Doctoral Thesis

Verification issues of virtual components in system-on-a-chip (SOC) designs

Author(s):
Stadler, Manfred

Publication Date:
2000

Permanent Link:
https://doi.org/10.3929/ethz-a-004039249

Rights / License:
In Copyright - Non-Commercial Use Permitted
Verification Issues of Virtual Components in System-On-a-Chip (SOC) Designs

A dissertation submitted to the SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of Doctor of Technical Sciences

presented by MANFRED STADLER Dipl. El. Ing. ETH born June 21, 1968 citizen of Jonschwil SG

accepted on the recommendation of Prof. Dr. W. Fichtner, examiner Prof. Dr. B. Plattner, co-examiner

2000
Acknowledgments

I would like to thank my supervisor, Prof. Dr. Wolfgang Fichtner, for his overall support and for his faith in me and my work. I would also like to thank Prof. Dr. Bernhard Plattner for reading and commenting on my thesis.

Special thanks to Hubert Kaeslin and Norbert Felber for their encouragement and support during my research work as well as for their proofreading of and commitment to my thesis. I also want to express my gratitude to all my colleagues at the Integrated Systems Laboratory who contributed to the good working environment. In particular, I wish to thank Markus Thalmann and Thomas Röwer for their wide-ranging discussions with me. Furthermore, I want to thank the secretaries for taking care of the administration, Hanspeter Mathys and Hansjörg Gisler of the installations, Christoph Wicki of the computers, and the Microelectronics Design Center of the VLSI design tools.

In addition, I want to thank Hans-Rudolf Graf and Thomas Bürgel for their excellent contributions to my thesis.

I acknowledge the financial support of KTI, a Commission for technology and innovation supported by the Swiss Government and Siemens Schweiz AG.

Last but not least, special thanks to my parents for their understanding and tolerance - they have not seen much of me in recent years - and my love especially to Rita for her support and patience during the last five years.
Contents

Acknowledgments  i

Abstract  ix

Zusammenfassung  xi

1 Introduction  1

1.1 Motivation ................................. 1

1.2 Goals of this Work ............................. 3

1.3 Structure of the Thesis .......................... 3

2 Background 5

2.1 Introduction ................................ 5

2.2 Formal Verification Approaches ................. 6

2.2.1 Equivalence Checking ........................ 8

2.2.2 Model Checking ............................. 10

2.2.3 Theorem Proving ............................ 13

2.2.4 Common Limitations ........................ 14
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3</td>
<td>Dynamic Verification Techniques</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Software Simulation</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Hardware Prototyping</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Hardware In-Circuit Emulation</td>
<td>19</td>
</tr>
<tr>
<td>2.4</td>
<td>Test Vector Set</td>
<td>19</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Buildup of the Test Vector Set</td>
<td>19</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Test Vectors to Verify Common Functionality</td>
<td>20</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Test Vectors to Verify Corner Cases</td>
<td>21</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Random Test Vectors</td>
<td>22</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Test Vectors from Real-World Applications</td>
<td>24</td>
</tr>
<tr>
<td>2.4.6</td>
<td>High-Level Test Vector Generation</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Quality Measurement Metrics</td>
<td>27</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Bug Discovery Rate</td>
<td>28</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Low-Level Profilers</td>
<td>29</td>
</tr>
<tr>
<td>2.5.3</td>
<td>High-Level Profilers</td>
<td>29</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Assertion Checkers</td>
<td>30</td>
</tr>
<tr>
<td>2.6</td>
<td>Implications</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Test Benches</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Basic Test Bench Structure</td>
<td>33</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Reactive Test Benches</td>
<td>34</td>
</tr>
<tr>
<td>3.2</td>
<td>Simulation Frameworks</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Co-Simulation Framework</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Self-Checking Framework</td>
<td>39</td>
</tr>
</tbody>
</table>
4 SPACEMAN Virtual Component

4.1 Motivation to Use Programmable Virtual Components in SOC Design

4.2 The Integrated TRAnsmision SYstem

4.2.1 System Functionality

4.2.2 Data Organization and Frequency Adaptation Mechanisms

4.2.3 ITRACHIP System-On-a-Chip (SOC)

4.2.4 Implementation Variants for Data Processing

4.2.5 Processing-Event Scheduling Strategy

4.2.6 System Integration

4.3 The SPACEMAN ASIP

4.3.1 Architecture Evaluation

4.3.2 SPACEMAN Architecture

4.3.3 Instruction Set

4.3.4 Pipeline Hazards

4.3.5 VC Structure

4.4 Parameterization Properties

4.4.1 Qualitative Parameters

4.4.2 Quantitative Parameters

4.4.3 Interface Customization Parameters

4.5 Implementations
5 Functional Verification Flow for Virtual Components

5.1 Problems of and Requirements for Simulation-Based VC Verification

5.2 General Aspects of the Verification Flow

5.2.1 Test Vector Preparation

5.2.2 Generic Framework for VC Verification

5.2.3 Concurrent versus Consecutive Simulation

5.3 Generic Vector Generation

5.3.1 Stimuli Preparation

5.3.2 Model Execution

5.4 Vector Generation for SPACEMAN

5.4.1 Stimuli Assembly

5.4.2 Stimuli Download

5.4.3 Run-Time Stimuli Generation

5.4.4 Snoopers

5.4.5 Carbon Model

5.5 Generic Model Checking

5.5.1 Stimuli Preparation

5.5.2 Model Execution

5.6 Model Checking for SPACEMAN

5.6.1 Stimuli Preparation
6 Synchronization Issues 113

6.1 Global Simulation Control 114

6.2 Interface Synchronization Strategies 114

6.2.1 Cycle-true Output Triggering 115

6.2.2 Selective Triggering with One Strobe 117

6.2.3 Selective Triggering with Multiple Strobes 117

6.3 Interface-Specific Activity Scheduling 120

6.3.1 Interface Grouping 120

6.3.2 Synchronization Issues of Vector Generation 121

6.3.3 Synchronization Issues of Model Checking 123

6.4 Synchronization of Interrupt Launching 124

6.4.1 Interrupt Format 125

6.4.2 Interrupt Scheduler 126

6.5 Simulation Results 131

6.5.1 Simulation Run Times for Code Download 131

6.5.2 Benchmark Programs 132

6.5.3 Reference Vector Sizes 134

6.5.4 Run Times for Vector Generation 134

6.5.5 Run Times for Model Checking 136

6.5.6 Discussion 136
7 Verification Flow for a Multi-VC Design

7.1 System Test Vehicle ........................................... 139
7.2 Vector Generation ............................................. 142
  7.2.1 Separation of Data Path and Controller ............... 142
  7.2.2 Test Bench ................................................. 145
  7.2.3 Reference Vectors ......................................... 153
7.3 Model Checking ................................................ 153
  7.3.1 Test Bench ................................................. 154
7.4 Validation of the VC System Verification Flow .......... 156

8 Conclusions .................................................... 159

8.1 Main Achievements ........................................... 160
  8.1.1 Stand-alone VC Verification ............................ 160
  8.1.2 Verification of Multi-VC Systems ..................... 162
  8.1.3 Reusability ................................................ 162
8.2 Drawbacks .................................................... 163
8.3 Outlook and future research topics ....................... 164

A Instruction Set ................................................ 167

B Design Representation ......................................... 169

Bibliography ...................................................... 171

Curriculum Vitae ................................................ 179
Abstract

Functional verification of digital designs is a formidable and time-consuming task. The introduction of pre-developed, generally usable functional blocks, called virtual components (VC) for system-on-a-chip (SOC) designs further complicates verification because of the large variety of possible implementations. Every VC parameter set requires different verification data. Furthermore, VC users are not familiar with the VC internals. As a consequence, for reuse, it is mandatory to automatically derive test benches for every parameterization of the VC.

This research project explores a novel methodology to verify VCs with a dynamic, simulation-based approach. This methodology is validated by taking a highly parameterizable application-specific instruction-set processor (ASIP) as VC example. The verification flow is subdivided into two simulation runs: vector generation and model checking.

For vector generation, a behavioral model, the carbon model, has been developed. As opposed to the golden model which represents the translation of the specification into one specific simulation model, the carbon model is generically designed and is able to generate reference vectors for any possible configuration of the VC implementation. The carbon model is data-driven, i.e. triggered with stimuli vectors, and therefore abstracted from timing. Interactions between test bench and carbon model are performed with full-handshake protocols, thus enabling sequential executions which shorten simulation run times.

In model checking the outputs of the clock-driven, customized implementation model are compared to the reference vectors. To adapt the two different model triggering mechanisms, a novel synchronization methodology is in-
Introduced: “interface-specific activity scheduling”. This methodology groups the primary outputs of the VC into several interfaces. Thereby, every interface triggers its comparator in the test bench only on request. Thus, the correct partial ordering of the output data rather than cycle-true correctness is checked. This approach efficiently reduces the number of reference vectors without loosing relevant information for the functional verification.

From the experience gained with the stand-alone VC verification, a methodology is elaborated to verify a system composed of several VCs, including an ASIP VC. Thereby, the carbon system is executed in a loop by sequentially invoking all data paths. An ASIP triggered on request is assigned to some data paths. Data exchange among carbon models and between carbon models and the test bench are performed with full-handshake protocols. To warrant synchronization between the carbon model and the implementation model of the ASIP, reference vectors are stored in different data bases, separated for data paths.

This thesis introduces a self-adaptable test bench which forms a verification platform for all different parameterization possibilities of the VC implementation model without the necessity of manual adaptations. The test bench is reusable, on the one hand to develop a system test bench if the VC is embedded into a multi-VC design and on the other hand to elaborate test benches for the functional verification of other VCs. The synchronization methodology “interface-specific activity scheduling” enables the test bench to handle varying latencies of different VC parameterizations. Furthermore, this methodology reduces the number of reference vectors and results in shorter simulation run-times without deteriorating the verification quality.
Zusammenfassung


Für die Modellprüfung werden die Ausgangsdaten des taktgesteuerten, kundenspezifischen VC Realisierungsmodells gegen die Referenzvektoren verglichen. Um die zwei unterschiedlichen Verfahren der Modellaktivierung aufeinander abzustimmen, wurde ein neuartiger Synchronisierungsmechanismus eingeführt: "interface-specific activity scheduling". In diesem Ansatz werden die primären Ausgänge der VC in mehrere, unabhängige Schnittstellen unterteilt, wobei jeder dieser Schnittstellen eine Vergleichsschaltung in der Testumgebung zugeordnet ist, die nur aktiviert wird, falls für die Funktionalität relevante Daten anliegen. Auf diese Weise wird vielmehr die korrekte Abfolge der Ausgangsdaten überprüft, als deren Richtigkeit in jedem einzelnen Zyklus. Dies ermöglicht es die Anzahl der Referenzvektoren effizient zu verkleinern ohne dabei wichtige Informationen für die funktionelle Verifikation zu verlieren.


In dieser Dissertation wird eine selbstanpassende Testumgebung vorgestellt, welche eine Plattform für die Verifikation von unterschiedlichen Parametrisierungsformen eines VC Realisierungsmodells bildet, ohne dass dabei manuelle Anpassungen vorgenommen werden müssen. Zudem ist die Testumgebung wiederverwendbar, auf der einen Seite um eine Systemtestumgebung zu entwerfen, wenn die VC in ein System bestehend aus mehreren VCs eingebettet wird und auf der anderen Seite wenn Testumgebungen für die funktionelle Verifikation von anderen VCs entworfen werden müssen. Mit der Synchronisierungsmethodik "interface-specific activity scheduling" kann die Testumgebung variierende Latenzen handhaben, welche von unterschiedlichen Parametrisierungsformen der VC herrühren. Des weiteren reduziert diese Methodik die Anzahl der Referenzvektoren, was die Ausführungszeit der Simulationen verkürzt, ohne dass dabei die Qualität der funktionellen Verifikation beeinträchtigt wird.
1

Introduction

1.1 Motivation

Ongoing improvements of semiconductor process technology result in progressively smaller feature sizes and continually better yields. This trend is reflected in increasing area per die. Thus, circuit designers can pack ever larger amounts of functionality in one chip. As a result, digital designs are progressively becoming faster and more complex, resulting in systems-on-a-chip (SOC). Over the last few years, VLSI circuit complexity increases constantly with Moore’s law, e.g. the number of transistors doubles every eighteen months [sia97]. This trend is in deep contrast to competitive pressure towards shorter product development cycles and faster time-to-market. The high circuit complexities (overall gate count, massive node/pin ratio, large number of test vectors required) makes the task of verifying that the design is free of functional bugs increasingly difficult. In general, larger designs tend to have
more concurrent interactions, all of which require testing. According to estimates in [sia97], functional verification presently already requires between one third and one half of the total design effort.

The SOC era asks for significant changes in design and verification methodology for application specific integrated circuits (ASIC). Today's ASIC design is often still characterized by writing the register transfer level (RTL) code from scratch in every new project. An efficient approach to enhance productivity is to come up with fairly general building blocks, from which various ASICs are then composed at the architecture level, essentially by putting together such predeveloped blocks or, as they are usually called, virtual components (VC) or intellectual properties (IP). However, to warrant multiple reuse of a VC in a variety of applications, its functionality and its interfaces must be made amenable to extensive parameterization and customization [PVvM97][GZ97]. Considerable design time reduction and greatly increased productivity are expected to result from this approach.

Although every description level from behavioral to physical could be used for VCs, HDL on RTL level is most commonly used. Its obvious advantages are easy parameterization, customization and high portability. While these features simplify the construction of very complex systems considerably, this is not true at all for verification of the resulting circuit.

The incorporation of virtual components makes it even more difficult to ensure that SOC circuits indeed function as intended. Scalability and customization features of VC modules lead to a large variety of possible implementations, each of which requires different verification data. Furthermore, unknown embedding environments and interface conditions as well as VC specific characteristics, such as the missing or unaffordable knowledge on internal details of the VCs, and unforeseen interactions between various parameter settings all contribute to this problem.

Due to the fact that the designer should not need to know the internals of the VC that he is going to use (on the one hand to enable easy reuse and on the other hand to protect the VC provider from illegal copying of VC implementation details) it is mandatory to provide a test bench which is able to deal with such problems. Reuse of VCs makes little sense as long as their test benches need to be manually rewritten for every application which would require knowledge of VC internals. To prevent functional verification from consuming a disproportionate amount of engineering time, it is essential to provide VC users with test benches that are self-configurable, and therefore
1.2 Goals of this Work

To contribute a solution, the following goals have been formulated for this work:

- Establish an overview of the basic methodologies for functional verification of digital circuits, their characteristics, their areas of application, and their limitations.

- Introduce a telecommunication SOC with an embedded application-specific instruction-set processor which will be used as example to validate our novel VC verification solution.

- Elaborate mechanisms which enable the test engineer the reuse of test benches for model descriptions at different abstraction levels with different timing concepts and process triggering mechanisms, such as data-driven, clock-driven, or event-driven.

- Develop a test bench for a VC, which is self-adaptable to different parameter settings of the implementation model. Thereby, constraints such as easy portability and platform independence on the one hand and short verification cycles on the other hand have to be considered.

- Based on the above verification environment, which is developed for a stand-alone VC, try to find a systematic methodology to functionally verify a design, consisting of several VCs, by reusing the test benches of the single components.

1.3 Structure of the Thesis

As a starting point, today’s commonly used approaches for functional verification of digital circuits are summarized in chapter 2. Benefits and drawbacks of various methodologies and their limitations are discussed.

Chapter 3 introduces the basic test bench structures which are used for functional verification by simulation.
SPACEMAN, an application-specific instruction-set processor (ASIP) which is a highly parameterizable VC forms the test vehicle for validating our VC verification approach. In chapter 4 its architecture as well as its parameterization properties are introduced. Furthermore, the telecommunication system embedding the ASIP is introduced in detail. Parts of this system are used in chapter 7 to elaborate and validate the multi-VC verification flow.

In chapter 5 a reusable test bench to verify SPACEMAN is introduced. The benefits of partitioning the verification process in two phases, namely vector generation and model checking are pointed out. Thereby, a new class of reference models is introduced, referred to as carbon model, which serves as the golden model\(^1\) for all different customizations of the VC. Furthermore, the influence of qualitative VC parameters on test vector generation is considered.

If a carbon behavioral model is used as a reference, the verification of the model under test by simulation causes the synchronization of the two models as each model is described in a different level of an abstraction. Chapter 6 introduces a solution, called “interface-specific activity scheduling”.

In chapter 7 a methodology is elaborated to verify, by reusing the test benches of the single components, whether a system consisting of several VCs is functionally correct.

Finally, the main results of the thesis are summarized and conclusions are drawn in chapter 8.

\(^1\) A golden model is the translation of the specification into a programmable language (VHDL, Verilog, or C) which serves as reference model for the functional verification of the implementation model.
In this chapter today's most common approaches for functional verification of integrated circuits (IC) are introduced. The benefits and drawbacks of the different techniques and their limitations with regard to their application are discussed.

2.1 Introduction

Functional verification deals with the question of whether the design functions correctly from a purely logical point of view. Electrical and timing aspects are not taken into consideration.

Fig. 2.1 gives an overview of different methodologies to verify functional correctness of digital circuits. Besides informal methods based on human reasoning without stringent mathematical formalism such as inspection of wave-
forms, code inspection or design review [Bal95], dynamic techniques in a wider sense such as circuit simulation, prototyping or in-circuit emulation have been the only approaches for a long time.

Improvements in semiconductor technology have increased not only the speed and size of integrated circuits, but also their complexity. While design tools such as synthesizers or place & routers can keep up with this rapid development, there is a shortage of functional verification and testing tools because of insufficient automation. Exhaustive verification is clearly not an option.

To resolve the verification bottleneck, researchers have introduced novel techniques based on formal verification in recent years.

![Figure 2.1: General Overview of Different Verification Techniques](image)

**2.2 Formal Verification Approaches**

Formal verification is a static technique to verify a circuit [Yoe90]. With mathematical approaches, 100% accuracy and coverage are theoretically obtained [PSB+96].

This predictability of verification completeness is the major benefit of formal verification techniques in contrast to simulation based techniques which can never categorically declare a complex design functionally correct due to
the unfeasibility of exhaustive simulation. Therefore, the application of formal techniques to verify the functionality of a design provides a powerful and definitive statement.

Formal verification techniques are subdivided in two categories, depending on the purpose of verification: design validation and implementation verification [Sar96].

- Design validation is the process of verifying the correctness of a design with respect to a certain specified behavior derived from the specification; i.e. design validation deals with building the right model [Bal95].

- Implementation verification is the process of discovering whether a design is equivalent to the original design after refinement or transformation, e.g. an algorithm into an RTL representation or an RTL representation into gates; i.e. implementation verification deals with building the model correctly [Bal95].

To perform formal verification, there exist three different methods:

- Equivalence checking
- Model checking
- Theorem proving

Tab. 2.1 gives an overview on how purposes and methods combine.

<table>
<thead>
<tr>
<th>Formal Method</th>
<th>Verification Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Design Validation</td>
</tr>
<tr>
<td>Equivalence Checking</td>
<td>no</td>
</tr>
<tr>
<td>Model Checking</td>
<td>yes</td>
</tr>
<tr>
<td>Theorem Proving</td>
<td>yes</td>
</tr>
</tbody>
</table>

*Table 2.1: Classification of Formal Verification Techniques*

In the next sections the three different formal methods are discussed in detail. Appendix B gives an introduction in the commonly used design representation on which these formal techniques are based.
2.2.1 Equivalence Checking

Equivalence checking is a formal technique to verify a circuit implementation. As a matter of principle this method is widely automated and is the most practical and least difficult of the three formal techniques. Due to these characteristics, equivalence checking is today's most popular approach. However, in reality there exist a lot of exceptional cases which necessitate interactions by the user.

Principally, equivalence checking represents the formalized static counterpart of the co-simulation approach in which the outputs of a golden reference model are compared to the outputs of the model under test, the MUT (see chapter 3). The difference compared to co-simulation approach is that equivalence checking attempts to prove the equivalence for all possible input values and not only for a selective subset.

For sequential designs the following procedure is applied (see also fig. 2.2):

- First, internal state registers, \( S_{xy} \), and primary outputs, \( Out_{xy} \) are identified in both models, so-called key points. Then, each key point of one model is mapped to the corresponding key point in the other model.

- In a second step, logic equations between two consecutive key points are calculated for both models, \( L_{xy} \). Every equation of the golden model is then compared to the corresponding equation in the implementation model (logic checking).

- If all key points of one model have a counterpart in the other model and all logic comparisons are successful, the two models are functionally equivalent. However, the converse is not correct. Two designs can have the same sequential behavior even though their key points do not correspond, e.g. if they use different state encodings or different register locations.

Strictly combinational circuits are compared in a similar way, except that the first step is omitted: key point identification and matching.

Logic checking of very large combinational nets is inherently a difficult problem having the worst case complexity, that is \( NP \)-hard [PSB+96]. However, many commercial designs for which equivalence checking is used con-
2.2 Formal Verification Approaches

Golden Model

Model Under Test

Key Points = \{S_{11}, S_{12}, S_{13}, Out_{12}\}

Key Point Mapping

Logic Equations

\begin{align*}
S_{11} & : L_{11} = f(In_{11}) \\
S_{12} & : L_{12} = f(In_{12}) \\
S_{13} & : L_{13} = f(S_{1}) \\
Out_{12} & : L_{14} = f(In_{11}, S_{12})
\end{align*}

\begin{align*}
S_{21} & : L_{21} = f(In_{21}) \\
S_{22} & : L_{22} = f(In_{22}) \\
S_{23} & : L_{23} = f(S_{2}) \\
Out_{22} & : L_{24} = f(In_{21}, S_{22})
\end{align*}

\textbf{Figure 2.2: Equivalence Checking for Sequential Models}

Consist of two models with quite a few equivalent intermediate subnets\(^1\) which are separated by cutpoints. If such isomorph subnets exist, it is possible to perform logic checking incrementally, a procedure which usually reduces the complexity and therefore the costs dramatically (see fig. 2.3).

Combinational Net

\begin{align*}
In_1 & \rightarrow Subnet_1 \rightarrow CP_{11} \rightarrow Subnet_2 \rightarrow Out_1 \\
In_1 & \rightarrow Subnet_3 \rightarrow CP_{12} \rightarrow Subnet_4 \rightarrow Out_2
\end{align*}

\begin{align*}
CP_{11} & : Subnet_1 = f(In_1) \\
Out_1 & : Subnet_2 = f(CP_{11}) \\
CP_{12} & : Subnet_3 = f(In_1) \\
Out_2 & : Subnet_4 = f(CP_{12}, CP_{12})
\end{align*}

\textbf{Figure 2.3: Incremental Equivalence Checking for Combinational Nets}

There are two commonly used commercial methods for equivalence checking of large designs that assume the availability of intermediate cutpoints: BDD based methods [BT89] and ATPG search algorithm based methods [Bra93].

---

\(^1\)Two intermediate subnets are equivalent from the viewpoint of equivalence checking if every input pattern causes the same logic values at the outputs of the subnets.
**Discussion**

Nowadays, equivalence checking fits well to compare two structurally almost equivalent models, i.e. one model is derived through design transformations from the other model (e.g. RTL vs. synthesized gate-level netlist, gate-level netlist vs. gate-level netlist after technology mapping, scan insertion, manual optimization, or engineering change orders) [Nor96].

However, equivalence checking is often far from being fully automated. For instance, using retiming to optimize throughput has the effect that registers and key points do not match, not even their numbers do. Retiming reallocates latches or flip-flops from the inputs of a logic gate to the output of the gate or vice versa and thus modifies the logic equation between key signals. In [HCC96] an equivalence checker is introduced which is able to handle comparisons of retimed circuits. Another limitation occurs when flip-flops in a design library are modeled as a master and a slave latch. In the gate-level representation of a circuit the equivalence checker extracts two different key points for one flip-flop whereas in the RTL model the counterpart is represented by one variable and consequently just one key point is extracted.

So far equivalence checking between a behavioral and an RTL circuit description has provided very poor results. Usually, behavioral models are largely abstracted by using high level HDL constructs or written in a non-hardware-related language while RTL models are synthesizable and therefore an image of the final hardware. The different degree of abstraction results in different internal state representations of the models which prevent successful key signal mapping by the equivalence checker.

EDA vendors make every effort to improve and further automate equivalence checking by introducing, for instance, the sequential equivalence checking which allows to verify two designs with different state encodings or different numbers of internal states.

### 2.2.2 Model Checking

Model checking is a formal technique to validate a design. It is a fully automated approach which operates on OBDD circuit representation (see appendix B).
The strategy of model checking is to take a description of a design and find all the possible states that it can ever reach. This state space exploration provides a straightforward method of exploring the range of possible behaviors of a system and of checking whether any of the results violates a property. The properties are simply statements and formalizations of important behaviors which are described in the specification of the design. This technique is powerful since it checks every reachable state in a design for an error. In addition, one of the by-products of the state exploration is a sequence from the initial state to any error state, which helps debugging greatly. When using model checking, additional assumptions with regard to the inputs of the model have to be defined to discard forbidden or illegal input sequences.

In [SBFH97] a popular model checking approach is introduced. This technique is based upon automation and employs BDDs of computation-tree-logic (CTL). The properties are specified with temporal formulas using temporal logic operators [McM93]. Basic temporal logic operators (CTL formulas) specify those computation paths which are possible in a computation tree. A CTL formula is defined true or false in a given state. The class of CTL formulas is defined recursively and has the following structure: it starts with the path quantifier (see tab. 2.2), followed by the state quantifier (see tab. 2.2), followed by a CTL formula.

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path</td>
<td>A</td>
<td>∀ paths</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>∃ paths</td>
</tr>
<tr>
<td>State</td>
<td>F</td>
<td>∀ states</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td>∃ states</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>⊨ state</td>
</tr>
</tbody>
</table>

**Table 2.2:** CTL State Quantifiers and Path Quantifiers

There are two very important classes of CTL formulas: safety formulas specify properties that must hold in the complete state space of the circuit, and liveness formulas describe the absence of deadlocks in the circuit.

An example of a CTL safety formula is shown in fig. 2.4. AG(P) is true at exactly the state when, for all computations beginning at this state, the propo-

---

2A computation tree is derived from a transition diagram. It is a graph structure which is unwound into an infinite tree rooted at the initial state. The paths in this tree represent all possible computations of the circuit being modeled.
osition P is true for each state of the computation. Thus, $AG(!error)$ means that error state may never occur. For the circuit with no input constraints the property is false, but when the assumption, $(i[k] \neq 1 \ \forall \ k \in \{1, 3, 5, \ldots\})$, is defined on the input $i$ the property is true.

![Finite State Machine](image)

**Figure 2.4: Example of a Safety Formula in CTL**

In fig. 2.5 an example of a liveness formula is illustrated. $EF(Q)$ is true at a state if there exists a computation beginning at the state in which formula $Q$ is eventually true. Therefore, the recursive combination $AG(EF(init))$ means that for all reachable states, there is a path to init state. This property is true for the circuit on the left hand side, but false for the circuit on the right hand side (once it enters state $s3$ the circuit stays in $s3$ forever).

![Finite State Machine](image)

**Figure 2.5: Examples of a Liveness Formula in CTL**

Several case studies have shown that the use of CTL based model checking is a powerful approach to find bugs in a design [BBDEL96][Eri96].


## 2.2 Formal Verification Approaches

### Discussion

Model checking can validate the behaviors which a design might exhibit to the specification provided in the form of logic properties. The technique fits especially well to verify control-intensive designs which are commonly described in terms of finite state machines (FSM).

Model checkers have the capability to automatically produce error tracks when the property under check is found to fail. This is an essential feature to support efficient design debugging. Another positive effect is that properties are reusable across different designs and different abstraction levels.

A drawback of model checking is its bad fitting into current design practices. For instance, verifying a sub block of a design necessitates a detailed specification of its interface behavior to guarantee that only legal inputs are checked. Real designs almost never have up-to-date, detailed specifications of interfaces between low-level blocks. Thus, preparing the verification environment would require painful reverse engineering.

### 2.2.3 Theorem Proving

Theorem proving \([BM79][Gor88][ORR^+]\) is a formal technique to validate a design. It is a poorly automated approach, but potentially very powerful and highly expressive.

Theorem proving shows that a formal description of a design satisfies a set of properties. A theorem prover takes a description of a design as a set of axioms which state properties of the design. It then accepts queries about other properties which the test engineer wants to know, for example, whether the result of an operation can ever be wrong. The theorem prover then uses inference rules to determine the answer of this query. This has been done in \([SM95]\).

### Discussion

Theorem proving is a rarely used formal verification technique. On the one hand it is difficult to use because test engineers have to be familiar with the
logic language and theorem proving techniques, on the other hand an application cycle of theorem proving evolves generally slower than a normal product design cycle, therefore even just keeping up with the project development schedule is a problem [Kur97]. Furthermore, the fact that theorem provers do not automatically produce error tracks by construction makes debugging difficult. Thus, this technique is very unpopular among test engineers.

2.2.4 Common Limitations

The most important drawback of formal verification techniques is the state space explosion of the OBDDs (see appendix B). Each additional bit which is added to the state description of the model potentially doubles the state space. Consequently, such a representation is inapplicable for large designs and therefore formal verification techniques are not feasible.

The characteristic of all formal techniques is to verify only the correct logical functionality. The model under test representation is abstracted from timing information and consequently its verification is disregarded. Nevertheless, to provide a complete verification strategy, i.e. checking of functionality and timing, formal verification has to be used in conjunction with static timing analysis tools or dynamic timing simulators.

Functionality within a design which is timing-dependent such as an oscillator or interconnection circuits between two asynchronous parts can not be handled by the mathematical representations on which formal verification algorithms are built. Therefore, these parts have to be isolated and verified with other techniques, e.g. event-driven simulation.

2.3 Dynamic Verification Techniques

Functional verification by simulation is the dynamic counterpart to the static formal methodology. Simulation-based approaches can be subdivided into two different frameworks:

- Software Simulation
- Hardware Operation
### 2.3.1 Software Simulation

The first framework are software simulations of circuit models which are described in a hardware-description-language (e.g. VHDL or Verilog) or a high-level programming language (e.g. C/C++). Software simulations are executed on workstations using dedicated simulators [Synb][Tec]. They are applicable for models with any degree of abstraction (behavioral, RTL or gate-level). Thereby, the concurrent data processing of the model in the final hardware implementation is emulated with the event queue mechanism\(^3\).

Depending on the accuracy which is demanded by the test engineer and the circuit type (synchronous or asynchronous), different kinds of software simulation methodologies are used:

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Level of Abstraction</th>
<th>Concept of Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event-Driven</td>
<td>Gates</td>
<td>Delays, Glitches</td>
</tr>
<tr>
<td>Cycle-Based</td>
<td>RTL, Architecture</td>
<td>Partial Ordering Relationships</td>
</tr>
<tr>
<td>Data-Flow</td>
<td>Behavioral, Algorithmic</td>
<td>Throughput, Latency</td>
</tr>
</tbody>
</table>

**Table 2.3: Classification of Simulation Methodologies**

The best accuracy is achieved with *event-driven* simulation. Event-driven simulation evaluates and interprets all signal transactions. This characteristic makes the technique feasible for the verification of any digital circuit type. However, especially in strictly synchronous design style, transitions between two active clock edges such as glitches have no influence on the functionality, but slow down simulation speed unnecessarily.

In recent years, especially for regression simulation\(^4\), functional verification is separated from timing verification which is left to static timing analysis tools. This circumstance makes it possible to use *cycle-based* simulation such

---

\(^3\)In a simplified model the event queue can be seen as a list where entries are arranged according to their virtual simulation time of occurrence. Every entry in the queue is referred to a transaction. Concurrent transactions are separated by \(\delta\)-delays, an infinitesimally small lapse of time greater than zero. This partition enables the event queue to emulate the concurrent circuit behavior with sequential processing in several iteration steps without advance in the virtual simulation time [Kae99].

\(^4\)When a bug is found during simulation, the design has to be modified to fix it. Then, all previous tests must be re-run to make sure that the change did not invalidate the previous tests. This procedure is called regression simulation.
as [Qui]. The characteristic of cycle-based simulation is that results are calculated only after relevant transitions of the clock signals. Thus, all events caused by inter-phase timing are suppressed. In this technique value calculation has also been simplified: on the one hand by restricting the number of state representations of binary signals to two (i.e. low and high, instead of multiple\textsuperscript{5}) and on the other hand by limiting the supported HDL language subset.

\textit{Data-flow} simulation is used for an abstract behavioral representation of a design to verify algorithms or architectures and to compare implementation variants. The area of application is rather to check system performance such as throughput than obtaining a high functional accuracy. Commercial examples are instruction-set simulators such as [Cor] or system simulators such as [Syna].

\textbf{Discussion}

Cycle-based simulations are faster than their event-driven counterpart by an order of magnitude. However, their application is restricted to synchronous circuits. Thus, to keep the high simulation performance, analog and asynchronous parts have to be isolated and verified separately with event-driven simulation.

Due to the restriction of cycle-based simulation on two-state-logic, the correct initialization sequence, i.e. the transformation from an unknown state into a well defined start-up state, has also to be verified by using event-driven simulation, before changing to cycle-base simulation for regression testing.

Another characteristic of cycle-based simulation is that for multi-processor simulation load scheduling can be performed statically due to deterministic value evaluation, compared to the non predictable process activities in event-driven simulation.

\textsuperscript{5}VHDL type \texttt{std\_logic} which is the common representation of binary signals during gate-level, logic simulation has the following nine states: 'U': uninitialized, 'X': forcing unknown, '0': forcing zero, '1': forcing one, 'Z': high impedance, 'W': weak unknown, 'L': weak zero, 'H': weak one, and '-': don't care
Operating Principles of HDL Simulators

As mentioned above different events are processed strictly sequentially in software simulation; this results in a bottleneck for large designs. Therefore, new operating principles for HDL simulators have been introduced to speed-up simulation.

The first generation of HDL simulators translates HDL source into pseudo code which is interpreted by the simulator kernel during simulation execution.

By changing from interpreted simulation to processor specific compiled program versions simulation performance is enhanced. Two different approaches can be found in commercial simulators. Either HDL source is translated into C, compiled into machine code on the target machine and finally linked to the simulator kernel, or HDL source is directly compiled to machine code (native compiled code). With these approaches a simulation speed-up of an order of magnitude is achieved.

Another approach to increase simulation speed is based on replication of hardware resources. In hardware acceleration [Sys][Inc] a design is downloaded to a network of custom processors tailored to the purpose after elaboration. This approach scales down simulation execution time by up to two orders of magnitude compared to the execution on a workstation. A related approach is parallel computation by task scheduling on multiprocessor systems or server ranches [SM]. This approach obtains good results as long as inter task data exchange is small.

2.3.2 Hardware Prototyping

Hardware prototyping is the high performance simulation methodology to verify a circuit. In this technique the MUT is first synthesized before being downloaded into dedicated hardware, mostly an FPGA board. Then, the “hardware prototype” is simulated and checked for functional correctness.

In hardware prototyping several approaches of test benches are commonly used (see fig. 2.6).

(a) The test bench is written in software, usually in a behavioral style, to obtain sufficient performance and is executed on a workstation. It interacts
with the emulated design via a fast link and dedicated bridging software to avoid a bottleneck in data exchange between emulation board and host computer.

(b) Stimuli vectors are precomputed and downloaded from the workstation into the vector memory of the emulator prior to simulation start. This approach is practical for MUT verification using just a small number of different stimuli vectors, such as a microprocessor whose stimuli are program code sequences compiled and assembled before execution.

(c) The test bench is written in synthesizable HDL code and downloaded together with the MUT into the reprogrammable hardware of the emulator. In this approach stimuli application and MUT response analysis is also performed concurrently resulting in an additional simulation speed-up.

![Diagram of test benches used in hardware prototyping](image)

**Figure 2.6: Test Benches Used in Hardware Prototyping**

**Discussion**

In spite of concurrent execution FPGA hardware prototyping achieves not at all the simulation performance of the final ASIC integration. Usually, operation frequency has to be scaled down when using generic and reprogrammable emulation hardware. However, the benefit of concurrent data processing results nevertheless in a simulation performance in the same order of magnitude as the final ASIC integration.
2.3.3 Hardware In-Circuit Emulation

In several applications the environment to embed the MUT already exists in hardware (e.g. when an ASIC has to be redesigned). To avoid time intensive development of the test bench, the existing hardware environment is being used and operates as a test bench. Thereby, the MUT is either simulated on a workstation and its inputs and outputs are connected to the existing hardware [K+98] or the MUT is plugged directly into the environment as synthesized emulation hardware prototype.

2.4 Test Vector Set

When using simulation to verify the MUT, test vectors have to be prepared to stimulate the MUT and to serve as references when checking the responses of the MUT on functional correctness, respectively.

On the one hand exhaustive simulation for large designs is much too time-consuming and therefore unrealistic and not feasible. On the other hand detecting failures in products already delivered to customers is a very cost-expensive affair [Tak94]. This antagonism makes it mandatory to choose the “right” test vector set as well as to get metrics that determine when the quality of a design is “sufficient” in respect to functional correctness.

In the remaining part of this chapter commonly used approaches for these requirements are introduced.

2.4.1 Buildup of the Test Vector Set

In design verification by simulation, the quality of the result depends mostly on the chosen test vectors. A bug within a design is only found if dedicated test vectors stimulate and exercise the erroneous logic.

Hence, one of the most difficult tasks is to select an adequate test vector set which obtains a high fault coverage and consequently guarantees a good verification quality. This can be achieved best if the test vector set is composed of several, different parts, namely:
• Test vectors to verify common functionality of the MUT
• Constraint-based test vectors which stress corner cases, worst case scenarios, and exception handling
• Random test vectors
• Test vectors collected from real-world applications
• Test vectors produced by high-level generators

to be discussed in sections 2.4.2 to 2.4.6.

2.4.2 Test Vectors to Verify Common Functionality

Motivation

The first step in functional verification process is to check the core functionality of a design. For instance, to verify a microprocessor, test vectors which stress every instruction in different modes have to be prepared. This verification phase gives a first indication whether the design works correctly in normal operation.

Realization

Test vectors to verify common functionality are usually produced by hand. To enable a systematic approach a test plan is written defining what needs to be verified [T+98]. Thereby, the different test cases are derived from the design specification. Guided by the test plan, test engineers devise vectors which stress the design towards a particular test case.

Discussion

A big risk in test vector generation by hand is that important behavior of the design is overlooked and never tested. A test plan leads to a more structured approach and helps to improve verification quality. However, test cases which are not defined in the plan are consequently not verified. This can partly
be avoided if besides test engineers design engineers which have a detailed knowledge of the design implementation are also involved in the test plan creation, at least for review. Other drawbacks of manual test vector generation are that the quality of the vectors depends on the day-to-day performance of the test engineer creating it and the fact that vector generation is very expendable. Thus, manual generated test vectors are only useful for a first rudimentary test of the design.

2.4.3 Test Vectors to Verify Corner Cases

Motivation

Once the design is stabilized and the bug rate is declined, the design can be stressed towards exceptional situations. When verifying a microprocessor, such situations may be: stack over-/underflow, data dependencies or division by zero. Consequently, this phase of verification checks whether the design handles exceptions correctly.

Realization

Producing good quality test vectors to check exceptional events is even a difficult and time-consuming task, because the test engineer must have a detailed knowledge of the design implementation. For large designs it is not manually feasible to think of all possible interactions of two or more exceptional situations and thereafter find dedicated test vectors. For this reason generators directed by test engineers are used to produce test vectors.

The AVPGen system [CI92][CIJ+93][CIJ+94] is an example for a constraint-based test vector generator. It uses templates of constraints to create tests that stress a design in corner-cases. The basic idea behind this technique is that verification scenarios are defined in a slightly more abstract form (SIGL language [CIJ+94]), whereby exact values of variables and inputs are not fixed in the test description. Instead, constraints are set on variables which are represented in symbolic form. From these templates test vectors are generated by pseudo-randomly assigning actual values, which meet the constraints, to the symbolic variables.

Typically, various small stress cases are described in test templates. For
example, one template may specify a pipelined microprocessor instruction sequence that forces data-dependencies across cycles (see fig. 2.7). Subsequently, different templates are composed and the constraints of each template are resolved. An actual test sequence is then created by taking the templates and assigning random values to the unconstrained components.

Template

\[ \text{example: SIGL} \]
\[
\begin{align*}
\text{LOAD} & \quad \text{with } x : R_x \\
\text{ADD} & \quad \text{with } R_v = x
\end{align*}
\]

Test Vector Sequence

\[ \begin{array}{c}
\text{LOAD} \quad R_1, R_2 \\
\text{ADD} \quad R_3, R_4
\end{array} \]

\[ \begin{array}{l}
\text{Instruction Definition} \\
\text{LOAD} \quad R_x : R_x ← \text{mem}[R_y] \\
\text{ADD} \quad R_x, R_y : R_z ← R_x + R_y
\end{array} \]

Figure 2.7: Template to Check Read After Write Register Dependency

Discussion

The power of constraint-based test vector generation is the feature which allows test engineers to devise corner cases and to create templates for them. Constraint solving automatically resolves constraints from these templates, produces a set of new templates, and finally assigns actual values to form test vectors. This late binding of values is powerful, since it enables reuse of templates to create many different test values. A drawback of this technique is that it still requires design knowledge provided by test engineers to create meaningful test templates.

2.4.4 Random Test Vectors

Motivation

Random test vectors are another approach to find errors which are not directly correlated to the core functionality of a design and which are difficult to devise.

---

6Constraint solving in test generation involves identifying conditions on outputs or intermediate values which should be tested and then calculating input values that lead to these conditions.
by test engineers. As opposed to the previous technique this method rather tries to accidentally find corner cases by using a large number of vectors than to pursue a systematic approach. Random vector generators also require no or at least simpler directives to produce vectors which makes this technique easy manageable.

Realization

Three different methodologies for random vector generation exist:

- Pseudo random
- Static biased random
- Dynamic biased random

The simplest approach for automatic random vector generation is to pseudo randomly produce values. However, for certain designs this kind of heuristical random vector generation provides very poor results, since the vectors always tend to force the MUT in the same states.

To enhance the quality of vector generation, the selection of test vectors can be biased towards certain situations [ABGea92][And92][KN95][MSY+95] [WGRH90]. Usually, these situations are identified by the designer as requiring extra attention due to the complexity of the circuit part which is stimulated by that input value. Thus, biasing simply raises the probability of rare situations or interactions provoking hopes of bringing up untested cases (see fig. 2.8).

In simple approaches the likelihood of different test vectors is statically weighted (static biased), in more complex approaches dynamic weighting [ABDDea91][Cha00] which is based on the history of the last few test vectors is utilized (dynamic biased), for instance, by considering a microprocessor as abstracted black-box model with an operation code input and a data in/output. The simulation with statistically weighted random test vectors provides very poor results because the use of several operation codes implies a preset processor status (e.g. an ALU operation makes little sense, as long as the registers involved carry invalid values.). With dynamic weighting of test vectors “suggestive” operation code sequences are enforced [RTJN97].
Discussion

Random vector generation has the advantage of being relatively simple to set up and the vectors generated can quickly find most of the obvious and many complex bugs within a design. Furthermore, random vectors can generate test interactions not thought of by test engineers. Over a long period of generation, the probability to exercise many of the complex interactions increases. And in theory, if test generation can proceed indefinitely, all possible interactions are created, assumed controllability of all parts in the MUT is warranted. Of course, this is hypothetical in a real design environment with finite resources. A drawback of verifying a design by random vectors is the missing metric to measure the progress of fault coverage in order to estimate when and if certain interactions will be tested.

2.4.5 Test Vectors from Real-World Applications

Motivation

The execution of real world vectors allows for a verification on a more abstract level. Thereby, only the passing of certain simulation states is of interest. Intermediate steps can be neglected. Running real-world applications not only allows to detect functional failures, but may also help to find failures in the
specification of the design.

**Realization**

Depending on the current design one can imagine several kinds of real-world vectors, for example, the use of vectors from the embedding target hardware which already exists (e.g. when in a larger system two ASICs are merged into one ASIC to reduce the costs, but the rest of the system remains unchanged) or running software on a microprocessor design such as an operating system.

**Discussion**

Real-world applications are useful to perform final functional verification before the design is declared to be ready for tape-out. Failure localization and debugging opportunities are limited in this technique (either the microprocessor has booted or not). Executing real-world applications is very time-consuming. Thus, in order to obtain sufficient simulation performance hardware prototyping is preferred to software simulation.

**2.4.6 High-Level Test Vector Generation**

**Motivation**

All techniques described so far are non-systematic approaches which result in a potential risk that certain interactions are overlooked and never verified. This is particularly true for control intensive parts of a design. Thus, one of the major challenges in high-level test vector generation is to find metrics to measure the quality of verification.

**Realization**

High-level test vector generation operates on HDL circuit descriptions. It is a more systematic approach to generate test vectors. Dedicated algorithms
analyze the source code of the MUT and, on the basis of the extracted information, generate test vectors to stimulate specific model parts. Two different methodologies are distinguished:

- Coverage-oriented
- Fault-oriented

A *coverage-oriented* method, the hierarchical test generation algorithm (HGBT), is introduced in [RPA93]. For every single process of the MUT a set of precomputed test vectors, which verify the functional correctness, is stored in a library. With this data, process model graphs [RPA93], and the source code of the MUT, a modified version of the D-algorithm [Rot66] hierarchically generate test vectors verifying the complete MUT as a black box. The goal of this technique is to obtain a high failure coverage by causing a large number of signal activities with a small test sequence.

*Fault oriented* methods for behavioral models are based on the same approaches that are used in test vector generation for gate-level models. However, instead of the stuck-at failure model more abstract failure models have been introduced. In [CA94] three failure classes are proposed: behavioral stuck-at faults (BSA) which indicate single bit signal failures, behavioral stuck-open faults (BSO) which detect incorrect signal assignments, and micro-operation faults (MOP) which indicate mutations of arithmetical functions, e.g. the mutation of an addition into a subtraction. Another failure model proposed by Ghosh [GC91] has four different failure classes. Once a failure model is defined, dedicated algorithms can be used to generate test vectors. In [CA94] such an approach is introduced (see also fig. 2.9). Good/Bad pairs (G/B)\(^7\) are assigned to internal states of the MUT for different failure models (injection). Then, the B-algorithm computes values which stimulate the MUT for a certain fault in such a way that a G/B pair is generated, \(G \neq B\) (generation). To assure observability, this G/B pair is propagated to a primary output of the MUT (check observability). As a last step the B-algorithm checks whether it is possible to stimulate this G/B pair (check controllability). Thus, signals used for G/B generation are recursively propagated to primary inputs.

\(^7\)G represents the logic value of an expression for a fault-free circuit model and B the logic value of this expression for the faulty circuit model.
2.5 Quality Measurement Metrics

1) Fault Injection: BSO (a)

```vhdl
signal a, b, sel, out : std_logic;

if (sel = 'l') then
    out <= a; G/B: 0/1
else
    out <= b;
end if;
```

2) G/B Generation

<table>
<thead>
<tr>
<th></th>
<th>sel</th>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>preloading</td>
<td>0</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>transferring</td>
<td>1</td>
<td>0</td>
<td>x</td>
<td>0/1</td>
</tr>
</tbody>
</table>

x: don't care

3) Check Observability

![MUT diagram with a stuck-open fault (BSO)](image)

4) Check Controllability

![MUT diagram with a stuck-open fault (BSO)](image)

**Figure 2.9:** Behavioral Checking of a Stuck-Open Fault (BSO)

Discussion

Conceptually, high-level test vector generation is appealing since it identifies where errors may occur and grades test vectors based on their ability to exercise that part of the code. A major problem of this approach is to develop an adequate set of design errors. In fabrication testing, the stuck-at fault model has a basis in physical defects, lending credibility to the methodology. However, behavioral design errors do not have a widely accepted counterpart. Furthermore, it is unlikely that exercising all defined design errors is feasible by using finite resources.

2.5 Quality Measurement Metrics

In design verification by simulation it is mandatory to check whether the test vectors used really exercise the MUT as intended. Sometimes, even a huge number of test vectors yields very poor results, or a design undergoes major changes which makes test vectors ineffective at checking those interactions they were originally designed for. In addition, the decision to tape out, though often depending on the market pressure, should be made based on projected "completion" of verification.
Therefore, different metrics are introduced which enable test engineers to track the improvement in verification. These metrics are shown in tab. 2.4.

<table>
<thead>
<tr>
<th>Metric</th>
<th>MUT Handling</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Modified</td>
</tr>
<tr>
<td>Bug Discovery Rate</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Low-Level Profiler</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>High-Level Profiler</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Assertion Checker</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Table 2.4: Classification of Quality Measurement Metrics**

The drawback of all introduced metrics is the heuristical approach resulting rather in a statistical prediction of the verification quality than a definite guarantee for the functional correctness of the circuit.

### 2.5.1 Bug Discovery Rate

An approach derived from software quality checking is statistical analysis of the bug discovery rate [Cla90]. In a simplified model during verification by simulation, the number of detected failures decreases monotonously with time because detected failures are gradually removed from the MUT. When the period of time between the detection of two successively failures (MTTF\(^8\)) [MZ98] exceeds a specified threshold the design is declared ready for tape-out.

**Discussion**

This metric is highly dependent on the quality of the test vectors, since it assumes a constant stream of different, high quality vectors. However, this assumption is often not true at all. Usually, whenever a new method of test vector creation is introduced, a burst of bugs is found since new methods tend to exercise new interactions. During the use of a new test creation method, the bug rate drops until the next method is introduced. Hence, the bug discovery rate is a better productivity measure than a verification completion measure.

---

\(^8\)MTTF: mean time to the next failure.
Furthermore, the metric does not predict anything about which interactions in the MUT have been tested and which have not.

2.5.2 Low-Level Profilers

Using profilers demands a readable MUT. Low-level profiling such as measuring node toggle rate is used when a gate-level representation of the MUT is verified. This metric results in a coverage that indicates which nodes in the netlist have low-to-high or high-to-low transition during simulation and how often this happens for each node.

Discussion

If a node never toggles the corresponding logic is certainly not verified. However, 100% node toggle coverage is no guarantee for a functional correct design. It is rather a measure which can be useful to get a better balance of toggles across nodes in a circuit. This would help to ensure that all parts of a circuit received about equal amounts of testing, to a first order.

2.5.3 High-Level Profilers

For designs described in more abstract forms (RTL, behavioral), a bundle of profiling coverage metrics is introduced, each stressing another aspect of the MUT [WT95].

Line coverage is a statistic on how often each HDL code line of the MUT is executed, while process triggering coverage is a statistic on how often each process is executed and which signals have triggered this process. These two metrics result in preliminary rudimentary information about the quality of test vectors used.

More detailed information, which is also useful for design debugging, is obtained from condition coverage, branch coverage, path coverage or signal trace coverage.

It is often insufficient to just verify that a complex conditional expression has been executed in both true and false condition. Condition coverage re-
solves the complex expression and returns a detailed statistic on the combination of sub-conditions driving the expression to false or true values and on the respective combination remaining uncovered by current test vectors. *Branch coverage* indicates whether all possible outcomes of conditional branches or case statements have been tested during simulation. *Path coverage* returns a coverage statistic of all possible combinations of routes through successive conditions. Finally *signal trace coverage* enables test engineers to monitor combinations of signal groups. Typically, this feature is used to verify whether a finite state machine has entered all valid states during simulation by tracing dedicated signals for values which determine the states. A further typical application is to verify that not more than one tri-state bus control signal have been switched-on at the same time.

**Discussion**

Even though these metrics are based on coverage of HDL code rather than on the function of the MUT, they provide a good baseline metric for testing “completeness”. While high-level profilers do not indicate accurately whether logic interactions have been tested, they ensure coverage of basic functionality before such interactions are considered.

**2.5.4 Assertion Checkers**

Another approach to measure the quality of test vectors is to use assertion checkers. They suppose full transparency of the MUT. Assertion checkers test the MUT for error conditions during simulation. If an error occurs the simulation is halted. This requires additional HDL code into the MUT which monitors the occurrence of certain conditions or transitions. Typically, it is tested whether a state is reached within a finite state machine. Furthermore, mutual exclusion of signals or the absence of unknown or 'X' are tested. Thus, assertion checkers do not model expected behavior of the MUT, they just check for obvious errors.
Discussion

In most cases assertion checkers are simple to generate since they are generally localized and small. The often time-intensive propagation of a failure occurring deep in the design to a primary output is not necessary as the simulation model is transparent (soft box). However, this technique slows down simulation speed and supposes a detailed knowledge of the MUT when adding assertion checkers to the design.

2.6 Implications

Although design verification by simulation can not be exhaustively performed for large designs and the quality of simulation is difficult to determine, software simulation is still the working horse to validate the implementation of a model against its specification. Thus, it is mandatory to keep up simulation-based verification approaches with new design methodologies such as VC-based SOC design. As will be shown in the following chapters, the resulting bottleneck with regard to verification can be partly resolved by designing reusable test benches. But first of all the principal structures of test benches will be described in the next chapter.
3 Test Benches

This chapter gives an overview of test environments which are used for simulation-based functional verification of digital circuits. Different methodologies of test bench arrangements are introduced with respect to their area of application. Furthermore, the influence of the model under test (MUT) on the simulation framework is studied and benefits and drawbacks of the different frameworks are pointed out.

3.1 Basic Test Bench Structure

When using simulation to verify functional correctness of a design, a test bench has to be prepared. A test bench is a model or collection of models and data files that applies stimuli to the MUT, compares responses of the MUT to the expected responses, and reports any differences observed during simulation [Bie97][Sch95][May92].

A test bench comprises the following parts (see also fig. 3.1):
- Stimuli vectors
- Reference vectors
- Stimuli application block
- Response acquisition block
- Comparator block
- Simulation control block with test bench triggering mechanism

Stimuli vectors form, together with reference vectors, the test vector set, the composition of which has been discussed in chapter 2. Once these vectors are generated, in simulation stimuli are applied to the MUT by the stimuli application block. Furthermore, the outputs of the MUT are sampled (response acquisition block) and then compared to the reference data (comparison block). All these actions have to be executed at the correct time in a well defined sequence. This task is handled by the simulation control block that synchronizes all blocks of the test bench.

![Figure 3.1: Basic Test Bench Structure](image)

In the second part of this chapter an abstracted model of a test bench is used to point out different verification frameworks.

### 3.1.1 Reactive Test Benches

In the test bench introduced above, stimuli vectors are applied strictly sequentially. However, this is insufficient for some designs because certain situations
occurring during simulation have an impact on the choice of the following stimuli vectors and reference data. Verification environments which are sensitive in such a way are called reactive. They are divided into two classes:

- Test benches with MUT feedback
- Test benches with comparator feedback

**Test Bench with MUT Feedback**

In reactive test benches with MUT feedback the outputs of the MUT influence the order of stimuli and the reference data (see fig. 3.2). The verification of a microprocessor is a typical example of a verification requiring a test bench with MUT feedback. In a microprocessor the program counter (PC) specifies the next instruction within the program code. In normal operation the PC is incremented and the instructions which form the stimuli data base are sequentially applied to the MUT. But after a branch or subroutine call the PC is set on the target address, which is reflected in a jump in the stimuli as well as in the reference data.

![Figure 3.2: Reactive Test Bench with MUT Feedback](image)

**Test Bench with Comparator Feedback**

In fig. 3.3 a reactive test bench is shown with a feedback from the comparator to the stimuli application block. In this configuration the comparator’s decisions influence the choice of stimuli. This is especially useful to enhance debugging opportunities. After a failure has been detected, no further functional
stimuli are applied to the MUT. Instead the MUT is stimulated to generate data for diagnosis (i.e. internal state dump of the MUT).

![Diagram of Reactive Test Bench with Test Bench Feedback](image)

**Figure 3.3: Reactive Test Bench with Test Bench Feedback**

### 3.2 Simulation Frameworks

In simulation-based functional verification an important issue is how to generate reference vectors. Accordingly, the simulation framework is chosen.

Depending on the characteristics of the MUT there are two simulation frameworks commonly used:

- Co-simulation frameworks
- Self-checking frameworks

#### 3.2.1 Co-Simulation Framework

In co-simulation frameworks a second, abstract behavioral model of the design, often called the golden model, is written without reference to the implementation model [Hab96]. The two models are then co-simulated using the same set of stimuli vectors. Equivalence is checked by comparing the outputs of the two models with regard to equality.

An important prerequisite to successfully perform this verification methodology is that a failure occurring in one model will not appear in the other. In other words failures in the two models will not be correlated. In practice this
prerequisite can be obtained best by strictly keeping apart the team responsible for the design of the golden model, i.e. the test or system engineers, from the team responsible for the realization of the hardware implementation, i.e. the circuit designers.

When applying a golden reference model three different methods are distinguished for functional verification:

- Consecutive simulation
- Concurrent simulation
- Involute or feed forward simulation

**Consecutive Simulation**

Fig. 3.4 shows a test bench arrangement where the verification process is split in two simulation runs. In a first run reference data is generated and stored in an intermediate data base. In the second simulation run this data is compared to the MUT outputs.

![Figure 3.4: Test Bench for Consecutive Simulation](image)

**Concurrent Simulation**

Another technique is concurrent simulation of the two models. The golden model generates reference data at run-time which is compared to the MUT outputs (see fig. 3.5). As the two models are described at different levels of abstractions, usually a latency compensation is necessary to synchronize their outputs.
Involute or Feed Forward Simulation

Involute or feed forward simulation is closely related to the last approach. However, this method is only convenient for the class of involute\(^1\) circuits (e.g. used in cryptology) or circuits which include besides a block to process a function another block to calculate the inverse function (e.g. designs with separated receiver and transmitter data path). This class of circuits makes it possible to again use the MUT or parts of it as golden model. When cascading two involute models (see fig. 3.6) or feeding forward the transmitter outputs of the MUT to the receiver inputs (see fig. 3.7) the final MUT outputs have to be equivalent to the applied stimuli after a certain latency.

Discussion

The verification of microprocessors is one large class of designs where co-simulation framework is feasible and in widespread use. Thereby, the system specification is transferred into a golden model. This reference model is usually abstracted from timing information and operates on an instruction-accurate

\(^{1}\)Involution represents a type of mapping \(f\), which is self inverse: \(f(f(x)) = x\).
level. Furthermore, it allows concurrent software and hardware development. On the one hand software engineers have a functionally correct, fast platform to validate and debug appropriate software such as assembler code. On the other hand test engineers have a behavioral description of the system specification which is compared to the design implementations in different levels of abstraction (RTL, gate-level, layout).

### 3.2.2 Self-Checking Framework

For some applications it is too costly to develop a golden model. In these cases another approach is appropriate: self-checking frameworks. In self-checking frameworks within the test bench, dedicated modules generate stimuli vectors and their corresponding expected response vectors. Thus, only the RTL implementation model has to be simulated. The burden of correctness checking now lies with the test vectors, having to perform checks on expected data to ensure correctness.

Two different self-checking frameworks are distinguished:

- Simulation with hard-coded test pattern
- Simulation with run-time test pattern generation

### Hard-Coded Test Pattern

In fig. 3.8 a self-checking framework with hard-coded test pattern is shown. In this approach test vectors are encoded in the source code of the test bench,
e.g. as hard-coded data quantities or as tables of constants. This results in one of the bottlenecks of current functional verification efforts. To create a good suite of such self-checking test vectors for a system-on-a-chip a large team of test writers has to work for many months.

**Figure 3.8: Test Bench for Simulation with Hard-Coded Test Pattern**

**Test Pattern Generator**

The second method dynamically generates test vectors during simulation (see fig. 3.9). This is supported by one or several dedicated algorithms which are implemented in the test environment and generate stimuli and their corresponding expected responses at run-time. As the vectors to be created, following the execution of complex sequences, have to make meaningful checks it is extremely difficult to design such automatic self-checking test vector generators. Therefore, this approach is not very popular.

**Figure 3.9: Test Bench for Simulation with Test Pattern Generation**
3.3 Implications

Discussion

Self-checking frameworks have disadvantages:

- The creation of a good suite of test vectors is an expensive, error-prone, and time-consuming step in the design process.

- Furthermore, such test vectors usually require maintenance during the design process to keep up with design changes.

3.3 Implications

Although self-checking frameworks are much easier to set up initially than co-simulation frameworks, they usually suffer from less complete correctness checking. Furthermore, as opposed to co-simulation frameworks which generate reference vectors anew in every simulation run in self-checking frameworks test bench reuse is restricted because reference vectors are encoded in the test bench. This benefit makes the co-simulation approach feasible for the verification of virtual components. The additional effort to develop a second model, the golden reference model is often justifiable because the behavioral model can be used for performance estimations in system simulations or to support hardware/software co-design.
In this chapter a microprocessor developed as a highly parameterizable virtual component (VC) is presented. As possible embedding environment the system-on-a-chip (SOC) ITRASYS is introduced. In particular, its data organization is analyzed in order to elaborate HW/SW partitioning as well as a task scheduling concept. In the second part of this chapter different architecture variants for an ASIP which is able to handle these tasks are evaluated. The most suitable architecture leads to an enhanced stack machine, the VC SPACEMAN. Features which enable easy reuse of SPACEMAN are especially emphasized. At the end of the chapter a series of results of hardware integrations are presented.
4.1 Motivation to Use Programmable Virtual Components in SOC Design

The advances in ASIC technology and design make ever more complex circuits a reality. As a result highly specialized SOC can be developed. Hard competition requires:

- Shorter product development cycles
- Application or customer specific solutions of high complexity

**Shorter Product Development Cycles:** An efficient approach to increase productivity is to create a library of building blocks for all required complex functions. ASICs are then designed at the architecture level, essentially by composing virtual components (VC\(^1\)) in such a way as to realize the overall functionality.

**Application or Customer Specific Solutions:** The development of a complex SOC, however, is very costly. Therefore, it is not economical to implement different hardware versions to fulfill application or customer specific requirements. Instead, a solution, whereby one ASIC covers all areas of application has to be found. Thus, it has to be application specific configured on site. This can be provided when programmable VC (micro-controllers or microprocessors) are implemented in the SOC. Thereby, SOC functionality is implemented partly in hardware and partly in software (HW/SW co-design). Fixed and highly repetitive computational intensive tasks are implemented in hardware. Irregular tasks or tasks likely to be modified can be implemented in software. This approach efficiently combines application-specific functionality with a limited, yet very welcome degree of flexibility in one common

---

\(^1\)Three types of VCs are distinguished: Soft-VCs are synthesizable HDL descriptions of a function on RTL level (VHDL or Verilog). They can be highly parameterized, are technology independent and portable but area and speed optimization is left to later design steps. Firm-VCs have been optimized in structure and in topology for performance and area through floor-planning/placement, for instance by using a generic technology library. The level of detail ranges from region placement of RTL sub-blocks over relatively placed data paths or parameterized generators to a roughly placed netlist. Hard-VCs are neither parameterizable nor portable, but highly optimized (power, size, or performance). They are mapped to a specific technology library, made available as custom physical layout, or a combination of the two. Hard VCs are therefore process/vendor specific.
design. Thus, by downloading different software versions the same basic hardware allows to create a variety of device subtypes. Likewise, later modification or extensions in standards or upgrades in the functionality do not require an expensive hardware redesign, but only a new software release. During system tests, special software running on the embedded processor can help to quickly verify the system or to support or even replace built-in test circuitry. Furthermore, with the realization of functionality in software the product design flow can be more parallelized by concurrent hardware and software engineering resulting in faster time-to-market.

4.2 The Integrated TRAnsmission SYStem ITRASYS

The synchronous digital hierarchy (SDH) [SS96] is a telecommunication system. It is used for broadband data transportation to support voice communication as well as new services such as web-cams or video-on-demand. ITRASYS [TSR+99] is an add/drop terminal multiplexer for this system.

Figure 4.1: ITRASYS in a Fiber Optical Ring
4.2.1 System Functionality

An SDH network is arranged in fiber optical double rings (e.g. bandwidth: \(2 \times 156\text{Mbit/s}\)). ITRASYS forms a network node which connects several users over coaxial or twisted pair cables to these rings. Data is transported from a user input (source) through an ITRASYS (add) to the fiber optical ring and from there through an ITRASYS (drop) to another user output (sink), see fig. 4.1.

4.2.2 Data Organization and Frequency Adaptation Mechanisms

ITRASYS handles protocols and data based on the SDH standard [IT97]. SDH groups data hierarchically into different layers. In every layer a data carrier of fixed size, called virtual container, is defined. These virtual containers are subdivided into the overhead section, which contains information for control and supervision, and the payload section (see fig. 4.2).

![Hierarchical Data Organization in Virtual Containers](image)

**Figure 4.2: Hierarchical Data Organization in Virtual Containers**

Two different kinds of layers are distinguished in SDH:

- Termination layers
- Adaptation layers

**Termination Layer**

In the receiver part of ITRASYS a termination layer evaluates the overhead section of the virtual container \(n\) and its data. Payload is directly passed to
the next layer, $n - 1$. In the transmitter part, the termination layer $n$ prepares data for its overhead section and combines it with payload from the previous layer $n - 1$ to a virtual container. Thereby, a pointer in the overhead section indicates the start of the sub-container $n - 1$ (see fig. 4.2).

Adaptation Layer

The data rates of different user inputs (see fig. 4.1) are not synchronized, they vary in frequency and phase. For 2Mbit/s inputs a bit rate tolerance of max. $\pm 50$ppm and for the 34Mbit/s input $\pm 20$ppm is specified [IT91]. To allow the multiplexing of inputs with slightly different data rates into a virtual container of fixed size, data rate adaptation layers are introduced on several layers. Two kinds of adaptation mechanisms are specified:

- Bit stuffing
- Pointer processing

**Bit stuffing:** This is a bandwidth adaptation technique, reserving two spare bits in the overhead section of a virtual container (see fig. 4.3). Justification control bits specify the status of these bits (0: empty, 1: data). If the incoming data rate corresponds to the nominal data rate no stuffing is performed; i.e. one bit is a data bit and the other bit is marked as empty. If the incoming data rate is too high, positive bit stuffing is done; i.e. in certain virtual containers both spare bits are data bits. Vice versa, if the incoming data rate is too slow, in certain containers negative stuffing is executed; i.e. both bits are marked as empty.

![Figure 4.3: Bit Stuffing within a Virtual Container](image-url)
**Pointer processing:** This is a technique with bandwidth adaptation on byte or multiple-byte granularity (see fig. 4.4). As opposed to bit stuffing, no justification control bits exist. Instead, in the transmitter a stuffing event is encoded in the pointer of the overhead section \( n \) by toggling all even pointer bits if the incoming data rate is too high (decrement indication) and by toggling all odd pointer bits if the incoming data rate is too low (increment indication). In the receiver this pointer is evaluated on layer \( n \). If a decrement indication is detected the spare byte in the overhead is marked as payload data and the pointer is decremented. Vice versa, if an increment indication is detected both spare bytes are marked as empty and the pointer is decremented. Therewith, after processing, the pointer still indicates the frame start of the sub-container \( n - 1^2 \).

![Diagram](image)

**Figure 4.4:** Pointer Processing within a Virtual Container

### 4.2.3 ITRACHIP System-On-a-Chip (SOC)

In ITRASYS the complete digital functionality of the data processing is integrated in one chip, which is called ITRACHIP. As simplified model this SOC consists of a transmitter part and a receiver part, shown in the functional block diagram in fig. 4.5.

In the transmitter, data from several users (63 × 2Mbit/s and 1 × 34Mbit/s) is multiplexed, overhead information is added on all different layers, and finally data is output to the fiber optical ring (156Mbit/s). In the receiver, data from fiber optical rings (2 × 156Mbit/s) is evaluated in different layers, demultiplexed, and output to several users (63 × 2Mbit/s and 1 × 34Mbit/s). The

\(^2\)This pointer tracking mechanism enables direct access to any payload data in any layer requiring no evaluation of the higher adaptation layers, just follow the pointers.
4.2 ITRASYS

Figure 4.5: Functional Block Diagram of ITRACHIP

functionality of the different blocks in the data path is classified into atomic
functions which are defined in [IT97]. Besides the data path there is one block to generate timings (Timing Source) and another block which forms the interface to the network management unit (SEMF and MCF).

<table>
<thead>
<tr>
<th>Event</th>
<th>Trigger Mechanism</th>
<th>Frequency</th>
<th>Frequency Tolerance (max.)</th>
<th>Number of Data Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OH_1$</td>
<td>periodical</td>
<td>72kHz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$OH_2$</td>
<td>periodical</td>
<td>72kHz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$OH_8$</td>
<td>periodical</td>
<td>8kHz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$OH_3$</td>
<td>periodical</td>
<td>8kHz</td>
<td>$\pm 0.32%$</td>
<td>1</td>
</tr>
<tr>
<td>$OH_4$</td>
<td>periodical</td>
<td>8kHz</td>
<td>$\pm 0.32%$</td>
<td>1</td>
</tr>
<tr>
<td>$OH_9$</td>
<td>periodical</td>
<td>8kHz</td>
<td>$\pm 0.32%$</td>
<td>1</td>
</tr>
<tr>
<td>$OH_6$</td>
<td>periodical</td>
<td>8kHz</td>
<td>$\pm 0.32% / \pm 0.49%$</td>
<td>4</td>
</tr>
<tr>
<td>$OH_{10}$</td>
<td>periodical</td>
<td>8kHz</td>
<td>$\pm 0.32% / \pm 0.49%$</td>
<td>3</td>
</tr>
<tr>
<td>$OH_5$</td>
<td>periodical</td>
<td>2kHz</td>
<td>$\pm 1.74% / \pm 0.89%$</td>
<td>63/126</td>
</tr>
<tr>
<td>$OH_{11}$</td>
<td>periodical</td>
<td>2kHz</td>
<td>$\pm 1.74% / \pm 0.89%$</td>
<td>63</td>
</tr>
<tr>
<td>$OH_7$</td>
<td>event</td>
<td>-</td>
<td>-</td>
<td>63</td>
</tr>
<tr>
<td>$OH_{12}$</td>
<td>periodical</td>
<td>2kHz</td>
<td>$\pm 1.74% / \pm 0.89%$</td>
<td>63</td>
</tr>
<tr>
<td>$MTS_a$</td>
<td>periodical</td>
<td>2kHz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$MTS_b$</td>
<td>periodical</td>
<td>10Hz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$MTS_c$</td>
<td>periodical</td>
<td>1Hz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$SMA_a$</td>
<td>periodical</td>
<td>233Hz</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$SMA_b$</td>
<td>event</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>$SMA_c$</td>
<td>event</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

Total Number of Processing Events per Second: $738 \cdot 10^3$

Table 4.1: Processing Events in ITRACHIP

Requirements for the Processing of Overhead Data

The shaded blocks in the data path in fig. 4.5 handle overhead ($OH_x$) from virtual containers. These blocks are periodically activated, according to the fixed sizes of the virtual containers. In the region connected to the fiber optical rings the data stream is processed in single data paths. Therefore, these blocks receive $OH_x$ events with high cadences. This is in contrast to data path blocks

---

1. Tolerance using bit stuffing.
2. Tolerance using pointer processing.
in the region connected to the users where the data stream is processed in multiple data paths and therefore $OH_1$ events occur with lower frequencies [R+97]. Tab. 4.1 shows that most $OH_1$ processing frequencies can vary within small ranges due to data rate adaptation mechanisms mentioned in the last section. Besides $OH_1$ events from the data path, tasks from the timing source ($MTS$) and from the management interface ($SMA$) have to be handled.

### 4.2.4 Implementation Variants for Data Processing

The characteristics of the events in tab. 4.1 form the basis to analyze different implementation variants for data processing.

![Figure 4.6: Hard-Wired Distributed $OH_1$ Processing Units](image-url)
Conventional, Hard-Wired Solution

In fig. 4.6 a hard-wired approach is shown. Local controllers are associated to those parts of the data path where $OH_x$ has to be handled. These controllers are highly optimized to compute atomic functions of the respective layer [IT97].

This approach has some drawbacks. On the one hand $OH_x$ processing is subject to changes resulting from modifications or updates of the standards (e.g. some overhead bytes are reserved for future use, others are utilized in different countries for different purposes). With a hard-wired solution every change in the standards demands for new silicon. On the other hand identical $OH_x$ evaluation parts which are used in different layers (e.g. trail trace identification [IT97]) have to be implemented multiple times.

Figure 4.7: Central Programmable $OH_x$ Processing Unit

Figure 4.7: Central Programmable $OH_x$ Processing Unit
Solution using HW/SW Co-Design

Fig. 4.7 shows the approach which is used in ITRACHIP. All tasks are computed in one programmable controller, an application-specific instruction-set processor (ASIP). The chosen HW/SW partitioning is motivated by the idea to preserve flexibility wherever needed and at the same time to provide high performance wherever required. Since payload consumes the major part of the bandwidth, payload handling is processed in hardware. The $OH_x$ data on the other hand is processed in software.

Thus, the functionality of the formerly dedicated hard-wired blocks is converted into assembler source code which is stored in the instruction memory of the ASIP (shaded block of fig. 4.7). This facilitates time sharing, because different atomic functions with a similar functionality can be executed by calling the same subroutine with different parameters. Furthermore, this solution reduces the gate count as shown in [RST+98].

4.2.5 Processing-Event Scheduling Strategy

Using one central ASIP for overhead processing demands for a scheduling mechanism which guarantees that all tasks are handled in time. Frequency adaptation mechanisms preclude $OH_x$ handling in a fixed sequence, because the order of events can change. Two different strategies can be applied to resolve this conflict:

- Polling
- Interrupts

Polling

Polling is a technique in which a plan that defines the order of task handling is explicitly established. A software routine periodically scans all event sources. If an event is detected the corresponding $OH_x$ data is processed. However, this strategy is unsuitable for ITRACHIP because of the large number of $OH_x$ sources and the real-time requirements for $OH_x$ data processing.
Interrupt Concept with Rate Monotonic Scheduling

Instead of an explicit task scheduling technique, an implicit strategy fits better. Rate monotonic scheduling is such an approach [ZA95][YW96]. In this technique every task $OH_x$ with repetition period $\Delta p_x$ is given a priority by applying the following rule:

$$prio(OH_i) < prio(OH_j) \iff \frac{1}{\Delta p_i} < \frac{1}{\Delta p_j} \forall \{i, j \mid i \neq j\}$$ (4.1)

A task switch depends on the priority of the current task and the priority of the requesting task. If $prio(requesting) > prio(current)$ the current task is suspended and the processor starts to execute the new task. Alternatively, if $prio(requesting) \leq prio(current)$ the requesting task is queued and the processor continues current execution.

Although rate monotonic scheduling does not support the optimal scheme [ZA95] for allocating resources of the ASIP to different tasks, its simple rule for task switching enables an efficient hardware implementation. This benefit makes the approach convenient, especially for real-time applications.

4.2.6 System Integration

In ITRACHIP a priority-driven interrupt handler using rate-monotonic interrupt scheduling is implemented.

Twenty-nine hardware interrupt sources with eleven priority levels and five software programmable interrupts with five priority levels result in a total interrupt load of more than 700k interrupts per second. In tab. 4.2 the priorities of the most important interrupt sources and their relative ASIP performance requirements are listed. Certain interrupt sources with equal frequencies have different priorities (compare tab. 4.1 and tab. 4.2). This is necessary to guarantee a suitable starve out sequence in an emergency. Less important tasks obtain a lower priority than more important ones and will thus be disabled first.
4.3 The SPACEMAN ASIP

An ASIP has been developed as central servicing unit which is able to handle the required interrupt rates. This ASIP has been implemented as soft-VC in order to support easy reuse in various projects. This implies the following major constraints for its architecture evaluation:

- Ability to efficiently handle high interrupt loads
- Flexibility to provide wide reusability
- Efficiency to reach small gate count and area

4.3.1 Architecture Evaluation

In tab. 4.3 different ASIP architecture variants using different context switching strategies are valued in respect to area efficiency and performance. The different architectures are discussed in the following.

<table>
<thead>
<tr>
<th>Interrupt Source</th>
<th>Priority</th>
<th>Required ASIP Performance</th>
<th>Interrupt Source</th>
<th>Priority</th>
<th>Required ASIP Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OH_1$</td>
<td>13</td>
<td>9.6%</td>
<td>$OH_5$</td>
<td>10</td>
<td>24.8%</td>
</tr>
<tr>
<td>$OH_2$</td>
<td>13</td>
<td>5.3%</td>
<td>$OH_{11}$</td>
<td>10</td>
<td>4.0%</td>
</tr>
<tr>
<td>$OH_8$</td>
<td>13</td>
<td>3.0%</td>
<td>$OH_7$</td>
<td>10</td>
<td>9.6%</td>
</tr>
<tr>
<td>$OH_3$</td>
<td>12</td>
<td>5.5%</td>
<td>$OH_{12}$</td>
<td>10</td>
<td>0.4%</td>
</tr>
<tr>
<td>$OH_4$</td>
<td>12</td>
<td>4.6%</td>
<td>$MTS_a$</td>
<td>8</td>
<td>~0%</td>
</tr>
<tr>
<td>$OH_9$</td>
<td>12</td>
<td>0.3%</td>
<td>$MTS_b$</td>
<td>7</td>
<td>~0%</td>
</tr>
<tr>
<td>$OH_6$</td>
<td>11</td>
<td>32.9%</td>
<td>$SMA_a$</td>
<td>3</td>
<td>~0%</td>
</tr>
<tr>
<td>$OH_{10}$</td>
<td>11</td>
<td>~0%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: ASIP Performance Distribution

4.3 The SPACEMAN ASIP

An ASIP has been developed as central servicing unit which is able to handle the required interrupt rates. This ASIP has been implemented as soft-VC in order to support easy reuse in various projects. This implies the following major constraints for its architecture evaluation:

- Ability to efficiently handle high interrupt loads
- Flexibility to provide wide reusability
- Efficiency to reach small gate count and area

4.3.1 Architecture Evaluation

In tab. 4.3 different ASIP architecture variants using different context switching strategies are valued in respect to area efficiency and performance. The different architectures are discussed in the following.

Memory-to-Memory Architecture

This architecture does not require registers. Instead, operands are fetched from the data memory, a calculation is performed and the result is written back to
the data memory. The architecture is very hardware efficient and supports context switching without extra effort. Unfortunately, the large number of data memory accesses constitutes a serious bottleneck, making this architecture very slow. Therefore, this approach is infeasible. Architectures based on registers resolve this bottleneck.

### Register-Based Architectures

There are two commonly known register-based architectures:

- Complex instruction set computer (CISC)
- Reduced instruction set computer (RISC)

**CISC architecture:** In this architecture calculations are performed on registers. Data memory access is decoupled and handled with the memory management unit (MMU). CISC processors provide a large number of instructions which support complex calculations and addressing modes and vary in length and execution time. The instructions are interpreted by control logic that is based on microprogram execution. Usually, this logic is quite complex and therefore hardware intensive.

**RISC architecture:** In contrast to CISC a RISC has a low number of elementary instructions, fixed in length and execution time (generally one instruction per clock cycle). This favors pipelining whereby multiple instructions are overlapped in execution. Pipelining results in higher throughput.
which boosts the performance. Simple hardwired control logic makes the architecture very hardware efficient. However, the smaller code density [FL94]) compared to a CISC demands for more instructions to execute one task.

**Interrupt Handling of Register-Based Architectures:** As a result of context switching interrupts cause certain problems in register-based architectures. Following an interrupt launch and prior to the performance of a task switch, all registers need to be saved. After completion of the interrupt routine they are restored. For this purpose, two different approaches are commonly used:

- Register saving
- Windowing

**Register saving:** A section of the data memory is reserved and utilized as stack (see fig. 4.8). It is referenced by the stack pointer which indicates the top element. After an interrupt launch the first \( n \) instructions of the interrupt routine push the actual register contents onto the stack. Before leaving the routine the stored register context is recovered from the data memory with \( n \) pop instructions. On the assumption that \( CPI^3=1 \) this results in a performance loss, i.e. a percentage of the available instruction is occupied for context switching of:

\[
P_{\text{Loss}} = \frac{2 \times n \times \frac{\text{IRQ}}{s}}{\text{CPU}\_\text{Frequency}}
\]

An estimation with regard to the use of this solution in ITRACHIP indicates a performance loss of \( P_{\text{Loss}} = \frac{2 \times (4+2+4) \times 738 \times 10^3}{52 \times 10^6} = 0.34 \). One third of the time would be required for context switching which is very inefficient.

**Windowing:** Another approach is the windowing technique (see fig. 4.9). Registers are organized in \( m \) windows, each window containing \( n \) registers (\( m \) equals the number of supported interrupt levels). For the programmer only the \( n \) registers in the active window are visible. After an interrupt launch the active window is incremented by one and after an interrupt completion it is

\(^3\)Clock cycles per instruction.
Windowing allows a context switch without performance loss, but the area overhead is large especially for applications with several interrupt levels. When windowing the ASIP in ITRACHIP, it would require \((4 \times 8 + 4 \times 16) \times 16 = 1536\) flip-flops to form the register bank. Furthermore, the requirement to randomly access all these registers additionally increases the area and makes it difficult to meet all timing constraints during synthesis.

---

**Figure 4.8:** Context Switching with Saving to and Restoring from Data Memory

**Figure 4.9:** Context Switching with Windowing Technique
**Classic Stack Machine**

Fig. 4.10 shows the approach in which the registers are organized as LIFO hardware stack [Koo89][Wir81] with exclusive access to the top element. Given that the ALU requires a pair of data elements to perform computations, the top entry of the stack is stored in a separate register (TOS register). A typical instruction sequence interacts with the stack as follows: First, data is pushed on the stack. Then, the ALU performs a computation with the TOS register and the top entry of the LIFO stack as operands, and finally the result is removed from the stack and written to the data memory. This architecture supports automatic context saving and restoring. Data of a new interrupt routine is pushed on top of the stack, handled and removed from the stack again. When the interrupt is finished the old context is restored. Thus, a classic stack machine has zero performance loss for interrupt handling. Furthermore, the strictly sequential register access makes it possible to realize the stack with a RAM, which is more area efficient than a register-based stack.

![Figure 4.10: Context Switching with Classic Stack Architecture](image)

The main drawback of a classic stack machine [Koo89] is that the access is limited to the top element of the LIFO stack. This restriction either makes great demands on compilers or results in high data exchange overhead between stack and data memory due to stack reordering operations.

**Enhanced Stack Machine**

An approach to combine the benefits of the classic stack machine and of register-based architectures results in the enhanced stack machine [STR+98], the register organization of which is illustrated in fig. 4.11. The registers are
arranged in a stack, but direct access to \( n \) of the uppermost stack entries is provided. Random access to several stack registers yields a more efficient code and increases programming flexibility, and the stack architecture results in a small gate count.

### 4.3.2 SPACEMAN Architecture

SPACEMAN is the implementation of the enhanced stack machine in Harvard\(^4\) architecture. Fig. 4.12 shows a general block diagram of the ASIP core [RST+99]. The code name SPACEMAN indicates its characteristic: a Stack Processor Adaptable, Customizable and Eligible for Multiple Applications.

SPACEMAN provides four different stacks with independent data buses to support concurrent accesses, namely:

- Return address stack (R-stack) to save the PC after an interrupt launch or subroutine call
- Address stack (A-stack) to support register indirect data memory access, register indirect jumps, and subroutine calls
- Condition code stack (C-stack) to save the ALU-flags after an interrupt launch
- General purpose data stack (D-stack)

\(^4\)Harvard architecture has separate memories for data and program opposed to the von Neumann architecture where data and program share one memory.
SPACEMAN operates with a four-stage pipeline: instruction fetch (IF) from memory, instruction decode (ID) including generation of control signals and calculation of target addresses, execute (EX) instruction and modify data registers, and write back (WB) which supports data memory access.

The data ALU provides common logic, arithmetic and shift operations. All calculations are based on unsigned numbers. Therefore, with the carry flag (C) that indicates an under- or overflow and the zero flag (Z), all logic relations are represented as shown in tab. 4.4. For concurrent operation with the data ALU, a separate address ALU is available. Consisting of an incrementer, decremrenter, and offset addition, it supports address calculations which enable efficient block access operations.
<table>
<thead>
<tr>
<th>Relation</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A &gt; B$</td>
<td>$C \cdot \bar{Z}$</td>
</tr>
<tr>
<td>$A \geq B$</td>
<td>$C$</td>
</tr>
<tr>
<td>$A &lt; B$</td>
<td>$\bar{C}$</td>
</tr>
<tr>
<td>$A \leq B$</td>
<td>$\bar{C} + Z$</td>
</tr>
<tr>
<td>$A = B$</td>
<td>$Z$</td>
</tr>
<tr>
<td>$A \neq B$</td>
<td>$\bar{Z}$</td>
</tr>
</tbody>
</table>

**Table 4.4: Representation of Logical Relations with Unsigned Numbers**

### 4.3.3 Instruction Set

The instruction set contains 70 instructions which are defined in appendix A. All instructions are encoded into a fix-length binary representation (16 bit). To enable high code density the opcode field varies in length (e.g. `pushD address` has two bits for opcode and fourteen bits of operand, while the opcode of a `nop` occupies all 16 bits).

### 4.3.4 Pipeline Hazards

In pipelined processor architectures the execution of the next instruction during its designated clock cycle is prevented in certain situations. These events are called hazards and are classified in three types [HP96]:

- *Structural hazards* indicate resource conflicts as concurrent accesses to single port memories.

- *Data hazards* arise when an instruction depends on the results of a previous instruction which are not yet ready.

- *Control hazards* are generated by branches that interrupt regular pipelining.

Hazards need to be minimized because they cause a pipeline stall which degrades the performance of the processor.
Fig. 4.13 shows a more detailed view of SPACEMAN's pipeline stages indicated by IF, ID, EX, and WB and the paths that could generate hazards are highlighted and signed with numbers:

**Pipeline Stages**

**IF**

**ID**

**EX**

**WB**

---

**Figure 4.13:** Pipeline Stages of SPACEMAN
To support relaxed RAM timing, the interface to the data RAM is buffered by registers (A in Fig. 4.13). To get data from this memory in time, signal forwarding for read access is implemented; i.e. path 1a) bypasses one pipeline stage. This is in contrast to data memory write operations, which are regularly pipelined and performed in EX, path 1b). Therefore, in a read after write sequence RAW a structural hazard occurs because the read instruction in ID and the write instruction in EX attempt to concurrently access the data memory. To resolve this conflict the write access is held back as long as consecutive read accesses occur. The realization details are shown in Fig. 4.14 and the waveforms for the RAW resolution in Fig. 4.15.

A reason for generating data hazard are WAR indication sequences. They occur by a write attempt when the top data stack register is not yet updated with data from the data memory. This is indicated with a dirty flag. To resolve this conflict a second bypass exists (WAR connection in Fig. 4.14) which is activated if the top entry of the data stack has to be written while it is indicated as dirty. In Fig. 4.15 waveforms for a WAR hazard resolution are shown.

The instruction memory interface is also registered by PC and IR (B in Fig. 4.13). Therefore, after decoding a branch in pipeline stage ID, two cycles are required until the first instruction of the branch target address is loaded into IR. That means the instruction directly following the branch instruction is also fetched from the memory. This causes a control hazard which can be resolved by flushing the IR to recover the correct instruction scheduling again. This inefficient solution is avoided in SPACEMAN by a technique called delayed branching [HP96]. Thereby, the instruction following the branch instruction (called delay slot) is executed whether the branch is taken or not.

For conditional branches another kind of control hazard occurs. In this case flags in the condition code registers which are not updated before EX decide whether a branch is taken or not. This would necessitate a second delay slot for conditional branches. However, code studies have shown that in most cases it is difficult, if not impossible, to fill

---

5 A read after write hazard occurs if data is read from memory before it is written back to memory.
6 A write after read hazard occurs if data is written back to memory before it is read from memory.
7 Program counter
8 Instruction register
two delay slots by rearranging the instruction code. Often a \texttt{nop} has to be inserted. To avoid a second delay slot a static branch prediction\(^9\) is implemented in ID. The prediction distinguishes between backward and forward branches. A backward jump is directly executed. Forward branches are not taken. In case of misprediction the pipeline is stalled for one cycle to adjust the pipeline for the jump. This prediction mechanism is hardware efficient and achieves good results. For typical applications a hit rate of 80\%-90\% has been measured.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{data_memory_interface_to_avoid_hazards.png}
\caption{Data Memory Interface to Avoid Hazards}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{waveforms_of_data_memory_interface.png}
\caption{Waveforms of Data Memory Interface}
\end{figure}

\(^9\)Static branch prediction uses an algorithm which does not depend on the dynamic behavior of the branch. This is opposed to dynamic prediction which adapts its prediction mechanism by statistical evaluation of the runtime program behavior.
4.3.5 VC Structure

SPACEMAN as VC is structured in two main parts:

- Functional parts
- Interface parts

Fig. 4.16 illustrates the hierarchical VHDL structure of SPACEMAN. When using the VC in different environments, its interfaces have to be adapted due to different I/O constraints, while the functionality of the core stays the same. On the basis of the subdivision into functional parts and interface parts, the user can adapt locally existing interfaces or add new interface types. Thereby, the functionality of the core is not modified. Each of the interfaces has the possibility to stall the core. This property additionally facilitates the plugging of SPACEMAN into different environments (e.g. multicycle data memory accesses).

SPACEMAN comprises 7300 lines of code; 2.6% are used for the parameter configuration part, 44.4% for the functional part, and 37.4% for the interface part.

Figure 4.16: File Structure of the SPACEMAN
4.4 Parameterization Properties

The most important feature of the soft-VC SPACEMAN as compared to a specialized application specific circuit block is its high parameterization and customization capability.

The properties required for this purpose can be allocated to three different classes:

- Qualitative adaptability parameters enable modifications to the functionality of the VC and therefore have a strong influence on the architecture.
- Quantitative parameterization enables adaptations of key parameters for sizing the VC with a given basic architecture.
- Interface customization enables to choose an interface from a set with different protocol descriptions. This helps to adapt the VC to different environments.

The adaptable key parameters of SPACEMAN will be printed in Greek letters as shown in fig. 4.17 and tab. 4.5.

4.4.1 Qualitative Parameters

The most significant adaption to the core’s functionality can be made by altering the number of supported instructions \( \pi \). From a set of 70 instructions the VC user can assemble the subset required for his current application. The basic units such as data ALU, address adder, stack registers, and Harvard memory partitioning are always needed, whereas hardware associated with disabled instructions will automatically be removed during logic synthesis.

4.4.2 Quantitative Parameters

Quantitative parameters allow to define bus widths and stack characteristics. The size of the core is influenced by these adaptations. The four stacks can be varied in three ways: stack width (\( \alpha, \delta \) and \( \iota \)), number of directly accessible
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$2^\alpha$: interrupt &amp; subroutine address range</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Number of R-stack registers</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>$2^\gamma + \beta$: total number of R-stack elements</td>
</tr>
<tr>
<td>$\delta$</td>
<td>$2^\delta$: register indirect data memory access range or register indirect jump</td>
</tr>
<tr>
<td>$\epsilon$</td>
<td>Number of directly accessible A-stack registers</td>
</tr>
<tr>
<td>$\eta$</td>
<td>$2^n + \epsilon$: total number of A-stack elements</td>
</tr>
<tr>
<td>$\zeta$</td>
<td>Number of C-stack registers</td>
</tr>
<tr>
<td>$\theta$</td>
<td>$2^\theta + \zeta$: total number of C-stack elements</td>
</tr>
<tr>
<td>$\iota$</td>
<td>Data path width</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>Number of directly accessible D-stack registers</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>$2^\lambda + \kappa$: total number of D-stack elements</td>
</tr>
<tr>
<td>$\mu$</td>
<td>$2^\mu$: data memory address range</td>
</tr>
<tr>
<td>$\nu$</td>
<td>Instruction width</td>
</tr>
<tr>
<td>$\xi$</td>
<td>$2^\xi$: instruction memory address range</td>
</tr>
<tr>
<td>$\omicron$</td>
<td>$2^\omicron$: number of trap levels</td>
</tr>
<tr>
<td>$\pi$</td>
<td>Number of supported instructions</td>
</tr>
</tbody>
</table>

Table 4.5: Configurable Parameter Set

registers ($\beta, \epsilon, \zeta$ and $\kappa$) and stack depth ($2^\gamma + \beta, 2^n + \epsilon, 2^\theta + \zeta$ and $2^\lambda + \kappa$). The selected bus and ALU width are then adapted to these parameters.

### 4.4.3 Interface Customization Parameters

SPACEMAN supports several interface communication protocols. They can be used to complement the respective stack heads with either of the following:

- **Registers:** If a stack occupies only a few bits of memory (in this case registers use less silicon area than a RAM).

- **Asynchronous RAMs.**

- **Fast synchronous RAMs:** This interface type is designed to attach a RAM with a short access time. Data from the RAM is fed into the VC core logic without intermediate registers.
4.4 Parameterization Properties

- Slow synchronous RAMs: For RAMs with a high access time, the interface contains additional flip-flops, which decouple timing of RAM and core logic.

Since memory access is critical for processor performance, only asynchronous and fast synchronous RAM interface types are available for data and program memory.
4.5 Implementations

Two different hardware integrations of the stack processor have been realized: a prototype integration of SPACEMAN as stand-alone component and a commercial integration of the stack ASIP embedded into a telecommunication SOC. In the following the results of both integrations are introduced.

4.5.1 Prototype of SPACEMAN

To examine the ability of the SPACEMAN VC for the complete design flow, a prototype ASIC was integrated (see fig. 4.18, tab. 4.7 and [SRT+99][RST+99][RST+00]). For this integration the parameter settings in tab. 4.6 were chosen; R-stack and C-stack are realized with registers, D-stack and A-stack as on-chip Synopsys DesignWare RAMs. For the external instruction and data memories asynchronous interfaces were chosen.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Type</th>
<th>Width</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>S-RAM</td>
<td>16</td>
<td>-</td>
</tr>
<tr>
<td>Data</td>
<td>S-RAM</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>D-Stack</td>
<td>DW-RAM</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>A-Stack</td>
<td>DW-RAM</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>R-Stack</td>
<td>Register</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>C-Stack</td>
<td>Register</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 4.6: Configuration of SPACEMAN VC for Prototype Integration

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0.6µm CMOS (3LM)</td>
</tr>
<tr>
<td>Chip Size (incl. Pads)</td>
<td>4.1mm × 4.5mm</td>
</tr>
<tr>
<td>Power Supply Voltage</td>
<td>5V</td>
</tr>
<tr>
<td>Operating Clock (max.)</td>
<td>110MHz</td>
</tr>
</tbody>
</table>

Table 4.7: Prototype SPACEMAN Characteristics
4.5 Implementations

Figure 4.18: Micro-Photograph of Prototype Implementation of *SPACEMAN*

4.5.2 ITRACHIP System-On-a-Chip

ITRACHIP SOC was the first commercial product in which the stack ASIP has been implemented for controlling and supervision tasks. Fig. 4.19 shows the photomicrograph of the complete ITRACHIP [TSR+99] with embedded ASIP [STR+98] which is highlighted. In tab. 4.8 the major characteristics of ITRACHIP are illustrated and in tab. 4.9 the parameters of the implemented stack ASIP are illustrated.
### ITRACHIP

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0.35μm CMOS 3LM</td>
</tr>
<tr>
<td>Size of Logic Units</td>
<td>638kGE</td>
</tr>
<tr>
<td>Size of RAMs</td>
<td>254kBit</td>
</tr>
<tr>
<td>Area</td>
<td>144mm²</td>
</tr>
<tr>
<td>Core Supply Voltage</td>
<td>2.5V</td>
</tr>
<tr>
<td>Pad Supply Voltage</td>
<td>3.3V</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>3.5W</td>
</tr>
<tr>
<td>Package</td>
<td>672 Ball-Grid-Array</td>
</tr>
</tbody>
</table>

**Embedded ASIP**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Frequency</td>
<td>52MHz</td>
</tr>
<tr>
<td>Size of Logic Units</td>
<td>13kGE</td>
</tr>
</tbody>
</table>

**Table 4.8: ITRACHIP Characteristics**

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Width</th>
<th># Stack Elements</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Direct Accessible</td>
<td>Core External</td>
</tr>
<tr>
<td>D-Stack</td>
<td>8</td>
<td>4</td>
<td>up to 256</td>
</tr>
<tr>
<td>C-Stack</td>
<td>2</td>
<td>1</td>
<td>up to 256</td>
</tr>
<tr>
<td>A-Stack</td>
<td>16</td>
<td>4</td>
<td>up to 256</td>
</tr>
<tr>
<td>R-Stack</td>
<td>16</td>
<td>8</td>
<td>up to 256</td>
</tr>
<tr>
<td>Instr. Memory</td>
<td>16</td>
<td>-</td>
<td>6kWord</td>
</tr>
<tr>
<td>Data Memory</td>
<td>8</td>
<td>-</td>
<td>8kByte</td>
</tr>
</tbody>
</table>

**Table 4.9: Parameters of the ASIP used in ITRACHIP**

---

1Core external elements of D-Stack and C-Stack share one memory (dynamic partitioning).
2Core external elements of A-Stack and R-Stack share one memory (dynamic partitioning).
Figure 4.19: ITRACHIP Micro-Photograph with Highlighted ASIP Part
In the next two chapters a simulation-based approach for the functional verification of stand-alone virtual components (VC) is elaborated. In particular, the influence of different parameterization classes on the verification flow is pointed out. In this chapter problems and requirements of VC verification are first listed. Then, general aspects of the verification flow which result in a co-simulation framework using two consecutive simulation runs are discussed, these are namely: vector generation and model checking. In the following part, both runs are introduced in detail, once for a generic model and once for the specific VC of the test vehicle SPACEMAN.
5.1 Problems of and Requirements for Simulation-Based VC Verification

As discussed in chapter 2 dynamic approaches based on simulation are the only feasible way to verify the functional correctness of a design, i.e. to compare the specification with the implementation.

However, VCs complicate this task as their parameterization possibilities lead to a large variety of implementations, each of which requires specific verification data. In general, unknown embedding environments and interface conditions further complicate functional verification. Therefore, VCs ask for a test bench that automatically adapts to any configuration. The manual rewriting of a test bench for each new application would make reuse of VCs unfeasible since too much knowledge of VC implementation details would be required.

In particular, the following problems are associated with simulation-based VC verification:

- Highly parameterized circuit models make test vector generation more difficult due to of the great variety of distinct circuit implementations, each of which requires its adapted test vector set. Consider the impact of scalable data widths, for instance, or of selectable storage elements (flip-flops, register files, on-chip RAMs, and off-chip RAMs).

- The diversity of implementations increases if alternative architectures can be selected. Different interface types or application-selectable instruction sets are examples for such alternative architectures.

- Parameterizable architectures for time, area, latency or throughput optimization maintain the input-to-output relationship and the overall functionality. However, the internal structure may differ (e.g. as a result of different architecture transformations: decomposing, pipelining, or replicating [Kae99]), thus leading to different static latencies. A typical example is pipeline depth of microprocessors.

- Data-dependent execution times lead to dynamic latency, e.g. as a result of program-controlled data processing.

In addition, some requirements associated with the use of VCs also influence the verification flow:
5.2 General Aspects of the Verification Flow

5.2.1 Test Vector Preparation

The methodology to prepare adequate test vector sets for different VC configurations has a strong impact on the verification flow. Three different approaches can be distinguished:

- Generic test vector data base
- Customized test vector data base
- Vector generation by a reference model

Generic Data Base, comprising Test Vectors for all Customizations

In this approach a dedicated vector set for every possible parameter constellation of the VC is provided. For functional verification the user picks that set from the data base, which fits for his application-specific parameterization. However, the parameter vector \( p = \{p_0, p_1, \cdots, p_n\} \), even for a small VC, often comprises many elements, several of them having a wide range of possible values. This is especially true for quantitative parameters, such as bus widths, for which potentially any value \( x \in \mathbb{N} \) can be chosen. However, in reality, the VC provider usually specifies boundaries to restrict the range of possible values to a suggestive subset. The number of provided sets is defined as follows:

- VCs are reused by many users and embedded into different environments. Therefore, easy portability and platform independence have to be guaranteed, not only for the VC, but also for its test bench.

- To evaluate the best parameter setting of the VC several approaches have to be analyzed with system simulations. Different application-specific requirements and basic conditions of the VC environment often require a trade-off. Such investigations can efficiently be supported, if besides the parameterizable, synthesizable implementation model a generic behavioral model is made available.
\[ \#sets = \prod_{i=0}^{n} \#values(p_i) \]  

As a complete collection of test vectors has to be prepared for each set, it is obvious that the total number of test vectors explodes, even for VCs with few parameters. Apart from the VC model and its test bench the user would receive an immense amount of additional test vectors most of which would not be relevant for his customization. For this reason, this approach is inapplicable.

**Customization-Specific Test Vectors prepared by the VC Provider**

If the test vector data base is centrally managed by the VC provider the user is discharged. The user receives just those vectors which are relevant for his specific configuration. However, a drawback concerning the actual idea of VC implementation makes this approach unusable too. To provide the accurate test vector set the VC provider has to know the parameterization of the VC in advance. In many cases this is not warranted because often the final parameter set is determined only after system architecture elaboration. Thus, every change of a VC parameter would ask for a new test vector set to be prepared by the VC provider. This procedure is costly, inflexible and slow.

**Vector Generation with a Carbon Reference Model**

Another idea is to generate the test vectors relevant for the actual parameterization on the user’s site. For this purpose, besides the customizable and synthesizable RTL model, a generic algorithmic, behavioral model\(^1\), the so called golden model is necessary. This golden model represents a form of the design specification that can be simulated. As opposed to the previous approaches where test vectors are derived from the specification, in this strategy reference vectors are not any longer precomputed and stored in the data base,

\(^1\text{VSIA [vsi] definition of a behavioral model: A description of the function and timing of a component without describing a specific implementation. A behavioral model can exist at any level of abstraction. Abstraction depends on the precision of implementation details. For example, a behavior model can be a model that describes the bulk time and functionality of a processor that executes an abstract algorithm, or it can be a model of the processor at the less abstract instructions-set level. The precision of internal and external data values depends on the model's abstraction level.} \)
but generated during simulation. This makes the approach more flexible and therefore convenient for a VC verification framework.

For VCs, the golden model takes different customization dependent forms, similar to the chemical element carbon, which may appear as coal, graphite, diamond or even in a more exotic form as buckyball. Because of this similarity, the golden VC model is hereafter called carbon model.

### 5.2.2 Generic Framework for VC Verification

In fig. 5.1 the verification framework based on a carbon model is shown. It is subdivided into two parts: vector generation (light grey) and model checking (grey). The carbon model is used to produce reference vectors for the later vector generation. These vectors are used as expected responses in model checking to verify the parameterized implementation model.

Two major attributes characterize this verification flow:

- Vector generation and model checking use stimuli from the same generic data base, although the carbon model and the implementation model operate on different abstraction levels. Thus, data inconsistencies are avoided.
- Vector generation is implemented in a generic way; i.e. the carbon model is able to generate reference vectors for every configuration of the implementation model.

5.2.3 Concurrent versus Consecutive Simulation

As considered in chapter 2 two different strategies exist in co-simulation verification: the carbon model and the customized implementation model are either simulated in parallel, called concurrent simulation, or the two models are sequentially simulated, called consecutive simulation. Consequently, reference vectors are either generated by the carbon model on demand at run-time or all vectors are first computed and stored in files before they are used as references in model checking.

![Diagram](image)

**Figure 5.2: Synchronization in Concurrent Simulation**

As mentioned, a problem of VC customization is the implementation-dependent latency. Therefore, run-time generation of reference vectors ne-
cessitates an adaptive synchronization of the generic carbon model with the implementation model. Either the vectors have to be delayed trough buffers ($\Delta$, in fig. 5.2) or the execution of the "faster" model has to be suspended. In model checking the implementation model is event-driven by one or several clocks, while in vector generation the carbon model is data- or request-driven. This points out another difficulty of concurrent simulation. Two different model triggering strategies have to be equalized in one test bench. This would require adapters in the test bench (in fig. 5.2 they are called synchronizers) which manage data transfers between the vector generation part (light grey) and the model checking part (grey).

With execution of vector generation and consecutive model checking all these problems can be avoided. Therefore, the consecutive version of the test bench forms the base for our verification flow.

In the next two sections this test bench is elaborated.

## 5.3 Generic Vector Generation

In the first simulation run of the consecutive framework, expected responses are generated with the carbon model. Fig. 5.3 shows a general flow chart. It is divided into two phases:

- Stimuli preparation (light grey)
- Model execution (grey)

### 5.3.1 Stimuli Preparation

The generic stimuli data base is stored in files. During stimuli preparation vectors are read from this data base with dedicated file handling functions which are implemented in the loader. When stimuli vectors are represented in a programming language or stimuli vectors are stored in symbolic form a converter block is necessary to transform the generic stimuli data base into an executable format. Other models can directly handle generic stimuli (e.g. when standardized formats of numerical video or audio data form the stimuli data base).
As shown in fig. 5.3 two different strategies exist to access stimuli:

**Figure 5.3: Generic Flow Chart of Vector Generation**
• During simulation discrete stimuli are read from the data base by the carbon model on request. This is shown in fig. 5.3 on the left.

• All stimuli are read from the data base prior to the actual simulation start. Then, they are buffered into an array of variables, which is hereafter called virtual memory. This is shown in fig. 5.3 on the right.

Request-Driven Strategy

Due to the sequential nature of file access, the request-driven strategy is convenient to simulate applications which demand stimuli in a well-defined sequence. A data path with a continuous input data stream is a typical example (see fig. 4.5).

Download Strategy

In reactive frameworks, outputs of carbon models or decisions in the test benches influence the stimuli sequence, as discussed in section 3.1.1. Therefore, reactive carbon models (see fig. 3.2) or reactive test benches (see fig. 3.3) require random stimuli access. Thus, certain sections in the stimuli data base are skipped, other sections are executed later, or repeated several times. To emulate random stimuli access with files is very inefficient and time-consuming and therefore not feasible. It is better to download stimuli vectors into a virtual memory prior to model execution as the complete stimuli data base is visible at any time during simulation. Repetitive sequences are provided by multiple access to the corresponding parts in the virtual memory. Furthermore, the number of vectors can be reduced by merging identical stimuli sequences.

Required Job Size in Simulation: As opposed to the request-driven strategy where the job size of the simulation is well defined by the size of the model and the test bench, in download strategy it additionally depends on the size of the stimuli data base. For large data bases, memory requirements for the hardware on which the simulation is running is a critical factor. In fig. 5.4 the job size of simulations is illustrated in relation to the number of downloaded stimuli, SPACEMAN VC being taken as example. Measurements are performed on two HDL simulation platforms: ModelSim from Mentor and VSS from Synopsys.
For \#Stimuli < 2^{15} the job size is dominated by the VC model and the influence of downloaded stimuli vectors can be neglected. For \#Stimuli > 2^{15} the job size grows linearly with the number of downloaded stimuli vectors.

![Graph showing the job size for the verification example of SPACEMAN VC with download strategy.](image)

**Figure 5.4:** *Job Size for the Verification Example of SPACEMAN VC with Download Strategy*

### 5.3.2 Model Execution

During the model execution phase the carbon model generates reference vectors. The test is realized as a single loop that is passed once for every execution of a stimuli vector. A pass includes the following actions:

- Stimuli fetch and application to the carbon model
- Execution of the carbon model
- Saving output data of the carbon model in a temporary data base (reference vectors).
- Checking the termination condition to end the simulation
5.4 Vector Generation for SPACEMAN

After the introduction of the generic vector generation flow in the preceding section, its implementation for SPACEMAN will be discussed. For this VC
the stimuli data base consists of an assembler code. Download strategy is used and an assembler parser serves as converter for stimuli editing.

5.4.1 Stimuli Assembly

To provide a single stimuli data base valid for all possible parameter sets vectors are stored in a generic format; i.e. as assembler code. In this representation a stimulus consists of a mnemonic and zero to three operands: mnemonic op1, op2, op3. In appendix A the entire instruction set is given. The stimuli vector set is composed thereof.

The stimuli data base is subdivided into three different parts each of which includes vectors to verify certain aspects of the model, namely:

- Stimuli sequences to verify the basic functionality of the model
- Stimuli sequences to verify corner cases and architecture specific peculiarities of the model
- Stimuli sequences to emulate real-world applications

Stimuli to Verify the Basic Functionality

Qualitative parameters have a strong impact on the choice of stimuli vectors which are used for the verification of the model's basic functionality. Changes of qualitative parameters add or remove functionality to or from the implementation model. In consequence of functional verification, parts of the vectors can be discarded. This characteristic is reflected in the structure of the stimuli data base as follows: for every functional block of the model which can be selectively enabled or disabled a stimuli vector subset is stored in the data base. Each subset is optimized to cover test cases that primarily stress the corresponding functional block and parts of the model which are influenced by its functionality. This kind of vector grouping supports the automatic assembly of stimuli to obtain a high verification coverage for any qualitative VC parameterization.

In SPACEMAN qualitative parameters determine the instruction set. They are defined in the configuration package of the carbon model, ModuleCfpkg,
Figure 5.6: File Structure of the Test Bench for Vector Generation

by the constant *INSTRUCTION_SET* (in fig. 5.6 all files of vector generation are listed and the hierarchical dependency of the appended packages and components, respectively). The structure of *INSTRUCTION_SET* is illustrated with Pseudo-Code 5.1.

**Pseudo-Code 5.1: Definition Instruction Set**

```plaintext
constant INSTRUCTION_SET : std_logic_vector(SIZE downto 0) :=
  (0 => '1', -- incB #imm, dx, dy
   1 => '0', -- cmp #imm, dx
   ..., SIZE => '1' -- sleep
  );
```

To every assembler instruction one bit is assigned in the vector. If the bit is set the functionality which supports this instruction is enabled in the implementation model, otherwise it is disabled.

In the stimuli data base an assembler sequence is implemented for each
such bit and its instruction, respectively. For the VC user the carbon model and the implementation model are black boxes; i.e. the internal states are hidden. To observe the impacts of stimuli application on the models, computed results have to be propagated to primary outputs; e.g. for SPACEMAN, data has to be stored in the data memory. Therefore, in all assembler sequences, besides the actual instruction, an additional instruction which performs a data memory write access is used. Likewise, it is impossible to write a stimuli sequence which covers all kinds of test cases for the stressed instruction by just using the instruction itself (e.g. if the instruction is \texttt{sub} $d_x$, $d_y$ then data has first to be pushed on the data stack before the ALU operation can be executed). Therefore, a minimal basic subset is always enabled in INSTRUCTION SET. This set comprises thirteen instructions which are marked with $\dagger$ in appendix A.

Dependent on the setting of INSTRUCTION SET the different fragments of the stimuli data base are then assembled to the stimuli file, stimuli.asm. The principle of this procedure is shown in fig. 5.7. If an instruction is enabled the corresponding assembler sequence is appended to stimuli.asm, otherwise it is omitted. With this approach stimuli.asm fits to the qualitative parameter customization.

\textbf{Figure 5.7: Assembly of Configuration Dependent Stimuli File}

\textbf{Assumption:} The parameter-dependent concatenation of assembler sequences to a final stimuli file supposes that all sequences are self-contained; i.e. before and after execution of any sequence the model state needs to be equal.
Therefore, supplementary instructions have to be appended to the assembler sequences (e.g. instructions which remove data from the stacks). Otherwise, certain constellations of INSTRUCTION_SET may lead to inadequate concatenations of assembler sequences which push more data on the stacks than they remove from the stacks or vice versa. During simulation this would result in an exception, namely a stack overflow or a stack underflow.

**Stimuli to Verify Corner Cases**

The second part of the stimuli data base checks architecture-specific peculiarities of the implementation model. For SPACEMAN stimuli vectors are prepared which stress the mechanisms to avoid structural hazards, data hazards and control hazards (introduced in section 4.3.4). In particular, assembler code sequences are written so as to verify the correct handling of successive read and write data memory accesses. Furthermore, the branch prediction mechanism (branch miss/hit) is checked as well as the correct handling of an indirect subroutine call when the target address in the address stack is not yet updated. Other tests covered in this part of the stimuli data base are corner cases of the model. Therefore, stimuli sequences are implemented which trigger different kinds of exceptions such as stack underflow or stack overflow.

To retain the adaptability to qualitative parameterization instructions from the *basic subset* are used to write these assembler sequences.

**Stimuli from Real-World Applications**

The last part of the stimuli data base is formed by assembler sequences used in real-world applications. For SPACEMAN these vectors verify mainly interrupt handling and interrupt scheduling. Thereby, critical scenarios such as hold back of an interrupt when a branch instruction is in the pipeline or handling of an interrupt launch when parts of the model are stalled are explored.

The above mentioned three parts of the data base form the basic stimuli set used for functional verification. To enhance the quality of verification the VC user can extend the data base by adding additional application-specific assembler code segments.
5.4.2 Stimuli Download

SPACEMAN's generic stimuli are stored as ASCII\(^2\) text strings in the data base. This representation is inapplicable to be handled in model execution. Therefore, the data base is converted into an executable format before stimuli are applied to the carbon model. This task is done by the assembler parser during stimuli download (see fig. 5.5).

The main routine of the parser is shown in Pseudo-Code 5.2, Pseudo Code 5.3, and Pseudo-Code 5.4. It is subdivided into two parts:

- Disassembling of mnemonics and operands and saving the results into a virtual memory (first pass)
- Resolving of branch target addresses represented as labels (second pass)

Mnemonic Disassembling

In the first pass, within a loop, assembler code is read line by line from the stimuli data base. In the procedure getOpCode each assembler string is parsed and disassembled. After a syntax check getOpCode extracts mnemonics, operands, and execution directives such as auto address increments or ALU test flags from the string. The values are saved in the record instructionType. This type represents the stimuli format which can be executed by the carbon model. Finally, the converted stimuli are stored in the virtual instructionMemory.

The parser is encapsulated in a separate package (IOPkgB, see fig. 5.6). This structure supports the evaluation of different stimuli representations by implementation of different parsing algorithms in different packages. Depending on the current stimuli format the user selects the dedicated package and links it to the test environment. If this approach is chosen parsing modules can easily be replaced without modifying the actual core of the test bench, thus enhancing reusability.

\(^2\)ASCII: Abbreviation for american standard code for information interchange.

\(^3\)In VHDL the type line is a pointer which accesses a dynamic string. It is declared in the package "textio".
**Pseudo-Code 5.2: Main Routine of Assembler Parsing**

variable `instructionMemory`: array of `instructionType`;
variable `labelMemory`: array of `labelType`;

/* first pass, instruction fetch */
i = 0;
while (not endfile(stimuli.asm)) {
    read(line³, stimuli.asm);
    getOpCode(line, instruction, label);
    instructionMemory(i) = instruction;
    labelMemory(i) = label;
    i = i + 1;
}
/* second pass, resolve labels */
for (j = 0 to i - 1) {
    if (labelMemory(j).targetLabel ≠ null) {
        for (k = 0 to i - 1) {
            if (labelMemory(j).targetLabel.all = labelMemory(k).branchLabel.all) {
                if (instructionMemory(j).mnemonic = "jsr") {
                    /* absolute branches */
                    instructionMemory(j).offsetValue = k;
                } else {
                    /* PC relative branches */
                    instructionMemory(j).offsetValue = k - j;
                }
            exit;
        }
    }
}
}

**Pseudo-Code 5.3: Definition of `instruction ∈ instructionType`**

type `instructionType` is
record
    mnemonic : string;
    immediateValue : natural;
    offsetValue : integer;
    popFlag : boolean;
    testFlag : boolean;
    incDec : boolean;
    adrOperand : natural;
    dataOperand : array of natural;
end record:

**Branch Resolution**

To facilitate assembler code writing branches to labels are supported. The parser resolves these branches, a mechanism which is especially helpful for
**Pseudo-Code 5.4: Definition of label ∈ labelType**

```plaintext

type labelType is
    record
        targetLabel : line;
        branchLabel : line;
    end record;
```

relative jumps. In the first pass the parser extracts branch labels as well as its corresponding target labels (see fig. 5.8, on the left). The values are saved in the second virtual memory, called `labelMemory`, at the same addresses as their corresponding instructions (see fig. 5.8, on the right).

---

**Instruction Memory**

<table>
<thead>
<tr>
<th>addr</th>
<th>mnemonic</th>
<th>labelMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n</td>
<td>add</td>
<td>...</td>
</tr>
<tr>
<td>n+1</td>
<td>decjmp</td>
<td>...</td>
</tr>
<tr>
<td>n+2</td>
<td>writeD</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Record Elements**

- | Branch Label | Target Label |
- | empty        | empty        |
- | label        | label        |
- | empty        | empty        |
- | empty        | empty        |

**Figure 5.8: Branch Resolution**

In the second pass branches to labels are resolved by a linear search in virtual `labelMemory` with `targetLabel` as key. After a hit, depending on the type of the branch (absolute or relative), the final numeric address of the branch is calculated and updated in the virtual `instructionMemory`.

---

### 5.4.3 Run-Time Stimuli Generation

In the last two sections preparation and compilation of stimuli have been introduced. Stimuli are called primary stimuli if they are an integrated part of the VC and applied to the primary inputs of the carbon model by the test bench (see fig. 5.9, 1). However, a test bench can be composed besides the carbon model of other models which represent parts of the embedding environment; these models are called support models. As a consequence some primary in-
puts of the carbon model become test bench internal signals which are stimu-
lated by the outputs of the support models. These stimuli are not fetched from
the precomputed primary stimuli data base, but generated dynamically during
simulation. Hereafter, they are called run-time stimuli (see fig. 5.9, ③).

Figure 5.9: Run-Time Stimuli Generation

Virtual memories are a typical example of support models. Supposing
that the support models of the test bench in fig. 5.9 are virtual memories, the
waveforms of the relevant signals to generate run-time stimuli are illustrated
in fig. 5.10. Primary stimuli are applied to the carbon model ①. The carbon
model generates the expected responses which are sent to one or several virtual
memories ②. In a later phase of simulation some of the stored data are fed back
from the support models to the carbon model and become therewith run-time
stimuli vectors ③.

Figure 5.10: Waveforms of Run-Time Stimuli Generation

The support models implemented in the test bench are not simply restricted
to look-up tables. One can imagine support models with complex algorithms
to compute new run-time stimuli vectors.

5.4.4 Snoopers

As mentioned, in run-time stimuli generation former interface signals of the
carbon model become internal signals. To monitor all outputs at the boundary
of the carbon models test points, hereafter called snoopers, are inserted into the test bench. Two types of snoopers are distinguished (see fig. 5.11):

- **Complex snoopers**: Test points between the interfaces of the carbon model and other support models
- **Primary snoopers**: Test points between the interfaces of the carbon model and primary outputs

![Diagram showing the types of snoopers](image)

**Figure 5.11: Types of Snoopers**

In fig. 5.12 the structure of a snooper used for vector generation and model checking is shown. During vector generation a snooper (generator) samples data and stores it to a file, as shown in fig. 5.12 a). Thereby, the primary snooper type redirects unidirectional data from the carbon model to the reference vector data base. The complex snooper type manages bidirectional data traffic. Data from the carbon model is broadcast to the reference vector data base and to the support model, and data from the support model is forwarded to the carbon model. As illustrated in fig. 5.12 b) during model checking the snooper (comparator) compares outputs of the implementation model to references. Thereby, the primary snooper type redirects unidirectional data from the implementation model to the comparator. The complex snooper type broadcasts data from the implementation model to the comparator and to the support model, and it forwards data from the support model to the implementation model.
5.4 Vector Generation for SPACEMAN

Carbon Model

\[ \text{Req. Accord. Data} \]

Implementation Model

\[ \text{Ctrl. Accord. Data} \]

Support Model

\[ \text{Reference Vectors} \]

\[ \text{Report} \]

(a) Vector Generation

(b) Model Checking

Figure 5.12: Structure of Snoopers

SPACEMAN VC has six memory interfaces (four stack-, one data-, and one instruction memory). Therefore, besides the VC model the test bench contains six memory support models with complex snoopers (see fig. 5.13).

5.4.5 Carbon Model

The carbon model is a black box that generates reference vectors for every possible configuration of the implementation model (see fig. 5.5). It is never synthesized to a gate representation and silicon. Beside reference vector generation the carbon model is also used in system simulations [Syna]. It can also be embedded into the target environment where it is used in different simulation runs to evaluate under varying constraints the best parameter set for the implementation model (e.g. instruction subset or memory sizes) under varying constraints. Furthermore, in system simulations the performance of the carbon model can be measured to validate the overall system specification. Thereby, a larger number of stimuli vectors is executed than in simulations for pure functional verification taking advantage of the much faster execution of
the carbon model.

The earlier mentioned set of tasks defines the major design directives for the carbon model:

- Efficient implementation to support short run-times in simulation

- Ability to generate reference vectors for any parameter set of the implementation model

- Flexible interface handling to easily exchange data between the carbon model and its environment (primary I/O, and other support models)
5.4 Vector Generation for SPACEMAN

Efficiency and Platform Independence Concerns

A carbon model realized in a general purpose programming language such as C or C++ fulfills the first design directive best by providing a simulation speed of over 100 times higher than the speed of the RTL implementation model [KN96][YPY+97]. However, this approach has some drawbacks. As mentioned in section 5.1, a major requirement for simulation-based functional verification of VCs is high portability and platform independence. A second programming language besides HDL demands supplementary knowledge from the VC user as well as dedicated simulation and debugging tools if he wants to verify the VC. Furthermore, HDL simulators have different restrictions concerning foreign language interface handling or subset support. These problems are avoided if the carbon model is described in the same hardware description language as the implementation model (VHDL or Verilog). To gain the highest simulation performance the model has to be written in a purely behavioral form, far from any hardware description.

In SPACEMAN VC the functionality is defined by the chosen instruction subset (see appendix A). As shown in fig. 5.1, in vector generation qualitative customization has no influence on the carbon model. Therefore, the carbon model needs to support the full instruction set. To avoid simulation performance penalties by HDL code of disabled instructions, the carbon model is subdivided into two parts:

- A compact small part which manages execution and interface handling, specified in the entity CoreB (see fig. 5.6).
- The main part which models the functionality. It is partitioned into different procedures, one for each instruction. A procedure is only called when the corresponding instruction has to be executed. All procedures are implemented in the package InstructionPkgB.

To compare the proportions of entities and packages used in vector generation, tab. 5.1 gives the number of code lines of the respective segments.

Model Implementation

To support efficient parameter handling when calling subroutines, the state of the carbon model is defined in one single record: processorState (see Pseudo-
In this record three elements are reserved for each of the four stacks: the stack modification status \([d,a,c,r]StackOp\), the actual values \([d,a,c,r]Stack\), and the number of stack entries \([d,a,c,r]StackCounter\). To access the data memory, record elements for data, address and access mode are specified: \(dMemoryData\), \(dMemoryAddress\) and \(dMemoryOp\). Together with the program counter, \(pC\), and two elements for simulation control, \(trapTrigger\) and \(trapLevel\), the state of the carbon model is completely defined with eighteen entries.

In Pseudo-Code 5.6 the architecture of the carbon model’s core is introduced. Execution managing and interface handling are described in one process as implicit state model\(^4\). This approach results in strictly sequential model execution in several successive steps. Thereby, a single loop is passed once to process one generic assembler instruction. In the loop several synchronization points are implemented. The first synchronization point waits for a new instruction which then triggers an iteration in the loop. The other synchronization points at the end of the loop perform data exchanges with the memory support models implemented in the test bench and their snoopers, respectively.

A simulation run for vector generation is performed as follows. After initialization of \(processorState\) and code download completion the actual model execution starts upon entering the loop. In each loop iteration the following

\(^4\text{Implicit state models are used to describe behavioral models. They have multiple synchronization points. This is in contrast to explicit state models which are hardware oriented descriptions. They provide only a single synchronization point with one triggering condition [Kae99].}\)
Pseudo-Code 5.5: Definition of \( \text{processorState} \in \text{processorStateType} \)

\[
\text{type processorStateType is} \\
\text{record} \\
\quad dStack \quad \text{: array of natural;} \\
\quad dStackOp \quad \text{: type } \{\text{push, pop, idle}\} \\
\quad dStackCounter \quad \text{: natural;} \\
\quad aStack \quad \text{: array of natural;} \\
\quad aStackOp \quad \text{: type } \{\text{push, pop, idle}\} \\
\quad aStackCounter \quad \text{: natural;} \\
\quad cStack \quad \text{: array of boolean;} \\
\quad cStackOp \quad \text{: type } \{\text{push, pop, idle}\} \\
\quad cStackCounter \quad \text{: natural;} \\
\quad rStack \quad \text{: array of natural;} \\
\quad rStackOp \quad \text{: type } \{\text{push, pop, idle}\} \\
\quad rStackCounter \quad \text{: natural;} \\
\quad dMemoryData \quad \text{: natural;} \\
\quad dMemoryAddress \quad \text{: natural;} \\
\quad dMemoryOp \quad \text{: type } \{\text{read, write, idle}\} \\
\quad pc \quad \text{: natural;} \\
\quad trapTrigger \quad \text{: boolean;} \\
\quad trapLevel \quad \text{: natural;} \\
\text{end record;}
\]

actions are done:

- A generic stimuli, \( \text{instruction} \), is fetched from the virtual \( \text{instruction-Memory} \).

- In a case statement, the corresponding procedure defining the functionality is called depending on the argument of \( \text{instruction} \). As parameters the generic instruction and the processor state are handed over.

- If \( \text{processorState} \) indicates a data transfer for one of the stacks or for the data memory the corresponding interface \( [d, a, c, r] \text{StackAccess} \) to the test bench is activated.

- Finally the criterion to terminate simulation is checked. If \( \text{trapTrigger} \) is set in \( \text{processorState} \) the execution of the loop is interrupted by entering into an unconstrained waiting state (wait forever).

Discussion: As set out above, a separate procedure with uniform interface is declared for every assembler instruction in the package \( \text{InstructionPkgB} \).
Pseudo-Code 5.6: Core of the Carbon Model

```plaintext
process CARBONMODEL {
    init(processorState);
    wait until DownLoadCompletion();
    loop {
        wait until InstructionFetch(instruction);
        case instruction.mnemonic is
            when "addA" => addA(instruction, processorState);
            when "addAhi" => addAhi(instruction, processorState);
            when "rol" => ALUrol(instruction, processorState);
            end case;
        if (processorState.dStackOp \neq idle) wait until dStackAccess(processorState);
        if (processorState.aStackOp \neq idle) wait until aStackAccess(processorState);
        if (processorState.cStackOp \neq idle) wait until cStackAccess(processorState);
        if (processorState.rStackOp \neq idle) wait until rStackAccess(processorState);
        if (processorState.dMemoryOp \neq idle) wait until dMemoryAccess(processorState);
        if (processorState.TrapTrigger) wait;
    }
}
```

This procedure implements the specified functionality of the instruction. Due to this encapsulation, changes in the specification of an instruction only cause local modifications in their corresponding procedure. Furthermore, extending the instruction set is straightforward by adding new procedures to the package InstructionPkgB. Due to this structure the VC provider might reuse the carbon model for other processor VCs which are derived from SPACEMAN.

The implementation of the carbon model in a single process as well as the data-driven timing scheme enable short run-times in simulation. This is opposed to the multi-process solution of the hardware-related implementation model where a clock triggers all sequential processes in every cycle, even if no calculations are executed.

Interface Handling

For data exchanges between the carbon model and the snoopers in the test bench full-handshake protocols are chosen. In fig. 5.14 a generic block diagram for bidirectional data exchange of two models using full-handshake is shown, and in fig. 5.15 the corresponding protocol sequence is illustrated, once as pseudo-code and once as Petri net. The benefits of full handshake
protocols are:

- They can be easily implemented
- They are robust and dead-lock free

![Diagram](image)

**Figure 5.14:** Generic Data Exchange between the Carbon Model and other Support Models

```
CMRequest <= {read, write};
wait until CMRequest = {read, write};
TBAcknowledge <= ready;
wait until TBAcknowledge = ready;
CMRequest <= idle;
wait until CMRequest = idle;
TBAcknowledge <= idle;
wait until TBAcknowledge = idle;
```

**Figure 5.15:** Full Handshake Protocol Represented as HDL Pseudo-Code and Petri Net

5.5 Generic Model Checking

In the second simulation run of the verification flow, the expected responses are compared to the outputs of the implementation model. As can be seen in
the flow chart of fig. 5.16 the implementation model as well as its test bench are user specifically parameterized to match the target application. Similar to vector generation model checking is subdivided into two phases:

- Stimuli preparation
- Model execution

### 5.5.1 Stimuli Preparation

In stimuli preparation generic stimuli from the data base are applied to the implementation model. Thereby, the stimuli have to be available in hardware comprehensible format, i.e. in binary representation, because in a later design step the implementation model will be translated into hardware.

As mentioned in the vector generation, for some VCs the generic stimuli are stored in a numerical representation (e.g. as audio or video data). This format enables the loader to directly fetch stimuli from the data base (see fig. 5.16, on the right) and to transform them into binary values according to the qualitative parameter settings. Dependent on the stimuli application strategy (downloading or request-driven) either all stimuli are loaded into a virtual memory before the model execution starts, or every stimulus is fetched on request.

For other models stimuli are stored in an abstract, non-hardware-related format in the data base (e.g. in a programming language or in a symbolic form). To convert this format into stimuli executable by the implementation model, a tool such as an assembler or compiler is necessary (see fig. 5.16, on the left). This converter is actually not part of the test bench. It is implemented on a stand-alone basis. Due to this decoupling from the test bench any programming language is convenient to develop this converter (e.g. a general purpose programming language such as C/C++), and existing commercial tools can also be used. Nevertheless, the converter tool is a part of the verification flow and is therefore validated as well.
5.5.2 Model Execution

As opposed to the data- or request-driven sequential processing of the carbon model in a single process, the hardware-related implementation model is
described as explicit state model; i.e. several processes are concurrently executed, triggered by a clock. To synchronize data exchanges of different processes and to correctly emulate their concurrent execution, the timing scheme shown in fig. 5.17 has to be met [Kae99].

![Timing Scheme](image)

**Figure 5.17: Timing Scheme**

According to this timing scheme the following actions are performed within a clock cycle:

- All processes of the implementation model (MUT$^5$) are triggered
- New stimuli vectors are applied to the MUT
- MUT outputs are compared to reference vectors, and the termination criterion is checked to terminate

## 5.6 Model Checking for SPACEMAN

After the introduction of the generic flow of model checking, its implementation for SPACEMAN is discussed now (see fig. 5.18).
5.6 Model Checking for SPACEMAN

5.6.1 Stimuli Preparation

In SPACEMAN the stimuli data base is transformed into binary representation by an assembler. This assembler is written in C and bases on dasm, developed \(^5\)MUT: Model under test
by Matt Dillon. In the implemented version the mapping directives for the assembler instructions are fixed; i.e. the binaries of the instruction set (see appendix A) are hard-coded in the assembler as well as in the implementation model. However, the introduced design flow would also allow the implementation of dynamic mapping. In dynamic mapping, directives are defined by the user in ModuleCfgPkg. The assembler parses ModuleCfgPkg and extracts this information. Based on the directives, a mapping which is adapted to the target application (e.g. one application demands for a large branch address range, but requires just half of the instruction set) is chosen. Similar to vector generation, the download strategy is used in model checking. A binary code is loaded into a behavioral VHDL RAM model before actual simulation starts.

![File Structure Diagram](image)

**Figure 5.19: File Structure of the Model Checking Test Bench**
As mentioned, the assembler is part of the verification flow (see fig. 5.18). To support efficient failure localization in the assembler, a disassembler has been developed. After the disassembling of the binary code, the result can be compared to the stimuli data base in a feedback loop.

<table>
<thead>
<tr>
<th>File Name</th>
<th># Lines of Code</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModuleCfgPkg.vhd</td>
<td>115</td>
<td>1.3%</td>
</tr>
<tr>
<td>ModuleTbCfgPkg.vhd</td>
<td>48</td>
<td>0.5%</td>
</tr>
<tr>
<td>ModulePkg.vhd</td>
<td>74</td>
<td>0.8%</td>
</tr>
<tr>
<td>ModuleTbPkg.vhd</td>
<td>731</td>
<td>8.1%</td>
</tr>
<tr>
<td>ModuleTb.vhd</td>
<td>900</td>
<td>10.0%</td>
</tr>
<tr>
<td>RAM.vhd</td>
<td>61</td>
<td>0.7%</td>
</tr>
<tr>
<td>Module.vhd</td>
<td>480</td>
<td>5.3%</td>
</tr>
<tr>
<td>DStackInterface.vhd</td>
<td>627</td>
<td>6.9%</td>
</tr>
<tr>
<td>AStackInterface.vhd</td>
<td>627</td>
<td>6.9%</td>
</tr>
<tr>
<td>RStackInterface.vhd</td>
<td>627</td>
<td>6.9%</td>
</tr>
<tr>
<td>CStackInterface.vhd</td>
<td>627</td>
<td>6.9%</td>
</tr>
<tr>
<td>OpCodeInterface.vhd</td>
<td>133</td>
<td>1.5%</td>
</tr>
<tr>
<td>DMemInterface.vhd</td>
<td>83</td>
<td>0.9%</td>
</tr>
<tr>
<td>Core.vhd</td>
<td>663</td>
<td>7.3%</td>
</tr>
<tr>
<td>InstructionDecoder.vhd</td>
<td>1169</td>
<td>12.9%</td>
</tr>
<tr>
<td>Sequencer.vhd</td>
<td>330</td>
<td>3.7%</td>
</tr>
<tr>
<td>ALUData.vhd</td>
<td>419</td>
<td>4.7%</td>
</tr>
<tr>
<td>ResetSleepCtrl.vhd</td>
<td>412</td>
<td>4.6%</td>
</tr>
<tr>
<td>DStack.vhd</td>
<td>224</td>
<td>2.5%</td>
</tr>
<tr>
<td>AStack.vhd</td>
<td>285</td>
<td>3.2%</td>
</tr>
<tr>
<td>RStack.vhd</td>
<td>98</td>
<td>1.1%</td>
</tr>
<tr>
<td>DMem.vhd</td>
<td>296</td>
<td>3.3%</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>9029</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

*Table 5.2: Model Checking: Lines of VHDL Code*

5.6.2 Model Execution

The implementation model of SPACEMAN is triggered by one clock. Simulation is terminated with the assembler instruction *trap*. Synchronization of MUT outputs with the expected responses is discussed in detail in the next
chapter. In fig. 5.19 the file structure for model checking is shown and in tab. 5.2 the sizes of corresponding entities and packages are illustrated.

### 5.6.3 Line Coverage

To study the impact of modifications in \textit{INSTRUCTION\_SET} on the line coverage (see section 2.5.3), measurements for three different instruction sets are reported\(^6\):

- Set 1: all instructions are supported (prototype configuration).
- Set 2: 75\% of the instructions are supported. Thereby, some instructions from all classes are disabled (see appendix A).
- Set 3: 50\% of the instructions are supported. Set 3 is derived from set 2 by further reduction. Thereby, the additional disabled instructions exclusively belong to the class \textit{Data ALU} (see appendix A).

Furthermore, for set 1 the impact of the different parts in the stimuli data base, namely stimuli vectors to verify the basic functionality (BF), corner cases (CC), and real-world applications (RA) on the line coverage is studied in detail.

### Results of Vector Generation Simulation Run

In tab. 5.3 the results of line coverage for the vector generation simulation run are listed. The table is subdivided into the carbon model and the test bench section (see fig. 5.6).

**Line Coverage for the Carbon Model:** For the carbon model almost maximum line coverage is achieved with BF and CC vectors. As mentioned in section 5.4.5 the functionality of the carbon model is encapsulated in different procedures. An interrupt launch is handled locally in its assigned procedure.

\(^6\)The results base on the VC configuration used for the prototype integration (see tab. 4.6).
and has little impact on the rest of the model. Thus, RA vectors which sensitively stimulate the interrupt handling parts only slightly enhance the line coverage of the carbon model. In the VHDL code of the carbon model debugging parts are implemented to report warnings or failures to the VC user (e.g. handling of illegal mnemonics or illegal assembler sequences such as branches in delay slots). As shown in tab. 5.3 these parts are not executed in regular operation and therefore, 100% line coverage is not obtained with the full supported instruction set (set 1). Furthermore, line coverage decreases proportionally to the number of disabled instructions (set 1 → set 2 → set 3) because in CoreB its subroutines are no longer called and thus, the corresponding procedures in InstructionPkg are no longer executed.

**Line Coverage for the Test Bench:** In the test bench some code segments are implemented to supervise simulation. For instance, the syntax of the assembler code is checked during stimuli download. As the assembler code is correctly constructed for the pre-developed assembler data base several assert statements\(^7\) in the package InstructionPkgB are never executed. This results in a moderate line coverage for the test bench. However, failure handling routines do not actually influence the functional verification quality of the VC. Hence, moderate line coverage of the test bench is acceptable. As shown in tab. 5.3 changes of qualitative parameters (set 1, set 2, and set 3) have little influence on the line coverage of the test bench.

<table>
<thead>
<tr>
<th>Entity/Package</th>
<th>Supported Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set 1 (100%)</td>
</tr>
<tr>
<td>BF*</td>
<td>85.2%</td>
</tr>
<tr>
<td>CC†</td>
<td>98.9%</td>
</tr>
<tr>
<td>RA‡</td>
<td>98.9%</td>
</tr>
<tr>
<td>CoreB</td>
<td>92.3%</td>
</tr>
<tr>
<td>InstructionPkgB</td>
<td>88.5%</td>
</tr>
<tr>
<td>Test Bench</td>
<td></td>
</tr>
<tr>
<td>ModuleTbB</td>
<td>74.5%</td>
</tr>
<tr>
<td>IOPkgB</td>
<td>81.8%</td>
</tr>
<tr>
<td>ModuleTbPkgB</td>
<td>47.4%</td>
</tr>
</tbody>
</table>

**Table 5.3:** Line Coverage of Vector Generation Simulation Run

\(^7\)A VHDL assert statement checks that expected conditions are met within the model.
Results of Model Checking Simulation Run

In tab. 5.4 the measurements of line coverage for the model checking simulation run are listed. The table is again subdivided into the implementation model and the test bench section (see fig. 5.19).

For the implementation model which supports the full instruction set (set 1), 100% line coverage for all entities has to be attained hypothetically. Indeed, every use of a VHDL case statement can reduce line coverage as shown in the following. To represent logic signals the VHDL type std_logic having nine possible values [AWM97] ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '—') is used. For a case statement with an expression of the type std_logic nine different choices exist. To describe finite state machines only state transactions evoked by ones and zeros are of interest to define its functionality. However, to successfully synthesize the implementation model all choices have to be handled, otherwise the synthesizer aborts with an error. This is done in Pseudo-Code 5.7 with the default construct when others, in which all choices which have not been treated by then are pooled. During simulation the when others path is never followed. This results in a decrease of the line coverage, yet it has no negative influence on the verification quality.

Pseudo-Code 5.7: VHDL case statement

```vhdl
    case a is
        when '0' =>
            statement;
        when '1' =>
            statement;
        when others => /* {T/'/XVZVWVLVtfV—} */
            null;
    end case;
```

Disabled instructions have different impacts on the line coverage of the implementation model. The coverage rate of interface handling parts are not influenced. In the functional parts, line coverage of InstructionDecoder decreases more or less proportionally to less supported instructions. Other functional parts are not only sensitive to the quantity, but also to the quality of supported instructions. For set 2 which disables instructions from all classes (see appendix A), the line coverage decreases in several entities (in tab. 5.4

*Coverage of stimuli to verify basic functionality.
†Add on coverage of stimuli to verify corner cases.
‡Add on coverage of stimuli to verify real-world applications.
marked with a $\Delta$). This is opposed to set 3 in which further ALU instructions are disabled. Thus, the line coverage is only reduced in the entity $ALUData$ (marked with $\Box$), while the other functional parts are not influenced.

<table>
<thead>
<tr>
<th>Entity/Package</th>
<th>Supported Instruction Set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Set 1 (100%)</td>
</tr>
<tr>
<td></td>
<td>BF $^*$</td>
</tr>
<tr>
<td>Core</td>
<td>100.0%</td>
</tr>
<tr>
<td>InstructionDecoder</td>
<td>89.0%</td>
</tr>
<tr>
<td>Sequencer $^\Delta$</td>
<td>83.9%</td>
</tr>
<tr>
<td>ALUData $^\Box$</td>
<td>94.9%</td>
</tr>
<tr>
<td>DStack $^\Delta$</td>
<td>98.0%</td>
</tr>
<tr>
<td>AStack $^\Delta$</td>
<td>100.0%</td>
</tr>
<tr>
<td>RStack</td>
<td>100.0%</td>
</tr>
<tr>
<td>DMem</td>
<td>97.2%</td>
</tr>
<tr>
<td>ResetSleepCtrl $^\Delta$</td>
<td>85.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation Model (Interface Handling Parts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCodeInterface</td>
</tr>
<tr>
<td>DStackInterface</td>
</tr>
<tr>
<td>AStackInterface</td>
</tr>
<tr>
<td>RStackInterface</td>
</tr>
<tr>
<td>CStackInterface</td>
</tr>
<tr>
<td>DMemInterface</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Bench</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModuleTb.vhd</td>
</tr>
<tr>
<td>ModuleTbPkg.vhd</td>
</tr>
</tbody>
</table>

Table 5.4: Line Coverage of Model Checking Simulation Run

$^*$Coverage of stimuli to verify basic functionality.

$^\dagger$Add on coverage of stimuli to verify corner cases.

$^\ddagger$Add on coverage of stimuli to verify real-world applications.
Seite Leer / Blank leaf
Synchronization Issues

As defined in the previous chapter, a critical problem in simulation-based verification of virtual components (VC) is the synchronization of the test bench with customizable static and dynamic latencies of the implementation model. In this chapter several synchronization methodologies with respect to co-simulation for verification are discussed, resulting in "interface-specific activity scheduling". This approach handles interface latencies. Furthermore, the synchronization of interrupt launches in the two simulation runs (vector generation and model checking) are pointed out. To validate the quality of our test bench simulation results are shown in the second part of the chapter. They are measured on the test vehicle SPACEMAN.
6.1 Global Simulation Control

To verify SPACEMAN by simulation, a synchronization mechanism has to be implemented in the test bench. It controls the processing of different tasks in a well determined sequence (e.g. stimuli vectors have to be downloaded first, before the models can be executed). Therefore, the four-state “global” signal \( \text{SimStatus} \) is declared in the test bench. “Global” means that multiple sources, namely different processes, can modify the value of \( \text{SimStatus} \). In VHDL syntax this is obtained by declaring \( \text{SimStatus} \) as resolved\(^1\) signal.

**Definition:** \( \text{SimStatus} \supset \{ \text{init}, \text{download}, \text{run}, \text{terminate} \} \)

In *init* state, files are opened and variables, e.g. representing virtual memories, are initialised. In *download* state, stimuli vectors are fetched from the data base and stored in the virtual instruction memory. In these states the carbon model and the implementation model, respectively, are inactive in both verification environments (vector generation and model checking). Switching \( \text{SimStatus} \) to *run* state wakes up the models and the simulation enters the actual execution phase, until a \text{trap} instruction (see appendix A) triggers a state transition of \( \text{SimStatus} \) to *terminate*. In *terminate* state, simulation reports are completed, files are closed and the event queue is starved out, which terminates the simulation.

6.2 Interface Synchronization Strategies

The reference vectors generated by the data-driven carbon model have to be synchronized with the outputs of the clock-driven implementation model in order to compare them. One can imagine three different approaches:

- **Cycle-true output triggering**

\(^1\)The characteristic of a resolved signal is that several sources can concurrently assign different values in contrast to a common signal where concurrent assignments of different values result in the unknown value ('X'). Thus, in the definition of a resolved signal a dedicated function is necessary, i.e. the resolution function, used to calculate the dominant signal value on the basis of the values of all of its sources.
• Selective output triggering with one strobe
• Selective output triggering with multiple strobes

6.2.1 Cycle-true Output Triggering

Cycle-true output triggering strategy is the commonly used technique. Fig. 6.1 shows the basic idea. In every iteration loop of data-driven vector generation (execution of a stimulus), a dedicated snooper samples the expected responses of all primary outputs of the carbon model and stores them as reference vector. Afterwards, during clock-driven model checking all primary outputs of the implementation model are compared to the corresponding reference vector in every clock cycle. Snoopers are activated with the model triggering signals (i.e. stimuli in vector generation and the clock in model checking).

![Diagram of Cycle-true Output Triggering]

(a) Vector Generation  (b) Model Checking

**Figure 6.1:** Strategy: Cycle-True Output Triggering

Discussion

Cycle-true output triggering strategy is the most accurate of the three techniques, because in every simulation cycle all primary outputs of the model are monitored. But this approach is restricted to VCs where the carbon model
already suggests a cycle-based implementation. For VCs which meet this requirement other drawbacks exists:

- Some reference vectors have little information content. Consequently, these vectors contribute less to enhance functional verification quality. For example, primary outputs of a RAM interface are only of interest during execution of a read or write access. The rest of the time the outputs are don't care.

- The tight coupling of all data in a reference vector to a certain simulation cycle restricts interface customization of the implementation model. In model checking, interface types with different latencies entail a varying number of cycle shifts of their corresponding primary output data. These varying offsets result in failures during cycle-true vector comparison (see fig. 6.2, implementation model type 2).

Figure 6.2: Restrictions in Interface Customization using Cycle-true Synchronization
6.2.2 Selective Triggering with One Strobe

In fig. 6.3 the idea of selective output triggering with one strobe is illustrated. This approach groups the primary outputs of the model into several blocks, called interfaces, each of which is provided with a signal to activate the snooper. Thus, in contrast to cycle-true synchronization snoopers are not triggered in every simulation cycle, but selectively; i.e. only data which is relevant for functional verification is sampled and compared.

For vector generation this means that in a stimulus execution pass, reference data for all primary outputs are generated and stored as one reference vector if at least one of the interfaces is activated (logical or of all Req_i).

In model checking the snooper compares primary outputs to the expected responses not every clock cycle, but merely when any interface carries valid data (logical or of En_i). Hence, valid data cycles are indicated by enable strobes which are either existing nodes of the interface or additional signals used exclusively for simulation. Therefore, enable strobes have to be provided for all interfaces when the model is being programmed by the VC provider.

Discussion

This approach reduces the number of reference vectors without loss of relevant information for functional verification. However, subsequent to any interface activation data of all primary outputs is sampled in vector generation and compared in model checking, respectively. The interdependence of the interfaces still prevents the test bench from self-adapting to varying latencies.

6.2.3 Selective Triggering with Multiple Strobes

Selective output triggering with multiple-strobes strategy is built on the previously mentioned technique. Primary outputs are also grouped into different interfaces which are independently triggered by activation strobes (Req_i and En_i). One snooper is assigned to every interface (see fig. 6.4).

Therewith, different interface activities are decoupled and can be handled separately. As a consequence the reference vector data base is portioned into
several files (the number of files equals the number of interfaces); i.e. one reference vector file is assigned to every snooper.

In vector generation the snooper $i$ is triggered if its corresponding interface is activated (indicated by $Req_i$). Thereby, every snooper only stores data as reference vector of those primary outputs which belong to the interface.

In model checking a snooper is triggered if the corresponding interface of the implementation model carries valid data. After activation the snooper fetches an expected response vector from its database and compares it to those outputs of the implementation model which belong to the interface.

**Discussion**

In this approach an activated snooper does not store data of all primary outputs as references, but only data of those outputs which are assigned to the corresponding interface and thus carries relevant data for functional verification. This finer granularity of reference vectors and the selective activation of snoopers reduce the quantity of reference data as well as the number of comparisons. Furthermore, partitioning of primary outputs into different interfaces
Figure 6.4: Strategy: Selective Triggering with Multiple Strobes

and their independent handling makes the test bench tolerant to varying interface latencies which is a stringent condition for reusability of a test bench.
6.3 Interface-Specific Activity Scheduling

Selective triggering with multiple-strobes strategy has been presented in [SRT+99] under the name of "interface-specific activity scheduling". This synchronization strategy was implemented in the test benches to verify SPACE-MAN VC.

6.3.1 Interface Grouping

A prerequisite for interface-specific activity scheduling is to find an adequate grouping of the primary VC outputs in different interfaces. Parameterization properties of the model have a strong impact on this partitioning, as will be shown.

![Figure 6.5: VC Parameterization: Different Latencies](image)

**Different latencies:** Some VC parameters modify the latencies of data paths (e.g. customizable number of stack registers with back-up memories or customizable number of pipeline stages to adjust performance for time critical units). In fig. 6.5, on the left, the problem resulting for interface grouping is illustrated. While data path $p_1$ is not affected by customization, data path $p_2$ is either pipelined or not, dependent on the parameter settings chosen. Merging the outputs of $p_1$ and $p_2$ into one interface which is triggered by $Enable_{p1}$ results in correct comparisons to references only for the unpipelined variant.
For the pipelined variant signals belonging to \( p_2 \) are snooped one cycle too early. To resolve this conflict, output signals of \( p_1 \) and \( p_2 \) are partitioned into two separate interfaces (see fig. 6.5, on the right). For the interface belonging to \( p_2 \) a separate trigger signal, \( Enable_{p2} \), has to be implemented in the model.

**Different functionality:** Other parameters modify the functionality of the model (e.g. instructions in a processor might be disabled, and thus result in unused outputs). In fig. 6.6, on the left, an example is shown. Dependent on the VC parameter settings, outputs of data path \( p_2 \) either carry valid data or the function \( F_2 \) is disabled and its corresponding outputs stunted. In the later case memory resources can be saved and compare operations disabled which would otherwise slow down simulation unnecessarily. This can also be avoided by partitioning outputs of \( p_1 \) and \( p_2 \) in two separate interfaces (see fig. 6.6, on the right).

\[
\begin{array}{c}
\text{Module} \\
\text{Core} \\
P_1 \quad \ldots \quad P_2 \\
F_1 \quad F_2 \\
\text{Interface} \\
\text{Enable}_{p1} \\
\text{Enable}_{p2} \\
\text{Reference Vectors} \\
\text{Snooper} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Module} \\
\text{Core} \\
P_1 \quad \ldots \quad P_2 \\
F_1 \quad F_2 \\
\text{Interface} \\
\text{Enable}_{p1} \\
\text{Enable}_{p2} \\
\text{Reference Vectors} \\
\text{Snooper} \\
\end{array}
\]

\[\text{□ : Customization Properties} \quad F_i : \text{Combinational Block}\]

**Figure 6.6:** VC Parameterization: Different Functionality

Interface grouping supposes a detailed knowledge of the architecture and customization properties of the model. Therefore, it has to be part of the VC model realized by the VC provider.

### 6.3.2 Synchronization Issues of Vector Generation

SPACEMAN is subdivided into six different interfaces (instruction memory, data memory, data stack, address stack, return address stack, and condition
code stack). To support interface-specific activity scheduling for every interface one complex snooper (see fig. 5.11) is instantiated in the test bench. The main parts of its implementation are illustrated in Pseudo-Code 6.1.

**Pseudo-Code 6.1: Snooper in Vector Generation**

```pseudo
process VectorGeneration {
    wait until SimStatus = run;
    while (SimStatus = run) {
        wait on CMRequest;
        case CMRequest is {
            when read ⇒
                writeFile('0', CMAddress);
                /* Complex Snooper */
                CMData = VirtualMemory(CMAddress);
            when write ⇒
                writeFile('1', CMAddress, CMData);
                /* Complex Snooper */
                VirtualMemory(CMAddress) = CMData;
        end case;
        TBAcknowledge = ready;
        wait until CMRequest = idle;
        TBAcknowledge = idle;
    }
    wait;
}
```

After every interface activation of the carbon model the corresponding snooper is triggered (CMRequest). In the case of a memory read access the snooper stores the memory address as reference vector. When a write access occurs, besides the address, data is also stored in the reference vector file. To distinguish a read from a write access all vectors are marked with a flag ('0' for a read access and '1' for a write access). At the end of the process the full-handshake protocol with the carbon model is accomplished before the snooper returns in sleep state again.

The basic structure of the snooper process is identical for all kinds of interfaces. Only the number and types of parameters have to be matched in the writeFile() procedure. Therefore, in SPACEMAN verification environment, the procedure writeFile() is encapsulated in a separate package `ModuleTbPkg` (see fig. 5.6) where it is defined as overloaded\(^2\). Overloading writeFile() supports the reuse of snoopers: The procedure call is the

---

\(^2\)Overloaded procedures have the same identifier, but different numbers and different types of formal parameters. Depending on the parameters of a subroutine call, the corresponding procedure is invoked.
same for all interface customizations, only the type declaration of its parameters has to be changed. Furthermore, thanks to the encapsulation the VC user can add new variants of the procedure by locally modifying it in ModuleTbPkg.

6.3.3 Synchronization Issues of Model Checking

During model checking simulation the processes in the test bench are synchronized with the clock. Pseudo-Code 6.2 shows the structure of the clock process:

**Pseudo-Code 6.2: Clock Generation**

```pseudocode
process ClockGeneration {
    wait for INIT_DELAY;
    loop {
        Clock ^'V;
        wait for CLOCK_HIGH_PHASE;
        Clock <= '0';
        wait for CLOCK_LOW_PHASE:
        if (SimStatus = terminate) {
            wait;
        }
    }
}
```

Similar to vector generation six complex snoopers (see fig. 5.11) are implemented in the test bench of model checking. The main part of its implementation is shown in Pseudo-Code 6.3. The snooper only compares reference vectors to the outputs of the implementation model if the corresponding interface of the implementation model is activated. This is indicated by setting IMEnable to high. If the snooper is activated a reference vector is read. ReadFlag indicates whether the vector corresponds to a read or write memory access. Depending on this flag, either only the address is compared (read) or address and data (write) are compared.

The basic structure of the snooper process is similar for all kinds of interfaces. Besides differences in the number and types of parameters of overloaded test bench support procedures readFile()³, check()⁴, and

³ readFile(): Reads a reference vector from the data base.
⁴ check(): Compares an entry of the reference vector to the corresponding output of the implementation model.
Pseudo-Code 6.3: Snooper in Model Checking

```plaintext
process VectorComparison {
    wait until SimStatus = run;
    wait on Clock until Clock = '1';
    wait for RESPONSE_ACQUISITION;
    while (SimStatus = run) {
        if (IMEnable = '1') {
            readFile(readFlag, address, data);
            if (readFlag = '0') {
                if not check(IMAddress, address)
                    writeFailureReport(IMAddress, address);
            } else {
                if not check(IMAddress, address)
                    writeFailureReport(IMAddress, address);
                if not check(IMData, data)
                    writeFailureReport(IMData, data);
            }
        } else {
            writeFailureReport();  
        }
    }
} wait;
```

writeFailureReport()⁵, the timing dependencies of output signals in the snoppers have to be emulated for different interface protocols. Consider the example in fig. 6.7. After triggering the snooper with IMEnable, Output2 has to be compared in the cycle n. In contrast to Output2, relevant data of Output1 is visible one cycle later (n + 1) and thus its comparison against reference data has to be delayed one cycle.

### 6.4 Synchronization of Interrupt Launching

In SPACEMAN verification environment, besides assembler instructions, another primary stimuli source exists: interrupts. The interrupt interface of the carbon model as well as of the implementation model consists of the interrupt entry address, and signals to support a full-handshake protocol to launch interrupts.

Two scenarios are distinguished for interrupt generation in simulation:

⁵writeFailureReport(): Writes a message to the report file in case of a failure detected.
6.4 Synchronization of Interrupt Launching

- SPACEMAN is embedded into its target environment and the complete system is verified
- SPACEMAN is verified as stand-alone model

If SPACEMAN is embedded into the target environment, interrupts are generated by other VC models in this environment. This case will be introduced in chapter 7. For stand-alone VC verification, the test bench has to generate interrupts which requires an interrupt scheduler to be implemented.

### 6.4.1 Interrupt Format

To support test bench reuse, it is essential to define interrupt launch directives in a generic way which are easy manageable by the VC user. In SPACEMAN interrupt launches are initiated within assembler stimuli files. In this approach neither the test bench nor the VC has to be modified to add or remove interrupts sequences.

An interrupt launch directive is defined by the following triple:

% <start cycle>, <repeat cycle>, <interrupt entry address>

To mark an interrupt directive it starts with the comment character %. The next entry defines the simulation cycle during which the interrupt is launched the first time. The following number indicates the repeat period. If an interrupt
is launched just once this number is to be set to zero, otherwise interrupts are repetitively launched: \(\text{launch cycle} = \text{start cycle} + n \times \text{repeat cycle}, \ n \in \mathbb{N}\). The last number specifies the interrupt entry address.

### 6.4.2 Interrupt Scheduler

In fig. 6.8 the functionality of the interrupt scheduler is illustrated. During simulation download, interrupt directives are extracted from the assembler stimuli data base and the corresponding data triples are stored in ascending sequence of their start cycles in a FIFO. Two pointers define the start (\(\text{readCnt}\)) and the end (\(\text{writeCnt}\)) addresses of these entries. In the execution phase of simulation the interrupt launch cycle of the first entry is compared to the current simulation cycle. After a hit the interrupt is triggered. If the interrupt is defined as repetitive its next launch cycle is calculated and ranged into the proper location of the FIFO. Otherwise the entry is deleted from the FIFO by incrementing \(\text{readCnt}\).

The interrupt scheduler is implemented as single process in the test bench. It is laid out for reuse in vector generation and model checking with as few modifications as possible (see Pseudo-Code 6.4, Pseudo-Code 6.7, and Pseudo-Code 6.8). Two parts in the scheduler are not reusable: \(\text{iRQ\_Synchronize}\) and \(\text{iRQ\_Launch}\). They accomplish the synchronization with other processes of the test benches for the different timing mechanisms of the two simulation runs. \(\text{iRQ\_Synchronize}\) (see Pseudo-Code 6.5) defines different start-up mechanisms of the interrupt scheduler for the two simulation runs.

### Interrupt Launching

When interrupt launches are assigned to simulation cycles several synchronization problems occur. All these problems come from the handling of exception mechanisms which are presented only in the implementation model:

**Synchronization with Simulation Cycles:** For example, as mentioned in chapter 4, one method to handle hazards is to stall the implementation model for one or several simulation cycles. Such a stall leads to a larger number of executed stimuli for the carbon model compared to the implementation model (see fig. 6.9, upper part). After an interrupt launch the stimuli sequences are
resynchronized (i.e. both models execute the same stimuli in equivalent simulation cycles again). As a consequence in model checking, the reference vectors appear in a “wrong” sequence. This results in wrong comparisons.

**Synchronization with Stimuli:** A better methodology to synchronize vector generation with model checking is to make interrupt launches depend on stimuli vectors instead of simulation cycles. In fig. 6.9 below this approach
Pseudo-Code 6.4: Interrupt Handling

```
process Interrupt_Scheduler {
    variable upperBoundary, lowerBoundary, middle : natural;
    variable launchCycle : natural;
    variable i : natural;
    iRQSynchronize;
    while (iRQ_fifo.readCnt ≠ iRQ_fifo.writeCnt) {
        /* interrupt launch */
        iRQLaunch;
        /* interrupt rearrangement */
        if (iRQ_fifo.iRQElement(iRQ_fifo.readCnt).repeatCycle > 0) {
            upperBoundary = iRQ_fifo.writeCnt;
            lowerBoundary = iRQ_fifo.readCnt;
            launchCycle = iRQ_fifo.iRQElement(iRQ_fifo.readCnt).startCycle +
                iRQ_fifo.iRQElement(iRQ_fifo.readCnt).repeatCycle;
            while ((upperBoundary − lowerBoundary) modulo FIFO_SIZE > 1) {
                middle = (lowerBoundary +
                    ((upperBoundary − lowerBoundary) modulo FIFO_SIZE) / 2) modulo FIFO_SIZE;
                if (launchCycle > iRQ_fifo.iRQElement(middle).startCycle)
                    lowerBoundary = middle;
                else
                    upperBoundary = middle;
            }
            if (middle = lowerBoundary) {
                if (launchCycle > iRQ_fifo.iRQElement(upperBoundary).startCycle)
                    middle = upperBoundary;
            } else {
                if (launchCycle < iRQ_fifo.iRQElement(lowerBoundary).startCycle)
                    middle = lowerBoundary;
            }
            i = iRQ_fifo.writeCnt;
            while (i ≠ middle) {
                iRQ_fifo.iRQElement(i) = iRQ_fifo.iRQElement((i − 1) modulo FIFO_SIZE);
                i = (i − 1) modulo FIFO_SIZE;
            }
            iRQ_fifo.writeCnt = (iRQ_fifo.writeCnt + 1) modulo FIFO_SIZE;
            iRQ_fifo.iRQElement(i) = iRQ_fifo.iRQElement(iRQ_fifo.readCnt);
            iRQ_fifo.iRQElement(i).startCycle = launchCycle;
        }
        iRQ_fifo.readCnt = (iRQ_fifo.readCnt + 1) modulo FIFO_SIZE;
    }
    wait;
}
```

is shown. In both simulation runs the number of applied assembler stimuli vectors are counted. Thereby, an interrupt is launched if the stimuli counter is equal to the launch cycle. With this approach a stall in the implementa-
6.4 Synchronization of Interrupt Launching

**Pseudo-Code 6.5: iRQSynchronize**

/* part used in vector generation */
wait until SimStatus = run;

/* part used in model checking */
wait until SimStatus = run;
wait on Clock until Clock = '1';
wait for STIMULI_APPLICATION;

**Pseudo-Code 6.6: iRQLaunch**

/* part used in vector generation */
while (currentCycle < iRQFifo.iRQElement(iRQFifo.readCnt).startCycle or iRQBranchFlag or iRQDelayFlag) {
    wait on currentCycle;
}
IRQAddress <= iRQFifo.iRQElement(iRQFifo.readCnt).iRQAddress;
TBiRQLaunch <= true;
wait until CMiRQAcknowledge;
TBiRQLaunch <= false;
wait until not CMiRQAcknowledge;

/* part used in model checking */
while (currentCycle < iRQFifo.iRQElement(iRQFifo.readCnt).startCycle) {
    wait on currentCycle;
}
IRQAddress <= iRQFifo.iRQElement(iRQFifo.readCnt).iRQAddress;
TBiRQLaunch <= '1';
wait for CYCLE_TIME;
while (IMiRQAcknowledge ≠ '1') {
    wait for CYCLE_TIME;
}

**Pseudo-Code 6.7: Definition of iRQElementType**

type iRQElementType is
    record
        startCycle : natural;
        repeatCycle : natural;
        iRQAddress : natural;
    end record;

This model delays the interrupt launch and therefore reference vectors always occur in the correct sequence.
Figure 6.9: Interrupt Launch Technique

**Pseudo-Code 6.8:** Definition of `iRQFifo` ∈ `iRQFifoType`

```plaintext
type iRQFifoType is
record
    readCnt : natural;
    writeCnt : natural;
    iRQElementType
end record;
```
Further problems which concern interrupt launching are:

- Branch misses in the implementation model due to pipeline flushes. After a flush the pipeline has to be refilled before instruction execution is carried on.

- Interrupt delay mechanisms which are implemented to avoid data inconsistencies. A data inconsistency occurs if a branch instruction is in the execution stage of the pipeline in the simulation cycle where an interrupt launch demands for a context switch.

To resolve these problems the carbon model predicts this kind of exceptions by setting two flags \( iRQBranchFlag \) and \( iRQDelayFlag \). These flags delay an interrupt launch in the interrupt handler during vector generation simulation (see Pseudo-Code 6.6, vector generation part).

### 6.5 Simulation Results

In this section simulation efficiency of the verification flow is introduced. SPACEMAN with the parameter setting used for the prototype integration (see tab. 4.6) is taken as test vehicle. Measurements are performed on three platforms:

- ModelSim-5.2 from Mentor (compiled)
- VSS-1998.08 from Synopsys (compiled)
- VSS-1998.08 from Synopsys (interpreted)

#### 6.5.1 Simulation Run Times for Code Download

In fig. 6.10 simulation run times for stimuli download are illustrated in logarithmic scale for vector generation and model checking.

Although stimuli download is significantly more expensive in vector generation than in model checking, due to assembler parsing and editing, instead
of just reading binaries, simulation run time is nevertheless shorter. In vector generation the carbon model is suspended during code download. This is opposed to model checking where besides the implementation model, VHDL RAM models are instantiated in the test bench. These models require the clock to store data. However, the enabled clock process in the test bench also triggers all sequential processes of the implementation model, and thus causes a large number of events slowing down simulation.

![Graphs of Run Times of Simulations](image)

**Figure 6.10:** Code Download: Run Times of Simulations

### 6.5.2 Benchmark Programs

To measure simulation run times for model execution two different benchmark programs were written. In tab. 6.1 instruction compositions for both programs are shown (the instruction classes are defined in appendix A). They result in the interface activation probabilities per stimuli execution pass or clock cycle, respectively, of tab. 6.2. Some instructions activate no interface (e.g. nop), other instructions activate one interface (e.g. pushA address) or even two interfaces (e.g. pushAlop dx). Because no interrupts are launched in both benchmark programs the condition code stack interface is never activated.

Adding up all interface activation probabilities for the two assembler pro-
results in a two-times higher interface activation probability per stimuli execution pass or clock cycle for benchmark2 compared to benchmark1.

\[
\sum_{n=1}^{7} P(n)
\]

<table>
<thead>
<tr>
<th>Instruction Class</th>
<th>Instruction Composition</th>
<th>Benchmark1</th>
<th>Benchmark2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address ALU</td>
<td>6.8%</td>
<td>13.6%</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>11.3%</td>
<td>13.6%</td>
<td></td>
</tr>
<tr>
<td>Data Memory</td>
<td>18.2%</td>
<td>31.8%</td>
<td></td>
</tr>
<tr>
<td>Stack Access</td>
<td>11.3%</td>
<td>25.0%</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>11.4%</td>
<td>6.8%</td>
<td></td>
</tr>
<tr>
<td>Data ALU</td>
<td>41.0%</td>
<td>9.2%</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Instruction Partitioning of Benchmark Programs

Both benchmark programs require approximately 500kCycles for execution. Measurements were performed on an Sun Ultra 60 (360MHz).

<table>
<thead>
<tr>
<th>Interface Type</th>
<th>Activation Probability, (P(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Benchmark1</td>
</tr>
<tr>
<td>Single Interface Activations</td>
<td>(1)^6</td>
</tr>
<tr>
<td>Instruction Memory</td>
<td>0.045</td>
</tr>
<tr>
<td>Data Memory</td>
<td>0.068</td>
</tr>
<tr>
<td>Data Stack</td>
<td>0.023</td>
</tr>
<tr>
<td>Address Stack</td>
<td>0.045</td>
</tr>
<tr>
<td>Return Address Stack</td>
<td>0</td>
</tr>
<tr>
<td>Condition Code Stack</td>
<td>0</td>
</tr>
<tr>
<td>Double Interface Activations</td>
<td>0.136</td>
</tr>
<tr>
<td>Data Memory &amp; Data Stack</td>
<td>0.023</td>
</tr>
<tr>
<td>Data- &amp; Address Stack</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Table 6.2: Interface Activation of Benchmark Programs

\(^6\)Separate handling of the permanently enabled instruction memory interface.
Results of a test bench with selective triggering by multiple strobes synchronization technique are compared to the results of a test bench with cycle-true synchronization and selective triggering with one strobe. Thereby, the instruction interface is handled separately. Otherwise measurements of cycle-true synchronization and selective triggering with one strobe would be almost equal because the instruction interface is always activated (except in the case of a model stall).

### 6.5.3 Reference Vector Sizes

In fig. 6.11 total file sizes\(^7\) of reference vectors for different synchronization techniques are compared. The reference data size is noticeably reduced with strobe synchronization techniques compared to the cycle-true approach (74% and 62%, respectively for selective triggering with one strobe, 87% and 84%, respectively for selective triggering with multiple strobes).

However, in strobe synchronization technique the reference vector size heavily depends on the number of interface activations. Although both benchmark programs execute approximately the same number of instructions, the double number of interface activations for benchmark2 results in a significant larger reference vector file size (46% increase for selective triggering with one strobe and 26% increase for selective triggering with multiple strobes).

### 6.5.4 Run Times for Vector Generation

In fig. 6.12 simulation run times for vector generation on the three platforms with different synchronization techniques are shown.

As mentioned in chapter 5 the carbon model is designed to obtain short run times in simulation. This characteristic involves that the file handling of the test bench requires a considerable part of the overall simulation run time in vector generation. Therefore, a noticeable correlation between run times and corresponding files sizes (see fig. 6.11) can be ascertained.

However, there exist major differences in relative savings of simulation run time between different software platforms when using the same synchro-

\(^7\)File sizes include reference data of the instruction interface.
nization techniques. Probably, they result from different implementations of HDL file handling routines in the different simulators.

Figure 6.11: File Sizes of Reference Vectors

Figure 6.12: Vector Generation: Run Times of Simulations
6.5.5 Run Times for Model Checking

In fig. 6.13 model checking simulation run times on the three platforms are shown for different synchronization techniques.

In model checking HDL file handling and the comparison of reference vectors to the outputs of the implementation model require less computing time than the processing of the implementation model. Therefore, when comparing the different synchronization strategies relative savings in simulation run times are smaller on all platforms (for selective triggering with multiple strobes compared to cycle-true: 5% up to 11%, and compared to selective triggering with one strobe: 2% up to 4%).

![Bar charts for different models and strategies showing simulation run times.

(a) Benchmark1

(b) Benchmark2

Figure 6.13: Model Checking: Run Times of Simulations

6.5.6 Discussion

The proposed synchronization technique, selective triggering with multiple strobes, reduces not only the size of reference vectors but also speeds up simulation. Furthermore, this technique supports self-adaptation of the test bench to the varying output latencies of differently customized virtual components, thus allowing reuse of the test bench.
Simulation of the behavioral VHDL carbon model is at least one magnitude faster than simulation of the implementation model. Therewith, the VC user has a "high-speed" behavioral model to evaluate the best parameter settings for the final implementation model in several trials.

Furthermore, the carbon model favors concurrent software and hardware development. On the one hand software engineers have a platform to develop and debug assembler codes, and on the other hand hardware designers have a behavioral description of the system specification for the functional verification of the implementation model.

The exclusive use of VHDL avoids restrictions concerning foreign-language interface handling of HDL simulators. Furthermore, there are no software platform-dependent restrictions in the supported language set.
Verification Flow for a Multi-VC Design

The previous chapters introduce a methodology to verify stand-alone VCs which results in a self-configurable and therefore reusable test bench. Based on this VC verification environment, a test bench to functionally verify a system composed of multiple VCs is elaborated in this chapter. The first part introduces a multi-VC test vehicle consisting of SPACEMAN and two VCs which form a data path. Then, on the basis of this system, test benches for vector generation and model checking are elaborated. Thereby, the emphasis is put on the possibilities for reuse of the stand-alone VC test benches in the system test bench.

7.1 System Test Vehicle

A test vehicle to elaborate a methodology for the functional verification of a system composed of several VC has been designed. Besides SPACEMAN, it
consists of sections of the ITRACHIP [TSR+99] data path and the interrupt controller as link between. Thus, the test system comprises three different VC types (see fig. 7.1), namely:

- Data path (VC1, VC2)
- SPACEMAN (VC3)
- Interrupt controller (VC4)

**Data path**

The STM-1 termination block of ITRACHIP is chosen as data path VC (see fig. 4.5). This block handles a continuous data input stream. Overhead information is evaluated and gets removed from the data stream, while the payload passes the block unchanged. Depending on the evaluated overhead information, pointers are processed (see chapter 4.2.2) and low level failure checks are executed.

For this project the data path block has been redesigned as parameterizable VC. The VC can be customized, similar to SPACEMAN, in all three parameterization classes (qualitative, quantitative and interface). However, most important is qualitative parameterization, enabling the user to select between two different overhead evaluation protocols (STM-1 and STS-3) [IT97]. Besides the implementation model the VC provides the behavioral carbon model. The core of this carbon model sequentially invokes several procedures, each handling a different part of the overhead data. As opposed to SPACEMAN, the implementation model of the data path VC is triggered by two different clocks and therefore a synchronization mechanism between the two clock domains was implemented.

The prototype system comprises two different data paths, one parameterized as STM-1 (VC1 in fig. 7.1) and the other as STS-3 (VC2 in fig. 7.1).

**SPACEMAN**

The embedded controller VC of the test system is SPACEMAN (VC3 in fig. 7.1). SPACEMAN gets overhead data from the data path VC and processes high-level filter functions [IT97]. Its data memory interface accesses three different
blocks: data memory, VC1, and VC2. The SPACEMAN test vehicle parameter set is shown in tab. 4.6. However, a smaller data memory and instruction memory address range (512Bytes and 1kByte, respectively) has been chosen.

**Interrupt Controller**

SPACEMAN (VC3) is coupled to the data paths (VC1 and VC2) through the interrupt controller (VC4 in fig. 7.1). As opposed to the priority-driven interrupt scheduling in ITRACHIP (see section 4.2.5), a simplified interrupt scheduling strategy is realized in the test vehicle. No priorities are assigned to the different interrupt sources. Instead, if a data path VC has evaluated a new overhead data set, it sends a request to the interrupt controller which directly launches an interrupt in SPACEMAN’s program execution. Thus, the interrupt routine of the last requesting source is executed (last come, first serve).

![Diagram of System Test Vehicle](image)

**Figure 7.1: System Test Vehicle**

In the next two sections, test benches of vector generation and model checking are introduced for the functional verification of the system test vehicle.
7.2 Vector Generation

In vector generation the system is formed by plugging together the carbon models of the different VCs. The resulting system carbon model then generates reference vectors to verify the customizable implementation model of the system.

7.2.1 Separation of Data Path and Controller

As mentioned in the previous two chapters carbon models are activated with stimuli (data-driven) for vector generation. A new stimulus triggers a pass in the core loop of a carbon model where its functionality and data exchanges with the test bench or other models are sequentially executed; i.e. each carbon model of the system is activated by different stimuli and therefore each of the models has its own synchronization mechanism. Thus, a methodology has to be elaborated to control the execution sequence of the different carbon models when simulating the entire system.

In fig. 7.2 the basic idea of our approach is illustrated by taking a generic system carbon model as example. This model comprises four data paths (shaded parts in fig. 7.2). Each data path is formed by one or several carbon models (CM_{DP}). To some of these data path models a controller is assigned (CM_{C}). Simulation is executed in the following way: A stimulus is applied to the primary input of data path one, which triggers the first carbon model of data path one (CM_{DP1} in fig. 7.2). After the sequential execution of this model data appears at its outputs and then activates the next carbon model in data path one (CM_{DP2}). This is indicated in fig. 7.2 by A. If CM_{DP1} has not generated output data in this pass, CM_{DP2} is not activated and the execution of data path one is stopped (B in fig. 7.2). After processing data path one, the procedure is repeated for the next data path, and so on. If the last data path is processed one iteration of vector generation is finished.

The top down execution of the data paths can be interrupted by calling controller carbon models (CM_{C}). That means every CM_{DP} to which a controller is assigned can evoke this controller at some point during its execution (e.g. as shown with C in fig. 7.2). Thereby, some controllers are shared by several CM_{DP}, as for instance CM_{C1} in fig. 7.2. In other cases a controller can evoke another controller such as (CM_{C2} \rightarrow CM_{C3}).
Discussion

In this approach the system carbon model is subdivided into several data paths which are successively processed, and in controller parts which can be evoked on demand by data path carbon models [RST+00]. All carbon models of the system are embedded into one loop. One can imagine the execution of this loop by means of the passing of an “activation token” from one carbon model to the next. Thus, only one carbon model is active at the same time, while the other models are suspended. The handing over of the “activation token” from one model to the next is realized by connecting the two models with a full handshake protocol (similar to the approach introduced in section 5.4.5). Thereby, the actually activated model sends a request to the model which gets activated next and it is then suspended as long as this model returns an acknowledgment.

Conditions

This methodology can only be used if the carbon model meets two conditions:
• No data dependencies between any two data paths may exist. In fig. 7.3, on the left, an example shows two interdepending data paths. In this case the successive activation sequence \(CM_{-} DP1 \rightarrow CM_{-} DP2 \rightarrow CM_{-} DP3 \rightarrow CM_{-} DP4\) may result in wrongly expected responses, because \(CM_{-} DP2\) depends on data of \(CM_{-} DP3\) which are not yet processed in the current iteration.

• In fig. 7.3, on the right, another restriction is illustrated. Two data paths, \(CM_{-} DP\), share one controller, \(CM_{-} C\); a typical example therefor is an interrupt-driven ASIP, in which each data path can launch an interrupt. In this case \(CM_{-} C\) has to independently handle data from the two \(CM_{-} DP\). Otherwise, depending on the sequence of \(CM_{-} C\) activation, the controller processes different results. For the ASIP \(CM_{-} C\) example data separation of the two data paths can be realized by subdividing the data memory address space into several parts and by assigning one part to every \(CM_{-} DP\). Furthermore, for every \(CM_{-} DP\) a separate program code is implemented in the ASIP which exclusively operates on the corresponding part of the data memory. Therewith, \(CM_{-} C\) is divided into two independently operating controllers and its activation sequences is of no importance anymore.

![Figure 7.3: Condition for the Carbon System Execution Methodology](image)
## 7.2.2 Test Bench

To verify the above mentioned activation methodology for the system test vehicle a test bench has been realized. Fig. 7.4 shows the block diagram of this test bench. It is partitioned into the system test bench (light grey part) and the system carbon model (white part).

**Loaders:** The stimuli for the two data paths (CM1 and CM2 in fig. 7.4) are generated by dedicated SDH generators and stored in a numerical format. Therefore, no stimuli converter is necessary and the continuous SDH data stream [IT97] demands for the request-driven strategy (see section 5.3.1) to realize loader1 and loader2. For SPACEMAN (CM3 in fig. 7.4) the loader used for the stand-alone verification (see section 5.4.2) is reused for the system test bench.

**Snoopers:** To sample reference vectors for the two data paths (CM1 and CM2) two primary snoopers (generator; introduced in fig. 5.12a) are implemented in the system test bench. For SPACEMAN, CM3, the complex snoopers introduced in section 6.3.2 are reused. As opposed to the stand-alone verification of SPACEMAN where reference vectors from all data stacks are sampled, only reference data of the data memory and the instruction memory are sampled in the system test bench, because the stacks are actually an internal part of SPACEMAN CM. Therefore, they are not relevant for system consideration in which inter-CM data transfer and data transfer between CMs and primary I/Os are monitored.

All data transfers between a block of the system test bench and a block of the system carbon model are executed with full-handshake protocols (see fig. 5.15).

### Simulation Control

In section 6.1 the four-state signal SimStatus (init, download, run, and terminate) has been introduced. This methodology is also used to control the system test bench. In init state, files are opened and variables are initialized. In download state, loader3 transfers assembler code from the stimuli data base into the virtual instruction memory (VMI) while all other parts of the test bench and
the carbon model are inactive. In run state, all parts are active and reference vectors are generated. Thereby, CM3 is stimulated with data from the test
bench internal VM1, while CM1 and CM2 are stimulated with SDH data read from their corresponding SDH stimuli source. Vector generation simulation is either terminated when a data path stimuli source is out of data or the assembler instruction trap is executed by CM3. Thus, assembler code debugging is efficiently supported.

As shown in fig. 7.4 several arbiters are necessary to connect two CMs as well as a CM and a VM. Three different types can be distinguished:

- Download arbiter
- Activation arbiter
- Bus arbiter

**Download Arbiter:** The download arbiter is implemented in the system test bench. It controls the access of VM1 during the different simulation phases defined by SimStatus.

**Activation Arbiter:** The activation arbiter is a part of the system carbon model (see fig. 7.4) needed when at least two different data path CM access the same controller CM. Thus, the activation controller represents the interrupt controller in vector generation simulation. Its core functionality is illustrated by Pseudo-Code 7.1. Two different interrupt sources, CM1 and CM2, can trigger the activation controller. After triggering, the “activation token” is forwarded from the activation arbiter to CM3, which executes that assembler routine corresponding to the interrupt source. After execution, the “activation token” returns from CM3 to the activation arbiter (full-handshake termination with CM3) which passes it back to the requesting interrupt source (full-handshake termination with CM1 or CM2, respectively).

**Bus Arbiter:** The bus arbiter is also a part of the system carbon model. It manages data transfers between a controller and different memory blocks. In fig. 7.4 the data memory address space of the controller VC3 is subdivided into three different parts: the actual data memory, VM2, and two memory mapped register blocks implemented in CM1 and CM2, respectively (VM3 and VM4). Pseudo-Code 7.2 illustrates the core functionality of the corresponding bus arbiter and fig. 7.5 shows a block diagram of this bus arbiter and the connected
Pseudo-Code 7.1: Activation Arbiter

```cpp
process ACTIVATION_ARBITER {
    wait until SimStatus = run;
    loop {
        wait on CM1Request, CM2Request;
        if CM1Request = set {
            /* full handshake with controller (CM3) */
            CM3Request <= set;
            wait until CM3Acknowledge = set;
            CM3Request <= idle;
            wait until CM3Acknowledge = idle;
            /* full handshake with data path 1 (CM1) */
            CM1Acknowledge <= set;
            wait until CM1Request = idle;
            CM1Acknowledge <= idle;
        }
        else {
            /* full handshake with controller (CM3) */
            CM3Request <= set;
            wait until CM3Acknowledge = set;
            CM3Request <= idle;
            wait until CM3Acknowledge = idle;
            /* full handshake with data path 2 (CM2) */
            CM2Acknowledge <= set;
            wait until CM2Request = idle;
            CM2Acknowledge <= idle;
        }
    }
}
```

models as well as the data memory address partitioning. A data memory access of CM3 is performed as follows: CM3 sends a request to the bus arbiter. Depending on the address and the access mode (read or write) the bus arbiter starts a full-handshake data transfer with one of the virtual memories. When this transfer is completed the "activation token" returns to CM3.

Model Triggering

During the execution phase of the system carbon model, stimuli from the SDH data bases are alternately applied to the two data paths (CM1 and CM2 of fig. 7.4). In Pseudo-Code 7.3 this loop which is implemented in the system test bench is shown. A new stimulus is read from the SDH data base (db1 or db2) and applied to the corresponding data path (CM1 or CM2).
Pseudo-Code 7.2: Bus Arbiter

```plaintext
process BUS_ARBITER {
    wait until SimStatus = run;
    loop {
        wait on CM3BusReq;
        if CM3BusReq = read {
            /* read access */
            if CM3BusAddress >= VM3BND {
                /* CM3 access VM2 */
                VM2BusAddress ← CM3BusAddress modulo (VM2BND - VM3BND);
                VM2BusReq ← read;
                wait until VM2BusAck = ready;
                CM3BusDataIn ← VM2BusDataOut;
                VM2BusReq ← idle;
                wait until VM2BusAck = idle;
                CM3BusAck ← idle;
                CM3BusAck ← idle;
            } elsif CM3BusAddress >= VM4BND {
                /* CM3 access VM3 */
                VM3BusAddress ← CM3BusAddress modulo (VM3BND - VM4BND);
                ...
            } else {
                /* CM3 access VM4 */
                VM4BusAddress ← CM3BusAddress modulo (VM4BND);
                ...
            }
        } else {
            /* write access */
            if CM3BusAddress >= VM3BND {
                /* CM3 access VM2 */
                VM2BusAddress ← CM3BusAddress modulo (VM2BND - VM3BND);
                VM2BusDataIn ← CM3BusDataOut;
                VM2BusReq ← write;
                wait until VM2BusAck = ready;
                VM2BusReq ← idle;
                wait until VM2BusAck = idle;
                ...
            } elsif CM3BusAddress >= VM4BND {
                ...
            } else {
                ...
            }
        }
    }
}
```
Thereby, the “activation token” is passed to the data path; i.e. the main loop is suspended until the full-handshake is finished with the data path and the “activation token” returns to the main loop. This procedure is then repeated for the next data path, and so on.

![Block Diagram of Bus Arbiter](image)

**Figure 7.5: Block Diagram of Bus Arbiter**

**Pseudo-Code 7.3: Stimuli Application**

```plaintext
process STIMULI.APPLICATION {
    wait until SimStatus = run;
    while (SimStatus ≠ terminate) {
        read(db1, stimuli);
        CM1DPData ← stimuli;
        CM1DRequest ← write;
        wait until CM1DPAcknowledge = ready;
        CM1DRequest ← idle;
        wait until CM1DPAcknowledge = idle;
        read(db2, stimuli);
        CM2DPData ← stimuli;
        CM2DRequest ← write;
        wait until CM2DPAcknowledge = ready;
        CM2DRequest ← idle;
    }
    wait;
}
```

Once the “activation token” is in a data path, the functionality of this data path is sequentially executed. Thereby, both data paths, CM1 and CM2 launch an interrupt if they have evaluated a new overhead set from the data stream.
This is done by passing the interrupt entry address and the “activation token” to CM3 alias SPACEMAN, which is controlled by the activation arbiter. The “activation token” remains as long in SPACEMAN as the execution of the interrupt routine is active. Thus, SPACEMAN is stimulated more than once per iteration of the main loop. To make this possible, the core loop of SPACEMAN is modified. This is shown in Pseudo-Code 7.4. SPACEMAN is activated by CM3Request. Then, its main loop is executed until rti assembler instruction (see appendix A) is detected, which indicates the end of the interrupt routine. If flag is true the loop is quit and the “activation token” is passed back to the data path.

Pseudo-Code 7.4: Carbon Model of SPACEMAN

```plaintext
process CARBON.Model {
    init(processorStatus);
    wait until DownLoadCompletion();
    L. irq: loop {
        wait on CM3Request;
        interruptLaunch(processorStatus, IRQAddress);
        flag = false;
        L. execute: loop {
            wait until InstructionFetch(instruction);
            case instruction.mnemonic is
                :
                when "rti" => rti(instruction, processorStatus);
                flag = true;
                end case;
                if (processorStatus.dStackOp ≠ idle) wait until dStackAccess(processorStatus);
                :
                exit L. execute when flag;
            }
        }
    CMSAcknowledge <= set;
    wait until CM3Request = idle;
    CMSAcknowledge <= idle;
}
```

In fig. 7.6 the activation sequence of the different blocks of the system test bench (see fig. 7.4) is illustrated. It equals the flow of the “activation token” during vector generation. Data path related blocks are executed in every iteration of the main loop (solid arrows in fig. 7.6), while controller related blocks are executed optionally when new overhead data is evaluated (dotted arrows in fig. 7.6).
Figure 7.6: System Test Bench Simulation Flow Chart
7.2.3 Reference Vectors

As mentioned in the last section every time an interrupt is launched in vector generation the "activation token" is passed to SPACEMAN which then executes the entire interrupt routine. When applying the newly introduced activation methodology, which executes the carbon models in the test system strictly successively, SPACEMAN can not be interrupted when it is already active. Thus, as opposed to model checking, where all implementation models of the system are concurrently processed in vector generation no nested interrupts are possible. Depending on the interrupt scheduling strategy, SPACEMAN can therefore be interrupted in the execution of the current routine (in the implemented model of the test vehicle the interrupt controller is laid out in such a way as to always process the last recently launched interrupt). Thus, nested interrupts occur in model checking and consequently the output sequence at the interfaces of SPACEMAN's carbon model are different in vector generation simulation from the output sequences of its implementation model in model checking simulation. This is illustrated in fig. 7.7 where a) represents a typical interface output sequence of vector generation and c) the counterpart for model checking. When the reference vectors are stored in one data base, a), failures would occur during comparisons of reference vectors to the corresponding outputs of the implementation model.

Therefore, another approach for vector sampling was implemented in the system test bench of vector generation. In fig. 7.4 the reference data bases for the instruction (VM1) as well as the data memory (VM2) are subdivided into several data bases. The number of data bases corresponds to the number of interrupt sources (i.e. the test vehicle generates two data bases for the data memory as well as for the instruction memory, see fig. 7.7b. During vector generation, depending on the current interrupt source (IRQi), reference data is stored in the corresponding data base. This approach facilitates vector comparison in model checking as it will be seen in the next section.

7.3 Model Checking

Model checking compares outputs of the system implementation model to the reference vectors. As opposed to the sequential execution of the carbon models in vector generation, in model checking the VC implementation models are concurrently executed.
7.3.1 Test Bench

In fig. 7.8 the block diagram of the test bench for model checking is shown. It is partitioned into the system test bench (light grey part) and the system VC (white part). The activation arbiter of fig. 7.4 has been replaced by the interrupt controller (VC6 in fig. 7.8). VC3, alias SPACEMAN, accesses three different memory blocks (VC4, register banks of VC1 and VC2) and controls the bus arbitration.

**Loaders:** Similar to vector generation, *loader1* and *loader2* get stimuli on request from their SDH data base and apply them to the corresponding data path (see section 5.5.1). For SPACEMAN the loader developed for its stand-alone verification can be reused.

**Snoopers:** To compare the outputs of the two data paths (VC1 and VC2) to their corresponding reference data, two primary snoopers (comparator; introduced in fig. 5.12b) are implemented in the test bench. For SPACEMAN (VC3) the complex snoopers introduced in section 6.3.3 are reused.
Simulation Control

Similar to vector generation the global simulation signal SimStatus controls the different phases of model checking (init, download, run, and terminate).

The blocks in the system test bench of model checking are triggered with three different clocks (Clk3, Clk1 and Clk5). The main part of the model runs with Clk5. The two data paths (VCI and VC2) are subdivided into the front end part which runs with Clk3 and Clk1, respectively and a back end part which runs with the system clock (Clk5). Thus, stimuli application to the data paths has to be done with Clk3 (loader1 in fig. 7.8) and Clk1 (loader2 in fig. 7.8).

Vector Comparison

As illustrated in fig. 7.7 the output sequences for the data and instruction memory differ in vector generation and model checking. The reference vectors are stored in separate data bases sorted according to the interrupt level. Therefore, the complex snoopers of SPACEMAN's stand-alone test bench can not be reused one-to-one. They have to be modified, as will be shown in the following.

In model checking the snooper has only to operate on that reference data base which is assigned to the current valid interrupt level. If the interrupt routine of VCI is executed, reference data base one is valid, and if the interrupt routine of VC2 is executed, reference data base two is valid. Thus, the snooper has to switch the reference data base, either if a new interrupt is launched or an interrupt routine is finished, indicated by an rti assembler instruction. This happens according to the following procedure:

- When a new interrupt is launched the interrupt level of the currently executed interrupt routine is saved.

- Then, the new interrupt level defines the actual valid reference data base and the snooper fetches reference vectors from this data base.

- If the interrupt routine is finished the snooper restores the old interrupt level and switches back to the old reference data base.

To support this strategy an interrupt stack has to be implemented in the snooper. The top-of-stack element always points on the actual valid reference
data base. If an interrupt is launched its level is pushed on the stack and if an interrupt routine is finished the top of stack entry is removed. In this approach the reference vectors are synchronized with the outputs of the implementation model and the snooper compares the correct data when it is activated by VC3 (see fig. 7.7).

7.4 Validation of the VC System Verification Flow

A methodology to verify a system composed of several VCs has been introduced. In vector generation, data paths are separated from controller parts. System execution is performed within a loop by successively triggering all data paths. During the execution of each data path, controllers can be evoked at certain points. In vector comparison, to guarantee the synchronization of reference vectors with the outputs of the implementation model of a shared controller, reference vectors are stored in several data bases depending on the actual interrupt level.

Most VHDL code of the test benches for the stand-alone verification of SPACEMAN could be reused when creating the system test bench. The stimuli data base, however, is not suitable for the system verification. For the stand-alone verification the stimuli source comprises generic assembler code segments which specifically stress certain functional parts of SPACEMAN. However, when SPACEMAN is embedded into a system, it has to interact with other models and therefore its functionality strongly depends on the embedding application; e.g., protocols have to be evaluated or filter functions have to be computed. Thus, only the VC user is able to generate dedicated stimuli vectors to verify a system composed of several VCs.

An insufficiency of the VHDL language set also restricts the reuse of test benches. Arrays of the type file are not supported in the VHDL standard. This makes the adaption of a test bench to different numbers of interrupt sources, resulting in different numbers of reference data bases, very laborious. The reference vector files can not be opened, accessed and closed within a loop of parameterizable range, indexed by the loop counter. Instead in a case statement every choice separately handles a reference file.
Figure 7.8: Block Diagram of System Test Bench in Model Checking
Conclusions

The introduction of virtual components (VC) for efficient system-on-a-chip (SOC) design complicates functional verification. The parameterization of VCs leads to a large variety of possible configurations, each of which requires specific verification data.

This thesis first gives an overview of the basic approaches for functional verification of digital designs. It is then shown that simulation is still the methodology to validate a design implementation against its specification even so exhaustive verification by simulation can never be performed for large designs and the quality of simulation is difficult to determine. As mentioned in chapter 2, all formal verification approaches are limited by the size of the models (state space explosion). Additionally, they either only validate two structurally almost equivalent models (equivalence checking) or do not fit into current design practices (model checking). Moreover, they are difficult for test engineers to use and generate no error tracks (theorem proving).
8.1 Main Achievements

A verification methodology for stand-alone VCs and systems composed of several VCs, consisting of a co-simulation verification framework based on two consecutive simulation runs, vector generation and model checking, has been developed. In vector generation, a generic behavioral model, called carbon model, generates reference vectors for any configuration of the VC implementation model. In model checking these reference vectors are compared to the outputs of the parameterized VC implementation model.

8.1.1 Stand-alone VC Verification

The major characteristics of this framework with regard to validating stand-alone VCs can be summarized as follows:

- The novel synchronization methodology “interface-specific activity scheduling” synchronizes the carbon model with the implementation model. Interdependent primary output signals of the VC are grouped into several interfaces which are independently triggered by activation strobes when an interface carries valid data.
  - The test bench is tolerant to varying interface latencies and therefore reusable for all different parameterizations of the VC implementation model.
  - The reference data set is significantly reduced since only data of the outputs of an activated interface are stored as reference vectors.
  - The simulation is sped up thanks to the decreased number of comparisons.

- The test bench is self-adaptable to different parameterizations of the VC implementation model.
  - Stimuli vectors are composed according to qualitative parameterization. This is a prerequisite for test bench reuse since the stimuli vectors have to stimulate different functionalities according to the VC customization.

- The complete VC verification environment, comprising test benches, a carbon model, and an implementation model, was written in VHDL.
Problems with foreign-language subset incompleteness as well as different foreign-language interface handling of HDL simulation platforms can be avoided.

The simulation run-time of the carbon model is at least one magnitude shorter than the simulation run-time of the implementation model, as measurements have shown in chapter 6. This was possible thanks to the description of the carbon model in a purely behavioral form and its sequential execution in a single process.

- The carbon model provided for verification by co-simulation additionally supports concurrent software and hardware development.
  - Software engineers have a platform to efficiently develop and debug assembler code.
  - Hardware designers have a behavioral description of the system specification for the functional verification of the implementation model.

To examine our simulation framework, a highly parameterizable application-specific instruction-set processor VC, SPACEMAN, was developed.

The major characteristics of this VC can be summarized as follows:

- The functionality of the carbon model is modular thanks to a separate procedure for every assembler instruction.
  - Changes in the specification of an instruction only cause local modifications in its corresponding procedure.
  - The VC provider can easily reuse the carbon model to extend processor VCs derived from SPACEMAN.

- In the implementation model the functional parts are separated from the interface handling parts.
  - The VC can easily be plugged into different environments, a capacity which is essential for reusability.
  - The VC user can adapt existing interfaces or add new interface types.
8.1.2 Verification of Multi-VC Systems

Based on the stand-alone VC validation framework, a test bench was elaborated to functionally verify a system composed of several VCs. This test bench was examined by the means of a system composed of two VCs for broadband telecommunication (STM-1, and STS-3), an interrupt scheduler VC, and SPACEMAN VC.

The major characteristics of the co-simulation framework for the system validation of this test vehicle can be summarized as follows:

- In system vector generation, the carbon model of the test vehicle is subdivided into two data paths, each of which is accessing a shared carbon controller.
  - The methodology to partition the carbon models into two data paths, each consisting of one data path carbon model, enables the test bench to successively process all carbon models in one single loop.
  - To shorten the simulation run times the shared carbon controller is evoked by a data path carbon model only at request (i.e. if the controller has to process data).
  - Robust and dead-lock free full-handshake protocols are used for data exchange between a carbon model and the system test bench. These protocols enable the user to easily plug together several carbon models and their corresponding stand-alone test benches to the system test bench.

- For system model checking the “interface-specific activity scheduling” synchronization strategy was improved.
  - Reference vectors for the shared controller are stored into several data bases, depending on the actual interrupt level.

8.1.3 Reusability

A novel verification methodology has been used to validate SPACEMAN VC. The different service blocks of this test bench (stimuli application, response acquisition, comparator, interrupt controller and simulation control) as well
8.2 Drawbacks

as the synchronization methodology and the activation mechanism with full-handshake protocols can also be reused with little modifications (e.g. complex snoopers have to be enhanced by an interrupt stack, as described in section 7.3.1) to develop a system test bench, if the VC is embedded into a multi-VC system. However, if the VC is embedded into a system, it has to interact with other models and therefore its functionality strongly depends on the embedding application (e.g. different protocol evaluation). Thus, for system verification the stimuli vectors provided by the stand-alone test bench have to be upgraded by application-specific vectors generated by the VC user.

The service blocks and the synchronization methodology can also be reused to design test benches for other VCs. However, in this case the stimuli vectors have to be created anew for every VC because the vectors have to stimulate different functional parts depending on VC customization.

The general structure of the carbon model can also be reused for the design of other VCs, i.e. subdividing the carbon model into the small core which is sequentially executed in a loop and the functionality which is implemented in several procedures.

8.2 Drawbacks

The VHDL simulation-based verification approach does not achieve the simulation performance of approaches using high-speed models written in non-hardware-related programming languages such as C, C++. However, with a model description using behavioral VDHL constructs and with its sequential execution, an acceptable simulation performance is obtained. Furthermore, the user has a portable test bench which is neither restricted in respect of foreign language interface handling nor in the supported language set.

As mentioned in chapter 7, the test bench introduced to verify multi-VC systems tolerates no data dependencies between any two data paths. This characteristic restricts the multi-VC verification flow to systems with several independent data paths (e.g. functional verification of the complete ITRACHIP, as shown in fig. 4.5, is not feasible because of the switching matrices implemented).

An insufficiency of the VHDL language set also restricts the reuse of test benches. Arrays of the type file are not supported in the VHDL standard. This
makes the adaption of a test bench to different numbers of interrupt sources, resulting in different numbers of reference data bases, very laborious. The reference vector files can not be opened, accessed and closed within a loop of parameterizable range, indexed by the loop counter. Instead, in a case statement, every choice separately handles a reference file.

8.3 Outlook and future research topics

- The basic principles elaborated to validate stand-alone SPACEMAN VC can be extended to establish a generic VC validation flow. Thereby, the same design directives as used to write the carbon model of SPACEMAN can be reused to write other carbon models (separation of the core and functional parts and use behavioral VHDL constructs). Furthermore, the test bench service blocks developed for SPACEMAN verification can also be reused to create test benches for new models.

- The methodology presented for the validation of the system test vehicle can be used to develop test benches for other multi-VC systems. A prerequisite is that all carbon models of the system use the same triggering mechanism with full-handshake protocols.

- As mentioned the system validation methodology is restricted to systems composed of several non-interacting data paths. Therefore, an important topic for future research is to find solutions which support the validation of systems with interacting data paths. One approach may be to introduce additional synchronization points in model execution. In fig. 8.1 the carbon model of the ITRASYS system is shown (see chapter 4). The system consists of five data paths. Two different kinds of synchronization points are introduced (see fig. 8.1). At the synchronization point ① two data paths exchange data and at the synchronization points ② two data paths are merged into one data path. The basic idea which might enable the correct model execution is to suspend the sequential execution of a data path at every synchronization point and to pass back the activation token to the test bench. In fig. 8.1 this happens for instance at ①. Then, the test bench evokes the next data path which is sequentially executed until another synchronization point is reached ③. This synchronization point indicates to the test bench that the execution of this data path has to be suspended and the execution of the original data path can be resumed ⑤. To support this approach, an
enhanced execution scheduling for the different carbon models would have to be implemented in the system test bench.

**Figure 8.1:** Execution Sequence of a System Carbon Model with Interacting Data Paths
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Stack Change</th>
<th>C-Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td><strong>Data ALU Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>incB #imm, d_x, d_y</td>
<td>if (d_x \neq #imm) then (d_x \leftarrow d_x + 1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cmp #imm, d_x, d_y</td>
<td>(d_x \leftarrow #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cmp d_x, d_y</td>
<td>(d_x \leftarrow d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>cmppp d_x, d_y</td>
<td>(d_x \leftarrow d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add #imm, d_x</td>
<td>(d_x \leftarrow d_x + #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>add test, d_x, d_y</td>
<td>(d_x \leftarrow d_x + #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>addp d_x, d_y</td>
<td>(d_x \leftarrow d_x + d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sub #imm, d_x</td>
<td>(d_x \leftarrow d_x - #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sub test, d_x, d_y</td>
<td>(d_x \leftarrow d_x - #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>subp d_x, d_y</td>
<td>(d_x \leftarrow d_x - d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>incm #imm, d_x</td>
<td>(d_x \leftarrow (d_x + 1) \mod #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>incm test, d_x, d_y</td>
<td>(d_x \leftarrow (d_x + 1) \mod #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>incm d_x, d_y</td>
<td>(d_x \leftarrow (d_x + 1) \mod d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>and #imm, d_x</td>
<td>(d_x \leftarrow d_x \text{ and } #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>and test, d_x, d_y</td>
<td>(d_x \text{ and } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>andp d_x, d_y</td>
<td>(d_x \leftarrow d_x \text{ and } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>or #imm, d_x</td>
<td>(d_x \leftarrow d_x \text{ or } #imm)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>or test, d_x, d_y</td>
<td>(d_x \leftarrow d_x \text{ or } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>orp d_x, d_y</td>
<td>(d_x \leftarrow d_x \text{ or } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>not d_x</td>
<td>(d_x \leftarrow \neg d_x)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>xor d_x, d_y</td>
<td>(d_x \leftarrow d_x \text{ xor } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>xorp d_x, d_y</td>
<td>(d_x \leftarrow d_x \text{ xor } d_y)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>shr d_x</td>
<td>(d_x \leftarrow d_x \gg 1), (d_{\text{MSB}} \leftarrow 0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>shl d_x</td>
<td>(d_x \leftarrow d_x \ll 1), (d_{\text{LSB}} \leftarrow 0)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rol d_x</td>
<td>(d_x \leftarrow d_x \gg 1), (d_{\text{MSB}} \leftarrow C)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ror d_x</td>
<td>(d_x \leftarrow d_x \ll 1), (d_{\text{LSB}} \leftarrow C)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

1. Disabled instructions for set 2 (75%)
2. Disabled instructions for set 3 (50%)
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Stack Change</th>
<th>C-Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address ALU Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addA #imm, a_i</code></td>
<td>$a_i \leftarrow a_i + #imm$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>addAhi d_i, a_i</code></td>
<td>$a_i \leftarrow a_i + (d_i &lt;&lt; \text{width}(d_i))$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>addAlo d_i, a_i</code></td>
<td>$a_i \leftarrow a_i + d_i$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Branch Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>bra offset</code></td>
<td>$PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>jmpAp</code></td>
<td>$PC \leftarrow a_0$</td>
<td>-</td>
<td>-1</td>
</tr>
<tr>
<td><code>jsr label</code></td>
<td>$PC \leftarrow \text{label}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>jsrA a_e</code></td>
<td>$PC \leftarrow a_e$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>bne offset</code></td>
<td>if $(Z)$ then $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>beq offset</code></td>
<td>if $(Z)$ then $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>bge offset</code></td>
<td>if $(C)$ then $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>boc offset</code></td>
<td>if $(C)$ then $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>decjmp d_i, offset</code></td>
<td>if $(d_i = 0)$ then $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>jmB #imm, d_i, offset</code></td>
<td>if $(d_i, #imm = 1)$ then $PC \leftarrow PC + \text{offset}$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><strong>Data Memory Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>pushD (a_i)+</code></td>
<td>$d_0 \leftarrow \text{mem}[a_i]$, $a_i \leftarrow a_i + 1$</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td><code>pushD (a_i)-12</code></td>
<td>$d_0 \leftarrow \text{mem}[a_i]$, $a_i \leftarrow a_i - 1$</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td><code>pushD offset(a_i)</code></td>
<td>$d_0 \leftarrow \text{mem}[a_i + \text{offset}]$</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td><code>storeD (a_i)+</code></td>
<td>$\text{mem}[a_i] \leftarrow d_0$, $a_i \leftarrow a_i + 1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>storeD (a_i)-</code></td>
<td>$\text{mem}[a_i] \leftarrow d_0$, $a_i \leftarrow a_i - 1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>storeD address</code></td>
<td>$\text{mem}[\text{address}] \leftarrow d_0$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>storeD offset(a_i)</code></td>
<td>$\text{mem}[a_i + \text{offset}] \leftarrow d_0$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>writeD d_i, (a_i)+</code></td>
<td>$\text{mem}[a_i] \leftarrow d_i$, $a_i \leftarrow a_i + 1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>writeD d_i, (a_i)-12</code></td>
<td>$\text{mem}[a_i] \leftarrow d_i$, $a_i \leftarrow a_i - 1$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>writeD d_i, offset(a_i)</code></td>
<td>$\text{mem}[a_i + \text{offset}] \leftarrow d_i$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Stack Access Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>popA</code></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>popAD</code></td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><code>popD</code></td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td><code>pushA address</code></td>
<td>$a_0 \leftarrow \text{address}$</td>
<td>-</td>
<td>+1</td>
</tr>
<tr>
<td><code>pushAlo d_i</code></td>
<td>$a_0 \leftarrow d_i$</td>
<td>-</td>
<td>+1</td>
</tr>
<tr>
<td><code>pushD #imm</code></td>
<td>$d_0 \leftarrow #imm$</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td><code>pushD d_i</code></td>
<td>$d_0 \leftarrow d_i$</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td><code>pushD address</code></td>
<td>$d_0 \leftarrow \text{mem}[\text{address}]$</td>
<td>+1</td>
<td>-</td>
</tr>
<tr>
<td><code>replAhi d_i, a_i</code></td>
<td>$a_i \leftarrow (d_i &lt;&lt; \text{width}(d_i))$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>replD #imm</code></td>
<td>$d_i \leftarrow #imm$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>swapD d_i, d_i</code></td>
<td>$d_i \leftarrow d_i$, $d_i \leftarrow d_i$</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><code>trap level</code></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><strong>Miscellaneous Instructions</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>nop</code></td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>reset</code></td>
<td>$PC \leftarrow 0$, condition code $\leftarrow c_0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><code>rti</code></td>
<td>$PC \leftarrow r_0$</td>
<td>-</td>
<td>-1</td>
</tr>
<tr>
<td><code>rts</code></td>
<td>$PC \leftarrow r_0$</td>
<td>-</td>
<td>-1</td>
</tr>
<tr>
<td><code>sleep</code></td>
<td>$PC \leftarrow PC \text{ until interrupt}$</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
B

Design Representation

Formal verification approaches require a data structure which efficiently reproduces the state space of a design and which forms a basis on which the algorithms behind these techniques can work. In fig. B.1 for a small sequential example (enabled clock divider) different possibilities for such representation are shown.

A simple approach is to look for all possible states which the circuit can ever reach and to store all these states and their interdependencies, respectively (transition diagram). A drawback of this explicit approach is that the state space often grows too large, a phenomenon known as state space explosion. Each additional bit which is added to the state description potentially doubles the state space.

An alternative to using explicit state representation is to store the set of reachable states using a binary decision diagram (BDD) [Bry86]. A BDD is a directed acyclic graph whose leaves are the constant nodes 0 and 1. It is represented as a boolean function $f : B^n \rightarrow B$, where $B = \{0,1\}$ which is a compact and canonical data structure. BDDs can be used to implicitly store the set of reachable states. If the set of state variables is encountered in a fix sequence (e.g. in fig. B.1: $I \leq S \leq S'$ in any path of the BDD, starting at the root and ending at a leaf), the BDD is called ordered BDD (OBDD).

When an OBDD is compressed, the commonly used data structure for formal verification techniques, which is called reduced OBDD, (ROBDD) is obtained. Although ROBDDs potentially have the advantage to represent a
large number of states using very little memory, they can also suffer from state space explosion. It is often difficult to predict the growth of a ROBDD circuit representation, because OBDD can be compressed very well for certain designs \([L+99]\) while for other designs the size can grow exponentially due to bad ordering. One can specify the worst case upper boundary \(O(2^n), n = \text{number of state bits}\), of the number of ROBDD nodes, but typically \(O(\frac{2^n}{2})\) nodes are used \([FJS87]\). Register files are examples of designs which fit very poorly into ROBDD representations. They contain an enormous number of storage elements, and the OBDD generator will naively attempt to build ROBDD of functions depending on all storage elements, a procedure which is usually not feasible \([PRBB96]\).

\[\begin{array}{|c|c|c|}
\hline
I & S[k] & S'[k+1] \\
\hline
0 & 0 & 0 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\hline
\end{array}\]

*I*: Input  
*S*: Current State  
*S*: Successor State

\(f : (S' = I \land S)\)

\(\rightarrow\): '1' Transition  
\(\rightarrow\rightarrow\): '0' Transition

**Figure B.1**: Different Kinds of Design Representations
Bibliography


Int. ASIC/SOC, pages 399–403. IEEE Communications Society, IEEE, September 1999.


Curriculum Vitae

I was born in Winterthur, Switzerland, on June 21, 1968. After finishing high school at the Kantonsschule Romanshorn (Matura Typus C (scientific branch)) in 1988, I studied Electrical Engineering at the Swiss Federal Institute of Technology ETH Zürich. I took a degree in Electrical Engineering (Dipl. El.-Ing. ETH corresponding to an M. Sc.) in 1995. In May 1995 I joined the Integrated Systems Laboratory (IIS) of ETH where I worked as a research assistant in the field of design and verification of digital VLSI circuits. In cooperation with an industrial partner I was involved in system specification and architecture evaluation of a telecommunication ASIC for fiber optical systems (SDH). My research interests include VLSI design and synthesis, system descriptions in VHDL, verification of VHDL and application specific microprocessors for telecommunication systems.
List of Publications


