System Verilog Assertion (SVA) to RTL Synthesizer

Abstract

Assertion-based verification has become a widely accepted validation technique in the pre-silicon SoC design stage. Among available platforms, System Verilog assertion (SVA) is a popular medium for writing assertion-based properties and looking for security vulnerabilities early in the SoC life cycle. However, often such assertions cannot be transferred to the post-silicon validation phase as they cannot be synthesized into hardware description language (HDL) formats. In this project, we develop a synthesizer capable of transforming SVA properties into synthesizable hardware modules in Verilog register-transfer level (RTL). HDL realization can be transferred to the post-silicon domain and implemented as hardware monitors to check security properties for validation or in-field operation stage. Our synthesizer tool is developed with maximum scalability to handle complex high-level abstraction properties described in System Verilog into a coherent RTL form with a conditional statement. In this work, we delve deep into the structure of SVA properties and decompose the high-level abstractions into basic communicating parallel hardware units that can operate as a monitor for that property. These synthesized assertions can be used to locate errors, ensuring faster debugging. We verified the synthesized modules with extensive test benches covering all corner case scenarios for functional correctness.

Introduction:

To run essential applications and manage sensitive data, System-On-Chip (SoC) is increasingly being employed in diverse areas such as transportation, industrial automation, healthcare, and telecommunication. These systems frequently involve not only financial and economic interests, but also human lives, posing significant safety concerns. As a result, one of the most important challenges is the system's reliability and security. According to recent industry surveys, the verification and debugging procedure consumes 70% of design time. As a result, many verification strategies have been developed to construct hardware monitors that check logic and/or temporal behaviors against known features via signals/registers snooping. Several EDA tools are attempting to find a solution that provides for observability of a running design to detect flaws and debug the design. Most of these tools, on the other hand, do not utilize assertions to catch problems; instead, they set a trigger condition, and when it occurs, the software tool records the status of signals, registers, and memories, and sends it to a graphical waveform display.

On the other hand, while various studies have been conducted on how to synthesize assertions into hardware checkers, each one takes different method assertions are implemented. Das *et al.* employ three primary blocks to implement any type of assertion [1]. But the implementation is not effective enough in this domain. Design proposed by Kastelan *et al.* [2] is more space-efficient than [1]. But authors did not give any solution to the synthesis of "throughout" operator. Some of the work was done with IBM's FOCs Property Checkers Generator [3], which is no longer accessible. MBAC [4] is an academic prototype that converts PSL and SVA assertions into hardware monitors using an automata-based technique. Some of these technologies, however, are no longer available, and there is little information on others' effectiveness. Amin *et al. [*5] propose architecture of system Verilog assertions (SVA) synthesis compiler. The compiler converts the un-synthesizable System Verilog assertions, into synthesizable equivalent Verilog modules.

From the above-mentioned literature study, we can find that very few automated design approaches have been proposed to automatically synthesize assertions from high-level specifications. There is a scope to mitigate this research gap. With such motivation, the present version of the tool can perform following operations:

1. The tool translates un-synthesizable SVA to equivalent synthesizable Verilog module.
2. The tool can generate Verilog modules for various assertion operations. These assertions include AND, OR, XOR, multiple clock cycle delays.
3. The tool can handle irregular indentation and spacing

4. The tool can perform operations on multiple properties together
5. The tool can perform cases where there are multiple clock cycle delays presents in both antecedent and consequent statements

Tool Workflow:

The workflow of the tool can be divided into two categories: SVA parsing and RTL generation. Figure 1 shows an overview of the workflow of the tool. In the following sections, the workflow is elaborated.

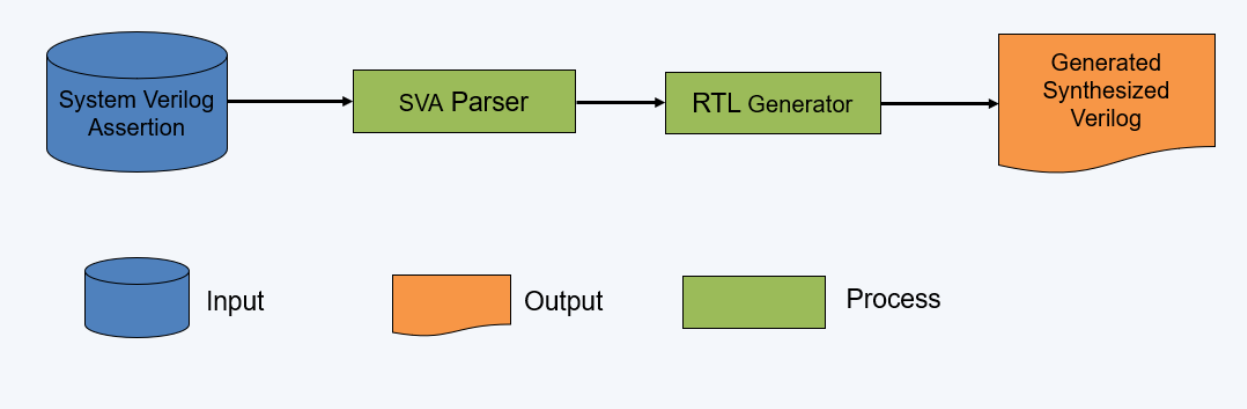

Figure: Overview of the workflow of tool

SVA parsing

As mentioned before, the tool takes system verilog assertions as input. The tool accepts these assertions written in any file format (..sv/.txt/.sva etc.). The first task of the tool is to extract required information from these assertions. With such motivation, a parser has been written in Python language. The parser finds out the required information from the SVA file and passes them to the RTL generator.
The parser extracts the following information:

- clock signal and its sensitivity
- presence of disable operation
- Implication operator
- antecedent statement
- consequent statement
- Inputs of the Verilog module
- Size of the input signals and registers
- Clock cycle delay

The parser finds out these sequence operators and operands by following these steps sequentially:

1. At first the parser removes the blank spaces and lines from the assertion statements.
2. Then it detects the main assertion statements and eradicates the declaration lines.
3. In order to deal with irregular and mission spacings, the parser forcefully impose space before every special logical-Boolean operator and other special signs (@, &&, >, ===, !=, etc.,).
4. Next, the parser detects the clock signal if present and detects its sensitivity (posedge or negedge)
5. Then, the parser tracks the presence of disable operations and puts a flag depending on the existence of the operation
6. Afterwards, the parser finds out the implication operator.
7. Based on the location of the implication operator the parser separates the remaining statement into two part: antecedent and consequent.
8. In the next few steps, the parser finds out the clock delay information from the antecedent portion and makes a list of antecedent sub-sequences separated by clock delays.
9. The parser repeats the step 8, for consequent sequence.
10. Finally the parser detects the input signals and registers from the full statement, and also their corresponding bit size.

RTL Generator

RTL Generator takes information from the parser on the number of input signals, signal names and signal width. These are used to instantiate the signals as input to the verilog module. Next it checks whether it is an immediate assertion or a concurrent assertion. (if immediate assertion parser returns an empty list as antecedent conditions).

Sequential/ combinational logic check is performed from the sensitivity list. According the sensitivity list the always block of the HDL is initialized (for combinational: always@(*) or always@(signal_sensitivity signal_name))

All the conditional blocks are declared inside the ‘always’ block. The HDL generator breaks all the conditional blocks into sequences of immediate branch conditions with temporary flag register to keep track of those conditions being fulfilled. For sequential cycle delay temporary flag register are conditioned using blocking assignment to keep track of consecutive cycles. The system first goes through the branch conditions and time delays for antecedent conditions and forward the

antecedent condition flag toward consequent conditions. A single clock cycle delay is added if non overlapping implication operator (|=>) comes up.
The system in a similar manner goes through the branch conditions in the consequent conditions and returns a HIGH pass value if all the consequent condition are satisfied otherwise returns a LOW pass value provided that the antecedent conditions were satisfied before.

User Interface

In order to make the tool more user-friendly, a graphical user interface has been developed using Python tkinter. A visual representation of the GUI is given in Figure 2. The GUI provides a user-friendly environment to the users to create synthesized Verilog files with the help of few clicks. In order to run the interface following steps are followed:

Figure 2: Overview of the user interface of the tool

Step 1: Open terminal in a folder where all python scripts (parser, RTL generator and GUI scripts) are present
Step 2: run by following statement
"python tool_app.py"
Afterwards, a Gui will be opened.

Step 3: Click “Browse” button to select file to generate synthesized RTL. After selecting the button, a new text box will be appeared that displays the property information. In the same time, synthesized RTL verilogs will be generated in the folder.
S**tep 4:** Click “Open” button to view generated Verilog module

Figure 3 shows a sample screenshot performed in step 3. If the input file has more than one property assertion, it will generate multiple verilog module files.

Figure 3: Step 3 of the running instructions of the tool

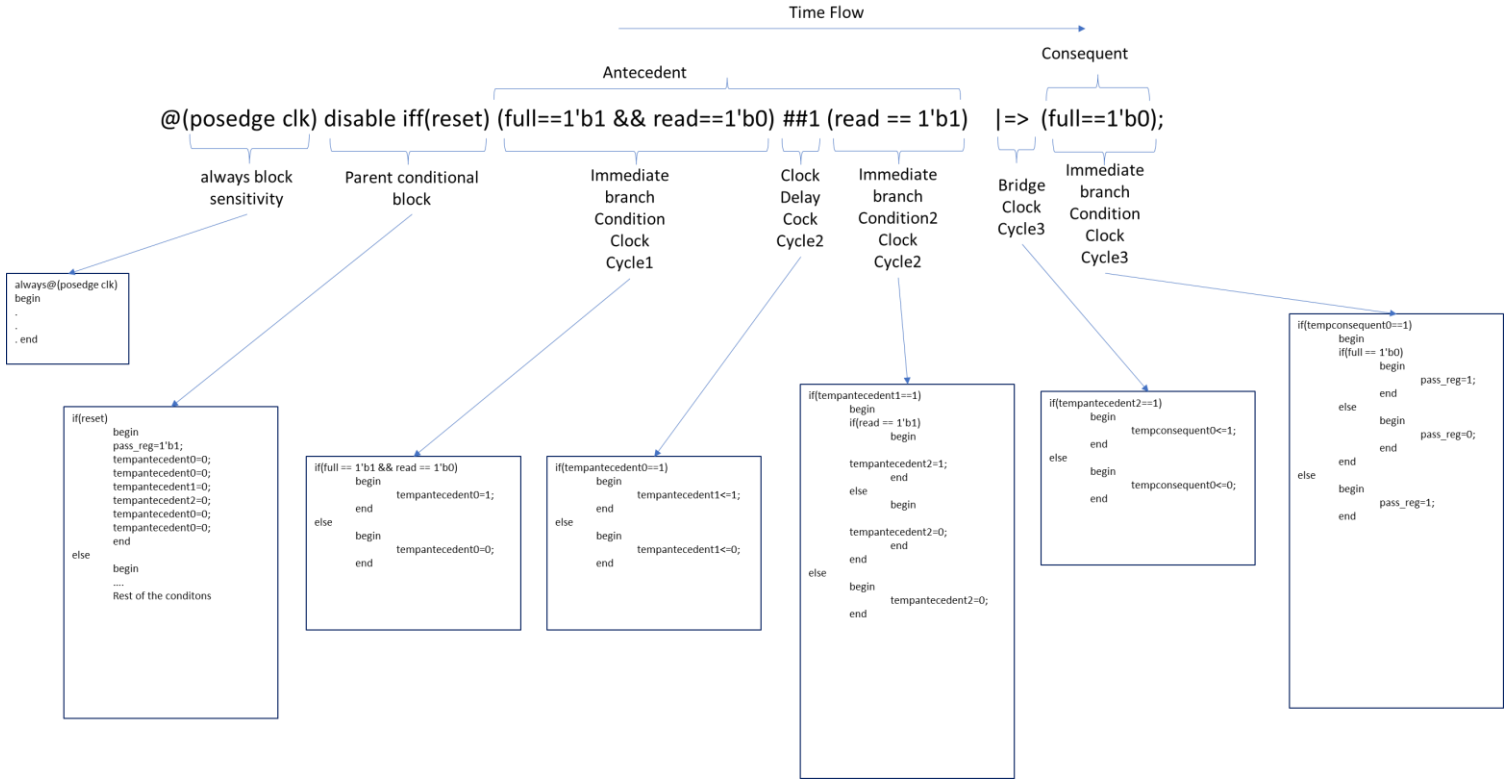

Figure 4: RTL generator workflow of the tool

Results

In order to evaluate the performance of the tool, in total SVA propertied have been provided. The tool can successfully generate synthesized Verilog files for all of these property files. ModelSim HDL simulator is used to check whether the generated file is working properly or not. All of the generated files have compiled successfully without any error and waveforms of simulation are analyzed. The tool can successfully perform operations on many corner cases not mentioned in the test suite

For example for the property 12 from benchmark file:

property 12;
@(posedge clk) disable iff(reset) (full==1'b1 && read==1'b0) ##1 (read == 1'b1) |=> (full==1'b0);

endproperty

The code generator creates an always block with associated ‘posedge’ sentivity of ‘clk signal’ as depicted in figure above. Then the system creates a parental condtional block which ignores all the input if reset is HIGH and only works when RESET is low. Then going through the antecedent conditions list a tempantecedent0 register is asserted to track the first antecedent condition occurring. And for the following ‘##1 operation a tempantecedent1 is assigned with non-blocking operator.

When tempantecedent1 is TRUE after one clock cycle we check whether the next branch condition “read==1’b1” is TRUE or FALSE which is tracked by the tempantecedent2 flag. Next moving on to the implication operator a one cycle delay is added by assigning tempconsequent0 with non-blocking operator which tracks that all the previous antecedent conditions has been true. Once this flag is true it checks for consequent condition “full==1’b0” which is the last branch condition and generates a Pass value.

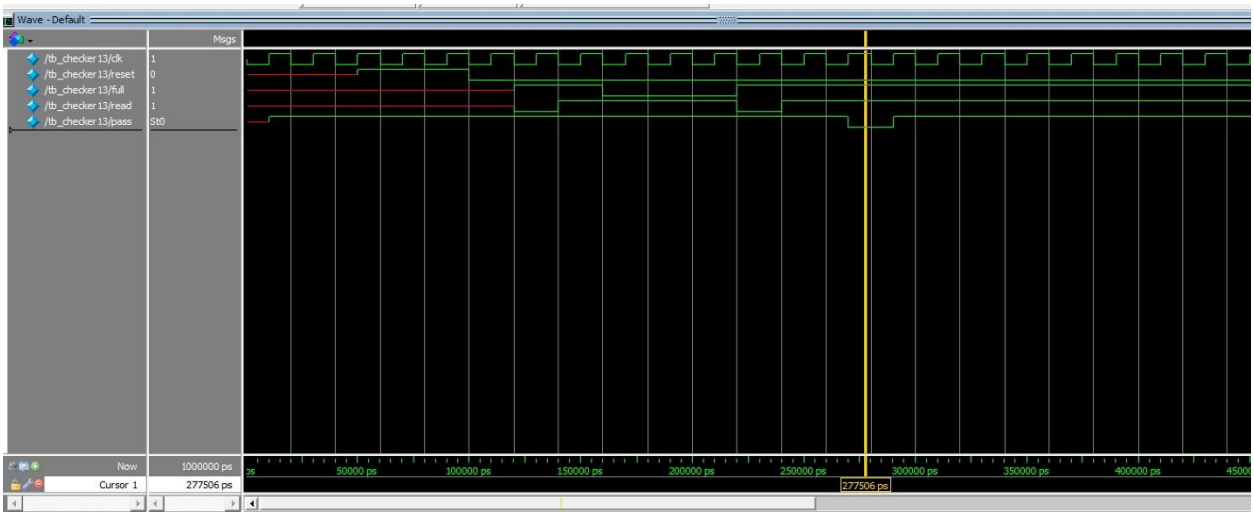

Figure 5: HDL functionality check for `property 12' of test suit

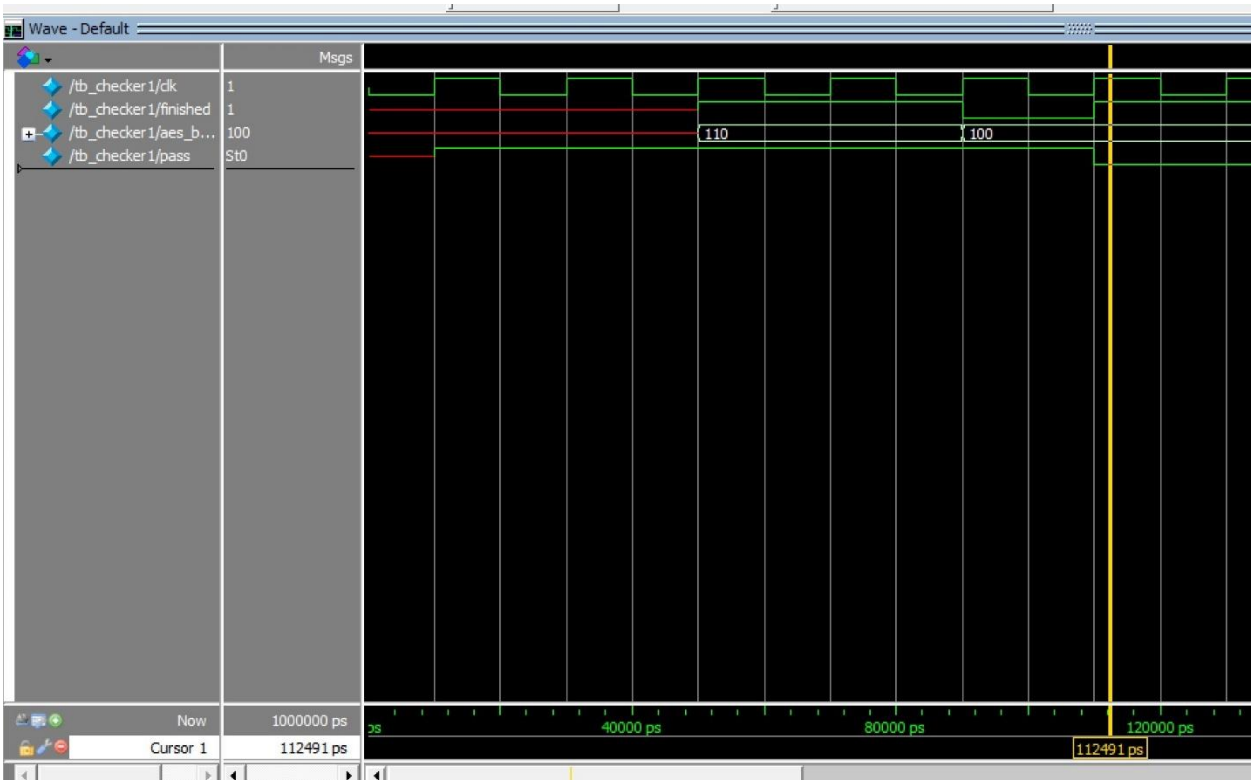

Figure 5: HDL functionality check for `property 1' of test suit

```
property 1;
@(posedge clk) (finished==1'b1) |-> aes_binary.FSM== 3'b110;
endproperty
```

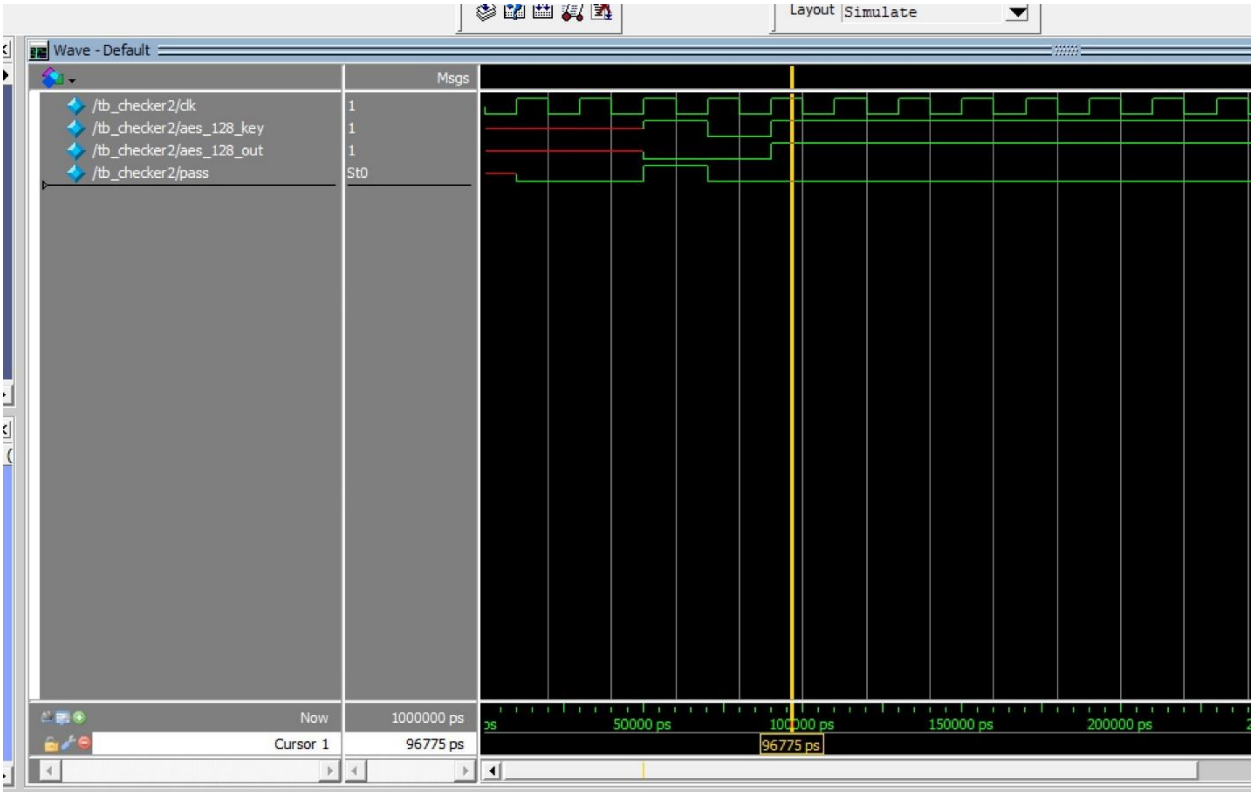

Figure 6: HDL functionality check for `property 2' of test suit

property 2;
@(posedge clk) aes_128.key!= aes_128.out;
endproperty

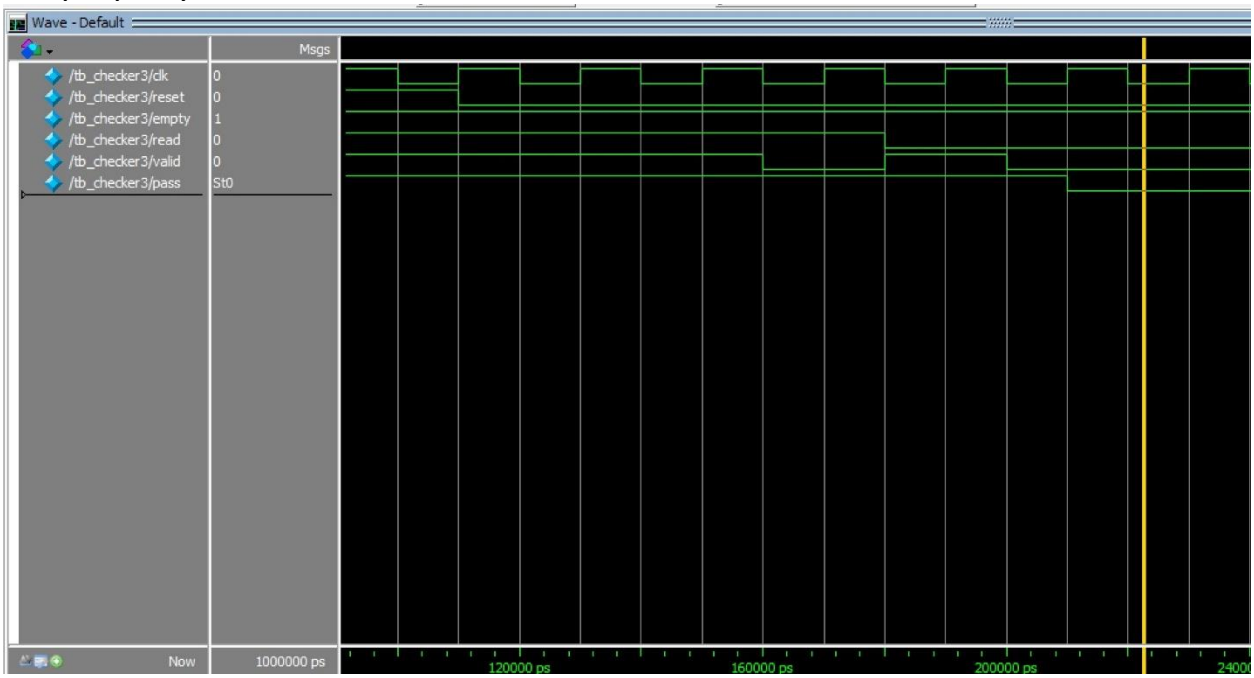

Figure 7: HDL functionality check for `property 3' of test suit

```
property 3;
@(posedge clk) disable iff(reset) (empty==1'b1 && read==1'b0) |-> (valid==1'b1 );
endproperty
```

Figure 8: HDL functionality check for `property 7' of test suit

```
property 7;
  @(posedge clk)  disable iff (reset) (user_input == 3'b111) || (user_input ==
3'b110) |=> (fsm_1.state == 2'h01);
endproperty
```

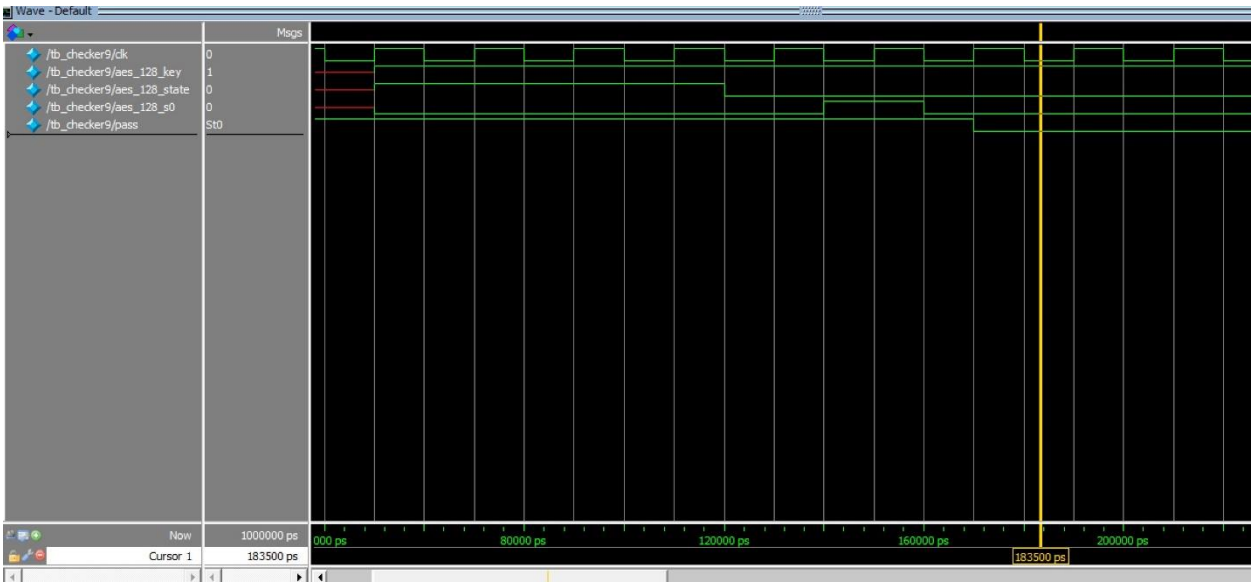

Figure 9: HDL functionality check for `property 9' of test suit

property 9;
@(posedge clk) ((aes_128.key) ^ (aes_128.state)) |=> aes_128.s0;
endproperty

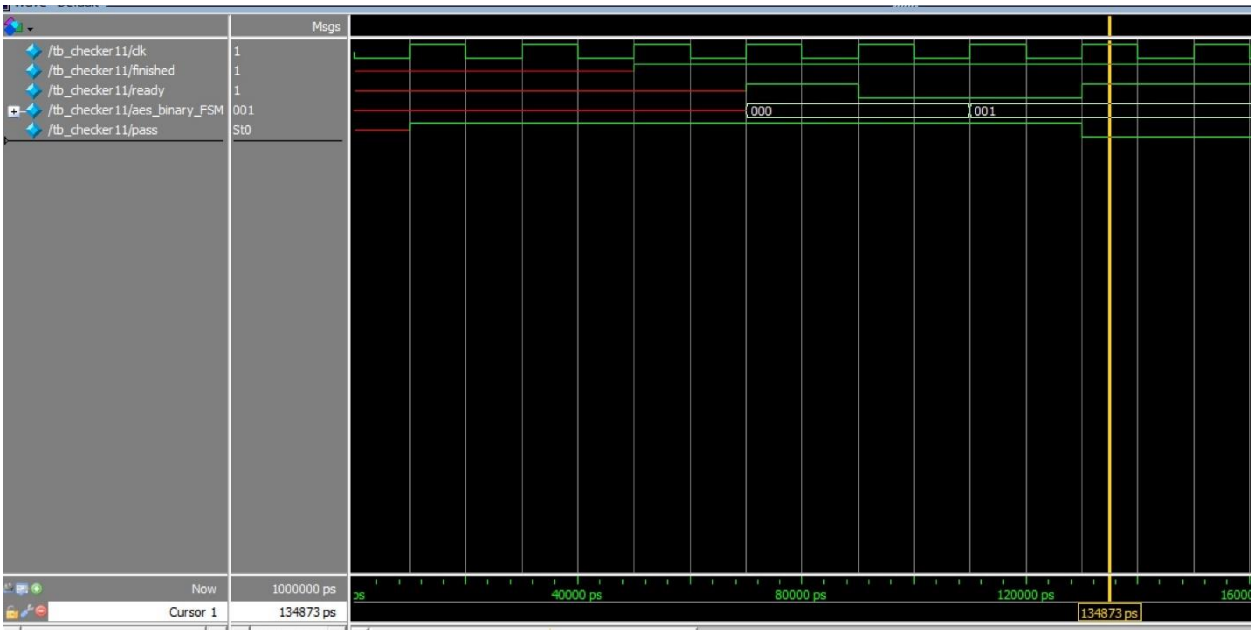

Figure 10: HDL functionality check for `property 11' of test suit

property 11;
@(posedge clk) (finished==1'b1) ##1 (ready==1'b1) |-> aes_binary.FSM== 3'b000;

endproperty

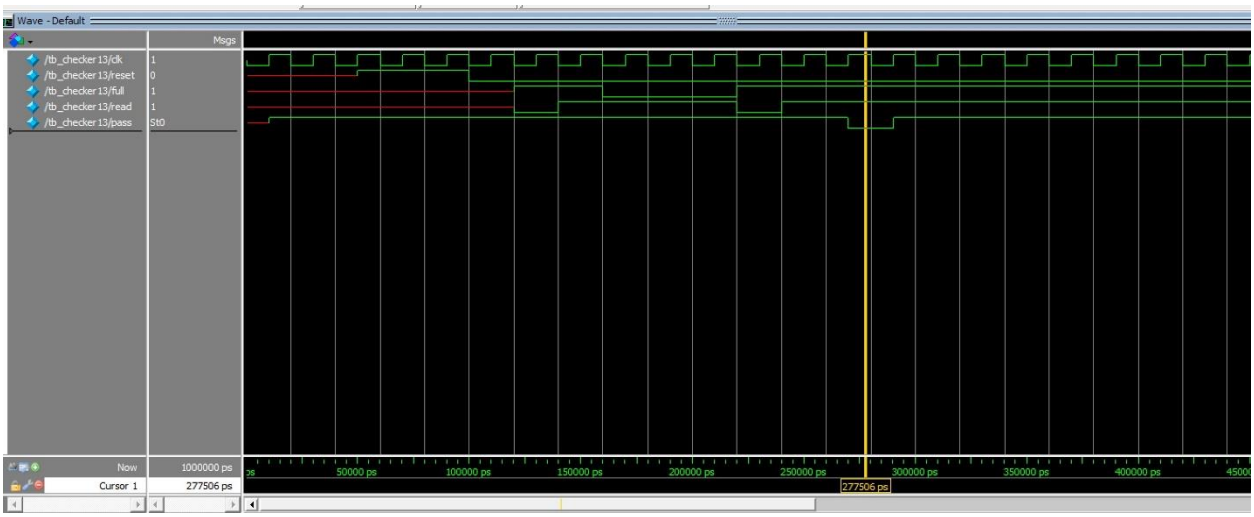

Figure 11: HDL functionality check for `property 12' of test suit

property 12;
@(posedge clk) disable iff(reset) (full==1'b1 && read==1'b0) ##1 (read == 1'b1)
|=> (full==1'b0);

endproperty

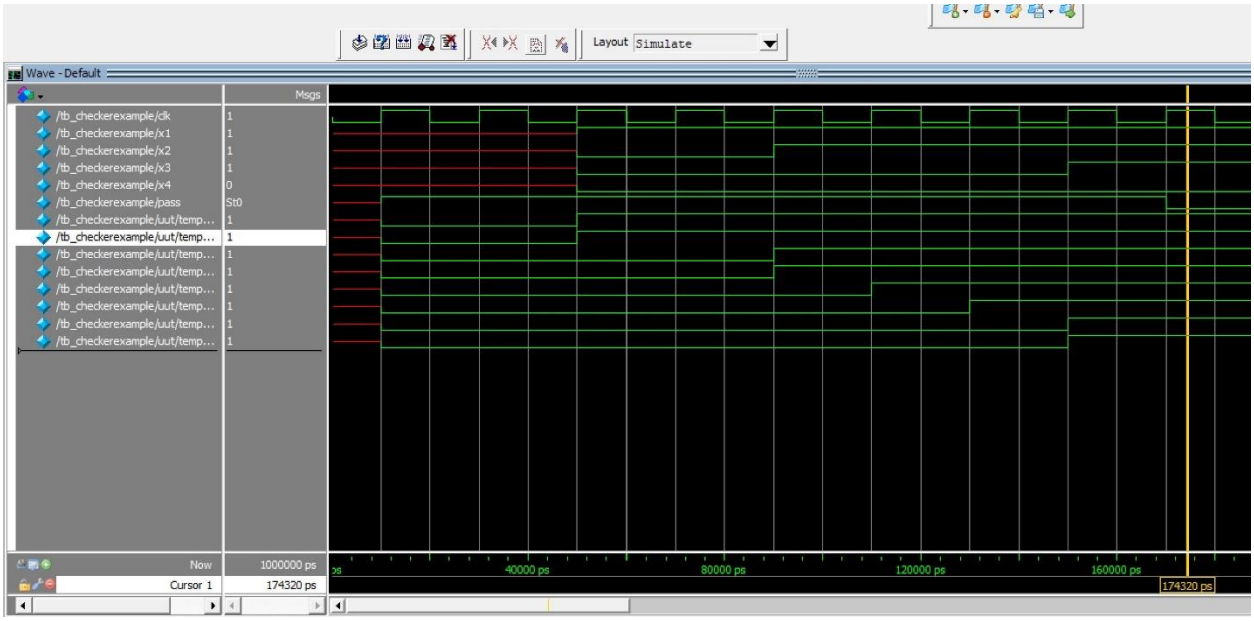

Figure 12: HDL functionality check for corner case

Future Works

The current version of the tool can operate on the basic assertion operations. Still, it has some advantages. The pros and cons of the current version of the tool are mentioned below:

Pros:

- ❑ The tool can perform operations on multiple properties together
- ❑ The tool can perform cases where there are multiple clock cycle delays presents in both antecedent and consequent statements
- ❑ The tool does not use any external library for parsing or generating RTL. Such easier installation makes the tool user-friendly and quite quick.
- ❑ The tool can handle irregular indentation and spacing
- ❑ The tool can intake assertions written in any file format

Cons:

- ❑ The tool cannot perform on system built-in functions
- ❑ The tool does not include some of the temporal operators (*, $)

The above-mentioned limitations can actually easily be mitigated and advanced functionality can be included in the tool in the future version. In the future version of the tool, the following functionality will be included.

1. ##[1:N] functionality by tweaking the flags to jump to consequents operation if the branch condition becomes trye in any of the cycle stages
2. Combinational Property check in the parser. In this scenario always@(*) will be used.
3. Adopting built in functions such as $countones(), $rose() etc by modifying the code structure.

References

[1] Sayantan Das, RiziMohanty, PallabDasgupta, P.P. Chakrabarti, “Synthesis of System Verilog Assertions”, Design, Automation and Test in Europe, München, Germany, 2006

[2] Ivan Kastelan, ZoranKrajacevic, “Synthesizable System Verilog Assertions as a Methodology for SoC Verification”, First IEEE Eastern European Conference on the Engineering of Computer Based Systems, 2009.

[3] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00), pages 538–542, 2000.
[4] Boulé, Marc, and Zeljko Zilic. Generating hardware assertion checkers. Berlin: Springer, 2008.
[5] Amin, O., Ramzy, Y., Ibrahem, O., Fouad, A., Mohamed, K., & Abdelsalam, M. (2016, December). System Verilog assertions synthesis based compiler. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)* (pp. 65-70). IEEE.