Copyright
by
Kshitiz Gupta
2017

**The Thesis Committee for Kshitiz Gupta**
**certifies that this is the approved version of the following thesis:**

# Automatic generation of coverage directives targeting signal relationships by statically analyzing RTL

APPROVED BY

SUPERVISING COMMITTEE:

Jacob Abraham, Supervisor

Mark McDermott

# Automatic generation of coverage directives targeting signal relationships by statically analyzing RTL

by

**Kshitiz Gupta, B.Tech.**

**THESIS**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**MASTER OF SCIENCE IN ENGINEERING**

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2017

Dedicated to my family

# Acknowledgments

I sincerely thank my advisor, Professor Jacob Abraham, for his guidance and support throughout this thesis. His enthusiasm is contagious and his advice is invaluable. A special thanks for Mark for his comments in the final versions of the thesis. I would also have to thank my colleagues in the CERC lab – Tosin, Rajesh and Erick – for the environment of constant discussion (and the occasional drinks) in the lab. I am grateful to my fellow graduate students Ronak Oswal, Harsh Yadav and Udit Dasgupta as for their constant support as we all navigated through graduate school. My friends from undergraduate – Amber, Aditya, Utkarsh, Sumeet – have always been constant supports and have played a big role in my progress over the years.

Finally, a note of profound gratitude for my parents, who have always supported my interests and guided me over the years as I pursue them. I would be nothing without them.

**Automatic generation of coverage directives targeting signal relationships by statically analyzing RTL**

Kshitiz Gupta, M.S.E.
The University of Texas at Austin, 2017

Supervisor: Jacob Abraham

The coverage problem has been a long standing issue in simulation-based verification. Coverage metrics are required to track the progress and justify completeness of simulation vectors. This thesis presents a scalable methodology to automatically generate coverage directives which augment line coverage for complete coverage of the RTL. The directives target ambiguity in register relationships derived by statically analyzing behavioral RTL. A Python-based tool has been built on the presented methodology to generates implication properties for coverage. The generated properties for two cores (Amber and V-scale) are instantiated within the testbenches provided with them. Simulation results with all the test programs highlight module instances with high line coverage but uncovered properties. These properties are formally proven to be reachable, thus highlighting coverage holes with the provided tests and the usefulness of our process.

# Table of Contents

**Acknowledgments** **v**

**Abstract** **vi**

**List of Tables** **ix**

**List of Figures** **x**

**Chapter 1. Introduction** **1**

**Chapter 2. The Coverage Problem** **3**

2.1 Basic code coverage metrics . . . . . . . . . . . . . . . . . . . . . 3
2.2 Coverage metrics . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.3 Automatic coverage generation . . . . . . . . . . . . . . . . . . . . 6
2.4 Tool Support . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.5 Uses of coverage metrics . . . . . . . . . . . . . . . . . . . . . . 7
2.6 Final Note . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

**Chapter 3. Generating Cover Properties** **9**

3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Analyzing a Verilog Module . . . . . . . . . . . . . . . . . . . . 10
3.3 Coverage goals . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.4 Wishbone as an example . . . . . . . . . . . . . . . . . . . . . . 15
3.5 Reducing the number of properties - Identifying state registers 17
3.5.1 State registers in Wishbone . . . . . . . . . . . . . . . . 19
3.5.2 State registers with combinational *always* blocks . . . . 20
3.6 Discussion about the property synthesis approach . . . . . . . 21
3.7 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.7.1 Introduction to Pyverilog . . . . . . . . . . . . . . . . . . 23

3.7.2 Finding state registers . . . 27
3.7.3 Writing cover properties . . . 28
3.7.4 Other salient features . . . 29

**Chapter 4. Results** **31**
4.1 Tool Results . . . 31
4.2 Case study : V-scale . . . 33
4.2.1 Multiply and divide unit . . . 34
4.2.2 CSR file . . . 35
4.2.3 Discussion . . . 35
4.3 Case study: Amber ARM . . . 37
4.3.1 Multiply unit . . . 37
4.3.2 Decode unit . . . 38
4.3.3 Wishbone unit . . . 38
4.3.4 Discussion . . . 39
4.4 Summary . . . 40

**Chapter 5. Conclusions and future work** **42**

**Appendices** **45**

**Appendix A. 2-way arbiter RTL** **46**

**Appendix B. Cover directives generated for V-scale** **49**

**Appendix C. Cover directives generated for Amber ARM** **53**

**Bibliography** **74**

# List of Tables

4.1 Preliminary results on selected designs . . . . . . . . . . . . . . 32
4.2 Results on V-scale . . . . . . . . . . . . . . . . . . . . . . . . 33
4.3 Results on Amber ARM . . . . . . . . . . . . . . . . . . . . . . 37

# List of Figures


3.1 Two always blocks in the same module . . . . . . . . . . . . . . 13
3.2 Deterministic Dependence . . . . . . . . . . . . . . . . . . . . . . 13
3.3 Non-deterministic dependence . . . . . . . . . . . . . . . . . . . . 15
3.4 Wishbone RTL . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
3.5 State and data registers . . . . . . . . . . . . . . . . . . . . . . . 19
3.6 Combinational always blocks . . . . . . . . . . . . . . . . . . . . . 21
3.7 Python-based tool flow . . . . . . . . . . . . . . . . . . . . . . . . 24
3.8 Dataflow Analyzer of Pyverilog - (RTL and generated graph) . 26

4.1 Code snippet from V-scale (md unit) . . . . . . . . . . . . . . . 34
4.2 Code snippet from V-scale (csr file) . . . . . . . . . . . . . . . . 35

# Chapter 1

# Introduction

The increasing size and complexity of modern microprocessors presents critical verification challenges in the design process. Verification engineers need to deal with an exponentially increasing state space when checking for correctness. It has been observed that more than 70% of development time goes in verification [20]. In this thesis, we will be focusing on simulation-based verification and the challenges this presents in coverage measurement. The present formal-based approaches do not scale well with new designs due to the state explosion problem [25]. As a result, simulation based approaches dominate the verification strategies used for modern designs. The advent of environments such as UVM [2] also brings scalability and reusability to the simulation environment. The main steps in simulation-based verification are test generation and response evaluation. Since it is not practical to check for all possible input stimuli, coverage models are essential to quantify the simulations (test vectors) for answering the all important question – "When are we done?". Many metrics have been proposed (as well as borrowed from software testing) and yet it remains an open problem.

In this thesis, we propose a coverage metric which we argue is easy

to understand, scalable and focuses on exercising the ambiguity in the control flow of the design. We take the approach of statically analyzing the given RTL and automatically generating coverage directives which target the ambiguity in signal relationships. The goal of these coverage metrics is to ensure complete coverage of RTL. We target cases which have some form of ambiguity, which makes line coverage ineffective in covering them. We propose the use of these directives alongside line coverage.

A Python-based tool has been built which synthesizes SystemVerilog [7] cover properties by static analysis of RTL (Verilog [17] HDL). It can be integrated with any simulation platform. These are the preferred languages for design and verification in the industry today. The generated properties target cases which traditional line coverage overlooks and are geared towards exercising signal relationships in the RTL.

The thesis is structured as follows. In Chapter 2, we look at the the coverage problem and list some metrics currently used to quantify coverage. We introduce our method in Chapter 3, where we look at the guiding principles of our approach as well as implementation details of a Python-based tool developed for analysis. Chapter 4 presents some results obtained by testing the Python-tool with different designs which will highlight both the benefits and potential drawbacks of our approach. We conclude the thesis and present some ideas on how this work can be extend in Chapter 5.

# Chapter 2

# The Coverage Problem

The success of simulation-based verification depends heavily on the quality of the test vectors. Coverage models are important to a simulation environment as they are used to measure progress and ensure complete testing of the DUV (Design Under Verification). In this chapter, we look at some major coverage measurement techniques that have been proposed over the years before we present our metric in Chapter 3.

## 2.1 Basic code coverage metrics

Coverage metrics can be broadly classified into two major categories, code coverage and functional coverage [1]. Code coverage metrics focus on ensuring that the implementation of the design (RTL) is thoroughly exercised by the simulation vectors. Some code coverage metrics such as line coverage, branch coverage etc., have been borrowed from software testing [32, 40]. The basic code coverage metrics [1, 36, 42, 53] are enumerated below:

1. **Line/Block/Statement coverage** – These metrics report which lines/statements of the RTL have and have not been executed. This metric ensures that every line in RTL should be executed a number of times

specified by a user. It is one of the most widely used metric to measure coverage.

2. **Branch/Decision coverage** – This metric targets the control flow and ensures each decision outcome (eg., both True and False for an if-else) has been tested.

3. **Path coverage** – Path coverage is a more complete control flow metric which reports the number of times every possible path through the code was executed.

4. **Condition coverage** – Condition coverage tests whether each permutation of the terms in the condition has occurred. For example, to get complete condition coverage for `(A&&B)||C`, all 8 combinations starting from (!A,!B,!C), (!A,!B,C) all the way to (A,B,C) need to exercised.

5. **Toggle coverage** – Toggle coverage aims to ensure that every bit of a register or a wire toggles its value.

6. **State coverage** – State coverage targets the extracted FSMs from RTL. State coverage of an FSM ensures that all the states in the FSM have been visited.

7. **Transition coverage** – This also targets the extracted FSMs and ensures that all transitions (arcs) in the extracted state machine are exercised.

The above metrics come with their share of drawbacks. Hardware designs have multiple communicating processes as interacting FSMs, and in such cases line coverage, branch coverage and condition coverage metrics are not complete for making sure all the possible signal interactions have been exercised. Path coverage on the other hand is a very stringent requirement as the number of paths can be exponential. Coverage metrics like toggle coverage are indirect metrics at best and do not help in identifying the coverage of the control behavior. Full state and transition coverage are not scalable metrics as the number of states and arcs in a design can be exponential.

## 2.2 Coverage metrics

The authors in [33, 34] build a global state graph and then identify important control events for coverage. They attempt to control the size of the graph using heuristics. In [35, 41], the authors use input from the designer about the important state registers to construct an extracted control flow machine (ECFM). The coverage of the test vectors is then measured as the state and transition coverage of the ECFM. The work done in [45, 46] uses heuristics to identify important control states and generate path tours to be used for coverage measurement. The major drawback of these approaches is use of some form of explicit FSM extraction, which challenges practical use of such approaches. The authors in [26, 29] propose tag coverage as a metric. The calculus presented allows for tag propagation with every test vector and coverage is computed as a percentage of the total number of tags. The authors

of [37] propose the TRIO (Input/Output TRansition) fault model to be used for coverage measurements. They propose this model for use in functional test selection.

## 2.3 Automatic coverage generation

Researchers in [54] propose an approach to generate coverage models from a CTL [24] specification. The authors of [27, 28] talk about a method to automatically generate coverage models by static analysis of the RTL and further mining the simulation traces. In [55], the authors generate coverage models in the form of coverage groups by analyzing behavioral RTL. The authors in [38] automatically extract a Semantic Finite State Machine Model (SFSM) from the HDL by grouping states with same behavior into one *semantic* state. They use that model to define state and transition coverage on the design. The work done in [48] extracts corner-case coverage metrics from a formal specification written for interface protocols. The researchers in [44] automatically generate functional coverage metrics from a specification written in SDL [50]. The authors of [43] propose the use of HLDDs for efficient computation of code coverage metrics. The authors of [19] talk about code coverage of assertions and proposes a method which uses both static and dynamic analysis of RTL to measure correctness based coverage. The authors in [18] talk about computing coverage for emulation/prototyping platforms like FPGAs by analyzing the source code.

## 2.4 Tool Support

Modern languages support coverage gathering by way of specifying covergroups and cover properties [7] (terminology for SystemVerilog) and tools have built-in support for collecting code coverage metrics. Tools from all major EDA companies – Cadence [11], Mentor [14] and Synopsys [15] – provide formal tools to handle coverage and even check for reachability. Cadence JasperGold provides engines like SPS (structural property synthesis) and BPS (behavioral property synthesis) [11] which can generate SystemVerilog assertions and cover properties by analyzing RTL. These properties are targeted at checking for deadlocks in FSMs, dead-code detection, arithmetic checks and state/transition coverage for detected FSMs. Mentor Questa AutoCheck and Property Generation engines support similar functionality as well [14].

## 2.5 Uses of coverage metrics

Though intended for purposes of measuring progress and completeness of test vectors, coverage metrics have now found use in other steps of the verification cycle. Coverage-driven verification is the most commonly used procedure for simulation testing. Significant work has also been done for incorporating coverage measurement with stimulus generation, by way of coverage directed test generation [21–23, 31, 51]. The use of data mining on coverage data to assist in test generation has been presented in [30]. The authors in [47] use a coverage guided mining algorithm for generating assertions for a design. Use of coverage metrics for formal verification has also been proposed [16, 49].

The work done in [37] talks about using coverage metrics for functional test selection for testing.

## 2.6 Final Note

In this chapter, we established the coverage problem and looked at different methods for coverage measurement. We also mentioned about the use of coverage models for coverage-driven verification and coverage-directed test generation. In the next chapter, we look at our proposed metric and how it helps alleviate the problems mentioned here.

# Chapter 3

# Generating Cover Properties

In this chapter, we will look at out proposed coverage metrics in detail before discussing a Python based implementation of the presented methodology.

## 3.1 Motivation

The major goals of this work can be expressed as follows:

1. We want to look at metrics which are more expressive than line coverage, while at the same time avoid the exponential nature of techniques such as state enumeration, symbolic execution and FSM extraction.

2. We want to use static RTL analysis for our metrics. The RTL model contains the detailed description of the design's functionality. Also, during implementation a designer may take decisions which may not be reflected well in coverage metrics built purely from specifications. We argue against analyzing simulation traces as there is uncertainty of ever hitting a corner case.

3. Automating coverage metrics can be a key for reducing verification time.

4. Coverage metrics should ensure that the RTL description has been thoroughly exercised.

5. Covering the control flow of the design is important, as that is where most of the bugs tend to be. Since only a subset of variables control the data-path, focussing on control interactions is crucial.

Overall, we are looking for metrics which are automatic, can be generated by static analysis of the RTL, avoid state-explosion and focus on the control flow of the design. Keeping the above pointers in mind, we look at property synthesis by static analysis of the RTL. We will focus on writing SystemVerilog cover properties by analyzing RTL written in Verilog HDL. The SystemVerilog properties can be easily integrated with any modern simulator to provide coverage goals.

## 3.2 Analyzing a Verilog Module

We will be looking at a *module* in Verilog HDL to write SystemVerilog cover properties. We assume that the RTL is written in the synthesizable subset of Verilog.

Consider an *always* block. We can say that an *always* block is a union of assignment statements (blocking or non-blocking) and the control structure around these assignments (if–else conditions, etc.). We name three sets of signals in an *always* block. The variables (**regs**) on the left-hand side of the assignments are the **output** variables of the block. The output variables can

also be thought of as state variables of that particular *always* block. The variables on the right-hand side of the assignments are called **Data inputs** and the variables in the predicates of the control structures are called the **Control inputs**.

We choose to visualize each *always* block as an independent FSM, which is modifying its output variables based on the current values of the data and control inputs. This way, the *always* blocks may be connected to each other, with the output variables of one *always* block as the control/data input for another. The entire *module* can be thought of as a set of interacting FSMs.

For writing meaningful cover properties for the *module*, we focus on those which cross *always* block boundaries. That means, we will consider cases where an output variable of some *always* block (say Always#1) toggling/changing leads to a change in the value of a variable in another block (say Always #2). To target control flow, we will consider only the control inputs set of Always #2, i.e., a state variable in Always#1 affects the control flow of a state in Always#2. It can also be thought of as writing *cause-effect* relationship properties. These will translate to implication properties, with the antecedent being from Always#1 and consequent from Always#2.

We look at the intersection between two *always* blocks and how coverage fits in the next section.

## 3.3 Coverage goals

Consider the simplest coverage metric – Line coverage. It is easy to measure over simulation traces. However, line coverage suffers from some known drawbacks[53]. 100% line coverage is known to be incomplete as it does not take into account the highly concurrent nature of hardware. We will look at an example to demonstrate this shortly.

On the other end of the spectrum, let us define control path coverage. Control path coverage attempts to look at all possible *paths* that can be taken by the code when executing. 100% control path coverage will be a cross-product of all the paths that all the *always* blocks individually can take. Full control path coverage includes all possible ways the RTL can be executed and therefore, we argue that it covers the control flow of the design. In this section, when we say path coverage, we mean control path coverage. It is also a superset of metrics like line coverage and branch coverage. However, the possible number of such paths can be exponential. Such a metric would not scale well with designs. In this thesis, instead of taking combinations over all the *always* blocks, we will look at the subset of all pairs of *always* blocks. Therefore, we will look at pairs of *always* blocks with the motivation of writing *cause-effect* properties to get better coverage.

Consider a case depicted in Fig. 3.1. These are code snippets from two different *always* blocks in the same module. To get complete line coverage, 2 tests would be enough : $(x = 0, y = 0)$ and $(x = 1, y = 1)$. However, to get complete path coverage, we need 4 tests : $(x = 0, y = 0)$, $(x = 0, y = 1)$,

```
//Always#1
//always@(*)
if(x==0) begin
       ...
end
else begin
       ...
end
```

```
//Always#2
//always@(*)
if(y==0) begin
       ...
end
else begin
       ...
end
```

Figure 3.1: Two always blocks in the same module

```
//Always#1
//always@(x)
if(x==0) begin
   ...//Block 2
end
else begin
   ...
end
```

```
//Always#2
//always@(y)
if(y==0) begin
   x=1;
   ...
end
else begin
   x=0;//Block 1
   ...
end
```

Figure 3.2: Deterministic Dependence

$(x = 1, y = 0)$ and $(x = 1, y = 1)$. This example attempts to highlight how complete line coverage is not enough for complete path coverage. Let us probe further in this example to better explore path coverage.

1. Consider a case where no signal in Always#1 is in any way connected to any signal in Always#2. Simply put, the signals $x$ and $y$ are not in each others' cone of influence. In such a case, the cross product is redundant. Path coverage is not required as the extra test cases we run to get complete path coverage between the two blocks cannot provide us with any new information that complete line coverage will not.

2. Consider another case where the two blocks are not independent. That is, $x$ and $y$ have some correlation. We can divide this case into 2 sub-cases. We call the first case as *deterministic dependence*. Consider Fig. 3.2 where $x$ and $y$ are completely linked. In this case, complete path coverage is not allowed by the code due to the dependence between the blocks. In this situation, complete line coverage is enough to ensure complete path coverage(considering only allowable cases). This is because `Block 2` can only execute when `Block 1` executes.

3. The second case is referred to as *non-deterministic dependence*. In this case (Fig. 3.3), we can see that though $x$ and $y$ have some dependence on each other, it is different from the above case as complete line coverage will not exhaust all the possible paths in the RTL. Cases can be built such that complete line coverage can be achieved without complete path coverage. For example, consider the following tests : $(x = 0, a = 0)$ followed by $z = 1$ and $(x = 1, a = X)$ followed by $z = 0$. All the lines have been covered through such a test, however the path arising when $(x = 0, a = 0)$ followed by $z = 0$ has been left uncovered. This case highlights the incompleteness of line coverage as a metric and this is where we need path coverage. We identify the reason of this ambiguity being the assignment of $y <= 0$ at two different locations, thus making the dependence non-deterministic.

We conclude with this discussion that control path coverage can include a lot of redundancy. A cross product between a set of paths should only be taken

```
//Always#1
//always@(posedge clk)
if(x==0) begin
   if(a==0) begin
       y<=0;
   end
end
else begin
   y<=0;
end
```

```
//Always#2
//always@(*)
if(z==0) begin
   if(y==0) begin
   ...
   end
end
else begin
   ...
end
```

Figure 3.3: Non-deterministic dependence

if they have some dependence on each other. At the same time, it must be ensured that the dependency is non-deterministic, or else line coverage can cover those cases.

As established in the previous section, we will be focusing on writing properties across *always* blocks. Combining it with the above observation, we can direct our tool to only write properties which have some ambiguity in the *cause* assignment (antecedent). **These properties can be used together with traditional line coverage to achieve good control path coverage.**

## 3.4 Wishbone as an example

We consider the *wishbone* module of the Amber ARM core available in [4]. Consider the RTL code in Fig. 3.4. The register *wishbone_st* has been used across the code in predicates and is connected with the control flow of the module. We see it being assigned the value of the parameter *WB_WAIT_ACK* twice in Always#2 for two different conditions. Now consider Always#1 where

*o_wb_dat* is assigned some value if *wishbone_st* is *WB_WAIT_ACK* along with some other conditions. It is possible to get 100% line coverage in this situation without necessarily exercising both the situations where *o_wb_dat* is assigned *extra_write_r* due to each of the wishbone_st assignments here. Therefore, additional coverage points are required, which we will present in the form of SystemVerilog cover properties.

We analyze wishbone_st and see that it is assigned *WB_WAIT_ACK* for two conditions :

`((wishbone_st == WB_IDLE) && (wait_write_ack))` and

`((wishbone_st==WB_IDLE) && (!wait_write_ack) && (dcache_cached_rreq_c || dcache_uncached_rreq_c) && (!dcache_read_qword_c))`.

We also analyze other registers and note that `((quick_n_reset)&&(wishbone_st == WB_WAIT_ACK && i_wb_ack && extra_write_r))` will lead to the assignment. Therefore, we write two SystemVerilog cover properties as follows:

```
cover property: (@(posedge i_clk) ((wishbone_st==WB_IDLE)
   &&(wait_write_ack)) |->##1 ((quick_n_reset)&&(
   wishbone_st == WB_WAIT_ACK && i_wb_ack &&
   extra_write_r)));
cover property: (@(posedge i_clk) ((wishbone_st==WB_IDLE)
   &&(!wait_write_ack)&&(dcache_cached_rreq_c ||
   dcache_uncached_rreq_c) && (!dcache_read_qword_c))
   |->##1 ((quick_n_reset)&&(wishbone_st == WB_WAIT_ACK
   && i_wb_ack && extra_write_r)));
```

We are writing implication properties representing a cause-effect relationship between two registers in the design, targeting the effect on the control flow. The antecedents are the cause conditions, leading to an effect on control flow

of another register. To ensure that the effect is covered, consequents are made from the conditions surrounding the effect assignment. The tool will analyze the RTL statically to create these relationships, generate the conditions and match them to write SystemVerilog properties.

## 3.5 Reducing the number of properties - Identifying state registers

It can be argued that the total number of *cause-effect* properties for all pairs of registers in a *module* can grow to be quite large. Therefore, we proceed to identify some important **state registers** as a subset of all the registers which may have the most effect on the control flow of a *module*. We then consider only these state registers as candidates for the *cause* part of the properties.

We avoid explicit state enumeration or FSM extraction to identify important registers. Instead, we rely on a heuristic to do the job. We consider a model of a Verilog *module* as illustrated in Fig. 3.5. In such a model, some registers in the design would be storing the history of the *module*, i.e. the value of some registers depends directly on their previous value (will be referred to as **state registers**) whereas for other registers, their current value is computed fresh every cycle (will be referred to as **data registers**). Therefore, we define state registers as a subset of all the registers in the design which affect the value of the same register in the next cycle. Since we are focusing on control flow, we will look for effect through control in the next cycle. **Data registers** can

```
always @( posedge i_clk or negedge quick_n_reset) begin
    //Always#1
    if ( !quick_n_reset)
        o_wb_dat <= 'd0;
    else if ( wishbone_st == WB_WAIT_ACK && i_wb_ack &&
       extra_write_r )
        o_wb_dat <= extra_write_data_r;
    else if ( start_access )
        o_wb_dat <= i_dcache_write_data;
end

always @( posedge i_clk or negedge quick_n_reset) begin
    //Always#2
    ...
    case ( wishbone_st )
        WB_IDLE :
            begin
            if ( wait_write_ack )
                begin
                wishbone_st <= WB_WAIT_ACK;
                end
            else if ( dcache_cached_rreq_c ||
               dcache_uncached_rreq_c )
                begin
                if ( dcache_read_qword_c )
                    wishbone_st <= WB_BURST1;
                else
                    wishbone_st <= WB_WAIT_ACK;
                end
            ...
    endcase
end
```

Figure 3.4: Wishbone RTL

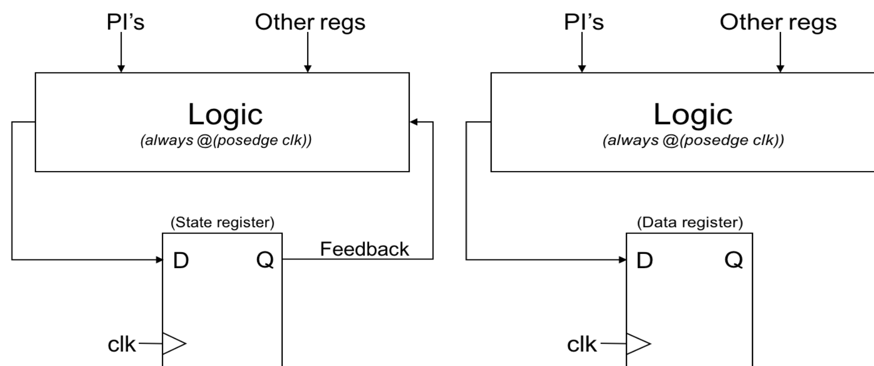



Figure 3.5: State and data registers

also depend on the history of the *module*. However, this dependency would be indirect as it would go through the aforementioned *state registers*. This heuristic is similar to the method use in Mentor Questa SIM to find FSMs in a design [12]. It is also similar to the work done in [39]. The wishbone *module* from Amber ARM [4] has been considered again as an example in the next section.

### 3.5.1 State registers in Wishbone

Analyzing the *wishbone module* in ARM Amber [4] by the above heuristic identifies 4 registers - `wishbone_st, extra_write_r, dcache_cached_rreq_r` and `dcache_uncached_rreq_r`. These registers are responsible for maintaining a *state* of the module, as they store in them some form of the history of the module's execution.

In a module written purely with clocked *always* blocks, the state registers can be identified by the intersection of the *output* and *control* signals.

However, when writing Verilog HDL with combinational *always* blocks, such an approach will not work. We look at that in the next section.

### 3.5.2 State registers with combinational *always* blocks

Consider the arbiter code in Appendix A. It is clear that the register *state* is important, however the first order analysis presented in the previous sections will not be able to identify this register. We need to expand our model to accommodate for such cases (as depicted in Fig. 3.6). The approach we follow is to continue exploring the assignment values till we reach a clocking event. The combinational *always* blocks will eventually be evaluated at some clocking event in practise, and we follow the assignments till such an event is reached. It can be argued that we are effectively taking the combinational block and analyzing the code as if that code were part of the clocked *always* blocks.

Following the above idea, we start with the combinational *always* block in the design. We see *state* in control and *next_state* as an output signal. We follow the signal *next_state* and reach the clocked *always* block, where *next_state* is a control signal and *state* is an output signal. We have terminated the process at a clocking event and conclude that *state* is an important register.

The basic approach has been presented. We will present a short discussion about the presented methodology before talking about a Python based implementation in Section 3.7.

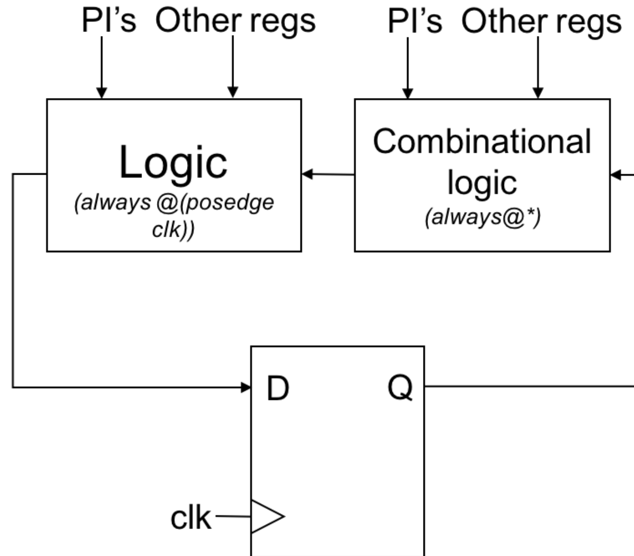



Figure 3.6: Combinational always blocks

## 3.6 Discussion about the property synthesis approach

We present a comparison of the proposed cover property synthesis method with other coverage metrics.

1. **Line coverage** – As we have established already, the proposed cover properties will work to assist line coverage (or statement coverage). By definition, the proposed metrics target ambiguity in the assignments, and this would cover more cases that line coverage. It can also be shown that if we write such properties for all the registers (and not just the state registers as mentioned in Section 3.5), all control predicates in the module will be covered, thus covering line coverage.

2. **Control path coverage** – The proposed cover properties attempt to

reduce the number of paths to be covered by identifying redundant and infeasible paths.

3. **Toggle Coverage** – Metrics like toggle coverage and bit coverage are indirect metrics at best. They do not actually help in identifying how much of the control behavior has been covered. Our proposed metric is directly associated with the control flow of the design and signal relationships, and thus would better in identifying holes in control flow coverage.

4. **Other Property synthesis approaches** – The properties generated by tool engines [11, 14] are first order checks for deadlocks and reachability. These properties do not attempt to exercise the control flow of the design or stress the corner case behavior. However, it should be noted that our proposed properties can easily be integrated into these tools.

The proposed cover properties are attempting to highlight states in the design by static analysis of the RTL. They are trying to identify corner-case execution paths which will not be covered by traditional code coverage metric. Every property attempts to target a particular state that the module can be in. A benefit of our approach is the scalability of our heuristics. All the methods proposed in the previous sections will by polynomial in both time and memory for the number of lines of RTL.

One expected drawback of this approach is due to unreachable states. Since we are looking for ambiguous signal interactions, it is possible that some

of the interactions we discover are not meant to be executed and the conditions around the design are such that they can never be executed. The states that some properties identify can be unreachable states. We used the formal tool Cadence JasperGold on the properties generated by our Python-based tool and found some unreachable properties. We address this issue in Chapter 4 with the help of case studies for more details and experimental evidence.

## 3.7 Implementation

We talk about a Python-based approach based on the above discussions in this section. Fig. 3.7 presents a brief overview of the flow. Pyverilog [52] is an open source tool available as a Verilog processing toolkit. Our tool has been built over Pyverilog as we have added to and modified this package. We start by giving a brief introduction to this package and the components we will be using.

### 3.7.1 Introduction to Pyverilog

Pyverilog [52] is an open-source toolkit for RTL analysis and code generation of Verilog HDL. It consists of 4 key engines: parser, dataflow analyzer, control-flow analyzer and a Verilog code generator.

- **Parser** – The parser engine analyzes Verilog HDL to generate an abstract syntax tree (AST) which is used later for analysis. It uses Icarus Verilog [10] as a preprocessor and PLY (Python Lex-Yacc) [13] for the main compiler.

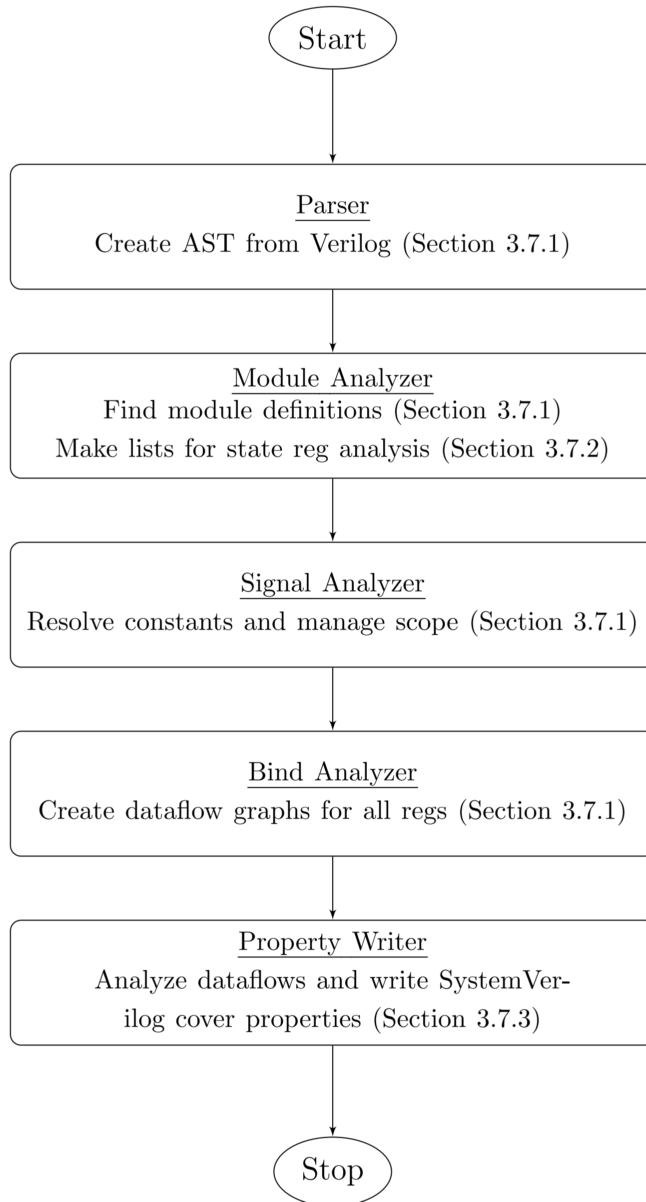
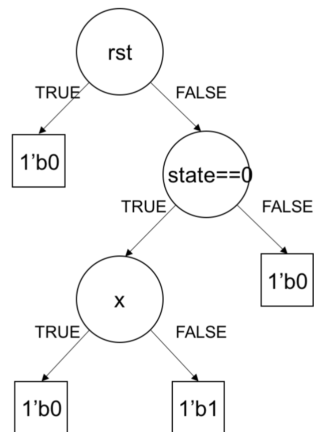



Figure 3.7: Python-based tool flow

- **Dataflow Analyzer** – The dataflow analyzer constructs a dataflow graph representing the relationship among signals by analyzing the AST generated by the parser. A Verilog code snippet and the extracted dataflow graph has been presented in Fig. 3.8. Both the graphical and textual representations of the graph have been shown in Fig. 3.8. The analyzer itself has three subsystems - *module analyzer*, *signal analyzer*, and *bind analyzer*. These subsystems have been implemented using a standard visitor pattern where a specific function for each visited AST node is called by its class name recursively. The module analyzer traverses the input AST to build a list of modules and their inputs, outputs and parameters. The signal analyzer traverses the AST tree again to analyze all the signal declarations and associate them with a scope. The constants are also resolved in this step. Finally, the bind analyzer generates a dataflow graph for each signal, i.e., a tree consisting of arithmetic and logical operators, assignment conditions (branches) and assigned values. The dataflow analyzer has been heavily modified and new engines have been added around it to support our approach.

- **Control-flow analyzer and code generator** – The analyzer uses pattern-matching to identify the control flow. To keep the control flow detection independent of variable names, the control-flow engine has not been used. A code generator engine has also been provide which can use an AST to write Verilog HDL. Since our goal is to write SystemVerilog cover properties, this engine has also not been used.

```
if(rst) begin
   state<=0;
end
else begin
   if(state==1'b0) begin
       if(x) begin
           state<=1'b0;
       end
       else begin
           state<=1'b1;
       end
   end
   else begin
       state<=1'b0;
   end
end
```




```
(Bind dest:TOP.state tree:(Branch Cond:(Terminal TOP.rst)
True:(IntConst 0) False:(Branch Cond:(Operator Eq Next:(Terminal
TOP.state),(IntConst 1'b0)) True:(Branch Cond:(Terminal TOP.x)
True:(IntConst 1'b0) False:(IntConst 1'b1)) False:(IntConst 1'b0))))
```

Figure 3.8: Dataflow Analyzer of Pyverilog - (RTL and generated graph)

### 3.7.2 Finding state registers

As mentioned in Section 3.5, we will search for a subset of all registers called the *state registers* which are important for the control flow of the design. We integrate this with the *module analyzer* component of the dataflow analyzer. For each *always* block in every module, we maintain three lists - data, control and state - referring to the definitions given in Section 3.2.

- When parsing the AST, the three lists are populated by analyzing the assignment statements and conditional operators.

- After the module has been analyzed, we search for the state registers. For a *clocked* always block, we look for an intersection within the control list and the state list. If this intersection is not empty, we append that to the list of module state registers.

- For combinational *always* blocks (no clock), we employ the method proposed in Section 3.5.2. We look for *always* blocks whose data or control set intersects with the output set of combinational block being analyzed. If any of the shortlisted *always* blocks themselves are combinational, the same search kicks off again till we terminate at a *clocked* always block. When a clocked block is reached, we look for an intersection of the state set of the clocked *always* block and the control set of the combinational *always* block. If this intersection is not empty, we add the register to the list of module state registers.

- Other heuristics (For eg. considering the signals with the most appearance in predicates) can also be integrated easily at this step if future works demands it.

After finding this list of state registers, we proceed towards looking at dataflow trees and writing SystemVerilog cover properties.

### 3.7.3 Writing cover properties

Another subsystem is added after the bind analyzer called the *property writer*. The property writer uses the list of module state registers(from Section 3.7.2) and dataflow trees(generated by the bind analyzer) to write properties.

- The system starts by analyzing the dataflow trees of the state registers to create a *valuetable*. It traverses the tree to generate a valuetable with the assignment value and the corresponding predicates which would result in that value. Since we are focusing on tackling ambiguity, the assignment values with only one condition are rejected, as those cases can be covered by line coverage. The remaining cases are stored in the table. Thus, the antecedents for the properties are ready.

- The dataflow graphs of all the registers are traversed and the conditions are analyzed and compared with the valuetable created in the previous step. When a match is found between a condition and an entry on the valuetable, it is noted by the tool and valid properties are written.

Another engine called the *SVWriter* is responsible for writing all the SystemVerilog properties with the correct syntax and declarations.

- This analysis follows the same visitor pattern as done by other engines in the toolkit. It starts with the top module, writes the necessary properties, and then continues along the hierarchy to write the properties for each design unit.

### 3.7.4 Other salient features

Some important features of the tool that were added have been mentioned in this section.

- The tool offers the capability to *black box* certain modules. This has been done to ensure the robustness of the tool. Black boxing can be happen in two cases. The first when the user provides a list of modules which should not be analyzed. This can be to ensure faster processing times for keeping system memory considerations. The second case is when the tool itself cannot find a particular module (when it is elaborating instance declarations), it automatically black boxes the module and continues with the processing instead of aborting.

- It was observed that the properties were not unique. This is possible when a control block has multiple assignment statements, which results in different registers having the same resulting property. To avoid verbosity in the output, a uniqueness check can be done to filter out multiple

instances of the same property.

- As stated in Section 3.7.3, state register entries in the valuetable with only one condition are removed. This approach is followed in the *normal* mode of the tool. However, there is also an *exhaustive* flag which can be used to ensure that such properties are not filtered out, resulting in complete coverage of the control flow related to the identified state registers.

# Chapter 4

# Results

## 4.1 Tool Results

The Python tool was run over multiple open-source designs, and the results obtained have been tabulated in Table 4.1. The presented designs include cores such as Amber ARM [4], V-scale [6], OR1200 [5] and a 32-bit MIPS [3]. Controller modules such as SDRAM controllers (16 and 32-bit), PID controller and an I2C controller have also been analyzed. These designs are available at OpenCores [3]. Other miscellaneous designs available at OpenCores such as an AES core, Reed Solomon decoder, Wishbone FLASH interface and a Wishbone controlled UART have also been analyzed. Table 4.1 presents for each design the total number of cover properties generated by running the tool in both *normal* and *exhaustive* mode (Section 3.7.4).

We would like to mention that there are designs such as the OpenSPARC [9] where the tool was unable to generate properties due to it not being able to identify state registers. The RTL has to be behavioral description of the design for the tool to identify ambiguity. Structural RTL with the state machine spread across multiple modules cannot be handled by the tool in its current form.

| **Design** | **Lines of RTL** | **State registers identified** | **Cover properties (normal)** | **Cover properties (exhaustive)** |
|---|---|---|---|---|
| Amber ARM | 6317 | 7 | 101 | 234 |
| AES128 | 2895 | 1 | 182 | 266 |
| I2C controller | 1968 | 3 | 229 | 250 |
| MIPS32 | 2185 | 1 | 70 | 81 |
| OR1200 | 31640 | 8 | 200 | 237 |
| PID controller | 2151 | 3 | 18 | 37 |
| Reed Solomon decoder | 4321 | 30 | 1512 | 1684 |
| 16-bit SDRAM controller | 741 | 3 | 75 | 186 |
| 32-bit SDRAM controller | 3961 | 5 | 181 | 212 |
| V-scale | 1727 | 3 | 26 | 29 |
| Wishbone FLASH | 112 | 1 | 6 | 8 |
| Wishbone UART | 1934 | 2 | 139 | 202 |

Table 4.1: Preliminary results on selected designs

A concern raised with these properties is their possible unreachability. A formal analysis using Cadence JasperGold of the designs presented in Table 4.1 found about 30% of the properties to be unreachable.

We present case studies with two different cores in the following sections. The arguments for the usefulness of these properties have been discussed. Formal analysis has also been used to analyze the properties and study the issue of unreachability in more detail.

## 4.2 Case study : V-scale

We discuss specific results with respect to the V-scale [6] core developed as an implementation of RV32IM [56]. The results presented are when our Python tool was run in *exhaustive* mode. As mentioned, a total of 29 properties were generated for the design. These were distributed as 24 properties for the *md* unit (multiply and divide) and 5 properties for the *csr* file unit (control and status registers). All 29 properties are attached in Appendix B. To test the validity of the properties, they were instantiated with the testbench provided by the designers to be run for all the **47** tests provided in the package. The results are presented in Table 4.2

| **Design unit** | **Total properties** | **Uncovered properties** | **Line coverage** | **Branch Coverage** |
|---|---|---|---|---|
| md | 24 | 10 | 100% | 100% |
| csr | 5 | 2 | 61.49% | 55.14% |

Table 4.2: Results on V-scale

```
//V-scale md unit code snippet
always @(*) begin
   case (state)
      s_idle : next_state = (req_valid) ? s_compute :
         s_idle;
      s_compute : next_state = (counter == 0) ?
         s_setup_output : s_compute;
      s_setup_output : next_state = s_done;
      s_done : next_state = s_idle;
      default : next_state = s_idle;
   endcase // case (state)
end
```

Figure 4.1: Code snippet from V-scale (md unit)

### 4.2.1 Multiply and divide unit

For the *md* unit, 10 properties were not covered (2, 3, 5, 7, 10, 17, 18, 19, 20, 22). The thing to note is that 100% line coverage was reported for the *md* unit. The full design was then formally checked for reachability of the properties using Cadence JasperGold. Of the 24 *md* unit properties, 4 properties were proven unreachable by the formal tool (5, 17, 18, 20). This implies that the other 6 uncovered properties are valid scenarios for the design to be exercised in, which the provided testbench does not cover.

It was observed that all the 4 unreachable properties have the same antecedent. Formal analysis found this antecedent unreachable, thus making the 4 implication properties unreachable. Consider Fig. 4.1 which shows the code snippet where the antecedent comes from. The *default* option can never be exercised, thus leading to unreachable code.

```
//V-scale csr file code snippet
next_htif_state = htif_state; //Highlight
case (htif_state)
  HTIF_STATE_IDLE : begin
     if (htif_pcr_req_valid) begin
        htif_fire = 1'b1;
        next_htif_state = HTIF_STATE_WAIT;
     end
  end
  HTIF_STATE_WAIT : begin
     if (htif_pcr_resp_ready) begin
        next_htif_state = HTIF_STATE_IDLE;
     end
  end
endcase // case (htif_state)
```

Figure 4.2: Code snippet from V-scale (csr file)

### 4.2.2 CSR file

An analysis similar to Section 4.2.1 was done for the *csr* file. The 2 uncovered properties were proven unreachable (26, 27). It was observed that both the properties have the same antecedent, which was proven unreachable. Consider the code snippet in Fig. 4.2. The tool generates a property attempting to make the highlighted statement as the *cause* (antecedent) for the property. However, the *case* statement following the statement will always take precedence, no matter what the input.

### 4.2.3 Discussion

There were properties written by the tool which were **uncovered** in all the tests but **proven reachable** through formal analysis (Section 4.2.1). We

present an analysis of these properties here. All the properties can be found in Appendix B. A total of 6 properties fall in this case (2, 3, 7, 10, 19, 22). 3 (7, 19, 22) out of these 6 properties have antecedent as `(reset)`. That means, these 3 properties target different cases of system startup. Another property (16) which has an antecedent as `(reset)` was covered by the tests. The other 3 properties (2, 3, 10) also share the same antecedent. The antecedent corresponds to the end of an operation (multiply/divide). All the three properties cover cases where a new request is given to the *md* unit right after the previous request ends. The properties differ in what kind of new request has been given. Property 12 (`prop_12`) shares the same antecedent and was covered by the tests. There are covered properties which share consequents with the three properties as well. In summary, the antecedents and consequents of the properties have been covered independently, however some special interactions (which are reachable) have not been exercised. It can be observed that the targeted ambiguity is not just an artifact of the RTL written, but rather covers interesting behavioral/functional cases. The cases uncovered by the tests (different start-up from reset, back to back requests) do qualify for corner-case behavior of the multiply and divide unit under consideration.

Through the formal analysis discussed in Sections 4.2.1 and 4.2.2, we suggest that all the **unreachable** properties for this design are due to verbose statements in the RTL which can never be exercised. Though 6 out of the total 29 properties (20.7%) of the properties can be proven unreachable, we have shown that this unreachability is due to verbose code in the design which

can never be exercised.

## 4.3 Case study: Amber ARM

We present an analysis similar to Section 4.2 for the Amber ARM [4]. Due to the large number of properties in the *exhaustive* mode (Table 4.1), we limit our discussion to results of *normal* mode execution of the tool. The tool when run with Amber ARM generates 101 properties distributed between *multiply* unit (3), *decode* unit (22) and the *wishbone* module (76). It should be noted that while running the tool, the *icache* and *dcache* in the design have been black-boxed. These properties have been enumerated in Appendix C. With all the **59** test cases available for the Amber unit, 47 out of 101 properties were covered. The results of the tests are given in Table 4.3.

| Design unit | Total properties | Uncovered properties | Line coverage | Branch Coverage |
|---|---|---|---|---|
| Multiply | 3 | 1 | 100% | 100% |
| Decode | 22 | 6 | 94.42% | 92.61% |
| Wishbone | 76 | 47 | 94.04% | 91.42% |

Table 4.3: Results on Amber ARM

### 4.3.1 Multiply unit

Formal analysis of the entire design found all 3 properties reachable. Therefore, 1 scenario remains unexercised even with full line and branch coverage.

### 4.3.2 Decode unit

Formal analysis found all uncovered 6 properties unreachable. We take the analysis one step further and analyze the antecedents and consequents of all the 6 properties separately for reachability. The results show that 4 out of the 6 properties have unreachable antecedents and all 6 properties have unreachable consequents. This points towards the existence of dead code in the design, as the antecedents and consequents are just control flow conditions to target specific sections of the RTL code.

Another formal test was done with the same properties, with the difference being the DUV. Instead of formally analyzing the complete Amber core, we instead focused on analyzing the *decode* module as a standalone unit. This analysis found a trace for 4 out of the 6 previously unreachable properties. The remaining 2 properties were found to be unreachable due to unreachable consequents (unreachable dead code).

### 4.3.3 Wishbone unit

Out of the 47 uncovered properties, 21 were proven unreachable by Cadence JasperGold. That means, there were 26 signal interactions in the design that were not exercised by the testbench. A formal analysis of the 21 unreachable properties found that 2 properties had unreachable antecedents and 2 others had unreachable consequents. Doing a standalone analysis with the *wishbone* unit found all consequents and antecedents to be reachable and a total of 8 properties unreachable. The other 13 properties were reachable in

the individual *wishbone* unit.

All the 8 properties (48. 52. 53. 58, 63, 85, 90, 91) have reachable consequents and antecedents, which means that they cannot be justified as dead code. Let us consider `prop_53` in Appendix C as an example. This property is about starting from a reset state and going to a state where the design waits for an acknowledgement. This situation can never happen. `prop_58` is about waiting for an acknowledgement after a previous acknowledgement was received and there is no other request. These properties targeted ambiguity in the design, however the states they target are unreachable. This presents evidence of the unreachability issue we expressed in Section 3.6.

### 4.3.4 Discussion

The observations made in Section 4.2.3 can also be made for this unit. We will discuss the properties **uncovered** by all the tests but **proven reachable** by formal analysis. For the *multiply* unit, the uncovered reachable property (24) deals with the reset state. 11 (31, 32, 33, 37, 38, 39, 40, 69, 81, 82, 95) out of the 26 uncovered reachable properties for the *wishbone* unit also deal with the reset state. The remaining 15 properties in the *wishbone* unit (29, 26, 24, 35, 36, 47, 50, 64, 72, 78, 79, 87, 89, 93, 100) correspond to behavioral/functional cases. For example, properties 26 and 29 cover cases where the *wishbone_st* register (the main control register in the design) goes from `BURST3` to `WB_WAIT_ACK` and receives an acknowledgement in the very next cycle. Property 87 also targets a similar case but where the state goes from `IDLE`

to WB_WAIT_ACK and receives a similar fast acknowledgement. Therefore, these uncovered properties are targeting behavioral corner-cases where the design should be tested (different system start-ups, fast acknowledgements).

Sections 4.3.2 and 4.3.3 also present arguments for the usefulness of some of the unreachable properties. It has been formally shown that of 19 out of the 27 unreachable properties point at either actual dead code in the individual units or pseudo-dead code, i.e., logic available in the design which cannot be exercised by the complete module. A total of 8 properties out of 101 (7.9%) are actually unreachable due to constraints in the specification.

## 4.4 Summary

We summarise the results obtained and the key observations made in the this chapter.

1. Table 4.1 demonstrates the basic capabilities of the tool and showcases its application on a range of designs.

2. Two case studies (Sections 4.2 and 4.3) to discuss the properties, present experimental results and tackle the issue of unreachability. For both cases, the properties generated by the tool were added to the testbenches provided with the designs. The coverage results obtained after executing all the results were analyzed. A formal analysis using Cadence JasperGold was also done with all the properties to test for unreachability.

3. Properties were found (Sections 4.2.1, 4.3.1, 4.3.3) which were uncovered but proven reachable. Discussions were presented (Sections 4.2.3 and 4.3.4) to show that these properties target interesting cases for the execution of the design.

4. The unreachable properties were analyzed for the reasons of their unreachability. It was presented in Sections 4.2.1 and 4.2.2 that some of the unreachable properties were due to verbose code in the design. Sections 4.3.2 and 4.3.3 also presented cases where the properties were reachable in the individual design units, but unreachable in the bigger system. Some properties (Section 4.3.3) still remained which targeted unreachable states in the design, highlighting a drawback of the approach. However, such properties only made up 29.6% of the total unreachable properties (8 out of 27 for the Amber).

# Chapter 5

# Conclusions and future work

In conclusion, we have presented a methodology which targets signal relationships within a design for coverage measurement. The cover properties are generated automatically and can be easily integrated with any verification environment. The process is scalable as none of the steps involved can lead to any form of state explosion. The motivation and guiding principles have been discussed and details about a Python implementation have been presented in this thesis.

Results have been presented to experimentally support the usefulness of our properties. Case studies on two different cores, representing two different ISAs, have been presented. The provided testbenches obtained high line and branch coverage on the units (even 100% in some cases), but still there were properties that were not covered. Some of these uncovered were proven reachable through formal analysis. This points towards coverage holes in the tests and incomplete testing of the individual design units. These case studies highlight the need of a better coverage metric which can target such corner cases. We have presented experimental evidence for extracting behaviorally relevant properties directly from RTL to fill coverage holes.

The drawback of unreachability has been discussed. Properties were found to be unreachable as they targeted unreachable states. The current approach also does not take into account multiple states together. For designs with multiple state registers with ambiguity, directives can be built by taking different combinations.

For future work, this can be extended for both correctness and completeness. For correctness, the tool can be made smarter to avoid unreachable properties. Using a constraint solver in the tool can help alleviate some problems. Heuristics (or even input from the user) can be used to avoid unreachable properties stemming from known unreachable code in the design (such as *default* in *case* statements).

For completeness, the number and types of properties being generated can be increased. The tool in its current state generates properties for each individual module. This can be extended to generate global properties, with the antecedents and consequents being in different modules in the design hierarchy. This can cover a variety of interface interactions and handle state machines which span across multiple modules.

Another possible approach for completeness can be the re-examination of the heuristic to find state registers. The purpose of the state registers is to limit the properties being generated to meaningful control relationships. They attempt to identify the registers controlling the underlying FSM in the module. Since designers use multiple ways to write RTL, robustness is required in the heuristic. Our tool was unable to identify state registers in some of the design

units it analyzed. Therefore, a better approach for finding these important control registers can assist the tool's completeness.

# Appendices

# Appendix A

# 2-way arbiter RTL

RTL for a 2-way arbiter.[8]

```
module top (
    clock      , // clock
    reset      , // Active high, syn reset
    req_0      , // Request 0
    req_1      , // Request 1
    gnt_0      , // Grant 0
    gnt_1
);
//-------------Input Ports-----------------------------
input   clock,reset,req_0,req_1;
//-------------Output Ports----------------------------
output  gnt_0,gnt_1;
//-------------Input ports Data Type-------------------
wire    clock,reset,req_0,req_1;
//-------------Output Ports Data Type------------------
reg     gnt_0,gnt_1;
//-------------Internal Constants
   --------------------------
parameter SIZE = 3;
parameter IDLE  = 3'b001,GNT0 = 3'b010,GNT1 = 3'b100;
//-------------Internal Variables
   ---------------------------
reg   [SIZE-1:0]          state        ;// Seq part of
   the FSM
reg   [SIZE-1:0]          next_state   ;// combo part of
   FSM
//----------Code startes Here------------------------
always @ (state or req_0 or req_1)
begin : FSM_COMBO
```

```
    next_state = 3'b000;
    case(state)
        IDLE : if (req_0 == 1'b1) begin
            next_state = GNT0;
        end else if (req_1 == 1'b1) begin
            next_state= GNT1;
        end else begin
            next_state = IDLE;
        end
        GNT0 : if (req_0 == 1'b1) begin
            next_state = GNT0;
        end else begin
            next_state = IDLE;
        end
        GNT1 : if (req_1 == 1'b1) begin
            next_state = GNT1;
        end else begin
            next_state = IDLE;
        end
        default : next_state = IDLE;
    endcase
end
//----------Seq Logic-----------------------------
always @ (posedge clock)
begin : FSM_SEQ
    if (reset == 1'b1) begin
        state <=  IDLE;
    end else begin
        state <=  next_state;
    end
end
//----------Output Logic-----------------------------
always @ (posedge clock)
begin : OUTPUT_LOGIC
    if (reset == 1'b1) begin
        gnt_0 <=  1'b0;
        gnt_1 <=  1'b0;
    end
    else begin
```

```
        case(state)
            IDLE : begin
                gnt_0 <=  1'b0;
                gnt_1 <=  1'b0;
            end
            GNT0 : begin
                gnt_0 <=  1'b1;
                gnt_1 <=  1'b0;
            end
            GNT1 : begin
                gnt_0 <=  1'b0;
                gnt_1 <=  1'b1;
            end
            default : begin
                gnt_0 <=  1'b0;
                gnt_1 <=  1'b0;
            end
        endcase
    end
end // End Of Block OUTPUT_LOGIC
endmodule // End of Module arbitergg
```

# Appendix B

# Cover directives generated for V-scale

```
//MD properties
prop_1 : cover property(@(posedge(clk)) ((!(reset))&&(!((
   state==0)))&&((state==1))&&(!((counter==0))))|->##1
   ((!((state==0)))&&((state==1))&&((op==0))&&(a[counter
   ])));

prop_2 : cover property(@(posedge(clk)) ((!(reset))&&(!((
   state==0)))&&(!((state==1)))&&(!((state==2)))&&((state
   ==3)))|->##1 (((state==0))&&(req_valid)&&((op==2))));

prop_3 : cover property(@(posedge(clk)) ((!(reset))&&(!((
   state==0)))&&(!((state==1)))&&(!((state==2)))&&((state
   ==3)))|->##1 (((state==0))&&(req_valid)));

prop_4 : cover property(@(posedge(clk)) ((!(reset))&&(!((
   state==0)))&&((state==1))&&(!((counter==0))))|->##1
   ((!((state==0)))&&((state==1))&&(!((op==0)))&&(a_geq))
   );

prop_5 : cover property(@(posedge(clk)) ((!(reset))&&(!((
   state==0)))&&(!((state==1)))&&(!((state==2)))&&(!((
   state==3))))|->##1 (((state==0))&&(req_valid)));

prop_6 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(!(req_valid)))|->##1 (((state==0))&&(
   req_valid)&&((op==2))));

prop_7 : cover property(@(posedge(clk)) ((reset))|->##1
   (((state==0))&&(req_valid)));

prop_8 : cover property(@(posedge(clk)) ((!(reset))&&((
```

```
   state==0))&&(req_valid))|->##1 ((!((state==0)))&&((
   state==1))&&((op==0))&&(a[counter])));

prop_9 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(req_valid))|->##1 ((!((state==0)))&&((
   state==1))));

prop_10 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&(!((state==1)))&&(!((state==2)))
   &&((state==3)))|->##1 (((state==0))&&(req_valid)&&(!((
   op==2)))));

prop_11 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&((state==1))&&(!((counter==0))))
   |->##1 ((!((state==0)))&&((state==1))));

prop_12 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&(!((state==1)))&&(!((state==2)))
   &&((state==3)))|->##1 (((state==0))));

prop_13 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&((state==1))&&((counter==0)))|->##1
    ((!((state==0)))&&(!((state==1)))&&((state==2))));

prop_14 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(!(req_valid)))|->##1 (((state==0))&&(
   req_valid)));

prop_15 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(!(req_valid)))|->##1 (((state==0))));

prop_16 : cover property(@(posedge(clk)) ((reset))|->##1
   (((state==0))));

prop_17 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&(!((state==1)))&&(!((state==2)))
   &&(!((state==3))))|->##1 (((state==0))));

prop_18 : cover property(@(posedge(clk)) ((!(reset))
```

```
   &&(!((state==0)))&&(!((state==1)))&&(!((state==2)))
   &&(!((state==3))))|->##1 (((state==0))&&(req_valid)
   &&(!((op==2)))));

prop_19 : cover property(@(posedge(clk)) ((reset))|->##1
   (((state==0))&&(req_valid)&&(!((op==2)))));

prop_20 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&(!((state==1)))&&(!((state==2)))
   &&(!((state==3))))|->##1 (((state==0))&&(req_valid)
   &&((op==2))));

prop_21 : cover property(@(posedge(clk)) ((!(reset))
   &&(!((state==0)))&&(!((state==1)))&&((state==2)))
   |->##1 ((!((state==0)))&&(!((state==1)))&&(!((state
   ==2)))&&((state==3))));

prop_22 : cover property(@(posedge(clk)) ((reset))|->##1
   (((state==0))&&(req_valid)&&((op==2))));

prop_23 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(req_valid))|->##1 ((!((state==0)))&&((
   state==1))&&(!((op==0)))&&(a_geq)));

prop_24 : cover property(@(posedge(clk)) ((!(reset))&&((
   state==0))&&(!(req_valid)))|->##1 (((state==0))&&(
   req_valid)&&(!((op==2)))));

//csr properties
prop_25 : cover property(@(posedge(clk)) ((!(htif_reset))
   &&((htif_state==0))&&(htif_pcr_req_valid))|->##1 ((!((
   htif_state==0)))&&((htif_state==1))&&(
   htif_pcr_resp_ready)));

prop_26 : cover property(@(posedge(clk)) ((!(htif_reset))
   &&(!((htif_state==0)))&&(!((htif_state==1))))|->##1
   ((!((htif_state==0)))&&((htif_state==1))&&(
   htif_pcr_resp_ready)));
```

```
prop_27 : cover property(@(posedge(clk)) ((!(htif_reset))
   &&(!((htif_state==0)))&&(!((htif_state==1))))|->##1
   (((htif_state==0))&&(htif_pcr_req_valid)));

prop_28 : cover property(@(posedge(clk)) ((!(htif_reset))
   &&(!((htif_state==0)))&&((htif_state==1))&&(
   htif_pcr_resp_ready))|->##1 (((htif_state==0))&&(
   htif_pcr_req_valid)));

prop_29 : cover property(@(posedge(clk)) ((htif_reset))
   |->##1 (((htif_state==0))&&(htif_pcr_req_valid)));
```

# Appendix C

# Cover directives generated for Amber ARM

```
//Decode Properties
prop_1 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&((control_state==16)))|->##1 ((!(
    instruction_valid))&&(!((control_state==16)))&&(!((
    control_state==14)))&&(!((control_state==13)))&&(!(((
    mtrans_num_registers==1)&&(control_state==11))))&&(!((
    control_state==10)))&&(!(((control_state==7)||(
    control_state==19))))&&(!((control_state==6)))&&(!((
    control_state==12)))&&((control_state==15))));

prop_2 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&(!((control_state==16)))&&((
    control_state==14))&&(i_multiply_done)&&(!(
    o_multiply_function[1])))|->##1 ((!(instruction_valid)
    )&&(!((control_state==16)))&&(!((control_state==14)))
    &&(!((control_state==13)))&&(!(((mtrans_num_registers
    ==1)&&(control_state==11))))&&(!((control_state==10)))
    &&(!(((control_state==7)||(control_state==19))))&&(!((
    control_state==6)))&&(!((control_state==12)))&&((
    control_state==15))));

prop_3 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&((control_state==16)))|->##1
    ((!((control_state==16)))&&((control_state==15))));

prop_4 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&(!((control_state==16)))&&(!((
```

```
   control_state==14)))&&(!((control_state==13)))&&(((
   mtrans_num_registers==1)&&(control_state==11)))&&(!(
   dabt))&&(write_pc))|->##1 ((!(instruction_valid))
   &&(!((control_state==16)))&&(!((control_state==14)))
   &&(!((control_state==13)))&&(!(((mtrans_num_registers
   ==1)&&(control_state==11))))&&(!((control_state==10)))
   &&(!(((control_state==7)||(control_state==19))))&&((
   control_state==6))));

prop_5 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&((
   control_state==14))&&(i_multiply_done)&&(!(
   o_multiply_function[1])))|->##1 ((!(((
   instruction_execute&&(control_state==20))&&(!(conflict
   )))))&&((control_state==15))&&(instruction[20])));

prop_6 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&(!((
   control_state==14)))&&(!((control_state==13)))&&(((
   mtrans_num_registers==1)&&(control_state==11)))&&(dabt
   ))|->##1 ((!(instruction_valid))&&(!((control_state
   ==16)))&&(!((control_state==14)))&&(!((control_state
   ==13)))&&(!(((mtrans_num_registers==1)&&(control_state
   ==11))))&&(!((control_state==10)))&&(!(((control_state
   ==7)||(control_state==19))))&&(!((control_state==6)))
   &&((control_state==12))));

prop_7 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&((control_state==16)))|->##1
   ((!(((instruction_execute&&(control_state==20))&&(!(
   conflict)))))&&((control_state==15))&&(!((dtype==1))))
   );

prop_8 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&((
```

```
   control_state==14))&&(i_multiply_done)&&(!(
   o_multiply_function[1])))|->##1 ((!(((
   instruction_execute&&(control_state==20))&&(!(conflict
   )))))&&((control_state==15))));

prop_9 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(
   instruction_valid)&&(!((interrupt&&(!(conflict)))))
   &&(!((((dtype==8)&&(!(und_request)))&&(!(conflict)))))
   &&(!(((dtype==2)&&(!(conflict)))))&&(!(((dtype==1)
   &&(!(conflict)))))&&(!(((((mtrans_num_registers!=1)&&(
   mtrans_num_registers!=0))&&(dtype==4))&&(!(conflict)))
   ))&&(!(((instruction[20]&&(mtrans_reg1==15))&&(dtype
   ==4))))&&(!((load_op&&(instruction[15:12]==15))))&&(
   current_write_pc))|->##1 ((!(instruction_valid))&&(!((
   control_state==16)))&&(!((control_state==14)))&&(!((
   control_state==13)))&&(!(((mtrans_num_registers==1)&&(
   control_state==11))))&&(!((control_state==10)))&&(!(((
   control_state==7)||(control_state==19))))&&(!((
   control_state==6)))&&(!((control_state==12)))&&(!((
   control_state==15)))&&(!((control_state==18)))&&(!((
   control_state==17)))&&(!((control_state==9)))&&((
   control_state==8))));

prop_10 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&(!((
   control_state==14)))&&(!((control_state==13)))&&(!(((
   mtrans_num_registers==1)&&(control_state==11))))&&((
   control_state==10))&&(!((mtrans_instruction_nxt
   [15:0]!=0)))&&(!(dabt))&&(write_pc))|->##1 ((!(
   instruction_valid))&&(!((control_state==16)))&&(!((
   control_state==14)))&&(!((control_state==13)))&&(!(((
   mtrans_num_registers==1)&&(control_state==11))))&&(!((
   control_state==10)))&&(!(((control_state==7)||(
   control_state==19))))&&((control_state==6))));

prop_11 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
```

```
    instruction_valid))&&(!((control_state==16)))&&(!((
    control_state==14)))&&(!((control_state==13)))&&(((
    mtrans_num_registers==1)&&(control_state==11)))&&(dabt
    ))|->##1 ((!(((instruction_execute&&(control_state
    ==20))&&(!(conflict)))))&&(!((control_state==15)))&&((
    control_state==12))&&(restore_base_address)));

prop_12 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&(!((control_state==16)))&&((
    control_state==14))&&(i_multiply_done)&&(!(
    o_multiply_function[1])))|->##1 ((!((control_state
    ==16)))&&((control_state==15))));

prop_13 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&(!((control_state==16)))&&(!((
    control_state==14)))&&(!((control_state==13)))&&(!(((
    mtrans_num_registers==1)&&(control_state==11))))&&(!((
    control_state==10)))&&(((control_state==7)||(
    control_state==19)))&&(write_pc))|->##1 ((!(
    instruction_valid))&&(!((control_state==16)))&&(!((
    control_state==14)))&&(!((control_state==13)))&&(!(((
    mtrans_num_registers==1)&&(control_state==11))))&&(!((
    control_state==10)))&&(!(((control_state==7)||(
    control_state==19))))&&(!((control_state==6)))&&(!((
    control_state==12)))&&(!((control_state==15)))&&(!((
    control_state==18)))&&(!((control_state==17)))&&(!((
    control_state==9)))&&((control_state==8))));

prop_14 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((!(i_access_stall)))&&(!(
    instruction_valid))&&(!((control_state==16)))&&((
    control_state==14))&&(i_multiply_done)&&(!(
    o_multiply_function[1])))|->##1 ((!(((
    instruction_execute&&(control_state==20))&&(!(conflict
    )))))&&((control_state==15))&&(!((dtype==1)))));

prop_15 : cover property(@(posedge(i_clk)) ((!((!(
```

```
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&(!((
   control_state==14)))&&(!((control_state==13)))&&(!(((
   mtrans_num_registers==1)&&(control_state==11))))&&((
   control_state==10))&&(!((mtrans_instruction_nxt
   [15:0]!=0)))&&(dabt))|->##1 ((!(instruction_valid))
   &&(!((control_state==16)))&&(!((control_state==14)))
   &&(!((control_state==13)))&&(!(((mtrans_num_registers
   ==1)&&(control_state==11))))&&(!((control_state==10)))
   &&(!(((control_state==7)||(control_state==19))))&&(!((
   control_state==6)))&&((control_state==12))));

prop_16 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(
   instruction_valid)&&(!((interrupt&&(!(conflict)))))
   &&(!((((dtype==8)&&(!(und_request)))&&(!(conflict)))))
   &&(!(((dtype==2)&&(!(conflict)))))&&(!(((dtype==1)
   &&(!(conflict)))))&&(!(((((mtrans_num_registers!=1)&&(
   mtrans_num_registers!=0))&&(dtype==4))&&(!(conflict)))
   ))&&(!(((instruction[20]&&(mtrans_reg1==15))&&(dtype
   ==4))))&&((load_op&&(instruction[15:12]==15))))|->##1
   ((!(instruction_valid))&&(!((control_state==16)))
   &&(!((control_state==14)))&&(!((control_state==13)))
   &&(!(((mtrans_num_registers==1)&&(control_state==11)))
   )&&(!((control_state==10)))&&(!(((control_state==7)||(
   control_state==19))))&&((control_state==6))));

prop_17 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&((control_state==16)))|->##1
   ((!(((instruction_execute&&(control_state==20))&&(!(
   conflict)))))&&((control_state==15))&&(instruction
   [20])));

prop_18 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&((control_state==16)))|->##1
   ((!(((instruction_execute&&(control_state==20))&&(!(
   conflict)))))&&((control_state==15))&&((dtype==1))));
```

```
prop_19 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&(!((
   control_state==14)))&&(!((control_state==13)))&&(!(((
   mtrans_num_registers==1)&&(control_state==11))))&&((
   control_state==10))&&(!((mtrans_instruction_nxt
   [15:0]!=0)))&&(dabt))|->##1 ((!(((instruction_execute
   &&(control_state==20))&&(!(conflict)))))&&(!((
   control_state==15)))&&((control_state==12))&&(
   restore_base_address)));

prop_20 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&((control_state==16)))|->##1
   ((!(((instruction_execute&&(control_state==20))&&(!(
   conflict)))))&&((control_state==15))));

prop_21 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(
   instruction_valid)&&(!((interrupt&&(!(conflict)))))
   &&(!((((dtype==8)&&(!(und_request)))&&(!(conflict)))))
   &&(!(((dtype==2)&&(!(conflict)))))&&(!(((dtype==1)
   &&(!(conflict)))))&&(!(((((mtrans_num_registers!=1)&&(
   mtrans_num_registers!=0))&&(dtype==4))&&(!(conflict)))
   ))&&(((instruction[20]&&(mtrans_reg1==15))&&(dtype==4)
   )))|->##1 ((!(instruction_valid))&&(!((control_state
   ==16)))&&(!((control_state==14)))&&(!((control_state
   ==13)))&&(!(((mtrans_num_registers==1)&&(control_state
   ==11))))&&(!((control_state==10)))&&(!(((control_state
   ==7)||(control_state==19))))&&((control_state==6))));

prop_22 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(!(
   instruction_valid))&&(!((control_state==16)))&&((
   control_state==14))&&(i_multiply_done)&&(!(
   o_multiply_function[1])))|->##1 ((!(((
   instruction_execute&&(control_state==20))&&(!(conflict
   )))))&&((control_state==15))&&((dtype==1))));
```

```
//Multiply Properties
prop_23 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(i_execute)
   &&((count==0))&&(!(enable)))|->##1 (((count==0))));

prop_24 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 (((count==0))));

prop_25 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((!(i_access_stall)))&&(i_execute)
   &&(!((count==0)))&&(((accumulate&&(count==35))||((
   count==34)&&(!(accumulate))))))|->##1 (((count==0))));

//Wishbone Properties
prop_26 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&((
   wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(
   extra_write_r)));

prop_27 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(!(
   dcache_read_qword_c))));

prop_28 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&((
   byte_enable==1))));

prop_29 : cover property(@(posedge(i_clk)) ((!((!(
```

```
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&((
    wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r))));

prop_30 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&((
    wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
    quick_n_reset))))&&(dcache_cached_rreq_r)&&(((
    wishbone_st==4)||(wishbone_st==0)))));

prop_31 : cover property(@(posedge(i_clk)) (((!(
    quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(start_access)));

prop_32 : cover property(@(posedge(i_clk)) (((!(
    quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(!(start_access))&&((!(
    wait_write_ack)))));

prop_33 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(!(
    wait_write_ack))&&((dcache_uncached_rreq_c||
    dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
    |->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)))
    &&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r))));

prop_34 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
    ))|->##1 ((!((!(quick_n_reset))))&&(
    dcache_cached_rreq_r)&&(((wishbone_st==4)||(
    wishbone_st==0)))));
```

```
prop_35 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&((dcache_uncached_rreq_c||
   dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
   |->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)))
   &&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)));

prop_36 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&(!((dcache_uncached_rreq_c||
   dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
   icache_read_req_c&&icache_read_qword_c)))&&(
   icache_read_req_c))|->##1 ((!((!(quick_n_reset))))
   &&(!((wishbone_st==0)))&&(!((wishbone_st==1)))&&(!((
   wishbone_st==2)))&&(!((wishbone_st==3)))&&((
   wishbone_st==4))&&(i_wb_ack)&&(!(extra_write_r))));

prop_37 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(
   dcache_read_qword_c)));

prop_38 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(
   dcache_cached_rreq_c)));

prop_39 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&(!((
   byte_enable==3)))&&(!((byte_enable==12)))));

prop_40 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
```

```
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(!(
   write_req_c))&&((icache_read_req_c&&
   icache_read_qword_c))));

prop_41 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(
   write_req_c)));

prop_42 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(
   write_req_c)));

prop_43 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&((byte_enable==8))));

prop_44 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(!(
   write_req_c))&&((icache_read_req_c&&
   icache_read_qword_c))));
```

```
prop_45 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(
   dcache_read_qword_c)));

prop_46 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(!(
   write_req_c))&&(!((icache_read_req_c&&
   icache_read_qword_c)))&&(icache_read_req_c)));

prop_47 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&((dcache_uncached_rreq_c||
   dcache_cached_rreq_c))&&(dcache_read_qword_c))|->##1
   ((!((!(quick_n_reset))))&&(!((wishbone_st==0)))&&((
   wishbone_st==1))&&(i_wb_ack)));

prop_48 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&(!((dcache_uncached_rreq_c||
   dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
   icache_read_req_c&&icache_read_qword_c)))&&(
   icache_read_req_c))|->##1 ((!((!(quick_n_reset))))&&(
   dcache_uncached_rreq_r)&&(((wishbone_st==4)||(
   wishbone_st==0)))));

prop_49 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
```

```
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&((
   byte_enable==3))));

prop_50 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&(!((dcache_uncached_rreq_c||
   dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
   icache_read_req_c&&icache_read_qword_c)))&&(
   icache_read_req_c))|->##1 ((!((!(quick_n_reset))))
   &&(!((wishbone_st==0)))&&(!((wishbone_st==1)))&&(!((
   wishbone_st==2)))&&(!((wishbone_st==3)))&&((
   wishbone_st==4))&&(i_wb_ack)));

prop_51 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&(!((
   byte_enable==3)))&&((byte_enable==12))));

prop_52 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&(!((dcache_uncached_rreq_c||
   dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
   icache_read_req_c&&icache_read_qword_c)))&&(
   icache_read_req_c))|->##1 ((!((!(quick_n_reset))))
   &&(((i_wb_ack&&extra_write_r)&&(wishbone_st==4)))));

prop_53 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(wait_write_ack)));

prop_54 : cover property(@(posedge(i_clk)) ((!((!(
```

```
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(!(start_access))&&((!(
    wait_write_ack)))));

prop_55 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(dcache_uncached_rreq_r)&&(((
    wishbone_st==4)||(wishbone_st==0))))|->##1 ((!((!(
    quick_n_reset))))&&(dcache_uncached_rreq_r)&&(((
    wishbone_st==4)||(wishbone_st==0)))));

prop_56 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(start_access)&&(!(dcache_req_c))));

prop_57 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!(dcache_uncached_rreq_r)))|->##1
    ((!((!(quick_n_reset))))&&(dcache_uncached_rreq_r)
    &&(((wishbone_st==4)||(wishbone_st==0)))));

prop_58 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(wait_write_ack)));

prop_59 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(start_access)));
```

```
prop_60 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&((
   byte_enable==1))));

prop_61 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&((byte_enable==2))));

prop_62 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&(!((
   byte_enable==3)))&&((byte_enable==12))));

prop_63 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&(
   dcache_uncached_rreq_r)&&(((wishbone_st==4)||(
   wishbone_st==0)))));

prop_64 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&((dcache_uncached_rreq_c||
   dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
   |->##1 ((!((!(quick_n_reset))))&&(((i_wb_ack&&
   extra_write_r)&&(wishbone_st==4)))));

prop_65 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&((
   byte_enable==4))));

prop_66 : cover property(@(posedge(i_clk)) ((!((!(
```

```
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&(!((
   byte_enable==3)))&&(!((byte_enable==12)))));

prop_67 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(
   dcache_cached_rreq_c)));

prop_68 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&((
   wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
   quick_n_reset))))&&(dcache_uncached_rreq_r)&&(((
   wishbone_st==4)||(wishbone_st==0)))));

prop_69 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))));

prop_70 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!(dcache_cached_rreq_r)))|->##1
   ((!((!(quick_n_reset))))&&(dcache_cached_rreq_r)&&(((
   wishbone_st==4)||(wishbone_st==0)))));

prop_71 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
```

```
    wishbone_st==0))&&(!(wait_write_ack))&&((
    dcache_uncached_rreq_c||dcache_cached_rreq_c))));

prop_72 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
    ))|->##1 ((!((!(quick_n_reset))))&&(((i_wb_ack&&
    extra_write_r)&&(wishbone_st==4)))));

prop_73 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(!(wait_write_ack))&&((
    dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(!(
    dcache_read_qword_c))));

prop_74 : cover property(@(posedge(i_clk)) (((!(
    quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(!(wait_write_ack))&&((
    dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(!(
    dcache_cached_rreq_c))&&(dcache_uncached_rreq_c)));

prop_75 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
    byte_enable==1)))&&((byte_enable==2))));

prop_76 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
    wishbone_st==0))));

prop_77 : cover property(@(posedge(i_clk)) ((!((!(
```

```
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&(
   dcache_uncached_rreq_r)&&(((wishbone_st==4)||(
   wishbone_st==0)))));

prop_78 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&((
   wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
   quick_n_reset))))&&(((i_wb_ack&&extra_write_r)&&(
   wishbone_st==4)))));

prop_79 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&(!((dcache_uncached_rreq_c||
   dcache_cached_rreq_c)))&&(!(write_req_c))&&((
   icache_read_req_c&&icache_read_qword_c)))|->##1
   ((!((!(quick_n_reset))))&&(!((wishbone_st==0)))&&((
   wishbone_st==1))&&(i_wb_ack)));

prop_80 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(dcache_cached_rreq_r)&&(((
   wishbone_st==4)||(wishbone_st==0))))|->##1 ((!((!(
   quick_n_reset))))&&(dcache_cached_rreq_r)&&(((
   wishbone_st==4)||(wishbone_st==0)))));

prop_81 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(!(dcache_req_c))));

prop_82 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))));

prop_83 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
```

```
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&((byte_enable==8))));

prop_84 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)));

prop_85 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&(
   dcache_cached_rreq_r)&&(((wishbone_st==4)||(
   wishbone_st==0)))));

prop_86 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&(!((
   byte_enable==4)))&&(!((byte_enable==8)))&&((
   byte_enable==3))));

prop_87 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
   ))|->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)
   ))&&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(
   extra_write_r)));

prop_88 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
   ))|->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)
   ))&&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r))));

prop_89 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
```

```
    ))|->##1 ((!((!(quick_n_reset))))&&(
    dcache_uncached_rreq_r)&&(((wishbone_st==4)||(
    wishbone_st==0)))));

prop_90 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(!(
    wait_write_ack))&&(!((dcache_uncached_rreq_c||
    dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
    icache_read_req_c&&icache_read_qword_c)))&&(
    icache_read_req_c))|->##1 ((!((!(quick_n_reset))))
    &&(!((wishbone_st==0)))&&(!((wishbone_st==1)))&&(!((
    wishbone_st==2)))&&(!((wishbone_st==3)))&&((
    wishbone_st==4))&&(i_wb_ack)&&(extra_write_r)));

prop_91 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(!(
    wait_write_ack))&&(!((dcache_uncached_rreq_c||
    dcache_cached_rreq_c)))&&(!(write_req_c))&&(!((
    icache_read_req_c&&icache_read_qword_c)))&&(
    icache_read_req_c))|->##1 ((!((!(quick_n_reset))))&&(
    dcache_cached_rreq_r)&&(((wishbone_st==4)||(
    wishbone_st==0)))));

prop_92 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
    wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
    extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&(
    dcache_cached_rreq_r)&&(((wishbone_st==4)||(
    wishbone_st==0)))));

prop_93 : cover property(@(posedge(i_clk)) ((!((!(
    quick_n_reset))))&&((wishbone_st==0))&&(!(
    wait_write_ack))&&((dcache_uncached_rreq_c||
    dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
    |->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)))
    &&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
    wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(
    extra_write_r)));
```

```
prop_94 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)&&(!(
   extra_write_r)))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&((
   dcache_uncached_rreq_c||dcache_cached_rreq_c))&&(!(
   dcache_cached_rreq_c))&&(dcache_uncached_rreq_c)));

prop_95 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)));

prop_96 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(!(wait_write_ack))&&(!((
   dcache_uncached_rreq_c||dcache_cached_rreq_c)))&&(!(
   write_req_c))&&(!((icache_read_req_c&&
   icache_read_qword_c)))&&(icache_read_req_c)));

prop_97 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&((dcache_uncached_rreq_c||
   dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
   |->##1 ((!((!(quick_n_reset))))&&(
   dcache_uncached_rreq_r)&&(((wishbone_st==4)||(
   wishbone_st==0)))));

prop_98 : cover property(@(posedge(i_clk)) (((!(
   quick_n_reset))))|->##1 ((!((!(quick_n_reset))))&&((
   wishbone_st==0))&&(start_access)&&(dcache_req_c)&&(!((
   byte_enable==1)))&&(!((byte_enable==2)))&&((
   byte_enable==4))));

prop_99 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(wait_write_ack
   ))|->##1 ((!((!(quick_n_reset))))&&(!((wishbone_st==0)
   ))&&(!((wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
```

```
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)));

prop_100 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&((
   wishbone_st==3))&&(i_wb_ack))|->##1 ((!((!(
   quick_n_reset))))&&(!((wishbone_st==0)))&&(!((
   wishbone_st==1)))&&(!((wishbone_st==2)))&&(!((
   wishbone_st==3)))&&((wishbone_st==4))&&(i_wb_ack)));

prop_101 : cover property(@(posedge(i_clk)) ((!((!(
   quick_n_reset))))&&((wishbone_st==0))&&(!(
   wait_write_ack))&&((dcache_uncached_rreq_c||
   dcache_cached_rreq_c))&&(!(dcache_read_qword_c)))
   |->##1 ((!((!(quick_n_reset))))&&(dcache_cached_rreq_r
   )&&(((wishbone_st==4)||(wishbone_st==0)))));
```

# Bibliography


[1] Coverage cookbook. *http://verificationacademy.com/cookbook/coverage*.

[2] http://accellera.org/downloads/standards/uvm.

[3] http://opencores.org.

[4] http://opencores.org/project,amber.

[5] http://openrisc.io.

[6] http://riscv.org/2015/09/risc-v-in-verilog/.

[7] http://standards.ieee.org/findstds/standard/1800-2012.html.

[8] http://www.asic-world.com/tidbits/verilog_fsm.html.

[9] http://www.oracle.com/technetwork/systems/opensparc/.

[10] Icarus verilog, http://iverilog.icarus.com.

[11] Jaspergold property synthesis apps property synthesis throughout the design flow for application in formal verification, simulation and emulation.

[12] Mentor questa sim 2017 user guide.

[13] Ply (python lex-yacc), http://www.dabeaz.com/ply/.

[14] Questa cdc and formal technologies datasheet.

[15] Vc formal coverage analyzer.

[16] Prashant Aggarwal, Darrow Chu, Vijay Kadamby, and Vigyan Singhal. Planning for end-to-end formal using simulation-based coverage: invited tutorial. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 9–16. FMCAD Inc, 2011.

[17] IEEE Standards Association et al. Ieee standard for verilog hardware description language. *Design Automation Standards Committee, IEEE Std 1364TM-2005*, 2, 2005.

[18] Viraj Athavale, Sam Hertz, Darshan Jetly, Vijay Ganesan, Jim Krysl, and Shobha Vasudevan. Using static analysis for coverage extraction fromemulation/prototyping platforms. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 207–214. ACM, 2012.

[19] Viraj Athavale, Sai Ma, Samuel Hertz, and Shobha Vasudevan. Code coverage of assertions using rtl source code analysis. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6. ACM, 2014.

[20] B Bailey. The wake of the sleeping giant-verification. *Scalable Verification Technical Publications*, 2002.

[21] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets. A study in coverage-driven test generation. In

*Design Automation Conference, 1999. Proceedings. 36th*, pages 970–975. IEEE, 1999.

[22] Jayanta Bhadra, Magdy S Abadir, Li-C Wang, and Sandip Ray. A survey of hybrid techniques for functional verification. *IEEE Design & Test of Computers*, 24(2):0112–122, 2007.

[23] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[24] Edmund Clarke and E Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of programs*, pages 52–71, 1982.

[25] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*, pages 1–30. Springer, 2011.

[26] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 418–425. IEEE Computer Society, 1997.

[27] Eman El Mandouh and Amr G Wassal. Automatic generation of functional coverage models. In *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*, pages 754–757. IEEE, 2016.

[28] Eman El Mandouh and Amr G Wassal. Covgen: A framework for automatic extraction of functional coverage models. In *Quality Electronic Design (ISQED), 2016 17th International Symposium on*, pages 146–151. IEEE, 2016.

[29] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Occom-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.

[30] Monica Farkash-presenter. Mining coverage data for test set coverage efficiency.

[31] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 151–158. IEEE, 2007.

[32] William C Hetzel and Bill Hetzel. *The complete guide to software testing.* John Wiley & Sons, Inc., 1991.

[33] Richard C Ho and Mark A Horowitz. Validation coverage analysis for complex digital designs. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 146–151. IEEE Computer Society, 1997.

[34] Richard C Ho, C Han Yang, Mark A Horowitz, and David L Dill. Architecture validation for processors. *ACM SIGARCH Computer Architecture News*, 23(2):404–413, 1995.

[35] Yatin Vasant Hoskote, Dinos Moundanos, and Jacob A Abraham. Automatic extraction of the control flow machine and application to evaluating coverage of verification vectors. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD'95. Proceedings., 1995 IEEE International Conference on*, pages 532–537. IEEE, 1995.

[36] Jing-Yang Jou and C Liu. Coverage analysis techniques for hdl design validation. *Proc. Asia Pacific CHip Design Languages*, pages 48–55, 1999.

[37] Jian Kang, Sharad C Seth, and Vijay Gangaram. Efficient rtl coverage metric for functional test selection. In *VLSI Test Symposium, 2007. 25th IEEE*, pages 318–324. IEEE, 2007.

[38] Chien-Nan Jimmy Liu and Jing-Yang Jou. An efficient functional coverage test for hdl descriptions at rtl. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 325–327. IEEE, 1999.

[39] Chien-Nan Jimmy Liu and Jing-Yang Jou. An automatic controller extractor for hdl descriptions at the rtl. *IEEE Design & Test of Computers*, 17(3):72–77, 2000.

[40] Joan C Miller and Clifford J Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[41] Dinos Moundanos, Jacob A Abraham, and Yatin Vasant Hoskote. Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14, 1998.

[42] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[43] Jaan Raik, Uljana Reinsalu, Raimund Ubar, Maksim Jenihhin, and Peeter Ellervee. Code coverage analysis using high-level decision diagrams. In *Design and Diagnostics of Electronic Circuits and Systems, 2008. DDECS 2008. 11th IEEE Workshop on*, pages 1–6. IEEE, 2008.

[44] Sébastien Regimbal, J-F Lemire, Yvon Savaria, Guy Bois, El Mostapha Aboulhamid, and A Baron. Automating functional coverage analysis based on an executable specification. In *System-on-Chip for Real-Time Applications, 2003. Proceedings. The 3rd IEEE International Workshop on*, pages 228–234. IEEE, 2003.

[45] Jian Shen and Jacob A Abraham. Verification of processor microarchitectures. In *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, pages 189–194. IEEE, 1999.

[46] Jian Shen and Jacob A Abraham. An rtl abstraction technique for processor microarchitecture validation and test generation. *Journal of Electronic Testing*, 16(1):67–81, 2000.

[47] David Sheridan, Lingyi Liu, Hyungsul Kim, and Shobha Vasudevan. A coverage guided mining approach for automatic generation of succinct assertions. In *Proceedings of the 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, pages 68–73. IEEE Computer Society, 2014.

[48] Kanna Shimizu and David L Dill. Deriving a simulation input generator and a coverage metric from a formal specification. In *Design Automation Conference, 2002. Proceedings. 39th*, pages 801–806. IEEE, 2002.

[49] Vigyan Singhal and Prashant Aggarwal. Using coverage to deploy formal verification in a simulation world. In *International Conference on Computer Aided Verification*, pages 44–49. Springer, 2011.

[50] CCITT Specification. description language (sdl). *ITU-T Recommendation*, (100):11, 1993.

[51] Rob Sumners, Jayanta Bhadra, and Jacob Abraham. Automatic validation test generation using extracted control models. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 312–317. IEEE, 2000.

[52] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *International Symposium on Applied Reconfigurable Computing*, pages 451–460. Springer, 2015.

[53] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 18(4):36–45, 2001.

[54] Shireesh Verma, Ian G Harris, and Kiran Ramineni. Automatic generation of functional coverage models from ctl. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, pages 159–164. IEEE, 2007.

[55] Shireesh Verma, IG Harris, and Kiran Ramineni. Automatic generation of functional coverage models from behavioral verilog descriptions. In *2007 Design, Automation&Test in Europe Conference&Exhibition*.

[56] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovi. The risc-v instruction set manual. volume 1: User-level isa, version 2.0. Technical report, DTIC Document, 2014.