**Laboratory 02.2.- “Suffix Implication - Deep Dive”**

Authors: Diego Hernandez Ramirez  
**Date: 01/12/19**

# Objective

Experience yourself the beauty of Model Checking Tools by analyzing the Suffix Implication and vacuity problems.

# Material

1. PC
2. SEDA tool, FPV App

# Practical development

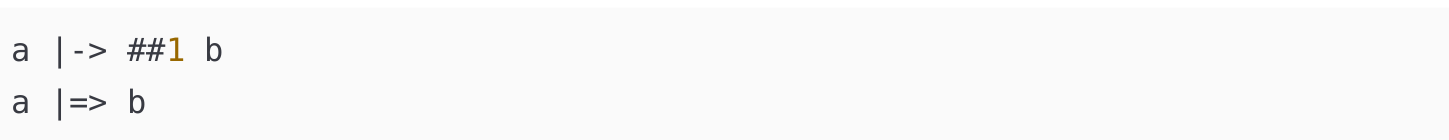
SystemVerilog Assertions (SVA) are a temporal language for model checking, whose declarations and semantics are similar to other Mathematical Temporal Logic languages. This does not mean that, anyone with interest in Formal Methods for hardware design and verification, should actually be an experienced mathematician to understand and apply such methods correctly.

As in most Temporal logic languages, a binary operator is used to express relations between atomic propositions, or speaking in SVA terms, a relationship between a sequence (antecedent), and its consequence (property).

**Material Implication**

The material implication, implication operator or simply "implies connective" is represented by **|->** for the overlapping operation, and **|=>** for the non-overlapping. The main difference is, since both operators are used in concurrent assertions (i.e., clocked assertions), the overlapping operator checks the consequent immediately after the antecedent is matched, whereas the non-overlapping checks the consequent one cycle after antecedent has a match.

Both operators are equivalent if used in this way:



As stated previously, this operator is used to express relationships of the form **p |-> q** which is read as "**if p then q**". It does **not** imply a causal relationship (i.e., if p is true, then causes q to be true" but an interpretation of the form "**if p is true, then it must be that q is also true**".

For example, to express this statement in SVA as an assertion: "*If the current state of the system is IDLE state, and Start signal is asserted, the FSM should move to the next state TRANSMIT in the next clock cycle*". This can be one solution:



If you started to ask, why using non-overlapping instead of overlapping operator?, as in:



The answer is, consistency. There may be case on which the consequent is checked after N cycles of the antecedent match. For example, this assertion: "The DONE signal is asserted between 2 and 8 clock cycles after WRITE is asserted", would be more readable in this case:



Rather than this another one:



**Lab Part I: Converting SymbiYosys Open Source immediate assertions to Symbiotic EDA Suite SVA assertions**

From [Dan's Gisselquist ZipCPU Blog, Ziptimer entry](https://zipcpu.com/zipcpu/2018/04/17/ziptimer.html), there is this code excerpt:



Which can be read as, "i*f r\_value is not empty or auto\_reload is asserted, then r\_running must be asserted*", which is very similar to the form "*if p then q*" or in SVA, *p |-> q* . From here, we can easily extract a sequence that works as an antecedent, and a property which will act as a consequent.

***Grade yourself:*** Convert that Verilog immediate assertion property to SystemVerilog SVA assertion.

* Hint: Remember all topics we have studied so far.
* Hint: Read the sentence again. The sentence implies the use of the suffix operator. Suffix is clocked because it works with concurrent properties.
* Is there any difference in the execution of both checks (immediate vs concurrent)?

As a rule of thumb, to convert from Open Source version of SymbiYosys to SVA, all logic before the

immediate assertions can be used as the antecedent of the implication operation (since is actually the sequence whose non-empty match triggers the check), and the logic inside assertion parenthesis, is the consequent (the property that needs to be satisfied).

This might not be always the case, so as a suggestion, always review if rephrasing the assertion in this manner actually makes sense, based on the "cause then effect" semantics of the logical implication.

**Part II - Vacuity**

The logical proposition **p |-> q** is **weak**, that means, whenever p is false, the check passes

vacuously (vacuous truth appears when the antecedent is not satisfied, therefore the model checker software cannot use it to infer the truth value of the consequent). This behavior conforms with the semantics of implication suffix in Temporal Logic, on where if the antecedent is false or no condition satisfies it, the evaluation is true because the set is empty and there is not counterexample of the contrary.

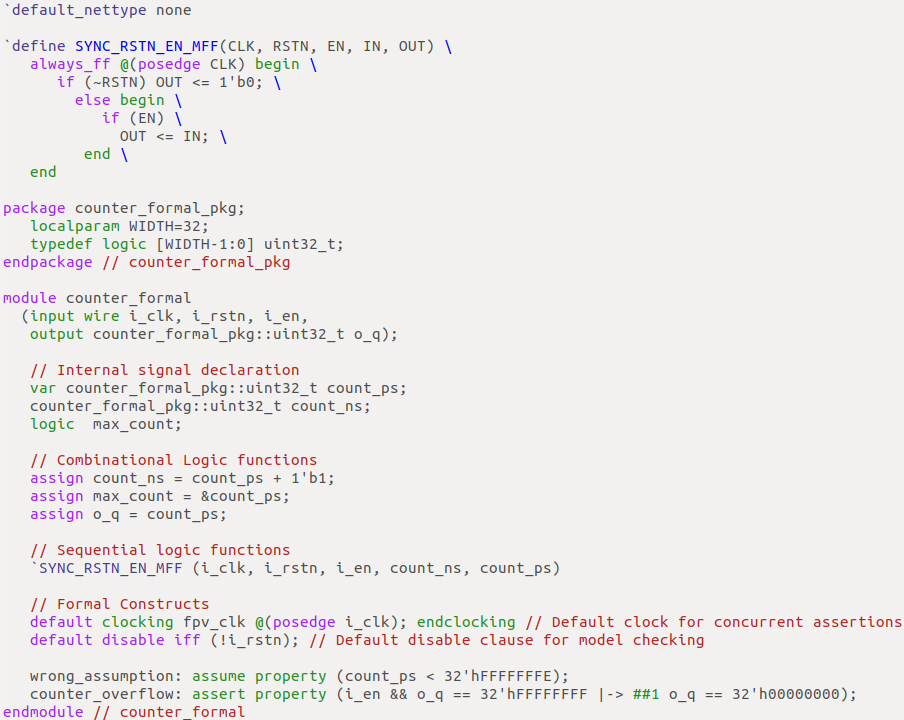
This does not means that the implication operator is difficult, or dangerous to use. Speaking in terms of SVA, at any moment an assertion passes vacuously, it might be for different reasons. In Formal Verification, the most important source of vacuity is:

* The model is empty (i.e., assumptions restrict the state space in a way in which the antecedent of an implication operator is removed from the model, or tautology exists).

In other words, the usefulness of the weak behavior of implication suffix in SVA lies in the fact that, in any moment an assertion results in a vacuous pass, it means our environment, or assumptions, or even sequences, are wrong.

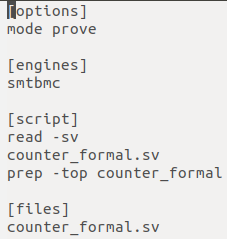
**Catching Vacuity: Manual Way**

Consider the following toy example written in fully synthesizable SystemVerilog:

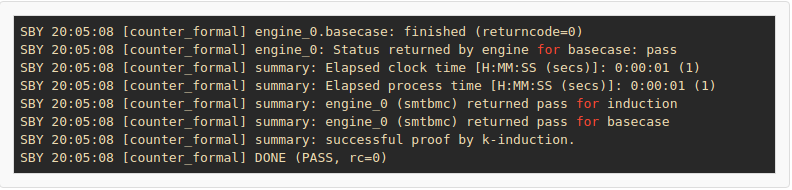


That RTL model represents a very simple but big binary counter whose state space is **restricted**, or **over-constrained** by the **wrong\_assumption** assumption.

By running it with the Symbiotic EDA tool, using this SBY file:



The result shown in the terminal says that there is a model, in other words, that the verification passed. But, analyzing the RTL again, we can see that there is an obvious issue. Since we are constraining the count\_ps register to be less than 32'hFFFFFFFE , we are causing a **vacuous pass.**

****

**Debugging Vacuity in Manual Way**

It is very simple to check if we are having an assertion passing vacuously. To do this, we need a **witness** of the actual trace of the property. In other words, instead of using an assertion, we will change it to a **cover** so we can look for satisfiability instead of validity, and also change the mode in the SBY file to cover too, as follows:

SBY file to witness a trace.



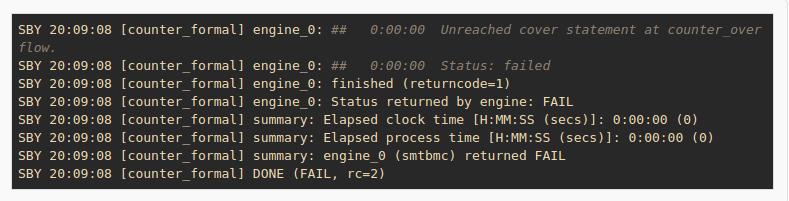
Changing the assertion to a cover statement.



The result is now different. This new message appears in the terminal: **Unreached cover statement**

**at counter\_overflow** , saying that *there is no path that satisfies our sequence*, clearly **falsifying** our

previous pass result.

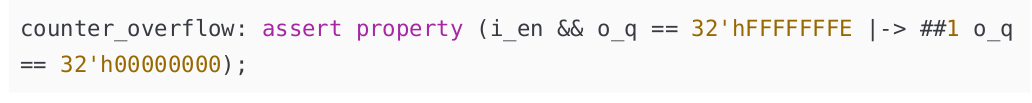


**Checking Vacuity Automatically**

Our assertion, which is shown again below for the sake of clarity, is composed of the sequence

(antecedent) i\_en && o\_q == 32'hFFFFFFFE and the property (consequent) o\_q ==

32'h00000000.



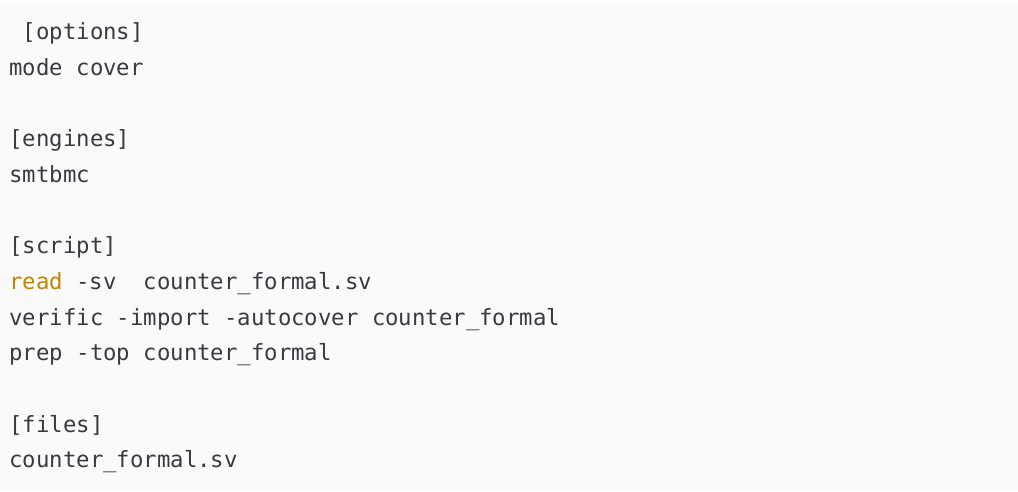
in order to avoid vacuous results, a cover property that witness a path that satisfies the sequence which composes both antecedent and consequent can be used. If this cover is unreachable, is because there is not such path that satisfies the sequence.

Such cover property can be written as follows:



Manually covering the sequence that satisfies the implication operator is not always easy, and errors can be introduced depending in how complex the properties and the designs are. Most model checking tools extracts properties which satisfies the sequence used in implications automatically.

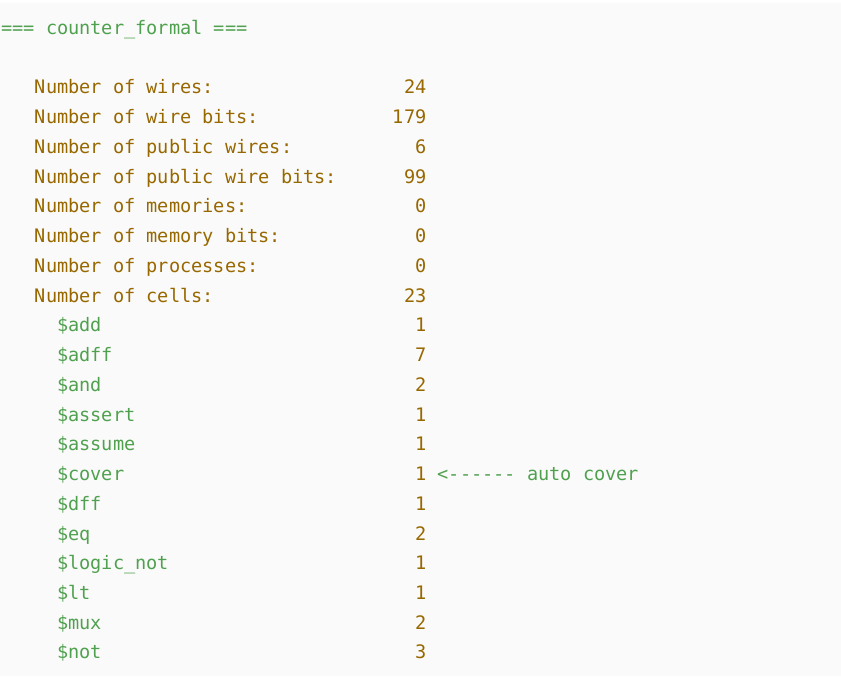
SymbiYosys Symbiotic EDA Suite version supports that feature too. To enable it, when importing Verific netlist to Yosys, adding **-autocover** argument in the SBY file will do the job. For instance, adding **verific - import -autocover counter\_formal** in our original SBY file as follows:



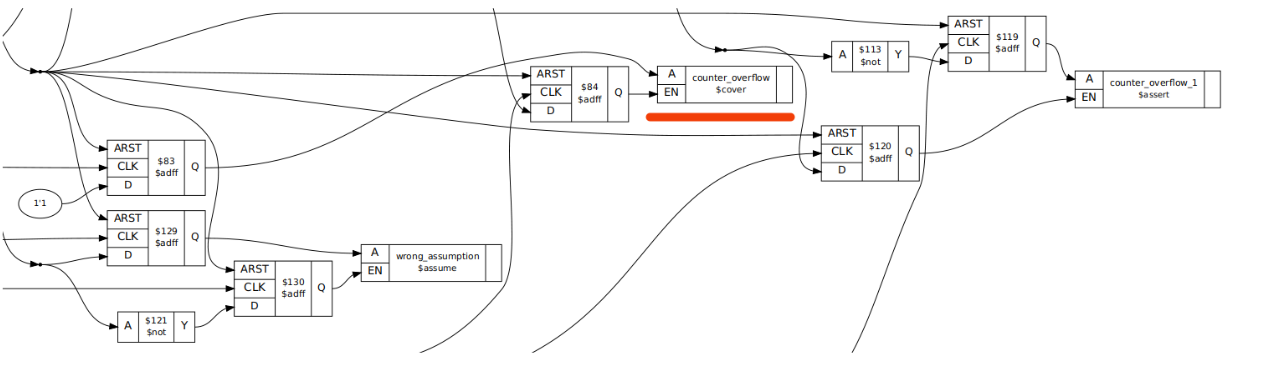
***Grade yourself:*** The **precondition\_check** shown above is known as a *witness trace***,** because it *witness* a case on where, starting from an initial state, there is a path that leads to the **antecedent** and the **consequent** satisfaction. This is useful when there are many ***assumptions or restrictions***involved in the assertion. But it uses more computational power that the ***late precondition***.

* Can you guess what ***late precondition*** is?
* Can you write a ***late precondition*** cover statement instead of the ***witness***?

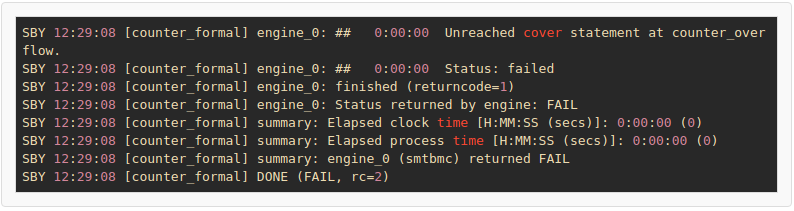
This will create a new $cover cell



Also, using the show command in Yosys, you can see the cell connected to the logic that is part of the main assertion, as shown below:



By running again SymbiYosys, we can see in the terminal an failure due unreachability, such as the one we had when adding the manual cover, but in this time, there is no cover property added manually anywhere in the SV file. In this example, the design is failing due the auto generated cover inserted into the netlist.

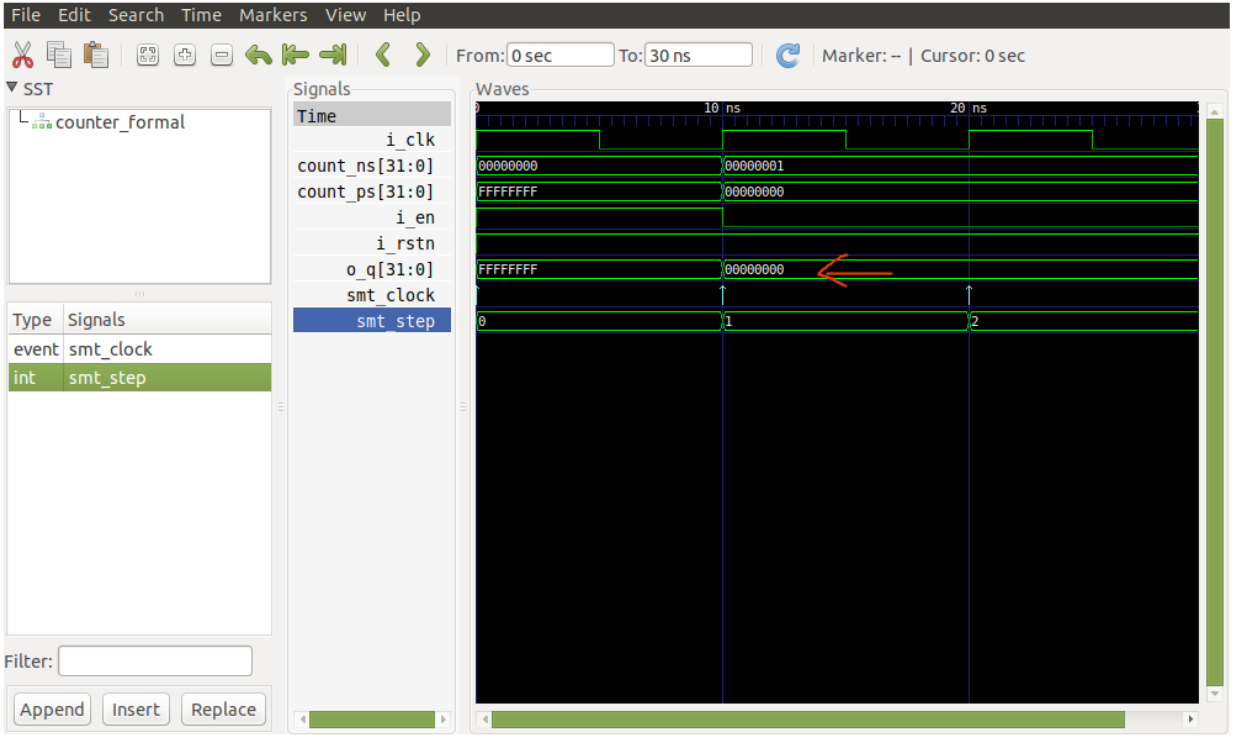


**Over-constraint**

The problem in this case is the assumption wrong\_assumption, because it over-constraints the design.

Such assumption is too strong, that it restricts meaningful model behavior.

Removing the assumption obviously solves the problem, and if you swap again the primary assertion into a cover statement, you gill get a trace (waveform) actually showing the expected result, as in the image below:



**Final Words and Suggestions**

The main purpose of this document is to understand what the implication operator does, how to use it and what are the important properties to consider when using it, such as vacuity. The example used here was rather simple, but the theory works exactly the same in all other designs.

As you might seen, swapping between cover and assert to analyze the properties can be tedious

depending on the number of properties in your design. A suggested, better way to work is to use

tasks.

For instance, one precondition task could work with verifying the auto generated cover

statements, and another task check will actually check the rest of the assertions, so if you have an unreachable problem in the precondition task, it will point out a problem, even if your assertions in check task passes.

A suggested SBY file is shown below:

