Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 11 Issue 07, July-2022

Design Verification Flow and System Verilog based Assertions for Verification

Sushan A Patel
Student,
Department of ECE,
RV College of Engineering,
Bangalore,

Sujatha Hiremath
Assistant Professor,
Department of ECE,
RV College of Engineering,
Bangalore.

Abstract:– **Design Verification is a very vast process and involves a lot of steps in order to ensure the RTL design is bug-free. Majority of verification flow is divided into 3 steps i.e. Generating Stimulus, Writing assertions to verify the design, and generating cover points to ensure the hit scenarios of the design. Stimulus and assertions for the same play a significant role in order to hit a high coverage ratio. Using Assertions along with the RTL design helps to ensure that the functionality implemented in the design is in coherence with the stimulus provided with the same. This paper describes various assertion scenarios and how to implement assertions to verify different functionalities of the design. This paper discusses 5 different varieties of assertions that can be implemented according to the requirement.**

1. INTRODUCTION

The role of any design verification person is basically divided into 3 steps [4]. (As shown in Figure 1.1):

1. To generate a stimulus for the given RTL design
2. Write assertions to validate the stimulus
3. Generate cover points to check if the scenario is hit or not.

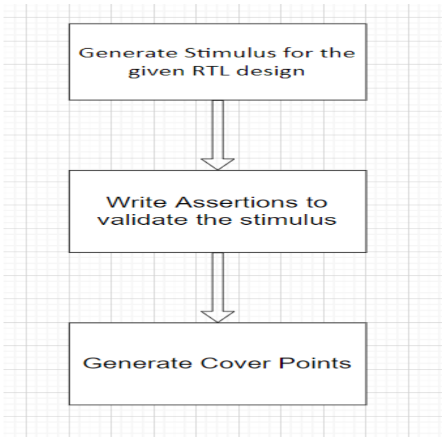
Generate Stimulus for the given RTL design
Write Assertions to validate the stimulus
Generate Cover Points

Figure1.1 Design Verification Flow

Stimulus plays a very important role in the design verification as it ensures how to play with the input of the RTL design. Stimulus helps to generate better coverage based on how robust and randomized the stimulus is. Verification done using directed test cases is not a vital or correct method of verifying a design. There should be enough randomization in the stimulus that is provided to the design so that the design can withstand with the logic even with the scenarios that were not included earlier. It can so happen that the design doesn't support that feature that the stimulus has generated. This is also a good way to understand the limitation of the design and what all features the RTL design supports. This can turn otherwise as well so it is very important to randomize the stimulus as per the requirement and not blindly randomize the complete flow. [5]

Assertions come into the picture after the stimulus is generated. It takes the generated stimulus and verifies whether the given logic is correct or not. So for Example, if input signals A and B goes through an AND gate to give output as signal C, then with the stimulus of both A and B as "1" and "0" respectively, the assertion will check whether the value of signal C is "0" or "1". So if the value is "1", the assertion fails and returns as an error in the RTL design. There is a lot of variety in assertions and they mainly fall into 2 different categories i.e. Immediate Assertions and Concurrent Assertions.

Immediate Assertions:

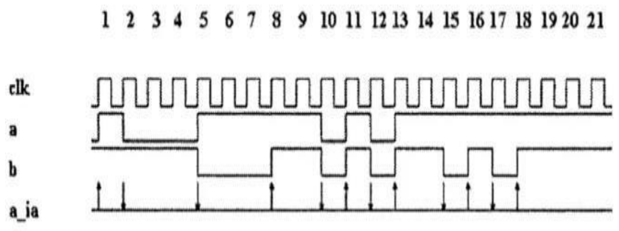

Figure 2.1 Flow of Immediate Assertion

Concurrent Assertions:

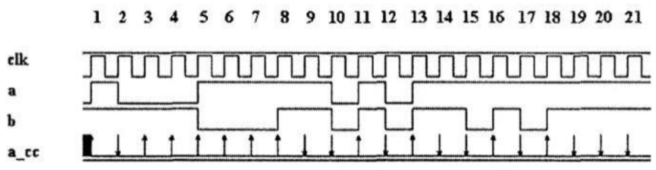

Figure 2.2 Flow of Concurrent Assertion

The third and most important part of any Verification is to check coverage. It is impossible to hit 100% coverage for any design because there will always be scenarios that will never be hit but the design will never support that. Coverage basically means that all features the stimulus generates are randomized i.e. it helps to check which all cases of the design are being hit. The reason for the coverage being most important is that it can so happen that some portion of the design is never checked at all . There

will be noassertions failing in that because the stimulus has never generated anything for that particular case. So that could be a big disaster if something like that happens. [3]
The only way to ensure the Verification result of the RTL design is to have assertions for all the logic of the design and to have enough coverage around thesame.

1. RTL Design Specification:

Before Jumping into the different types of Assertions, it is must to have complete understanding of the RTL design and how it will behave in different situations. Here I am considering an example of a simple RTL design that has 4 blocks named as A, B, C, D. [1]. As shown in Figure 3.1

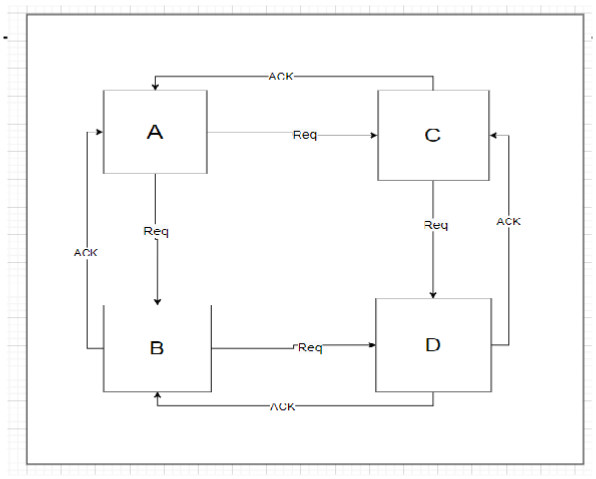

Figure 3.1 Request Ack handshake Concept

1. Block A interacts with block B, and C that works like a 4 way handshake mechanism of request and Acknowledge.
2. Block B provides Acknowledgement back to block A Acknowledgement provided to block A should be immediately as soon as the request comes from blockA.
3. Block C provides Acknowledgement back to block A (after 5us). Acknowledgement provided to block A shouldn't come before 5us of time period.
4. Block D interacts with block B, and C that works like a 4 way handshake mechanism of request and Acknowledge.
5. Block D provides Acknowledgement back to block B. Acknowledgement provided to block B should be within 10us of time duration and it shouldn't cross this time.
6. Block D provides Acknowledgement back to block C. Acknowledgement provided to block C should be immediately as soon as the request comes from blockC.

3. ASSERTION IMPLEMENTATION:

Based on this RTL design functionality described, this paper will discuss about the major 5 assertion categories. Note that there can be a lot of other assertion category that can be implemented but this paper focuses only on these 5 categories.[2].

Assertion Category:

1. Assertions to check whether the given signal doesn't go unknown in the given entire duration (signal = X)
2. Assertions to check whether the given test_expr is 0 or 1 (concept of Always and Never)
3. Assertions to check the one-hot condition of the signal (Remaining completely 1 or 0 during the entire duration)
4. Concept of checking whether the requested signal is provided with the Acknowledgement signal in a given duration of time
5. Concept of checking whether the requested signal should not be provided with the Acknowledgement signal until a given duration of time

Category 1:

This Assertion type checks for any of the signal in the RTL design should never be X. As seen in the image below the Ack signal is remaining X for the complete duration, even when the Request is being received.
So this should never happen. The assertion for this logic is very simple and it should fail every time the signal is Driving X and not toggling at all.

Ideal Behavior:

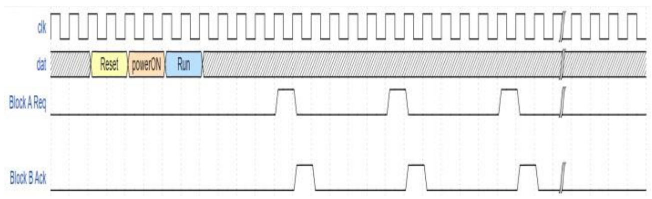

Figure 4.1 Ideal behavior of category 1

Wrong Behavior:

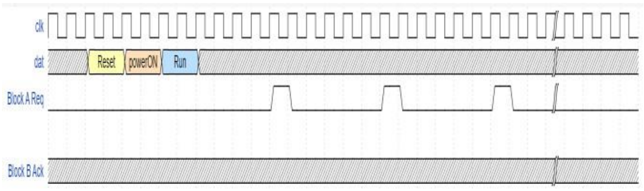

Figure 4.2 Wrong Behavior of Category 1

Category 2: (This is the case where Ack did not come immediately once the block receives the Req)

This Assertion case is also very simple wherein once the block A sends Req to another block B for a particular operation. At this time block B should send Ack back to block A immediately in the next cock cycle otherwise it's a design bug. So the assertion here will check for the Ack signal as soon as the Req gets deasserted and wait for the Ack for 1 clock pulse otherwise return the status of assertion as failure.

Ideal Behavior:

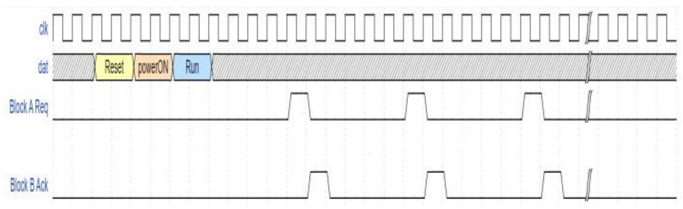

Figure 4.3 Ideal behavior of category 2

Wrong Behavior:

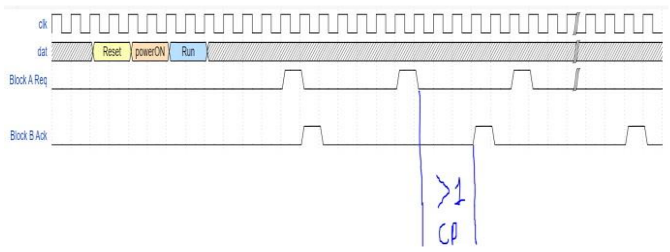

Figure 4.4 Wrong Behavior of Category 2

Category 3:

This Assertion will check whether a particular signal is not remaining completely “1” or “0” for the entire duration of the test case. This is very similar to category 1 but here the issue is that the design will go into hang state as the complete logic would be driving only 1 value i.e. 0 or 1 for the entire duration. This case is not as easy to identify like the case of Category 1. It might not even fail in certain situations as the logic would satisfy the conditions sometimes.

Ideal Behavior:

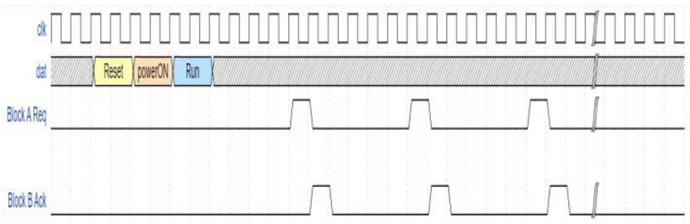

Figure 4.5 Ideal behavior of category 3

Wrong Behavior:

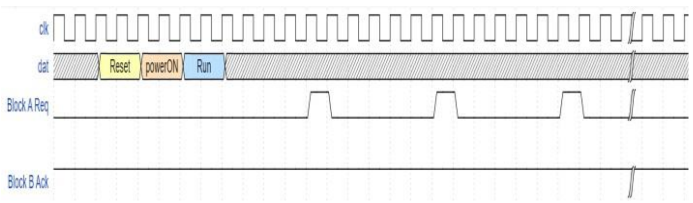

Figure 4.6 Wrong Behavior of Category 3

Category 4:

This case is a complex case wherein the combination of a counter and an assertion of category 3 is required. In this case we have to check that the block D provides Ack back to block B within 10us. In order to implement this, a counter is required. This counter will count for the duration of clock cycles. Now the main point is we are required to check for the Ack before the counter expires. So here we will have to use the category 3 wherein the output signal to be polled will be the counter_expired one. So the assertion should be enabled when block B send a Reqto block D and check for only when the counter_exp = 0 and within this duration the Ack should come.

Below is the implementation of the followingassertion.

COUNTER_MACRO(.counter_exp(), .counting(), .rst(rst) , .en() , .counter_value())

Lets assume the Frequency of clk = 100MHz

Next step is to convert the 10us into the no of clock cycles based on the frequency of clk

Counter_value = 1000 clock cycles

BLOCK_D_CNTR (.counter_expired(), .counting(), .reset() , .en(Req) , .counter_value(1000))

So here the counter will start down counting from 1000 to 0 once the block B sends Req to Block D.

Assertion :

ASSERTION_CHECK_0 (.MSG (Ack didn’t come within 10us of Req))
BLOCK_D_ACK_ASSERT(.clk(clk), .rst(), .en(**Req**), .(test_expr(**Ack** & **BLOCK_D_CNTR.counter_expired**)))

The assertion macro ASSERTION_CHECK_0 will show error whenever the value of test_expr becomes 1 otherwise it works fine.This following BLOCK_D_ACK_ASSERT will print the msg as defined if the assertion fails. Now here the assertion will enable when the Req from Block C is received. So if the Ack comes within this time frame and the value of counter_expired=0 then the assertion is valid, if it comes after the counter_expired=1, the assertion willfail and show error.

Ideal Behavior:

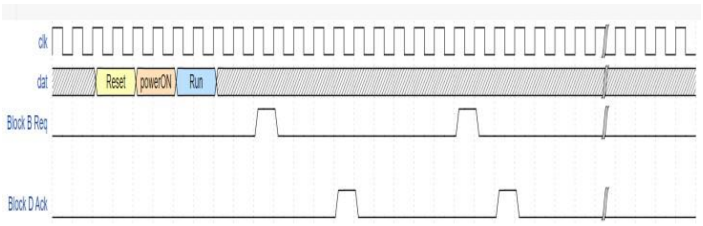

Figure 4.7 Ideal behavior of category 4

Wrong Behavior :

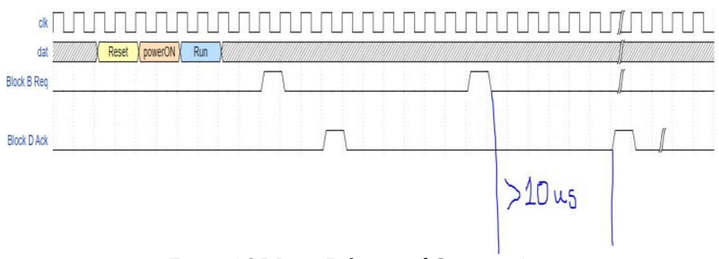

Figure 4.8 Wrong Behavior of Category 4

Category 5:

This case is also a complex case wherein the combination of a counter and an assertion of category 3 is required. In this case we have to check that the block C provides Ack back to block A within 5us. In order to implement this also, a counter is required. This counter will count for the duration of clock cycles. Here also we are required to check for the Ack but not before the counter expires. Here we need to wait for 5us and then only provide the Ack signal to block A. So here we will have to use the category 3 wherein the output signal to be polled will be the counter_expired one. So the assertion should be enabled when the Req comes and wait for the counter_exp = 1 and then only the Ack should come. Below is the implementation of the following assertion.

COUNTER_MACRO(.counter_exp(), .counting(), .rst(rst) , .en() , .counter_value())

Lets assume the Frequency of clk = 100MHz

Next step is to convert the 5us into the no of clock cycles based on the frequency of clk

Counter_value = 500 clock cycles

BLOCK_C_CNTR (.counter_expired(), .counting(), .reset() , .en(Req) , .counter_value(500))

So here the counter will start down counting from 500 to 0 once the block A sends Req to Block C.
Assertion :

ASSERTION_CHECK_0 (.MSG (Ack didn't come within 10us of Req))
BLOCK_C_ACK_ASSERT(.clk(clk), .rst(), .en(**Req**), .(test_expr(**Ack & ~BLOCK_C_CNTR.counter_expired**)))

The assertion macro ASSERTION_CHECK_0 will show error whenever the value of test_expr becomes 1 otherwise it works fine.This following BLOCK_C_ACK_ASSERT will print the msg as defined if the assertion fails. Now here the assertion will enable when the Req from Block A is received. So if the Ack comes after the 5us time frame and the value of counter_expired=1 then the assertion is valid, if it comes before the counter_expired=1, the assertion will fail and show error.

Ideal Behavior:

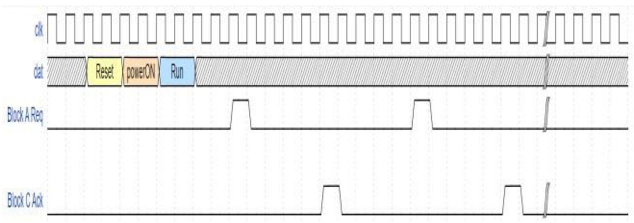

Figure 4.9 Ideal behavior of category 5

Wrong Behavior:

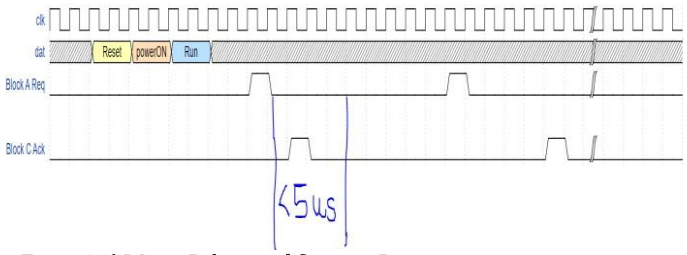

Figure 4.10 Wrong Behavior of Category 5

4. RESULTS:

This paper discussed how useful assertions are in thedesign verification environment. They help to make the design Bug free and more robust to withstand in the market. Deploying Assertions has several advantages which can be summarized below:

- Debug time can be reduced drastically.
- Bugs can be found along with the timestamp andeven during which particular scenario.
- Verbosity can be controlled along with writing assertions.
- Assertions go handy with C++ functions.
- Deep understanding of the Specification and Functionality of the design can be developed.
- Assertions are used for writing Coverage as well.

As I mentioned before as well, assertion can be written based on the requirement of the verifier and the design specification. An individual can modify the assertion and play with it according to the need. [6]

5. REFERENCES:

[1] P. Ghosh, "*Analysis and verification of SoC design RTL parameters*", IEEE 1st International Conference on Power Electronics, Intelligent Control and Energy Systems (ICPEICES), pp. 1-6, July 2016.
[2] P. Gurha and R. R. Khandelwal, "*SystemVerilog Assertion Based Verification of AMBA-AHB*", International Conference on Micro-Electronics and Telecommunication Engineering (ICMETE), pp. 641-645, September 2016.
[3] C. Cheng, C. C. Yen and J. Y. Jou, "*A formal method to improve SystemVerilog functional coverage*", IEEE International High Level Design Validation and TestWorkshop (HLDVT), pp. 56-63, Nov. 2019.
[4] Xiao Bin Chu, L. U. Tie-Jun, Y. Zong, "*Combination ofAssertion and Coverage-Driven Verification Methodology*", Microelectronics & Computer, ISSN: 2080-8755, Vol. 25, No. 9, 2018.
[5] V. Jusas, T. Neverdauskas, "*Stimuli generator for testing processes in VHDL*", NORCHIP, pp.1-4, 2017.
[6] Y. N. Yun, J. B. Kim, N. D. Kim and B. Min, "*Beyond UVM for practical SoC verification*", International SoC Design Conference, pp. 158-162, November 2018.