from a specific type of specification, it is often the case that there is a significant difference between source and target structure. Essentially, the metamodel has to lie somewhere in between the interval of input and output structure. If the metamodel is close to the specification, it is trivial to develop a custom reader, however, the development of the templates becomes a very complex task. This is especially apparent when considering the generation of Hardware Description Language (HDL) code from a specification of a digital filter, let alone more complex and irregular structures. On the other hand, when the metamodel is close to the view, developing the respective template is simple but transforming a specification into the respective model becomes increasingly tedious. Moreover, this choice would defeat the purpose of being able to transform a specification into many different views. Therefore, a flow for e.g. HDL generation needs to allow the metamodel to be close to the specification while also providing a sufficiently high-level API.

2.2.3 Model Driven Architecture

The concept of MDA has been adopted by the OMG for the purposes of reducing complexity and cost and increasing interoperability and reuse about two decades ago [15,

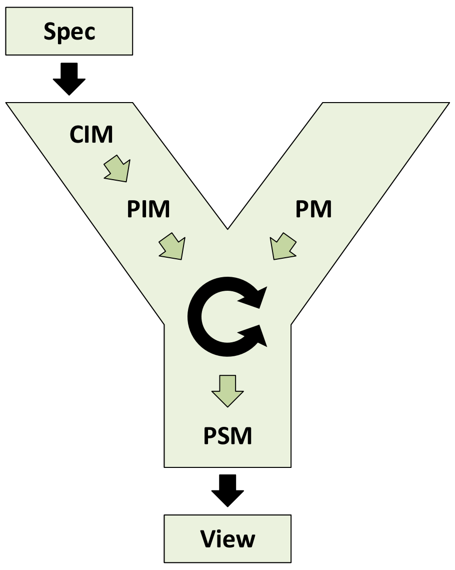


Figure 2.9: Y-chart of the MDA approach [14].

16, 17]. In principle, MDA makes use of models as formal descriptions of systems at different levels of abstraction. These levels of abstraction are often called viewpoints. The key idea is that well-defined model transformations between adjacent viewpoints (i.e. relatively close levels of abstraction) are more straightforward to develop, reusable and allow for clearer semantics at higher levels of abstraction.

MDA defines three models whose relation can be seen in Figure 2.9:

- The Computation Independent Model (CIM) captures the requirements of a system without going into any detail regarding implementation. It is very close to the specification and serves as a high-level representation of the system. Essentially, it delivers an answer to the question: "What is our goal?".
- The Platform Independent Model (PIM) is more detailed in that it describes the required behavior of the system. Yet, it is still somewhat coarse in its level of abstraction in that it is not bound to a specific platform (i.e. a certain framework or environment). It basically answers the question: "With what strategy can we achieve our goal?".
- The Platform Specific Model (PSM) is a more refined description of the system's behavior and takes highly platform-specific details into account. It can be seen as an answer regarding the question: "How can we implement our strategy under given circumstances?"

As the Y-chart depicts, a more abstract model of a system is transformed in a step-wise fashion to a less abstract model of the system. From a higher point of view, a specification is translated to a respective view via a series of model-to-model transformations. This multi-level process allows bridging the gap of enormous differences in abstraction while sticking to a model (CIM) that is close to the specification.

In the step from PIM to PSM, a Platform Model (PM) is required which describes the capabilities and restrictions of the target platform. Depending on the use case, the platform may be for example a programming language (code generator) or an instruction set (compiler).

It is noteworthy that multiple classes of CIMs (i.e. conforming to different metamodels) can be mapped to the same class of PIMs, provided that the PIM's metamodel is powerful enough. In other words, the platform-independent metamodel defines what kinds of computation-independent metamodels are supported.

2.2.4 RTL Code Generation

MetaRTL is an adaptation of the fundamental principles of MDA to the Metagen framework with the goal of allowing efficient development of RTL generators [18, 19, 20, 21]. This is achieved by introducing three types of models analogous to the CIM, PIM and PSM:

- The Model-of-Things (MoT) defines components (things) of the design, their attributes and how they are related. The defining metamodel is close to the specification (and hence use-case-specific) so mapping informal requirements to an MoT is very straightforward. A simple example of an MoT could be the definition of a digital Finite Impulse Response (FIR) filter in terms of its coefficients.
- The Model-of-Design (MoD) defines the design in terms of an RTL schematic. Its metamodel offers a rich set of digital primitives to support the construction of a wide range of digital circuits. In the exemplary case of a digital FIR filter, the MoD would be a chain of registers whose outputs are connected to multipliers (according to the coefficients) and then summed up by an adder.
- The Model-of-View (MoV) is close to the desired view which is HDL code. There is a separate metamodel for each target backend (e.g. VHDL, Verilog, ...).

Figure 2.10 visualizes how MoTs are passed through several model transformations to generate a low-level MoV which can then be unparsed into the desired HDL code.

This structure has significant advantages over a Metagen-only approach when it comes to implementing a generator for a specific class of circuits. Figure 2.11 highlights the responsibilities of the developer. First, an appropriate metamodel is defined and the respective framework is generated using Metagen. Afterwards, the transformation between a specific MoT and its respective MoD has to be developed. For this purpose, a template called Template-of-Design (ToD) is written in Python. To accomplish this, the developer

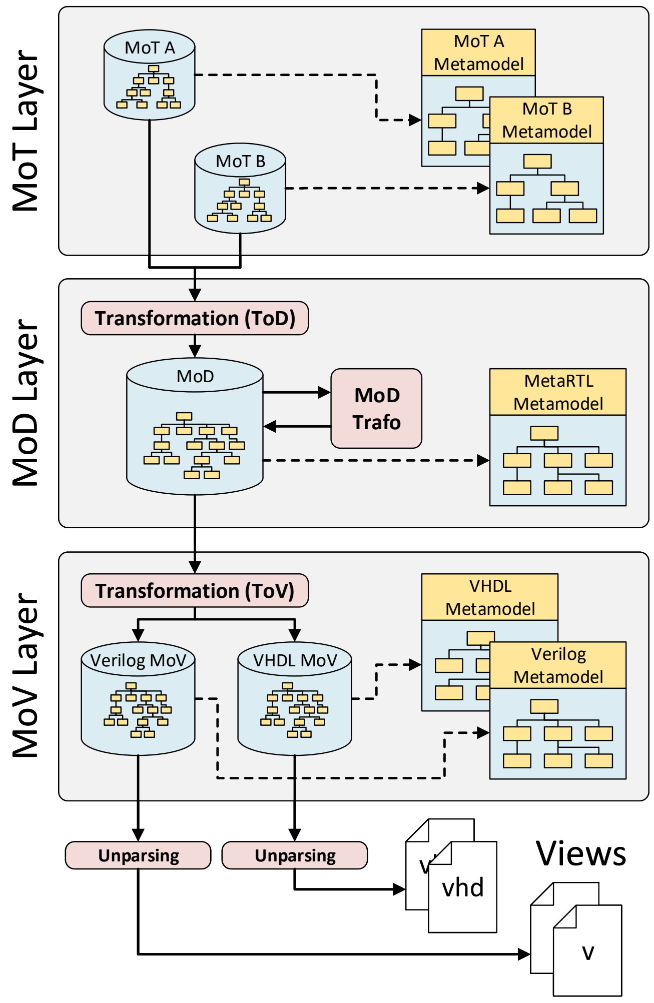


Figure 2.10: An MDA-oriented approach to RTL generation called MetaRTL.

can use the MoD-specific API (generated by Metagen) as well as domain-specific high-level extensions to ease the process of constructing the digital design.

The rest is handled by the MetaRTL core and the HDL-specific backends. Notably, the developer does not require knowledge of the MoD to MoV transformation via the Template-of-View (ToV) much like implementing compiler optimizations at the level of abstract syntax trees does not require knowledge of the microarchitecture backends. Overall, the reduction in code required by the MetaRTL flow compared to a Metagen-only flow is a factor of 5x to 10x[21].

The power, flexibility and reusability of a generator grow with the freedom of the defining metamodel. However, there are certain concepts in digital design that are orthogonal to the specification of a circuit and may be applied frequently. For example, regardless of the functionality of a circuit, there is always the option of replacing registers with

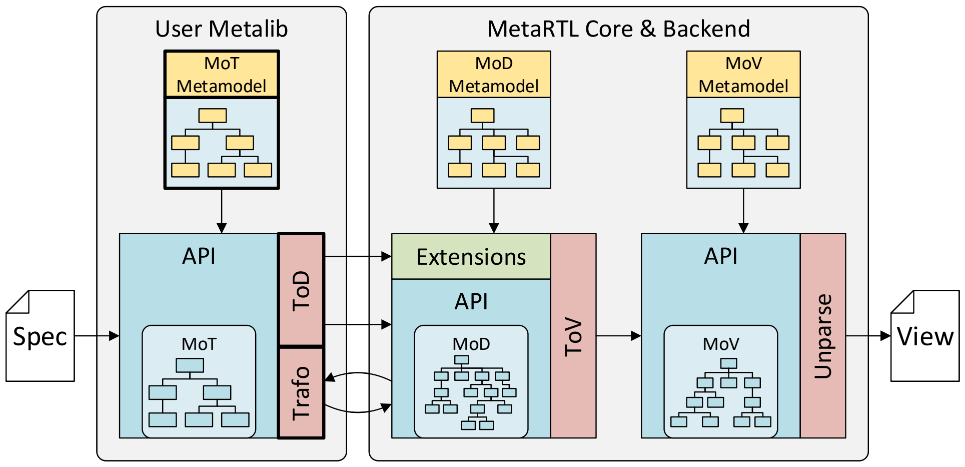


Figure 2.11: Display of the boundary between user-defined code and the MetaRTL environment.

hardened registers for functional safety. Of course, this only makes sense if the use case really requires it. Since implementing this feature in ToDs would quickly lead to code duplication and incidental complexity, this can be solved with MoD transformations. Essentially, the extended API provides features to apply transformations to an already existing MoD regardless of how it was originally constructed. Therefore, given an arbitrary MoD and a set of transformations, an altered variant of the MoD can be produced in a transparent and flexible fashion.

2.2.5 Firmware Generation

With the help of MetaRTL, a rich variety of similar hardware designs can be generated. However, some hardware components are at the hardware/software boundary and therefore require an appropriate HAL and driver. It follows that variations in the generated design are likely to cause variations in the required software stack.

For this reason, an MDA-inspired flow for firmware generation called MetaFirm is provided to complement MetaRTL. MetaFirm is relatively similar to MetaRTL in that it is also a MDA-based three-layer flow. The idea is to use the same MoT (i.e. a single source with abstract concepts concerning hardware and software) as a basis for both, MetaRTL and MetaFirm.

An essential part connecting MetaRTL and MetaFirm is the Control Status Configuration (CSC) metamodel whose models serve as hardware/software interfaces. Every MoT at the hardware/software boundary also contains a CSC MoT. On the hardware side, the CSC MoT is transformed into a register interface that can be connected to a data bus. The software side is the HAL of said register interface in the C programming language.

Since the same MoT is used by MetaFirm and MetaRTL, the first layer in the MDA-inspired structure is essentially identical. The second model in MetaFirm is called

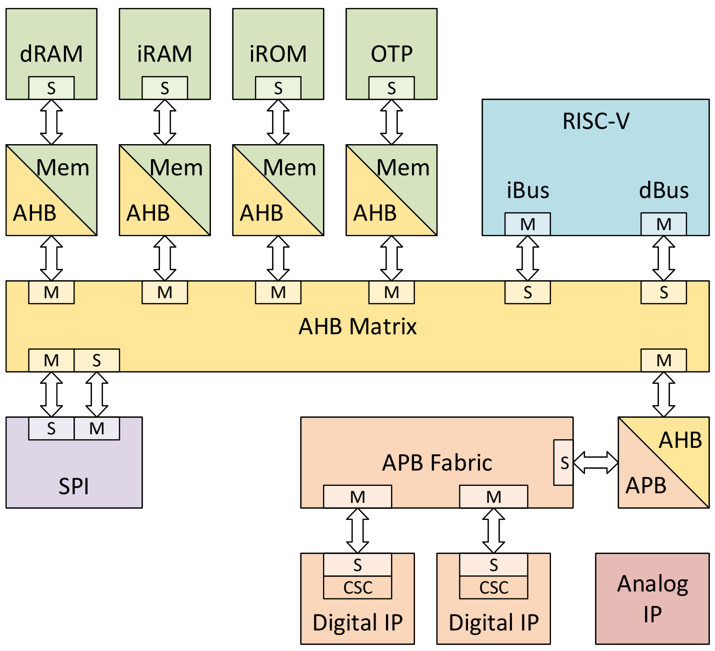


Figure 2.12: Overview of the RiVal 2 test chip.

Model-of-Firmware (MoF) and it is created by transforming the MoT via a Template-of-Firmware (ToF). It captures programming concepts like functions, loops, conditions and variables and can make use of the HAL. The model at the third layer is again called MoV but is concerned with representing the firmware in the C programming language and should not be confused with its MetaRTL counterpart. Unparsing yields a view in the form of C source and header files.

2.3 SoC Architecture

Infineon is developing an SoC called RiVal 2 on the basis of a RISC-V [22] implementation based on MetaRTL. Besides the CPU, several other parts of the architecture are generated via MetaRTL like the multi-layer Advanced High-Performance Bus (AHB) matrix. A Serial Peripheral Interface (SPI) module makes it possible to program the SoC. Figure 2.12 gives a high-level overview of the architecture.

As already outlined in Section 2.2.5, peripherals at the hardware/software boundary require a register interface. This can be generated via the MetaRTL-based CSC metalib. It offers a simple data bus interface that is very easy to bridge to other peripheral interfaces. In the case of the RiVal 2, an Advanced Peripheral Bus (APB) bridge allows the peripheral to be connected to the APB fabric of the SoC. This APB fabric then connects to the bus matrix via an AHB-to-APB bridge. The data widths of the AHB and APB buses are 32 bit and 16 bit respectively.

Any functional access to the digital peripherals can be handled via the bus. This is in contrast to any analog peripherals which do not have a bus interface. Of course, both digital and analog IPs still need to be scannable for EOL testing. Additionally, analog IPs must be trimmed at least once after the manufacturing process. Moreover, configurability of the analog peripherals based on the found trim values is necessary for operation in the field.

CHAPTER 3

Related Work

The subject of this thesis is related to the possibilities that scan architectures open up during EOL testing as well as in the field with respect to BIST and analog IP trimming and configuration. Another corner stone is the addressing of design productivity by means of harmonizing these aspects with the concept of automated firmware generation. This chapter provides an overview of state-of-the-art proposals to tackle self-testing, trimming and configuration via scan chains and RSNs as well as the possibilities that EDA tools already offer.

3.1 Self-Testing via Scan Chains

Using (and reusing) scan chains for BISTs is attractive because the hardware interface is very simplistic and lightweight and the introduction of scan chains may be a requirement for EOL testing anyways. Therefore, there are multiple proposals targeting the opportunities especially in combination with the recent IJTAG standard. Moreover, processor-driven self-test opens up a lot of flexibility on its own.

In larger systems, one possibility is the employment of a dedicated service processor acting as a JTAG master [23]. This way, BISTs can be performed in the field via the use of software-based test routines. While this opens many possibilities, it is not exactly a resource-saving approach and therefore not feasible for integration on an SoC.

A more lightweight approach is described in [24] where blocks of memory are connected as alternative drivers of the TAP controller. This memory is used to store the signal transitions required for the application of test vectors. The TAP interface is multiplexed to allow switching between off-chip and on-chip operation. In the paper, an IJTAG network is used to allow access to sensors for temperature monitoring but the system can be used for virtually any monitoring or test-related task. It is, however, worth noting that the design is relatively static with respect to test data and test flow since they are essentially programmed into the memory.

An example of an EDA tool that offers the compilation of IJTAG networks for the purpose of testing is Tessent IJTAG [25]. It supports inclusion of Tessent MemoryBIST [26] and LogicBIST [27] components as well as any other IJTAG-compatible IPs. With the help of Tessent MissionMode [28], a so-called In-System Controller can be generated which enables on-chip operation of the RSN. MissionMode offers memory-based and CPU-based implementations. The former option is similar to the approach described in [24] in that test data is stored in memory and access to the TAP controller is multiplexed. Another possibility is to embed a controller that allows the CPU to operate on the RSN by mapping CPU signals to the TAP controller. As of June 2021 [29], another CPU-driven option is available where the detour via the TAP controller is bypassed and an APB-slave device is employed to map memory transactions to scan operations. The software required for the CPU-targeted can be generated from Verilog test benches which only supports limited functionality. Alternatively, it can also be written by hand to enable more dynamic control flow but this quickly becomes a tedious task especially due to arbitrary bit widths and the dynamic nature of RSNs.

On a sidenote, the interest in using alternative interfaces (like Inter-Integrated Circuit (I^2C), SPI or in the above example APB) to access RSNs has generally been picking up in the last years. Consequently, an entirely new standard, currently known as P1687.1 [30], is expected to evolve from the ongoing research. A strong focus of this work is enabling highly dynamic accesses to the IJTAG network for the use in BISTs where the use of static predefined test vectors is not sufficient.

In [31, 32] an on-chip RSN controller is presented that is supported by a cross compiler flow for efficient development and retargeting of test procedures in high-level programming languages. This allows software developers to use PDL-derived operations on parts of the IJTAG network without having to care about the network's hierarchy, TDR definitions or the required scan operations. The proposal also features a dynamic on-line retargeting engine which provides a great deal of flexibility with respect to control flow in access routines and is especially valuable for access in interrupt service routines.

Making use of the structural information of an RSN and retargetable scan operations during the test software development is certainly an important aspect in terms of usability and productivity. The PDL and ICL are the IEEE 1687 conforming methods of capturing this information. For this purpose, the software layer, especially PDL and ICL parsers, provided in the open IEEE 1687 ecosystem in [33] can be extremely valuable.

IJTAG networks have also been used in [34] for the purpose of efficient fault monitoring. Specifically, a slightly modified SIB is introduced which is capable of opening its respective subnetwork based on some events. This allows the RSN to reconfigure itself and provide scan-chain-based fault monitoring and fault localization.

3.2 Trimming and Configuration via Scan Chains

The usual way of performing digital trimming would be to use ATE for the application of digital trim stimulus and also measurement of the produced analog signal. In [35, 36] the concept of self-trimming is outlined where the ATE is instead concerned with the provision of an analog reference signal. The analog signal produced by the IP is then compared against this reference using an on-chip Analog Digital Converter (ADC) or analog comparator. This (digital) result can then be used to systematically adjust the trim value which can effectively be carried out on-chip by a respective state machine.

As outline in [8], going with an ADC instead of just a comparator does allow to implement a so called full BIST approach instead of just a partial BIST (i.e. no reference from ATE needed) but also increases the required area. Moreover, this freedom comes at the cost of initial ADC trimming requirements. The authors describe a BIST trim controller which is capable of adaptively switching between linear, binary and hybrid search including early response analysis.

In [7] an IJTAG based approach is introduced to integrate the mentioned concepts within a digital core which can be operated transparently via I^{2}C. Specifically, for each analog block, a digital island consisting of the required test and trim logic as well as a TDR for the operation thereof is introduced. The islands are then combined into an RSN and augmented with a TAP controller as well as an I^{2}C-to-TAP mapper. Their self-trim logic and configuration registers can therefore be accessed via both interfaces.

Interestingly, the authors chose to use the TDRs as an addressing mechanism to operate on the digital islands' register banks instead of using it as a direct access mechanism. While this promotes transparent operation via I^{2}C and simplifies the software aspect for programmers by a great deal, it also means that the flexible and in some sense parallel nature of RSNs can not be used to its full potential. This also means that the maximum number of analog blocks and their registers is fixed which again results in a relatively limited and rigid approach.

In general, configuration is not any different from testing in that it greatly benefits from a suitable software abstraction layer and reuse through retargeting. Therefore, software ecosystems and ICL- and PDL-specific parsers like in [33] are also relevant to this task.

CHAPTER 4

Design Implementation

The aim of this chapter is to give an overview of the hardware and firmware generators implemented in the course of this thesis. To that end, it provides insights into how the requirements outlined in Section 1.2 affect design decisions with respect to the overall design structure. Based on these considerations, the metamodel that is employed in the generation flow is presented. This chapter also goes into detail on how the generated hardware architecture is structured, what possibilities the firmware generation flow offers and how specification parameters and design transformations can be employed to influence code generation. In the end, it should become clear how various aspects of the implemented flow empower reuse, flexibility and extensibility on a hardware level and further support software development for self-test, self-trim and configuration routines.

4.1 Design Requirements and Structure

One of the basic requirements identified in Section 1.2 is compatibility with existing ATE. For this reason, an IEEE 1149.1 conforming *JTAG-Module* is required including a TAP controller and the TDRs and instructions mandated by the standard. The design-specific aspects like the content of the *IDCODE* register or instruction indices shall be configurable. Moreover, it is essential to offer straightforward extensions with other standardized instructions as well as custom instructions. Ideally, both can be achieved by simply adapting the specification from which the hardware is generated. This would allow putting together varying feature sets based on the actual product requirements with minimal effort. Of course, some instructions (standardized or custom) may require very specific logic even within the *JTAG-Module* which means that changes to the code generator may be required from time to time which is fine as long as it enhances the generator in a modular and reusable way. Section 4.3.1 gives an overview of how these requirements can be met with the presented design.

Arguably, the most important aspect of the entire design is the interaction of the CPU with scan chains which enables self-testing, self-trimming and runtime configuration. For this purpose, a module that is capable of performing scan operations on TDRs via the serialization of input data and deserialization of output data is required. As mentioned in Section 2.3, the RiVal 2 SoC offers an AHB matrix with a data width of 32 bits as well as an APB bus with a data width of 16 bits. Since serialization is slower than the memory transactions, the high performance of the AHB interface is certainly not necessary. Therefore it makes sense to opt for an APB-driven CPU-to-TDR interface which will be referred to as *APB-Module*. This thesis will focus on the specific choice of an APB interface with a data width of 16 bits.

From a functional point of view, the *APB-Module* must support the selection of instructions and TDR registers similar to the *JTAG-Module*. Since there may be TDRs that are accessible via the *JTAG-Module* but not via the *APB-Module* and concurrent operation on disjunct TDRs may be necessary in the future (e.g. a JTAG-driven OCDS), the *APB-Module* must have its own IR. Moreover, one must be able to trigger *capture* and *update* operations as well as a series of *shift* operations. Since the APB bus of the RiVal 2 has a data width of 16 bits, in theory, up to 16 shifts can be carried out per transaction. However, TDRs can have arbitrary lengths. Therefore, it must also be possible to shift only a fraction of the 16 data bits. While *shift* operations supply data to TDRs via the serial input, they also produce data at the serial output. Consequently, the *APB-Module* must deserialize this data and provide it to the CPU as necessary.

All in all, the register interface of the *APB-Module* must map sequences of memory transactions to scan operations according to the discussed requirements in a transparent way. Section 4.3.2 goes into detail on how this is achieved in the presented design.

Since strong support for the reuse of IPs and related test routines is another key requirement, it is natural to employ an IJTAG network for the scan chains which shall be accessible via the *JTAG-Module* and the *APB-Module*. Most importantly, it shall be possible to define IP-specific and reusable subnetworks on a per-IP basis. Within a subnetwork, the common types of standardized TDR cells and custom configuration cells shall be supported. The generation of an *IJTAG-Module* which combines these subnetworks into a corresponding RSN shall be driven by the system's specification. For maximum flexibility, reconfigurability of the RSN shall be possible at the network- and subnetwork-level. Therefore, scan multiplexers (e.g. SIBs) must be employed within the subnetworks and the *IJTAG-Module*. To enable operation via the *JTAG-Module* or the *APB-Module*, the *IJTAG-Module* must define a set of instructions and TDRs. Signals towards the *IJTAG-Module* must be multiplexed depending on the mode of operation (off-chip vs on-chip). The details on how the related requirements are achieved are presented in Section 4.3.3 and Section 4.3.4.

Finally, since TDRs can have arbitrary widths and the state of the RSN affects the total scan chain length, operating the *IJTAG-Module* via the *APB-Module* is inherently complex and can easily lead to convoluted code. This negatively impacts productivity and increases the probability of bugs. Moreover, this is not in line with the idea of fostering

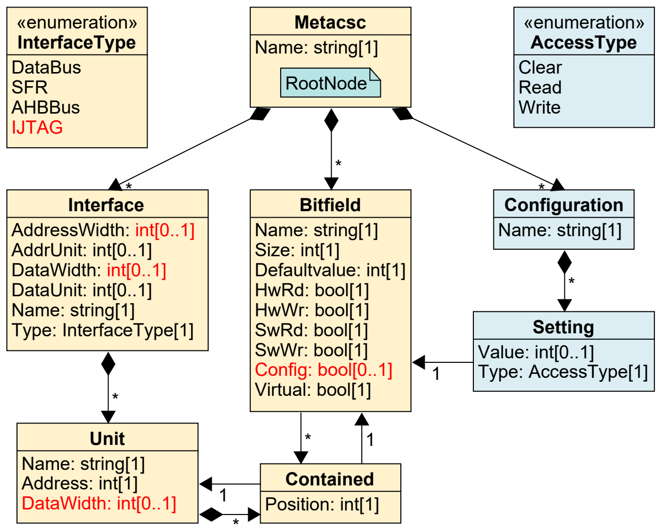


Figure 4.1: UML description of the extended CSC metamodel. Hardware-related aspects are displayed in yellow while firmware-related aspects are displayed in blue. All changes are highlighted in red.

code reuse since the source code would have to change whenever the IJTAG network structure defined in the system specification or a subnetwork specification changes. Therefore, the generation of source code from high-level descriptions of RSN interactions is necessary to support software developers. The primary purpose is to handle TDR interactions for testing, trimming and configuration in the generated driver while the developers can focus on the algorithmic aspects of the software. Ideally, the high-level description of these RSN procedures shall be close to PDL so that a combination with existing workflows and tools is possible later on. Section 4.4 outlines the possibilities and inner workings of the firmware generation.

4.2 Design Metamodel

For the full specification of a design as presented in this thesis, two types of metamodels are needed. The first metamodel is an extension of the existing CSC metamodel mentioned in Section 2.2.5 which is concerned with the generation of register interfaces. Figure 4.1 illustrates the parts of the CSC metamodel relevant for this thesis. The second metamodel is the JTAG metamodel and can be seen as the main metamodel in the context of this thesis. It is depicted in Figure 4.2.

The CSC metamodel serves two purposes in the presented design. On the one hand, the *APB-Module* contains an instance of the CSC since it must be controlled via a register interface. On the other hand, the CSC metalib was adapted to also handle the generation of the IJTAG subnetworks.

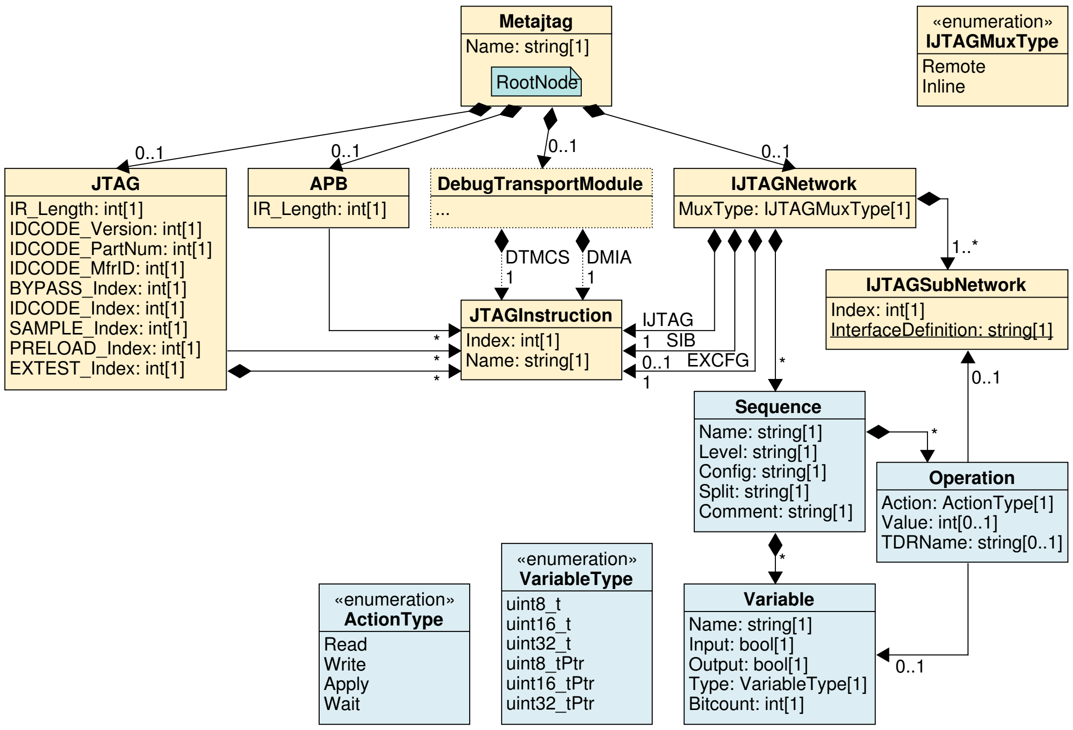


Figure 4.2: UML description of the extended CSC metamodel. Hardware-related aspects are displayed in yellow while firmware-related aspects are displayed in blue. The *DebugTransportModule* represents a future addition. The underlined attribute *InterfaceDefinition* is an external reference to a CSC *Interface* (i.e. a JTAG MoT can reference multiple CSC MoTs).

A CSC instance contains a set of *Bitfields*, which can be seen as the actual physical registers. They have a *Name* and *Size* and can be readable and writable from the hardware and the software side (*HwRd*, *HwWr*, *SwRd*, *SwWr*). The CSC also contains a set of *Interfaces* that can be of various *Types* like the *DataBus* which is a single-cycle bus with *data_in*, *address*, *rd_en*, *wr_en* inputs and *data_out*, *error* outputs. Moreover, *Interfaces* are assigned specific data and address widths and unit definitions. They can be used to access the *Bitfields* via their respective *Units* which are essentially logical registers with a defined *Address*. Generally, *Bitfields* and *Units* have a many-to-many relationship which is defined by the *Contained* relation and the translation from logical to physical registers is handled by the CSC circuit accordingly.

The CSC metamodel also captures some firmware-related information in the form of *Configurations* that describe sequences of operations on *Bitfields*. These can be seen as an addition to the basic HAL generated for the register interface via MetaFirm.

The new option that is required for the design presented in this thesis are interfaces of

Type IJTAG. In this case, the address and data information does not serve any purpose since the described subnetwork always has a serial input and a serial output and the JTAG control signals. Moreover, the notion of a *Unit's* address is a bit different. Essentially, a *Unit* is still equivalent to an addressable unit in that its design analogue is an SIB at a specific position in the described subnetwork. This SIB guards its related *Bitfields* which are realized as TDRs. The *HwRd* and *HwWr* flags denote whether the respective TDR has update and capture functionality while the new *Config* flag is used to state that a configuration cell (see Section 4.3.4) shall be generated instead. Since the test-, trim- and configuration-procedures for the presented design must be able to work across multiple subnetworks, the firmware-related information in the CSC metamodel is of no direct use in this case.

The JTAG metamodel describes the composition of the presented architecture as well as the system-specific details of the individual modules. Therefore, it can have a *JTAG-Module*, an *APB-Module* and an *IJTAG-Module* which are configured by the respective child objects of an MoT.

In the case of the *JTAG-Module*, the length of the IR, the contents of the *IDCODE* register and the indices of the mandatory *IDCODE*, *SAMPLE*, *PRELOAD* and *EXTEST* instructions can be configured. While the *IDCODE* register is strictly speaking not mandatory according to the JTAG standard, it is definitely considered good practice to include one and is therefore enforced by the metamodel. The *JTAG-Module* can also have additional *JTAGInstructions* which could be any optional standardized instructions or also custom instructions. Moreover, other modules may require their own *JTAGInstructions* to which the *JTAG-Module* can have a reference to model the fact that it can interact with them.

The *APB-Module* is similar to the JTAG. However, it cannot have its own *JTAGInstructions*. The only way for the APB to interact with TDRs is to have a reference to other modules' *JTAGInstructions*.

The most complex part of the metamodel is the *IJTAG-Module*. It has a set of *IJTAGSubNetworks* that correspond to CSC interfaces of *Type IJTAG*. Each of these subnetworks has a unique *Index* to define its position in the network. The module also has a set of *JTAGInstructions* for the purpose of accessing the RSN (*IJTAG* and *EXCFG*) and an optional *SIB* instruction which may be required depending on the *MuxType*. The *MuxType* is there to give control over the type of scan multiplexer that is generated between the subnetworks. While the *Inline* setting corresponds to SIBs, the *Remote* setting uses the TDR of the *SIB* instruction for the control of the scan multiplexers.

On the firmware side, the model can have a set of *Sequences*. Each *Sequence* has a set of *Variables* and *Operations*. A *Variable* has a *Name*, a *Type* that corresponds to the data type in the code and a *Bitcount* which defines the usable bits within the *Variable* to provide some form of documentation and additional safety. Moreover, the *Input* and *Output* flags denote whether a *Variable* is local, a return variable or some kind of function parameter. The *Operations* can describe *Reads* from TDRs into *Variables* and *Writes* of

Name	Width	Direction	Description
TCK	1	Input	The TCK signal
TMS	1	Input	The TMS signal
TDI	1	Input	The TDI signal
TDO	1	Output	The TDO signal

Table 4.1: Overview of the *JTAG-Interface* from the perspective of the SoC.

Name	Width	Master	Slave	Description
Clock	1	Output	Input	The clock signal
Capture	1	Output	Input	The capture enable signal
Shift	1	Output	Input	The shift enable signal
Update	1	Output	Input	The update enable signal
SI	1	Output	Input	The serial input signal

Table 4.2: Overview of the *JTAG-TAP-Interface* which provides shared control and data signals between the TAP controller and the TDRs. The control signals are not masked yet.

Values or *Variables* to TDRs. The respective TDR is identified by the reference to the *IJTAGSubNetwork* and the *Name* of the *Bitfields*. Similar to PDL, the *Apply* is required to define when a scan operation is complete and shall be executed (i.e. all formerly issued *Reads* and *Writes* take effect with the *Apply*). For time-sensitive procedures, *Wait* can be used to wait at least the specified number of cycles.

Each *Sequence* has a *Name* and a *Comment* which are used for code generation. Moreover, it has a *Level* that can be used to define driver feature subsets. The *Config* flag is required to specify whether the *Sequence* targets normal TDRs or configuration cells. The latter needs to be handled slightly differently in that an additional update with the *EXCFG* instruction is required. In this case, the *Split* flag defines whether the two configuration phases should be split into two functions in the generated driver.

4.3 Hardware Generation

Instances of the metamodel presented in Section 4.2 serve as a specification for the implemented hardware generators. Therefore, they directly affect the features of the generated hardware description. This section goes into detail on the capabilities of the individual modules, how they are generated and how they are combined to form the presented design.

4.3.1 JTAG-Module

The generator of the *JTAG-Module* expects an instance of the *JTAG* child object of the JTAG metamodel illustrated in Figure 4.2. It instantiates a finite state machine in

Name	Width	Master	Slave	Description
Idle	1	Output	Input	Active in the *Idle* state
Reset	1	Output	Input	Active in the *Test Logic Reset* state
Pause	1	Output	Input	Active in the *Pause DR* state

Table 4.3: Overview of the *JTAG-TAP-EXT-Interface* which provides extension signals from the TAP controller.

Name	Width	Master	Slave	Description
Select	1	Output	Input	The TDRs select signal
SO	1	Input	Output	The serial output signal

Table 4.4: Overview of the *JTAG-TDR-Interface* which provides the TDR-specific control and data signals. The *Select* signal is required to mask the control signals of the *JTAG-TAP-Interface*.

the form of a MetaRTL *FSM* primitive, which models the TAP controller depicted in Figure 2.2. Moreover, it creates shift registers for the IR as well as for the *IDCODE* and *BYPASS* instructions. These are described as MetaRTL *Register* primitives with *Mux* primitives at their input and *SLICE* and *CONCAT* primitives to describe the shift operations. As the IR requires consistent updates, a separate update register is placed after the shift register. The clock sensitivity of all generated registers is in line with the IEEE 1149.1 standard and also the TAP signals generated by the state machine are delayed accordingly.

In contrast to the *BYPASS* and *IDCODE* TDRs, the BSC and any custom instructions are external scan chains. They must therefore be connected to the *JTAG-Module* via appropriate interfaces. For this purpose, a shared *JTAG-TAP-Interface* (see Table 4.2) and per-TDR *JTAG-TDR-Interfaces*(see Table 4.4) are provided.

Since the BSC is a special TDR in that it must be selected by multiple standardized instructions, the generator has an internal record of the implemented instructions which target the BSC. The *Select* signals are merged and only one *JTAG-TDR-Interface* is provided for the BSC. However, the individual *Select* signals are also provided as sideband signals. Another specialty of the BSC is that a *Mode* signal, which is required to control the multiplexers of boundary scan cells to switch between normal operation and test mode, is also provided as a sideband signal.

Apart from the peculiarities of the standardized instructions and registers, the selection logic and interfaces of any extension instructions are embedded fully automatically. This drastically eases the integration of future extensions without the need to change the generator of the *JTAG-Module*. The generator also checks the instructions' indices against any IEEE 1149.1 requirements or suggestions to warn the user about violations of the standard or questionable design choices.

Name	Address	Width	Readable	Writable	Description
Instruction	0	*IR_Length*	yes	yes	The internal IR
Action	1	2	no	yes	Trigger capture/update
WriteF	2	16	no	yes	Full 16 bit shift
WriteP	3	16	no	yes	Partial <16 bit shift
Read	4	16	yes	no	Read output data
Control	5	0	yes	yes	Reserved for future
Status	6	1	yes	no	Poll state of the module

Table 4.5: Overview of the *APB-Module's* register interface.

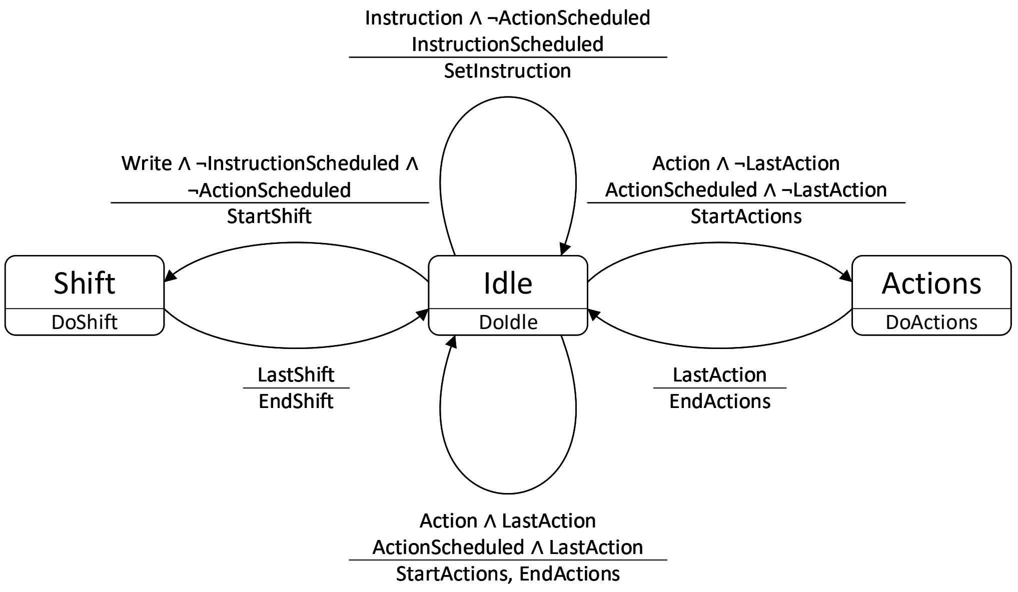


Figure 4.3: State machine of the APB module. Transitions are labelled with their input conditions and their outputs. If none of the conditions of incident transitions are met, the state machine stays in its current state.

4.3.2 APB-Module

While the *APB-Module's* metamodel object is very simple, the respective hardware implementation is a bit more complex. As already outlined in Section 4.1, the *APB-Module* must expose quite a bit of functionality via a register interface. The register interface (see Table 4.5) consists of seven addresses that provide controllability and observability of scan chains. Similarly to the *JTAG-Module*, the *APB-Module* provides a single shared *JTAG-TAP-Interface* as well as one *JTAG-TDR-Interface* per implemented instruction.

As the *APB-Module* must serialize and deserialize data and enforce a strict sequence while mostly operating concurrently to the CPU, it contains a state machine as described in Figure 4.3. The transitions' input signals are based on transactions via the register interface (*Write*, *Instruction*, *Action*), the data serialization circuit (*LastShift*), the instruction circuit (*InstructionScheduled*) and the action circuit (*LastAction*, *Action-Scheduled*). Transition and state output signals directly control the data serialization circuit (*StartShift*, *DoShift*, *EndShift*), the instruction circuit (*SetInstruction*) and the action circuit (*StartActions*, *DoActions*, *EndActions*).

The first register is the *Instruction* register which can be used to select a TDR in a JTAG-style. However, in contrast to JTAG the register's bits are written and read in parallel without any need for serialization or deserialization. It is implemented by the generator as two simple MetaRTL *Register* primitives whose widths depend on the definition in the MoT. If the state machine is not in its idle state, writing to the logical *Instruction* register causes the data to be cached in the secondary register and *InstructionScheduled* is asserted. The physical register is only updated when *SetInstruction* is asserted by the state machine (i.e. after any serialization or actions on the selected TDR are finished).

The *Action* register is only two bits wide and cannot be written to in the conventional sense. Instead, writing to this register causes the currently set bits to be merged with the written bits. The 0-bit (LSB) and 1-bit correspond to a scheduled capture and update operation respectively. When setting any bits in this register, *Action* is asserted. If any bits are already set, *ActionScheduled* is asserted. Moreover, if only one bit is set *LastAction* is asserted. When *StartActions* or *DoActions* is asserted by the state machine, the bits in the register are cleared from the MSB to the LSB on successive cycles while performing the assigned actions on the selected TDR (i.e. asserting the respective TDR enable signals). At first, it may seem counterintuitive that a scheduled update action is processed before a scheduled capture action since the state machine in Figure 2.2 does it the other way around. However, in the APB-driven flow, this choice is much more appropriate because it allows to schedule the end and start of back-to-back scan operations.

Shift operations can be started by writing the data to be shifted to the *WriteF* or the *WriteP* register. Data supplied during the write is cached and *Write* is asserted. The cached data is serialized from LSB to MSB to the shared *SI* on successive cycles as soon as *StartShift* or *DoShift* are asserted. During serialization of the 16th bit, *LastShift* is asserted to leave the shift state. The difference between the two registers is that the *WriteF* is used to shift the full 16 bits while the *WriteP* only performs a partial shift of the supplied data. Partial shifts require a special format to clearly encode the number of bits to be shifted which will be called first-zero-encoding. This is achieved by MSB-aligning the partial data and then padding it with a single zero and as many one bits as are necessary to reach the full 16 bits. For example, padding the data `1101101` to 16 bits would result in `1101101011111111`. Hence, when the serialization circuit encounters a *WriteP*, the only difference to a *WriteF* is that the shift enable signal must be kept low until the first zero bit (in the example the 8-bit) has been serialized. This is achieved by a register within the serialization circuit which is only set during a *WriteP* operation.

Reading the data shifted out of the selected TDR during a *WriteF* or *WriteP* operation can be accomplished by reading the *Read* register. During serialization to the *SI*, the

Name	Width	Master	Slave	Description
Clock	1	Output	Input	The clock signal
CaptureEn	1	Output	Input	The capture enable signal
ShiftEn	1	Output	Input	The shift enable signal
UpdateEn	1	Output	Input	The update enable signal
ExCfg	1	Output	Input	The configuration mode signal
ConfigEn	1	Output	Input	The configuration lock signal

Table 4.6: Overview of the *IJTAG-TAP-Interface* which provides shared control and data signals to the IJTAG subnetworks. The *ExCfg* and *ConfigEn* signals are required to drive the custom configuration cells presented in Section 4.3.4

data at the selected *SO* is shifted into an internal buffer. The content is provided via the *Read* register as soon as the serialization is complete. It is worth noting that, since the input data supplied during a *WriteP* is MSB-aligned, the same is true for the output data in the *Read* register.

Finally, the *Status* register is supplied to provide some information on the status of the *APB-Module*. It can be used to poll whether the state machine is idle or busy which returns a 0 or 1 respectively. Therefore, it provides valuable information for software flow control.

As already mentioned, the *APB-Module* is accessible via an APB bus. However, the CSC MoD that is instantiated by the generator and connected to the internal logic of the *APB-Module* does not offer an APB interface. Therefore, a simple purely combinational APB-to-CSC bridge is connected to the CSC circuit.

In some circumstances, it is not possible to accept an access to a register and the APB transaction must be stalled. For example, in order to impose a strong sequence of all TDR interactions, it must never occur that an instruction change is scheduled while actions are scheduled. Therefore, a stall occurs when writing to the *Instruction* or *Action* register when the respective other operation has already been scheduled. Moreover, writing to *WriteF* or *WriteP* while a previous serialization has not yet finished or actions are being processed results in a stall. Finally, reading from *Read* while a serialization is ongoing would result in inconsistent data and therefore also causes the transaction to be stalled. This way, it is not necessary to always poll the *Status* register until the next operation can be issued. Nevertheless, it can still make sense to use the polled information to optimize scan operations in terms of overall test time.

4.3.3 IJTAG Subnetworks

As outlined in Section 4.2, the IJTAG subnetworks are generated from CSC MoTs. The generator creates a scan chain which includes an SIB per *Unit* and a corresponding chain of *tdr* cells or configuration cells (see Figure 4.4) for each contained *Bitfield*. If there are gaps between SIB addresses or the contained TDRs, these are interpreted as reserved

Name	Width	Master	Slave	Description
Select	1	Output	Input	The TDRs select signal
SI	1	Output	Input	The serial input signal
SO	1	Input	Output	The serial output signal
Error	1	Input	Output	The configuration error signal

Table 4.7: Overview of the *IJTAG-TDR-Interface* which provides the subnetwork-specific control and data signals. The *Error* signal is driven by the error detection units in the custom configuration cells presented in Section 4.3.4

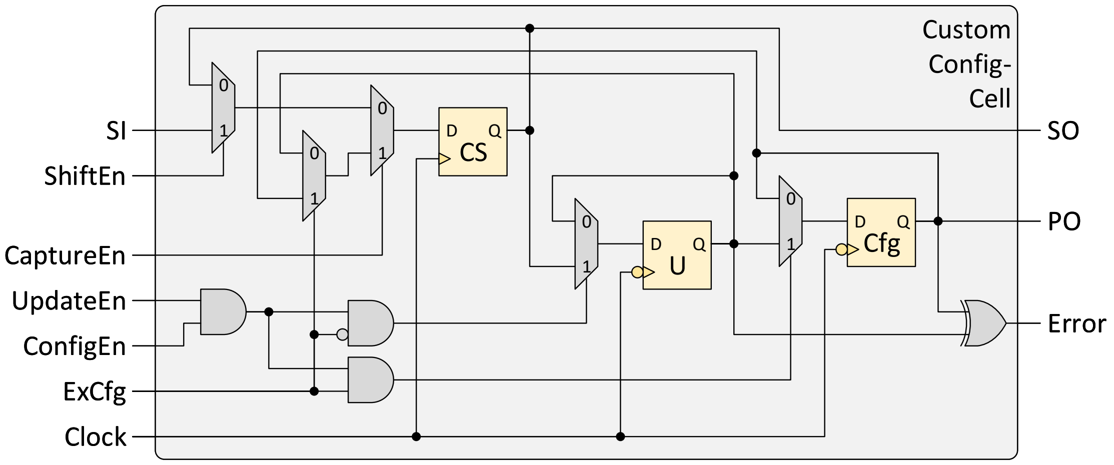


Figure 4.4: Schematic of the custom configuration cell. It is similar to a self-capturing update-TDR with an additional config register. The config register can only be updated in a special *EXCFG* instruction. Moreover, updating of the config cell can be locked and deviations between the update and config registers are detected during normal operation.

positions and shift-only cells are generated. Since the shift order as defined by the JTAG standard is always LSB to MSB, the cells are arranged in such a way that the *SI* is connected to the SIB with the highest address.

The choice of whether capture, update, capture-update or configuration cells are used depends on the combination of *HwRd* (update), *HwWr* (capture) and *Config* flags. A configuration cell is considered hardware readable but not hardware writable. Each *Bitfield* is checked for illegal configurations before generation.

A current restriction is that *Bitfields* in an *IJTAG Interface* cannot be addressed via multiple *Units* since that is not applicable within a scan chain. The generator detects this besides other illegal patterns like overlapping TDRs and informs the user accordingly.

Each subnetwork has an *IJTAG-TAP-Interface* (see Table 4.6) and an *IJTAG-TDR-Interface* (see Table 4.7) which make it possible to perform the scan operations. Additionally, for each *Bitfield* with capture or update logic, respective inputs and outputs are provided as sideband signals so they can connect to the IP or EI.

4.3.4 IJTAG-Module

The primary job of the *IJTAG-Module* is to connect all subnetworks into a single RSN. For that purpose, the generator must create a global scan multiplexer network in which the subnetworks can be embedded. In principle, there are two ways to generate the scan multiplexers based on the *MuxType*, namely remote-controlled multiplexers or SIBs. At the moment, only the *Remote* variant is implemented which means that the control bits of the scan multiplexers are located in a separate TDR whose length is equal to the number of subnetworks.

Since the module must be operable via the *JTAG-Module* and *APB-Module*, it contains a *JTAG-TAP-Interface*. Moreover, with the *Remote* style three instructions are generated which implies that three *JTAG-TDR-Interfaces* are required. In addition, a shared *IJTAG-TAP-Interface* and an *IJTAG-TDR-Interface* per subnetwork are created. Essentially, the *IJTAG-Module* can be seen as a bridge between *JTAG-* and *IJTAG-Interfaces*.

In order to construct the RSN, the generator creates a multiplexer per subnetwork. The two inputs of each multiplexer are the serial input and serial output of the respective subnetwork. Its output is the serial input to the next subnetwork. If the control bit is enabled, the *Select* signal of the subnetwork is asserted and the multiplexer chooses the respective *SO*. Otherwise, the *Select* signal is de-asserted and the *SI* signal is forwarded. Just like with the generation of subnetworks from TDRs and SIBs, the subnetwork with the highest address must be the first element in the chain in order to comply with the LSB to MSB convention.

In addition to handling the inclusion and exclusion of subnetworks, the *IJTAG-Module* also collects all error signals of the configuration cells in the connected subnetworks. The result is masked based on the *ConfigEn* sideband signal so that no errors are triggered during any configuration processes. This signal can be used by an on-chip monitor to detect bitflips in the configuration registers which could cause malfunction.

The three instructions allow operation of the *IJTAG-Module* as follows. In order to configure the scan multiplexers, the *SIB* instruction selects the TDR which contains the control bits of the multiplexers. The *IJTAG* and *EXCFG* instructions both select the RSN itself. This means that the signals of their *JTAG-TDR-Interfaces* are basically merged. The difference between the two is that the *EXCFG* also asserts the *ExCfg* signal of all configuration cells causing any occurring update actions to target the config register instead of the update register.

4.3.5 Top-Level

The top-level generator of the entire design is what instantiates all the modules (provided that the MoT demands it) by calling their respective generators. It then has to connect the generated MoDs into a single top-level MoD. Hence, it has to contain all the logic that is required to make the existing modules (and also future modules) interface with one another. For the most part, this is a matter of systematically connecting interfaces and

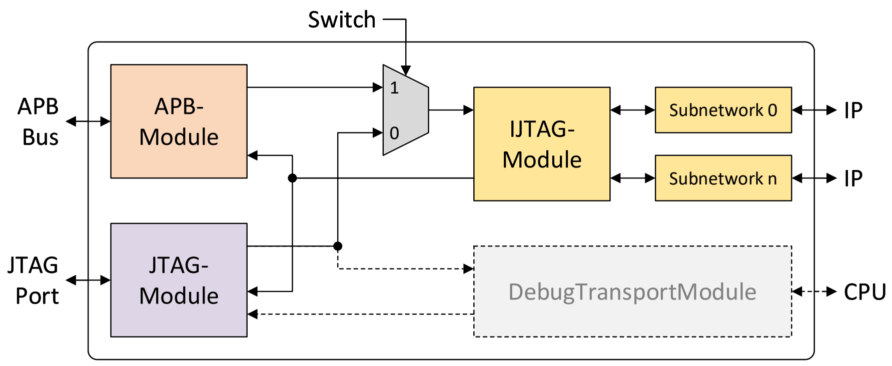


Figure 4.5: A high-level overview of how the design's top level connects the *JTAG-Module*, *APB-Module* and *IJTAG-Module*. Also illustrates how a future *DebugTransportModule* for an OCDS would integrate.

wiring sideband signals which can be automated to a large extent. However, one special case that needs to be covered is the multiplexing of the interfaces to the *IJTAG-Module*.

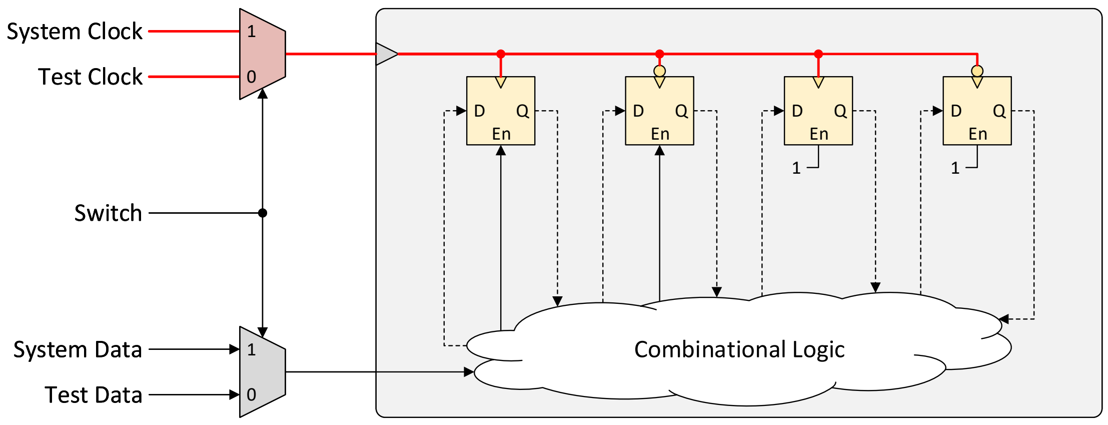
As described in Section 4.3.4, the *IJTAG-Module* has one *JTAG-TAP-Interface* and three *JTAG-TDR-Interfaces*. If the MoT specifies that the design should also include a *JTAG-Module* and an *APB-Module* and they both implement the three instructions of the *IJTAG-Module*, these interfaces have to be shared. This involves the introduction of a *Switch* signal which can be used to select between the two modules as the current controller of the *IJTAG-Module*. Basically, all inputs to the *IJTAG-Module* coming from the controllers are multiplexed via the *Switch* as illustrated in Figure 4.5. Therefore, the *Switch* signal is static in comparison to the signals related to scan chain operation and must not change while scan operations are being performed.

4.3.6 Edge Detection Transformation

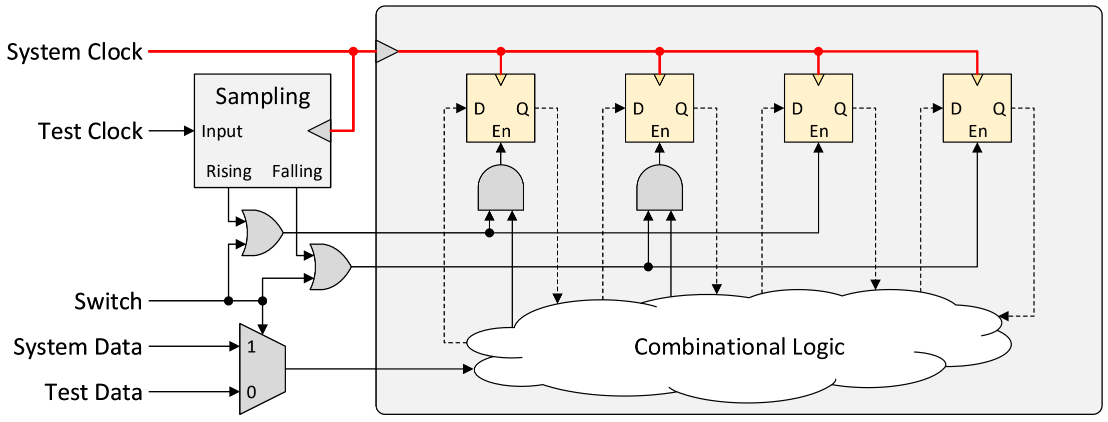
Section 4.3.5 mentions how the *JTAG-Module's* and *APB-Module's* signals related to the *IJTAG-Module* are multiplexed. For the most part, this does not cause any problems. However, while the *APB-Module* is in the SoC's clock domain, the *JTAG-Module* has its own *TCK* clock domain. This causes the introduction of a multiplexer in the clock path of the *IJTAG-Module* which is not safe for synchronous design.

Fortunately, MetaRTL makes it possible to apply transformations to the generated top-level MoD. For this reason, the presented generators come with an edge detection transformation that can be used to eliminate the problematic multiplexer. The basic idea is to rebuild the TCK-clocked parts of the design into an analogue circuit that is clocked by the SoC's clock and samples the TCK. Usually rewriting a design in this way would not be particularly hard but definitely tedious. With an automatic transformation, this can be accomplished in a matter of seconds for any possible design configuration.

The transformation consists of three main steps whose effects can be seen in Figure 4.6. First, an edge detector is introduced that samples the TCK signal and provides *Rising*



(a) System with two clock domains and unsafe multiplexer in clock path.



(b) System transformed to a single clock domain with sampling of second clock.

Figure 4.6: Example of how the edge detection transformation can modify a system to eliminate the second clock domain.

and *Falling* signals. Then, all synchronous components of the circuit (e.g. registers, state machines, ...) that were connected to the TCK (directly or indirectly) can be modified hierarchically. Based on their clock sensitivity, their enable signal is masked by either of the two signals produced by the edge detection circuit. Moreover, they are reconfigured to be sensitive only to the rising edge. Lastly, the multiplexer in the clock path is removed and every transformed component is connected to the system clock.

4.4 Firmware Generation

Since the *APB-Module* described in Section 4.3.2 has to be operated via the CPU, a corresponding HAL is required for software development. To provide higher levels of abstraction, the metamodel outlined in Section 4.2 offers the possibility to describe interactions with the IJTAG network. In this Section, the generation of two layers of drivers with increasing levels of abstraction is presented.

Function	Description
`INSTRUCTION_WRITE(uint8_t)`	Sets the *Instruction* register
`INSTRUCTION_READ()`	Returns content of the *Instruction* register
`ACTION_WRITE(uint8_t)`	Schedules capture and update actions
`WRITEF_WRITE(uint16_t)`	Initiates 16 shift operations
`WRITEP_WRITE(uint16_t)`	Initiates <16 shift operations
`READ_READ()`	Returns deserialized data of last shift operations
`CONTROL_WRITE(bool)`	Not in use yet
`CONTROL_READ()`	Not in use yet
`STATUS_READ()`	Returns whether the *APB-Module* is busy

Table 4.8: Overview of the *APB-Module's* HAL.

4.4.1 Hardware Abstraction Layer

The HAL is automatically generated from the *APB-Module's* CSC definition (see Listing 1 and Listing 2). It can be used to control the *APB-Module's* register interface described in Section 4.3.2 via the `JTAG_HAL` object. Therefore, it already offers the capabilities to perform any possible interaction with the attached *IJTAG-Module*. However, it is not specific to the *IJTAG-Module* and is also extremely low-level which means that developers would still have to take care of complying with the module's protocol. Table 4.8 gives an overview of the provided functionality.

4.4.2 Basic Driver

The basic driver is generated to provide a more straightforward interface to the *IJTAG-Module* (see Listing 3 and Listing 4). It makes use of the HAL and the information in the *IJTAGNetwork* object of the design's MoT as well as the knowledge about the *APB-Module's* protocol. The idea is to provide high-level functions for manual interaction with the attached *IJTAG-Module* which improve productivity and reduce the potential for bugs by abstracting the module's protocol (see Table 4.9). Nevertheless, the developers still have to be aware of the actual structure of the generated RSN and have to ensure proper SIB states at all times. This is also a major drawback in the sense that a slight change in the network's structure can render large chunks of source code obsolete. Moreover, input and output data must often be masked, concatenated and padded which is a tedious and error-prone process.

4.4.3 Custom Driver

In order to further boost productivity and enable reuse, it is necessary to leverage the power of MetaFirm in combination with the complete specification in the form of the design's MoT for the generation of system-specific code. As mentioned in Section 4.2, the MoT can contain high-level specifications of RSN access sequences. The task of the driver generator can be seen as a form of retargeting of these abstract sequences to the

Function	Description
`JTAG_Select_IJTAG()`	Selects the *IJTAG* instruction
`JTAG_Select_EXCFG()`	Selects the *EXCFG* instruction
`JTAG_Select_SIB()`	Selects the *SIB* instruction
`JTAG_Capture()`	Schedules a capture action
`JTAG_Update()`	Schedules an update action
`JTAG_ShiftF(uint16_t)`	Shifts exactly 16 bits of data
`JTAG_ShiftP(uint16_t, uint8_t)`	Pads data and shifts specified amount of bits
`JTAG_ShiftData(uint16_t *, uint16_t)`	Shifts an entire array of data given a bit count
`JTAG_ShiftF_R(uint16_t)`	Like `JTAG_ShiftF()` but returns read data
`JTAG_ShiftP_R(uint16_t)`	Like `JTAG_ShiftP()` but returns read data
`JTAG_ShiftData_R(uint16_t *, uint16_t *, uint16_t)`	Like `JTAG_ShiftData()` but provides read data
`JTAG_Wait()`	Waits for the *APB-Module* to become idle

Table 4.9: Overview of the *IJTAG-Module*-specific basic driver for the *APB-Module*.

system's IJTAG network definition into source code which makes use of the *APB-Module's* features.

The description of these sequences is close to PDL as outlined in the IEEE 1687.1 in that a set of *Read* and *Write* operations are provided and the entire set is applied with an *Apply* action. If multiple *Writes* target the same TDR only the last one succeeds. It is worth noting that the order in which the actions are applied to the RSN is not dependent on the order they are listed in but on the network's structure. Essentially, a single *Apply* action corresponds to a single scan operation where the input data and the configuration of the network depend on the set of TDR-accesses. In order to generate efficient code, it is important that the scan operation of an *Apply* action already pre-configures the RSN for the scan operation of the following *Apply* action.

The generator for the custom driver instantiates functions from the *Sequence* objects in the MoT. In the current state, a function can be either a test or a configuration function which is denoted by the *Config* flag. While the former can only target standardized TDRs, the latter can only target registers made up of configuration cells. This distinction is used to ease the process of code generation since the two groups of registers need to be handled slightly differently.

For configuration *Sequences*, in addition to the code for the usual scan operations, a postfix code must be generated which triggers the actual update of the config registers. This can occur within the function itself or, alternatively, a separate function can be generated for this finalizing step by setting the *Split* flag.

Apart from this, all *Sequences* are handled similarly. First, the generator creates the function signature and variables based on the assigned *Variable* objects. It partitions

	Bitcount > *Datatype*		*Bitcount* ≤ *Datatype*	
	Pointer	Non-Pointer	Pointer	Non-Pointer
Input	Paramter	Error	Parameter	Parameter
Output	Paramter	Error	Parameter	Return*
Both	Parameter	Error	Parameter	Parameter + Return*
Neither	Local Array	Local Array	Local Variable	Local Variable

Table 4.10: Overview of how the *Variables* of a *Sequence* are classified into function parameters, local variables and return parameters. Note that there can only be one return parameter per function!

them into function parameters, local variables and up to one return variable based on their *Type*, whether they are *Inputs* or *Outputs* and their *Bitcount* attribute which describes how many of the bits may actually be used. Table 4.10 gives a concise overview over this process.

Next, the sequence of *Operations* is separated into groups which all consist of *Reads* and *Writes*, an *Apply* and an optional *Wait*. All *Reads* and *Writes* must have a TDR assigned. Furthermore, *Reads* must reference a *Variable* while *Writes* must have either a *Value* or a *Variable*. Based on the referenced TDRs, the active subnetworks as well as the active SIBs can be identified for each group of *Operations*. In the current state, each TDR must be in a defined state after each *Apply* which means that each included updateable TDR must be written to. Moreover, it is not yet possible to use a *Variable* as a source of a *Write* and destination of a *Read* in the same group. Both conditions are checked by the generator and if they are violated generation is aborted with the respective warnings.

As soon as all groups are processed, a scan operation schedule must be built according to Algorithm 4.1. This is done because the actual scan operation of one group can be overlapped with the network configuration for the following group. Therefore, for each pair of adjacent groups, the respective state in the schedule is computed. First, the union of the required subnetworks for both groups is formed. Moreover, the SIBs required by the first group, the SIBs required by the second group and the *Operations* of the first group are used to build a write stream. Figuratively, this is done by assembling a sequence of *Variables* and constants according to the *Writes* and the SIBs that must be activated (second group's SIBs) and deactivated (only first group's SIBs) during the scan operation. Notably, this sequence is in the order of the respective SIBs and TDRs in the RSN. In a similar way, a read stream is computed from the *Reads* of the first group. The streams can be interpreted as a compressed sequence of data to be serialized and deserialized which must then be split into individual transactions of up to 16 bits.

At this point, the actual code generation is started as described in Algorithm 4.2. It basically consists of a loop over all the states that were computed before. First, if the currently active subnetworks are not equal to the required subnetworks, the *SIB* instruction must be selected and the *IJTAG-Module's* scan multiplexers must be configured accordingly using shift and update operations. Afterwards, the RSN must be selected by

Algorithm 4.1: Creating states of a scan schedule from operation groups. It uses the two functions $CreateWriteStream(current\_sibs, next\_sibs, writes)$ and $CreateReadStream(reads)$ to create the compressed sequences of a state.

```
   Data: $groups$
   Result: $states$
1  $g = groups[0]$;
2  $wstr = GenerateWriteStream(\{\}, g.SIBs, g.Writes)$;
3  $rstr = GenerateReadStream(g.Reads)$;
4  $states.append(State(g.Subnetworks, wstr, rstr))$;
5  forall (g1, g2) in AdjacentPairs(groups) do
6      $snets = Union(g1.Subnetworks, g2.Subnetworks)$;
7      $wstr = CreateWriteStream(g1.SIBs, g2.SIBs, g.Writes)$;
8      $rstr = CreateReadStream(g.Reads)$;
9      $states.append(State(snets, wstr, rstr))$;
10 end
11 $g = groups[-1]$;
12 $wstr = GenerateWriteStream(g.SIBs, \{\}, g.Writes)$;
13 $rstr = GenerateReadStream(g.Reads)$;
14 $states.append(State(g.Subnetworks, wstr, rstr))$;
```

switching to the *IJTAG* instruction. A capture action must be triggered to latch data into the included TDRs. Now, the current write and read streams are used to generate the actual scan operation code which performs the *Writes* and *Reads* and also engages and disengages SIBs as required for the next state. Finally, an update action is triggered and the loop continues with the next state.

Generation of the register interface accesses according to the write and read streams are outlined in Algorithm 4.3. For the most part, this process is relatively straightforward. The first write packet is prepared and sent to the *APB-Module*. Afterwards, a loop starts which always prepares a write packet during serialization, then reads back the output of the previous packet (if necessary), transmits the prepared write packet and unpacks the received read packet. This goes on until no more write packets must be generated in which case the loop terminates and the last read packet is retrieved and handled.

The most intricate part of this process is of course the fact that arbitrary bit widths of the TDRs and *Variables* must be handled. All shift operations but the very last are triggered using *WriteF* transactions to keep the code efficient (although not necessarily optimal). Therefore, proper masking, shifting and merging of constants and variables is essential during the preparation of write packets and unpacking of read packets.

There is one more special case that has to be covered. In the case of a *Config* operation, config registers of the config cells must be updated after the loop in Algorithm 4.2 or in a separate function (depending on the *Split* attribute). This is done by generating code that selects all subnetworks and asserts all the SIBs that were active during the sequence.

Algorithm 4.2: Generating the code from the schedule states. Generation is modelled via the *code* object.

```
Data: $states$
Result: $code$
$subnetworks = \emptyset$;
forall $s$ in $states$ do
    if $subnetworks \neq s.Subnetworks$ then
        $code.SetInstruction(SIB)$;
        $code.SerializeSubnetworks(s.Subnetworks)$;
        $code.Update()$;
        $code.SetInstruction(IJTAG)$;
    end
    $code.Capture()$;
    $code.GenerateScanOperation(s.WriteStream, s.ReadStream)$;
    $code.Update()$;
end
```

Then, with the *EXCFG* instruction selected, zeros are shifted into the RSN. Another update action brings the RSN in its closed state while performing the configuration on all selected config cells. Since the *EXCFG* instruction is selected, the update register of the config cells is not affected in any way by the zeros that were shifted into the shift register.

Algorithm 4.3: Generation of *WriteF*, *WriteP* and *Read* transaction from *readstream* and *writestream*. Generation is modelled via the *code* object.

```
Data: $read\_stream, write\_stream, bitcount$
Result: $code$
/* Send first write packet                                    */
$writes = GetWritesInPacket(counter)$;
$code.PackWrites(writes)$;
if $16 \leq bitcount$ then
    $code.TransmitWriteF()$;
end
else
    $code.TransmitWriteP(bitcount - counter)$;
end
$counter = 16$;
while $counter \leq bitcount$ do
    /* Prepare next write packet                              */
    $writes = GetWritesInPacket(counter)$;
    $code.PackWrites(writes)$;
    /* Retrieve last read packet                              */
    $reads = GetReadsInPacket(counter - 16)$;
    if $reads \neq \emptyset$ then
        $code.RetrieveRead()$;
    end
    /* Send next write packet                                 */
    if $counter + 16 \leq bitcount$ then
        $code.TransmitWriteF()$;
    end
    else
        $code.TransmitWriteP(bitcount - counter)$;
    end
    /* Unpack last read packet                                */
    if $reads \neq \emptyset$ then
        $code.UnpackRead()$;
    end
    $counter = counter + 16$;
end
/* Retrieve and unpack final read packet                      */
$reads = GetReadsInPacket(counter - 16)$;
if $reads \neq \emptyset$ then
    $code.RetrieveRead()$;
    $code.UnpackRead()$;
end
```

CHAPTER 5

Methodology Showcase

The presented design was integrated into the RiVal 2 SoC (see Figure 5.1) to explore and demonstrate its capabilities as well as the benefits of the implemented firmware generation flow. This should show how the work presented in this thesis can offer on-chip and off-chip test, trim and configuration functionality via an SoC-specific IJTAG network. In addition, an example of firmware-generation-aided software development shall reveal how the hardware/software boundary can be bridged in a productive way.

The example chosen for this demonstration is a comparator-based self-trimming process since it uses the test and configuration aspects of the presented firmware generator and essentially forms a closed loop. On the one hand, this process requires repeated reconfiguration of trim values via the RSN. On the other hand, the comparator result which is also supplied via the RSN has to be read back after every configuration step. The reference voltage could either be supplied externally (partial self-trim) or using a pre-trimmed on-chip voltage source (full self-trim). Once a valid trim value is found, it can be stored in non-volatile on-chip memory and used on every startup to configure the trim register.

In order to generate the design, a suitable specification in the form of an MoT is required which is visualized in Figure 5.2. According to this model, the generated design shall contain a *JTAG-Module* and an *APB-Module* which shall both contain an 8-bit wide IR. The *JTAG-Module* further has combined *Sample* and *Preload* instructions since the indices are the same. Of course, the design also needs an *IJTAG-Module* which provides an RSN. In this specific case, the network consists of three subnetworks that are guarded with remote-controlled scan multiplexers. The *IJTAG-Module* instructions are implemented by the *JTAG-Module* and the *APB-Module* which means that both can be used to access the RSN.

The *InterfaceDefinition* attributes in the subnetworks point to their respective CSC MoTs. For the specific self-trim example, only the third subnetwork (*subnetwork2*) is of interest

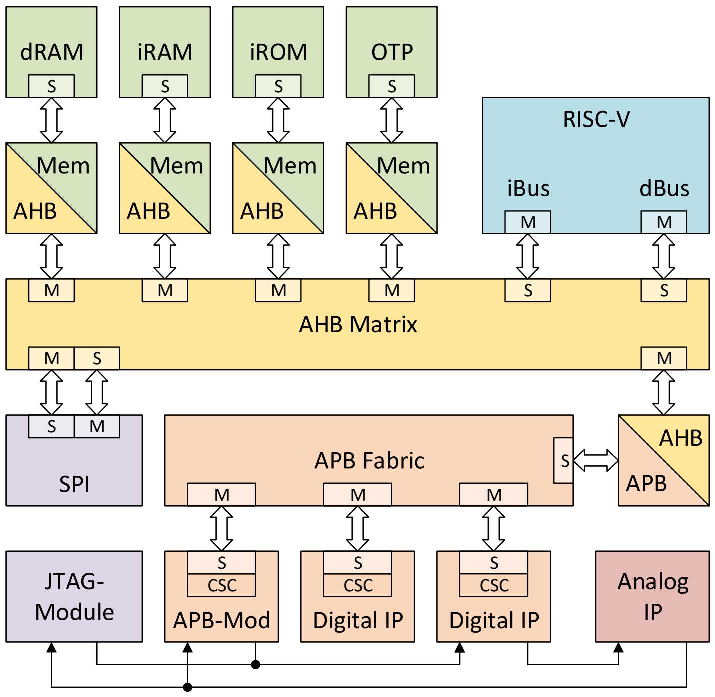


Figure 5.1: High-level overview of the RiVal 2 extended with the presented design.

Sequence	Variables				
	Name	Input	Output	Type	Bitcount
set_trim_value	trim_value	True	False	uint16_t	10
get_comparator_result	result	False	True	uin8_t	1

Table 5.1: Variables of the two *Sequences* defined in the MoT.

Sequence	Operations			
	Action	Subnetwork	TDR	Variable
set_trim_value	Write	subnetwork2	TrimValue	trim_value
	Apply			
get_comparator_result	Read	subnetwork2	ComparatorResult	result
	Apply			

Table 5.2: Operations of the two *Sequences* defined in the MoT.

since it implements the registers required for trimming. The MoT of this subnetwork is visualized in Figure 5.3. Its respective implementation would be a subnetwork consisting of two SIBs which guard a 10-bit configuration register for trimming and a 1-bit capture register for the comparator feedback.

What the described MoTs have not captured so far are the firmware-related *Sequences*. In practice, these are described in the main MoT but they are not particularly understandable in a graphical representation. Therefore, the definition of the two *Sequences* that are

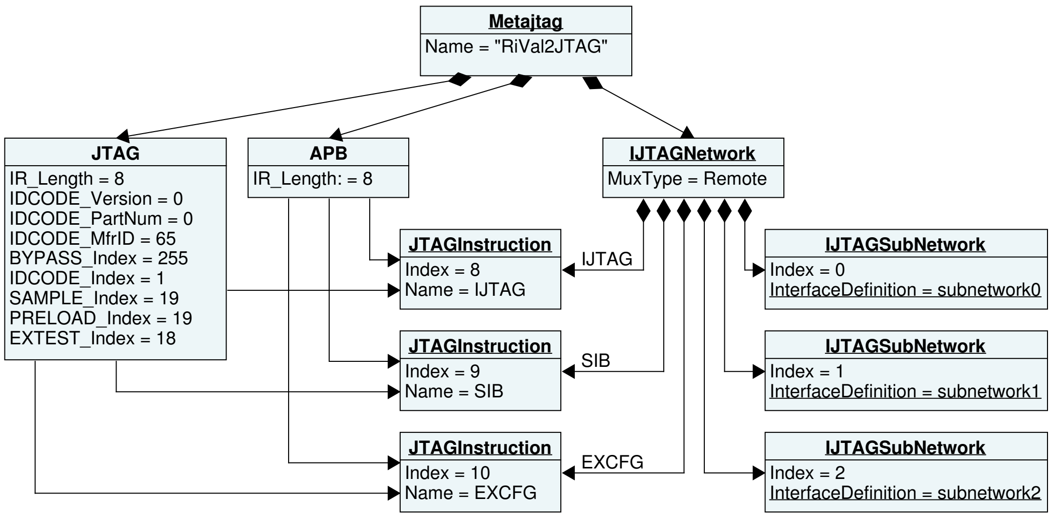


Figure 5.2: The MoT used for generation of the design for the RiVal 2. The *Sequences* are omitted for brevity.

used for the presented example are instead outlined in Table 5.1 and Table 5.2. The *set_trim_value* can be used to update the value of the trim register. It has set *Config* and *Split* flags which means that two functions are generated for split-phase configuration. The *get_comparator_result* is used to capture and return the comparator result. Therefore, the *Config* flag is set to false.

The final design generated from the presented MoT has an extensive interface which can be seen in Table 5.3. It is apparent that there are two interfaces for interaction with the implemented scan chains (*jtag* and *apb*). The *jtag_apb_switch* signal can be used to choose which one should be controlling the IJTAG network. All the external scan chains that are implemented by the *JTAG-Module* can be operated via the *jtag_** interfaces and sideband signals. The IJTAG subnetworks can be attached to the *ijtag* interface as well as their respective *IJTAG-TDR-Interfaces*. One can also also see that there are **_clock_rising* and **_clock_falling* signals. These were introduced by the edge detection transformation that was applied to the generated MoD before generating the design sources. Therefore, they also have to be forwarded to the transformed subnetworks or any other future additions to the scan architecture.

The generated *subnetwork2* connects to this via the *ijtag* and *subnetwork2* interfaces. Moreover, it provides a 10-bit output value and a 1-bit input value which connect to the unit to be trimmed and the comparator output respectively. This way, everything that is required for the self-trim routine on the hardware side is wired up accordingly.

On the firmware side, three functions are provided. The generated code can be seen in Listing 5 and Listing 6. Since the required *Sequences* for this application are rather simple and only one transaction is required per scan operation, the code is relatively easy

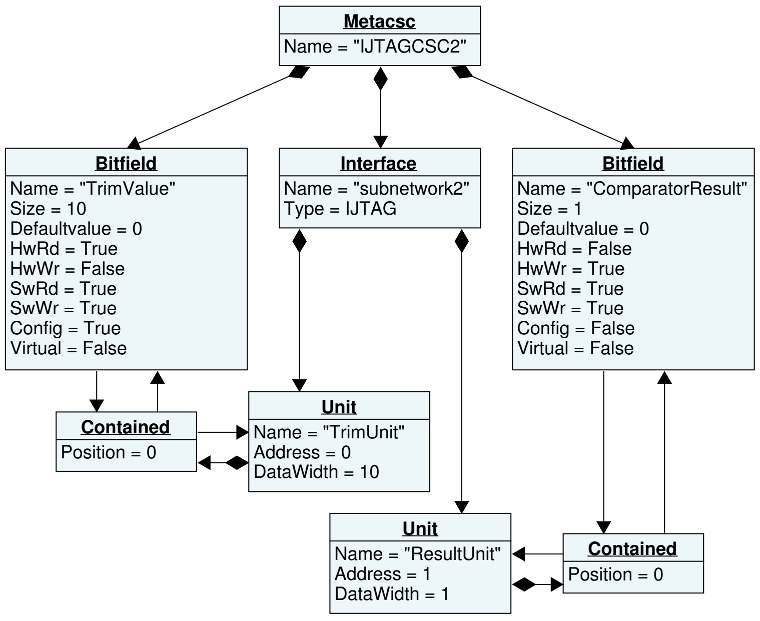


Figure 5.3: The MoT used for description of *subnetwork2* in the generated design.

to comprehend. Essentially, it always selects the *SIB* instruction (index 9) and activates the required subnetwork(s) (in this case *subnetwork2*). Then, the *IJTAG* instruction is selected to enable the SIB of the *TrimUnit* or the *ResultUnit*. Finally, the writing of the trim value or reading of the result and closing of the SIB is performed. While this is a simple piece of code, it is already apparent that, even though the SIBs ensure high flexibility, keeping track of them and changing their states when necessary can complicate scan operations significantly. This is mainly because, from a programming perspective, they are mixing up the separation between control flow and data signals.

Fortunately, because of the provided driver generation, it is not necessary for the application programmers to care about these details of the RSN. Instead, they can focus on the self-trimming algorithm that they want to employ in the final design. One example of trimming with a comparator would be a simple binary search within the range of possible values. Listing 7 shows an example of a binary search routine that can be employed for the 10 bit trim value.

This application of the proposed flow is a perfect example of how code and IP reuse can be improved. On the one hand, the definition of the subnetwork which is used for self-trimming can be seen as a building block that can be embedded in any JTAG MoT as required. Since the self-trim-related functions are generated automatically, the source code which makes use of *subnetwork2*-specific driver code is also independent of the RSN structure.

It is also noticeable how such a routine can improve the trimming process in general.

Name	Type	Direction
sysclk	clk	in
sysrst_n	reset	in
jtag_apb_switch	logic	in
apb	APB-Interface	in
jtag	JTAG-Interface	in
jtag_apb_switch	logic	in
jtag_tap	JTAG-TAP-Interface	out
jtag_tap_ext	JTAG-TAP-EXT-Interface	out
jtag_bsc	JTAG-TDR-Interface	out
jtag_bypass_sel	logic	out
jtag_idcode_sel	logic	out
jtag_sample_preload_sel	logic	out
jtag_extest_sel	logic	out
jtag_tap_clock_rising	logic	out
jtag_tap_clock_falling	logic	out
ijtag	IJTAG-TAP-Interface	out
subnetwork0	IJTAG-TDR-Interface	out
subnetwork1	IJTAG-TDR-Interface	out
subnetwork2	IJTAG-TDR-Interface	out
ijtag_config_lock	logic	in
ijtag_error	logic	out
ijtag_tap_clock_rising	logic	out
ijtag_tap_lock_falling	logic	out

Table 5.3: Interface of the generated design.

Usually, even without the possibility of on-chip RSN-access, this setup would already enable ATE-driven trimming. Using dedicated on-chip trim instruments would also allow for self-trimming. With the power of the CPU-based access via the *APB-Module*, it is possible to employ self-trimming without any application-specific hardware overhead. Moreover, since the self-trim solution is software-based, it is very easy to make changes to the trim routine. Finally, more parallelization is now possible since less ATE resources are needed and multiple chips can be programmed and trimmed in parallel.

This is just one example of a CPU-driven self-trim and configuration routine. In principle, any proven self-test or self-trim method which would usually require a dedicated hardware state machine can be realized with this flow. Notably, the *APB-Module* is a one-time investment in terms of hardware overhead that can be used for any element embedded in the entire RSN.

CHAPTER 6

Discussion and Outlook

The primary objective of this thesis was the development of a modular on-chip testing infrastructure that supports IEEE 1149.1 conforming EOL testing as well as CPU-driven self-testing. Furthermore, employment of an RSN according to IEEE 1687 should foster IP reuse and test procedure reuse and ensure high flexibility and performance during scan operations. Finally, a high-level abstraction of CPU-driven scan operations was a key requirement to ensure productive software development.

The flow presented in this thesis aims at providing all these aspects via the generation of hardware and firmware from a single specification as visualized in Figure 6.1. Provision of a standard-conforming *JTAG-Module* for external accesses as well as a specially designed *APB-Module* for interactions via the CPU enable partitioning of EOL testing into off-chip and on-chip procedures. Moreover, the CPU-access allows for self-testing and IP configuration in the field with little to no per-IP hardware overhead. The specification also contains all information that is required to generate a system-specific RSN from an arbitrary set of IP- and EI-specific subnetworks in a modular way. This eases the reuse of IPs and EIs while also ensuring high flexibility in the development of SoCs. Further, the automatic generation of system- and application-specific drivers from high-level descriptions is made possible via the firmware-related information which is also encoded directly in the specification. This releases the burden of writing RSN-specific code from software developers and therefore boosts productivity while eliminating the probability of bugs.

Although the presented approach addresses the mentioned requirements, it is certainly not in a completely finalized state simply due to the fact that it connects to so many topics related to DFT, EOL testing and BIST. As hinted in Figure 6.1, one of the most important future additions is surely the extraction of an RSN description from the specification in the standardized ICL format. This information is what enables EDA tools to perform retargeting of test procedures in the PDL format. Therefore, this addition is necessary to link EDA tools like the Tessent suite [25, 28] and EOL test procedures

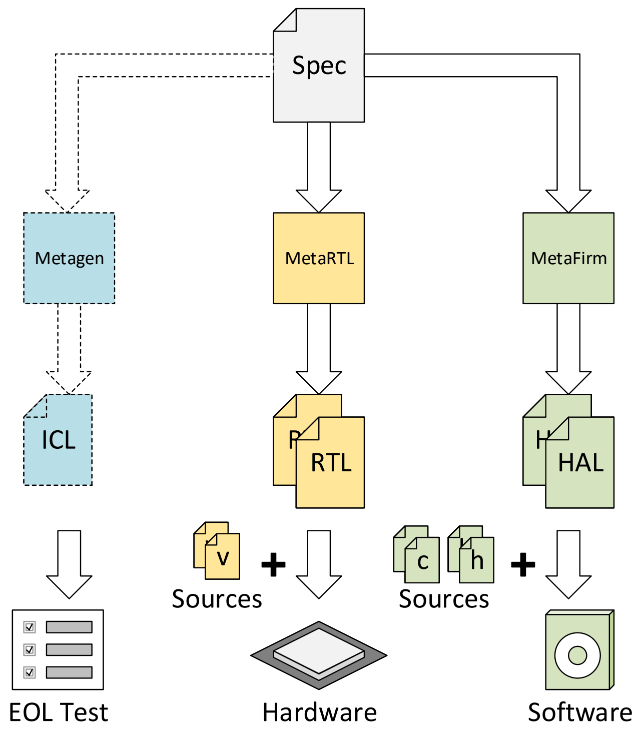


Figure 6.1: Concluding overview of the flow presented in this thesis. Dashed lines mark future extensions of the approach.

with the generated RSN of the physical chip. Since the metamodel was created with this feature in mind, a given MoT already contains all information that is required for the generation of a corresponding ICL definition using an additional Metagen-based generator.

Another future improvement related to EOL testing is addressing the unsafe multiplexer in the clock path of the *IJTAG-Module* (or any other future shared scan chains). As outlined in Section 4.3.6, the edge detection transformation can be used to circumvent this problem to some extent. However, since the TCK must now be synchronized and sampled, some constraints must be fulfilled to ensure stable external operation via the *JTAG-Module*. According to the IEEE 1149.1 standard, inputs are clocked in at the rising edge while outputs are clocked out at the falling edge. Usually, given TCK's high period t^h_{tck} and low period t^l_{tck} as well as TCK input delay d_{tck}, TDI input delay d_{in} and TDO output delay d_{out} it must hold that

$$d_{tck} + d_{out} < t^l_{tck}$$

$$d_{tck} - d_{in} < t^h_{tck}$$

It is worth noting that the second constraint is inherently fulfilled for reasonable delay values much smaller than the clock period and similar d_{tck} and d_{in}.

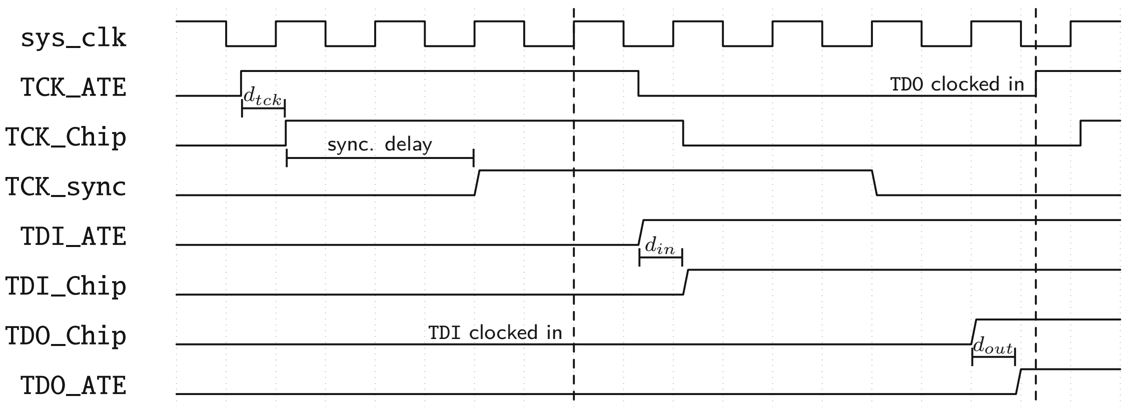


Figure 6.2: Timing diagram visualizing the oversampling of the TCK signal with the system clock and how transmission and synchronization delays affect satisfiability of the JTAG protocol.

After the transformation, the system clock must be fast enough to sample TCK without the overhead of the synchronization delay causing violations of the above constraints. For a 2 stage synchronizer, it must hold that

$$3 \cdot t_{sys} + d_{tck} + d_{out} < t_{tck}^{l}$$

$$3 \cdot t_{sys} + d_{tck} - d_{in} < t_{tck}^{h}$$

Again, the second constraint is not as critical as the first one. When assuming a 50% duty cycle, complying with the first constraint even implies satisfaction of the second provided that $d_{in} < -d_{out}$. In this case, the criterion reduces to

$$3 \cdot t_{sys} + d_{tck} + d_{out} < \frac{t_{tck}}{2}$$

In other words, TCK should be oversampled by at least a factor of 8 to ensure that the JTAG protocol is not violated. Of course, depending on the propagation delays, this factor may be even higher. Figure 6.2 illustrates this situation.

While totally reasonable for exploration of the design capabilities, this may already entail a significant speed limit on EOL testing. Morover, during EOL testing, the system clock may not be trimmed yet which could complicate the testing process and lead to even higher test times. Therefore, a viable alternative would be the introduction of a clock multiplexer which would also have to be provided via a MetaRTL primitive. The nice thing about this addition is that no change of the design generator is required to also offer an alternative clock multiplexer transformation.

It is also of high interest to develop a good on-chip vs. off-chip partitioning approach. Unfortunately, at the time of writing this thesis, the RiVal 2 was not yet in a stable state and while the presented design was verified in [37], extensive exploration of on-chip testing and trimming was not possible. Therefore, an in-depth analysis of the time required to perform common EOL testing routines is needed in the future. When moving test routines from ATE to the SoC-internal software-driven applications, there are some factors that have to be taken into account. On the one hand, being able to free up ATE resources and the fact that each chip can do its own testing can be expected to increase the potential parallelism. On the other hand, some scan operations may take more time when they are run on the CPU simply because of the delays introduced during memory transactions and required instructions. Moreover, the time it takes to program and boot the SoCs must be considered. Therefore, it is necessary to evaluate the possible parallelism p and the sequential and parallelizable processing times t_{seq} and t_{par} of both approaches. They could then be compared to one another in terms of their overall time required per tested unit $t_{unit} = t_{seq}/p + t_{par}$. By collecting this information for further showcases, the profitability of the approach can be evaluated accordingly.

Of course, there is also room for the development of the individual modules of the presented design. The *APB-Module* acts as a bridge between the CPU and the RSN. Therefore, it is certainly worth exploring variations of the proposed design.

In the current state, it is required to issue a *Read* transaction after a *Write** transaction if the output data is required. However, it would be possible to employ an output queue that makes it possible to buffer multiple packets of output data. The existing *Control* register could be used to enable and disable the buffering of deserialized results as needed. Moreover, input queues could also be employed to reduce the number of stalling cycles when a lot of *Write** transactions are performed in rapid succession.

Another possible way of boosting the *APB-Module's* performance would be an option to perform parallel writes to the TDR which controls the scan multiplexers between the subnetworks. This would make the inclusion and exclusion of subnetworks much faster. As a matter of fact, the *IJTAG-Module* already supports the optional generation of a parallel input for this TDR. It is merely a matter of extending the *APB-Module's* interface with an option to make use of this feature.

This thesis has only addressed the APB interface with a data width of 16 bits for this module. An obvious extension would be to offer alternative interfaces and data widths. This is actually very straightforward since a special metalib has been developed to handle the mapping of arbitrary bus interfaces to the CSC's *DataBus* and almost no changes to the module's generator will be necessary.

On the IJTAG side, it would be possible to further extend the CSC metamodel or extract the IJTAG-specific parts altogether to provide more possibilities with respect to the RSN structure. For example, it could be interesting to allow multiple levels of hierarchy per subnetwork. Moreover, special SIBs could be introduced that capture a constant zero when the *EXCFG* instruction is selected. This would make the update of the config registers more efficient since it is not necessary to fill the entire active scan path with zeros. Instead, engaging the relevant SIBs and then performing a capture action followed by an update action would perform the configuration and close all the SIBs within a few cycles.

Finally, there are also a few possible improvements with respect to firmware generation. An obvious one is the fact that the *Read* operation currently does not support any expected values like PDL does. The current flow does of course allow to return the read values and the application can check these against the expected values. Nevertheless, supporting the generation of routines that automatically check the read values against expected values would certainly be useful.

Another useful feature to strengthen the bonding with third-party IP and EDA tools would be the implementation of a PDL-to-*Sequence* translator. Since *Sequence* descriptions are already close to PDL (level 0), this would allow retargeting of PDL definitions into driver routines for the proposed design without any changes in the current generators. The cross compiler flow outlined in [31, 32] is an example of how this can boost productivity and compatibility even further.

A major difference to the off-line retargeting presented in this thesis is that the RSN controller in [31, 32] performs on-line retargeting to enable the use of PDL level 1 definitions. Of course, on-line retargeting comes with a major hardware overhead in comparison to the lightweight design presented in this thesis. Interestingly, the support of PDL-1 is not at all impossible with off-line retargeting since the power of the on-chip CPU is available. The key extension would be to introduce wrappers around the static *Sequences* that allow the mapping of control flow commands. These could then be translated into the corresponding control flow constructs in the C programming language.

Another interesting feature would be to embed callbacks in the high-level description of scan operations. The basic idea would be to let the generator embed a call to a specific function with arguments. This could be used to allow software developers to hook into the scan operation assembling process which could increase the flexibility drastically.

An important aspect concerning reusability would be the modification of the metamodel in such a way that the *Sequences* can also be provided as modular building blocks just like the subnetworks. This would provide support for what one could call libraries of *Sequences* which could be included in the specification as required. However, this process is not trivial since some *Sequences* may rely on the presence of very specific or very broad classes of subnetworks. The most appropriate approach would probably be a dedicated generator pass that is capable of transforming standalone *Sequence* libraries into MoT-specific *Sequences*. While it would be very beneficial for code reuse and reuse of custom *Sequences* across multiple MoTs, PDL-to-*Sequence* translation is certainly more important.

A final point to consider is how the driver code is generated. At the moment, the generator tries to pack all shift operations into as little APB transactions as possible. However, this does not take the runtime of the code on the CPU into account. Realistically, even though serialization is running in parallel to data preparation, it may sometimes be worth it to not squeeze a few bits of data from a variable into an almost full packet if that would mean that for the next packet less shift and mask operations are required. However, this would already require very platform-specific knowledge of the target system. Moreover, finding efficient schedules for the operation of RSNs is in general NP-hard. Therefore, any changes in the code generation should be driven by the incentive for more optimal code based on the results of actual benchmarks.

CHAPTER 7

Summary

The presented framework of hardware generators enables the straightforward provision of RSN-based on-chip testing infrastructure which supports conventional EOL testing as well as CPU-driven self-testing, self-trimming and IP configuration. A corresponding firmware generation flow helps in bridging the hardware/software boundary by raising the abstraction of CPU-interaction with the embedded scan chains. The generators are based on Infineon's metamodeling code generation frameworks Metagen, MetaRTL and MetaFirm which enable the use of single-source specifications and support the modularity, flexibility and reusability of the proposed flow.

The backbone of the modular on-chip testing infrastructure is an IEEE 1687 conforming RSN which can be assembled from reusable subnetwork definitions on a case-by-case basis to foster IP- and EI-reuse. Employing the principles of the IJTAG standard also resolves shortcomings of the JTAG standard without interfering with its requirements. This way, the generated design can also offer a IEEE 1149.1 conforming JTAG interface to enable transparent operation via existing ATE and EDA toolchains. Moreover, the set of instructions implemented by the TAP controller is extensible which further boosts flexibility with respect to future additions. A specially developed peripheral allows the CPU to address the RSN which makes it possible to augment EOL testing with smart software-driven self-testing and self-trimming. Moreover, it enables self-testing, self-trimming and IP configuration in the field by reusing embedded scan chains with minimal to no hardware overhead.

The firmware generator enhances this approach by enabling the generation of test-, trim- and configuration-related code from high-level descriptions of scan operations close to the standardized PDL. By abstracting away the system-specific details of the generated RSN and the CPU-to-RSN controller, it decouples software routines from the hardware description and potential changes. This way, it equips software developers with the tools that are required to focus on the algorithmic aspects of self-test, self-trim and configuration routines and fosters code reuse.

The generation of a particular design and its integration in an SoC is demonstrated on the basis of the RiVal 2 SoC. The implementation of a software-driven self-trim routine is outlined as an example of how the presented flow can be employed to improve EOL throughput and enable self-testing, self-trimming and IP configuration in an actual product.

List of Figures

List of Tables

List of Algorithms

Acronyms

ADC	Analog Digital Converter. 21
AHB	Advanced High-Performance Bus. 17, 24
APB	Advanced Peripheral Bus. 17, 20, 24, 27, 30–32, 52, 53, 57
API	Application Programming Interface. 11, 12, 15, 16
ATE	Automated Test Equipment. 1, 21, 23, 47, 51, 52, 55
BIST	Built-In Self-Test. xiii, 1, 19–21, 49
BSC	Boundary Scan Chain. 6, 7, 29
CIM	Computation Independent Model. 13, 14
CPU	Central Processing Unit. xi, xiii, 2, 17, 20, 24, 30, 36, 47, 49, 52, 53, 55
CSC	Control Status Configuration. 16, 17, 25–27, 32, 37, 43, 52, 57
DFT	Design For Test. xi, xiii, 1, 3, 49
EDA	Electronic Design Automation. 1, 7, 19, 20, 49, 53, 55
EI	Embedded Instrument. xi, xiii, 1, 7–9, 33, 49, 55
EOL	End of Line. xi, xiii, 1, 2, 18, 19, 49–51, 55, 56
FF	Flip Flop. 3, 7
FIR	Finite Impulse Response. 14
GUI	Graphical User Interface. 11
HAL	Hardware Abstraction Layer. 2, 16, 17, 26, 36, 37, 59, 74, 75
HDL	Hardware Description Language. 12, 14, 15

IC Integrated Circuit. xi, xiii, 1, 3, 6, 7

ICL Instrument Connectivity Language. 9, 20, 21, 49, 50

IJTAG Integrated Joint Test Action Group. xi, xiii, 8, 19–21, 24, 25, 32, 36, 38, 43, 45, 52, 55, 57, 59

IP Intellectual Property. xi, xiii, 1, 2, 7, 9, 18–21, 24, 33, 46, 49, 53, 55, 56

IR Instruction Register. 4, 9, 24, 27, 29, 30, 43

I²C Inter-Integrated Circuit. 20, 21

JTAG Joint Test Action Group. xi, xiii, 3, 7, 19, 24–28, 31, 33, 46, 51, 55, 57, 58

LSB Least Signigicant Bit. 5, 31, 33, 34

MDA Model Driven Architecture. 9, 12–16, 57

MoD Model-of-Design. 14–16, 32, 34, 35, 45

MOF Meta-Object Facility. 12

MoF Model-of-Firmware. 17

MoT Model-of-Things. 14, 16, 17, 26, 27, 31, 32, 34, 35, 37, 38, 43–46, 50, 53, 57–59

MoV Model-of-View. 14, 15, 17

MSB Most Signigicant Bit. 5, 31–34

NRE Non-Recurring Engineering. 12

OCDS On Chip Debug System. 7, 24, 35, 58

OCL Object Constraint language. 10

OMG Object Management Group. 12

PCB Printed Circuit Board. 3, 5–7

PDL Procedural Description Language. xi, xiii, 9, 20, 21, 25, 28, 38, 49, 53, 55

PIM Platform Independent Model. 13, 14

PM Platform Model. 14

PSM Platform Specific Model. 13, 14

RSN	Reconfigurable Scan Network. xi, xiii, 8, 9, 19–21, 24, 25, 27, 34, 37–39, 41, 43, 46, 47, 49, 50, 52, 53, 55
RTL	Register Transfer Level. 9, 14, 15, 57
SFF	Scannable Flip Flop. 7
SIB	Segment Insertion Bit. 8, 9, 20, 24, 27, 32–34, 37, 39, 40, 44, 46, 52, 57
SoC	System On Chip. xi, xiii, 1, 2, 7, 9, 17, 19, 24, 28, 35, 43, 49, 51, 52, 56, 59
SPI	Serial Peripheral Interface. 17, 20
TAP	Test Access Port. 3–8, 19–21, 23, 28, 29, 55, 57, 59
TCK	Test Clock. 4, 5, 28, 35, 36, 50, 51, 58
TDI	Test Data Input. 4, 5, 28, 50
TDO	Test Data Output. 4, 5, 28, 50
TDR	Test Data Register. 4–9, 20, 21, 23–25, 27–29, 31–34, 38–40, 52, 57–59
TMS	Test Mode Select. 4, 5, 28, 57
ToD	Template-of-Design. 14, 16
ToF	Template-of-Firmware. 17
ToV	Template-of-View. 15
UML	Unified Modeling Language. 10, 11, 25, 26, 57

Bibliography

[1] B. Vermeulen, C. Hora, B. Kruseman, E.J. Marinissen, and R. van Rijsinge. Trends in testing integrated circuits. In *2004 International Conferce on Test*, pages 688–697, 2004.

[2] Manibha Sharma and Jasdeep Dhanoa. Smart logic built in self-test in soc. In *2020 5th IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, pages 1–4, 2020.

[3] G. Giles. Is scan (alone) sufficient to test today's microprocessors? not quite, but we can't get the job done without it. In *Proceedings. International Test Conference*, pages 1197–, 2002.

[4] Paolo Bernardi, Lyl Mercedes Ciganda, Ernesto Sanchez, and Matteo Sonza Reorda. MIHST: A hardware technique for embedded microprocessor functional on-line self-test. *IEEE Transactions on Computers*, 63(11):2760–2771, 2014.

[5] R.C. Aitken. On-chip versus off-chip test: an artificial dichotomy. In *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, pages 1146–, 1998.

[6] Sarveswara Tammali. Industrial practices of test cost reduction: Perspective, current design practices. In *2010 28th VLSI Test Symposium (VTS)*, pages 124–124, 2010.

[7] Hans Martin von Staudt and Alexios Spyronasios. Using IJTAG digital islands in analogue circuits to perform trim and test functions. In *2015 IEEE 20th International Mixed-Signals Testing Workshop (IMSTW)*, pages 1–5, 2015.

[8] Rajesh Mittal, Mudasir Kawoosa, and Rubin A. Parekhji. Systematic approach for trim test time optimization: Case study on a multi-core rf soc. In *2014 International Test Conference*, pages 1–9, 2014.

[9] IEEE standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444, 2013.

[10] IEEE standard for access and control of instrumentation embedded within a semiconductor device. *IEEE Std 1687-2014*, pages 1–283, 2014.

[11] Kristof Van Beeck, Filip Heylen, Jan Meel, and Toon Goedemé. Comparative study of model-based hardware design tools. 03 2010.

[12] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. Metasynthesis for designing automotive socs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[13] Wolfgang Ecker, Michael Velten, Leily Zafari, and Ajay Goyal. The metamodeling approach to system level synthesis. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–2, 2014.

[14] Liangora Research Lab. What is MDA? why considering BNPM. `https://research.linagora.com/pages/viewpage.action?pageId=3639295`, 2006. Last Accessed: 12-12-2021.

[15] Frank Truyen. The fast guide to model driven architecture. `https://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf`, 2006. Last Accessed: 12-12-2021.

[16] OMG. MDA - the architecture of choice for a changing world. http://www.omg.org/mda/, 2016. Last Accessed: 12-12-2021.

[17] John M. Siegel. Model driven architecture® (MDA): The MDA guide rev 2.0. https://www.omg.org/mda/presentations.htm, 2014. Last Accessed: 12-12-2021.

[18] Wolfgang Ecker and Johannes Schreiner. Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient hardware generators. In *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, 2016.

[19] Johannes Schreiner, Rainer Findenigy, and Wolfgang Ecker. Design centric modeling of digital hardware. In *2016 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 46–52, 2016.

[20] Johannes Schreiner and Wolfgang Ecker. Digital hardware design based on metamodels and model transformations. pages 83–107, 09 2017.

[21] Zhao Han, Keerthikumara Devarajegowda, Michael Werner, and Wolfgang Ecker. Towards a python-based one language ecosystem for embedded systems automation. In *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, pages 1–7, 2019.

[22] RISC-V International®. Risc-v specifications. https://riscv.org/technical/specifications/. Last Accessed: 12-12-2021.

[23] Alan Sguigna. JTAG/boundary scan for built-in test. In *2018 IEEE AUTOTESTCON*, pages 1–3, 2018.

[24] Ghazanfar Ali, Ahmed Badawy, and Hans G. Kerkhoff. Accessing on-chip temperature health monitors using the IEEE 1687 standard. In *2016 IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 776–779, 2016.

[25] Siemens. Tessent™ IJTAG fact sheet. `https://static.sw.cdn.siemens.com/siemens-disw-assets/public/3fhajguyzbra67gu0lrW90/en-US/Siemens-SW-Tessent-IJTAG-FS-82810-C3.pdf`, 2021. Last Accessed: 12-12-2021.

[26] Siemens. Tessent™ MemoryBIST fact sheet. `https://static.sw.cdn.siemens.com/siemens-disw-assets/public/81225/en-US/Siemens-SW-Embedded-memory-self-test-repair-and-debug-Tessent-MemoryBIST-FS-81225-C1`, 2015. Last Accessed: 12-12-2021.

[27] Siemens. Tessent™ LogicBIST fact sheet. `https://static.sw.cdn.siemens.com/siemens-disw-assets/public/7kVdJZfTR9JCxdNplREEpY/en-US/Siemens-SW-tessent-logicbist-FS-82711-C2.pdf`, 2020. Last Accessed: 12-12-2021.

[28] Siemens. Tessent™ MissionMode fact sheet. `https://static.sw.cdn.siemens.com/siemens-disw-assets/public/81226/en-US/Siemens-SW-In-system-test-and-diagnosis-of-automotive-ICs-Tessent-MissionMode-FS-81226-C1`, 2017. Last Accessed: 12-12-2021.

[29] Siemens. *Tessent™ MissionMode User's Manual*. Siemens.

[30] Alfred L. Crouch, Bradford G. Van Treuren, and Jeff Rearick. P1687.1: Accessing embedded 1687 instruments using alternate device interfaces other than JTAG. In *2019 IEEE AUTOTESTCON*, pages 1–7, 2019.

[31] Ahmed Ibrahim and Hans G. Kerkhoff. Towards an automated and reusable in-field self-test solution for mpsocs. In *2016 28th International Conference on Microelectronics (ICM)*, pages 249–252, 2016.

[32] Ahmed M. Y. Ibrahim and Hans G. Kerkhoff. An on-chip ieee 1687 network controller for reliability and functional safety management of system-on-chips. In *2019 IEEE International Test Conference in Asia (ITC-Asia)*, pages 109–114, 2019.

[33] Anton Tsertov, Artur Jutman, Konstantin Shibin, and Sergei Devadze. Ieee 1687 compliant ecosystem for embedded instrumentation access and in-field health monitoring. In *2018 IEEE AUTOTESTCON*, pages 1–9, 2018.

[34] Farrokh Ghani Zadegan, Dimitar Nikolov, and Erik Larsson. On-chip fault monitoring using self-reconfiguring IEEE 1687 networks. *IEEE Transactions on Computers*, 67(2):237–251, 2018.

[35] Hans Martin von Staudt. Comparator based self-trim and self-test scheme for arbitrary analogue on-chip values. In *2010 IEEE 16th International Mixed-Signals, Sensors and Systems Test Workshop (IMS3TW)*, pages 1–6, 2010.

[36] Hans Martin von Staudt. Trim DAC design with minimum DNL for self-trim with self-test schemes. In *2011 IEEE 17th International Mixed-Signals, Sensors and Systems Test Workshop*, pages 19–24, 2011.

[37] Timotei Muresan. Pre-silicon verification environment for virtual test simulation of IJTAG-based testing. Bachelor's thesis, FH Joanneum, 2021.

APPENDIX A

Source Code

A.1 Hardware Abstraction Layer

JTAG_HAL.h

```c
#ifndef _JTAG_HAL_H_
#define _JTAG_HAL_H_

// AUTO GENERATED CODE //
/*****************************************************************************************************************/
/**
 * @file     JTAG_HAL.h
 * @author   pircher
 * @date     11:02:51    27/12/2021
 * @version  1.0
 * HAL File containing bit field mapping with
 * predefined function to access bit fields
*/
#include <stdint.h>
#include "types.h"
#include "csr.h"

/**
 * Interface: APB2TDR
*/

//
//Connected to Address Bus with Width: 16
//Data Bus Width: 16
//Selected Byte Order: Little
//Addressable Unit: 16

/**
 * Register: InstructionReg_JTAG
 * The value contained at the given address, so the data starts at 0xe80.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define INSTRUCTIONREG_JTAG_16_0     *(volatile uint16_t*) (0xe80)
//Bitfield: 0 // T, T, T, T; [8]
#define INSTRUCTION_JTAG_16_MASK     255
#define INSTRUCTION_JTAG_16_SHIFT     0

/**
 * Register: ActionReg_JTAG
 * The value contained at the given address, so the data starts at 0xe82.
*/
```

```

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define ACTIONREG_JTAG_16_0     *(volatile uint16_t*) (0xe82)
//Bitfield: 0 // F, T, T, T; [2]
#define ACTION_JTAG_16_MASK     3
#define ACTION_JTAG_16_SHIFT     0

/**
 * Register: WriteFReg_JTAG
 * The value contained at the given address, so the data starts at 0xe84.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define WRITEFREG_JTAG_16_0     *(volatile uint16_t*) (0xe84)
//Bitfield: 0 // F, T, T, F; [16]
#define WRITEF_JTAG_16_MASK     65535
#define WRITEF_JTAG_16_SHIFT     0

/**
 * Register: WritePReg_JTAG
 * The value contained at the given address, so the data starts at 0xe86.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define WRITEPREG_JTAG_16_0     *(volatile uint16_t*) (0xe86)
//Bitfield: 0 // F, T, T, F; [16]
#define WRITEP_JTAG_16_MASK     65535
#define WRITEP_JTAG_16_SHIFT     0

/**
 * Register: ReadReg_JTAG
 * The value contained at the given address, so the data starts at 0xe88.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define READREG_JTAG_16_0     *(volatile uint16_t*) (0xe88)
//Bitfield: 0 // T, F, F, T; [16]
#define READ_JTAG_16_MASK     65535
#define READ_JTAG_16_SHIFT     0

/**
 * Register: ControlReg_JTAG
 * The value contained at the given address, so the data starts at 0xe8a.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define CONTROLREG_JTAG_16_0     *(volatile uint16_t*) (0xe8a)
//Bitfield: 0 // T, T, T, F; [1]
#define CONTROL_JTAG_16_MASK     1
#define CONTROL_JTAG_16_SHIFT     0

/**
 * Register: StatusReg_JTAG
 * The value contained at the given address, so the data starts at 0xe8c.
*/

//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
//Contained Bitfields: Position // SwRd, SwWr, HwRd, HwWr; [Size];
#define STATUSREG_JTAG_16_0     *(volatile uint16_t*) (0xe8c)
//Bitfield: 0 // T, F, F, T; [1]
#define STATUS_JTAG_16_MASK     1
#define STATUS_JTAG_16_SHIFT     0


/**
 * Struct with all function pointers used to access bitfields of JTAG
*/
typedef struct JTAG_HAL_Config {
    void (*INSTRUCTION_WRITE) (uint8_t Instruction);
    uint8_t (*INSTRUCTION_READ) (void);
    void (*ACTION_WRITE) (uint8_t Action);
    void (*WRITEF_WRITE) (uint16_t WriteF);
    void (*WRITEP_WRITE) (uint16_t WriteP);
    uint16_t (*READ_READ) (void);
    void (*CONTROL_WRITE) (bool Control);
    bool (*CONTROL_READ) (void);
    bool (*STATUS_READ) (void);
} JTAG_HAL_Config;

extern JTAG_HAL_Config JTAG_HAL; /**< Handler pointing to all Bitfield Sequences*/;

/**
 * Access Function: SIMPLE
 * Write INSTRUCTION_JTAG, on INSTRUCTIONREG_JTAG_16_0
 * @param  Instruction_JTAG
 * @return void
**/
static inline void INSTRUCTION_WRITE_JTAG(uint8_t Instruction_JTAG) __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Read INSTRUCTION_JTAG, on INSTRUCTIONREG_JTAG_16_0
 * @param
 * @return uint8_t
**/
static inline uint8_t INSTRUCTION_READ_JTAG() __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Write ACTION_JTAG, on ACTIONREG_JTAG_16_0
 * @param  Action_JTAG
 * @return void
**/
static inline void ACTION_WRITE_JTAG(uint8_t Action_JTAG) __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Write WRITEF_JTAG, on WRITEFREG_JTAG_16_0
 * @param  WriteF_JTAG
 * @return void
**/
static inline void WRITEF_WRITE_JTAG(uint16_t WriteF_JTAG) __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Write WRITEP_JTAG, on WRITEPREG_JTAG_16_0
 * @param  WriteP_JTAG
 * @return void
**/
static inline void WRITEP_WRITE_JTAG(uint16_t WriteP_JTAG) __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Read READ_JTAG, on READREG_JTAG_16_0
 * @param
 * @return uint16_t
**/
static inline uint16_t READ_READ_JTAG() __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Write CONTROL_JTAG, on CONTROLREG_JTAG_16_0
 * @param  Control_JTAG
 * @return void
**/
static inline void CONTROL_WRITE_JTAG(bool Control_JTAG) __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Read CONTROL_JTAG, on CONTROLREG_JTAG_16_0
 * @param
 * @return bool
**/
static inline bool CONTROL_READ_JTAG() __attribute__((always_inline));

/**
 * Access Function: SIMPLE
 * Read STATUS_JTAG, on STATUSREG_JTAG_16_0
 * @param
 * @return bool
**/
static inline bool STATUS_READ_JTAG() __attribute__((always_inline));

inline void INSTRUCTION_WRITE_JTAG(uint8_t Instruction_JTAG) {
    INSTRUCTIONREG_JTAG_16_0 = (Instruction_JTAG & INSTRUCTION_JTAG_16_MASK);
}

inline uint8_t INSTRUCTION_READ_JTAG() {
    return (INSTRUCTIONREG_JTAG_16_0 & INSTRUCTION_JTAG_16_MASK);
}

inline void ACTION_WRITE_JTAG(uint8_t Action_JTAG) {
    ACTIONREG_JTAG_16_0 = (Action_JTAG & ACTION_JTAG_16_MASK);
}

inline void WRITEF_WRITE_JTAG(uint16_t WriteF_JTAG) {
    WRITEFREG_JTAG_16_0 = (WriteF_JTAG & WRITEF_JTAG_16_MASK);
}

inline void WRITEP_WRITE_JTAG(uint16_t WriteP_JTAG) {
    WRITEPREG_JTAG_16_0 = (WriteP_JTAG & WRITEP_JTAG_16_MASK);
}

inline uint16_t READ_READ_JTAG() {
    return (READREG_JTAG_16_0 & READ_JTAG_16_MASK);
}

inline void CONTROL_WRITE_JTAG(bool Control_JTAG) {
    CONTROLREG_JTAG_16_0 = (Control_JTAG & CONTROL_JTAG_16_MASK);
}

inline bool CONTROL_READ_JTAG() {
    return (CONTROLREG_JTAG_16_0 & CONTROL_JTAG_16_MASK);
}

inline bool STATUS_READ_JTAG() {
    return (STATUSREG_JTAG_16_0 & STATUS_JTAG_16_MASK);
}

static inline void initJTAG(void) {
    INSTRUCTIONREG_JTAG_16_0 = 0;
    ACTIONREG_JTAG_16_0 = 0;
    WRITEFREG_JTAG_16_0 = 0;
    WRITEPREG_JTAG_16_0 = 0;
    READREG_JTAG_16_0 = 0;
    CONTROLREG_JTAG_16_0 = 0;
    STATUSREG_JTAG_16_0 = 0;
}

#endif
```

Listing 1: Header file of the *APB-Mopdule's* HAL generated by MetaFirm.

JTAG_HAL.c

```
// AUTO GENERATED CODE //
#include "JTAG_HAL.h"

JTAG_HAL_Config JTAG_HAL = {
  INSTRUCTION_WRITE_JTAG,
  INSTRUCTION_READ_JTAG,
  ACTION_WRITE_JTAG,
  WRITEF_WRITE_JTAG,
  WRITEP_WRITE_JTAG,
  READ_READ_JTAG,
  CONTROL_WRITE_JTAG,
  CONTROL_READ_JTAG, STATUS_READ_JTAG
};
```

Listing 2: Source file of the *APB-Mopdule's* HAL generated by MetaFirm.

A.2 Basic Driver

JTAG.h

```
#ifndef _JTAG_H_
#define _JTAG_H_

// AUTO GENERATED CODE //
/*****************************************************************************************************/
/**
 * @file     JTAG.h
 * @author   pircher
 * @date     11:07:02    27/12/2021
 * @version  1.0
 * Authors: Clemens Pircher
 * Driver for APB-Module to control attached IJTAG-Module
*/
#include "BitfieldOperations.h"
#include "JTAG_HAL.h"
#include "stdint.h"
#include "SystemConfig.h"
#include "types.h"

/**
 * Select IJTAG TDR
**/
void JTAG_Select_IJTAG();

/**
 * Select EXCFG TDR
**/
void JTAG_Select_EXCFG();

/**
 * Select SIB TDR
**/
void JTAG_Select_SIB();

/**
 * Perform Capture action
**/
void JTAG_Capture();

/**
 * Perform Update action
**/
void JTAG_Update();

/**
 * Full Shift
 * @param  data ['data to be shifted']
**/
void JTAG_ShiftF(uint16_t data);

/**
 * Partial Shift with Padding
 * @param  data ['data to be shifted']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftP(uint16_t data, uint8_t count);

/**
 * Shift specified number of bits from a byte array
```

```c
 * @param  wdata ['pointer to the input data']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftData(uint16_t *wdata, uint16_t count);

/**
 * Full Shift
 * @param  data ['data to be shifted']
**/
uint16_t JTAG_ShiftF_R(uint16_t data);

/**
 * Partial Shift with Padding
 * @param  data ['data to be shifted']
 * @param  count ['number of bits to be shifted']
**/
uint16_t JTAG_ShiftP_R(uint16_t data, uint8_t count);

/**
 * Shift specified number of bits from a byte array
 * @param  wdata ['pointer to the input data']
 * @param  rdata ['pointer for the output data']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftData_R(uint16_t *wdata, uint16_t *rdata, uint16_t count);

/**
 * Wait for ongoing operations to finish
**/
void JTAG_Wait();

```

Listing 3: Basic functions in header file of the *IJTAG-Module*-specific driver for the *APB-Mopdule* generated with the presented Firmware generation flow.

JTAG.c

```c
// AUTO GENERATED CODE //
/*****************************************************************************************************/
/**
 * @file     JTAG.c
 * @author   pircher
 * @date     11:07:02    27/12/2021
 * @version  1.0
 * Authors: Clemens Pircher
 * Driver for APB-Module to control attached IJTAG-Module
*/
#include "JTAG.h"

/**
 * Select IJTAG TDR
**/
void JTAG_Select_IJTAG(){
    JTAG_HAL.INSTRUCTION_WRITE(8);}

/**
 * Select EXCFG TDR
**/
void JTAG_Select_EXCFG(){
    JTAG_HAL.INSTRUCTION_WRITE(10);}

/**
 * Select SIB TDR
**/
void JTAG_Select_SIB(){
    JTAG_HAL.INSTRUCTION_WRITE(9);}

/**
 * Perform Capture action
**/
void JTAG_Capture(){
    JTAG_HAL.ACTION_WRITE(1);}

/**
 * Perform Update action
**/
void JTAG_Update(){
    JTAG_HAL.ACTION_WRITE(2);}

/**
 * Full Shift
 * @param  data ['data to be shifted']
**/
void JTAG_ShiftF(uint16_t data){
    JTAG_HAL.WRITEP_WRITE(data);}

/**
 * Partial Shift with Padding
 * @param  data ['data to be shifted']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftP(uint16_t data, uint8_t count){
    uint16_t ret; /**< return variable*/;
    uint8_t pad_width; /**< number of padding bits required*/;
    pad_width = (16 - count);
    uint16_t data_padded; /**< padded data*/;
    data_padded = ((data << pad_width) | (1 << (pad_width - 1)));
    JTAG_HAL.WRITEP_WRITE(data_padded);
}

/**
 * Shift specified number of bits from a byte array
 * @param  wdata ['pointer to the input data']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftData(uint16_t *wdata, uint16_t count){
    uint16_t num_groups; /**< number of write transactions required*/;
    num_groups = (count >> 4);
    uint8_t last_count; /**< number of bits in the last transaction*/;
    last_count = (count & 0x0F);
    if (last_count){
        num_groups = (num_groups - 1);
    };
    uint16_t cur_group; /**< */;
    while((cur_group < num_groups)){
        JTAG_ShiftF(*(wdata+cur_group));
        (cur_group++);
    };
    if (last_count){
        JTAG_ShiftP(*(wdata+cur_group), last_count);
    };
}

/**
 * Full Shift
 * @param  data ['data to be shifted']
**/
uint16_t JTAG_ShiftF_R(uint16_t data){
    JTAG_HAL.WRITEP_WRITE(data);
    return JTAG_HAL.READ_READ();
}

/**
 * Partial Shift with Padding
 * @param  data ['data to be shifted']
 * @param  count ['number of bits to be shifted']
**/
uint16_t JTAG_ShiftP_R(uint16_t data, uint8_t count){
    uint16_t ret; /**< return variable*/;
    uint8_t pad_width; /**< number of padding bits required*/;
    pad_width = (16 - count);
    uint16_t data_padded; /**< padded data*/;
    data_padded = ((data << pad_width) | (1 << (pad_width - 1)));
    JTAG_HAL.WRITEP_WRITE(data_padded);
```

```c
    ret = (JTAG_HAL.READ_READ() >> pad_width);
    return ret;
}

/**
 * Shift specified number of bits from a byte array
 * @param  wdata ['pointer to the input data']
 * @param  rdata ['pointer for the output data']
 * @param  count ['number of bits to be shifted']
**/
void JTAG_ShiftData_R(uint16_t *wdata, uint16_t *rdata, uint16_t count){
    uint16_t num_groups; /**< number of write transactions required*/;
    num_groups = (count >> 4);
    uint8_t last_count; /**< number of bits in the last transaction*/;
    last_count = (count & 0x0F);
    if (last_count){
        num_groups = (num_groups - 1);
    };
    uint16_t cur_group; /**< */;
    while((cur_group < num_groups)){
        *(rdata+cur_group) = JTAG_ShiftF_R(*(wdata+cur_group));
        (cur_group++);
    };
    if (last_count){
        *(rdata+cur_group) = JTAG_ShiftP_R(*(wdata+cur_group), last_count);
    };
}

/**
 * Wait for ongoing operations to finish
**/
void JTAG_Wait(){
    while(JTAG_HAL.STATUS_READ()){
    };
}

```

Listing 4: Basic functions in source file of the *IJTAG-Module*-specific driver for the *APB-Mopdule* generated with the presented Firmware generation flow.

A.3 Custom Driver

JTAG.h

```c

/**
 * Sets a new trim value.
 * @param  trim_value ['Input variable trim_value (10 bits)']
**/
void set_trim_value(uint16_t trim_value);

/**
 * Sets a new trim value. (CONFIG ONLY)
**/
void set_trim_value_config();

/**
 * Gets the current comparator output.
**/
uint8_t get_comparator_result();

#endif
```

Listing 5: Cumstom functions in header file of the *IJTAG-Module*-specific driver for the *APB-Mopdule* generated with the presented Firmware generation flow.

JTAG.c

```c

/**
```

```
 * Sets a new trim value.
 * @param  trim_value ['Input variable trim_value (10 bits)']
**/
void set_trim_value(uint16_t trim_value){
    //+----------------------------+;
    //| Create all local variables |;
    //+----------------------------+;
    uint16_t _tempw; /**< */;
    uint16_t _tempr; /**< */;

    //+-------------------------+;
    //| Perform scan operations |;
    //+-------------------------+;
    //Select Subnetworks for Apply 0;
    JTAG_HAL.INSTRUCTION_WRITE(9);
    JTAG_HAL.WRITEP_WRITE(0b1000111111111111);
    JTAG_HAL.ACTION_WRITE(2);

    //Access TDRs for Apply 0;
    JTAG_HAL.INSTRUCTION_WRITE(8);
    _tempw = 0b0101111111111111;
    JTAG_HAL.WRITEP_WRITE(_tempw);
    JTAG_HAL.ACTION_WRITE(2);

    //Already selected Subnetworks for Apply 1;

    //Access TDRs for Apply 1;
    JTAG_HAL.ACTION_WRITE(1);
    _tempw = (0b0000000000000111 | (((trim_value & 1023) << 1) << 4));
    JTAG_HAL.WRITEP_WRITE(_tempw);
    JTAG_HAL.ACTION_WRITE(2);
}

/**
 * Sets a new trim value. (CONFIG ONLY)
**/
void set_trim_value_config(){
    //+--------------------------+;
    //| Apply all configurations |;
    //+--------------------------+;
    //Select Subnetworks;
    JTAG_HAL.INSTRUCTION_WRITE(9);
    JTAG_HAL.WRITEP_WRITE(0b1000111111111111);
    JTAG_HAL.ACTION_WRITE(2);
    //Enable all SIBs;
    JTAG_HAL.INSTRUCTION_WRITE(10);
    JTAG_HAL.WRITEP_WRITE(0b0101111111111111);
    JTAG_HAL.ACTION_WRITE(2);
    //Update Config TDRs and disable all SIBs;
    JTAG_HAL.WRITEP_WRITE(0b0000000000000111);
    JTAG_HAL.ACTION_WRITE(2);
}

/**
 * Gets the current comparator output.
**/
uint8_t get_comparator_result(){
    //+----------------------------+;
    //| Create all local variables |;
    //+----------------------------+;
    uint16_t _tempw; /**< */;
    uint16_t _tempr; /**< */;
    uint8_t result; /**< Output variable result (1 bits)*/;

    //+-------------------------+;
    //| Perform scan operations |;
    //+-------------------------+;
    //Select Subnetworks for Apply 0;
    JTAG_HAL.INSTRUCTION_WRITE(9);
    JTAG_HAL.WRITEP_WRITE(0b1000111111111111);
    JTAG_HAL.ACTION_WRITE(2);

    //Access TDRs for Apply 0;
    JTAG_HAL.INSTRUCTION_WRITE(8);
    _tempw = 0b1001111111111111;
    JTAG_HAL.WRITEP_WRITE(_tempw);
    JTAG_HAL.ACTION_WRITE(2);

    //Already selected Subnetworks for Apply 1;

    //Access TDRs for Apply 1;
    JTAG_HAL.ACTION_WRITE(1);
    _tempw = 0b0000111111111111;
    JTAG_HAL.WRITEP_WRITE(_tempw);
    _tempr = (JTAG_HAL.READ_READ() >> 13);
    JTAG_HAL.ACTION_WRITE(2);
    return result;
}
```

Listing 6: Custom functions in source file of the *IJTAG-Module*-specific driver for the *APB-Mopdule* generated with the presented Firmware generation flow.

A.4 Trim Routine

main.c

```c
/**
 * @file main.c
 * @author Clemens Pircher
 * @brief
 * @version 1.0
 * @date 2021-12-27
 */

#include "JTAG.h"

void main(void)
{
  uint16_t trim_min = 0;
  uint16_t trim_max = 1023;

  uint16_t trim_current;

    while (trim_max - trim_min > 2)
    {
      trim_current = (trim_min + trim_max) >> 1;
      set_trim_value(trim_current);
      set_trim_value_config();

      uint8_t result = get_comparator_result();

      if (result > 0)
        trim_max = trim_current;
      else
        trim_min = trim_current;
    }
}
```

Listing 7: The source code of the trim routine that makes use of the custom driver.