NTNU
Innovation and Creativity

Verification of an AES RTL Model with an Advanced Object-Oriented Testbench in SystemVerilog

Henrik Ruud

Master of Science in Electronics
Submission date: January 2007
Supervisor: Einar Johan Aas, IET
Co-supervisor: Torstein Hernes Dybdahl, Falanx Microsystems AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Verification is the process that is performed to show that the design is according to the specification.
SystemVerilog was developed to improve the Verilog language with attention to the verification tasks. The assignment involves the following:

- Develop a verification plan for the AES module
- Develop a testbench architecture using SystemVerilog constrained randomisation
- Use assertions and functional coverage to track progress and quality of verification

Assignment given: 29. August 2006
Supervisor: Einar Johan Aas, IET

Summary

This Master's thesis reports the verification planning and verification process of a Verilog RTL model. Modern verification techniques like constrained randomization, assertions, functional coverage analysis and object orientation are demonstrated on an AES RTL model.

The work of this thesis was naturally divided in three phases: First, a phase of literature studies to get to know the basics of verification. Second, the creation of a verification plan for the selected module. Third, implementation of the testbench, and simulation tasks.

The verification plan created states the goals for the simulation. It also states plans for details about the testbench, like architecture, stimuli generation, randomization, assertions, and coverage collection. The implementation was done using the SystemVerilog language. The testbench was simulated using the Synopsys VCS verification software.

During simulation, coverage metrics were analyzed to track the progress and completeness of the simulation. Assertions were analyzed to check for errors in the behavior during simulation. The analysis carried out revealed high code coverage for the simulations, and no major errors in the verified module.

Preface

This Master's thesis was submitted to the Norwegian University of Science and Technology (NTNU), Department of Electronics and Telecommunication. It is the result of a thesis problem given by ARM Norway AS. The work was carried out during the autumn of 2006, starting in August 2006 and ending in January 2007.

I would like to thank my supervisors, Professor Einar J. Aas, NTNU and Torstein Hernes Dybdahl, ARM Norway AS, for their guidance and feedback through the whole thesis process.

Trondheim, January 23rd, 2007

Henrik Ruud

Contents

List of Figures

List of Tables

Abbreviations

AES ..... Advanced Encryption Standard
BFM ..... Bus Functional Model
DITL ..... Day in the Life
DUT ..... Device Under Test
DVCon ..... Design & Verification conference & exhibition
DVE ..... Discovery Visualization Environment
FIFO ..... First In First Out
GCC ..... GNU Compiler Collection
HDL ..... Hardware Description Language
HDVL ..... Hardware Description and Verification Language
HVL ..... Hardware Verification Language
IP ..... Intellectual Property
SVA ..... SystemVerilog Assertions
URG ..... Unified Report Generator
VMM ..... Verification Methodology Manual

Chapter 1

Introduction

THE MAIN TOPIC of this master thesis is the planning and employment of modern verification techniques to verify the functionality and properties of a selected open source RTL model.

1.1 Motivation

Verification is the task of verifying that an implementation meets its functional intend. As digital hardware designs grow more and more complex, the verification task gets more important in order to ensure that bugs are found before the design hits the market. At the same time, a more complex chip requires a more complex verification process.

From the 1990s until today, several new technologies have been introduced to help the verification process. To speed up the verification process, stimuli is generated by constrained random functions, instead of manual generation of each input set. Code coverage and functional coverage is used to track the progress and quality of the simulations. Assertions are implemented to monitor the behavior of the device being tested. Testbenches are split up using object oriented techniques to make them more maintainable and reusable.

Many of these techniques have been introduced by the verification languages E and Vera in the late 1990s. The recent years, a new language called SystemVerilog has been standardized. It is a verification and high-level modelling language based on Verilog, which makes it easy to integrate with Verilog models to be verified.

1.2 Problem Formulation

When the thesis was given, it was made clear that the work would naturally consist of three main parts, where most of the focus would be on the first two:

The first part of the job was to get to know the theory about verification planning, and verification using SystemVerilog. A number of books and articles on the subject were recommended by the supervisors. This part also contained the task of selecting a module suitable for being verified.

The second part involved making a verification plan for the selected module. The verification process and testbench planned should exploit modern verification techniques from the theory, like constrained randomization, functional coverage, object orientation and assertions.

The third part was the practical work: To implement the testbench planned in the verification plan, carry out the simulations using Synopsys VCS and analyze the results with the available tools.

1.3 Contribution

In this thesis a verification plan for an AES RTL model has been created, based on theory about constrained randomization, functional coverage, object orientation and assertion based simulation. The planned verification system has then been implemented using SystemVerilog, and simulated using Synopsys VCS. Simulation results have been analyzed to ensure that the verification process has been adequate.

1.4 Organization of the Thesis

Chapter 2 of the thesis is a theory chapter that explains the techniques used in the verification process, and will serve as the theory basis for the planning and implementation process.

Chapter 3 is a theory chapter introducing the SystemVerilog language and the Synopsys VCS tools. It contains the theory background for the implementation of the testbench in SystemVerilog, and the simulation of the system using Synopsys VCS.

Chapters 4, 5 and 6 present my contributions. Chapter 4 steps through the creation of a verification plan for the chosen module. The module selection process is also described.

In Chapter 5, the implementation of the testbench is presented. The simulation of the testbench is described in Chapter 6.

Chapter 7 contains the discussion of the simulation results. It also contains a discussion on the testbench design and future work. Conclusions are drawn in chapter

8.

Chapter 2

Verification Theory

THIS CHAPTER gives an introduction to techniques commonly used in the verification process, and gives the necessary background for the verification planning and implementation described in later chapters.

2.1 What is Verification?

The purpose of the verification process is to verify that the design functions like it is intended to. This process should both verify that the implemented functions behave correctly, and that all parts of the planned system are implemented.

2.1.1 Verification vs Test

Verification must not be confused with the test process. Verification is the process of verifying the functionality of a design, while the test process tests if the design has been manufactured correctly. This is shown in Figure 2.1.

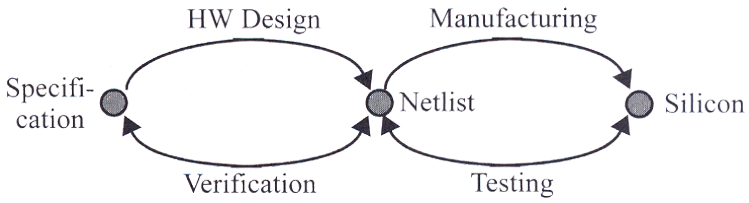

Figure 2.1: Verification vs test [1]

2.1.2 Verification Technologies

Simulation based verification and formal verification are the two most common technologies used in verification.

Simulation Based Verification

Simulation has traditionally been the main technology for verification. In its simplest form, a testbench is used to generate relevant input stimuli, which is sent to the device being verified. The testbench will also fetch the output from the device, to check for its correctness, as illustrated in Figure 2.2. The device being verified is often called *device under test* (DUT)

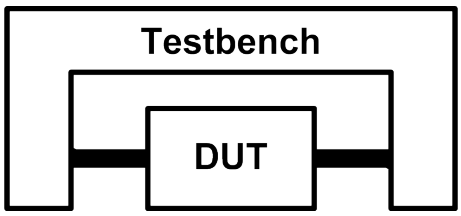

Figure 2.2: Testbench and DUT

Seen from the device under test, the testbench will act like a simulation of the outside world. It is usually impossible to test all combinations of indata, this could require years of simulation time. It is therefore important that the testbench stimuli, and also the sequence of the stimuli, reflects the randomness a normal user would provide. There will be more about random stimuli generation in Section 2.3.

Simulation can show only the presence of bugs, never prove their absence [1]. There is, for designs of normal sizes, never time enough to simulate all combinations and sequences of stimuli and outside influence. There will always be a possibility for undiscovered bugs in the design. Section 2.2 is about coverage, which is a way to measure the completeness of a simulation, and therefore predict the possibility of remaining bugs.

Formal Verification

Formal verification is a way to mathematically check that a design is functionally correct. This is done using special tools acting on assertions, which is a way to specify the properties of the design.

Formal verification on large designs is an operation with very high complexity, and therefore not used on full designs. Formal verification is more commonly used to verify corner cases of the design, which is not easily reached by simulation.

Some tools combine the power of simulation based and formal verification. This combination is often called hybrid verification.

2.2 Coverage

Coverage is a way to measure how thoroughly the design has been verified during simulations. This is important, since there is never infinite time available for the verification process. We need to know when we could be confident enough that the design is bug-free, and ready for the market. There are two main types of coverage: Code coverage and functional coverage.

2.2.1 Code Coverage

Code coverage answers the following question: How much of the design *implementation* has been executed during verification?

Code coverage is collected automatically by the simulation tools. Even though collecting code coverage does not represent much extra work for the verification engineer, it might be a good indicator to see if enough simulations have been carried out. In addition, code coverage analysis can often reveal redundant code in the DUT.

There are many different types of code coverage metrics. These are the main ones, listed in [4]:

- Line Coverage tells which lines, statements and blocks that have been executed during the simulation.
- Condition Coverage shows which combinations of inputs to conditional expressions that are covered.
- FSM Coverage reports about which states, state transitions and sequences of state transitions that are covered in the finite state machines of the design.
- Branch Coverage shows which parts of each single if and case expressions that have been covered.
- Path Coverage shows which sequences of if and case selections that have been covered.
- Toggle Coverage monitors the value change from 0 to 1 and 1 to 0 for every signal, and reports which combinations that have been covered.

Often, the goal for each metric is set to 100% *explained coverage*. The non-covered parts of the code will then have to be explained why it is not covered.

2.2.2 Functional Coverage

Code coverage, as shown in the previous chapter, is a good way to find out how thoroughly the implementation has been tested. But what if some planned features are simply missing in the implemented design? This is where functional coverage can help.

Functional coverage answers the following question: How much of the design *specification* has been verified?

In contrast to code coverage analysis, functional coverage analysis is associated with quite a lot of manual work. Since functional coverage measures the completeness of the simulation compared to the specification, the specification has to be somehow entered into the system. This is done manually, using cover groups or cover properties.

The most flexible of the two is the cover group. It consists of one or more cover points, which are specified to sample data at specified signals or expressions in the design. Expected data are grouped in ranges called *bins*. When data is sampled, the corresponding bin is marked as covered. Coverage analysis will reveal which bins that are covered, and which ones are not.

Two cover points could be combined to form a matrix-style data bin. This is called cross-coverage.

The second, simpler, mechanism for functional coverage collection is cover properties. Cover properties look much like assertions, described in later chapters. Analysis of cover properties will reveal how many times the property has been hit.

2.2.3 When Are We Done?

The main purpose of using coverage analysis is to get a measure about how thoroughly the design has been tested, and to give an indication on whether the verification process is complete or not.

Figure 2.3 shows a table about how to react on the coverage percentages. If the functional coverage is high, but the code coverage is low, it might be necessary to include more functional cover points. If the code coverage is high while the functional coverage is low, it might be worthwhile to check out the design code, to see if the specification is fully implemented.

The goal is high code coverage and high functional coverage. If this is achieved, the bug rate could be checked. Figure 2.4 shows an example of a bug rate graph for

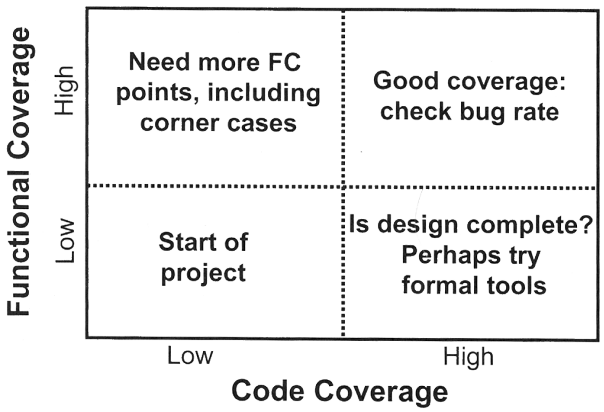

Figure 2.3: Analyzing functional and code coverage results [2]

a verification process. The graph, showing bugs found per week during the project, could be an indicator on how much work there is left.

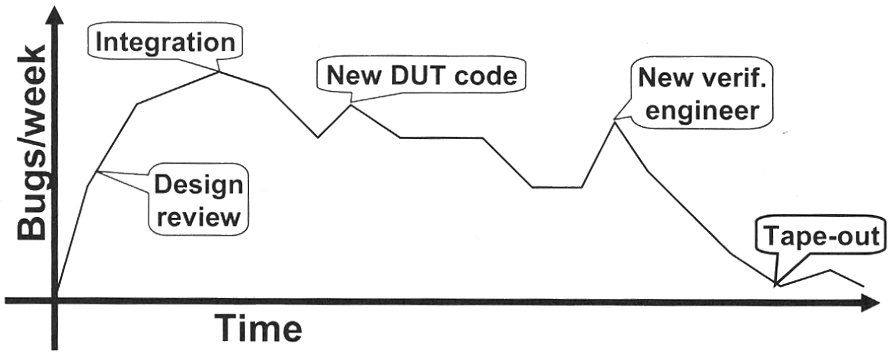

Figure 2.4: Bug rate graph [2]

2.3 Randomization

If the design to be verified is small, with only a few inputs (and input data combinations), a classic, *directed* approach to the verification process might be adequate. Each set of input data will then carefully be written manually, in order to achieve the wanted coverage.

With today's complex designs, however, writing directed testcases will usually take way too much time. A better way to generate test inputs and scenarios is to

use pseudo-random functions. Instead of spending a lot of time of aiming one bug at the time with a directed test, a set of random tests will hit a larger area at the same time, with automatically generated stimuli. Peet James [5] compares this with using a shotgun to hit bugs, instead of using a peashooter.

The graph of covered design space versus time will be a bit different between the two approaches, as shown in figure 2.5. Using directed tests the graph will be quite linear, since each testcase will be written manually. Using random testing, the graph will first be flat, representing the time spent on creating the random testbench. The graph will then typically rise faster than the directed graph, and then (hopefully) reach the coverage goal faster than the directed tests.

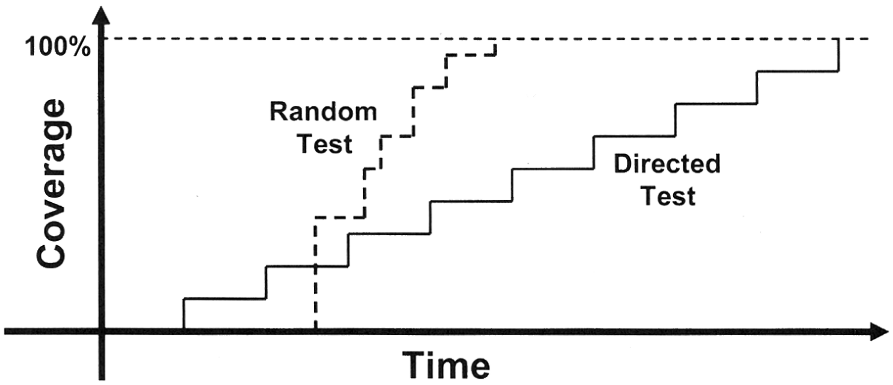

Figure 2.5: Coverage versus time [2]

2.3.1 What to Randomize

The obvious thought about randomization is to randomize the input data. In this way the need for manually created stimuli is eliminated. However, randomization is much more powerful than that.

The purpose of a testbench is to simulate the outside world as good as possible. It should test the device to its limits, to create special situations often discovering bugs. It should act random in more ways than just generation random input data, just like a normal user would do. [2] gives some examples on what could be randomized:

- Device configuration
- Environment configuration
- Primary input data

- Encapsulated input data
- Protocol exceptions
- Communication delays: Delay responses with a number of clock cycles
- Transaction status
- Errors and violations

Scenarios are often planned in order to implement the more advanced random features. Each valid operation of the design is then defined as a scenario in the testbench. This could for instance be a memory read operation. The scenarios are then randomly selected by the testbench during the simulation.

2.3.2 Constraints

Randomization without constraints, like described previously, would be like shooting in the dark. By specifying constraints for the random stimuli generator, you could "aim the gun" at the design parts with high probability of bugs. And, more importantly, try to reach the parts of the design which are not already covered.

One type of constraints is the specification of the data format. For instance the width of a bus address, or valid configuration bits.

Constraints could be used to create a higher probability for interesting data. Often data in the upper and lower extremes of the allowed range could provoke overflow errors. To achieve this, a so-called bathtub distribution of the data could be made, as shown in Figure 2.6.

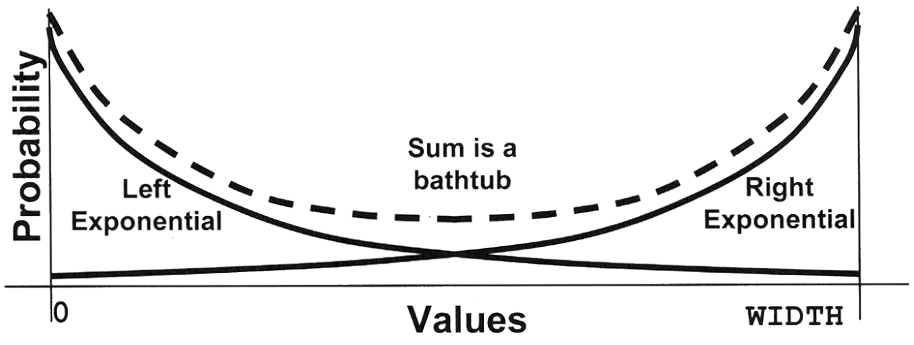

Figure 2.6: Bathtub distribution [2]

Using a feedback mechanism like the one in Figure 2.7 it is (at least theoretically) possible to automatically seed the random generator, so that areas not already covered are targeted. I practice, this is difficult and yet not very common to do with today's verification tools [6].

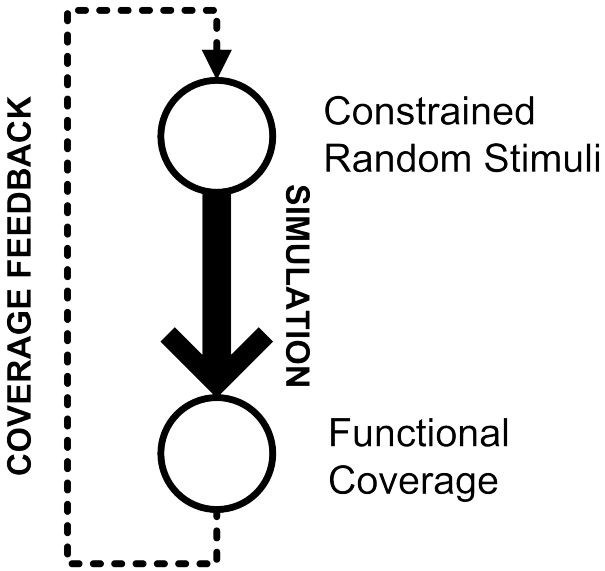

Figure 2.7: Coverage feedback

2.3.3 Seeds

Given the exact same testbench, and the same simulator software, the pseudo-random number generator will always generate the same numbers. It is possible to seed the number generator to achieve different random number series. This could be interesting to be able to run many short series of simulations, instead of one long series.

2.4 Layers and Reuse

If you are verifying a very small design, almost any testbench architecture could be adequate. However, as DUTs and verification systems grow more complex, it is important to have more structured testbenches. The code should be easy maintainable even when the project grows bigger. Code will also often have to be reused in future verification projects.

To achieve this the testbench has to be split up in layers and components. With smaller units it is easier to achieve goals of maintainability and reusability.

2.4.1 Layered Testbench Architecture

Using layers is a way to split the testbench in different levels of abstraction. The lowest level of abstraction will typically be the functionality communicating with the DUT. The upper layers deal with tasks of higher abstraction, like stimuli generation.

Figure 2.8 shows an example of a layered testbench, based on figures and examples published in [2]. This must be regarded as an example only. For instance, DUTs with more complex communication might require more layers. Small devices could maybe do with even fewer layers than four.

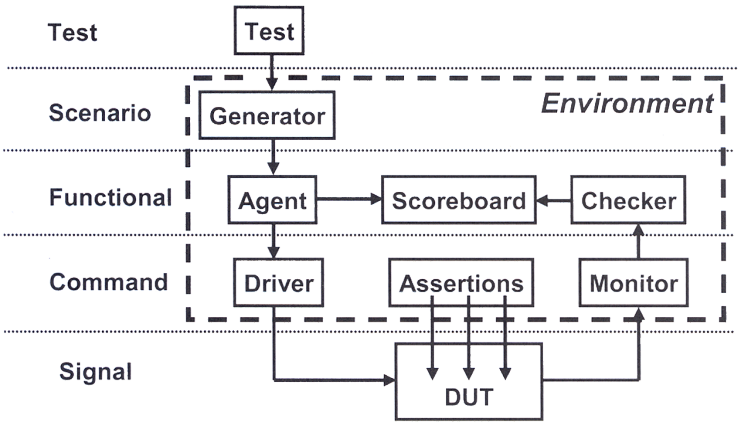

Figure 2.8: Layered testbench example [2]

The components of the layers are implemented as classes. Features already well-known from object oriented programming languages are therefore available, like inheritance, instantiation and threads. This opens for a more elegant way of handling data, as well as greater possibilities for component reuse.

Test Layer

The test layer consists of the test component. Test is the top-level component. It contains the constraints for the stimuli. It should be easy to change, so the simulation easily could be fine-tuned between each run.

Scenario Layer

The scenario layer contains the generator component. Generator generates the stimuli on behalf of the constraints from the test component. It also contains the scenarios. All randomization tasks are carried out in the generator. Generator also contains the scenarios. The randomization mechanisms are discussed in detail in Section 2.3.

Functional Layer

In the functional layer data could be further processed and broken up before it is sent to the command layer. This layer also contains the tasks related to checking the correctness of the DUT outputs. It contains the agent, scoreboard and checker components.

- Agent could be used to further process the data before it is sent to the driver.
- Scoreboard contains a high-level reference model of the design. It receives the same stimuli as the DUT, and predicts the correct DUT response.
- Checker compares the DUT output received from the monitor with the predictions received from the scoreboard.

Command Layer

The command layer is where the direct communication with the DUT is maintained. Here the signals of the DUT are driven and monitored.

- Driver is the component which drives the inputs of the DUT, typically using bus commands.
- Monitor is the second component communicating with the DUT. It monitors the outputs of the DUT, and sends the received outdata to the checker.
- As seen in the figure, external assertions could be though of as a part of the command layer.

Signal Layer

The signal layer simply contains the device which is being verified, and its signals.

2.4.2 Callbacks

Often it would be useful to be able to do some randomization tasks also in low-level layers like the Driver. An example could be to simulate disturbances in the DUT communication, like delaying a response signal, or dropping and retransmitting a data packet.

However, specifying details about the randomization inside the Driver is not a good idea. Randomization settings should be easy to change later. Therefore, callback functions are often defined. They are defined in their own class, and can therefore be easily changed later using inheritance. Driver should always call callback functions right before and right after data transmission.

2.4.3 Verification IPs

For verification of standard components, like buses, it could often be more worthwhile to buy already finished code than writing it oneself. Verification IPs are IPs with verification functionality.

2.5 Assertions

Assertions are small code blocks inserted in the design, containing statements about the expected behavior of the design code. In its simplest form, an assertion can be viewed as an *if* statement that does some error action if the DUT fails to fullfill its requirements. Assertion statements are quite common in software design, but has only become available in hardware design tools recent years.

Assertions could be specified by both the designer and the verifier. Who does what should be decided in the verification planning process.

Assertions could either be coded from scratch, or picked from a library. Synopsys VCS includes a library of SystemVerilog assertions [7]. In this way, a lot of common functionality can be easily checked using assertions already made.

In addition to monitoring design properties during simulation, assertions can also be used with formal verification tools. The properties specified in the assertions are then used as the basis for the mathematical proofs.

A good, practically oriented guide to SystemVerilog Assertions is written by Vijayaraghavan and Ramanathan [8].

2.5.1 Assertion Types

In SystemVerilog, there are two ways to implement assertions: Immediate and concurrent assertions.

Immediate assertions are, as the name suggests, executed immediately along with the code. They are placed in procedural blocks.

Concurrent assertions are clock-based. A concurrent assertion will monitor a signal or an expression on each clock edge through the simulation, and fire if the expression fails.

2.5.2 Assertion Placement

Assertions could either be placed directly in the code of the DUT, or in the testbench monitoring the bus. Assertions in the DUT are called internal assertions, while as-

sertions placed in the testbench are called external assertions.

External assertions often monitor the bus signals between the testbench and the DUT. If an interface is used, a module containing assertions could easily be connected to the interface like any other module (interfaces are introduced in Section 3.2.2). If not, the assertions could be bound to the signals using a *bind* statement.

Internal assertions could either be written by the designer during the design phase, or by the verifier during the verification process. Special internal conditions and assumptions should be stated in assertions created by the designer. In this way they are not accidentally left out from the verification process. The VMM (Verification Methodology Manual) [6] even suggest designers to replace the ordinary code comments with assertions. This might probably be a bit too dramatic to most designers, but at least well illustrates the idea of internal assertions.

Assertions are left out during synthesis, and therefore does not affect the finished product after synthesis.

2.5.3 Error Reporting

By default assertions print an error message when they fail. However, the error action can be customized.

This gives a possibility to create an error report mechanism. If an assertion fails, it could trigger a mechanism that reports the relevant states and data for the design. Previous data and states should also be included in the report.

With this kind of reports it would be easier to regenerate the situation where the error was found, and to find out what caused the error to happen in the first time.

2.6 Verification Planning

Since verification projects are growing more and more complex, it is also becoming more and more important to create a good plan before the verification is started. Creating a good plan will save a lot of time later. It is also essential to have a plan in order to know when the verification is finished and the coverage results are considered good enough.

The theory about verification planning is based on Peet James' book Verification Plans [5], which proposes a five-day approach to the verification planning process. However, some adjustments are made to better fit SystemVerilog testbenches. Theory about verification planning for SystemVerilog is to some degree found in [2], [1] and [9]

2.6.1 The Verification Team

A verification team should be assembled to do the verification planning. In addition to the verifiers, the team could consist of other people that has relevant input. First of all, this should be the architects and designers, who have first-hand knowledge of the functionality of the design.

The verification planning should start as early as possible. If possible, it should start before the design phase of the design core starts. In this way, the parts of the core could be verified before the entire design is ready. This is shown in Figure 2.9.

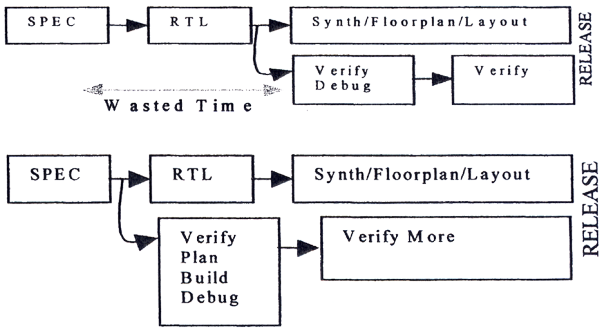

Figure 2.9: Wasting time vs starting early [5]

2.6.2 Day-in-the-Life Document

To be able to create a verification plan for a design it is important to know its features. A day-in-the-life document (DITL) is a short document demonstrating the most common tasks of the design. It is short and consise, typically a few A4 pages. This document can by no means exclude the needs for more detailed specifications of the DUT, but it can give a good overview of the design.

By starting the verification plan process by making such a document, the verification team will get an overview of the tasks of the DUT. This makes it easier to make a good and relevant verification plan.

2.6.3 Tools and Technologies

One of the things to be stated in the verification plan is the choice of verification tools, verification languages and verification technologies. This is typically done gradually in the verification planning process, as the verification needs for the design are revealed.

There is a wide variety of languages and tools available. There is more about some of these in Chapter 3. Often many of the languages and tools could be adequate for the process. Knowing which languages that are mastered by verification team is of course also important to make the right decision.

The verification plan should also contain information about the verification technologies to be used. If some areas of the design are verified using formal tools, there should be information about this in the verification plan.

2.6.4 Architecture

An important thing to define in the verification plan is the architecture of the testbench. The verification plan examples given in [5] starts off with a short list of typical operations of the testbench.

Figures showing the blocks and the data flow of the testbench should be added to describe the architecture. In addition, the layers and the components of the testbench should be described in greater detail in text or a table.

Planned scenarios should be listed and described.

A plan for the randomization features should be made. If there are any planned constraints, they should also be described.

For complex verification systems, it could be smart to also work out more detailed implementation plans for the testbench. In [5], these are called breakout documents and appear in the appendix of the verification plan.

2.6.5 Assertions

The assertion plan is a natural part of the verification plan. It could well be specified in a table form, such as the examples in [9]. As a minimum, the following information should be stated:

- Which functionality to verify
- Where the assertion is placed (external or internal assertion)
- Assertion type (immediate or concurrent assertion)
- Who implements it (designer or verifier, or name of the responsible if the verification team is large)

Who implements which assertions? The following rule of thumb is given in [9]: *Designer engineers should write assertions to verify assumptions that affect the functionality of a design block.* That could be, for instance, that an internal module

never gets X or Z values as an input. *Verification engineers should write assertions that verify design functionality meets the design specification.* For example, that bus behavior is correct.

2.6.6 Implementation Phases

Phases define the milestones in the implementation process. They are the steps in the implementation of the testbench. Each phase will build on some of the previous phases.

If the design to be verified is not yet finished, phases should be planned even more carefully. It is important that the implementation of the verification system closely follows the implementation of the design. In that way parts of the design could be verified before the full design is complete, and the time spent on verification after the design phase has ended will be shorter.

Layers and phases could be crossed in an XY grid to show which layers that needs to be implemented in which phases.

2.6.7 Coverage Collection and Goals

A section should list which code coverage types that will be analyzed, and the approximate goals for each of them.

The functional coverage points should be planned, and listed. This list should at least cover:

- Which signal or expression to cover
- Purpose
- How many bins there should be, and how the bins are specified
- The placement of the cover group (internal or external)
- The coverage goal for the cover group (expressed as a percentage)

Chapter 3

Tools and Languages

THE BACKGROUND THEORY about verification tools and verification languages is collected in this chapter. SystemVerilog and Synopsys VCS are treated in particular, since they will be used for implementation and simulation in later chapters.

3.1 HVLs and HDLs

Traditionally, verification has been done using either a hardware descriptions language (HDL) or a hardware verification language (HVL) . As long as simple, directed testbenches are sufficient to verify a design, the easiest would be to use a HDL. The same language and tools as during the design could be used, which the designer already is familiar to.

With more complex digital designs in the 1990s, there was need for specialized verification languages. *Vera* and *e* were two of them. More advanced features like contrained randomization and object-orientation were introduced. The disadvantage of the HDL is that they represent another language that has to be learnt. Vera and e are both widely used today. However, in a survey in connection with DVCon 2005, 78 percent of the verification engineers thought special verification languages like e and Vera would disappear in the coming years [10].

Even if better solutions are introduced, it does not necessarily mean that they will be adopted by the industry. Most companies already have verification code for most of the functionality of their designs. This legacy code is often also reused in new verification projects. A change of verification language would mean a lot of work, since the legacy code will have to be translated and adjusted to fit new simulators.

SystemVerilog, which is described in the next section, is an HVL with most of the advantages of the HDLs.

3.2 SystemVerilog

3.2.1 Introduction

SystemVerilog has sometimes been called an HDVL, since it combines the strengths of HDLs and HVLs. SystemVerilog is based on the widely used Verilog HDL, but has new functionality for verification and high-level system design. This makes it both powerful and easy to learn for designers already familiar with Verilog. Using the same language for both design and verification also makes it easier to access the internals of the DUT, no special interfaces are needed.

The verification features of SystemVerilog include:

- Assertion-based verification
- Random constrained stimuli generation
- Functional coverage
- Advanced object-orientation

The creation of the SystemVerilog language was initiated in the late 1990s by Accellera, a consortium of EDA companies and users who wanted to create the next generation of Verilog [2]. The OpenVera language was the basis for the verification functionality of the new language. SystemVerilog became an IEEE standard in 2005.

The Verification Methodology Manual (VMM) [6] is a manual written by Synopsys and ARM to give guidelines for advanced verification in using SystemVerilog. As well as being a reference for the SystemVerilog language, it also states rules about important verification planning topics, like layering, randomization, coverage and assertion usage.

3.2.2 Language Properties

In this section, some important language properties of SystemVerilog are collected. This is based on the book by Spear [2], which has good examples of practical use of the SystemVerilog language.

Data types

In addition to the 4-state types available in Verilog, SystemVerilog offers 2-states data types. Since 2-states only considers 0 and 1 values, not X and Z, they are faster in simulation.

Two of the 2-state types are the *bit* and *int* types. Bit is a 2-state single bit type. Like regs, it can be given a range to support multiple bits. Int is a 32-bit

signed integer which works much like in C. Fixed-size int arrays are available, which is convenient for indexes and loop operations.

Strings, queues and associative arrays are also available in SystemVerilog.

Modules and Programs

In SystemVerilog, the active part of an testbench is usually contained in a *program*. Programs could be compared to *modules*. The difference is that *always* blocks are not allowed in programs.

The *automatic* keyword is often added to *program* to make the storage of variable values more like programming languages than Verilog. Local variables are then stored in the stack instead of being shared between all processes.

Signal Interface

An interface is a collection of signal definitions. Modules will then connect to the interface instead of directly to each other. In this way signals definitions are collected in one place, and does not need to be defined in each module. Signal directions could be specified by creating one *modport* for each module.

Functions and Tasks

Tasks can consume time, while functions cannot. Functions must have a return type. In SystemVerilog, task inputs and outputs can be declares in parenthesis, much like in C.

Object Orientation

Object orientation syntax in SystemVerilog looks very similar to the syntax of C++. A special pair of keywords, *fork* and *join*, represents the threads functionality of SystemVerilog. All statements between fork and join will be executed at the same time, as in parallel.

The class-implemented testbench components like Scoreboard and Generator are often called transactors.

External Functions

SystemVerilog (using Synopsys VCS) has at least three different technologies to call external functions placed in C code. DPI is the new technology for this purpose in SystemVerilog. PLI is the corresponding functionality inherited from Verilog. The last one, DirectC, is a VCS-specific technology. A user guide for DirectC [11] is shipped with VCS.

Mailbox

Using mailboxes is a flexible way to enable communication between two transactors. A mailbox could be compared with a FIFO queue. The mailbox could contain any number of objects.

3.3 Synopsys VCS

Synopsys VCS is a verification software package which supports, among other languages, Verilog and SystemVerilog. It is the most common software solution for SystemVerilog verification. In addition to SystemVerilog, it also supports the OpenVera verification language. In its MX version it is also capable of verifying VHDL designs.

Details about VCS could be found in the VCS/VCSi User Guide [12]. There are still parts of SystemVerilog not implemented in VCS, although most of the important features are supported. VCS contains several tools to report simulation results, described below.

Unified Report Generator (URG) is a new, easy-to-use report generator in VCS that reports statistics for both functional coverage, code coverage and assertions coverage.

cmView is a tool to report code coverage. It could be run in batch mode outputting text-based reports, or in graphical mode showing the results interactively. cmView is started using the following commands: *vcs -cm_pp gui* for GUI mode, and *vcs -cm_pp -b* for batch mode. Batch mode creates a number of text files in the *simv.cm/reports/* folder. For each code coverage type, 4 reports are generated. The different report types are shown in Table 3.1 . In addition, definition reports are created, denoted by an added *d* in the filename. They show the cumulative code coverage for all module instances. Examples of the text-based reports are found in appendix D, while a screenshot of the graphical interface is shown in Appendix E.1.

Discovery Visualization Environment (DVE) is the signal-level debugger in VCS. A screenshot of DVE is shown in Appendix E.1.

assertCovReport and *fCovReport* are VCS commands to generate reports for assertions and functional cover properties. They add a set of HTML based reports to the *simv.vdb/reports/* folder. The reports provide statistics for each assertion: The number of assertion hits, the number of statement success, failures and incomplete attempts. The reports also contain summaries grouped on modules and failure severity.

Report	**Explanation**
long	Main reports showing both covered and non-covered code parts. Includes summaries.
short	Short reports only showing non-covered code parts and summaries.
hier	Sums up coverage for module instances, also taking into account sub-hierarchy.
mod	Sums up coverage for module instances, without regard to any sub-hierarchy under the module.

Table 3.1: cmView code coverage reports

Chapter 4

Verification Plan

The Verification Plan is the plan for the entire verification process. This chapter steps through the creation of the verification plan for an AES module with background in the theory in Section 2.6. The full verification plan is given in appendix B, while the DITL document is available in appendix A.

In an ordinary verification process many different persons would be involved, as mentioned in Section 2.6. In this thesis work, there was no such team available for verification planning. Making a DITL took more time than it would with the designer available for inputs, since all details now had to be found in the literature and source code.

It is also worth noting that the DUT was finished at the time the verification planning process started. This makes the verification flow more direct, instead of verifying component-by-component.

4.1 Module Selection

One of the first tasks of the thesis work was to find a module to be verified. The verification tools and techniques were stated in the problem description. In the presentation of the thesis problem it was suggested that the most relevant place to find such a module would be OpenCores [13].

OpenCores is a collection of free digital hardware cores on the Internet. They are often published under the GNU general public license, and usually written in Verilog or VHDL. OpenCores is often thought of as a hardware parallel to the much larger community of open source software.

Some requirements were discussed before the search for a module started:

- It should be finished and have code of high quality.

- It should be a design that fits well as an example for the verification techniques.
- The size and complexity should be kept in manageable bounds.
- It should be written in Verilog, so all SystemVerilog features can be used.
- It should be well documented.

Among of the best alternatives were a few microcontroller designs, two USB cores, and some encryption cores. A core implementing the *Advanced Encryption Standard* (AES) was chosen. The core, called 128 AES [14], is written by Javier Castillo in 2000.

The core fullfills most of the requirements: it was finished, and it looked like its code was of high quality. The design fits well as an example, containing both arithmetic modules and FSMs. Even though the operation is quite complex, the design has few inputs and few configuration choices. The core is written in Verilog.

No documentation follows the module, and no code comments are written, which is seems to be quite typical for all cores available at OpenCores. However, the AES specification is good, and could in many cases act as the documentation for the selected core. In addition, the communication with the core is done using Wishbone, a documented bus standard.

4.1.1 Advanced Encryption Standard

AES is an encryption and decryption algorithm. The AES title was given after a selection process initiated in 1997 by the US National Institute of Standards and Technology (NIST), in order to find replacement for the Data Encryption Standard (DES) and triple-DES. The winner of the AES selection process was a candidate called Rijndael, developed by Belgian cryptographers Joan Daemen and Vincent Rijmen. Rijndael strongest sides in the competition was good performance in both hardware and software, and low memory requirements.

Rijndael operates on data blocks of 128 bits, while the keys are of sizes 128, 192 or 256 bits. The chosen AES core uses 128 bit keys, which is the least secure, but fastest option.

The operation of the core was further explored in the creation of the DITL document in Section 4.2. The DITL document is placed in Appendix A. Used literature on AES includes an AES manual by Dr. B. Gladman [15], the AES specification [3] and a book written by the AES developers [16].

4.1.2 Wishbone Bus

Wishbone is a public domain System-on-Chip bus. The public domain status is one of Wishbones strongest sides, as it can be used completely royalty-free. The Wishbone Bus project is maintained by OpenCores. It was originally developed by Silicore Corporation.

The Wishbone bus supports communication using one master and either one or multiple slaves. Data transfer is done using a handshake behavior when data is interchanged on the data signals. Wishbone supports data blocks in any size up to 64 bits. The width of the bus has to be defined in the specification documents. The core to be verified supports only 32 bit blocks.

The simplest method of Wishbone communication are Single Read and Single Write cycles, which transfers one block at the time. Every block transfer will then require repetition of the handshake process. A faster and more convenient way to transfer data could be implemented using the Block Read and Block Write cycles. Here, the handshake process initiates a series of data transfers, with a data block transferred on each clock cycle.

The Wishbone Specification [17] states a set of requirements that has to be fulfilled before the design can be called "Wishbone compliant". One of the requirements is to provide a document specifying the Wishbone properties (like data block size and error behavior). The distribution of the AES core does not contain such a document, so the source code had to be inspected to reveal the Wishbone properties.

The Wishbone Bus properties of the AES module was further examined during the DITL creation process. This process is documented in Section 4.2.

4.2 Day-in-the-Life Document

The finished Day-in-the Life document can be found in Appendix A. The work with the DITL was started by collecting general information about AES and Wishbone. Since the module is not documented, the source code also had to be examined to find out the detailed design properties of the design.

A short introduction to the module was written, based on the information given on the OpenCores hosted web site of the project [14] and the AES specification [3].

The source code of the project was examined to find the module structure, given in Section A.2. The structure does not differ much from the examples given in the specification. The *SubBytes* and *ShiftRows* implementations are collected in one module, while the *AddRoundKey* and *MixColumns* operations are placed in their

own modules. The output and input signals in the implementation are also close to the examples.

A section about the Wishbone bus properties of the design was written in Section A.3. By examining the source code, it was clear that the module supports only single read and single write operations, since the ACK signal is automatically lowered after each data fetch. No connection to the AES module is provided, so a top module will have to be written in order to simulate the full design.

The supported bus signals were found in the source code and documented in Figure A.2 and Table A.1. The signals implemented are an absolute minimum of signals to provide Wishbone compliant communication. The valid addresses of the wb_adr_i signal were also found and documented in Table A.2.

A AES flow diagram was created to show the internal flow of the AES module. It was based on the pseudo code found in the AES specification [3]. The flow diagram is shown in Figure A.3. In 128 bit AES encryption, 10 rounds of the loop is used to encrypt a message. The AES specification also formed the basis for the figures and descriptions of each step, placed in Section A.5.

4.3 Tools and Technologies

The verification language, SystemVerilog, was already given in the text. Synopsys VCS was recommended by the supervisors to be a good choice of tool for SystemVerilog simulations. Synopsys VCS in version 2005.06-SP2 was available at the university.

The module is verified with only simulation-based verification, no formal verification is used. The focus of the thesis was on simulation based verification from the start, and using formal verification in addition to this would require much more time than allowed for study, planning and implementation.

However, the AES module does contain a lot of mathematical operations, so formal verification could probably have been an effective way to verify some parts of the design.

4.4 Architecture

A simple, typical operation of the testbench was outlined in the first section of the verification plan. The chosen testbench flow is a very common one: Stimuli is generated at the top, and then applied to both the DUT and a reference model. The

results of the two are checked for equality.

An alternative way of checking this particular model could be to first encrypt the data, and then decrypt the output from the encryption using the same key. This approach was not chosen, since the more general testbench using a reference model acts better as an example of a verification system.

Layers are planned in Section B.4 of the verification plan. The layer division of the testbench follows the theory in [2], and could be seen in Figure B.1. The planned components of the testbench are a bit different compared to the theory.

The Driver and Monitor components are combined in a component called *bus functional model* (BFM. The two Wishbone operations from Driver and Monitor are more efficiently modelled in one component, since common functionality can be shared between the two operations. In addition, the two operations will have to communicate directly, since the reading of the output has to be performed a fixed number of clock cycles after the writing of the input. This is more easily modelled with one component (otherwise, a handshake signal would be necessary to allow direct communication).

Agent is removed, since there is no need for further processing of the input data in this case. The data is sent directly from the Generator to the BFM.

A plan of which classes that shall implement each other is provided in the same chapter. This is done much in the same way as in the examples in [2].

A figure showing the data float of the testbench has beed included as Figure B.2 in Appendix B. The data floating though the system is planned to be contained in objects. In this way, the data that belongs together keeps together through the system. A short description of the operation of each of the testbench components is given in Section B.7, as well as a description of the inputs and outputs.

The scenarios are planned in chapter B.6. The AES model does not have many different operations, so the scenarios are few. The three scenarios planned are encryption, decryption and system reset.

The last part of the architecture planning is the planning of the randomization features. The most obvious random feature of the testbench is to randomize the key and data inputs. A common constraint was planned for two inputs: A bathtub data distribution. The probability of values in the two extremes of the scale will then be higher, and more interesting values will be generated.

Scenarios will also be randomized. The testbench will use random functions to select between the three planned scenarios. The planned probability for each sce-

nario is listed in the scenarios list in Section B.6.

To make sure the handshake functions in the Wishbone bus works as good as proposed, a random delay is planned in the BFM. Each delay should be between 0 and 10 clock cycles.

4.5 Assertions

7 assertions were planned, and listed in Table B.10. In a real verification project it would be necessary to check a lot more details using assertions. To be able to complete the verification process in reasonable time, only a much smaller number of assertions were planned. However, since the planned assertions have different types and placements, they are good examples of the assertion planning (and implementation) tasks of a real-world verification project.

The assertion listing table (Table B.10) is inspired by [9]. For a larger verification project it could be better to create one table for each DUT block, like in [9].

4.6 Implementation Phases

As described in Section 2.6.6, phases are the milestones of the testbench implementation process. Seven phases were planned. Since the DUT was a complete design when the verification planning started, there was no need for the phases to follow the steps in a design process. The phases were therefore quite straight forward planned, heading straight for the full testbench architecture.

The first phase is a bit special, since no other phases will use the functionality implemented in this phase. Its purpose is to make sure, at an early stage, that the chosen AES module actually communicates and could be simulated. The last six phases follows a natural flow of testbench implementation:

- Communication using Wishbone interface: The implementation of the BFM, which is the building block for all communication with the module
- Output checking using reference model: Implement the reference model (off the shelf model written in C) to run encryption in parallel with the DUT
- Assertions for checking properties: Implement the planned assertions in the DUT and on the bus interface
- Constrained random stimuli: Implement the random features of the testbench

- Coverage calculation: Implement functional cover groups, and enable the system for coverage report creation
- Verification of decryption functionality: Add features to enable verification of decryption functionality (most of the features in this phase are probably already implemented in previous phases)

The phases are listed in Table B.3. This list is also combined with the layers planned earlier to form a layer implementation list, as shown in Table B.4. This gives an overview of when (in which phase) each of the layers will need to be implemented. This is also shown graphically in the XY-grid of phases and layers in Table B.5.

With this phase plan, the system is built almost entirely from bottom to top with regard to the layers, which is natural as long as the DUT design is finished. If there were parts of the DUT not already finished, the flow might have been different.

4.7 Coverage Collection and Goals

First, code coverage was planned. A list of the code coverages to be analyzed was created, and placed in table B.11 in Appendix B. The goals are all set to 100%, meaning 100% explained coverage. All design space not covered will therefore have to be explained (e.g. impossible states).

The collection of functional coverage is planned in Table B.12. The first four are straight-forward cover groups to check data distribution of different signals. The two external ones will be checking the distribution of the Wishbone data signals. The two internal ones will check the distribution of the AES input signals. In this way, it can be verified that the bathtub function, as planned in earlier sections, works.

The last planned cover group will be checking the coverage of the S-Box. It will use cross coverage to check that all possible combinations of the two inputs are covered.

All function coverage collection was planned to be done using cover groups, not cover properties. The goals for each of them are 100% coverage.

Chapter 5

Testbench Implementation

THIS CHAPTER describes the implementation of the verification system planned in the verification plan. The most important relevant properties of the SystemVerilog language are given in the language theory chapter, Section 3.2.2. Important references for the SystemVerilog language include Spear [2] and the VMM [6].

5.1 AES Module Simulation Test

This section describes the implementation of the phase 1 in the verification plan. Since the module to be verified was open source code, delivered with no guarantees or earlier test results, a small testbench was used to check if the module actually could be simulated with the available software.

This was done using an AES testbench posted on the same OpenCores project as a basis. The module worked well with this test. A few basic functionalities of the Synopsys VCS simulation software were also investigated during this test. A sample waveguide for this communication is shown in Appendix E.

5.2 Testbench Modules

The module structure of the testbench is given in Figure 5.1. The blocks shown in this figure are the modules instantiated by top. Each one of them are implemented in their own file. Table 5.1 shows more details.

5.2.1 Interface

The communication signals between the modules are gathered in an interface object. The reason to use an interface object is to gather all signal declarations in one place,

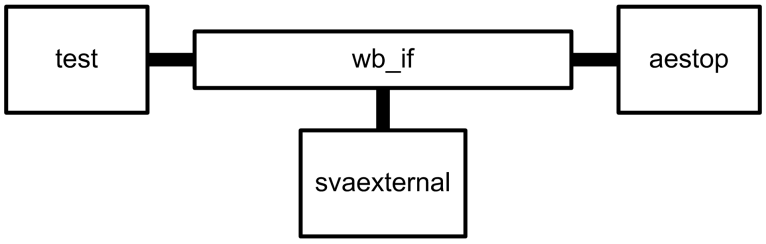

Figure 5.1: Testbench module structure

Module name	File name	Implemented as	Purpose
top	top.sv	Module	Top-level module
test	test.sv	Program	Containing layers and transactors
aestop	aestop.sv	Module	New top module for DUT
svaexternal	svaexternal.sv	Module	Containing external assertions
wb_if	wb_if.sv	Interface	Signal interface

Table 5.1: Testbench modules and files

as mentioned in the theory chapter.

First, the Wishbone signals are defined. The are defined as logic, a 4-state type. Then, three modports are defined, one for each of the modules to connect to the interface. Here the signal directions relative to the connecting module are stated. This is defined using *output* and *input* keywords.

5.2.2 AES Top Module

An AES top module had to be designed, since the Wishbone and AES functionalities in the design were not connected. This is a normal, simple module. Interface and internal signals are defined, and the *aes* and wb_aes_controller modules are instantiated and connected.

The *aes* module has negative reset, while the wb_aes_controller has positive reset. A reset signal sensitive block is written to invert the reset signal for *aes*.

5.2.3 Test Module

The test module implements the layers and components part of the testbench, the ones in Figure B.1 in the verification plan. It is implemented using the *program automatic* keyword. Test instantiates Environment, which is further described in Section 5.3.2.

5.2.4 External Assertion Module

Assertions monitoring the bus, the so-called external assertions, are placed in their own module. This module is called *svaexternal*, and is directly connected to the interface. Assertion implementation is further described in Section 5.4.

5.3 Transactors

This section describes the implementation of the Test program. The test program contains the layers and components shown in Figure B.1 from Appendix B. The components are implemented as classes. They are commonly referred to as transactors in the literature ([2] and [6]).

The VMM [6] has several finished classes for transactors, that could be used as a basis for transactor implementation. In this testbench they are not used, since the system specified will be easy to build using ordinary classes. For a more complex testbench it could probably be more interesting to use more building blocks from the VMM.

Variables in the transactors are implemented as 2-state data types. 2-state types are also used when fetching data from the DUT, mostly because of their convenient array properties. This means that X and Z values from the DUT outputs will not be visible for the transactors, which is a potential problem. However, in this testbench the X and Z value checking is done in the assertion module. It is directly connected to the 4-state types in the interface, and will therefore detect any X or Z values. [18] gives more details about X/Z-value issues in RTL design.

5.3.1 Data Flow

The data flow between the transactors is implemented using mailboxes. Four mailboxes are used in the testbench, to enable the data flow in Figure B.2 in Appendix B. They are shown in Table 5.2.

Mailbox	Communication path
gen2bfm	Generator to BFM
gen2scb	Generator to Scoreboard
bfm2chk	BFM to Checker
scb2chk	Scoreboard to Checker

Table 5.2: Mailboxes

The data sent in the mailboxes are objects of the Transaction class. This class is defined in the test program. The properties of transaction objects are shown in

Table 5.3.

Name	Data type	Width	Description
data_in	bit	128	Input data
key	bit	128	Key data
data_out	bit	128	Output from DUT
data_c_out	bit	128	Output from reference model
decrypt	bit	1	Decryption selector
reset	bit	1	Request reset operation

Table 5.3: Transaction class properties

A transaction is put into a mailbox using the *put* function of the mailbox with the transaction object as a property. An object is fetched using the function called *get*. Get is a blocking function. If get is called on an empty mailbox, the program flow will not proceed before an object is available for fetching.

The DUT operation is a time-consuming task. In the time of one calculation of the DUT, the Generator could generate hundreds of new objects. To prevent this, a handshake mechanism is implemented between the BFM and the Generator. Now the Generator will not produce more transaction objects than the BFM can consume.

A more elegant way of solving this problem would be to restrict the size of the mailboxes. The put function would then be blocking when the mailbox is full. Size restriction of mailboxes is a feature of the SystemVerilog language. Unfortunately it is not implemented in this version of VCS, according to VCS' *SystemVerilog Testbench Constructs* manual [19].

5.3.2 Environment

Environment is the transactor that is instantiated by the test program. Its only purpose is to instantiate and start the other four transactors, as indicated in Figure B.1.

The four other transactors should be started at the same time, run in parallel, and the program flow should not continue until all transactors are finished. This is specified with the fork..join block. The operations between *fork* and *join* will be run in parallel.

5.3.3 Bus Functional Model

The BFM was the first part of the testbench to be implemented, it was implemented in phase 2. The BFM drives the Wishbone bus to communicate with the DUT.

The Wishbone interface of the DUT is not documented, so the source code of the DUT had to be carefully examined to be able to communicate with it. It appeared that the module only supported the simplest form of data transfer, single transfer. The signals supported by the module are presented in Appendix A, Figure A.2.

Four tasks were created in the BFM to make a compact implementation: *writecycle*, *readcycle*, *setsignals* and *sendreset*.

- *sendreset* simply sets the *reset* signal low, high and then low again at following rising clock edges to reset the DUT.
- *setsignals* is the lowest level of Wishbone communication found in the testbench. It sets the signals on the bus. However, before it does so, it calls a callback task to get a random delay.
- *writecycle* is a higher level function to create one Wishbone write cycle. It uses *setsignals* to set the desired signals on the bus. Then it waits for a *wb_ack* from the DUT.
- *readcycle* uses the *setsignals* task to present the needed Wishbone signals to the bus. Data is fetched from the bus when *wb_ack* is set high from the DUT, and put in the right section of the *data_cycle_out* array.

The operation of the BFM is placed in a task called *run*, like in the other transactors. The operation of *run* is the following:

- Get a transaction object from the Generator-BFM mailbox.
- Send a handshake to the Generator to tell it to make a new transaction object.
- Split the key and data objects into eight blocks of 32 bits.
- Compose a block to tell the DUT if this is an encryption (value 3'b001) or decryption (3'b101) operation.
- Check if the transaction object is a reset object. If it is, *sendreset* is called and nothing more except for the last point is done.
- If not, the nine cycles of data are sequentially set on the bus using *writecycle*.
- Wait for 600 clock cycles for the DUT to finish the AES calculations.
- Read the outdata using four *readcycle* operations.
- Add the received DUT outputs to the *data_out* variable of the transaction object.
- Put the transaction object in the BFM-Checker mailbox.

The callback task referred to earlier is defined in its own class, called *BFM_ cbs*. In this way, it could more easily be changed between simulations, as described in Section 2.4.2. The pre-transfer callback task itself is called *pre_ tx*. The only thing it does is to call for a random number using *$urandom_ range*, and then wait the corresponding number of clock cycles. A post-transfer callback is also defined, but not in use.

An example of a Wishbone write cycle with random delays could be found in Appendix E.2.

5.3.4 Generator

As described in the theory chapters, the Generator transactor is responsible for the randomization and data generation tasks. The main functionality of the Generator was implemented in phase 5. However, a simple generator without the randomization functions was implemented in phase 3 to support the scoreboarding operation.

Like in the other transactors, the main operation of Generator is placed in a *run* task. The *run* task first runs a reset task, since the DUT will not operate at all before reset. After this, the task contains a for loop which creates a specified number of stimuli sets. The number is specified in the global constant *ROUNDS*.

In the loop, a *randsequence* call is performed to randomly select between three tasks: *encrypt_ task*, *decrypt_ task* and *reset_ task*. The probabilities of each of the three are specified in the stream expression, here set to be level between the three. The selected task is called for randomization and stimuli generation.

The *encrypt_ task* and *decrypt_ task* have much in common. They both start off by creating a new transaction object. New transaction objects must be created for every simulation round to make the randomize call work. The new transaction object is randomized using the *randomize* function. An assertion is added here to check that the randomize operation does not fail (if it fails, it returns 0, and the assertion is triggered). The bit variable specifying encryption or decryption is set to 0 or 1, depending on the task. At last, the transaction object is sent to the BFM and the Scoreboard using the corresponding mailboxes.

It could be noted that the random selection of encryption/decryption could more easily be implemented by simply randomizing the decryption bit of the transactor object, just like the key and data are randomized. However, it was chosen to implement it using randsequence to show how random scenarios work.

The last task, the *reset_ task*, also creates a new transaction object. But, instead of randomizing, it leaves the stimuli empty and sets the reset bit high. The object is then sent to the BFM and Scoreboard mailboxes. When it arrives in the BFM, it

will tell the BFM to do a system reset instead of the normal AES operation.

In Section 4.4, a bathtub data distribution was planned for the inputs. Unfortunately this was not implemented, due to difficulties with such large data types and limited time.

5.3.5 Scoreboard

Scoreboard functionality was implemented in phase 3. This phase was the one to require the most time spent on testing different technologies and reference models to find a suitable solution.

Reference model selection

First of all, a reference model had to be found. Literature and websites were searched for a suitable model. In [15], Dr. B. Gladman gives a reference model written in the C language. On his website [20], a brand new low-level implementation was found later. This one had quite simple input and output funtions, which made it suitable for use with the testbench. However, no examples on use were given, so some hours had to be spent reading C literature [21], and trying and failing, before a test program communicating with the reference model could be written. In the end, it turned out to be even more simple to communicate with it than expected.

A small test program was written. There were some problems because of unfamiliarity with the use of unsigned char arrays, but this was solved by reading C reference literature [21]. The program was compiled with GCC on Red Hat Linux.

C model communication

The next important step to fullfill phase 3 was to enable communication between the SystemVerilog testbench and the C reference model. SystemVerilog and Verilog (using VCS) has at least three different techologies for C model communication: DPI, PLI and DirectC.

DPI was first tested, since it is the technology tightest bound to SystemVerilog. A DPI communication example was found in the VCS examples directory. VCS was used to try to compile it. But, it turned out that the VCS licence installed did not include the DPI functionality. The system administrator did some checking about the licences available, and it turned out that this functionality was unavailable with the licence of the university.

PLI was then checked out. A book about PLI, [22], was obtained, and some work was done to create a working example. PLI is more complicated than DPI, and this took some time. However, at some point, a VCS specific functionality called DirectC

was discovered. This seemed to be a much more usable solution than PLI, and the PLI investigation was discontinued.

A DirectC user guide [11] was found in the VCS documentation catalog. DirectC was easier to use than the two previous technologies, however, it took some work to create a communication prototype supporting as large variables as 128 bits. The C communication prototype worked well with VCS.

Final Scoreboard implementation

The communication prototype was used as a basis for the scoreboard implementation. A reference to the external function (described below) is created in the top of the test.sv file. In the Scoreboard transactor does the following, repeated for each simulation round:

- Get transaction object from the Generator mailbox
- If the reset bit is not high, send transaction object data to the external function
- Receive the output and add it to the *data_c_out* variable of the transaction object
- Put the transaction object in the Checker mailbox

In the C model, an input and output function specially designed for this testbench was implemented. It receives data as UB type, a char based special type for DirectC. The input variables, containing the key and data inputs, are reversed in order to be compatible with Dr. Gladman's C model. Decrypt or encrypt operation is selected by the decrypt scalar input. After AES operation, an UB array is reversed and passed back to the SystemVerilog testbench.

A detail worth noting is a difference in the decryption behavior between the the c model and the DUT. The DUT takes the original key as input, while the c model needs the pre-processed key for input. This is solved by running an encryption with the same key before the decryption, and using the output key from the encryption as the input key for the decryption.

5.3.6 Checker

The checker transactor is implemented in phase 3 to enable output checking of the parallel DUT and Scoreboard operations. Checker has the following behavior, repeated for each simulation round:

- Receive transaction objects from BFM and Scoreboard. Since the *get* function is blocking, the checker will wait here until the BFM/DUT is finished.

- Check that the output from both AES operations are equal.

The output check is implemented as an immediate assertion. In this way, its results could be analyzed using the normal assertion reports.

5.4 Assertions

This section is about the implementation of the assertions. The assertions are planned in Section B.9 in Appendix B. The assertions are implemented during phase 4.

The external assertions are placed in a separate module connected to the interface. In this way, they are able to monitor the wishbone signals directly. There are three assertions implemented in this module. The fourth external assertion, the one that compares the outputs of the DUT and scoreboard, is a special case and is implemented inside the testbench: In the Checker transactor.

- *ACK is never set high without a preceding STB.* This assertion uses the *$rose* keyword to be evaluated every time the wb_ack signal rises. The indication arrow requires wb_stb to be high when this happened.
- *Wishbone bus never has any X or Z bits when reading is allowed.* This assertion uses *$isunknown* to look for unknown values on the wb_dat_o signal. The expression contains AND operators to express that no bits should be X or Y when wb_we (write enable) and wb_ack is high.
- *Request from testbench results in ACK from DUT within 10 clock cycles.* This is checked using an assertion from the OVL library, *assert_handshake*. A sub-assertion of this library assertion is used to also check that the length in the ACK signal is not more than 3 clock cycles.
- *DUT output value is the same as the scoreboard value at the end of each run.* This one is implemented in the Checker transactor, as an immediate assertion that compares the data_out and data_c_out of the transaction object.

The internal assertions are placed directly in the DUT code. Due to limited time, only one internal assertion was implemented: The one that checks that *number of rounds of AES operation does not exceed 10.* It is placed in the aes.v file, inside the module called *aes*. The code for this assertion is placed in Figure 5.2

5.5 Functional Coverage

Functional Coverage groups were to planned be implemented in phase 6. The work began with the implementation of the first cover group in Table **??**. The cover group

```
// Round count assertion
property p_rounds;
        @(posedge clk) not (round > 11);
endproperty

a_rounds: assert property(p_rounds);
c_rounds: cover property (p_rounds);
```

Figure 5.2: Source code for a_rounds assertion

was implemented inside the BFM transactor. The cover group definition is shown in Figure 5.3. It consists of one cover point, monitoring the Wishbone in data signal (wb_dat_i). The sampling of coverage data is done using the *covWbDatIn.sample()* expression, which is placed in the writecycle task of the BFM. In this way, coverage data is sampled from the Wishbone bus every time something is written to it.

```
// Define cover group for WB indata
covergroup covWbDatIn;
        coverpoint wb_if.wb_dat_i;
endgroup
```

Figure 5.3: Source code for covWbDatIn cover group

As mentioned earlier, the plan was to analyze the functional coverage using URG. However, because of licencing problems, older analysis tools had to be used. None of them could analyze cover groups. Some efforts were done to try to implement the planned functional cover groups as cover properties. Cover properties are much simpler than cover groups, and are normally only used to monitor visited signal, not the values of those. For instance, they don't have any functionality comparable to bins. More details about cover groups and cover properties could be found in Section 2.2.2.

As a test, a cover property based on the assertion in Figure 5.2 was created. It is specified in the last line of the source code in the same figure.

Chapter 6

Simulation

THIS CHAPTER describes the setup of the software tools and the compilation and simulation of the testbench.

6.1 Synopsys VCS Setup

The testbench was compiled, simulated and debugged using Synopsys VCS 2005.06-SP2. The Synopsys VCS software was installed by the system administrator, but some minor adjustments of the settings were made to make simulation easier. The software had been installed at the central Linux system running Red Hat Linux. Communication with this system was carried out using partially a SSH client, and partially an X-Windows client (to enable use of graphical tools).

In order to make VCS work at my Linux account, the environment variables VCS_HOME and PATH i .bash_profile were edited. Some editing was also necessary to connect the system to the right license server.

6.2 Testbench Compilation

To compile the testbench, the *vcs* command was used. Switches were applied to make the settings needed for the operation. The compile time switches are listed and explained in Table 6.1.

The final testbench was compiled using the following command:

```
vcs +define+ASSERT_ON -debug -cm assert+cond+tgl+line+fsm+branch+path  \
-sverilog -PP -assert enable_diag *.v *.sv aes.c -l compile.log \
-y ~/sva-lib +libext+.sv +vc +sysvcs \
+verilog2001ext+.v+incdir+$~/sva-lib
```

Name	Description
-debug	Compile for debug in DVE
-debug_all	Compile for debug in DVE with linestepping features
-cm assert+path	Enable collection of code coverage
-sverilog	Enable SystemVerilog compilation
+define+ASSERT_ON	Assertion calculation
-PP	Enable debugging
-assert enable_diag	Extended SVA reporting
-l compile.log	Save compilation log file
+vc	Enables DirectC
-y	Specify Library Directory
+verilog2001ext	Specify Verilog filename ext.
+incdir	Specify include directory

Table 6.1: Compile-time switches

The compilation results in an executable file to be run to start the simulation. The name of the file could be defined using a switch. The default name of the file is *simv*.

6.3 Simulation

Simulation is started using the executable file that was created during compilation. Like in compilation, switches are applied to change settings. Some of the available switches for the simulation are listed and explained in Table 6.2.

Name	Description
-cm assert+path	Collect code coverage
-assert	Assertion settings
-l run.log	Save simulation log file
+ntb_solver_mode=1	Randomization preprocessing mode

Table 6.2: Runtime switches

The final testbench was simulated using the following command:

```
simv +ntb_solver_mode=1 -cm assert+cond+tgl+line+fsm+branch+path \
 -l run.log -assert filter
```

6.4 Testbench Debug

In order to be sure that the final testbench would be functionally correct, every part of the testbench code was tested and debugged during implementation. This was done using normal simulations and the debugging tool in VCS, DVE. DVE was used to show waveforms for signal debugging. There is more about DVE in Section 3.3. DVE is started by applying the *-gui* switch to the simulation executable. For instance: *simv -gui*.

6.5 Assertion and Coverage Reporting

URG, described in Section 3.3, was the tool that was planned to be used for generating reports: Both code coverage, functional coverage and assertion reports. Unfortunately, URG was not included in the VCS licence held by the university. The system administrator was in contact with the licencing organization to try to add access to URG to the licence, but this was not possible.

Instead, older tools had to be used. To generate and view code coverage, cmView was used. For assertion reports: assertCovReport. For functional coverage properties: fcovReport. The tools are described in Section 3.3. No tool capable of creating reports from the functional coverage groups was available with the licence held by the university.

AssertCovReport and fcovReport were executed by calling the assertCovReport and fcovReport commands, respectively. They placed a HTML based report in the simv.vdb/reports/ folder. The full reports for the final simulations are shown in Appendix D.8, and the results are discussed in Chapter 7.

cmView was used, both in GUI and batch mode, to generate reports about code coverage. cmView in batch mode puts out text-based reports like the reports in appendix D. The reports from the different runs are analyzed and compared in Chapter 7.

Chapter 7

Discussion

This chapter will discuss the coverage results achieved during the simulations. Unfortunately, because of software licence problems, the functional coverage results are very limited. The code coverage results, however, will be presented and discussed. The last section of the chapter contains a discussion on the testbench design and possible future work.

7.1 Coverage Progress

For the final testbench 5 different simulation sessions were carried out. The sessions were done using the same testbench, but with different number of simulation rounds: Respectively 10, 100, 1000, 10 000 and 200 000 encryptions/decryptions, by changing the ROUNDS constant in the testbench between each run. This was done to track the coverage progress. Table 7.1 contains the progress of the most important metrics.

	10	100	1000	10 000	200 000
Line Coverage (Lines)	99.57	99.57	99.57	99.57	99.57
Condition Coverage	87.72	87.72	87.72	87.72	87.72
FSM Coverage (States)	100.00	100.00	100.00	100.00	100.00
Branch Coverage	94.51	94.51	94.51	94.51	94.51
Path Coverage	37.58	37.58	37.58	37.58	37.58
Toggle Coverage (Nets)	84.85	87.88	87.88	87.88	87.88

Table 7.1: Code coverage progress (percentage)

As the table shows, the numbers for the code coverage is already high at very few simulations, already at the 10 simulations case. They do not increase when more simulations are carried out. The reason is probably that a large part of the AES design is invoked at each simulation. The module also has few scenarios to cover, so

they are all covered relatively quick.

Unfortunately, since the functional coverage groups could not be analyzed, there is no progress data available for functional coverage. Probably we would have seen much of the same progress as with the code coverage, at least with the cover groups planned in the verification plan. They have relatively few bins, and are hit by every simulation.

7.2 Code Coverage

The following sections analyzes the code coverage achieved in the run with 200 000 simulations. However, the results are also quite relevant to the shorter runs, since high code coverage was already achieved at a very low number of runs.

7.2.1 Line Coverage

The line coverage summary from the short report is shown in Table 7.2. The line coverage short report is placed in Appendix D.1. Only the following lines of code are reported not covered:

- *Lines 153-154 in wb_aescontroller.v.* Contains an data address not used by the testbench during the Wishbone communication. The functionality not covered returns the operation mode carried out (decryption or encryption). Whether this code is redundant or it should have been covered depends on the (non-existing) specification.

	Total	Covered	Percent
Lines	462	460	99.57
Statements	511	509	99.61
Blocks	122	121	99.18

Table 7.2: Line coverage summary

7.2.2 Condition Coverage

The condition coverage summary is shown in Table 7.3. The condition coverage short report is placed in Appendix D.2. The uncovered conditions were:

- *Line 104 in wb_aescontroller.v: 0-1-1-1 and 1-0-1-1 not covered.* The line contains a statement that returns true if wb_stb_i, wb_cyc_i and wb_we_i are

high while wb_ack_o is low. It indicates a request for data from the testbench. Since Single Read mode is used by the testbench (which is the only read mode supported by the Wishbone slave), wb_stb_i and wb_cyc_i will always have the same value.

- *Line 147 in wb_ aescontroller.v: 0-1-1-1, 1-0-1-1 and 1-1-0-1 not covered.* Almost same expression as above, except for the wb_we_i which now should be low to make the statement return true. The same reason as above applies to the first two non-covered combinations. Since this is an elseif statement of the expression above, the 1-1-0-1 combination will be fetched by the if sentence, and never get to the elseif.
- *Line 327 in aes.sv: 1 - 0 not covered.* Impossible to cover, since it is placed in an elseif statement where the condition is fetched by the if condition.
- *Line 345 in aes.sv: 0 - 1 not covered.* Same as the previous one.

	Total	Covered	Percent
Conditions	57	50	87.72
Logical	57	50	87.72

Table 7.3: Condition coverage summary

7.2.3 FSM Coverage

The FSM coverage summary is shown in Table 7.4. The FSM coverage short report is placed in Appendix D.3. States are 100%, which means that all states have been covered.

Transition coverage is low. Comparing the FSM coverage report with the FSMs, it appears that all the transitions not covered are transitions from each state and back to the start state. This means that reset has not been done when the FSM was in any other state than the start state. Which is correct, since the testbench does not run resets in the middle of the encryption/decryption operations.

Sequence coverage and FSM coverage are low since they both rely on the transition coverage.

7.2.4 Branch Coverage

The branch coverage summary is shown in Table 7.5. The branch coverage short report is placed in Appendix D.4. The uncovered conditions were:

	Total	Covered	Percent
FSMs	3	1	33.33
States	17	17	100.00
Transitions	18	13	72.22
Sequences	78	44	56.41

Table 7.4: FSM coverage summary

- *In subbytes.v near line 244.* A missing default is reported not covered. The case is only checking values 0 and 1 of a 4-state signal. No X or Z values have been inserted, so the missing default clause is not covered.
- *In mixcolum.v near line 190.* The state machine in MixColumns has an empty default state which is not covered.
- *In wb_aescontroller.v near line 107.* Missing default of the Wishbone address case selection not covered.
- *In wb_aescontroller.v near line 150.* Same as the previous, now for write address selection instead of read address selection.
- *In wb_aescontroller.v near line 153.* Wishbone address 8'h0 not covered. Same as reported in the line coverage reports.

	Total	Covered	Percent
Branches	91	86	94.51

Table 7.5: Branch coverage summary

7.2.5 Path Coverage

The path coverage summary is shown in Table 7.6. The path coverage short report is placed in Appendix D.5. Path coverage was reported to be low, but most of the paths reported as not covered seems to be paths that are impossible, because of an illegal combination of branches.

- *In wb_aescontroller.v near lines 86-177.* Multiple paths not covered due to two *if* blocks that are never activated at the same time. The AES module will never finish while the Wishbone bus is being used.

	Total	Covered	Percent
Paths	157	59	37.58

Table 7.6: Path coverage summary

7.2.6 Toggle Coverage

The toggle coverage summary is shown in Table 7.7. The toggle coverage short report is placed in Appendix D.6. Many of the signals not toggled turns out to be unused signals, or unused bits ranges of signals. These are almost exclusively from the Wishbone module, as shown in the toggle coverage module report in Appendix D.7.

	Total	Covered	Percent
Regs	156	136	87.18
Reg bits	5104	4568	89.50
Reg bits(0->1)	5104	4568	89.50
Reg bits(1->0)	5104	4568	89.50
Nets	66	58	87.88
Net bits	1496	1331	88.97
Net bits(0->1)	1496	1331	88.97
Net bits(1->0)	1496	1331	88.97

Table 7.7: Toggle coverage summary

7.3 Assertions

The assertion reports were analysed to see if any of the assertions had failed during the simulations. No errors were reported in any of the assertion reports.

In the 200 000 run simulation report, all the concurrent assertions report 92 941 707 attempts. This corresponds to the total number of clock cycles.

The assertion in the Checker, which controls that Scoreboard and DUT results are equal, reports 200 000 hits. This is correct, since the assertion is implemented as an immediate assertion triggered once for each round.

The two assertions monitoring the randomization of data in each of the scenarios, the a_randomize_d and a_randomize_e, reports 132549 and 133097 hits, respectively. This is about the double of expected hits, since they both have probability set to 33%, and the total number should be 200 000. An error with the definition is

suspected, since their sums added equals more than the total number.

The two handshake assertions both reports a real success number of 1 727 245. This corresponds to the total number of handshakes. It gives an average of about 8.63 handshakes per round. The number needed for a full encryption/decryption round is 13 handshakes. However, the reset operations (representing 33% of the operations) does not require any handshakes. Therefore, the real success number is very close to the expected 66% of 13 * 200 000 (which is about 1 733 000).

The one incomplete attempt of the assertion checking the max length of the ACK signal has probably happened because the testbench terminates before the last ack signal is lowered.

7.4 Functional Coverage

Due to the lack of tools for analysis, the only functional coverage data available is the cover property based on the rounds assertion. This cover property, called c_rounds, reports 92941707 matches. The number only tells that the assertion has been triggered every clock cycle.

7.5 Testbench Discussion and Future Work

The code coverage analysis shows that the testbench covers the DUT well. However, some lacks of functionality were discovered.

First of all, a more sophisticated reset behavior should have been implemented. Now, reset is only run between the simulation rounds, never in the middle of an encryption or decryption operation. A more realistic reset behavior would also include mid-operation resets, which is more relevant to how a real-life user probably would use the reset. This is required to achieve a higher FSM transition coverage. Such a reset functionality could be implemented as callbacks (providing resets in the middle of the communication only), or even better as its own thread (providing resets at random times).

A bathtub-shaped data distribution, as introduced in the theory, was planned. However, it was not implemented, due to difficulties with data types and a lack of time. With a further detail study of SystemVerilog data types this would probably be possible to implement.

An error reporting mechanism was described in the theory. It would record a sliding window of simulation data, and report it when an assertion fails. This func-

tionality was not implemented due to time limitations. The implementation of such a feature would require a memory recording data and testbench status, as well as a report functionality trigged by the action clause of each assertion.

It could be useful to be able to stop, and afterwards restart a simulation. New seeding of the testbench would be required to make the random generator create new values. In order to achieve this, it would be necessary to implement a seeding functionality.

More functional coverage groups should have been implemented to enable a more complete coverage analysis. The implementation of coverage groups was stopped when it became clear that the analysis software did not support functional coverage analysis based on cover groups.

Chapter 8

Conclusion

THE MAIN TASKS of this thesis was to study literature about modern verification to create a verification plan for an AES RTL model, and to implement and simulate the planned verification system using SystemVerilog and Synopsys VCS.

The verification planning was done according to the recommended literature, and follows the main guidelines given in the Verification Methodology Manual (VMM). The planned verification process builds on modern verification techniques like constrained randomization, functional coverage, object orientation and assertions.

A testbench was implemented with the requirements and features planned in the verification plan. It has a layered architecture to ensure high maintainability and reusability. It exploits the advanced randomization functionalities of SystemVerilog. It uses assertions to check the RTL model for errors. The testbench was used to simulate the AES model using the Synopsys VCS software.

Due to some problems with the Synopsys VCS licence held by the university, some features of VCS was not available. Unfortunately, functional coverage analysis was almost non-supported in the available software. Therefore, the functional coverage analysis carried out is limited.

Code coverage metrics were used to track the progress and quality of the simulations. Many of the metrics reached the planned goals of 100% explained coverage. This means that all functionality of the DUT was tested. Most problems that arised were due to three factors: Bad coding style in the Wishbone controller of the DUT, misunderstandings due to the lack of documentation, and a limited reset behavior in the testbench.

Since no relevant functional coverage data was available for analysis it is not possible to draw any conclusions about the adequacy of the random data generated, and the completeness of the DUT implementation. However, since all 200,000+ en-

cryption and decryption results from the DUT were correct in the simulation, there is reason to believe that the functionality of the model is correct and adequate in most situations.

Some topics for possible improvement of the verification system were given in the discussion chapter. The most important ones would be a more sophisticated reset functionality, as well as a more advanced functionality for random data distribution.

In conclusion, the verification plan has been created and the verification process has been carried out, both satisfying the requirements given.

Bibliography

[1] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Springer Science + Business Media, Inc., 2006.

[2] C. Spear, *SystemVerilog for Verification*. Springer Science + Business Media, Inc., 2006.

[3] Federal Information Standards Publication, *Specification for the Advanced Encryption Standard*, November 2001.

[4] Synopsys, Inc., *VCS / VCS MX Coverage Metrics User Guide*, April 2006.

[5] P. James, *Verification Plans*. Kluwer Academic Publishers, 2004.

[6] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, *Verification Methodology Manual for SystemVerilog*. Springer Science + Business Media, 2006.

[7] Synopsys, Inc., *SystemVerilog Assertions Checker Library Quick Reference*, April 2006.

[8] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer Science + Business Media, Inc., 2005.

[9] D. Mills and S. Sutherland, "Systemverilog assertions are for design engineers too!," *SNUG*, 2006.

[10] J. Cooley, "The dvcon'05 verification census," October 2005.

[11] Synopsys, Inc., *VCS DirectC Interface User Guide*, April 2006.

[12] Synopsys, Inc., *VCS/VCSi User Guide*, April 2006.

[13] "Opencores website." http://www.opencores.org/.

[14] J. Castillo, "128 aes verilog module." http://www.opencores.org/projects.cgi/systemcaes/.

[15] B. Gladman, "A specification for rijndael, the aes algorithm," September 2003.

[16] J. Daemen and V. Rijmen, *The Design of Rijndael*. Springer-Verlag, 2002.

[17] "Specification for the: Wishbone system-on-chip (soc) interconnection architecture for portable ip cores," September 2002.

[18] M. Turpin, "The dangers of living with an x (bugs hidden in your verilog)," *ARM Ltd.*, 2003.

[19] Synopsys, Inc., *SystemVerilog Testbench Constructs*, April 2006.

[20] B. Gladman, "Brian gladman's home page." http://fp.gladman.plus.com/.

[21] L. Hancock, M. Krieger, and S. Zamir, *The C Primer*. McGraw-Hill, Inc., 3rd ed., 1990.

[22] S. Sutherland, *The Verilog PLI Handbook*. Kluwer Academic Publishers, 2nd ed., 2002.

Appendix A

Day-in-the-Life Document

A.1 General Information

The 128 AES module by Castillo is a module that encrypts and decrypts data according to the AES specification. In this implementation, key size is 128 bits. According to the AES specification, data block size is also 128 bits. Other properties:

- The module is optimized for low area.
- The S-box is implemented as logic, not as a table stored in memory.
- There are 3 FSMs in the design. They are situated in the *subbytes*, *keysched* and *mixcolum* modules.

The operation in AES is not pipelined, one data block must be processed completely before the module is ready for a new data block.

A.2 Module Blocks

Blocks and signals for the AES module are shown in figure A.1.

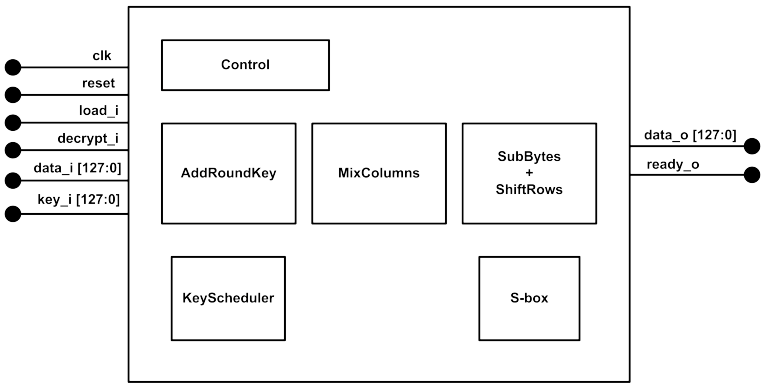

Figure A.1: AES module blocks

A.3 Communication Bus

Communication with the module is done via a Wishbone bus interface. Unfortunately, the WishBone Interface implementation is not documented or code-commented. The Wishbone feature is delivered as a stand-alone module, so a top module connecting the AES and Wishbone modules has to be developed in order to make the system able to communicate using the Wishbone bus. The signals of the Wishbone bus block are shown in figure A.2 and table A.1.

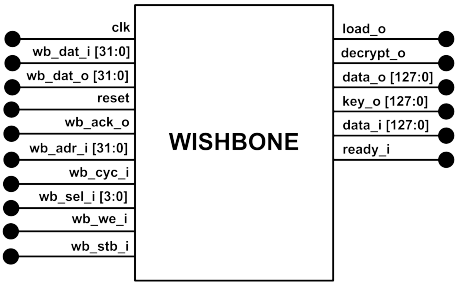

Figure A.2: Wishbone module

From the code it appears that the module only supports the absolute minimum of features to be Wishbone compliant. The module can only act as a slave module. It also only supports the simplest form of WishBone data transfer: Single Transfer.

The wb_sel_i signal is not used by the module in any operation mode. Valid addresses for the wb_adr_i signal are shown in table A.2.

Signal	Direction	Width	Description
clk	Input	1	System clock
reset	Input	1	System reset
wb_stb_i	Input	1	Slave selection indicator
wb_dat_o	Output	32	Data output
wb_dat_i	Input	32	Data input
wb_ack_o	Output	1	Acknowledge signal
wb_adr_i	Input	32	Address input
wb_we	Input	1	Write enable
wb_cyc_i	Input	1	Valid bus cycle indicator
wb_sel_i	Input	4	(Not in use)

Table A.1: Signals of the Wishbone module

Address	Bus data description
8'h0	Encryption/decryption Configuration
8'h4	Input data, bit range [127:96]
8'h8	Input data, bit range [95:64]
8'hC	Input data, bit range [63:32]
8'h10	Input data, bit range [31:0]
8'h14	Input key, bit range [127:96]
8'h18	Input key, bit range [95:64]
8'h1C	Input key, bit range [63:32]
8'h20	Input key, bit range [31:0]
8'h24	Output data, bit range [127:96]
8'h28	Output data, bit range [95:64]
8'h2C	Output data, bit range [63:32]
8'h30	Output data, bit range [31:0]

Table A.2: Valid addresses of wb_adr_i

A.4 AES Operation Flow

A flow diagram of the AES encryption is shown in figure A.3.

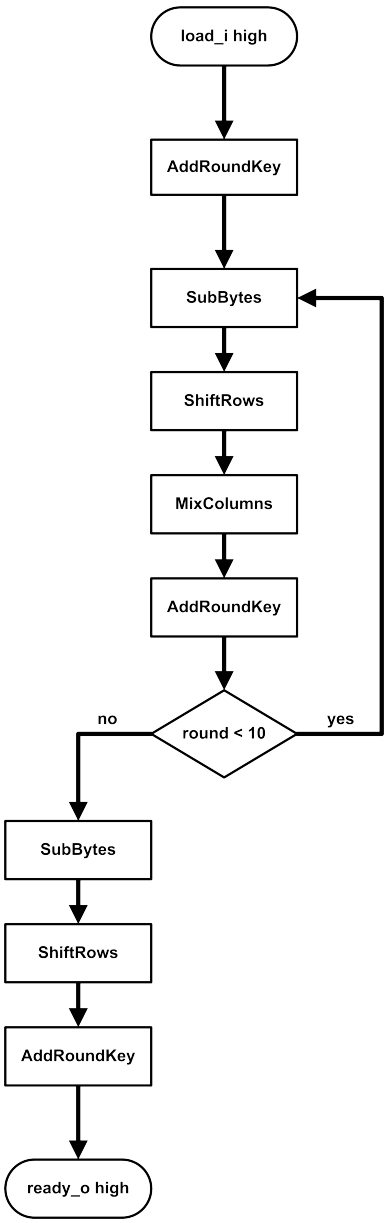

Figure A.3: AES flow

A.5 AES Internal Operations

SubBytes is a non-linear transformation that applies the S-box to each element of the State table, as shown in figure A.4.

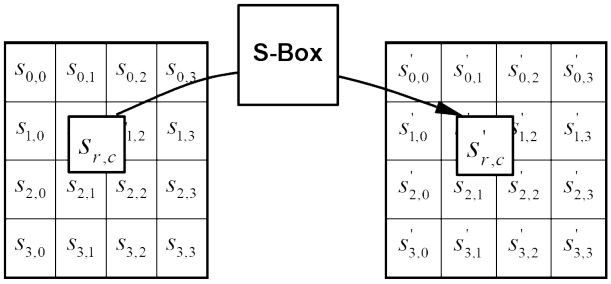

Figure A.4: SubBytes operation [3]

S-box could be viewed as the lookup table for the substitution in SubBytes. Byte value xy is changed with the corresponding value in figure A.5.

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure A.5: S-box table [3]

ShiftRows shifts every byte in State after a predefined pattern: First row is shifted 0 positions to the left, second row is shifted 1 position to the left, and so on. Shown in figure A.6.

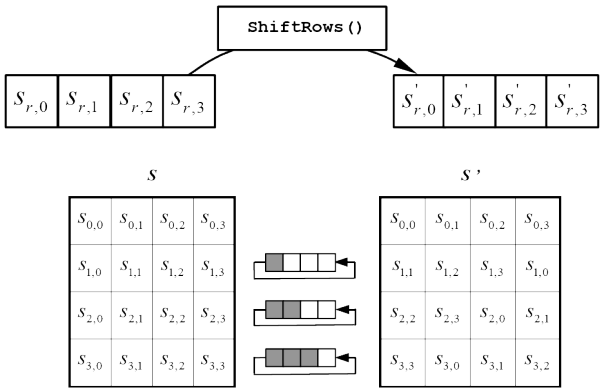

Figure A.6: Shiftrows operation [3]

MixColumns, in figure A.7, operates on each column using a multiplication operation.

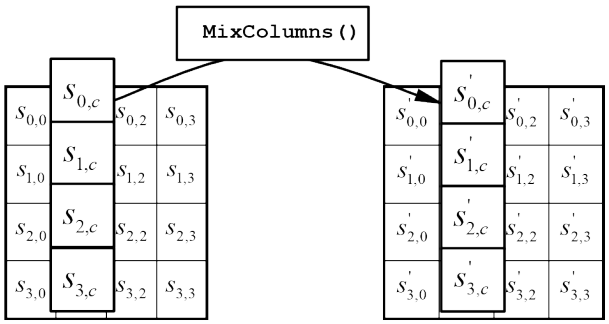

Figure A.7: MixColumns operation [3]

AddRoundKey combines State with a round-specific key using bitwise XOR, shown in figure A.8.

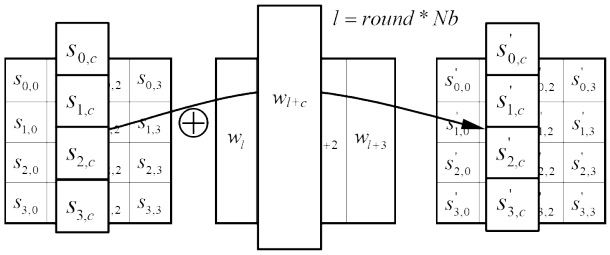

Figure A.8: AddRoundKey operation [3]

Appendix B

Verification Plan

B.1 Typical Operation

Input:	Randomize key, data and operation (encryption or decryption)
Input:	Send inputs to the module
Output:	Receive outputs from the module
Output:	Compare outputs with expected results
Output:	Measure coverage to track progress

Table B.1: Typical operation

B.2 Tools and Technologies

Languages:	SystemVerilog
Technology:	Simulation
Tools:	Synosys VCS ver. 2005.06-SP2

Table B.2: Tools, language and technologies

B.3 Implementation Phases

See table B.3.

Phase	Goal
1	Simple communication with module (without Wishbone interface)
2	Communication using Wishbone interface
3	Output checking using reference model
4	Assertions for checking properties
5	Constrained random stimuli
6	Coverage calculation
7	Verification of decryption functionality

Table B.3: Phases

B.4 System Architecture

Some changes from the layer figure: Monitor and driver is merged in the BFM block. Agent is removed. This gives the architecture in figure B.1.

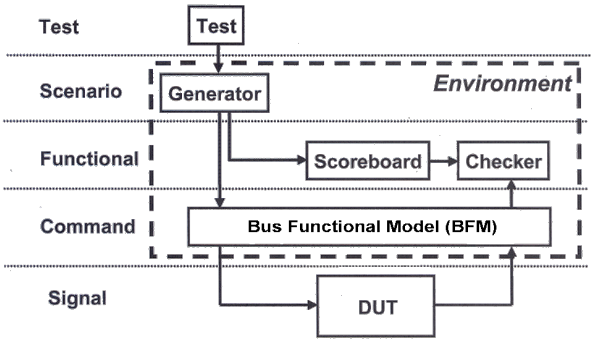

Figure B.1: Testbench data flow

- Test instantiates Environment
- Environment instantiates Generator, Scoreboard, BFM, Checker
- Communication in the system is done using mailboxes

Figure B.2 shows the flow of data through the components of the testbench. The flowing data will be contained in transaction objects, created in the Generator.

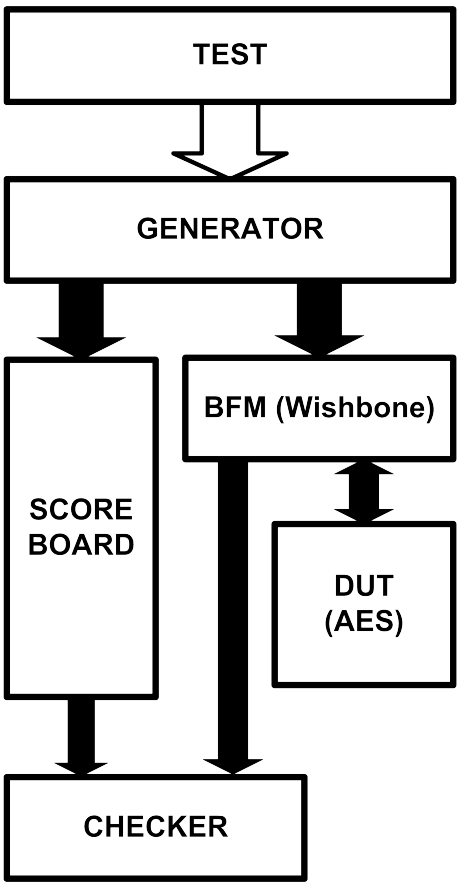

Figure B.2: Testbench data flow

B.5 Layer Implementation

See tables B.4 and B.5.

Component	Layer	Phase
BFM	Command	2
Scoreboard	Functional	3
Checker	Functional	3
Generator	Scenario	5

Table B.4: Component Implementation

	Command	Functional	Scenario
Phase 1			
Phase 2	BFM		
Phase 3		Scoreboard, Checker	
Phase 4			
Phase 5			Generator
Phase 6			
Phase 7			

Table B.5: XY-grid phases/layers

B.6 Scenarios

- Encryption (Probability: 33.3%)
- Decryption (Probability: 33.3%)
- System Reset (Probability: 33.3%)

B.7 Block Descriptions

The properties of the Scoreboard, Checker, BFM and Generator blocks are shown in Tables B.6, B.7, B.8 and B.9, respectively.

Component:	Scoreboard
Operation:	Block containing reference model in C language.
Inputs:	128 bit key, 128 bit data to be encrypted/decrypted, 1 bit to select encryption/decryption
Output:	encrypted/decrypted data

Table B.6: Scoreboard description

Component:	Checker
Operation:	Compare outputs from DUT and scoreboard.
Inputs:	128 bit outputs received from BFM and scoreboard
Output:	Status

Table B.7: Checker description

Component:	BFM
Operation:	Communicate with DUT via the Wishbone bus
Inputs:	128 bit key, 128 bit data, 1 bit to select encryption/decryption
Output:	encrypted/decrypted data

Table B.8: BFM description

Component:	Generator
Operation:	Generate constrained-random values. Contains scenarios.
Inputs:	Constraints
Output:	128 bit key, 128 bit data, 1 bit to select encryption/decryption

Table B.9: Generator description

B.8 Randomization

- Randomize key
- Randomize input
- Randomize operation: Encryption, decryption or reset
- Random delays in Wishbone communication (0 to 10 clock cycles)

Input and key data should have a bathtub distribution to provoke overflow errors.

B.9 Assertions

Functionality to verify	Placement	Type	Assigned to
ACK is never set high without a preceding STB	External	Concurrent	Verifier
Wishbone bus never has any X or Z bits when reading is allowed	External	Concurrent	Verifier
Request from testbench results in ACK from DUT within 10 clock cycles	External	Concurrent	Verifier
FSM in AES control module follows the right sequence	Internal	Concurrent	Verifier
Number of rounds of AES operation does not exceed 10	Internal	Concurrent	Verifier
AES block inputs does not have any X or Z bits when load_i is high	Internal	Concurrent	Verifier
DUT output value is the same as the scoreboard value at the end of each run	External	Immediate	Verifier

Table B.10: Assertions

B.10 Code Coverage

Code Coverage	Goal
Line Coverage	100% explained
Condition Coverage	100% explained
FSM Coverage	100% explained
Branch Coverage	100% explained
Path Coverage	100% explained
Toggle Coverage	100% explained

Table B.11: Code Coverage Analysis Plan

B.11 Functional coverage

Cover group	Bins	Placement	Coverage Goal
DAT_I signal of Wishbone bus (Check input data distribution of Wishbone cycles)	10	External	100%
DAT_O signal of Wishbone bus (Check output data distribution of Wishbone cycles)	10	External	100%
Key in signal on AES module (Check key data distribution)	20	Internal	100%
Data in signal on AES module (Check input data distribution)	20	Internal	100%
Inputs to the SubBytes module (Check that S-Box is fully verified) (Cross coverage)	16 x 16	Internal	100%

Table B.12: Cover groups

Appendix C

Testbench Source Code

Appendix C.1: top.sv

```
1  `include "timescale.v"
2
3  module top;
4    bit clk=0;
5    always #5 clk = ~clk;
6
7    wb_if          wbif (clk);
8    test           tst  (wbif);
9    aestop         dut  (wbif);
10   svaexternal    sva  (wbif);
11
12 endmodule
```

```
 1
 2  interface wb_if(input bit clk);
 3
 4    logic reset, wb_stb, wb_ack, wb_we, wb_cyc;
 5    logic [31:0] wb_dat_o, wb_dat_i, wb_adr;
 6    logic [3:0] wb_sel;
 7
 8    modport TST (output reset, wb_stb, wb_we, wb_cyc, wb_adr, wb_dat_i, wb_sel,
 9                 input clk, wb_dat_o, wb_ack);
10
11    modport DUT (output wb_dat_o, wb_ack,
12                 input reset, wb_stb, wb_we, wb_cyc, wb_adr, wb_dat_i, wb_sel, clk);
13
14    modport SVA (input wb_dat_o, wb_ack, reset, wb_stb, wb_we, wb_cyc, wb_adr, wb_dat_i, wb_sel, clk);
15
16  endinterface
17
18
19
20
```

```
 1
 2  // Reference to AES c-model
 3  extern void aescmodel(input bit [127:0], input bit [127:0], input bit, output bit [127:0]);
 4
 5
 6  program automatic test (wb_if.TST wbif);
 7
 8          event handshake;        // Handshake event for BFM and Generator interaction
 9          const int ROUNDS=10;    // Number of tests
10
11
12
13
14
15          // *** TRANSACTION CLASS
16          // *** Objects for data exchange between transactors
17          // ***
18          class Transaction;
19                  rand bit [127:0] data_in;
20                  rand bit [127:0] key;
21                  bit [127:0] data_out;
22                  bit [127:0] data_c_out;
23                  bit decrypt;
24                  bit reset = 0;
25          endclass
26
27
28
29          // *** GENERATOR CLASS
30          // ***
31          class Generator;
32
33
34                  mailbox gen2bfm;        // Mailbox to BFM
35                  mailbox gen2scb;        // Mailbox to Scoreboard
36
37                  function new(mailbox gen2bfm, gen2scb);
38                          this.gen2bfm = gen2bfm;
39                          this.gen2scb = gen2scb;
40                  endfunction
41
42                  task run;
43                          begin
44                                  // Start with one reset to make the AES module work
45                                  reset_task();
46
47
48                                  for (int i=0; i<ROUNDS; i++) begin
49
50                                          // Start new encryption, decryption or reset
```

```
51                                      randsequence (stream)
52                                      stream : encrypt := 5 |
53                                               decrypt := 5 |
54                                               reset := 5;
55                                      encrypt : { encrypt_task; } |
56                                                { encrypt_task; } encrypt;
57                                      decrypt : { decrypt_task; } |
58                                                { decrypt_task; } decrypt;
59                                      reset :   { reset_task; } |
60                                                { reset_task; } reset;
61                                      endsequence
62
63                                      //wait for BFM to fetch object
64                                      @handshake;
65                              end
66                      end
67              endtask
68
69
70              // Create tr object for encryption
71              task encrypt_task;
72                      begin
73                              // new tr object for data storage
74                              Transaction tr = new();
75
76                              // randomize key and indata
77                              a_randomize_e: assert(tr.randomize());
78                              tr.decrypt = 0;
79
80                              // send tr object to BFM and Scoreboard mailboxes
81                              gen2bfm.put(tr);
82                              gen2scb.put(tr);
83                      end
84              endtask
85
86              // Create tr object for decryption
87              task decrypt_task;
88                      begin
89                              // new tr object for data storage
90                              Transaction tr = new();
91
92                              // randomize key and indata
93                              a_randomize_d: assert(tr.randomize());
94                              tr.decrypt = 1;
95
96                              // send tr object to BFM and Scoreboard mailboxes
97                              gen2bfm.put(tr);
98                              gen2scb.put(tr);
99                      end
100             endtask
```

```
101
102                 // Create tr object for reset
103                 task reset_task;
104                         begin
105                                 // new tr object for data storage
106                                 Transaction tr = new();
107
108                                 tr.reset = 1;
109
110                                 // send tr object to BFM and Scoreboard mailboxes
111                                 gen2bfm.put(tr);
112                                 gen2scb.put(tr);
113                         end
114                 endtask
115
116
117         endclass
118
119
120
121         // *** SCOREBOARD CLASS
122         // ***
123         class Scoreboard;
124
125                 Transaction tr;
126
127                 mailbox gen2scb, scb2chk;
128
129                 function new(mailbox gen2scb, scb2chk);
130                         this.gen2scb = gen2scb;
131                         this.scb2chk = scb2chk;
132                 endfunction
133
134                 task run;
135                         repeat (ROUNDS) begin
136                                 gen2scb.get(tr);
137
138                                 if (!tr.reset) aescmodel(tr.data_in,tr.key,tr.decrypt,tr.data_c_out);
139                                 scb2chk.put(tr);
140                         end
141                 endtask
142         endclass
143
144
145         // *** BFM CALLBACK CLASS
146         // ***
147         class BFM_cbs;
148                 task pre_tx();
149                         repeat ($urandom_range(0,9)) begin
150                                 @(posedge wbif.clk);
```

```
151                     end
152             endtask
153
154             task post_tx();
155                     // do nothing for now
156             endtask
157     endclass
158
159
160
161     // *** BUS FUNCTIONAL MODEL CLASS
162     // *** Wishbone bus communication with device-under-test
163     // ***
164     class BFM;
165             int data_cycle_out[4];          //Outdata from Wishbone cycles
166             int data_cycle_in[9];           //Indata for WB cycles
167
168             BFM_cbs cbs;
169             Transaction tr;
170
171
172             // Define cover group for WB indata
173             covergroup covWbDatIn;
174                     coverpoint wb_if.wb_dat_i;
175             endgroup
176
177
178             mailbox gen2bfm;        //Mailbox from Generator
179             mailbox bfm2chk;        //Mailbox to Checker
180
181             function new(mailbox gen2bfm, bfm2chk);
182                     this.gen2bfm = gen2bfm;
183                     this.bfm2chk = bfm2chk;
184                     covWbDatIn = new;
185             endfunction
186
187
188             task run;
189                     repeat (ROUNDS) begin
190                             // Get data from Generator
191                             gen2bfm.get(tr);
192
193                             // Tell Generator to continue
194                             ->handshake;
195
196
197                             // Fill array with data from received tr object
198                             data_cycle_in[8] = tr.key[31:0];
199                             data_cycle_in[7] = tr.key[63:32];
200                             data_cycle_in[6] = tr.key[95:64];
```

```
201                             data_cycle_in[5] = tr.key[127:96];
202                             data_cycle_in[4] = tr.data_in[31:0];
203                             data_cycle_in[3] = tr.data_in[63:32];
204                             data_cycle_in[2] = tr.data_in[95:64];
205                             data_cycle_in[1] = tr.data_in[127:96];
206                             data_cycle_in[0] = (tr.decrypt*4)+1;    //decrypt: send 3'b101
207                                                                     //encrypt: send 3'b001
208
209
210
211                             if (tr.reset == 1) begin
212                                     // Reset device
213                                     @(posedge wbif.clk);
214                                     sendreset();
215                                     $display("RESET!");
216                             end
217                             else begin
218
219                                     // Send 9 cycles of data
220                                     for (int i=8; i>=0; i--) begin
221
222                                             @(posedge wbif.clk);
223                                             writecycle((i*4),data_cycle_in[i]);
224
225                                     end // for
226
227
228                                     // Wait for AES calculation to complete
229                                     repeat(600) @(posedge wbif.clk);
230
231
232                                     // Read outdata using 4 cycles
233                                     for (int i=3; i>=0; i--) begin
234
235                                             @(posedge wbif.clk);
236                                             readcycle((i*4)+36);
237
238                                     end // for
239
240
241                                     // Add out data to tr object
242                                     tr.data_out[31:0] = data_cycle_out[3];
243                                     tr.data_out[63:32] = data_cycle_out[2];
244                                     tr.data_out[95:64] = data_cycle_out[1];
245                                     tr.data_out[127:96] = data_cycle_out[0];
246                             end
247
248                             // Send tr object to Checker mailbox
249                             bfm2chk.put(tr);
250
```

```
251                     end
252             endtask
253
254
255             // Reset device
256             task sendreset();
257             begin
258                     @(posedge wbif.clk);
259                     wbif.reset = 1'b0;
260                     @(posedge wbif.clk);
261                     wbif.reset = 1'b1;
262                     @(posedge wbif.clk);
263                     wbif.reset = 1'b0;
264                     wbif.wb_cyc = 1'b0;
265             end
266             endtask
267
268
269             // Set signals on Wishbone bus
270             task setsignals(bit we, bit [31:0] adr, dat_i);
271             begin
272                     cbs.pre_tx();           // Pre-transfer callback
273
274                     wbif.wb_adr = adr;      // data address
275                     wbif.wb_dat_i = dat_i;  // data in
276                     wbif.wb_cyc = 1'b1;     // cycle indicator
277                     wbif.wb_stb = 1'b1;     // strobe
278                     wbif.wb_we = we;        // write enable
279
280                     cbs.post_tx();          // Post-transfer callback
281             end
282             endtask
283
284
285             // write data to WB bus
286             task writecycle(bit [31:0] adr, dat_i);
287             begin
288                     // Set signals on WB bus
289                     setsignals(1'b1,adr,dat_i);
290
291                     @(posedge wbif.clk);
292
293                     // Sample functional coverage data
294                     covWbDatIn.sample();
295
296                     // Wait for ack from device
297                     while (~wbif.wb_ack) @(posedge wbif.clk);
298
299                     // End write cycle
300                     wbif.wb_cyc = 1'b0;
```

```
301                         wbif.wb_stb = 1'b0;
302
303                 end
304                 endtask
305
306
307                 // read data from WB bus
308                 task readcycle(bit [31:0] adr);
309                 begin
310                         // Set signals on WB bus
311                         setsignals(1'b0,adr,32'b0);
312
313                         @(posedge wbif.clk);
314
315                         // Wait for ack from device
316                         while (~wbif.wb_ack) @(posedge wbif.clk);
317
318                         // Fetch data from WB bus
319                         data_cycle_out[((adr-36)/4)] = wbif.wb_dat_o;
320
321                         // End read cycle
322                         wbif.wb_cyc = 1'b0;
323                         wbif.wb_stb = 1'b0;
324
325                 end
326                 endtask
327
328
329
330         endclass
331
332
333
334
335         // *** CHECKER CLASS
336         // ***
337         class Checker;
338                 Transaction tr;
339
340                 mailbox bfm2chk, scb2chk;
341
342                 function new(mailbox bfm2chk, scb2chk);
343                         this.bfm2chk = bfm2chk;
344                         this.scb2chk = scb2chk;
345                 endfunction
346
347                 task run;
348                         repeat (ROUNDS) begin
349
350                                 // Receive transaction object from BFM. Wait for Scoreboard too
```

```
351                             bfm2chk.get(tr);
352                             scb2chk.get(tr);
353
354                             a_output: assert (tr.data_out == tr.data_c_out);
355
356                             $display("");
357                             $display("Data in:        %h",tr.data_in);
358                             $display("Key in:         %h",tr.key);
359                             $display("Decrypt:        %h",tr.decrypt);
360                             $display("Reset:          %h",tr.reset);
361                             $display("BFM data out:   %h",tr.data_out);
362                             $display("SCB data out:   %h",tr.data_c_out);
363                             $display("");
364
365                     end
366             endtask
367     endclass
368
369
370
371
372     class Environment;
373
374             mailbox gen2bfm;
375             mailbox gen2scb;
376             mailbox bfm2chk;
377             mailbox scb2chk;
378
379
380             BFM bfm;
381             Generator gen;
382             Scoreboard scb;
383             Checker chk;
384
385             task run;
386             begin
387                     gen2bfm = new;
388                     bfm2chk = new;
389                     gen2scb = new;
390                     scb2chk = new;
391
392                     gen = new(gen2bfm,gen2scb);
393                     bfm = new(gen2bfm,bfm2chk);
394                     scb = new(gen2scb,scb2chk);
395                     chk = new(bfm2chk,scb2chk);
396
397                     $display("*************************************************");
398                     $display("* AES WB Test ...                               *");
399                     $display("*************************************************");
400
```

```
401
402                         fork
403                                 bfm.run();
404                                 gen.run();
405                                 scb.run();
406                                 chk.run();
407                         join
408
409                         $display("*** Finished ***");
410                 end
411                 endtask
412
413         endclass
414
415
416
417
418         // Define environment
419         Environment env;
420
421         // Start environment on program start
422         initial begin
423                 env = new;
424                 env.run();
425         end
426
427
428
429
430
431
432 endprogram
```

```
 1
 2
 3  // *** SVAEXTERNAL module
 4  // *** Module with external assertions
 5  // ***
 6  module svaexternal(wb_if.SVA wbif);
 7
 8
 9
10          // Check for acks without strobe
11          property p_stb_ack;
12                  @(posedge wb_if.clk) ($rose(wb_if.wb_ack) |-> wb_if.wb_stb);
13          endproperty
14
15          // Check for unknown data on bus while bus reading is allowed
16          property p_data_read;
17                  @(posedge wb_if.clk) not ($isunknown(wb_if.wb_dat_o) && (~wb_if.wb_we) && (wb_if.wb_ack));
18          endproperty
19
20
21          // Check for proper handshake (library assertion)
22          assert_handshake #(0, 0, 10, 0, 0, 3, 0,"ERROR: handshake incorrect")
23                  invalid_handshake (wb_if.clk, ~wb_if.reset, wb_if.wb_cyc, wb_if.wb_ack);
24
25
26
27          a_stb_ack: assert property(p_stb_ack);
28          a_data_read: assert property(p_data_read);
29
30  endmodule
```

```
1  `include "timescale.v"
2
3  module aestop (wb_if.DUT wbif);
4
5    logic resetwb, resetaes, load, decrypt, ready;
6    logic [127:0] key, data_i, data_o;
7
8    always @(wbif.reset)
9    begin
10      resetwb = wbif.reset;
11      resetaes = ~(wbif.reset);
12   end
13
14   aes                    aes (wbif.clk,resetaes,load,decrypt,data_i,key,ready,data_o);
15   wb_aes_controller      wbc
           (wbif.clk,resetwb,wbif.wb_stb,wbif.wb_dat_o,wbif.wb_dat_i,wbif.wb_ack,wbif.wb_adr,wbif.wb_we,wbif.wb_cyc
           ,wbif.wb_sel,load,decrypt,ready,data_i,key,data_o);
16
17 endmodule
```

Appendix C.6: C communication function (placed in aes.c)

```
1  void aescmodel(UB *rev_test_in, UB *rev_test_key, scalar decrypt, UB *rev_test_out)
2  {
3
4      int i;
5      unsigned char test_in[N_BLOCK];
6      unsigned char test_out[N_BLOCK];
7      unsigned char test_key[N_BLOCK];
8      unsigned char test_o_key[N_BLOCK];
9
10
11     // Reverse indata
12     for (i=0;i<N_BLOCK;i++) {
13         test_in[N_BLOCK-1-i] = rev_test_in[i];
14         test_key[N_BLOCK-1-i] = rev_test_key[i];
15     }
16
17
18     // Encrypt or decrypt
19     if (decrypt == 1) {
20         aes_encrypt_128(test_in,test_out,test_key,test_o_key);
21         aes_decrypt_128(test_in,test_out,test_o_key,test_key);
22     }
23     else {
24         aes_encrypt_128(test_in,test_out,test_key,test_o_key);
25     }
26
27
28     // Reverse outdata
29     for (i=0;i<N_BLOCK;i++) {
30         rev_test_out[N_BLOCK-1-i] = test_out[i];
31     }
32
33 }
34
35
36
```

Appendix D

Coverage Reports

D.1 cmView.short_ld File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006


         SHORT SOURCE LINE COVERAGE REPORT



//****************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module


Test Coverage Result: Total Coverage

MODULE wb_aes_controller

 FILE /home/henrikru/oo2/wb_aescontroller.v

   Line No            Coverage     Block Type
    144.1             0            MISSING_DEFAULT
```

```
    153               0            CASEITEM
    154               0
    171.1             0            MISSING_DEFAULT

//-----------------------------------------------------------------


//*****************************************************************

//           Total Module Definition Coverage Summary

                   TOTAL        COVERED         PERCENT
lines              462          460             99.57
statements         511          509             99.61
blocks             122          121             99.18

  ALWAYS            29           29              100.00
  CASEITEM          47           46              97.87
  IF                28           28              100.00
  ELSE              17           17              100.00
  MISSING_ELSE      11           11              100.00
  ROUTINE           1            1               100.00
  MISSING_DEFAULT   3            0               0.00
```

D.2 cmView.short_cd File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



         SHORT CONDITION COVERAGE REPORT



//*************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module


Test Coverage Result: Total Coverage

MODULE aes

 LINE   327
 STATEMENT   if ((addroundkey_start_i && (round != 4'b0)))
                  ---------1---------    -------2-------
 EXPRESSION     -1-   -2-
                 1     0  | Not Covered

 LINE   345
 STATEMENT   if (((addroundkey_round == round) && keysched_ready_o))
                  -------------1--------------    -------2--------
 EXPRESSION     -1-   -2-
                 0     1  | Not Covered


//-------------------------------------------------------------

MODULE wb_aes_controller

 FILE  /home/henrikru/oo2/wb_aescontroller.v
```

```
 ----------------------------------------

 LINE   104
 STATEMENT   if ((wb_stb_i && wb_cyc_i && wb_we_i && ((~wb_ack_o))))
                  ---1----    ---2----    ---3---    ------4------
 EXPRESSION     -1-   -2-   -3-   -4-
                 0     1     1     1  | Not Covered
                 1     0     1     1  | Not Covered

 LINE   147
 STATEMENT   if ((wb_stb_i && wb_cyc_i && ((~wb_we_i)) && ((~wb_ack_o))))
                  ---1----    ---2----    -----3------    ------4------
 EXPRESSION     -1-   -2-   -3-   -4-
                 0     1     1     1  | Not Covered
                 1     0     1     1  | Not Covered
                 1     1     0     1  | Not Covered


//-----------------------------------------------------------------


//*****************************************************************

//           Total Module Definition Coverage Summary

//                      TOTAL        COVERED         PERCENT
//    conditions        57           50              87.72
//    logical           57           50              87.72
```

D.3 cmView.short_fd File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



         SHORT FSM COVERAGE REPORT



//************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module


Test Coverage Result: Total Coverage

MODULE subbytes


  FILE    subbytes.v
  FSM        state
// state coverage results









// state transition coverage results



// sequence coverage results
```

```
             'h1->'h0                          | Not Covered
             'h10->'h0                         | Not Covered
             'h10->'h0->'h1                    | Not Covered
             'h0->'h1->'h0                     | Not Covered Loop
             'h1->'h0->'h1                     | Not Covered Loop
//-----------------------------------------------------------------
//              Single FSM Coverage Summmary

                  TOTAL        NOT COVERED     PERCENT
States            8            0               0.00
Transitions       4            0               0.00
Sequences         8            5               62.50
//-----------------------------------------------------------------

//-----------------------------------------------------------------

//             Module Coverage Summary

                  TOTAL        COVERED         PERCENT
Fsms              1            1               100.00
States            8            8               100.00
Transitions       4            4               100.00
Sequences         9            3               33.33



//-----------------------------------------------------------------

MODULE mixcolum


  FILE    mixcolum.v
  FSM        state
// state coverage results




// state transition coverage results

             'h1->'h0                          | Not Covered

             'h2->'h0                          | Not Covered
```

```
// sequence coverage results
             'h1->'h0                          | Not Covered
             'h2->'h0                          | Not Covered
             'h1->'h2->'h0                     | Not Covered
             'h2->'h0->'h1                     | Not Covered
             'h0->'h1->'h0                     | Not Covered Loop
             'h1->'h0->'h1                     | Not Covered Loop
             'h0->'h1->'h2->'h0                | Not Covered Loop
             'h1->'h2->'h0->'h1                | Not Covered Loop
             'h2->'h0->'h1->'h2                | Not Covered Loop
//------------------------------------------------------------------
//              Single FSM Coverage Summmary

                   TOTAL        NOT COVERED     PERCENT
States             4            0               0.00
Transitions        6            2               33.33
Sequences          25           9               36.00
//------------------------------------------------------------------

//------------------------------------------------------------------

//             Module Coverage Summary

                   TOTAL        COVERED         PERCENT
Fsms               1            0               0.00
States             4            4               100.00
Transitions        6            4               66.67
Sequences          25           16              64.00



//-----------------------------------------------------------------

MODULE keysched


  FILE    keysched.v
  FSM        state
// state coverage results
```

```
// state transition coverage results

             'h1->'h0                          | Not Covered

             'h2->'h0                          | Not Covered

             'h3->'h0                          | Not Covered


// sequence coverage results
             'h1->'h0                          | Not Covered
             'h2->'h0                          | Not Covered
             'h3->'h0                          | Not Covered
             'h1->'h2->'h0                     | Not Covered
             'h2->'h0->'h1                     | Not Covered
             'h2->'h3->'h0                     | Not Covered
             'h3->'h0->'h1                     | Not Covered
             'h1->'h2->'h3->'h0                | Not Covered
             'h2->'h3->'h0->'h1                | Not Covered
             'h3->'h0->'h1->'h2                | Not Covered
             'h0->'h1->'h0                     | Not Covered Loop
             'h1->'h0->'h1                     | Not Covered Loop
             'h0->'h1->'h2->'h0                | Not Covered Loop
             'h1->'h2->'h0->'h1                | Not Covered Loop
             'h2->'h0->'h1->'h2                | Not Covered Loop
             'h0->'h1->'h2->'h3->'h0           | Not Covered Loop
             'h1->'h2->'h3->'h0->'h1           | Not Covered Loop
             'h2->'h3->'h0->'h1->'h2           | Not Covered Loop
             'h3->'h0->'h1->'h2->'h3           | Not Covered Loop
//------------------------------------------------------------------
//              Single FSM Coverage Summmary

                  TOTAL        NOT COVERED     PERCENT
States            5            0               0.00
Transitions       8            3               37.50
Sequences         44           19              43.18
//------------------------------------------------------------------

//------------------------------------------------------------------

//             Module Coverage Summary

                  TOTAL        COVERED         PERCENT
```

```
Fsms                1           0               0.00
States              5           5               100.00
Transitions         8           5               62.50
Sequences           44          25              56.82


//-----------------------------------------------------------------


//*****************************************************************

//           Total Module Definition Coverage Summary

                    TOTAL       COVERED         PERCENT
Fsms                3           1               33.33
States              17          17              100.00
Transitions         18          13              72.22
Sequences           78          44              56.41
```

D.4 cmView.short_bd File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



         SHORT BRANCH COVERAGE REPORT



//************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module


Test Coverage Result: Total Coverage

MODULE subbytes

145       if(!reset)
            -1-
146       begin
147
148         data_reg = (0);
149         state = (0);
150         ready_o = (0);
151
152       end
153       else
154       begin
155
156         data_reg = (next_data_reg);
157         state = (next_state);
158         ready_o = (next_ready_o);
159
160       end
161
162     end
```

```
163
164
165     //sub:
166     reg[127:0] data_i_var,data_reg_128;
167     reg[7:0] data_array[15:0],data_reg_var[15:0];
168


 FILE  /home/henrikru/oo2/subbytes.v
 ----------------------------------

217       case(state)
              -1-
218
219         0:
220          begin
221            if(start_i)
                 -2-
222            begin
223
224              sbox_data_o = (data_array[0]);
225              next_state = (1);
226
227            end
228         end
229
230        16:
231         begin
232
233           data_reg_var[15]=sbox_data_i;
234           //Make shift rows stage
235           case(decrypt_i)
                  -3-
236             0:
237              begin
238                `shift_array_to_128
239              end
240             1:
241              begin
242                `invert_shift_array_to_128
243              end
244           endcase
245
246           next_data_reg = (data_reg_128);
```

```
247           next_ready_o = (1);
248           next_state = (0);
249
250         end
251        default:
252         begin
253
254        sbox_data_o = (data_array[state]);
255           data_reg_var[state-1]=sbox_data_i;
256           `assign_array_to_128
257           next_data_reg = (data_reg_128);
258           next_state = (state+1);


BRANCH             -1-        -2-        -3-
                   16          -   MISSING_DEFAULT    | Not Covered
//***********


//-----------------------------------------------------------------

MODULE mixcolum

104       outmux = (decrypt_i?outy:outx);
                           -1-

105
106     end
107
108
109     //registers:


113       if(!reset)
            -1-
114       begin
115
116         data_reg = (0);
117         state = (0);
118         ready_o = (0);
119         data_o_reg = (0);
120       end
121       else
122       begin
```

```
123
124         data_reg = (next_data_reg);
125         state = (next_state);
126         ready_o = (next_ready_o);
127         data_o_reg = (next_data_o);
128
129       end
130
131     end
132
133
134     //mixcol:
135     reg[127:0] data_i_var;
136     reg[31:0] aux;
137     reg[127:0] data_reg_var;
138
```

FILE /home/henrikru/oo2/mixcolum.v

```
152       case(state)
              -1-
153
154         0:
155           begin
156             if(start_i)
                  -2-
157             begin
158               aux=data_i_var[127:96];
159               mix_word = (aux);
160               data_reg_var[127:96]=outmux;
161               next_data_reg = (data_reg_var);
162               next_state = (1);
163             end
164           end
165         1:
166           begin
167             aux=data_i_var[95:64];
168             mix_word = (aux);
169             data_reg_var[95:64]=outmux;
170             next_data_reg = (data_reg_var);
171             next_state = (2);
172           end
```

```
173         2:
174           begin
175             aux=data_i_var[63:32];
176             mix_word = (aux);
177             data_reg_var[63:32]=outmux;
178             next_data_reg = (data_reg_var);
179             next_state = (3);
180           end
181         3:
182           begin
183             aux=data_i_var[31:0];
184             mix_word = (aux);
185             data_reg_var[31:0]=outmux;
186             next_data_o = (data_reg_var);
187             next_ready_o = (1);
188             next_state = (0);
189           end
190         default:
191           begin


BRANCH              -1-        -2-
                  default       -     | Not Covered
//***********


//-----------------------------------------------------------------

MODULE wb_aes_controller

 FILE  /home/henrikru/oo2/wb_aescontroller.v
 ----------------------------------------

87           if(reset==1)
               -1-
88           begin
89             wb_ack_o<=#1 0;
90             wb_dat_o<=#1 0;
91             control_reg <= #1 32'h0;
92             cypher_data_reg <= #1 127'h0;
93             key_o <= #1 127'h0;
94             data_o <= #1 127'h0;
95           end
96           else
```

```
97          begin
98            if(ready_i)
                -2-
99            begin
100            control_reg[1] <= #1 1'b1;
101            cypher_data_reg <= #1 data_i;
102           end
103
104           if(wb_stb_i && wb_cyc_i && wb_we_i && ~wb_ack_o)
                -3-
105           begin
106             wb_ack_o<=#1 1;
107             case(wb_adr_i[7:0])
                    -4-
108                 8'h0:
109                 begin
110                     //Writing control register
111                     control_reg<= #1 wb_dat_i;
112                 end
113                 8'h4:
114                  begin
115                     data_o[127:96]<= #1 wb_dat_i;
116                 end
117                 8'h8:
118                 begin
119                     data_o[95:64]<= #1 wb_dat_i;
120                 end
121       8'hC:
122                  begin
123                     data_o[63:32]<= #1 wb_dat_i;
124                 end
125                 8'h10:
126                 begin
127                     data_o[31:0]<= #1 wb_dat_i;
128                 end
129                 8'h14:
130                 begin
131                     key_o[127:96]<= #1 wb_dat_i;
132                 end
133                 8'h18:
134                 begin
135                     key_o[95:64]<= #1 wb_dat_i;
136                 end
137                 8'h1C:
```

```
138                 begin
139                     key_o[63:32]<= #1 wb_dat_i;
140                 end
141                 8'h20:
142                 begin
143                     key_o[31:0]<= #1 wb_dat_i;
144                 end
145               endcase
146           end
147           else if(wb_stb_i && wb_cyc_i && ~wb_we_i && ~wb_ack_o)
                       -5-
148           begin
149               wb_ack_o<=#1 1;
150               case(wb_adr_i[7:0])
                      -6-
151                 8'h0:
152                 begin
153                     wb_dat_o<= #1 control_reg;
154                     control_reg[1]<=1'b0;
155                 end
156                 8'h24:
157                 begin
158                     wb_dat_o<= #1 cypher_data_reg[127:96];
159                 end
160                 8'h28:
161                 begin
162                     wb_dat_o<= #1 cypher_data_reg[95:64];
163                 end
164       8'h2C:
165                 begin
166                     wb_dat_o<= #1 cypher_data_reg[63:32];
167                 end
168                 8'h30:
169                 begin
170                     wb_dat_o<= #1 cypher_data_reg[31:0];
171                 end
172               endcase
173           end
174           else
175           begin
176               wb_ack_o<=#1 0;
177               control_reg[0]<= #1 1'b0;
```

```
BRANCH       -1-  -2-  -3-        -4-        -5-        -6-
             0    -    1   MISSING_DEFAULT   -          -          | Not Covered
             0    -    0          -          1        8'h00        | Not Covered
             0    -    0          -          1   MISSING_DEFAULT   | Not Covered
//***********


//-----------------------------------------------------------------


//*****************************************************************

//           Total Module Definition Coverage Summary

//                      TOTAL   COVERED    PERCENT
//    branches          91      86         94.51
```

D.5 cmView.short_pd File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



         SHORT PATH COVERAGE REPORT



//************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module


Test Coverage Result: Total Coverage

MODULE aes

179
180       if(decrypt_i&&round!=10)
            -1-
181         begin
182
183           addroundkey_data_i = (subbytes_data_o);
184           subbytes_data_i = (mixcol_data_o);
185           mixcol_data_i = (addroundkey_data_o);
186
187         end
188         else if(!decrypt_i&&round!=0)
                      -2-
189         begin
190
191           addroundkey_data_i = (mixcol_data_o);
192           subbytes_data_i = (addroundkey_data_o);
193           mixcol_data_i = (subbytes_data_o);
194
```

```
195         end
196         else
197         begin
198
199           mixcol_data_i = (subbytes_data_o);
200           subbytes_data_i = (addroundkey_data_o);
201           addroundkey_data_i = (data_i);
202
203         end
204
205
206         case(state)
                -3-
207
208           0:
209            begin
210              if(load_i)
                   -4-
211              begin
212
213                next_state = (1);
214
215                if(decrypt_i)
                     -5-
216                  next_round = (10);
217                else
218                  next_round = (0);
219
220                next_first_round_reg = (1);
221
222              end
223            end
224
225           1:
226            begin
227
228             //Counter
229             if(!decrypt_i&&mixcol_ready_o)
                  -6-
230             begin
231
232               next_addroundkey_start_i = (1);
233               addroundkey_data_i = (mixcol_data_o);
234               next_round = (round+1);
```

```
235
236             end
237             else if(decrypt_i&&subbytes_ready_o)
                        -7-
238             begin
239
240               next_addroundkey_start_i = (1);
241               addroundkey_data_i = (subbytes_data_o);
242               next_round = (round-1);
243
244             end
245
246             //Output
247             if((round==9&&!decrypt_i)||(round==0&&decrypt_i))
                  -8-
248             begin
249
250               next_addroundkey_start_i = (0);
251               mixcol_start_i = (0);
252
253               if(subbytes_ready_o)
                    -9-
254               begin
255
256                 addroundkey_data_i = (subbytes_data_o);
257                 next_addroundkey_start_i = (1);
258                 next_round = (round+1);
259
260              end
261
262             end
263
264             if((round==10&&!decrypt_i)||(round==0&&decrypt_i))
                  -10-
265             begin
266
267               addroundkey_data_i = (subbytes_data_o);
268               subbytes_start_i = (0);
269
270               if(addroundkey_ready_o)
                    -11-
271               begin
272
273                 next_ready_o = (1);
```

```
274                 next_state = (0);
275                 next_addroundkey_start_i = (0);
276                 next_round = (0);
277
278               end
279
280             end
281
282            end
283
284         default:
285          begin
286
287            next_state = (0);
288
289          end
290
291       endcase
292
293     end
294
295
296     //addroundkey:
297     reg[127:0] data_var,round_data_var,round_key_var;
```

FILE /home/henrikru/oo2/aes.sv

```
306
307       if(addroundkey_round==1||addroundkey_round==0)
            -1-
308         keysched_last_key_i = (key_i);
309       else
310         keysched_last_key_i = (keysched_new_key_o);
311
312       keysched_start_i = (0);
313
314       keysched_round_i = (addroundkey_round);
315
316       if(round==0&&addroundkey_start_i)
            -2-
317       begin
318
```

```
319         //Take the input and xor them with data if round==0;
320         data_var=addroundkey_data_i;
321         round_key_var=key_i;
322         round_data_var=round_key_var^data_var;
323         next_addroundkey_data_reg = (round_data_var);
324         next_addroundkey_ready_o = (1);
325
326       end
327       else if(addroundkey_start_i&&round!=0)
                -3-
328       begin
329
330         keysched_last_key_i = (key_i);
331         keysched_start_i = (1);
332         keysched_round_i = (1);
333         next_addroundkey_round = (1);
334
335       end
336       else if(addroundkey_round!=round&&keysched_ready_o)
                -4-
337       begin
338
339         next_addroundkey_round = (addroundkey_round+1);
340         keysched_last_key_i = (keysched_new_key_o);
341         keysched_start_i = (1);
342         keysched_round_i = (addroundkey_round+1);
343
344       end
345       else if(addroundkey_round==round&&keysched_ready_o)
                -5-
346       begin
347
348         data_var=addroundkey_data_i;
349         round_key_var=keysched_new_key_o;
350         round_data_var=round_key_var^data_var;
351         next_addroundkey_data_reg = (round_data_var);
352         next_addroundkey_ready_o = (1);
353         next_addroundkey_round = (0);
354
355       end
356
357     end
358
359     //sbox_muxes:
```

```
PATH         -1-  -2-  -3-  -4-  -5-
              0    1    -    -    -     | Not Covered
              0    0    1    -    -     | Not Covered
//***********


//-----------------------------------------------------------------

MODULE subbytes

 FILE  /home/henrikru/oo2/subbytes.v
 ----------------------------------

216
217       case(state)
              -1-
218
219         0:
220          begin
221            if(start_i)
                 -2-
222            begin
223
224              sbox_data_o = (data_array[0]);
225              next_state = (1);
226
227            end
228          end
229
230        16:
231          begin
232
233            data_reg_var[15]=sbox_data_i;
234            //Make shift rows stage
235            case(decrypt_i)
                   -3-
236              0:
237               begin
238                 `shift_array_to_128
239               end
240              1:
241               begin
```

```
242                `invert_shift_array_to_128
243              end
244           endcase
245
246           next_data_reg = (data_reg_128);
247           next_ready_o = (1);
248           next_state = (0);
249
250         end
251        default:
252         begin
253
254        sbox_data_o = (data_array[state]);
255           data_reg_var[state-1]=sbox_data_i;
256           `assign_array_to_128
257           next_data_reg = (data_reg_128);
258           next_state = (state+1);


PATH               -1-        -2-        -3-
                   16          -   MISSING_DEFAULT    | Not Covered
//***********


//-----------------------------------------------------------------

MODULE mixcolum

 FILE  /home/henrikru/oo2/mixcolum.v
 ----------------------------------

151
152       case(state)
              -1-
153
154         0:
155           begin
156             if(start_i)
                  -2-
157             begin
158               aux=data_i_var[127:96];
159               mix_word = (aux);
160               data_reg_var[127:96]=outmux;
161               next_data_reg = (data_reg_var);
```

```
162               next_state = (1);
163             end
164           end
165         1:
166           begin
167             aux=data_i_var[95:64];
168             mix_word = (aux);
169             data_reg_var[95:64]=outmux;
170             next_data_reg = (data_reg_var);
171             next_state = (2);
172           end
173         2:
174           begin
175             aux=data_i_var[63:32];
176             mix_word = (aux);
177             data_reg_var[63:32]=outmux;
178             next_data_reg = (data_reg_var);
179             next_state = (3);
180           end
181         3:
182           begin
183             aux=data_i_var[31:0];
184             mix_word = (aux);
185             data_reg_var[31:0]=outmux;
186             next_data_o = (data_reg_var);
187             next_ready_o = (1);
188             next_state = (0);
189           end
190         default:
191           begin


PATH               -1-        -2-
                  default       -     | Not Covered
//***********


//-----------------------------------------------------------------

MODULE wb_aes_controller

 FILE  /home/henrikru/oo2/wb_aescontroller.v
 ----------------------------------------
```

```
86      begin
87           if(reset==1)
               -1-
88           begin
89             wb_ack_o<=#1 0;
90             wb_dat_o<=#1 0;
91             control_reg <= #1 32'h0;
92             cypher_data_reg <= #1 127'h0;
93             key_o <= #1 127'h0;
94             data_o <= #1 127'h0;
95           end
96           else
97           begin
98             if(ready_i)
                 -2-
99             begin
100             control_reg[1] <= #1 1'b1;
101             cypher_data_reg <= #1 data_i;
102           end
103
104           if(wb_stb_i && wb_cyc_i && wb_we_i && ~wb_ack_o)
                -3-
105           begin
106             wb_ack_o<=#1 1;
107             case(wb_adr_i[7:0])
                    -4-
108                 8'h0:
109                 begin
110                     //Writing control register
111                     control_reg<= #1 wb_dat_i;
112                 end
113                 8'h4:
114                  begin
115                     data_o[127:96]<= #1 wb_dat_i;
116                 end
117                 8'h8:
118                 begin
119                     data_o[95:64]<= #1 wb_dat_i;
120                 end
121       8'hC:
122                  begin
123                     data_o[63:32]<= #1 wb_dat_i;
124                 end
125                 8'h10:
```

```
126                 begin
127                     data_o[31:0]<= #1 wb_dat_i;
128                 end
129                 8'h14:
130                 begin
131                     key_o[127:96]<= #1 wb_dat_i;
132                 end
133                 8'h18:
134                 begin
135                     key_o[95:64]<= #1 wb_dat_i;
136                 end
137                 8'h1C:
138                 begin
139                     key_o[63:32]<= #1 wb_dat_i;
140                 end
141                 8'h20:
142                 begin
143                     key_o[31:0]<= #1 wb_dat_i;
144                 end
145               endcase
146           end
147           else if(wb_stb_i && wb_cyc_i && ~wb_we_i && ~wb_ack_o)
                        -5-
148           begin
149               wb_ack_o<=#1 1;
150               case(wb_adr_i[7:0])
                      -6-
151                 8'h0:
152                 begin
153                     wb_dat_o<= #1 control_reg;
154                     control_reg[1]<=1'b0;
155                 end
156                 8'h24:
157                 begin
158                     wb_dat_o<= #1 cypher_data_reg[127:96];
159                 end
160                 8'h28:
161                 begin
162                     wb_dat_o<= #1 cypher_data_reg[95:64];
163                 end
164       8'h2C:
165                 begin
166                     wb_dat_o<= #1 cypher_data_reg[63:32];
167                 end
```

```
168                 8'h30:
169                 begin
170                     wb_dat_o<= #1 cypher_data_reg[31:0];
171                 end
172               endcase
173           end
174           else
175           begin
176               wb_ack_o<=#1 0;
177               control_reg[0]<= #1 1'b0;


PATH         -1-  -2-  -3-        -4-        -5-        -6-
              0    0    0          -          1        8'b0          | Not Covered
              0    0    0          -          1   MISSING_DEFAULT   | Not Covered
              0    0    1   MISSING_DEFAULT   -          -          | Not Covered
              0    1    0          -          1        8'h30         | Not Covered
              0    1    0          -          1        8'h2c         | Not Covered
              0    1    0          -          1        8'h28         | Not Covered
              0    1    0          -          1        8'h24         | Not Covered
              0    1    0          -          1        8'b0          | Not Covered
              0    1    0          -          1   MISSING_DEFAULT   | Not Covered
              0    1    1   MISSING_DEFAULT   -          -          | Not Covered
              0    1    1        8'b0         -          -          | Not Covered
              0    1    1        8'h04        -          -          | Not Covered
              0    1    1        8'h08        -          -          | Not Covered
              0    1    1        8'h0c        -          -          | Not Covered
              0    1    1        8'h10        -          -          | Not Covered
              0    1    1        8'h14        -          -          | Not Covered
              0    1    1        8'h18        -          -          | Not Covered
              0    1    1        8'h1c        -          -          | Not Covered
              0    1    1        8'h20        -          -          | Not Covered
//***********


//-----------------------------------------------------------------


//*****************************************************************

//           Total Module Definition Coverage Summary

//                       TOTAL   COVERED    PERCENT
//    paths              157     59         37.58
```

D.6 cmView.short_td File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



//       SHORT TOGGLE COVERAGE REPORT



//****************************************************************
//            MODULE DEFINITION COVERAGE

// This section contains coverage for module definitions.
// The coverage is cumulative over all the instances of the module

MODULE aes

//                                          Net Coverage

// Name                                      1->0  0->1
   keysched_sbox_decrypt_o                   No    No

//                                       Register Coverage

// Name                                      1->0  0->1
   ready_o                                   No    No
   data_o[127:0]                             No    No



MODULE sbox

//                                          Net Coverage

// Name                                      1->0  0->1

//                                       Register Coverage
```

```
// Name                                  1->0  0->1
   aA[3:1]                               No    No
   aB[3:1]                               No    No
   mul1_aA[3:1]                          No    No
   mul1_a[3:1]                           No    No
   mul2_aA[3:1]                          No    No
   mul2_aB[3:1]                          No    No
   mul3_aA[3:1]                          No    No
   mul3_aB[3:1]                          No    No
   intermediate_aA[3:1]                  No    No
   intermediate_aB[3:1]                  No    No
   inversion_aA[3:1]                     No    No


MODULE mixcolum

//                                     Net Coverage

// Name                                  1->0  0->1

//                                   Register Coverage

// Name                                  1->0  0->1
   data_reg[31:0]                        No    No
   next_data_reg[31:0]                   No    No


MODULE keysched

//                                     Net Coverage

// Name                                  1->0  0->1

//                                   Register Coverage

// Name                                  1->0  0->1
   sbox_decrypt_o                        No    No
   zero[23:0]                            No    No


MODULE wb_aes_controller
```

```
//                                          Net Coverage

// Name                                      1->0  0->1
   reset                                     No    No
   wb_adr_i[1:0]                             No    No
   wb_adr_i[31:6]                            No    No
   wb_sel_i[3:0]                             No    No
   load_o                                    No    No
   decrypt_o                                 No    No
   data_i[127:0]                             No    No
   ready_i                                   No    No

//                                       Register Coverage

// Name                                      1->0  0->1
   data_o[127:0]                             No    No
   key_o[127:0]                              No    No
   control_reg[31:3]                         No    No



//****************************************************************

//           Total Module Definition Coverage Summary

//                 TOTAL        COVERED         PERCENT
regs               156          136             87.18
reg bits           5104         4568            89.50
reg bits(0->1)     5104         4568            89.50
reg bits(1->0)     5104         4568            89.50

nets               66           58              87.88
net bits           1496         1331            88.97
net bits(0->1)     1496         1331            88.97
net bits(1->0)     1496         1331            88.97
```

D.7 cmView.mod_td File

```
// Synopsys, Inc.
//
// Generated by: cmView X-2005.06-SP2
// User: henrikru
// Date: Sat Dec 30 21:58:19 2006



//****************************************************************
//            MODULE DEFINITION COVERAGE SUMMARY


// This section summarizes coverage by providing statistics for each
// module definition. The coverage is cumulative over all the instances
// of the module


Test Coverage Result: Total Coverage

Module Name                    NetBits  NetBits              RegBits  RegBits
                               (%)                           (%)
aes                            99.85    673/674              91.10    1321/1450
sbox                           100.00   11/11                85.27    191/224
subbytes                       100.00   140/140              100.00   917/917
mixcolum                       100.00   196/196              93.59    934/998
word_mixcolum                  100.00   96/96                100.00   192/192
byte_mixcolum                  100.00   32/32                100.00   88/88
keysched                       100.00   143/143              96.82    761/786
wb_aes_controller              19.61    40/204               36.53    164/449
svaexternal                    --       0/0                  --       0/0
assert_handshake               --       0/0                  --       0/0

//****************************************************************

//           Total Module Definition Coverage Summary

//                TOTAL        COVERED         PERCENT
regs              156          136             87.18
reg bits          5104         4568            89.50
reg bits(0->1)    5104         4568            89.50
```

reg bits(1->0)	5104	4568	89.50
nets	66	58	87.88
net bits	1496	1331	88.97
net bits(0->1)	1496	1331	88.97
net bits(1->0)	1496	1331	88.97

Functional Coverage Report

For design with the following top-level modules:

- top

Design

Total number of Assertions	Assertions Not Covered	Assertions with at least 1 Real Success	Assertions with at least 1 Failure	Assertions with at least 1 Incomplete	Assertions without Attempts
9	1 (11%)	8 (88%)	0 (0%)	1 (11%)	0 (0%)

Total number of Cover Directives for Properties	Cover Directive for Property Not Covered	Cover Directive for Property with Matches	Cover Directive for Property with Vacuous Matches
1	0 (0%)	1 (100%)	0 (0%)

Total number of Cover Directives for Sequences	Cover Directive for Sequence Not Covered	Cover Directive for Sequence with All Matches	Cover Directive for Sequence with First Matches
0	0 (0%)	0 (0%)	0 (0%)

Total number of Events	Events Not Covered	Events with at least 1 real Match	Events without any match or with only vacuos match	Events without any Attempts
0	0 (0%)	0 (0%)	0 (0%)	0 (0%)

- **List of Assertions Not Covered**

Assertion	Attempts	Real Success	Failure	Incomplete
top.sva.invalid_handshake.assert_multiple_req_violation	92941707	0	0	0

- **List of Assertions with at least 1 Real Success**

Assertion	Attempts	Real Success	Failure	Incomplete
top.dut.aes.a_rounds	92941707	92941707	0	0
top.sva.a_data_read	92941707	92941707	0	0
top.sva.a_stb_ack	92941707	1727245	0	0
top.sva.invalid_ handshake.handshake_max_ ack.assert_handshake_max_ ack_cycle	92941707	1727245	0	0
top.sva.invalid_ handshake.max_ack_length_ chk.assert_handshake_max_ ack_length	92941707	1727244	0	1
top.tst.\Checker::run.a_output	200000	200000	0	0
top.tst.\Generator::decrypt_ task.unnamed$$_0.a_ randomize_d	132549	132549	0	0
top.tst.\Generator::encrypt_ task.unnamed$$_0.a_ randomize_e	133097	133097	0	0

- **List of Assertions with at least 1 Failure**

- **List of Assertions with at least 1 Incomplete**

Assertion	Attempts	Real Success	Failure	Incomplete
top.sva.invalid_ handshake.max_ack_length_ chk.assert_handshake_max_ ack_length	92941707	1727244	0	1

- **List of Assertions without Attempts**

- **List of Cover Directive for Sequence Not Covered**

- **List of Cover Directive for Sequence with All Matches**

- **List of Cover Directive for Sequence with First Matches**

- **List of Cover Directive for Property Not Covered**

- **List of Cover Directive for Property with Matches**

Cover Directive for Properties	Attempts	Matches	Vacuous Matches	Incomplete
top.dut.aes.c_rounds	92941707	92941707	0	0

- **List of Cover Directive for Property with Vacuous Matches**

- **List of Events Not Covered**

- **List of Events with at least 1 real Match**

- **List of Events without any match or with only vacuos match**

- **List of Events without any Attempts**

More Detailed Reports

- List of tests merged to generate this report
- Hierarchical coverage report
- Category based coverage report

Report Generation Info

Report generated by "henrikru" at Sat Dec 30 21:57:45 2006 , with following cmdline:

  - `assertCovReport`

- VCS_HOME: `/dak_install2/synopsys/2005.06-SP2`
- VCS Version: `X-2005.06-SP2`

Appendix E

Signal Waves and Screenshots

E.1 Screenshots

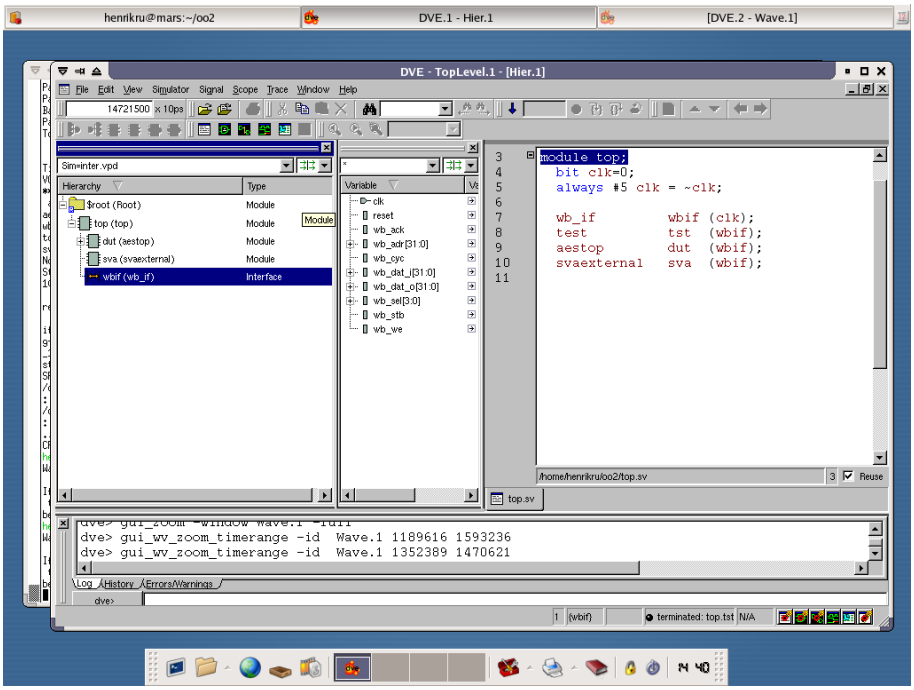

Figure E.1: DVE screenshot

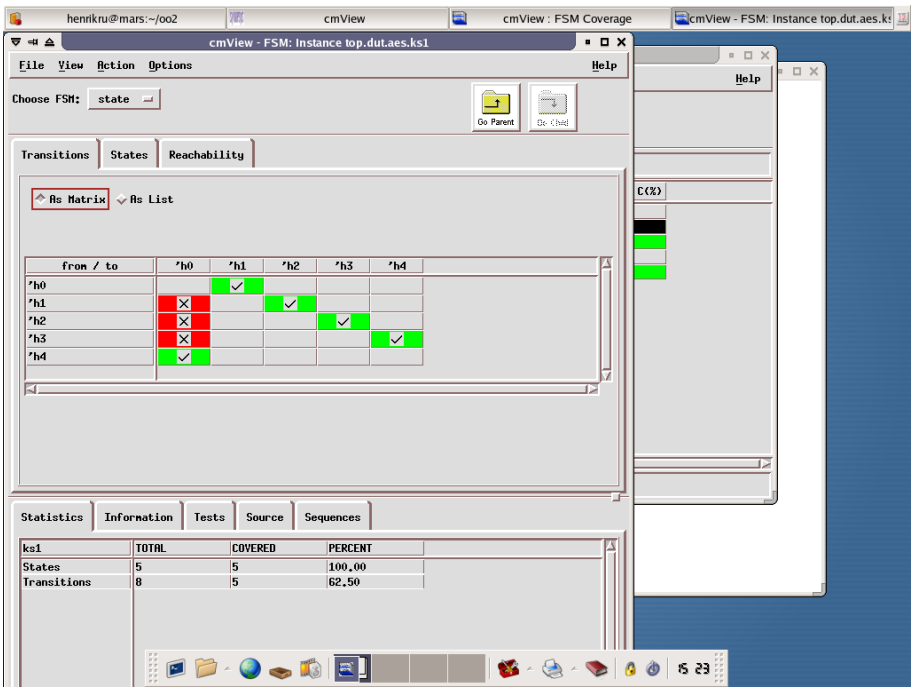

Figure E.2: cmView screenshot

E.2 Signal Waves

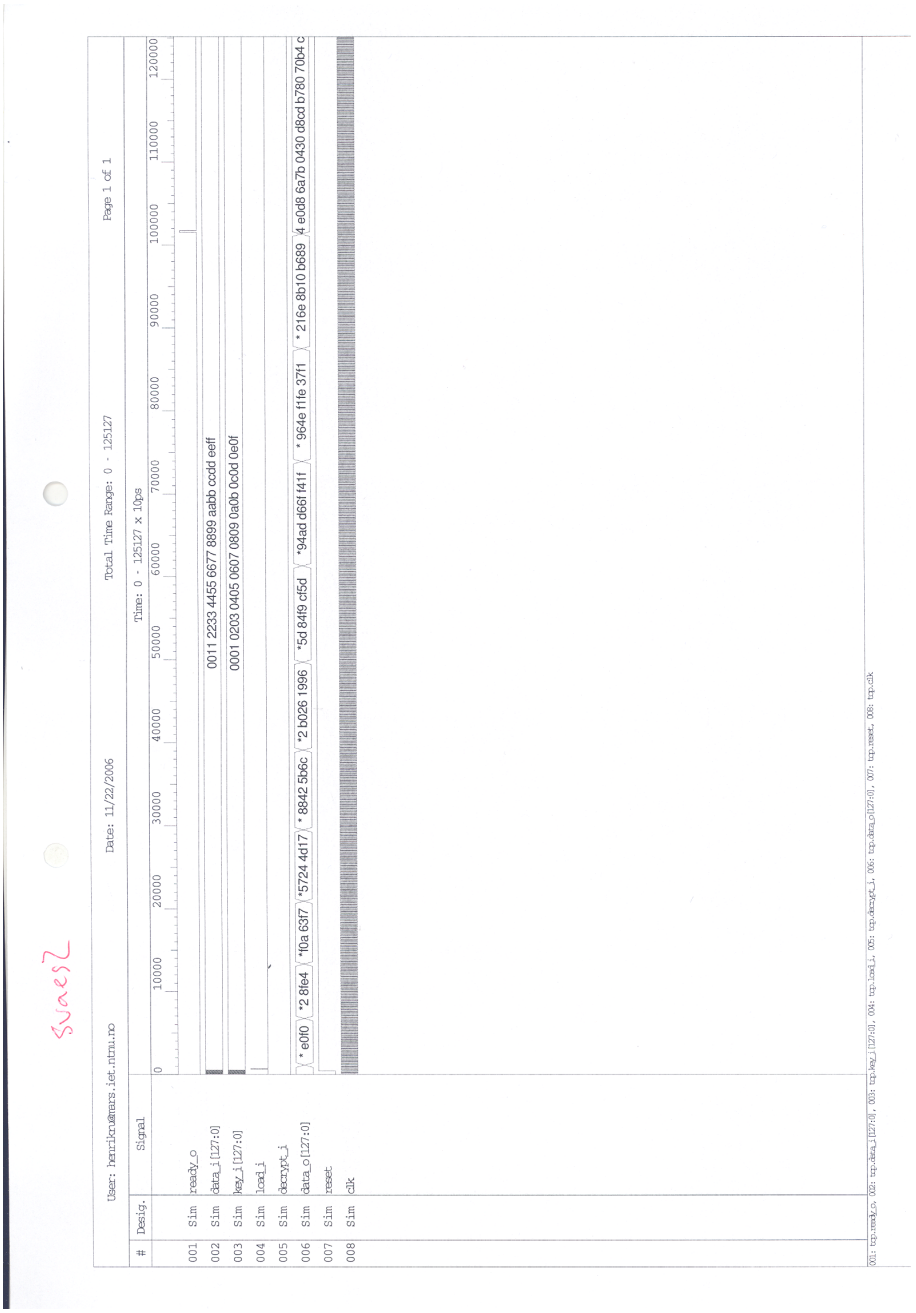

Figure E.3: AES module signal waves

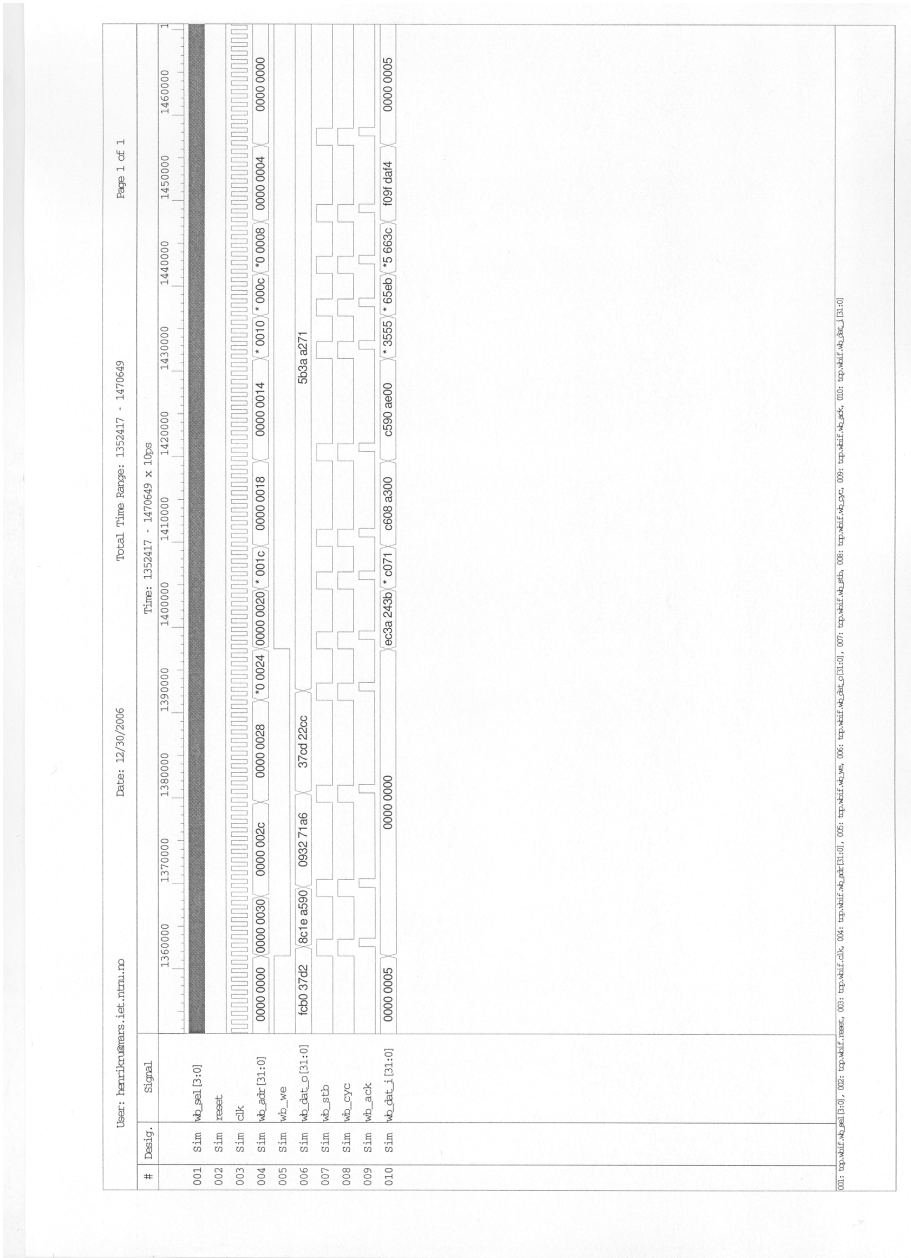

Figure E.4: Wishbone write operation