Leveraging Assertion Based Verification by using Magellan

Jacob Andersen
Peter Jensen

SyoSil

Himmelev Bygade 53, 4000 Roskilde, Denmark
www.syosil.com
{jacob, peter}@syosil.com

ABSTRACT

Design reuse of Intellectual Property (IP) is today a commonly used approach to decrease design and verification time, by using already verified design blocks in multiple chip designs. The IP reuse strategy heavily relies on standardized on-chip interfaces between IP blocks, shifting the verification problem from the block level to the system level. Verification IP (VIP) has emerged to support not only verification of protocol compliance of design blocks, but also to support system level verification of connected IP blocks from ultimately multiple vendors. As an important part of VIP, assertions describe the temporal actions on block level interfaces.

This paper describes how to employ assertions with the state-of-the-art *hybrid verification* tool Magellan, which combines simulation and formal engines into an effective bug-hunter tool.

Based on an existing standardized on-chip interface protocol, SystemVerilog Assertions (SVAs) were written to constitute the VIP of that protocol. We then show how these assertions are employed with Magellan to prove the interface protocol compliance of two different design blocks attachable to the interface, without needing to write any test benches for the design blocks. This process was also a part of validating the VIP.

In this work, Magellan is primarily used to hunt bugs in existing designs - and when no more bugs can be found – formally prove that the design complies with the bus protocol. We also demonstrate how Magellan can be used to generate a stand-alone test bench for a design module, only using the assertions. This test bench environment includes a number of automatically generated *constrained random* test cases, yielding full assertion coverage. This test bench environment can be extracted from the Magellan setup and used as a design regression with VCS to verify that the module continuously complies with the protocol.

Table of Contents

1.0 Introduction 3
2.0 Assertion Based Checker as Verification IP 4
3.0 Verification of Protocol Compliance using the WB Checker Module 10
4.0 RTL Design Supported by Magellan and the WB Checker Module 16
5.0 Conclusions and Recommendations 18
6.0 Acknowledgements 18
7.0 References 18

1.0 Introduction

Design reuse of Intellectual Property (IP) is today a commonly used approach to decrease design and verification time, by using already verified design blocks in multiple chip designs. The IP reuse strategy heavily relies on standardized on-chip interfaces between IP blocks, shifting the verification problem from the block level to the system level. Verification IP (VIP) has emerged to support not only verification of protocol compliance of design blocks, but also to support system level verification of connected IP blocks from ultimately multiple vendors. As an important part of VIP, *assertions* describe the temporal actions on block level interfaces.

The SystemVerilog Assertion (*SVA*) language subset is becoming a de-facto standard for writing temporal checkers for both simulation and formal analysis. SVAs today integrate easily with Synopsys' VCS when simulating Verilog, SystemVerilog and VHDL.

Formerly, assertion based verification (*ABV*) was focused on simulating the assertions. Compared to classic directed testing, the ABV methodology improves the efficiency of the verification by enhancing the observability of critical locations inside the design under verification, making it easy to capture bugs and their origin. The problem with simulation based ABV remains that the verification engineer has to write or generate enough stimuli to exercise all critical situations for each and every assertion. *Assertion coverage* is advantageous to measure how well assertions are exercised, but full coverage does not guarantee that all critical situations have been exercised.

A very advantageous use of assertions is to prove these formally. Once a proof has been obtained, no more simulation cycles are required to verify a single assertion, as it will always hold with the given constraints. However, even if proved formally on modular basis, assertions should be employed as well in system level simulations, as this might reveal bugs not found on modular basis due to over-constraining.

EDA tools able to formally prove assertions have been available for some years, but design size limitations, poor temporal languages and tool usability have been limiting factors in obtaining a wide use of formal tools throughout the industry. Recently, the *hybrid formal* tool Magellan has been introduced, addressing these limitations. Magellan comprises both simulation and formal techniques in a single, fully integrated easy-to-use tool, which supports both SVAs and most hardware description languages.

This methodology case study investigates how an SVA based checker module for a standardized protocol is composed to support both simulation and formal methods. The checker is written to obtain maximum value from using Magellan, and to promote reuse while allowing the protocol checker assertions to be used both as constraints, checkers and coverage goals.

The SVA based checker module is described in section 2.0, along with guidelines which promote good assertion coding practices for formal verification. Section 3.0 shows how the developed assertion checker module is applied in order to verify the protocol compliance of a design IP block, an Ethernet MAC. The value of the hybrid formal approach of Magellan will be discussed

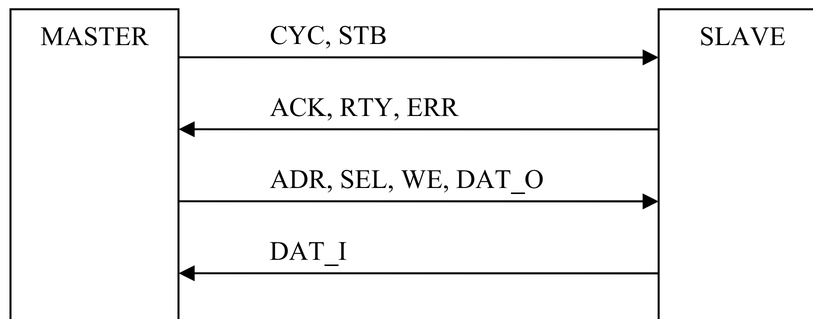
here, compared to classic formal verification tools. Section 4.0 shows how the design of a novel RTL core initially is verified using the developed assertion checker module.

2.0 Assertion Based Checker as Verification IP

In this methodology study we chose to build an SVA checker for the SoC on-chip WISHBONE [3] protocol (hereafter *WB* protocol). The *classic* subset of this protocol is fairly simple, but contains sufficient features to support a checker study. Other SVA checker studies have also been carried out while focusing on this protocol [5].

2.1 Classic WISHBONE Protocol

The protocol is designed for point-to-point SoC connections between a master and a slave unit. Any number of masters and slaves can be implemented with the sufficient amount of master arbitration logic and slave response multiplexing logic. The WB protocol as used in this work contains the signals listed below. Note that the WB protocol optional *lock* and *tag* signals are unimplemented.



In short, a master initiates a transaction by asserting the CYC signal. While CYC is asserted, the master can chose to send one or more transactions using the strobe (STB) signal, to which the slave responds after a number of cycles by asserting ACK, RTY or ERR. The master uses WE to request a write to or a read from the slave with the indicated address and byte select signals (ADR, SEL). For a detailed description of the protocol signal functionality refer to [3].

2.2 Reusable WB Checker Module with Multiple Uses

The WB assertion based checker module developed in this work is usable both as

- Constraint
- Checker
- Coverage goal

The checker is usable both for WB master and slave ports, for formal analysis and simulation, and is configurable with regards to WB bus latencies. These features, and how they are used, are described in the following.

2.3 General Naming Conventions

SVA based checker modules typically contain numerous identifiers and labels needed to construct the assertions. In this work, we employ a strict coding convention, identifying the various Boolean expressions, SVA *sequences* etc. inside the checker module, to support easy coding and understanding of the checker. Furthermore, such conventions are very important to ease debug and hierarchical browsing of checker modules, and supports tool selection of e.g. assertions (but not sequences and properties) to be activated in the checker. The following conventions are used.

Label	Description
s__wbs__<rule_ID> s__wbm__<rule_ID>	SVA *sequence* used to compose SVA *property* or *sequence* for slave/master *rule_ID*.
s__wbs__<rule_ID>_antc s__wbm__<rule_ID>_antc	SVA *sequence* constituting SVA implication *antecedent* (implication left hand side) for slave/master *rule_ID* Used to build SVA *property*.
s__wbs__<rule_ID>_cons s__wbm__<rule_ID>_cons	SVA *sequence* constituting SVA implication *consequent* (implication right hand side) for slave/master *rule_ID*. Used to build SVA *property*.
p__wbs__<rule_ID> p__wbm__<rule_ID>	SVA *property* used to construct SVA *assertion* for slave *rule_ID*. SVA *property* used to construct SVA *assertion* for master *rule_ID*.
a__wbs__<rule_ID> a__wbm__<rule_ID>	SVA *assertion* for slave *rule_ID*. SVA *assertion* for master *rule_ID*.
c__wbs__<rule_ID> c__wbm__<rule_ID>	SVA *cover property* for slave *rule_ID*. SVA *cover property* for master *rule_ID*.
no prefix	Checker input signal, internal signal and/or Boolean expression used to construct SVA *sequence*.

General guideline: Use a firm labeling convention to identify SVA *sequences, properties, assertions* and *cover properties*.

General guideline: *Sequences* should be built of Boolean expressions and/or input signals. *Properties* should be built of *sequences*. *Assertions* and *cover properties* should be built of *properties.* For simple rules, or if special conditions apply, some constructs can be bypassed, e.g. omitting *sequences* and building *properties* directly from Boolean expressions.

2.4 Master / Slave Ports & Naming Conventions

The WB assertion based checker developed in this work can be connected to both WB master and slave ports. The naming of the checker input signals follow a convention which makes the checker master/slave neutral, and the assertions inside the checker follow a specific convention to identify whether a certain assertion is a master property or a slave property. In our work, the following rules were extracted from the WB specification to fully capture the WB protocol:

Group	Label	Description
WB Master Assertions	a__wbm_R3_25	No master STB without a CYC
	a__wbm_R3_75_1	ADR, SEL, WE, DAT_O stable during transaction
	a__wbm_R3_75_2	STB stable until slave acknowledges
	a__wbm_R3_75_3	CYC must be followed by STB within configurable no. of cycles
WB Slave Assertions	a__wbs_O3_10	No slave acknowledge without a STB
	a__wbs_R3_35	Slave should acknowledge STB within configurable no. of cycles
	a__wbs_R3_45	ACK, RTY, ERR must be mutually exclusive

The assertion labeling is based on the rule name from the WB specification, and whether the assertion applies as a rule for either WB master or slave ports. This allows the user of the checker to select either the master or the slave rules inside the checker by using a wildcard (*a__wbm_** or *a__wbs_**). In section 3.2, we show how this labeling method is used together with Magellan in order to select e.g. the master assertions to be used as constraints and the slave assertions for checkers.

For all rules, cover properties were also implemented. These are prefixed by *c__wbm_** or *c__wbs_**, respectively.

General guideline: Use a firm labeling convention, making it easy to identify the rule(s) in the specification captured by the individual assertion.

General guideline: When an assertion checker module is usable for multiple interface flavors (e.g. master or slave), use a labeling convention to identify the flavor by a wildcard mechanism.

2.5 Macro Properties

To get the maximum value from using formal verification tools, assertions should be built using implications. This allows the tools to conduct a reach-ability analysis for the implication *antecedent*, while simultaneously proving the implication *consequent*. To ensure a safe SVA modeling style without redundant typing, this work is based on a small – but efficient – set of implication property macros. This allows easy configuration of the circuit reset method (asynchronous or synchronous) and ensures correct implication modeling, depending on whether an assertion or a coverage property is implemented.

Per the WB specification, all WB devices should use a synchronous reset. Nevertheless, the WB implementations analyzed in this work were designed using asynchronous reset. To avoid hard-coding the reset method in the checker module, a parameter to the checker module named *async_reset* is used to select the used macro property.

The macro properties used in this work are defined as shown below. The *p_impl* macro is used for assertions, on the form *assert property (p_impl(antc, cons))*. The *p_cov* macro is used for coverage properties, on the form *cover property (p_cov(antc, cons))*. The reason for the *p_cov* implementation shown below is to obtain a reach-ability analysis of the antecedent followed by the consequent. This is discussed in the context of Magellan in section 3.4.

```
generate if (async_reset == 1) begin

      property p__impl(a, c);
         @(posedge clk) disable iff (rst) a |-> c;
      endproperty

      property p__cov(a, c);
         @(posedge clk) disable iff (rst) a ##0 c;
      endproperty

end else begin

      property p__impl(a, c);
         @(posedge clk)
            (~rst throughout a) |-> (rst or c);
      endproperty

      property p__cov(a, c);
         @(posedge clk)
            (~rst throughout (a ##0 c));
      endproperty

end
```

A property using these macro properties is easy to compose. The assertion below describes that a master shall keep STB asserted until the slave responds by asserting one of ACK, ERR or RTY.

```
wire [2:0] ack_signals = {ack, err, rty};
wire       all_ack     = |ack_signals;

sequence s__wbm_R3_75_2_antc;
   stb & ~all_ack;
endsequence
sequence s__wbm_R3_75_2_cons;
   ##1 stb;
endsequence

a__wbm_R3_75_2 : assert property (p__impl(s__wbm_R3_75_2_antc,
                                          s__wbm_R3_75_2_cons));

c__wbm_R3_75_2 : cover property (p__cov(s__wbm_R3_75_2_antc,
                                        s__wbm_R3_75_2_cons));
```

By using the described macro property style, the clock signal and the reset method is not modeled for each and every property. This prevents errors and enforces a strict assertion

modeling, as the reset method and clock typically are the same across multiple or all properties in a certain checker module. Also, a coverage property has been added to limit the checker to be used for reach-ability search only.

General guideline: Use macro properties to abstract assertions from reset and clock, and to enforce reset modeling styles.

It should be noted that multiple pre-coded assertion libraries exist, also with implication macros. These assertion libraries are implemented as Verilog modules, and used by *binding* the checker modules to design and interface signals. In this work, we chose to only use plain SystemVerilog assertions, as the SVA language itself is very powerful. Also, we wanted to explore how Magellan interprets the various SVA constructs. Furthermore, the modeling style we chose using the *p__impl* and *p__cov* macros cannot be obtained using any existing pre-defined assertion libraries, as these do not accept SVA sequences as arguments.

2.6 All-Permutations Coverage

During this work we realized that ordinary checker assertions are insufficient when trying to obtain assertion coverage. For instance, consider the following WB assertion, describing that a WB master must assert STB not later than a configurable number of clock cycles after CYC has been asserted, and before CYC is de-asserted. Note that this assertion does not forbid multiple STB transactions from happening within a single asserted period of CYC.

```
sequence s__wbm_R3_75_3_antc;
   1 ##1 $rose(cyc);
endsequence

sequence s__wbm_R3_75_3_cons;
   cyc throughout ##[0:latency] stb;
endsequence

a__wbm_R3_75_3 : assert property (p__impl(s__wbm_R3_75_3_antc,
                                          s__wbm_R3_75_3_cons));

c__wbm_R3_75_3 : cover property (p__cov(s__wbm_R3_75_3_antc,
                                          s__wbm_R3_75_3_cons));
```

The cover property above is satisfied if STB just happens between zero and *latency* clock cycles after CYC has risen. However, when analyzing coverage and reach-ability, it is relevant to check for all permutations of the assertion, e.g. all possible delays between zero and *latency*. With that in mind, we rebuilt the checker as shown below.

```
sequence s__wbm_R3_75_3_antc;
   1 ##1 $rose(cyc);
endsequence

sequence s__wbm_R3_75_3_cons(latency1, latency2);
   cyc throughout ##[latency1:latency2] stb;
endsequence

a__wbm_R3_75_3 : assert property (
   p__impl(s__wbm_R3_75_3_antc, s__wbm_R3_75_3_cons(0, latency)));

generate begin
   genvar i;
   for (i = 0; i<=latency; i++) begin : c__wbm_R3_75_3
      c_i : cover property (
         p__cov(s__wbm_R3_75_3_antc, s__wbm_R3_75_3_cons(i, i)));
   end
end
endgenerate
```

A reach-ability analysis will now list all the permutations of the protocol for the WB 3_75_3 master rule which can be reached. Magellan will produce simulation traces for each and every permutation which is reachable. Interesting are also the permutations which cannot be reached, either due to limitations in the WB implementations or constraints imposed by the environment. Such results are valuable during the verification process when comparing the specification against the actual implementation.

General guideline: Use a *generate* statement to create coverage properties capturing all permutations of a protocol.

2.7 Limitations and Pitfalls for Using SVAs for Formal Analysis

During this work, we collected a few coding rules for assertions for formal analysis. More rules can be found in [5].

2.7.1 Always use Implications

To optimize the usability of formal analysis, a tool must be able to recognize the *antecedent* and *consequent* of an assertion. Therefore, it is not advised to use the basic Boolean form *~a* | *a & c* instead of the implication operator *a* |-> *c*. If the tool cannot isolate the antecedent, vacuity cannot be detected, which in turn allows the tool to claim a property always to hold if the antecedent constantly is false.

2.7.2 No Nested Properties

With the current version of Magellan, it was not possible to nest properties. As implications can only appear inside properties, it was not possible to build nested implications. However, nested sequences are possible, which is the most common thing to do.

2.7.3 System Function Calls

When using system function calls such as *$rose*, *$past* etc, the engineer must be careful not to look back into "negative" time or back past a reset, as this might lead to unpredicted results. Therefore, the assertion in some cases must be modified beyond what the basic user would expect. For instance, the implication *$rose(a)* |-> *b* must be pre-pended in time to prevent looking back in negative time or back over a reset: *1 ##1 $rose(a)* |-> *b*.

2.7.4 No Infinite Delays

Avoid using the SVA infinity operator, for instance when creating an implication like *a* |-> *c1 ##[0:$] c2*. This is a liveness property, which cannot be proven formally. In special cases however the infinity operator is advantageous to use, if another part of the SVA temporal expression limits an infinite range, e.g. *a* |-> *(c1 ##[0:$] c2) within c3[*10]*.

2.7.5 Keep it Simple

Be sure to write the SVAs for formal analysis as simple as possible, with as few delay ranges as possible. This will ease the job for the formal tool. For instance, complex properties can be broken up into several smaller properties – or might just be simplified while preserving the functionality, as shown in the example below.

Complex property for WB master, describing that STB must remain asserted until acknowledge happens:

```
1 ##1 ($rose(stb) | stb & $past(all_ack)) |->
   stb throughout (~all_ack[*0:latency] ##1 all_ack)
```

Same functionality is obtained by using a far simpler property:

```
stb & ~all_ack |-> ##1 stb
```

3.0 Verification of Protocol Compliance using the WB Checker Module

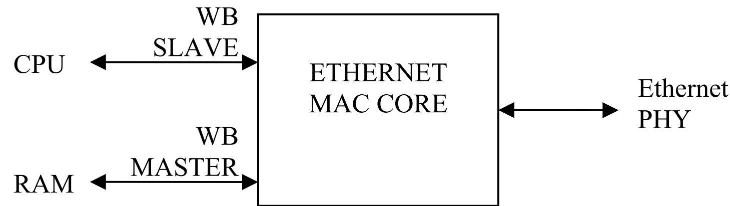
The goal of this section is to use the developed WB checker module to verify that an RTL IP design complies to the WB interface specification, using hybrid formal verification with Magellan.

3.1 Ethernet MAC

An Ethernet MAC core [6] modeled in Verilog was used as a case study of how to verify protocol compliance of a piece of design IP, while using the WB checker module described previously. This case study also supported the validation of the WB checker module.

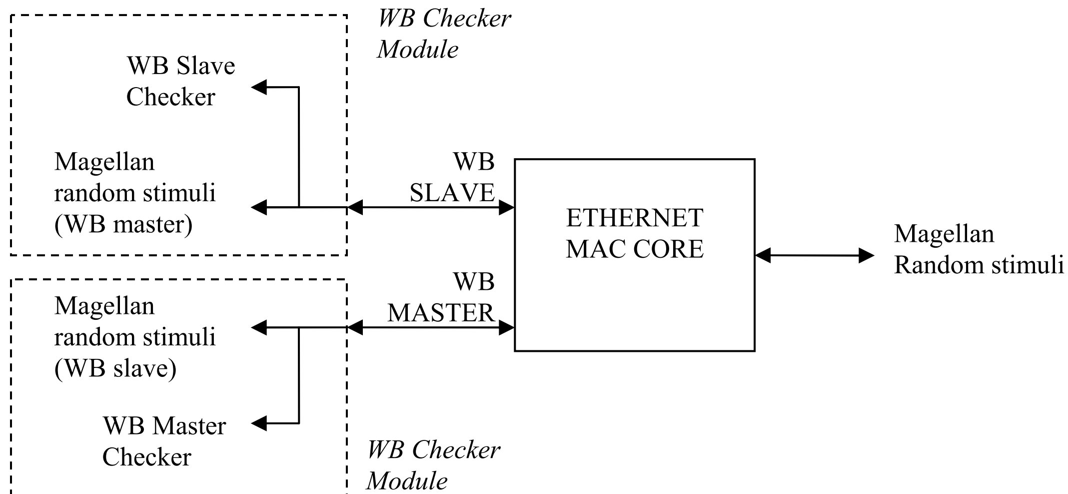
Simplified, the Ethernet MAC core has three interfaces: A WB slave interface connecting to the system CPU for control of the MAC core, a WB master interface connecting to the RAM

containing the data being transferred through the Ethernet port, and finally an interface to an Ethernet PHY.



3.2 Using WB Checker Module for Constraints and Checkers with Magellan

In order to constrain Magellan to comply to the protocol on the two WB interfaces, the WB checker module is connected (using the SystemVerilog *bind* construct) to the two WB interfaces, where they act both as constraints and checkers. The PHY interface is constrained as well, but that is beyond the scope of this document. The use of the WB checker module is depicted below.



On the WB slave port, Magellan has to behave as a WB master and check the slave behavior. On the WB master port, Magellan has to behave as a WB slave and check the master behavior. By using the labeling conventions from previous sections, this setup is easily configured in the Magellan project file:

```
# WB master assertions act as constraints for the WB slave interface
define_property const_wbs
add_property_info const_wbs -constraint –scope *wb_sva_slave_i.a__wbm*

# WB slave assertions act as checkers for the WB slave interface
define_property prop_wbs
add_property_info prop_wbs -checker     -scope *wb_sva_slave_i.a__wbs*

# WB slave assertions act as constraints for the WB master interface
define_property const_wbm
add_property_info const_wbm -constraint -scope *wb_sva_master_i.a__wbs*

# WB master assertions act as checkers for the WB master interface
define_property prop_wbm
add_property_info prop_wbm -checker     -scope *wb_sva_master_i.a__wbm*
```

3.3 Finding Bugs and Proofs with Magellan

Once the above directives have been applied along with a few other environment constraints, Magellan is invoked. The hybrid formal search engines inside Magellan simultaneously start to

- Prove reach-ability of the checker antecedents, both by formal search and hybrid search
- Prove the checker consequents by formal search
- Perform bug hunt, i.e. try to obtain counterexamples of the checker consequents by using hybrid search

The Magellan run is completed when the antecedent for each checker has been proved to be (un)reachable, and the consequent has been (dis)proven. When counterexamples are obtained, Magellan replays these in the VCS simulator from the reset state, and displays the waveform leading to the bug. The engineer can then investigate whether the bug trace is due to a design fault or an erroneous assertion.

It is interesting to look at the counterexample waveforms produced by Magellan for the Ethernet MAC core. Activity is happening on the WB slave port immediately after reset, as the Magellan hybrid search engines generates random stimuli, acting as a CPU programming the Ethernet MAC core in a random fashion. The WB master port is quiet just after reset, as the core is disabled. After a number of random cycles, the Ethernet MAC core registers have reached a certain value by the random programming, and the core starts to read and write data on the WB master port. Due to this internal functionality of the Ethernet MAC core, the hybrid search extremely quickly finds any bugs in the implementation of the WB slave port logic, whereas it takes more time to find bugs in the WB master port logic.

The Magellan hybrid bug-finder architecture shows its superior strength in several situations when compared to traditional formal-only proof engines:

- Bugs are found extremely fast. When using a traditional formal-only tool, a formal disproof has to be found, which typically takes some hours. With the Ethernet MAC core,

the hybrid search engine typically traps a counter-example within minutes by doing a mix of simulation and formal search.

- When writing assertions, the verification engineer often creates minor errors, which have to be fixed. When no hybrid search is performed, the engineer may have to wait many hours before a counterexample is produced by a formal-only tool. By using a hybrid search tool, the counterexamples are produced within minutes when working on commercial design sizes. This gives the verification engineer a development turn-around time which is an order of magnitude smaller than when using formal-only tools.

- The counterexamples produced by formal-only tools may not be reachable in real simulation due to an under-constrained model (*false negatives*). The hybrid search tool either finds bugs directly from a valid reset state, or it ensures that counterexamples found formally can be replayed in the simulator before reporting the bugs to the user.

When writing assertions and trying to prove these with Magellan, the small turn around times ensures a smooth working flow when fixing bugs in the assertions and the design. Once the design is close to bug free and the assertions are correct, Magellan starts working for longer periods of time to obtain real formal proofs similar to traditional formal-only tools. First, when such real formal proofs have been obtained, the engineer is done with his or her work, as few bugs might not be discoverable by the hybrid approach.

Magellan did find a single bug in the Ethernet MAC core implementation: If the RAM WB slave unit acknowledges later than the clock cycle where STB is asserted, the master WB interface breaks the protocol. This bug probably never was found using traditional verification by the designer, as his RAM test bench always responds immediately, but the bug would have been critical once a slower RAM was attached to the core.

Beyond the single bug found, Magellan formally proved all remaining checker assertions to be correct, i.e. the Ethernet MAC core is proven to be WB protocol compliant.

It took one person with no prior knowledge of the Ethernet MAC core less than half a day to download the Verilog design IP, create the Magellan setup script, attach the WB checker module and create a reset sequence. Once Magellan was invoked, we had the bug trace within 10 minutes, showing the WB master defect. A session performing formal proofs of all checker assertions took less than 30 minutes on an average single-processor Linux machine with 1 GB RAM.

3.4 Obtaining Assertion Coverage using Magellan

In order to analyze the Ethernet MAC Wishbone interfaces, we also used Magellan with the coverage properties set up in the WB checker module. The Magellan commands used for this are:

```
# WB master assertions act as constraints for the WB slave interface
define_property const_wbs
add_property_info const_wbs -constraint –scope *wb_sva_slave_i.a__wbm*

# WB slave coverage properties for the WB slave interface
define_coverage cov_wbs
add_coverage_info cov_wbs -scope *wb_sva_slave_i.c__wbs*

# WB slave assertions act as constraints for the WB master interface
define_property const_wbm
add_property_info const_wbm -constraint -scope *wb_sva_master_i.a__wbs*

# WB master coverage properties for the WB master interface
define_coverage cov_wbm
add_coverage_info cov_wbm -scope *wb_sva_master_i.c__wbm*
```

Note how the master and slave assertions (prepended by *a__**) are used to constrain the Magellan search engines and random stimuli, similar to the assertion checking in section 3.2. The coverage properties are prepended by *c__**.

When Magellan is invoked, it starts reach-ability analysis runs for all coverage properties in the WB checker module. These properties were written covering all permutations of the protocol, respectively 104 cover properties for the WB slave port and 15 cover properties for the WB master port. These figures depend on the allowed maximum bus latencies in various situations.

Within one hour of compute time, Magellan had found 63 coverage properties to be reachable, 43 to be unreachable, and 13 to be not determined. Later, these 13 properties were found to be unreachable. Reach-ability traces to the 63 reachable coverage properties were generated by Magellan for later inspection.

3.4.1 SVA Implications with the COVER Construct

From reading the SystemVerilog language LRM [1], one would expect that the SVA keyword *cover* should be used with an SVA implication, like in the *cover_DONOTUSE* example below.

```
assertion: assert property (
   @(posedge clk) disable iff (rst) a |-> c
);

cover_DONOTUSE: cover property (
   @(posedge clk) disable iff (rst) a |-> c
);

cover_OK: cover property (
   @(posedge clk) disable iff (rst) a ##0 c
);
```

However, this leads to misleading results with Magellan, as the tool interprets the *assert* and *cover* statements as follows:

SVA *assert*: FAILURE = a and (not c)

SVA *cover*: COVER = (not a) or (a and c)

Whereas the *assert* failure interpretation is trivial, *cover* realizes the temporal SVA equivalent of the property implication. For Magellan to detect a success for an implication inside a *cover* statement, it needs to prove that the antecedent is reachable and that the COVER equation above is true, either using a hybrid or formal approach. But this does not guarantee that Magellan produces the trace that interests us, namely the antecedent followed by the consequent. Rather, Magellan might just report a trace showing a case where the antecedent is not true, even without reporting that the success was *vacuous*.

This is why we chose to use the *cover_OK* implementation shown above, where the implication operator is removed. As the antecedent is concatenated with the consequent, Magellan is forced to produce a reach-ability trace where *a* is followed by *c*.

The SystemVerilog LRM also defines the *assume* statement, which can replace *assert* and *cover*. The LRM prescribes that *assume* can be used to describe constraints which are not assertions. However, this statement is not required with Magellan, as the tool is able to use assertions built using *assert* directly as constraints as shown in section 3.2. Furthermore, the prescribed methodology of reusing assertion checkers from one module as constraints for the next module opposes the use of the *assume* statement.

General guideline: Never use a *cover* construct on an implication property with Magellan.

General guideline: Never use the *assume* construct with Magellan.

3.5 Bounded Proofs from the Reset State

While trying to obtain a complete proof for an assertion, Magellan reports what we name *bounded proofs from the reset state*. Such a bounded proof tells the user that Magellan has found an assertion to be true for all possible simulations for a certain number of clock cycles, starting from the reset state. Once Magellan finds a real proof not bound to the reset state, the bounded proof from the reset state is discarded.

We find the bounded proofs from the reset state valuable indicators of the likelihood of an assertion to always hold, but the bounded proofs from reset do not produce sign-off verification quality. If Magellan only produces a bounded proof from the reset state, and not a complete proof, other means of verification must be employed to close the verification. This might be splitting the assertion into several other functional equivalent assertions, and proving these in separate Magellan runs.

In this work, we fortunately obtained formal proofs for all assertions where no bugs could be found.

3.6 Producing a Simulation-Only Regression Suite using Magellan

If the Magellan license availability is limited or the formal proof run time is very large, the design flow might call for an alternative easy-to-run regression suite, checking that the design module is in a sane state. Magellan can be used to produce such a simulation-only regression suite for the module under verification.

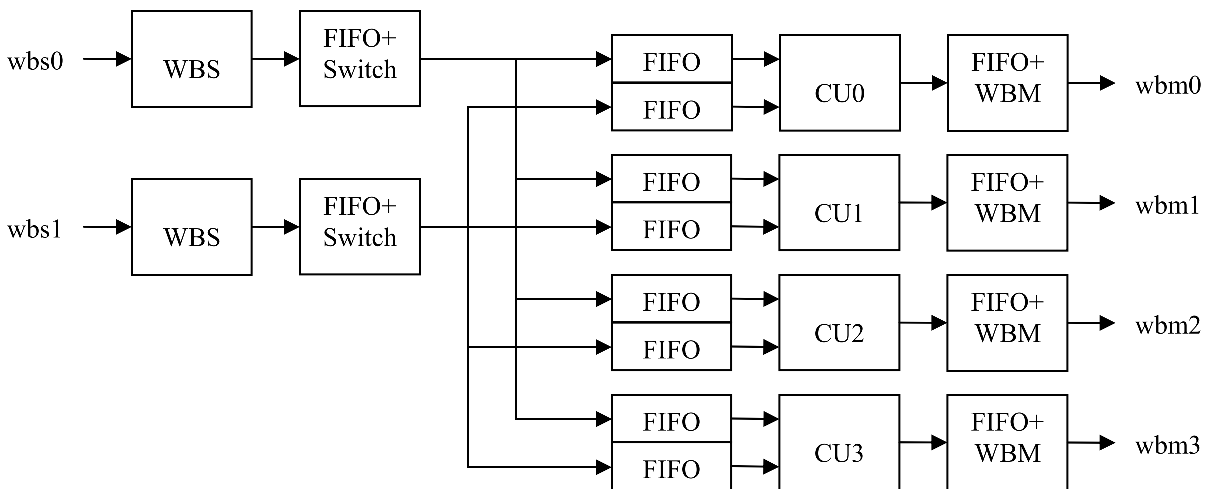
When Magellan is invoked, it performs a reach-ability analysis for all assertions, either by using the hybrid search engines or by performing a formal-only analysis. Once reach-ability has been obtained for all assertions, the Magellan *write_session_test* command is used to produce a test script. This test script can be called on the shell command line with options selecting the individual property to be exercised in the VCS simulator without using any Magellan licenses. A regression suite should invoke this test script for each and every assertion while checking that a non-vacuous match is obtained for each assertion – of course without detecting any assertion violations.

The obtained regression suite ensures that each assertion is exercised at least once – and given the antecedent, the implication consequent still holds. However, design changes might be introduced which makes the assertion fail in situations different from what is being exercised by the simulation regression test. Therefore such a regression test cannot be used as a verification sign-off criteria following design changes, this requires a full re-run of Magellan where formal proofs for all assertions can be re-obtained.

When assertion checker modules such as the WB checker are used for simulation-only ABV, all assertions should be used as checkers. The concept of using assertions as constraints does not apply here, unless used to constrain the random stimuli generated when applying coverage based constrained random verification.

4.0 RTL Design Supported by Magellan and the WB Checker Module

To support some methodology studies, a new design modeled in Verilog RTL was needed. We chose to do the data flow oriented application depicted below with two WB slave interfaces and four WB master interfaces. Incoming packages on the two WB slave interfaces are directed into four different computational channels, depending on the package target address. Once each computational channel has received a package from each WB slave interface, the computation is performed in the CU unit, and the result is submitted to the WB master interface for that channel.



Once the RTL designer has completed his/her first shot on such an RTL design, it is desirable to see that things are working as expected. Often, a designer writes a rudimentary test bench to obtain initial test runs. The test bench is typically discarded later without being reused in the regression suites, or for later designs.

With Magellan available, things get easy. We simply attached the developed WB checker to the six interfaces, applying a correct configuration of the slave/master capabilities. Less than an hour after the first RTL draft was completed, Magellan was running and producing random simulations to be inspected by the designer, resulting in the discovery of a few obvious design bugs to be fixed. Next step was to let Magellan prove the WB master/slave properties for all six ports, these proofs were fast to obtain. Also, the designer included *design assertions* capturing design hot-spots inside the RTL design. These assertions were proven as well in this initial design cycle.

As a result, a few hours after initial design RTL was ready, the designer had seen basic functionality of the design work well, and ensured correct interface behavior. At this point, the module was ready for initial system level integration, while the full module sign-off verification still remained to be done. As the application is data driven, we chose to employ VCS with constrained random verification (CRV) using SystemVerilog Test Bench (SVTB). Describing this task is beyond the scope of this document.

It should however be mentioned that *coverage holes* apparently not reachable while employing CRV on the design can be checked with Magellan. The hybrid search engines of the tool will quickly produce a reach-ability trace - if possible - to such a coverage hole, allowing adjustments of the solver constraints. Alternatively, Magellan would tell the user that the coverage hole in fact is unreachable, allowing adjustments of the coverage metrics.

Furthermore, the attached WB checkers are continuously employed in the CRV verification process to capture illegal stimuli produced by the CRV environment. When inserting the design block into a larger system and performing system level verification, the WB checker module

should continue to check all interfaces. This will ensure that the adjacent design blocks do not violate the WB protocol.

5.0 Conclusions and Recommendations

In this work we have investigated how assertion based checker modules can be written using SystemVerilog assertions. Such checker modules are reusable across designs, but also usable both as checkers, constraints and as coverage goals, and can be used both for formal verification as well as traditional assertion based verification using simulation.

The *hybrid formal verification* tool Magellan allows maximum benefit when using such SVA checker modules. The tool allows us to prove assertions, but also to create reach-ability simulations, find the real bugs easily and fast, and finally support easy bring-up verification of new designs.

Assertion based verification using SVAs is a key component of the verification flow of the future. The authors believe that the path to success is to create assertion based checkers and verification IP which is compatible with formal and *hybrid* methods to obtain competitive verification productivity.

6.0 Acknowledgements

Kind greetings go to Synopsys employees Jin Hou, Dan Benua, Martine Chegaray and - last but not least - Allessandro Fasan for their kind support of our work.

We also thank Stig Kofoed for reviewing our work.

7.0 References

[1] Accellera: SystemVerilog 3.1a Language Reference Manual, www.systemverilog.org
[2] Synopsys: Magellan User Guide, Version MG_2004.12-SP2
[3] WISHBONE Specification, www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf
[4] Roeder et al: Using Magellan to Optimize Functional Verification Turnaround Time and Quality. European SNUG 2005.
[5] Hou, Jin: Best Practices of Assertion Based Verification using SystemVerilog Assertions, VCS and Magellan. Tutorial, European SNUG 2005.
[6] Ethernet RTL IP design, www.opencores.org/projects.cgi/web/ethmac/overview

About SyoSil:

SyoSil is a consulting company holding broad expertise within the field of System-on-Chip and ASIC solutions, including specification, methodologies, design and verification. We are specialized on verification strategies, advanced EDA verification tools including formal methods (property checking) and SystemVerilog for RTL design, assertions (SVA) and object oriented testbenches (SVTB).