

ABSTRACT

The aim of this project is to research, design and derive hybrid methodologies which combine simulation based verification and formal verification methods for achieving earlier functional coverage closure with less engineering effort. The methodologies will be applied to the microcontroller TriCore 1.6.1 produced by Infineon.

In this project, the starting point is a flow already in place for code coverage. This flow will be fundamentally modified and extended to deal with functional coverage. Achieving 100% functional coverage is usually a very hard requirement when using simulation-based method for verification of the design. In order to get high functional coverage, verification engineers should generate and run a large number of tests and then get the coverage report. Based on the functional coverage report, a formal property checker is then applied to the coverage holes to establish whether this coverage is reachable. This requires generating a suitably named property for each functional coverage hole, running the property on a formal property checker and filtering any unreachable coverage. Finally, the flow should be automated by scripts. Manual analysis of functional coverage holes can be extremely time consuming. The hybrid method presented here aims to save engineering effort and time.

ACKNOWLEDGEMENT

I would like to thank Dr Kerstin Eder for her continuous encouragement, involvement, guidance and planning which kept me motivated to successfully complete the project.

I would also like to thank the staff of Infineon namely Tim Blackmore , James Buckingham, Phil Barker, Fabio Bruno who took good care of me and patiently guiding me throughout the course of the project.

I would also like to thank my parents and friends for the strength and support they gave to me for completing my project without any hassles.

Contents

CHAPTER 1 INTRODUCTION.....	1 -
1.1 INTRODUCTION.....	1 -
1.2 AIMS AND OBJECTIVES.....	1 -
1.3 ORGANIZATION OF THIS DISSERTATION.....	2 -
CHAPTER 2 CURRENT VERIFICATION TECHNIQUES.....	4 -
2.1 HDL SOFTWARE SIMULATOR	4 -
2.2 ACCELERATED SIMULATION.....	4 -
2.3 ASIC HARDWARE ACCELERATOR	5 -
2.4 HARDWARE EMULATOR	6 -
2.5 FPGA PROTOTYPE	6 -
2.6 SUMMARY.....	7 -
CHAPTER 3 COVERAGE AND COVERAGE METRICS	8 -
3.1 COVERAGE	8 -
3.2 CONTROLLABILITY VERSUS OBSERVABILITY	8 -
3.3 COVERAGE METRICS	8 -
3.3.1 <i>Code Coverage</i>	8 -
3.3.2 <i>Structural Coverage</i>	9 -
3.3.3 <i>Functional Coverage</i>	9 -
3.3.3.1 Introduction.....	9 -
3.3.3.2 Cross-product coverage.....	10 -
3.3.3.3 Example (alignment trap)	10 -
3.3.4 <i>Hybrid Coverage Method</i>	11 -
3.4 COVERAGE CLOSURE	11 -
3.5 SUMMARY.....	12 -
CHAPTER 4 METHODOLOGIES.....	13 -
4.1 INTRODUCTION.....	13 -
4.2 SIMULATION-BASED VERIFICATION	13 -
4.3 FORMAL VERIFICATION	14 -
4.3.1 <i>Introduction</i>	14 -
4.3.2 <i>Property</i>	15 -
4.3.3 <i>Methodology</i>	16 -
4.3.3.1 Logical Implication and Vacuous Truth.....	16 -
4.3.3.2 Inductive proof	16 -
4.3.3.3 Temporal Logic and Temporal Induction	17 -
4.3.3.4 Formal sequential verification	19 -
4.4 SIMULATION-BASED VERIFICATION VERSUS FORMAL VERIFICATION	20 -
4.4.1 <i>The advantages and disadvantages for each method</i>	20 -
4.4.2 <i>Methodology comparison Example</i>	21 -
4.5 HYBRID METHOD.....	22 -
4.5.1 <i>Introduction</i>	22 -
4.5.2 <i>Flow</i>	23 -
4.5.3 <i>Advantages and Disadvantages of hybrid method</i>	25 -
4.6 ASSERTION-BASED VERIFICATION	25 -
4.7 SUMMARY.....	26 -
CHAPTER 5 COMPLETING THE VERIFICATION CYCLE	27 -
5.1 VERIFICATION CYCLE	27 -
5.2 REGRESSION	27 -
5.3 TAPE-OUT READINESS	28 -
5.4 ESCAPE ANALYSIS.....	29 -
CHAPTER 6 EXPERIMENTS AND ISSUES.....	30 -

EXPERIMENT 1	- 30 -
EXPERIMENT 2	- 33 -
EXPERIMENT 3	- 36 -
EXPERIMENT 4	- 39 -
EXPERIMENT 5	- 41 -
EXPERIMENT 6	- 45 -
EXPERIMENT 7	- 46 -
ISSUE 1	- 47 -
ISSUE 2	- 48 -
ISSUE 3	- 49 -
CHAPTER 7 SUMMARY AND CONCLUSION.....	- 50 -
CHAPTER 8 POSSIBLE IMPROVEMENTS.....	- 51 -
BIBLOGRAPHY.....	- 52 -
Appendix A: System Verilog Assertion	- 54 -
Appendix B: Tricore Microcontroller	- 55 -
Appendix C: Onespın 360MV	- 57 -
Appendix D: Source code and declaration from Infineon Technology UK Limited.....	- 58 -

List of figures

Figure 1 Structure of HDL software simulator.....	- 4 -
Figure 2 Structure of accelerator simulator.....	- 5 -
Figure 3 Structure of ASIC hardware accelerator.....	- 6 -
Figure 4 Progress of coverage closure.....	- 11 -
Figure 5 The diagram of concept for 80/20 split.....	- 12 -
Figure 6 Simulation-based verification environment flow.....	- 13 -
Figure 7 The modern testbench architecture.....	- 14 -
Figure 8 Workflow of formal method.....	- 15 -
Figure 9 The diagram of temporal induction.....	- 18 -
Figure 10 Fixed-point reachable states.....	- 20 -
Figure 11 The flow of hybrid method.....	- 23 -
Figure 12 Basic assertion description.....	- 26 -
Figure 13 The diagram of verification cycle.....	- 27 -
Figure 14 When is verification done?.....	- 28 -
Figure 15 Bug rates and coverage closure of functional coverage.....	- 29 -
Figure 16 The flow of translating and simulating.....	- 31 -
Figure 17 The example result of single property checking.....	- 37 -
Figure 18 The result of multi property checking.....	- 38 -
Figure 19 Flow of writing script for coverage points.....	- 41 -
Figure 20 The example of script code in Perl.....	- 42 -
Figure 21 Perl script running results.....	- 43 -
Figure 22 Simulation results for alignment trap.....	- 44 -
Figure 23 The functional coverage report in txt version.....	- 45 -
Figure 24 The similarities and differences for the two temporal language.....	- 46 -
Figure 25 The structure of the personal monitor.....	- 48 -

List of tables

Table 1 Comparison of each techniques.....	- 7 -
Table 2 Comparison between code and functional coverage.....	- 11 -
Table 3 Strengths and weaknesses for each method.....	- 20 -
Table 4 Assertions usage	- 26 -

Chapter 1 Introduction

1.1 Introduction

Design verification is widely seen as the bottleneck of the design development cycle. This is due to a combination of several correlated factors. One is the exponential increase in design complexity. As modern integrated circuits technology becomes more complex, design verification plays a critical role in the development of integrated circuits and determines the success of products. Tighter time-to-market requirements mean that verification activities must be completed in a limited time before tape-out whilst finding all critical bugs. Also higher quality expectations are a factor and means that as few bugs as possible can be left in the design. The cost of the late discovery of bugs is enormous. It justifies the fact that for a typical microprocessor design project, more than half of the overall resources spent are devoted to its verification. In fact, roughly 70% of design effort is spent on verification and debug [[HYPERLINK \l "Ker10" 1](#)].

In this project, the starting point is a flow already in place for code coverage. This flow will be fundamentally modified and extended to deal with functional coverage. Functional coverage is essential when assessing the effectiveness of random test generation and achieving 100% functional coverage is usually a hard requirement when using simulation-based method for verification of a design. In order to get high functional coverage, we should generate and run a large number of test cases and then get the coverage report. Based on this report an analysis is performed to identify missing coverage. A formal property checker is then applied to the missing functional coverage to establish whether or not this coverage is reachable. This requires generating a suitably named property for each functional coverage hole, running the property on a formal property checker and filtering any unreachable coverage. For reachable coverage the formal property checker will produce a trace that should be possible to translate into simulation stimulus to target the coverage hole. Manual analysis of functional coverage holes can be extremely time consuming. The hybrid method presented here aims to save engineering effort and time.

1.2 Aims and Objectives

The aim of the project is to research, design and derive hybrid methodologies which combine simulation based verification and formal verification methods for achieving earlier functional coverage closure with less engineering effort. The methodologies will be applied to the TriCore microcontroller from Infineon.

The overall objective of the project is to provide an easier and less time consuming approach for determining whether functional coverage holes are really reachable.

The list of specific objectives for the project is defined below:

1. Review of functional coverage metrics.
2. Review of Assertion based verification and formal property checking techniques.
3. Review of specification and verification requirements specific to this project.
4. Review of simulation based verification knowledge.
5. Review of coverage closure knowledge.
6. Study and skillful use of a scripting language such as Perl.
7. Understanding of the formal property language Property Specification Language (PSL).
8. Study and skillful use of the temporal language such as System Verilog Assertions (SVA).
9. Understanding of temporal induction for formal property checking.
10. Understanding of previous work on closing code coverage with a hybrid method.
11. Write some functional coverage for TriCore in a temporal language.
12. Simulate test cases and generate coverage reports.
13. Design and implement a procedure to automatically generate suitable properties for functional coverage holes.
14. Extend this procedure to identify and automatically exclude unreachable coverage holes from consideration.
15. Analysis of the results and propose improvements.

1.3 Organization of this dissertation

Chapter 2 Discusses about the current verification techniques, especially the hardware technology and simulation engine.

Chapter 3 Discusses about the coverage, various coverage metrics especially functional coverage and coverage closure.

Chapter 4 Lists the various methods for verification, which includes the simulation-based method, formal method and hybrid method used for this project.

Chapter 5 Introduces the latter stages in verification cycle, which include regression, tape-out readiness and escape analysis.

Chapter 6 Lists the main experiments in this project, also explains the main issues we met during the project.

Chapter 7 Concludes the study and research for this project.

Chapter 8 Lists the possible improvement of this project and some further work after this project.

Appendix A Explains about the details of SystemVerilog Assertion. The simple syntax and functions used in this project.

Appendix B Introduces about the architecture, functionality and features of Tricore 1 from Infineon.

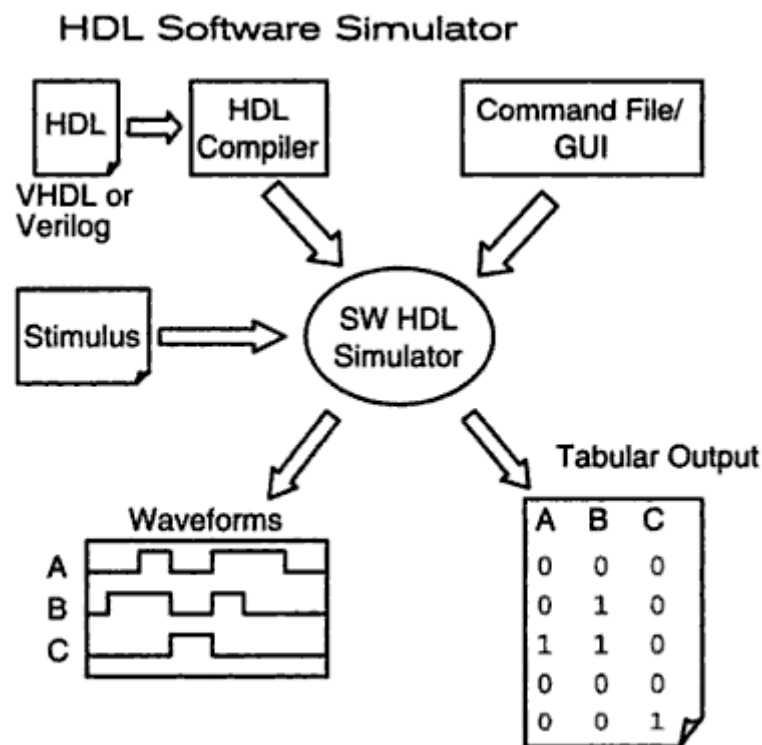
Appendix C Introduces the formal checker Onespin 360 MV.

Appendix D Source code and declaration from Infineon.

Chapter 2 Current Verification Techniques

2.1 HDL Software Simulator

An HDL software simulator is a software application which used on engineering workspace or PC. The HDL software simulator uses the hardware description language (for example: VHDL, Verilog HDL) as its input which describes the functional operation of the design. Afterwards, the simulator applies the stimulus and commands when it is executing. At last, the simulator generates output data for analysis. It is illustrated in Fig.1. Generally, an HDL description describes design behaviors when the processes change along with the input signal changes. The main advantages of this method are low-cost and it can handle very large designs. However, the biggest drawback is the running time is huge.



. Figure 1[2] Structure of HDL software simulator.

2.2 Accelerated Simulation

In order to reduce the running time of HDL software simulator, many researchers dedicated themselves to this area. Accelerated simulation attempts to solve this problem. There are two different types of approaches have been used to increase the simulation time [3]. One is software approaches and the other one is hardware approaches.

The software approaches ignores some kind of the simulation accuracy to reduce the running time. However, the reduction of running time is not enough and these approaches are only used if the accuracy factor is not important.

Comparing with the software approaches, the hardware approaches are more effective. Some approaches use custom-built hardware which integrates a set of interconnected processors. Some other approaches use Field-programmable gate array (FPGA) devices to interconnect the FPGA processors. In this kind of system, each of the processors executes a piece of the total simulation in parallel and FPGA devices make the appropriate connections between the processors. The diagram is shown in Fig.2. The main problem of this kind of systems is the communication between each device. The delay occurs here and it influences the running time. With the number of processors increases, the increment of speed increases nonlinear and the increment curve is similar as the logarithmic curve. Although these accelerators run much faster than HDL software simulators, they would be more expensive. Some of these accelerators cost 10 to 20 times than an HDL software simulator [3].

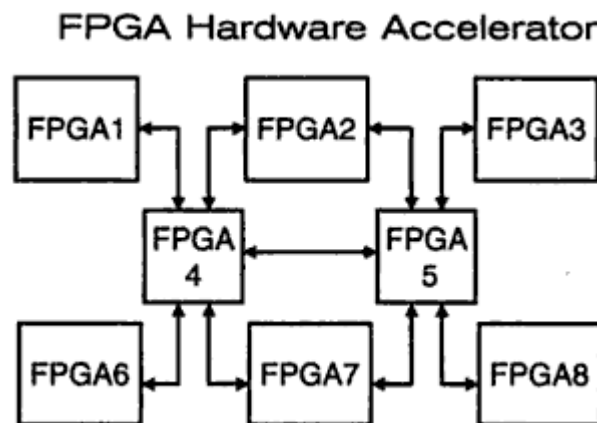


Figure 2 [3] Structure of accelerator simulator.

2.3 ASIC Hardware Accelerator

To further increase the simulation speed, the application-specific integrated circuit (ASIC) processors are used for simulation. ASIC processors optimise the simulation speed to a very high speed. The processors are connected in a grid or matrix and communicate each other by using interface busses [3]. The structure is shown in Fig.3. Although the speed of ASIC-based systems is much higher than FPGA-based systems, this speed also comes the tremendous cost.

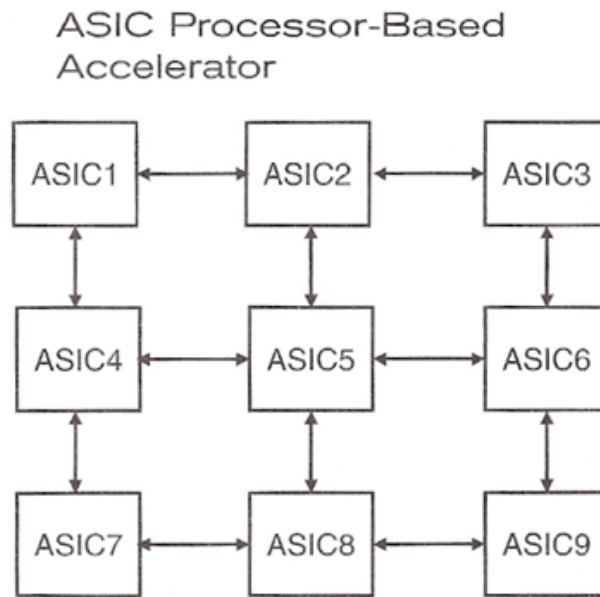


Figure 3 [3] Structure of ASIC hardware accelerator

2.4 Hardware Emulator

Along with the FPGA devices developed large enough, hardware emulators were developed. An emulator is built with a large number of flexible FPGA devices on a circuit board and the many circuit boards implement an entire system. The main advantage of an emulator is the faster speed. The disadvantage is the complex hardware structure.

2.5 FPGA Prototype

An FPGA prototype also uses a set of FPGA devices to implement designs in parallel. The designs are separated into many blocks and these blocks are mapped by using FPGA software through FPGA devices. For these approaches of systems, the main improvement is that they use a particular board to interconnect the FPGA devices instead of using FPGA interconnect devices. The board provides a fixed connection for devices and it can also be used as a workspace. The use of this kind of board makes the higher speed between FPGA devices. The price of this approach of system depends on the size and the number of FPGA devices used. Generally, it uses fewer devices than an emulator. Therefore, the price would be much cheaper than emulators. The main weakness of this approach of system is the low internal visibility. There are limited pins available for internal signals. This factor would make debugging difficult.

2.6 Summary

This chapter described the main techniques of verification. For the merits and drawbacks of each technique, the table below will show the comparisons for each factor.

	Real Hardware	HDL Software Simulator	Hardware Accelerator	ASIC Hardware Accelerator	Hardware Emulator	FPGA Prototype
Speed	Real time	Low	Medium	Medium to high	High	Approaching of meeting the speed of real device
Visibility	Poor	Excellent	Excellent	Excellent	Excellent	Poor
Compile time	Poor	Fast	Fast	Fast	Slow	Slow
Debugging tools	Poor	Excellent	Excellent	Excellent	Good	Poor
Checkpoint	No	Yes	Yes	Yes	Yes	No
Cost	Very high	Low	Medium to high	High	High	Low to medium
Testbench required	No	Yes	Yes	Probably	Probably	Maybe
Simulation coverage	High	Low	Low to medium	Low to medium	Low to medium	Medium

Table 1 Comparison of each techniques

Chapter 3 Coverage and Coverage Metrics

3.1 Coverage

In design verification, in order to run all possible scenarios and to know whether all areas of DUV are verified. Coverage measurement and analysis are used in DUV. Coverage is defined as events (or scenarios) or families of events that span the functionality and code of the design [1]. Coverage provides a stopping criteria for unit testing [1]. It gives engineers observably result and confidence of functionality of the design.

3.2 Controllability versus Observability

The two properties are the fundamental to the discussion of coverage.

Controllability indicates the ease at which the verification engineer creates the specific scenarios that are of interest [1].

Observability indicates the ease at which the verification engineer can identify when the design acts appropriately versus when it demonstrates incorrect behaviour [1].

3.3 Coverage Metrics

3.3.1 Code Coverage

Code coverage analysis is the process of finding areas of a program not exercised by a set of test cases and creating additional test cases to increase coverage. It describes the degree of the source code which has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing [3]. It is a measure of controllability [4]. The intent of code coverage is to directly indicate the rate of run code and whole source code, but also to document what the code is expected to do [4]. Indirectly, code coverage implies the quality of the code. However, code coverage cannot imply the functional correctness of the product. Even 100% of the code coverage cannot mean very much [1]. It needs to combine some other form coverage.

There are a lot of model types for code coverage.

Control flow checks program from statement/block, branch/path, expression/condition and so on.

Data flow checks program based on flow of data/variables during running. It checks the variables at first and checks whether the variables are used in program.

Mutation coverage is designed to check simple mistakes in the program. For example, the wrong operators and wrong variables would be detected in this model.

For hardware, toggle coverage and all-value coverage would be useful.

3.3.2 Structural Coverage

State-machines are the essence of RTL designs. The finite state-machines (FSM) are mostly used for structural coverage models. This kind of models checks whether all the states have been reached by the design.

3.3.3 Functional Coverage

3.3.3.1 Introduction

Functional coverage is a measure of which design features have been exercised by the tests. It checks for the functionality of the DUV [1]. This category of coverage refers directly to the computation of the whole system rather than its source code or structure. It is based on the functionalities, features and properties of the design. During simulation, each scenario and functionality fragment must be exercised [6]. Functional coverage is essential when assessing the effectiveness of random test generation. Achieving 100% functional coverage is usually a hard requirement. However, high functional coverage implies the high quality of the product. In order to get high functional coverage, we should generate and run a large number of test cases and then get the coverage report. Based on this report an analysis is performed to identify what we have not verified (holes) and what functionality we have verified (from the specification).

Generally, functional coverage models would cover the inputs and the outputs, internal states, scenarios, parallel properties and bug models. In contrast with code coverage, although functional coverage can indicate the functional correctness of the product, this method is only as good as the coverage metrics. The main drawback is this category of coverage requires manual effort and needs a lot of engineers' experience.

3.3.3.2 Cross-product coverage

Functional coverage models can be built in four types. They are a collection of discrete events, trees and hybrid model which combines trees and cross-product [7]. In this project, we only interested in the cross-product model.

A cross-product functional coverage model can be constructed in the following manner [8].

1. A semantic description of the model as a story.
2. A list of the attributes mentioned in the story.
3. A set of all the possible values for each attribute.
4. A list of restrictions on the legal combinations in the cross-product of attribute values.

The cross-product functional coverage example will be described in section 3.3.3.3.

3.3.3.3 Example (alignment trap)

Alignment rules:

1. Alignment requirements differ for addresses and data.
2. Address variables loaded into or stored from address registers must always be word-aligned.
3. Data can be aligned on any half-word boundary, regardless of size, except where noted below. This facilitates the use of packed arithmetic operations in DSP applications, by allowing two or four packed 16-bit data elements to be loaded or stored together on any half-word boundary.

Alignment trap:

1. An alignment trap is raised when the address for a data memory operation does not conform to the required alignment rules.
2. An alignment trap is also raised when the size, length or index of circular buffer is in correct.

In order to check the functionality of the system, we want to see an alignment trap on every load and store instruction. It crosses three features in the system.

1. Alignment trap (1-bit: the bit goes 1 implies that the alignment trap occurs);
2. Valid instruction (We only interested in the valid instructions);
3. Instruction opcode (It identifies the different instructions)

3.3.4 Hybrid Coverage Method

Interpretation	Code coverage	Functional coverage
Start of project	Low	Low
Multi-cycle scenarios, corner cases, cross-correlations to be covered	High	Low
Functionalities, check for "dead" code	Low	High
High confidence of quality	High	High

Table 2 [2] Comparison between code and functional coverage

From the table above, it is obviously to see that the code coverage and functional coverage complement each other. The good way is to use both code and functional coverage in DUV.

3.4 Coverage Closure

Coverage closure is the process of finding the areas which not been verified (coverage holes) by a set of tests and then creating another set of tests to target the coverage holes. Coverage closure implies whether the design is verified thoroughly and whether all coverage goals have been reached. Coverage closure is significant for the whole verification progress. Furthermore, tighter time-to-market require to receive the coverage closure early. The right part of Fig.3 shows the progress of coverage closure.

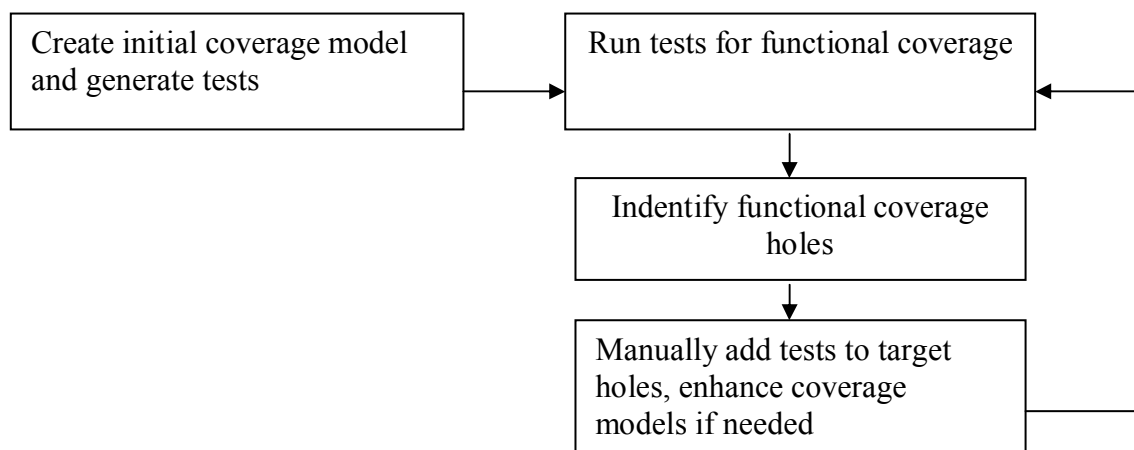


Figure 4 Progress of coverage closure.

In practice, the 80/20 split theory is as a flag for the project. The meaning is achieving 80% of the coverage just takes 20% of the time and effort, while achieving 20% of the coverage spends 80% of the time and effort. The former must be a good news for verification. On the contrary, it is unlucky to receive the result of latter one.

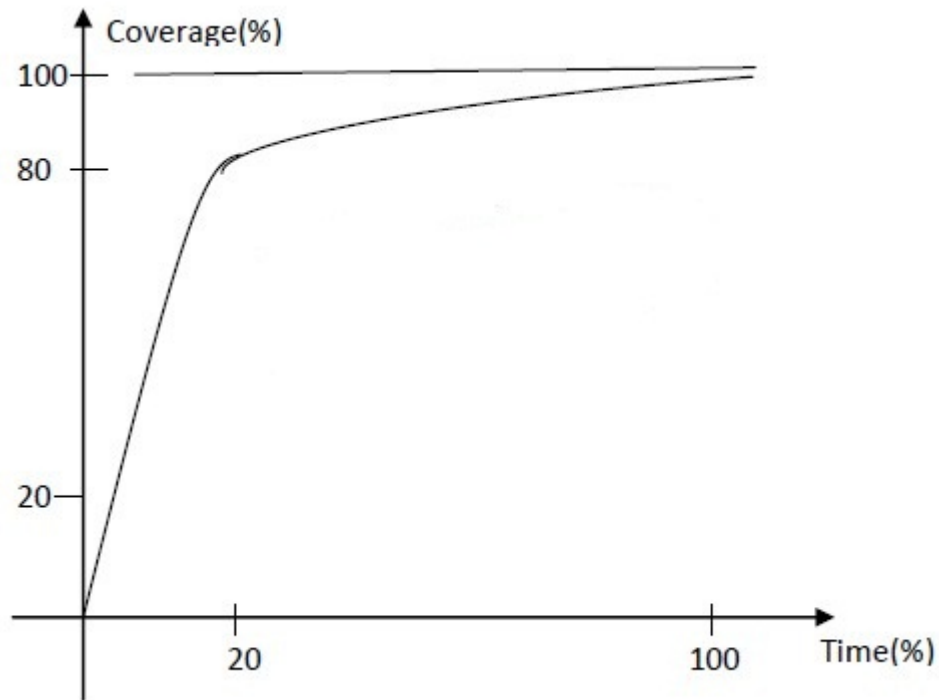


Figure 5 [1]. The diagram of concept for 80/20 split

3.5 Summary

This chapter introduced the basic concept of coverage and main metrics of coverage, especially the functional coverage. This chapter also discussed the function example (alignment trap) we used in this project.

Chapter 4 Methodologies

4.1 Introduction

In general, functional verification is done using two different methodologies. One is formal verification and the other one is simulation based verification. Simulation based verification checks the behaviors of the design with selected input stimuli called test programs. Increasing design complexity, and the other considerations discussed above, mean that it is necessary to produce high quality test programs in large quantities. This is usually done by an automatic random test generator. However, simulation can only demonstrate the presence of bugs but not the absence. Moreover, with the complexity of integrated circuits growing with time, the simulation tool engine consumes even more time. Hence the industry had to find other methods for verification. The formal verification approach attempts to prove the correctness of properties on the design without having to write testbenches. This approach is useful for hardware validation [7]. Compared with the simulation based verification, simulation cannot handle all possible behaviors. However, the formal verification conducts exhaustive exploration of all possible behaviors [7].

4.2 Simulation-based Verification

Simulation is one of the most common methods of verification used since the 1960s [2]. In an early time, simulations were done at transistors, but now most of simulations are simulated at register transfer level (RTL). The HDL languages are supported to write on it. The typical simulation-based method environment is shown in Fig.6. Simulation-based method is a traditional, basic and flexible technique and applicable at any design level.

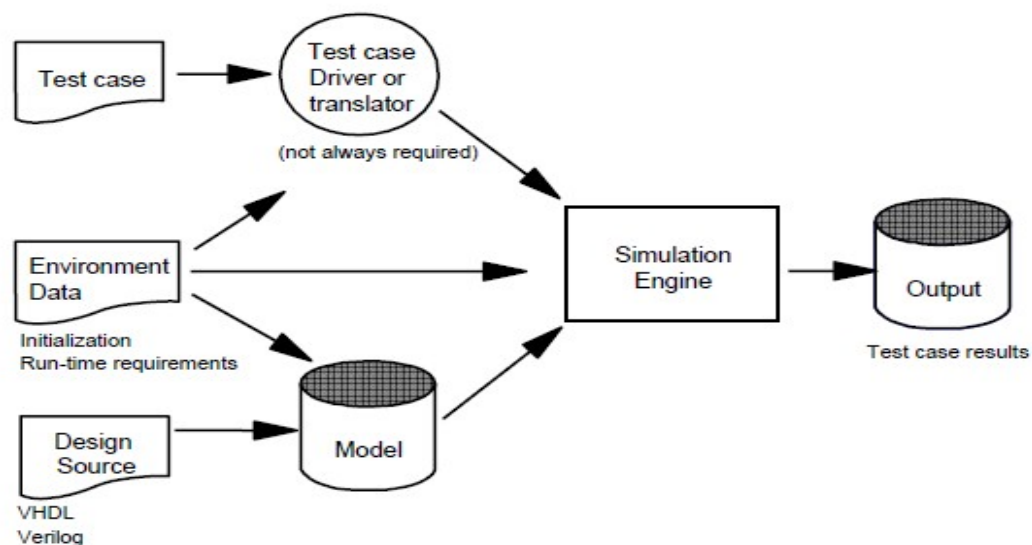


Figure 6 [1]. Simulation-based verification environment flow

At first, a device model which actually is design description should be built by engineers. Following, the simulators would read the device model and convert it to its data input. There is a main advantage of this system. Device models do not need to contain stimulus. In the other word, designer can use different design stimulus to drive the same device model to test different design functionalities. Stimulus is an input which can take some different types of information. For example, data read from a file or models which represent the target system environment. Next, the simulator will drive the stimulus to the design and give a feedback. In the end, the simulator will generate the output data. Generally, most of the forms to the data are tabular output and waveform.

When the number of signals is small, it is easy to identify the results by visual inspection. However, along with the design becomes more and more complex, it is no longer possible to verify in this way. Therefore, the testbench is written to drive and monitor the design and results. The modern testbench architecture is shown in Fig. 7. The heart of this method is the simulation engine. As the complexity of design grew with time, the simulator engine is no longer powerful to satisfy the exercises. It consumed great deal of time to run the testbench. Engineers had to find some better methods which can get the results faster.

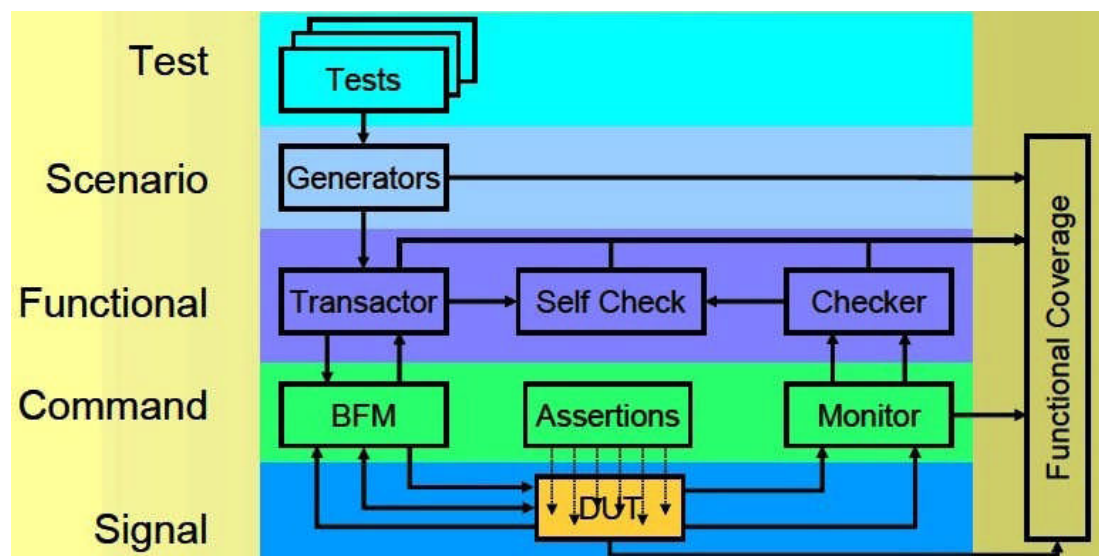


Figure 7 [8]. The modern testbench architecture

4.3 Formal Verification

4.3.1 Introduction

Formal verification is a systematic process of ensuring, through exhaustive algorithmic techniques, that a design implementation satisfies the requirements of its specification [2]. All possible executions of the process are mathematical analysis without building testbench or import test cases. Formal verification aims to prove

the correctness of a design with respect to a mathematical formal specification [9]. The general workflow of formal method is shown in Fig.8. There are two main approaches of the formal verification. One is formal Boolean verification, the other one is formal sequential verification.

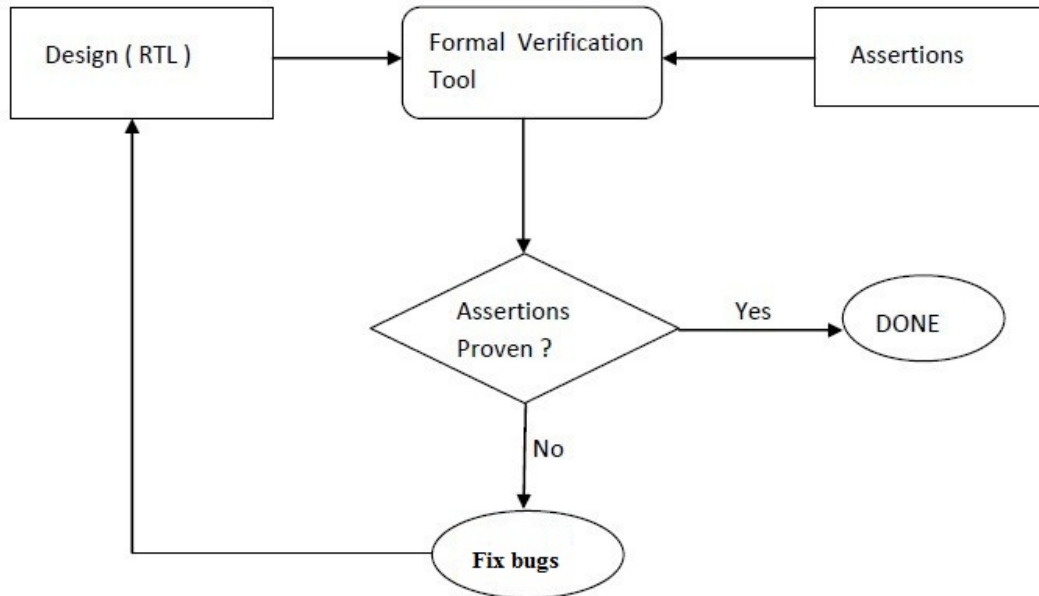


Figure 8 Workflow of formal method

4.3.2 Property

The definition of property is: a collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviour [4].

The classification of property is shown below:

Safety property

Undesirable things never happen

Desirable things always happen

Examples:

A bus arbiter never grants the requests to two masters

Elevator does not reach a floor unless it is requested

Message received is the message sent

Liveness (progress) property

Desirable state repeatedly reached

Desirable state eventually reached

Examples:

Every bus request is eventually granted

A car at a traffic light is eventually allowed to pass

4.3.3 Methodology

4.3.3.1 Logical Implication and Vacuous Truth

Logical Implication

Whenever p is true, q is true [12].

$$p \Rightarrow q$$

p implies q

The truth table is shown below:

p	q	$p \Rightarrow q (\neg p \vee q)$
T	T	T
T	F	F
F	T	T
F	F	T

Vacuous Truth

From the truth table shown above.

The statement S is “vacuous true” if the statement p implies q and p is false [13].

4.3.3.2 Inductive proof

The basic theory of inductive proof is mathematical induction. It describes that proving the first statement in an infinite sequence is true, and then proving if any statement in the infinite sequence is true, then the next statement is true [10].

Inductive proof example:

The basic proof of induction involves statements which depend on the natural numbers, $n = 1, 2, 3 \dots$. It often uses summation notation.

The sum of the natural number up to value n is shown below:

$$1 + 2 + 3 + \dots + (n-1) + n = \sum_{i=1}^n i$$

Suppose that

$$\sum_{i=0}^n i(n) = \frac{n(n+1)}{2}$$

Inductive base:

For the base case, when $n = 0$, $\sum_{i=0}^0 i(0) = \frac{0(0+1)}{2} = 0$ It is true.

Inductive step:

We assume $n = k$ now, then $\sum_{i=0}^k i(k) = \frac{k(k+1)}{2}$ Equation (1)

For $n = k+1$, the sum of $k+1$ numbers should be equal to the sum of k numbers plus the number $k+1$. It is shown below:

$$\sum_{i=0}^{k+1} i(k+1) = \sum_{i=0}^k i(k) + (k+1)$$
 Equation (2)

Combining the equation (1) and (2):

$$\sum_{i=0}^{k+1} i(k+1) = \frac{k(k+1)}{2} + (k+1)$$
 Equation (3)

$$\text{According to equation (3), hence, } \sum_{i=0}^{k+1} i(k+1) = \frac{(k+1)(k+2)}{2}$$
 Equation (4)

If we substitute $(k+1)$ by n in equation (4), then:

$$\sum_{i=0}^n i(n) = \frac{n(n+1)}{2}$$

So from the principle of inductive proof, we see $i(n)$ satisfies for all n .

4.3.3.3 Temporal Logic and Temporal Induction

Temporal induction is an extension of mathematical induction. In logic, the term temporal logic is used to describe any system of rules and symbolism for representing, and reasoning about, propositions qualified in terms of time [10]. It is sometimes also used to refer to tense logic [11].

A simple example is shown below to describe the temporal logic.

For the statement “I am hungry”.

In a temporal logic we can then express the statements like “I am always hungry”, “I will eventually be hungry”, or “I will be hungry until I eat something”.

Temporal logic has found an important application in formal verification, where it is used to state requirements of hardware or software systems [12]. Two early contenders for formal verification are linear temporal logic and computation tree logic. However, linear temporal logic is currently the most widely used for concurrent system [13]. Therefore, in this project, the linear temporal logic has been used in properties checking.

There are two main types of properties which we have described above in section 4.3.2. Both safety and liveness properties can be expressed by linear temporal logic. The difference is shown below:

Safety properties usually state that something bad never happens. In general, every safety properties is a counterexample and it has a finite prefix, then it is extended to the every safety properties is still a counterexample in an infinite time.

Liveness properties state that something good keeps happening. More generally, for liveness properties, on the other hand, every finite prefix of a counterexample can be extended to an infinite path that satisfies the formula [12].

For this project, in practice, the properties are written and checked in the rules which are shown below:

Prove a behaviour **B** for the design.

1. Prove behaviour **B** is not true at reset (inductive base).
2. Hold that **B** is not true at time t , and then prove **B** is not true at time $t+1$ (inductive step).

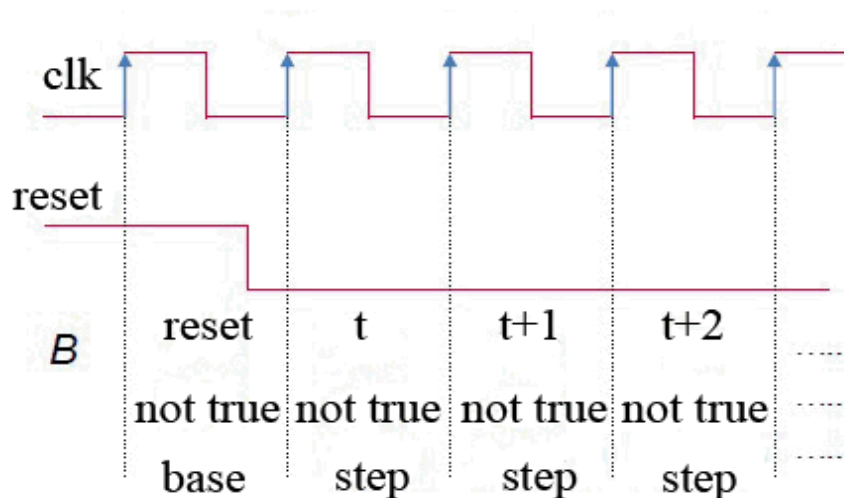


Figure 9 The diagram of temporal induction.

4.3.3.4 Formal sequential verification

Formal sequential method is used for formal property checking. The steps of this method are shown below:

Firstly, compile a formal model of the design.

Secondly, create a specific and clear specification.

Thirdly, build an efficient proof algorithm and apply it automatically.

Finally, analyse the proof results.

The first step of formal property checking is to build a formal model of the design. The model is like a state transition graph structure as a Kripke Structure.

Kripke Structure is a five tuple $K = \langle S, S_0, R, L, AP \rangle$

Which S is a set of states in a system.

S_0 is a set of initial states in the system. It should be a subset of S . $S_0 \subseteq S$

$R \subseteq S \times S$ indicates the relationships of states transition.

AP is a set of all atomic propositions and their negative propositions.

$L : S \rightarrow 2^{AP}$ is a marking function which labels each state with a set of atomic propositions that are true at that particular state.

In the second step, we will specify properties as a proposition of the design. The definition and types of property are introduced above.

The third step is the most important step, especially for this project. Before we have built formal model and properties which we wish to verify. The proof algorithm is used to connect the model and properties. In the process of verification, the algorithm should be improved again and again.

Using the Kripke structure which we described before:

$$\{s \in S \mid M, s \models f\}$$

Which f is the desired property of the design.

The mission is to find set in which all states satisfy the formula. An example proof algorithm named fixed-point algorithm is shown below:

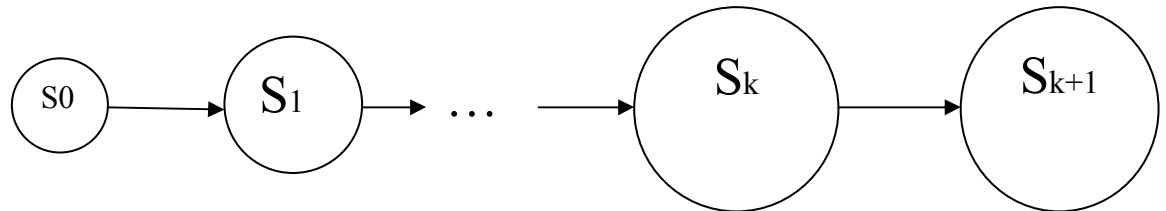


Figure 10 [4] Fixed-point reachable states

From this figure, S_0 is the initial set of states and all new reachable states are from here. S_1 is the first new set of reachable states. Following, the new sets will be generated one by one until no new reachable states generated. In short, when the adjacent sets are matched, the goal is reached.

Finally, the results will be analysed. There are three possible results: pass, fail or undecided.

Pass. The formula is reached. That means the design meets the property specification.

Fail. The algorithm should be improved.

Undecided. The process aborts because of some unexpected reason.

4.4 Simulation-based Verification versus Formal Verification

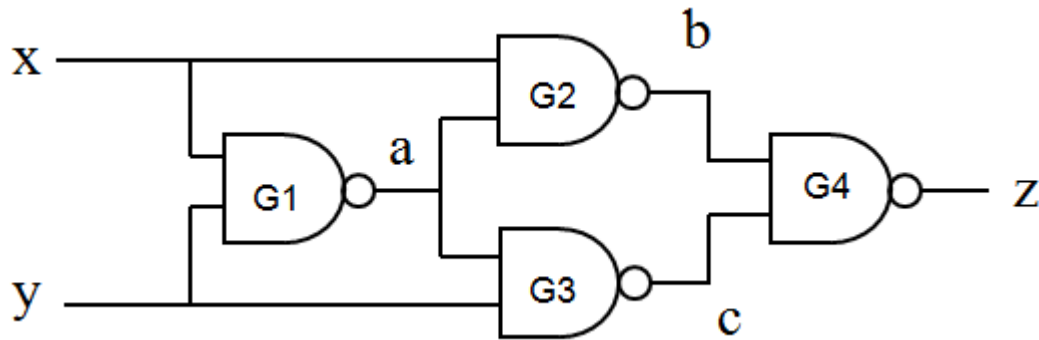
4.4.1 The advantages and disadvantages for each method

	Simulation-based method	Formal method
Strengths	All checking is implemented with independent checker. Simple test writing Coverage is desirable to test.	Complement the simulation
Weaknesses	Consumes a lot of time and effort when design is complex. Depends on designer's understanding of the design.	Does not check for the real program

Table 3 Strengths and weaknesses for each method

4.4.2 Methodology comparison Example

For a basic circuit:



The equation should be $z = \bar{x} \cdot y + x \cdot \bar{y}$

For formal method:

$$z = \bar{b} + \bar{c}$$

$$b = \bar{x} + \bar{a}$$

$$c = \bar{a} + \bar{y}$$

$$a = \bar{x} + \bar{y}$$

$$z = \bar{b} + \bar{c} = \overline{(\bar{x} + \bar{a})} + \overline{(\bar{a} + \bar{y})}$$

$$= a \cdot x + a \cdot y$$

$$= (\bar{x} + \bar{y}) \cdot x + (\bar{x} + \bar{y}) \cdot y$$

$$= x \cdot \bar{y} + \bar{x} \cdot y$$

This is the mathematic transformation of expression to satisfy the specification. It is really a mathematic proof.

For simulation-based method:

The truth table is shown below:

x	y	z
0	0	0
0	1	1
1	0	1
1	1	0

The four test cases should be written to simulate. For the four test cases, each case should satisfy the left hand side is equal to the right hand side of the equation.

4.5 Hybrid method

4.5.1 Introduction

For design verification, all the engineers expect to receive a 100% coverage. One major mission of this is to test whether all the code is exercised. This needs the support by code coverage. In practice, the problem comes with whether the code is really dead or it is just exercised in some particular mode. For example, when the CPU runs in the power-saved mode, the performance decreases to the lowest level. A lot of operations have been terminated. Only a part of code will be executed. As this reason, designers should write a lot of test cases to deal with the different mode or situations. Finally, analyse the all results to decide whether the code is dead indeed.

However, another problem comes. For the complex design, both the RTL code and testbench are much greater than 1000 lines of code. For further work, the coverage analysis and checking would be more complex to implement. The complicated work requires a lot of debugging time and engineering effort. In section 4.2 and 4.3, we have discussed the simulation-based verification and formal verification for achieving coverage closure. Obviously, the semi-conductor industry is not satisfied with these methods and hunger for achieving the coverage closure faster. In section 4.4, we discussed the strengths and weaknesses of both the methods. Therefore, according to the merits and drawbacks which are listed in section 4.5.3, the hybrid method of combining both the two methods is used in the semi-conductor industry in practice to achieve coverage closure early.

The flow of the hybrid method is shown in the figure below. The flow had been implemented successfully in Naresh's project for code coverage analysis last year. In this project, the flow will be fundamentally modified and extended to deal with functional coverage. Functional coverage is essential when assessing the effectiveness of random test generation and achieving 100% functional coverage is usually a hard requirement when using simulation-based method for verification of a design. The hybrid method gives an edge to the project especially when the complexity of the design is huge.

4.5.2 Flow

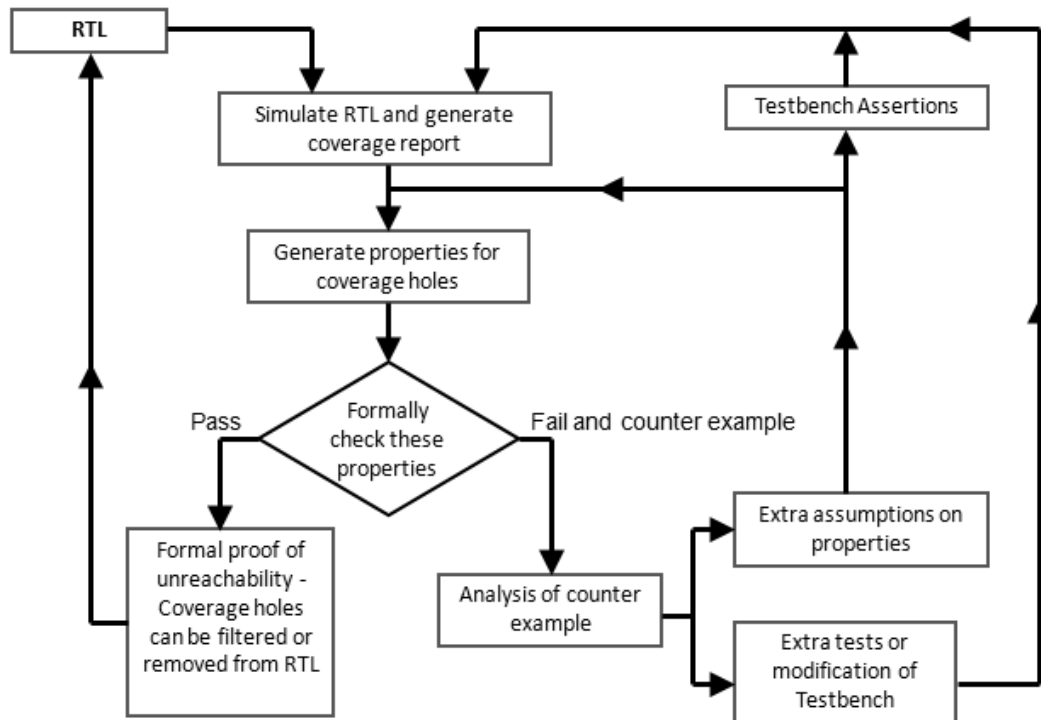


Figure 11 The flow of hybrid method

The flow of hybrid method was already in place for code coverage last year. In this project, this flow will be fundamentally modified and extended to deal with functional coverage. The main jobs for this project focuses on simulation and checking, and finally automate the procedures.

The development of the hybrid method involves six stages.

- Automatically generate coverage points
- Simulation on Modelsim
- Generate coverage report
- Translate coverage holes to properties
- Property checking
- Results analysis

Automatically generate coverage points

The start point for automatically generating coverage points is writing the coverage point sample. However, owing to the temporal language SVA is quite new for using in Infineon and the entire design and monitor are quite new for me, we moved the

start point to translating the functional coverage points from PSL to SVA. More details about translating will be discussed in **Experiment 1**. Through being familiar with the writing of SVA, we started to write the coverage point sample for new function alignment trap, and then generate all possible coverage points with a script which is written in Perl. More details about writing coverage point sample and script writing will be discussed in **Experiment 4** and **Experiment 5**.

Simulation on Medelsim

The RTL is simulated with the testbench, guided by proper design specific environmental constraints. The coverage metrics such as code, structural and functional coverage are defined prior to simulation. In this project, the functional coverage is defined for simulation. More details and results about simulation will be described in **Experiment 5**.

Generate coverage report

The functional coverage report for each coverage point is generated at the end of simulation manually. These reports serve as the primary input for the formal checking. The format in which the coverage report appears varies across different simulator tools. In this project, the coverage report was generated in txt version. The example of coverage report will be displayed in **Experiment 5**.

Translate coverage holes to properties

Due to the temporal language SVA had never been used in property writing in Infineon, the start point for this stage should be writing the simple property. More details about writing properties will be discussed in **Experiment 2**. For automating the properties translating, the procedure is similar as the coverage points generating. The more challenging work is extracting the coverage holes, but not all coverage points. More details about this stage will be described in **Experiment 6**.

Property checking

In this stage, all the properties are checked by formal checker Onespin 360MV. More details about property checking will be explained in Experiment 3 and Experiment 6.

Results analysis

In this stage all the result files are parsed and an overall report will be generated in a proper format. This final result file gives information on how many properties got executed, hold and failed. If the property fails, the formal proof tool would provide a

counter example. This would help verification engineers to generate some extra tests quickly and easily. On the other hand, if a property holds or passes, this is a bug in the design. In traditional method, even if there is a small change in RTL or testbench, the entire regression should be rerun, and then generate coverage report and analyse it again. However, in hybrid method, just a change in testbench assertions is required to modify verification environment. This saves a lot of verification time. More details about this stage will be discussed in **Experiment 6** and **Issue 3**.

4.5.3 Advantages and Disadvantages of hybrid method

Advantages

1. It is easy to see the results and it consumes less time.
2. Debugging of coverage reports is fast and more efficient.
3. Property/Assertion failing provides a counter example and the counter example helps engineers in writing the extra tests.
4. As the method involves more automation, the results from hybrid method have less human errors.
5. More controllability and observability.
6. Using more machine time instead of the human effort.
7. More transparency on the hybrid method and it brings the confidence of the results.

Disadvantages

1. It is necessary to educate engineers about the formal method knowledge, the formal property language.
2. The license of formal checking tool might be very expensive. Especially in this project, there is only one license for formal tool with GUI and it is not possible to check properties in parallel.

4.6 Assertion-based verification

Assertion has become very popular for verification in recent years [1]. An assertion is an if statement with an error condition which indicates that the conditions for expected results or unexpected results. It is the description for the behaviors of the design. During the simulation, if the statement of a behavior is not we expected or the behavior itself is not we expected, the assertion will be failed. The basic assertion description is shown in Figure 12. The flow which the assertion used in verification has been shown in Figure 11.

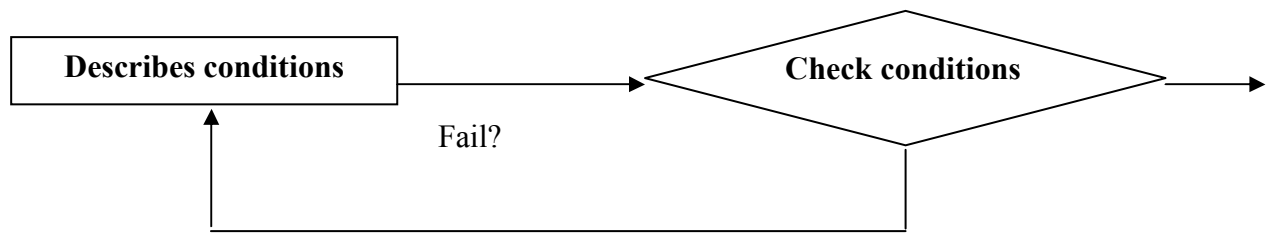


Figure 12 Basic assertion description

The table below describes the merits and why are assertions not used.

	Assertion-based verification
Advantages	<ul style="list-style-type: none">Increase the observability.Reduce the debug time (It would isolate bugs).Improve the integration through correct usage checkingImprove the verification efficiency.Improve the communication through documentation
Why not used	<p>According to the designers' response:</p> <ul style="list-style-type: none">Many designers do not have enough time to add assertionsMany designers believe the assertions increase the code in size.Many designers believe debugging the assertions requires more debugging time.

Table 4 Assertions usage

4.7 Summary

This chapter discussed the basic concept of the methodology and knowledge points we used in this project. For the hybrid method, the specific flow has been described in section 4.5.2. In chapter 6, more details about the implementation of the methodology will be listed with diagrams and results.

Chapter 5 Completing the Verification Cycle

5.1 Verification Cycle

In order to build a robust verification environment, engineers have to work on sweeping the design of the last bugs before the initial hardware fabrication [15]. In the other words, the engineers have to verify the entire chip and system's functionality. Especially in a complex design, it is required hierarchical verification.

From the diagram shown below, this chapter discusses the latter three stages in the verification cycle, from regression to escape analysis.

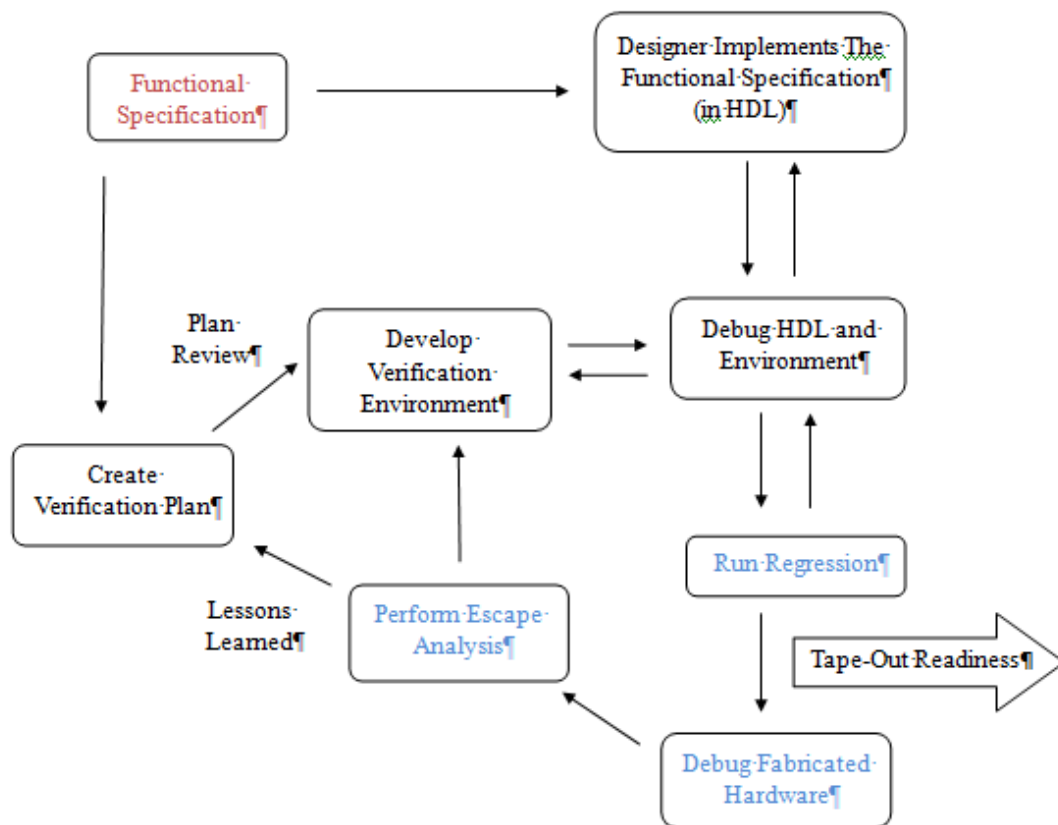


Figure 13 The diagram of verification cycle

5.2 Regression

Regression is the continuous running of tests defined in the verification plan [20]. The regression goal is detecting “unexpected” bugs. The verification team performs a regression simulation on every verification hierarchy level. The regression also ensures the bug fixes work. If the regression completes successfully, the verification teams can proceed quickly to verify the new functionality of the design.

A regression suite contains a huge number of tests that cover as much as possible of the currently implemented functionality of the DUV [20]. The verification team selects the tests based on the functionality and coverage metrics. Generally, the tests are written by experienced engineers or generated by simulation tool automatically.

In semi-conductor company, the regression runs every night and every weekend. For the new function alignment trap we introduced in section 3.3.3.3, we did not run the entire regression for it. We only selected one test to simulate for it because running regression would cost much engineer time. More details about running test will be discussed in **Experiment 5**.

5.3 Tape-Out Readiness

Finally, before the hardware tape-out, it brings us the question of when is the design ready for manufacturing. Certainly, the design must meet the tape-out criteria. The tape-out criteria is a series of checklist and different company has the different tape-out criteria. The flow shown below indicates when is the verification done and when is the design ready for manufacturing.

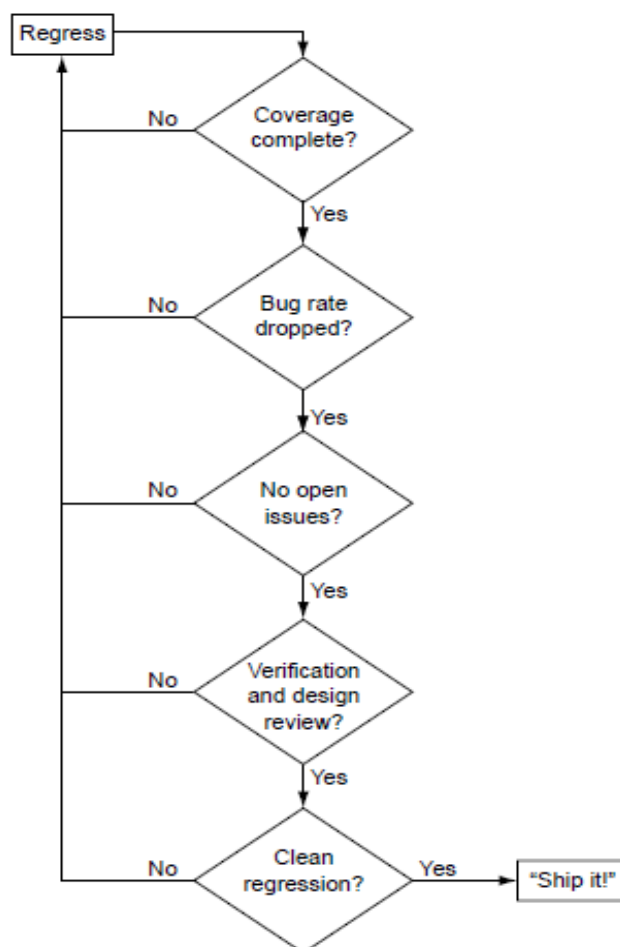


Figure 14 [20] When is verification done?

For tape-out readiness, the most common verification metrics are bug rates and coverage closure.

The bug rate shows the pace at which verification team discovers bugs and designers fixes the them [20].

The coverage closure has been discussed in section 3.4.

The example diagram of bug rates and coverage closure is shown below.

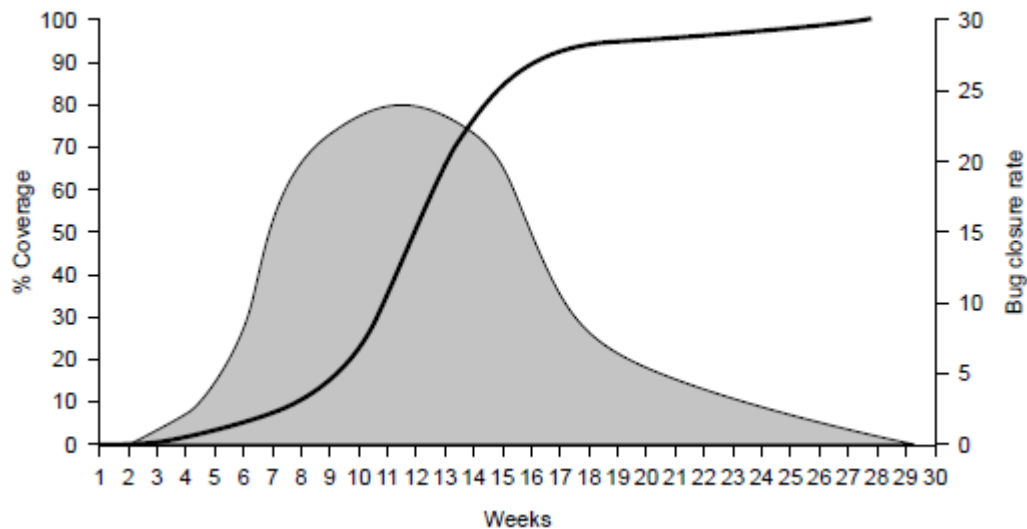


Figure 15 [20] Bug rates and coverage closure of functional coverage

In Infineon, for each version of Tricore, the verification process always lasts more than two years to meet the expected coverage goals.

5.4 Escape Analysis

Escape is the process later than bug found in verification process. Usually, this is a process of bug found in hardware.

Escape analysis has two important aspects:

1. Make sure the bug is fully understood and fix correctly because we do not want another tape-out owing to a bad fix.
2. Understand why the bug escape occurs at first place, and then try to modify the verification plan to avoid it occurring in the future.

Chapter 6 Experiments and Issues

This chapter lists the motivation, objectives, justification and result of each experiment which is performed in this project.

This chapter also lists the interesting issues I met during the project.

The order of the experiments that presented in the chapter is followed by my practising timeline.

Experiment 1

Motivation

The aim of this experiment is getting start with the simulation part of hybrid method. The main goals of this experiment are reading the functional coverage points already done by other engineers in PSL, translating the points from PSL to SVA and finally simulating the functional coverage points with Modelsim.

Objectives

1. Review of functional coverage metrics.
2. Review of simulation based verification knowledge.
3. Understanding of previous work on closing code coverage with a hybrid method.
4. Understanding and study of the formal property language PSL.
5. Study of the temporal language SVA.
6. Write some functional coverage points in temporal language.
7. Skillful use of Modelsim and its commands.
8. Simulate corresponding test cases and generate coverage report for coverage point examples.

Justification

This was the beginning of this project. In order to be familiar with the functionality of Tricore 1.6.1 and get start writing with the language SVA, the first step is reading the functional coverage points already done by the engineers in PSL, and then translating the points from PSL to SVA. The current PSL functional coverage report was already in place. We selected some functional coverage points which were representative from the PSL coverage report. For example, the point `external_halt_request` was never hit after simulation, and the point `asynchronous_halt_trap` was hit a lot of times after simulation. If my SVA simulation results were same as the PSL simulation results, it was implied that my coverage

points in SVA were absolute correct. If they were different, the SVA functional coverage points must be written in another way. Finally, seven functional coverage points had been selected for translating and simulating. The flow of this step is shown below.

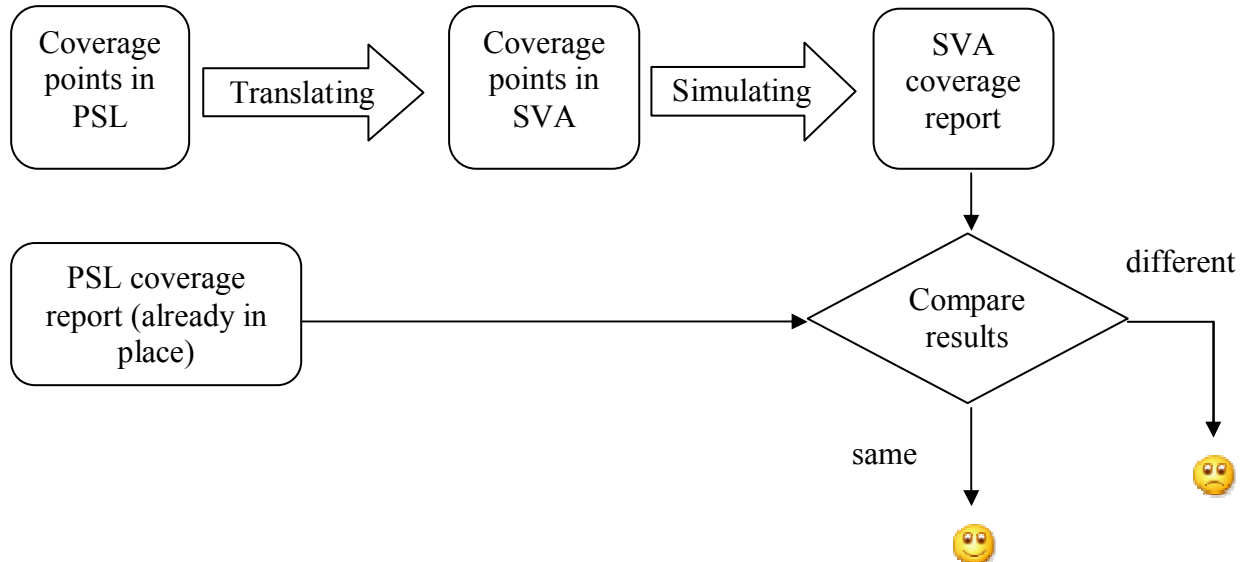


Figure 16 The flow of translating and simulating

The PSL and SVA program examples are shown below. In all the program examples, the key words are highlighted and described below the program. The signals are selected purposelessly and pointlessly in examples.

PSL program example

```
vunit DMBI_EC (monitor_block){  
    default clock = (posedge clk);  
    %for addr_range in 0..15 do  
        cover_name_range_ %{addr_range}: cover {  
            (ls_ex1_instr_opcode[7:0]) && (ls_ex1_st_addr[15:12] == %{addr_range})  
            && !stall_ls_ex1 && cancel_ls_ex1  
        };    //end cover  
    %end  
}    //end vunit
```

vunit --- The verification unit in PSL. PSL is a property language and it needs to be linked to a design unit. The verification unit is as a container integrated a set of properties and lined properties to the design. In this case, DMBI_EC is the name of verification unit and the unit is linked to monitor block.

default clock = (posedge clk) --- For each property or sequence in this unit, set the default clock to positive edge clock. The properties can be either clocked or unclocked. We only focused on the clocked properties in this case.

addr_range --- The variable in this program, the values of this variable are from 0 to 15.

%for, %end --- The macros in PSL. It is different as the “for” loop in C. This is the loop for the variables. For the variable addr_range, its values are {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}. At first loop, all the %{addr_range} is assigned to 0. At second loop, all the %{addr_range} is assigned to 1. The variable is assigned from 0 to 15 in this way and the program will run out of the loop after substituting all the values.

cover --- The cover directive directs the verification tool to check if the property or sequence has been exercised by the test suite or given constraints [16]. The cover directive may optionally include a character string containing a message to report when the specification in property or sequence occurs [17].

SVA program example

The macro %for is not supported by SVA. Therefore, in this case, the values of variable must be assigned manually one by one.

```
program DMBI_EC (input [15:0] ls_ex1_instr_opcode, input [15:0]
ls_ex1_st_addr, stall_ls_ex1, cancel_ls_ex1){

cover_name_range_0: cover property (

@(posedge clk)

(ls_ex1_instr_opcode[7:0]) && (ls_ex1_st_addr[15:12] == 0)
&& !stall_ls_ex1 && cancel_ls_ex1);

...

cover_name_range_15: cover property (

@(posedge clk)

(ls_ex1_instr_opcode[7:0]) && (ls_ex1_st_addr[15:12] == 15)
&& !stall_ls_ex1 && cancel_ls_ex1);

endprogram

bind monitor DMBI_EC DMBI_EC_BIND (ls_ex1_instr_opcode,
ls_ex1_st_addr, stall_ls_ex1, cancel_ls_ex1)
```

program --- Define a new SVA module definition, all the input and output posts must appear in parentheses after the module name.

@(posedge clk) --- Only in this property or sequence, set the default clock to positive edge clock.

cover property --- It is the same as the directive **cover** in PSL.

bind --- bind command. The SVA bind file is externally instantiated into the target design module without making any modification to the target module itself [18]. In this case, **monitor** is the name of bind source, **DMBI_EC** is the name of the file needs to be bound, **DMBI_EC_BIND** is the name of bind block.

Result

We expected the simulation results were the same as PSL simulation results. Fortunately, after running the relevant tests, the results were the same. In the seven selected coverage points, three points were hit and four points were not hit. These were the same as the PSL coverage report. It was implied my SVA coverage points were correct. Owing to this experiment was related to the original PSL coverage points, the screen shot cannot be provided here.

Experiment 2

Motivation

The aim of this experiment is following the hybrid method flow, getting start with formal verification part in hybrid method. The main goal of this experiment is skillful using the temporal language SVA and writing the property examples in temporal induction.

Objectives

1. Review of assertion based verification and formal property checking techniques.
2. Understanding of temporal induction for formal property writing.
3. Understanding of previous work on closing code coverage with a hybrid method.
4. Write some property examples for formal checker.
5. Skillful use of the temporal language SVA.

Justification

This was the step for developing the SVA programming. This is a challenging work because the language support is relatively new in the formal tool and the engineers have little practical experience in this language as the announcement from Infineon (Appendix D). In the previous work, engineers in Infineon used ITL for properties describing. Through the hard work on programming, the properties are written in three ways. Through proving on Onespin 360MV, all the properties could be checked successfully.

Result

The property in SVA can be written in three ways.

1. Use the property implication in SVA

```
property_specification: assert property(property_name)
property property_name;
    @(posedge clk)
    !(signal_1 && signal_2 && signal_3 == 16'hffff)
    |=> !(signal_1 && signal_2 && signal_3 == 16'hffff) ;
endproperty: property_name
```

"|=>" --- Property implication. A implies B and B delays one cycle. In this case, we assume the behaviour `(signal_1 && signal_2 && signal_3 == 16'hffff)` is **B** for a design. **!B => !B** means whenever **B** is **not true**, at the **next** cycle, **B** is still **not true**.

property, endproperty --- Build a block for SVA allows us to compose expressions and specify operations into temporal sequences [19].

assert property --- Concurrent assertion. Concurrent assertions are used to check behaviour such as this. These are statements that assert that specified properties must be true [20]. Actually, the "assert property" is similar as a proof.

2. Use the assumption in SVA.

```
property property_name;  
    @(posedge clk)  
    !(signal_1 && signal_2 && signal_3 == 16'hfff);  
endproperty: property_name  
property property_name_next;  
    @(posedge clk)  
    ##1 !(signal_1 && signal_2 && signal_3 == 16'hfff);  
endproperty: property_name_next  
assume property(property_name)  
assert property(property_name_next)
```

“##1” --- One cycle delay.

assume --- The assume key word in SVA is used to describe the environment of a block [21]. It is used to define the legal behaviour of inputs to a block for formal analysis. They are assumed to be true when proving an assertion. As the property shown above, the keyword “assume” is always used with the keyword “assert” together.

In this program, we assume the property “`property_name`” is not true at time t , and then prove the property “`property_name_next`” is not true at the next cycle (time $t+1$).

3. Use the simplified property.

```
property property_name;  
    @(posedge clk)  
    !(signal_1 && signal_2 && signal_3 == 16'hfff);  
endproperty: property_name  
assert property(property_name)
```

This is the simplified property. However, this kind of property is only used in the checker that checks the properties cycle by cycle. For example, Onespin 360MV.

Experiment 3

Motivation

The aim of this experiment is continuing the formal method and focusing on the property checking. The main goal of this experiment is checking the property examples on onespın 360MV and analysing the results.

Objectives

1. Review of the formal property checking techniques.
2. Review of the methodology for formal method.
3. Entirely understanding of temporal induction for formal property checking.
4. Study of the commands, database and checking tool.
5. Analysis of the checking results.

Justification

After writing the properties, the properties should be checked on a formal checker. The formal checker we used is Onespın 360MV. More details of Onespın 360MV will be listed in **Appendix C**. Only SVA is supported by Onespın 360MV as the properties. On the formal checker, property checking is always reflected in assertion checking. We met a lot of problems when we were compiling the properties at first. More details about the problems and solutions will be discussed in section **Issue 1**.

Using formal method to analyse functional coverage holes

Result

The example of checking result from onespın 360MV is shown below:

OneSpin 360 (R) - Database "default"

Session Setup File Edit CC/MV EC-Flow EC-View Tools Window Help

golden

Module Verification Status

Design: golden Unit: default

all objects

Name	Proof Status	Validity
<any status>	<any status>	<any validity>
Assertions		
sva/i_tc16/top_level_inst/top_level_assert_1	hold_bounded (10)	up_to_date
base	hold	up_to_date
step	fail	up_to_date
vacuous	unproven	up_to_date
bounded	hold_bounded (10)	up_to_date
Constraints		
Macros		
Properties		
SVA declarations		

Shell

Messages

```
-I- Checking radius 9 ... (limit is 10)
-I- Examination window: [t-2,t+9]
-R- Bounded case of assertion 'sva/i_tc16/top_level_inst/top_level_assert_1' holds, assumption not contradictory (checked in 49.70 sec CPU, 4517 MB used)
-I- Checking bounded case of assertion 'sva/i_tc16/top_level_inst/top_level_assert_1'
-I- Checking radius 10 ... (limit is 10)
-I- Examination window: [t-2,t+10]
-R- Bounded case of assertion 'sva/i_tc16/top_level_inst/top_level_assert_1' holds, assumption not contradictory (checked in 01 min 24 sec CPU, 6101 MB used)
-R- Assertion 'sva/i_tc16/top_level_inst/top_level_assert_1' holds.
mv>
```

Hide Shell Shell Mode Interrupt Help

Launch Thu Sep 23, 12:07:54 File Browser: Documents brslog1: /work/scratch1/ge /work/scratch1/gengrui/old OneSpin 360 (R) - Database

Figure 17 The example result of single property checking

Using formal method to analyse functional coverage holes

OneSpin 360 (R) - Database "default"

Session Setup File Edit CC/MV EC-Flow EC-View Tools Window Help

Module Verification Status

Design: golden Unit: default

all objects

Name	Proof Status	Validity
Assertions	<any status>	<any validity>
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_XNOR_RC	hold	up_to_date
base	hold	up_to_date
step	hold	up_to_date
vacuous	hold	up_to_date
bounded	hold_bounded (unlimited)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_ST_A_BO_idx	hold_bounded (1)	up_to_date
base	hold	up_to_date
step	fail	up_to_date
vacuous	unproven	up_to_date
bounded	hold_bounded (1)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_ST_H_BO_post	hold_bounded (1)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_ST_B_BO_pre	hold_bounded (1)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_PACK_RRR	hold	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_LEA_BOL_base_offset	hold_bounded (1)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_MSUBMS_H_RRR1_DDddule	hold	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_ST_DD_BO_post	hold_bounded (1)	up_to_date
sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_MSUBRS_H_RRR1_ddddlle	hold	up_to_date

Shell

Messages

```

-R- Base case of assertion 'sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_AND_SRR' holds, assumption not contradictory (checked in 7.52 sec CPU, 5334 MB used)
-I- Checking bounded case of assertion 'sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_AND_SRR'
-I- Checking radius 1 ... (limit is 10)
-I- Examination window: [t-2,t+0]
-R- Bounded case of assertion 'sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_AND_SRR' holds
-I- Checking step case of assertion 'sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_AND_SRR'
-I- Examination window: [t+0,t+1]
-R- Step case of assertion 'sva/i_tc16/top_level_new_point_inst/gr_alignment_trap_AND_SRR' holds, assumption not contradictory (checked in 6.09 sec CPU, 5334 MB used)
  
```

check_assertion 1% Show Details Hide Shell Shell Mode Interrupt Help

Launch Thu Sep 23, 12:27:22 [File Browser: Documents] [brslog1: /work/scratch1/gen /work/scratch1/gengrui/top_p OneSpin 360 (R) - Database

Figure 18 The result of multi property checking

From the screen shot, four items are checked during assertion checking. They are "base", "step", "vacuous" and "bounded".

Base --- inductive base. In temporal induction, it means that the checker checks at reset. If this item fails, the property fails at reset.

Step --- inductive step. If this item fails, the property checking fails at inductive step.

Vacuous --- vacuous truth. Generally, this item should be held. If the result of this item is unproven, the property failed before proving vacuous truth.

Bounded --- checking property cycle by cycle. By default, the limit is set to 10 (10 cycles). If the result of this item is bounded hold, the property failed before checking 10 cycles. If the result is hold, the property has been held 10 cycles. However, this is not really property held because the limit is just 10. If the result is unlimited hold, the property has been held forever.

Experiment 4

Motivation

The aim of this experiment is to research the new functional coverage point alignment trap which was never verified by other engineers. The main goal of this experiment is writing the samples of this coverage point and property.

Objectives

1. Review of functional coverage metrics, especially cross-product coverage.
2. Review of coverage closure knowledge.
3. Research of the functional coverage point alignment trap.
4. Research of the corresponding architecture and instruction set of Tricore 1.6.1.
5. Simulate test cases for alignment trap and generate coverage report.

Justification

This is the new function which has never been verified. The Alignment Trap has been introduced in section 3.3.3.3. There are totally 877 instructions used for Tricore 1.6.1. However, we are only interested in load and store instructions. The other instructions can be the reference when simulating and checking. Through understanding this function, a coverage point sample and a property sample should be written manually. The coverage point sample and property sample would be used as the template for Perl programming. More details will be described in section **Experiment 5**.

Result

Coverage point, the sample of alignment trap

```
program alignment_trap (  
  input clk, demi_aln_trap, ls_ex1_instr_valid,  
  input [15:0] ls_ex1_instr_opcode)  
  property_specification: cover property (instruction_name)  
    (@(posedge clk)  
      demi_aln_trap && ls_ex1_instr_valid && ls_ex1_instr_opcode ==  
      16'hffff);  
  
  endprogram  
  
  bind monitor alignment_trap alignment_trap_bind (  
    clk, demi_aln_trap, ls_ex1_instr_valid, ls_ex1_instr_opcode)
```

Property sample

```
property_specification: assert property (instruction_name)  
  
  property instruction_name;  
    @(posedge clk)  
      !(demi_aln_trap && ls_ex1_instr_valid && ls_ex1_instr_opcode == 16'hffff);  
  
  endproperty: instruction_name
```

demi_aln_trap --- alignment trap signal.

ls_ex1_instr_valid, ls_ex1_instr_opcode

ls --- On the pipeline LS (load and store). There are three pipelines for Tricore 1.6.1: load/store (LS) pipeline, Integer pipeline (IP) and Loop pipeline.

ex1 --- On the stage Execute 1. 6-stage Pipeline: Fetch 1, Fetch 2, Predecode, Decode, Execute 1 and Execute 2.

instr --- Instruction

valid --- If the instruction is valid.

opcode --- The opcode of instruction. Generally, the opcode is 16-bit. The bit[1] defines which pipeline instruction it is. If the value of this bit is 1, this instruction is IP pipeline instruction. If the value of this bit is 0, this instruction LS or Loop pipeline instruction.

Experiment 5

Motivation

The aim of this experiment is to design and implement a procedure for simulation to automatically generate the functional coverage report. The main goal of this experiment is to write a script to generate all the alignment trap points automatically in Perl, and then simulate the corresponding tests for these points on Modelsim.

Objectives

1. Study and skillful use of the scripting language Perl.
2. Based on the coverage point sample, generate coverage point for each instruction automatically (877 instructions in total).
3. Simulate all coverage points and generate functional coverage report for alignment trap.
4. Review of the coverage closure knowledge and requirement.

Justification

There are 877 instructions used for Tricore 1.6.1. It is impossible to assign the 877 instruction names and opcodes to coverage point manually. A script should be written to generate the 877 points automatically. The flow of writing this script is shown below.

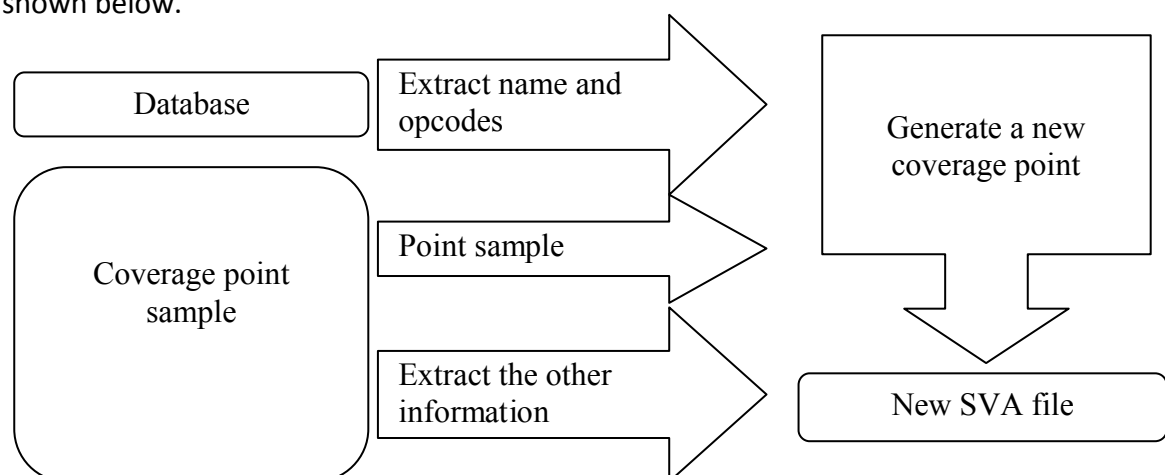


Figure 19 Flow of writing script for coverage points

Using formal method to analyse functional coverage holes

From the flow, there are 5 steps for writing this script.

Step 1. Extract the instruction name and opcodes from the database and store them in a hash as keys and values.

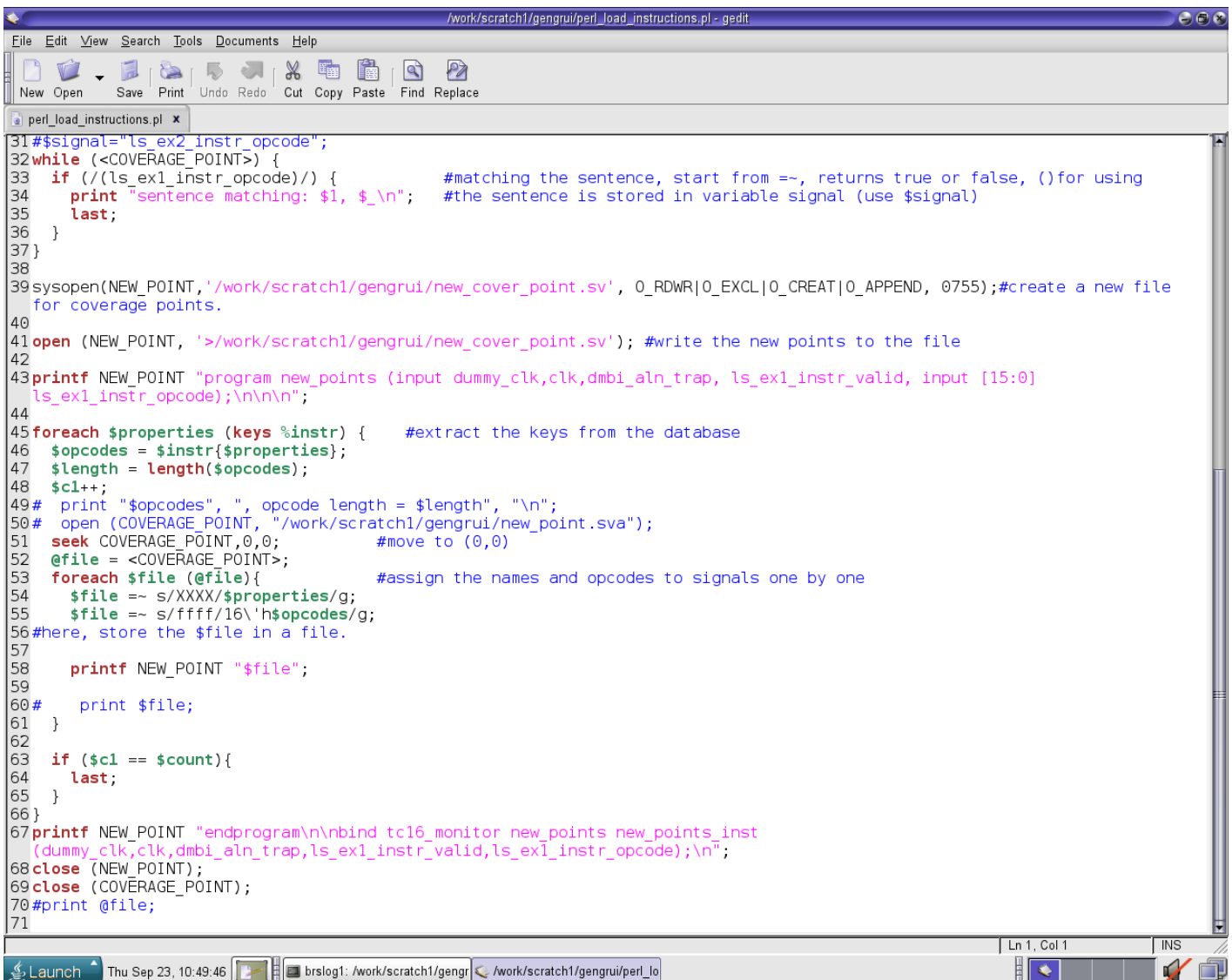
Step 2. Find the locations of instruction name and opcode in the sample.

Step 3. Replace the name (**instruction_name**) and opcode (**opcode == ffff**) by the new name and opcode from database one by one. These are the new coverage points.

Step 4. Open a new SVA file, store the new coverage points one by one in this file.

Step 5. Extract the other information and store them in the new SVA file. For example, the inputs and the bind information.

After running the script, the important information had been extracted from database and coverage point sample. The result is shown below.



```
31 #signal="ls_ex2_instr_opcode";
32 while (<COVERAGE_POINT>) {
33     if (/ls_ex1_instr_opcode/) {          #matching the sentence, start from =~, returns true or false, ()for using
34         print "sentence matching: $1, $_\n"; #the sentence is stored in variable signal (use $signal)
35         last;
36     }
37 }
38
39 sysopen(NEW_POINT, '/work/scratch1/gengrui/new_cover_point.sv', O_RDWR|O_EXCL|O_CREAT|O_APPEND, 0755); #create a new file
    for coverage points.
40
41 open (NEW_POINT, '>/work/scratch1/gengrui/new_cover_point.sv'); #write the new points to the file
42
43 printf NEW_POINT "program new_points (input dummy_clk,clk,dmbi_aln_trap, ls_ex1_instr_valid, input [15:0]
    ls_ex1_instr_opcode);\n\n";
44
45 foreach $properties (keys %instr) {      #extract the keys from the database
46     $opcodes = $instr{$properties};
47     $length = length($opcodes);
48     $cl++;
49     # print "$opcodes", ", opcode length = $length", "\n";
50     # open (COVERAGE_POINT, "/work/scratch1/gengrui/new_point.sva");
51     seek COVERAGE_POINT,0,0;             #move to (0,0)
52     @file = <COVERAGE_POINT>;
53     foreach $file (@file){                #assign the names and opcodes to signals one by one
54         $file =~ s/XXXX/$properties/g;
55         $file =~ s/ffff/16'h$opcodes/g;
56         #here, store the $file in a file.
57
58         printf NEW_POINT "$file";
59
60     # print $file;
61     }
62
63     if ($cl == $count){
64         last;
65     }
66 }
67 printf NEW_POINT "endprogram\n\nbind tcl6_monitor new_points new_points_inst
    (dummy_clk,clk,dmbi_aln_trap,ls_ex1_instr_valid,ls_ex1_instr_opcode);\n";
68 close (NEW_POINT);
69 close (COVERAGE_POINT);
70 #print @file;
71
```

Figure 20 The example of script code in Perl

Using formal method to analyse functional coverage holes

```
File Edit View Terminal Tabs Help
property gr_aln_trap_LDMST_BO_idx; ls_ex1_instr_opcode==0x2169
property gr_aln_trap_ST_A_ABS; ls_ex1_instr_opcode==0xa5
property gr_aln_trap_MSUB_Q_RRR1_dddudue; ls_ex1_instr_opcode==0x63
property gr_aln_trap_INSERT_RCRW; ls_ex1_instr_opcode==0xd7
property gr_aln_trap_CSUB_RRR; ls_ex1_instr_opcode==0x2b
property gr_aln_trap_SEL_RCR; ls_ex1_instr_opcode==0xab
property gr_aln_trap_MUL_U_RR2_Ddd; ls_ex1_instr_opcode==0x6873
property gr_aln_trap_MSUBS_H_RRR1_Ddddule; ls_ex1_instr_opcode==0x38a3
property gr_aln_trap_MADD_F_RRR; ls_ex1_instr_opcode==0x6b
property gr_aln_trap_XOR_EQ_RR; ls_ex1_instr_opcode==0x2f0b
property gr_aln_trap_ST_Q_ABS; ls_ex1_instr_opcode==0x65
property gr_aln_trap_MSUBS_Q_RRR1_Ddddle; ls_ex1_instr_opcode==0x3963
property gr_aln_trap_ADD_A_SRR; ls_ex1_instr_opcode==0x30
property gr_aln_trap_MOV_A_SRR; ls_ex1_instr_opcode==0x60
property gr_aln_trap_MOV_SRC_long; ls_ex1_instr_opcode==0xd2
property gr_aln_trap_SH_XOR_T_BIT; ls_ex1_instr_opcode==0xa7
property gr_aln_trap_CACHIA_WI_BO_pre; ls_ex1_instr_opcode==0x1d89
property gr_aln_trap_LDLCX_ABS; ls_ex1_instr_opcode==0x15
property gr_aln_trap_MADDR_Q_RRR1_dddudue; ls_ex1_instr_opcode==0x43 (1)
property gr_aln_trap_XOR_LT_RR; ls_ex1_instr_opcode==0x310b
property gr_aln_trap_LOOP_SBR; ls_ex1_instr_opcode==0xfc
property gr_aln_trap_MADD_Q_RRR1_dddde; ls_ex1_instr_opcode==0x43
property gr_aln_trap_MADD_Q_RRR1_dddudue; ls_ex1_instr_opcode==0x43
property gr_aln_trap_EXTR_U_RRRW; ls_ex1_instr_opcode==0x57
property gr_aln_trap_CMPSWAP_BO_brev; ls_ex1_instr_opcode==0x69
property gr_aln_trap_LD_HU_BO_base_offset; ls_ex1_instr_opcode==0x2309
The following values are in the database: 8f 26a9 89 1089 6b d9 3ca3 89 2ea3 2a4b 12a9 1409 300b 6b0b 1a89 3bc3 2743 1ea9 26 1c0f 1229 6f0b 2163 820b b3 214b
1589 2fe3 1a 2643 2d 250b 1cc3 86 1c89 73 0d fa 88 2f89 8b 38e3 58 2529 6b /f0b de 8a/3 2649 54 230b 390b dd ce 1883 184b 1/ 9d 2d 2009 b/ 3a 29 /d 3a0b 0f
830b 19 290b 2d 8f 1e83 1e4d d7 8b 3aa3 6301 3e43 0c 338b 7e 6a73 89 05 b3 5c0b 43 3cb3 48 680b 05 0f 790b 4e 9f b4 a8 1ae3 7d e0 220b 2da9 89 a6 01 1f89 720
b 1269 1689 1149 2a8b 09 1f 0d 530b 97 5d 108b 4101 3a4b 8b 2789 104b 2c0b 2149 75 c3 4b 20 03 29 53 270b 2c83 01 a0 fe 29 89 3b63 2c89 00 c7 130b 3ce3 140b
17 49 80 89 2329 1729 2063 09 40 b9 6b ad 3eb3 3d83 6a03 79 4075 93 2989 1ac3 49 8f 7b0b 1e43 e1 288b 04 3f 0b 23 9f e9 00 3fa3 1d43 b0 ab 3ee3 05 5a0b 0b bb
1809 a9 87 900b 370b 67 32 0b 13 2463 84 69 14 174b 43 bc 110d 14a9 1b43 1b8b 15 1349 e823 0b 1aa3 6d 63 2dc3 2629 2543 2443 69 2da3 a7 118b 930b 47 408f 1e
89 180b 4b 204b 61 5a4b 28 29 0d 64 89 0f 2929 a9 4801 1c43 47 620b 27 ba 1b83 2289 8f 580b 07 ad 3da3 3e83 6b 330b 1863 37 5f0b 2249 25 a9 a7 bf 3cc3 00 1b8
9 2589 e4 00 190b 62 3f 2709 2663 15 0e 2e83 1309 1d0b 2689 7e0b 09 89 45 33 1963 a9 4a4b 0b 25 13 83 2069 8f 09 1993 8f 2a89 3ae3 520b 8e 74 1489 0f 10 500b
7f 0d 09 1129 1fe3 3be3 13 3b df a7 2449 298b 87 1529 2889 0b 1789 18c3 3dc3 7f 8873 91 aa ac 09 1249 d4 2ca3 5e c8 93 19a3 3863 b3 18a3 6f 2b 43 8a03 ad 8f
2c 3983 0d 6e0b c5 75 4b a3 c2 1829 87 33 2043 76 38c3 39 7d0b 0f 124b 09 6001 630b 2b8b c7 2349 39c3 3b43 27 398b 3ea3 6b c4 378b 200b ab 85 be 69 7c0f 340
b 07 4b ab 75 a5 c3 150b 4b 1ec3 e5 2129 208b 2ec3 1509 75 1983 37 6803 2243 8f 24a9 4001 7b 600b f8 3ba3 2a0b 8a23 a9 700b 32 da 144b 410f 1a4b 19a9 33 df 2
609 114b 4b 400f 100b 4d 8f 1f83 2f 3db3 2ea9 bd 89 ed 3b83 6b 13 2489 bf 3fc3 93 2ee3 ff 4c01 2b0b e3 0b 2909 2809 8c 2c8b 1de3 32 1029 32 0d 400b 1bb3 0f 9
9 47 148b 17a9 1209 8a 83 0b 89 158b 94 6b 6b 2a 49 154b 13 02 4b 3d43 140d 2f8b 1989 2269 308b a3 18a9 8b 1069 f9 1289 32 1c0b 240b 7c0b 0f 1c63 2109 27a9 1
30d 53 5e0b 43 2829 210b 89 4b 120b 1b0b 63 1b93 1943 8f 63 ea23 2ca9 1d63 57 0b 8b 1eb3 134b 8b 16 6b 1da3 2d89 6201 1709 33 4c 2029 28a9 3d63 07 b3 1a0b 22
8b 2509 96 318b 63 2563 29 0d 4301 7a 2143 29 e3 3a8b 0b 46 8823 a9 1f0b 93 2cc3 c3 418f 238b 63 f6 07 3e63 a3 1ea3 87 1ca9 29 0d 49 2ce3 d8 6b b7 800b 2b 29
22a9 a9 3c43 194b 5f 160d a9 0b 6a0b 1609 1a83 19c3 00 08 2d83 5f 188b 164b 7d0f 218b 39a3 4b 128b 1ab3 33 e8 93 c3 5a fd 730b 8b 6823 0f 67 13 93 17 2b89 8
b 2d 2229 1369 ec 2409 01 9b 68 1cb3 17 3fe3 e3 a4 83 78 a9 1109 3de3 920b e803 198b 63 15a9 2b 0f 1ee3 7e0f 2749 1009 1ce3 27 1893 92 1dc3 25a9 16a9 37 1e 3
28b 8803 82 1889 b5 a3 3c e2 9e 57 2fc3 278b 85 1929 3c8b 568b 34 c7 bd ff 3883 3ec3 c6 a9 89 a5 8f 3943 8b 20a9 13 3b8b 760b 5b0b 0d 5c 89 2f83 1629 0d ea03
3f83 a5 2049 6c 1fb3 47 1d0f e5 0d 6f 3843 fd 1ca3 1d4b 280b 09 18e3 29 810b 6a 2b 1bc3 3ac3 59 1329 1429 3c83 dc 1a8b 4b 1ba3 110b 1da9 1843 0f 89 1be3 85
3e 348b 2e89 248b 768b 44 29 9a 39e3 37 ca 2de3 83 4901 06 ea 4e0b 8b 09 09 7a0b 6e 138b 3c63 258b 75 0b 53 1fc3 150d 2b 2729 6b 2263 1b0f 0f 22 2849 fb 18b3
f4 380b e3 19b3 780b 85 6b 2549 52 15 10a9 2e 3a83 3fb3 4201 1049 57 320b cc 93 1e63 4b d5 12 ae 1c83 3c0b cd 0d 6a23 1b63 480b 4b a9 1909 120d 2089 6b 11 5
90b 53 1169 2763 2429 3b0b ee c7 19e3 77 1d 560b 29a9 1db3 53 33 2fa3 27 1d83 a2 33 42 1f 1b 2209 7c c9 24 05 388b 1fa3 2169 a5 63 d7 2b ab 6873 38a3 6b 2f0b
65 3963 30 60 d2 a7 1d89 15 43 310b fc 43 43 57 69 2309
There are 877 opcodes.

sentence matching: ls_ex1_instr_opcode, dmbi_aln_trap && ls_ex1_instr_valid && ls_ex1_instr_opcode == ffff

[1] + Done (3) gedit perl_load_instructions.pl
fgengrui:tc2/iw5/tc2.5/tricore_1.6.1@brslog1:/<2>gengrui (4)
Launch Thu Sep 23, 10:50:13 brslog1: /work/scratch1/gengrui
```

Figure 21 Perl script running results

Location (1) --- All the instruction names and their opcodes had been found and printed out by the terminal. On this line, the instruction name is “MADDR_Q_RRR1_dddudue” and its opcode is 0x43.

Location (2) --- All the instruction opcodes had been extracted and printed out.

Location (3) --- The signal “ls_ex1_instr_opcode” had been found in the coverage point sample.

Location (4) --- The signal “ls_ex1_instr_opcode” and its original value in sample file had been located.

Result

After running a test named “msvhdl.do”, the cover directives were produced. We met a problem during the simulation. More details about the problem and solution will be introduced in **Issue 2**. The simulation result is shown below.

Using formal method to analyse functional coverage holes

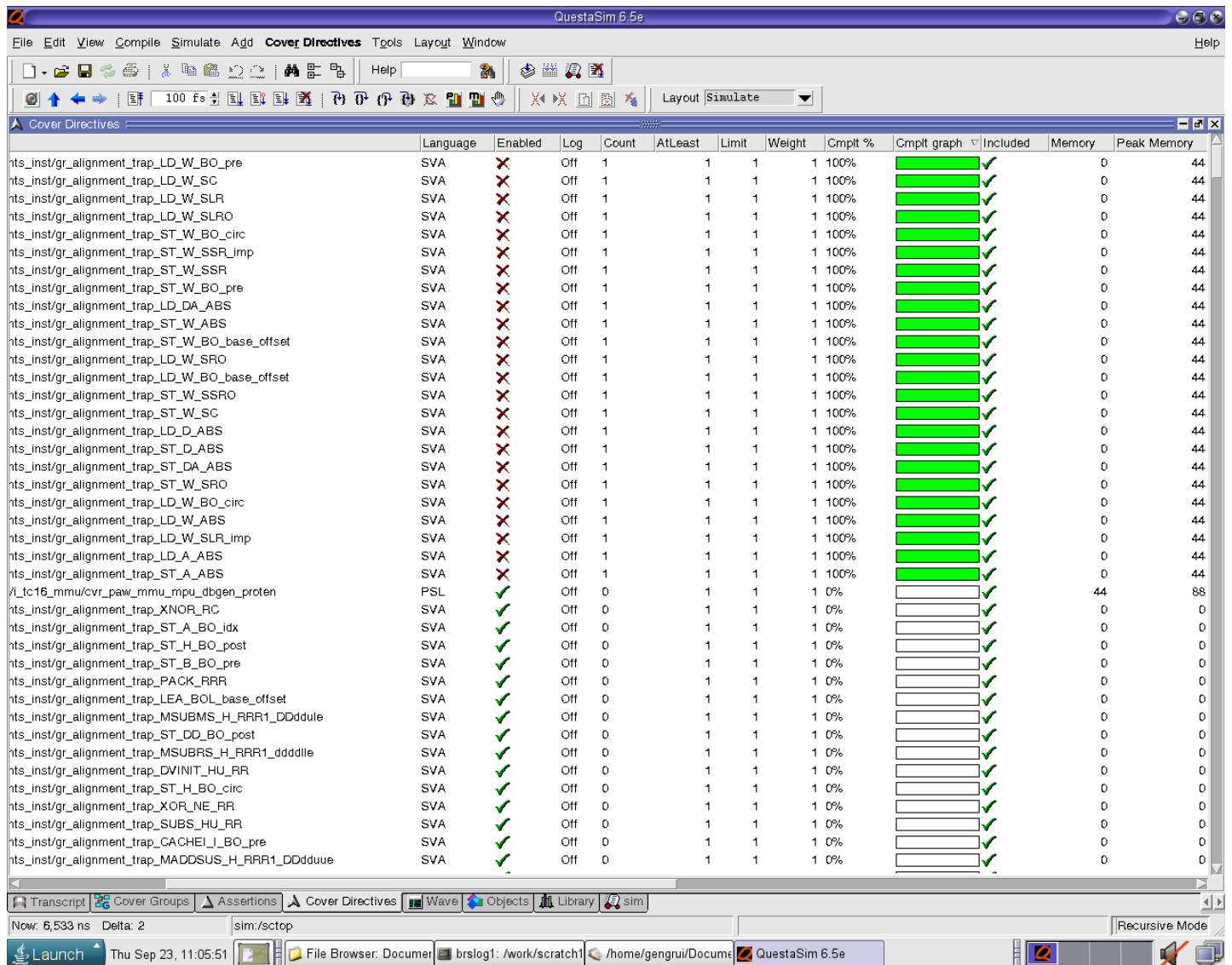
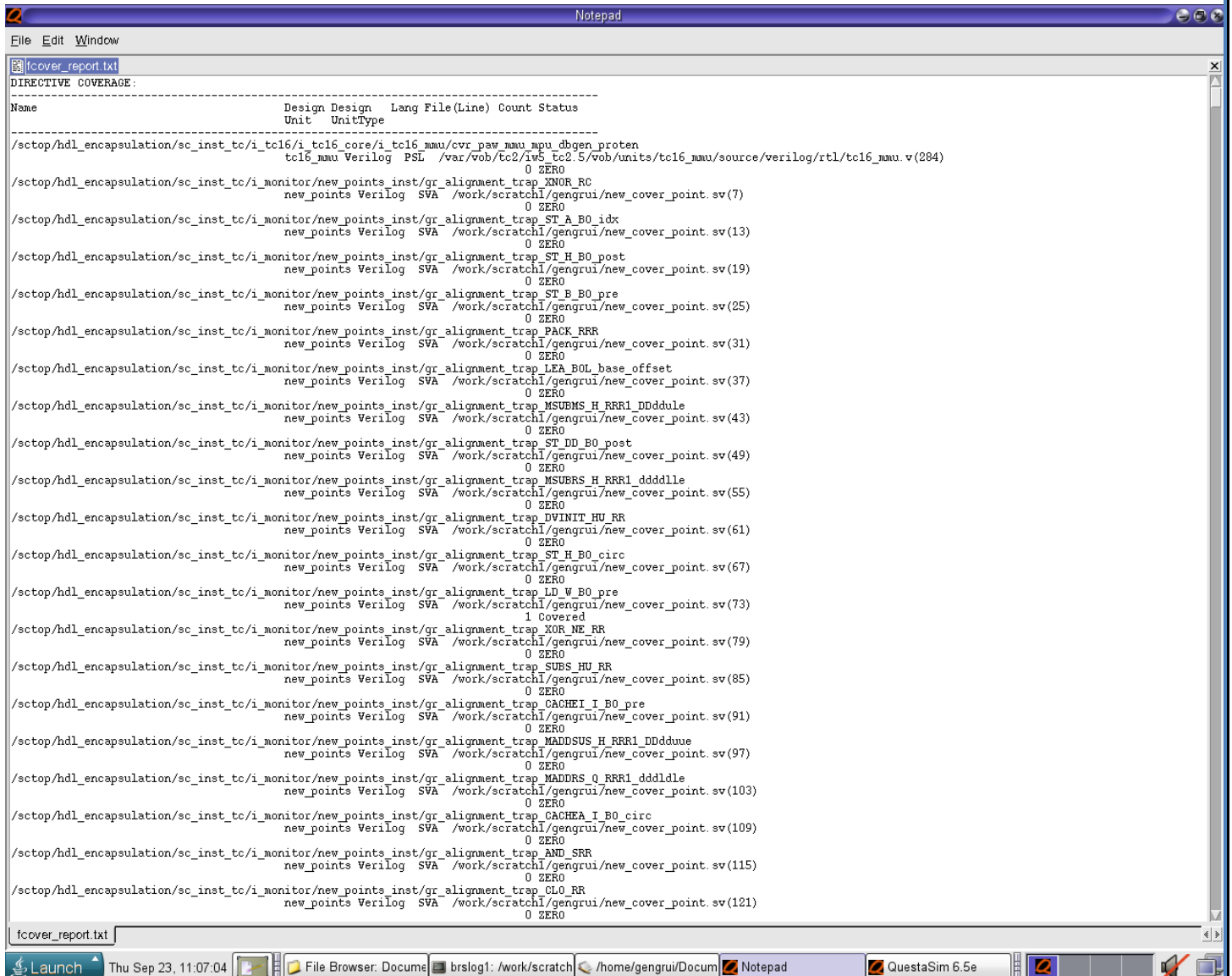


Figure 22 Simulation results for alignment trap

From the simulation result, only 24 of 877 coverage points are hit. That is because we only simulated one test on these points. If we run a regression, more points would be hit. However, running regression would cost a lot of time. We did not do it at present.

After simulating the coverage points, the functional coverage report had been generated as shown below.

Using formal method to analyse functional coverage holes



```
Notepad
File Edit Window
fcover_report.txt
DIRECTIVE COVERAGE:
-----
Name                               Design Design  Lang File(Line) Count Status
Unit UnitType
-----
/sctop/hdl_encapsulation/sc_inst_tc/i_tc16/i_tc16_core/i_tc16_mmu/cvr_paw_mmu_mpu_dbgen_proten
tc16_mmu Verilog PSL /var/vob/tc2/iw5_tc2.5/vob/units/tc16_mmu/source/verilog/rtl/tc16_mmu.v(284)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_XMOR_RC
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(7)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_ST_A_B0_idx
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(13)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_ST_H_B0_post
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(19)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_ST_B_B0_pre
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(25)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_PACK_FRR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(31)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_LEA_B0L_base_offset
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(37)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_MSUBMS_H_RRR1_DDDdle
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(43)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_ST_DD_B0_post
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(49)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_MSUBRS_H_RRR1_dddldle
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(55)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_DVINIT_HU_RR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(61)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_ST_H_B0_circ
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(67)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_LD_W_B0_pre
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(73)
1 Covered
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_XOR_NE_RR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(79)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_SUBS_HU_RR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(85)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_CACHEI_I_B0_pre
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(91)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_MADDUSUS_H_RRR1_DDDduue
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(97)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_MADDRS_Q_RRR1_dddldle
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(103)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_CACHEA_I_B0_circ
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(109)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_AND_SRR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(115)
0 ZERO
/sctop/hdl_encapsulation/sc_inst_tc/i_monitor/new_points_inst/gr_alignment_trap_CL0_RR
new_points Verilog SVA /work/scratch1/gengrui/new_cover_point.sv(121)
0 ZERO
```

Figure 23 The functional coverage report in txt version

Experiment 6

Motivation

The aim of this experiment is to design and implement a procedure to automatically generate suitable properties for functional coverage holes. The main goal of this experiment is to write another script to translate the coverage holes into properties and finally check the properties.

Justification

The flow of writing this script is similar as the flow in experiment 5. The more challenging work is that this script must filter the hit points and extract the holes from the txt coverage report.

Result

Unfortunately, as the problem of the database, the properties are not able to be checked successfully. The problem and solution will be described in **Issue 3**.

Experiment 7

Motivation

The aim of this experiment is comparing the two formal languages PSL and SVA.

Result

Through studying and using formal language PSL and SVA, the similarities and differences for the two languages are summarised as the diagram shown below.

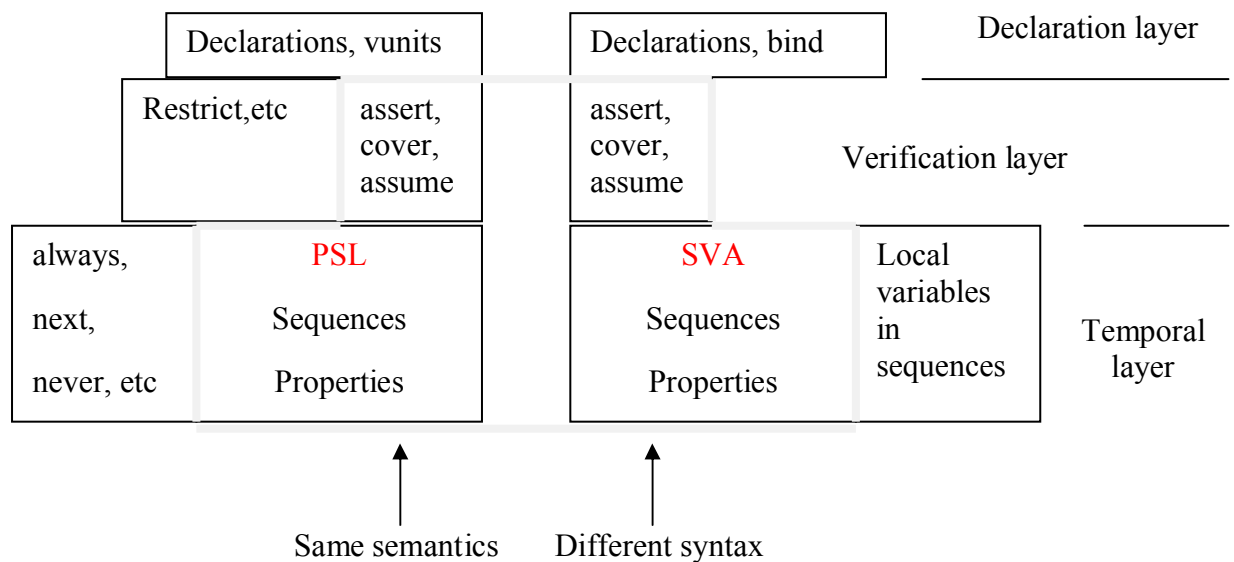


Figure 24 The similarities and differences for the two temporal language

On the declaration layer, vunit in PSL defined the verification unit and it can be linked to the design unit directly. In SVA, unlike the PSL vunit, bind file require the assertions be wrapped in a module that includes the input and output ports [18].

For the clock definition, the clock must be defined inside the property or sequence in SVA. However, the clock could be defined either inside or outside the property or sequence in PSL.

Issue 1

Problem

In order to check the properties on Onespin 360MV, the database must be built at first. Generally, the monitor should be set as the database because all the signals used in properties are defined in the monitor. However, for some reasons, the monitor block cannot be read by Onespin 360MV.

Solution

We set the design as the database directly. However, it brings another problem. In the original monitor block, it defines all the signals with their full path from the design. In order to deal with this problem, a personal monitor block is set to map my properties to the design and the personal monitor was written in SVA because Onespin 360MV can only read SVA during compiling and checking. That was a very challenging work because every signal is assigned complicatedly. Every related signal must be extracted from the original monitor block.

For example:

The signal_1 is the signal we used in a property.

```
assign signal_1 = signal_2 && signal_3 && signal_4_full_path;  
assign signal_2 = signal_5[16'hffff] && signal_6;  
assign signal_3 = signal_3_full_path;  
...  
...
```

This is not a clever method, but more useful for simple properties.

In order to increase the observability and readability of the program, a top level program is written to include all the subroutines. The diagram is shown below.

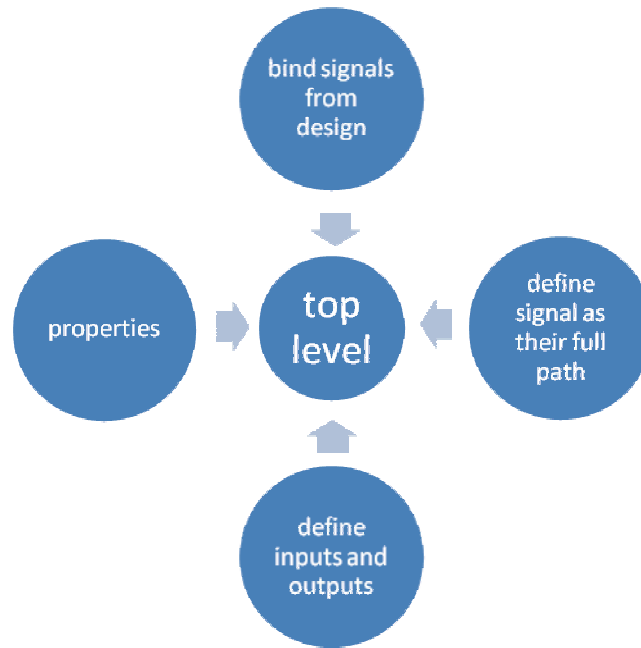


Figure 25 The structure of the personal monitor

Issue 2

Problem

When we was simulating the test “msvhdl.do” for alignment trap coverage points, the window “cover directives” was always empty. In the other word, the Modelsim did not display any results. The coverage points were absolutely correct we have proven before. The test file which the company provided is only used for PSL coverage points before. In the other word, it is never simulated for SVA points. For this reason, I wrote a PSL alignment trap point and run the test for it. Finally, we located the problem was in the test file.

Solution

The test file “msvhdl.do” is integrated a lot of Modelsim command. There were lots of arguments following the command “VSIM” and these arguments decided the operations during simulation.

Important commands and arguments used in modifying the test:

vsim --- The vsim command is used to invoke the VSIM simulator and load a new design[22]. When we run this command, the environment will be rebuilt, but not updated.

vsim -gui --- Starts the ModelSim GUI without loading a design and redirects the standard output to the GUI Transcript window. Optional [22].

vsim -lib --- Define a default library where the simulator will look for the design.

vsim -nopsl --- Ignore compiled assertions [23].

vcom/vlog -nopsi --- Ignore the embedded assertions [23].
vcom/vlog -pslfile <name> Specify external assertion file [23].

Issue 3

Problem

From the Issue 1, a personal monitor block was used to deal with the checking problem. However, that kind of solution was only suited to a simple signal definition. When we used a register in the definition program, Onespin 360MV could not pass the syntax all the time. The reason is that Onespin 360MV can only pass the sequential logic syntax inside the property or sequence.

Solution

After a short meeting, we analysed the current situation.

1. Onespin 360MV can only pass SVA program during compiling and checking.
2. Onespin 360MV can pass verilog program from the design (database).

Finally, we decided to rebuild the database. The original build file is written by Tool Command Language (TCL) and it should be modified at that time. We tried to include the original monitor block into the design tc16. That seemed strange. However, that was the last effort we could do. Unfortunately, Onespin 360MV is quite sensitive to read the monitor block.

Failure analysis

1. The build.tcl should be modified. However, as the time pressure, there is no enough time for me to learn the scripting language TCL very well.
2. The engineer must know the entire design and monitor block very well. However, I only know a part of the design and monitor. There is no enough time for me to be familiar with them.

Chapter 7 Summary and Conclusion

Actually, this project is an extension of Naresh's project last year. The main targets of his project are dead code elimination and using formal method for property checking based on structural coverage model. The new target in my project is using formal method for property checking not just from the structure, but also the functionality. The main difference is the based coverage model.

The aim of this project is to research, derive and design a novel hybrid method which uses the simulation method and formal methods to analyse the functional coverage holes. The project involves about researching the industrial approach towards verifying the complex designs and knowing whether they are really sufficient for complete verification of the design. That is not only the learning of two programming language, but also the complete verification operation and hierarchical design analysis. This project touches each and every point and is explained in detail throughout this thesis.

Chapter 8 Possible improvements

After solving the Onespin 360MV problem, this method would be perfect for functional coverage analysis.

For some further work:

1. Fix the database problem and check the alignment trap properties.
2. Run the properties in batch mode (no GUI). The checking would be faster if we run the properties in batch mode. Moreover, we have only one license for Onespin 360MV, but we have five licenses for batch mode.
3. Do not run the properties in bounded clocks.
4. Based on the alignment trap, look for all possible traps. For example, the interrupt trap, halt trap etc.

BIBLIOGRAPHY

- [1] Kerstin Eder, *Design Verification lecture slides*.: University of Bristol, 2009-2010.
- [2] Harry D.Foster L.Perry, *Applied Formal Verification*.: Douglas, McGraw-Hill Companies, Inc, 2005.
- [3] Adam, Huizinga, Dorota Kolawa, *Automated Defect Prevention, Best Practices in Software Management*.: Wiley-IEEE Computer Society Press, 2007.
- [4] Adam C Krolnik, David.J.Lacey Harry.D.Foster, *Assertion-Based Design*. New York, NY: Kluwer Academic Publishers, 2003.
- [5] Lasse Koskela, *Introduction to code coverage*.: Accenture Technology Solutions, Copyright 2004.
- [6] Serdar Tasiran, *Coverage Metrics for Functional Validation of Hardware Designs*.: Compaq Systems Research Centre, Kurt Keutzer University of California, Berkeley, 2001.
- [7] Laurent Fournier, Avi Ziv, Keren Zohar Hezi Azatchi, *Advanced Analysis Techniques for Cross-Product Coverage*.: IBM Research Laboratory in Haifa.
- [8] E. Harel, M. Orgad, S. Ur, and A. Ziv. R. Grinwald, *User defined coverage - a tool supported methodology for design verification*.: In Proceedings of the 35th Design Automation Conference, June 1998.
- [9] Sandeep K Shukla, *An Introduction to Formal Verification*.: CECS/ICS/UCI.
- [10] Oski Technology, *Coverage: the link between SIMULATION and FORMAL*., 2006.
- [11] John Harrison, *Formal Verification In Industry (I)*.: Intel Corporation, 1999.
- [12] David H. Sanford, *The Cambridge Dictionary of Philosophy*, 2nd ed., vol. "Implication".
- [13] Simon Blackburn, *The Oxford Dictionary of Philosophy*.: Oxford University Press, p. 388, 1994, vol. "vacuous".
- [14] J, A. Daoud Franklin, *Proof in Mathematics: An Introduction*.: Sydney: Quakers Hill Press., 1996.
- [15] J. van Benthem, *The Logic Of Time*, second edition, Ed.: Kluwer, Dordrecht, 1991.
- [16] Yde Venema, *Temporal Logic*., vol. Chapter 10.
- [17] M. Hansen and Zhou Chaochen, *Duration Calculus: logical foundations, Formal*

Aspects of Computing.: 9:283

- [18] Alessandro Artale, *FORMAL METHODS.*: Faculty of Computer Science – Free University of Bolzano, vol. LECTURE III: LINEAR TEMPORAL LOGIC.
- [19] Naresh.Ramaram, *Using a Formal Property Checker for Simulation Coverage Closure.*: A collaboration between Infineon Technologies UK Ltd. and the University of Bristol, 2009.
- [20] John C.Goss, Wolfgang Roesner Bruce Wile, *Comprehensive Functional Verification.*: Elsevier Inc, 2005.
- [21] DOULOS, *Assertion-Based Verification with PSL*, 3rd ed.: Copyright by Doulos Ltd. All rights reserved, 2004-2007.
- [22] Accellera, *Property Specification Language Reference Manual*, 11th ed.: Copyright by Accellera. All rights reserved, June,9 2004.
- [23] Clifford E. Cummings, *SystemVerilog Assertions Design Tricks and SVA Bind Files.*: Sunburst Design, Inc., SNUG-2009 San Jose, CA Voted Best Paper 1st Place.
- [24] Palos Verdes Peninsula, *SystemVerilog Assertions Handbook.*: Copyright by VhdlCohen Publishing, 2005.
- [25] DOULOS, *SystemVerilog Assertions Tutorial.*:
<http://www.doulos.com/knowhow/sysverilog/tutorial/assertions/>.
- [26] Ph.D. Ming-Hwa Wang, *SystemVerilog Assertions (SVA).*: COEN 388 Principles of Computer-Aided Engineering Design, Department of Computer Engineering, Santa Clara University.
- [27] *ModelSim® Reference Manual*, 63rd ed.: © 1991-2008 Mentor Graphics Corporation All rights reserved., May 2008.
- [28] www.model.com, *ModelSim 6.0, PSL Quick Guide.*: Copyright Mentor Graphics Corporation, 2004.
- [29] *Technical Presentation TriCore™ 32-bit Unified Processor.*: Infineon, November 2002.
- [30] <http://www.onespin-solutions.com/360mv.php>,.: Onespin Solutions.
- [31] *Getting Started with System Verilog Assertions.*: Sutherland HDL, Inc, Portland, Oregon, 2006.
- [32] M. Fisher, D. Gabbay, and G. Gough, editors H. Barringer, *Proceedings of the Second International Conference on Temporal Logic.*, ICTL'97,to appear.

Appendix A: System Verilog Assertion

System Verilog Assertions (SVA) is a powerful subset of the IEEE 1800 System Verilog standard which is a component of the SystemVerilog language donated to IEEE by Accellera [2]. It is a verification technique which is embedded in the language. Generally, the Property Specification Language (PSL) temporal operators may be more expressive. For SVA, although its operators cannot support all the Linear-time Temporal Logic (LTL) operators, the operators are still expressive enough for properties. SVA is the extension to the Verilog language. Verilog language is designed for hardware description, not for verification. It cannot describe sequential logic very well. For describing a complex sequential logic, engineers should spend a lot of time on writing code and the errors always occur during simulation. SVA is a perfect language to describe sequential logic and the embedded functions are used for particular sequential logic and generating functional coverage automated.

For example, for a property, when the signal “a” is high in one cycle, the signal “b” must be high in the following 2-4 cycles. For Verilog, a large number of code is required for describing this property. For SVA, we just use five lines of code.

```
property a2b_p;  
    @(posedge sclk) $rose(a) |-> [2:4] $rose(b);  
  
endproperty  
  
a2b_a: assert property(a2b_p)  
a2b_c: cover property(a2b_p)
```

“property” and “endproperty” are the key words used for describing property.
“a2b_p” is the name of property.

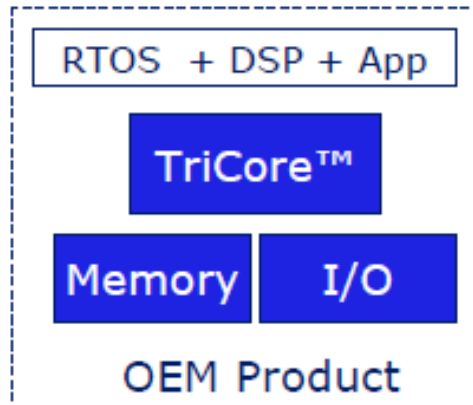
“\$rose” is the function to detect the rising edge of signals.

“assert property” and “cover property” are also the key words of SVA. “assert property” is used for assertion and “a2b_a” is the assertion name. “a2b_p” is another property which used as a reference for “a2b_a”. “cover property” is a cover statement used to record the success of the assertion.

Assertions are permitted to be used and SVA supports two types of assertions specification. They are immediate assertions and concurrent assertions. Immediate assertions are procedural statements and are mainly used in simulation. In evaluation, they evaluate using simulation event-based semantics [2] [10]. However, formal property checkers always evaluate assertions by using cycle-based semantics. The semantic inconsistency occurs here. Therefore, generally, we use concurrent assertions which are clock based semantic for formal method.

Appendix B: Tricore microcontroller [29]

Unified Processor Reduces Complexity & Cost



- 1 processor does the work of 2.
No need for inter-processor communication
- Dynamic assignment of DSP vs. controller code in response to changes in the system requirements
- Fast context switch is the key
- Reduced number of resources (no duplications)
- Smaller, simpler silicon



Target Segments

Automotive

- Engine Management
- Transmission Control
- ABS
- Active Suspension
- Infotainment
- X-by Wire

Industrial Control

- Robotics
- PLC's
- Servo-Drives
- Motor Control
- Power-Inverters
- Machine-Tool Control (CNC)

Data Storage & Processing

- Hard Disk Drives
- Tape Drives
- Scanners
- Digital Copiers
- FAX Machines

Telecom/Datacom

- Communication Boards (LAN)
- Modems
- Mobile Communication
- Switches
- Routers



Consumer

- DVD / CD-ROM
- HDTV
- Set Top Boxes
- Games
- Printers

TriCore™
UNIFIED PROCESSOR

Unique 3-in-1 Feature Set!



Microcontroller

- Fast interrupt response
- Fast context switch
- Low code size through use of 16-bit & 32-bit instructions
- Powerful bit manipulation unit
- Powerful comparison instructions
- Integrated peripheral support

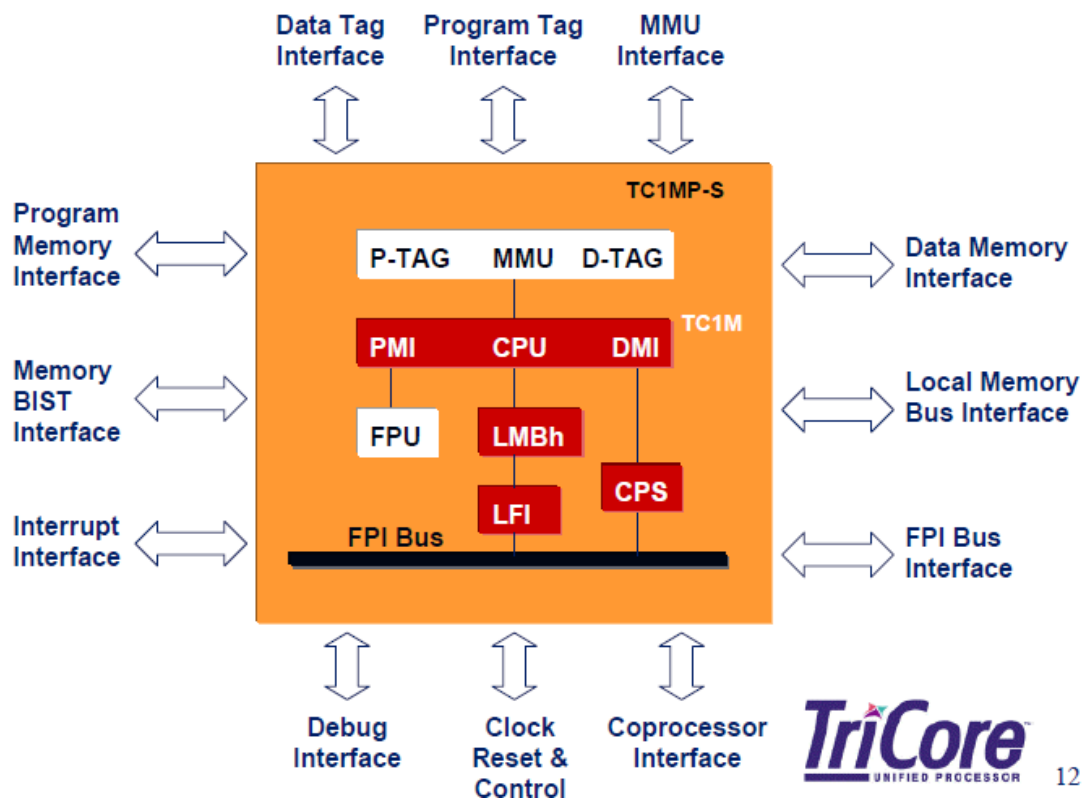
DSP

- Sustainable single-cycle dual MAC
- Packed/SIMD instructions
- DSP addressing modes
- Zero overhead loops
- Saturation
- Rounding
- Q-Math

RISC Processor

- 32-bit load/store Harvard architecture
- Super-scalar execution
- Shallow 4-stage pipeline
- Uniform register set
- Single data-memory model
- Memory protection
- C/C++ and RTOS support

TC1MP-S: General View & Connections



Appendix C: Onespin 360MV

These details are taken from Onespin Solutions homepage <http://www.onespin-solutions.com/360mv.php> as reference.

When mentioned the most comprehensive formal assertion-based verification (ABV) solution for RTL designs, OneSpin's 360 MV product family is the best choice. To be frank, it is the only verification solution in the industry, which provides an exhaustive coverage analysis, and ensures users to consistently find intricate design errors missed by other approaches. Moreover, this fact has been confirmed in more than 300 verification projects.

360 MV, which covers the widely range of formal ABV applications, is highly suitable for formal verification starters, experienced users and experts – from fully automatic RTL checks all the way to OneSpin's patented, highest quality gap-free verification. Besides, it offers design, verification and integration engineers the best approach (simulation or formal techniques) and give users full control over the right verification effort/quality mix for their design and project schedule.

As to standard formal ABV applications, such as automatic RTL checks, implementation intent verification, and verification of end-to-end functional requirements, 360 MV powerfully supports and extends them with a unique Operational ABV methodology.



Figure 1: 360 MV - the most comprehensive formal ABV solution

Operational ABV supplies several functions, such as simplifying verification planning, easing assertion writing working from timing diagrams, and is the key for an exhaustive coverage analysis ensuring no design behaviour is missed during verification.

In that case, 360 MV's comprehensive spectrum of formal ABV applications guarantees step-by-step learning and adoption of formal ABV techniques, and allows new users to become productive in short time. At every level, assertions are exhaustively verified – equivalent to running and checking every possible simulation trace of a design. Due to its requirement of no input stimuli, verification can start weeks or months before testbenches become available. Proven assertions can be reused in system-level testbenches as coverage points, or to speed integration verification

Appendix D: Source code and declaration from Infineon Technology UK Limited



University of Bristol,
Department of Computer Science,
Merchant Venturers Building,
Woodland Road,
Bristol,
BS8 1UB,
UK

23rd September 2010

To whom it may concern,

Rui Geng has been undertaking work for his project 'Using Formal Methods to Analyse Functional Coverage Holes' under my guidance at Infineon Technologies in Bristol. The main goal of the project was to check the feasibility of using a formal property checking tool to identify unreachable functional coverage points. A secondary goal of the project was to develop a scripted flow to automate the checking of all functional coverage points not hit during a simulation regression for reachability. The work was intended to be based on a flow already in place which uses a formal property checking tool to identify unreachable *code* coverage. A major motivation is the fact that both formal property checking tools and functional coverage directives can be written in the same languages – so called Hardware Verification Languages (HVL).

The HVL chosen for the project was System Verilog Assertions (SVA), since this was the only suitable language supported by the formal property checking tool in use at Infineon Technologies (the OneSpin 360MV tool). There were a couple of potential disadvantages in the use of this language. Firstly the language support is relatively new in the formal tool. Secondly we have little practical experience in using this language at Infineon, Bristol.

These two disadvantages have indeed proved problematic for the project and Rui Geng has had to spend considerable time overcoming difficulties with the use of SVA, both in the formal property checking tool and in the collection of functional coverage with a simulator. Despite this Rui Geng has succeeded in achieving

Infineon Technologies UK Limited, Infineon House, Great Western Court, Hunts Ground Road, Stoke Gifford, Bristol, BS34 8HP
Registered Office: Infineon House, Great Western Court, Hunts Ground Road, Stoke Gifford, Bristol, BS24 8HP
Registered No: 3782938, England

the primary goal of the project. Some functional coverage has been written in SVA, and properties have been generated in order to check whether the functional coverage could be hit. The properties were successfully run on the formal property checking tool and unreachable functional coverage was identified. As such I regard the project as having been successful. Unfortunately the problems encountered mean that up until now there has not been sufficient time to fully automate the flow.

In order to write the functional coverage Rui Geng has written code to interrogate the instruction data base, identify particular types of instruction and generate SVA code for functional coverage. He has also written code to modify the SVA functional coverage points into properties to be checked by the formal property checker. In total he has written around 650 lines of code in PERL and SVA. As agreed at the start of the project with his academic supervisor, Dr. Kerstin Eder, the code developed remains the property of Infineon Technologies. Since the code is confidential it unfortunately cannot be published as part of Rui Geng's dissertation.

Yours sincerely,



Dr T. D. Blackmore