LAB SE517: Functional Verification of an Arbiter Design [136 pts]

Acknowledgements: This work benefit from the french government IRT Nanoelec program, reference ANR-10-AIRT-05

Introduction

This LAB aims at familiarizing students to perform the functional verification of digital designs. For this LAB subject (3 sessions), you will have to prepare only one final report (per team of 2 students) about the functional verification of an arbiter design and to do the following tasks:

- Writing a test plan for an arbiter design,
- Writing a VHDL RTL description of an arbiter,
- Writing several VHDL testbenches: dynamically generating stimuli based on the Design Under Verification (DUV) outputs, generating random stimuli…
- Writing assertion based verification using OVL and PSL.

At the end of the LAB you will create an archive (named LAB_PSL_SE515_20XY_20XY+1_NAME1_NAME2.zip) containing all your files and your final pdf report. Then, you will upload it into Chamilo (SE515 course, Verification and Test part, LAB_PSL_20XY_20XY+1).

As a reminder, the *zip command* allows to compress and create a zip archive.

Preliminary

This LAB will be done on a Debian Operating System and with the use of the QuestaSim/ModelSim tool.

All the files and other useful documents are available in Chamilo (SE517 course, Verification and Test part, TP).

At first, create the LAB_PSL directory in your home/Dev directory where you will copy all the files to be completed.

Then, initialize the environment variables with:

source PATH={PATH}:/opt/Questa/questasim/bin

Finally, launch ModelSim with the command:

vsim &

In ModelSim, create a new project: menu File/New/Project, take LAB_PSL as project name and set LAB_PSL as Project Location.

This operation creates by default a work library corresponding to a work directory in LAB_PSL in which all VHDL design units will be compiled by default.

Add the uncompleted VHDL source files: (arb, arb_tb1 arb_tb2, driver, property_checker, protocol_checker) in your project in order to compile them.

File/Add To Project/Existing File …

Notes:

- If the file is in write-protected form, you must use the command: *chmod +w file-name*
- When there are several entities and architectures, they must be compiled in hierarchical order (subcomponents, then components, then testbench).To simulate, it is necessary to indicate which entity is chosen: *Simulate/Start Simulation…*

Choose the design to simulate (for example arb_tb.vhd) in the design tab.

Choose the work library in the libraries tab.

The sim tab appears in the workspace window.

In the objects window, select the signals, which you want to see in the chronograms, then right click and select *Add to Wave/Selected Signals*. The chronogram appears with the selected signals available on the left.

Note: In the sim tab, you can select the entity including the signals you want to select in the objects window. In our case, simply select the description arb_tb.

Increase the default time increment (from 100 fs to 100 ns).

Select the button Run (click several times if necessary).

1) ARB description

ARB is a simple design that arbitrates the access to an unspecified resource between three requestors. The requestors apply for resource access by raising their assigned input bit of vector *req[0:2]*. Once ARB sees the input *cmd* asserted, it must grant the resource to one of the requestors by raising their corresponding bit on ARB's output vector *gnt[0:2]* after a fixed amount of cycles. Figures 1a–d lay out the details of the specification for ARB.

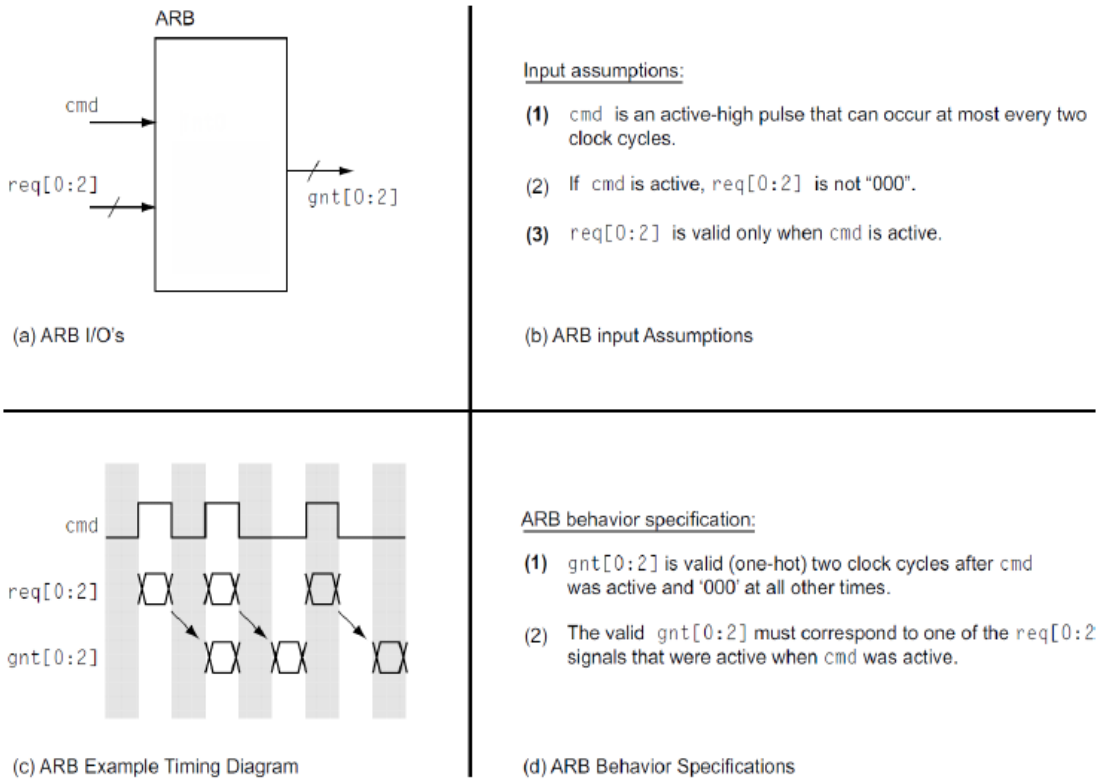


Fig.1 specification of the ARB. (a) The input/output pin specification. (b) The assumptions about the environment of ARB on its inputs. (c) The behavior of ARB with an exemplary timing diagram. (d) The specification ARB is expected behavior both on the outputs.

The ARB must allocate the resource to only one of the three requestors (among those who requested it) by setting '1' to the vector gnt [0: 2] corresponding to the requestors. For example, if only one requestor applies for resource access, the ARB allocates the resource to the according requestor. In other hand, if many requestors (two or three) apply for resource access simultaneously, the ARB detects a conflict case. In this conflict case, **the ARB associates each requestor (R1, R2 and R3) with a resource access number (N1, N2 and N3) respectively**. Then it selects one requestor Ri, which has the minimum resource access number N. **The resource access number Ni according to the selected requestor Ri must be incremented and the resource access numbers according to the unselected requestor must be decremented.** The figure 2 shows a chronogram of the ARB simulation.

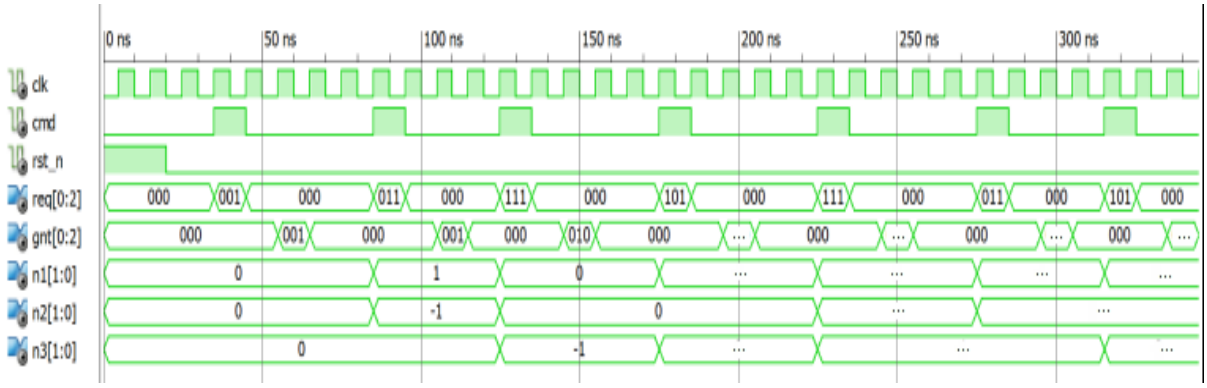


Fig.2 Chronogram of the ARB example DUV

1. Describe the inputs / outputs of the ARB and manually complete the signals gnt, n1, n2 and n3 after 200 ns [6 pts].

In the following, we will propose a low cost arbiter design minimizing as much as possible the flip-flop number and the number of bits used to code the counters.

2. Write the RTL code of the ARB design (arb_tobe_completed.vhd) considering its specification and the chronogram example showed before. Calculate the number of Flip-Flop of your design [12 pts].
3. Propose a simple verification plan, which considers the most important scenarios to verify the functionalities of the arbiter [6 pts].
4. Complete a simple testbench file (arb_tb1_tobe_completed.vhdl) to verify the functionality of your arbiter design according to your previous simple test plan [6 pts].
5. Simulate your testbench and check your arbiter design [6 pts].

Note: If you have not succeeded to develop an arbiter design that achieves the described properties, in the following you can use the simple arbiter "simple_arb.vhd" to complete the LAB.

2) Verification environment description

The figure 3 shows the translation of the ARB input assumptions and behavior specification into the topology of a Design Under Verification (DUV) connected to a protocol checker and a property checker. The purpose of the protocol checker is simply to watch over the environment of the DUV and to raise an error if the actual signal interaction violates the input assumptions of Figure 1. The main focus is on the property checker with its various inputs and the three *fails()* output signals. Every *fails()* bit represents one of the three properties that we check. If verification finds any legal way to stimulate ARB such that one of the *fails()* bits turn active, a property violation and therefore a design bug is identified.

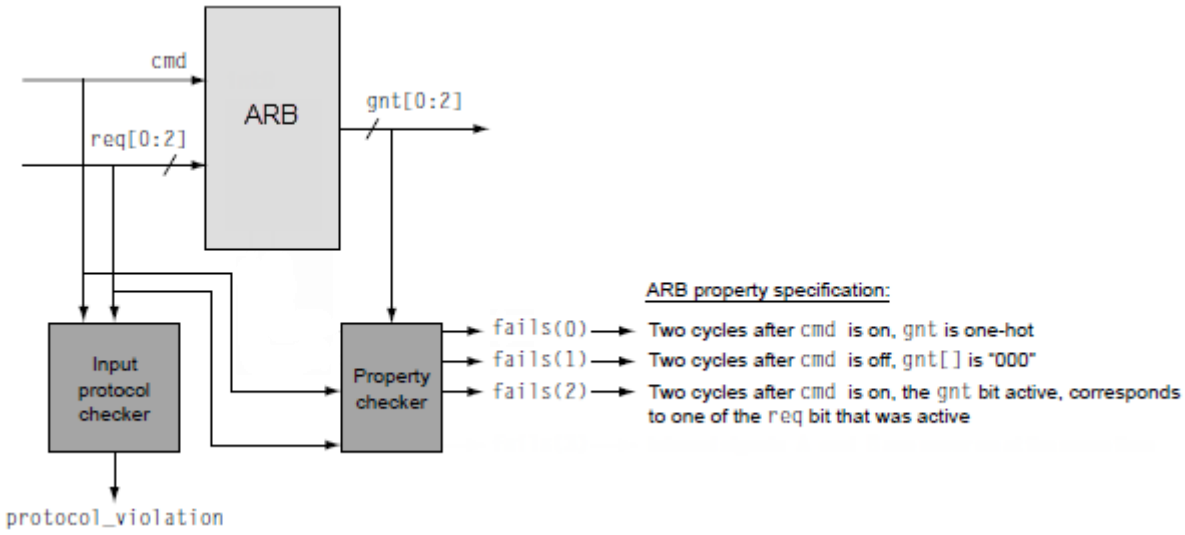


Fig.3 ARB Verification model

This model connects an input protocol and a property checker block to the DUV (ARB). The outputs of the property checker represent the three properties mentioned in figure 3 and connected next to their corresponding *fails()* bits. The *fails()* bit turns on if the DUV violates the specification.

a) Property checker description

The property checker block verifies the property specification of the ARB. It can detect three kinds of fails during the functionality of the ARB.

- *Fails(0)*=1, when the cmd was ON before two clock cycles and the actual gnt is not one-hot.
- *Fails(1)*=1, when the cmd was OFF before two clock cycles and the actual gnt is not "000".
- *Fails(2)*=1, when the cmd was ON before two clock cycles and the actual gnt multiplied to the saved req is "000".

b) Protocol checker description

The protocol checker block generates a protocol_violation signal if the actual signal interaction (cmd or req) violates the input assumptions.

- protocol_violation = 1, when cmd appears in two successive clock cycles.
- protocol_violation = 1, when cmd is one and req is "000".

6. Complete the protocol checker VHDL code using the protocol_checker_ tobe_completed.vhd [8 pts].

7. What does it mean if a fail signal is activated while the protocol_violation signal remains inactive? [4 pts]. What is the function of the property checker block? [2 pts]
8. Complete the property checker VHDL code using the property_checker.vhd file. [6 pts]
9. Complete the second testbench (arb_tb2_tobe_completed.vhd), then simulate this testbench and correct the potential errors. If there is no error, manually inject errors into the arbiter to see if your verification model detects them. Show your simulation to the teacher for validation. [6 pts]

3) DUV driver using dynamic and random test pattern generation

In Figure 3, we protect the functional verification (FV) environment from false input behavior with a protocol checker. This is only a passive function. In addition, just like in simulation, the FV effort needs the specification of an active driver that provides supplies all possible input stimuli. The property checker traces inputs, internal states, and stimulus over time and asserts every cycle via the fails() signals, whether the DUV complies with the properties of the functional specification.

Figure 4 shows how the complete functional FV environment creates a closed model with a driver on the DUV inputs and the property checker connected to the outputs. The whole purpose of the FV effort is to prove that the fails () signals are always “0.” If this exhaustive proof via FV is successful, we have 100% confidence that the DUV is without bugs for the properties specified. This is a verification result with the highest confidence possible.

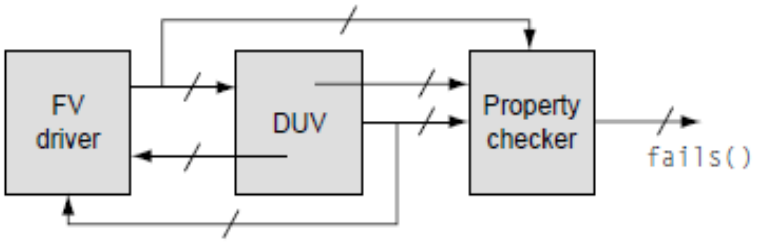


Fig.4 Model for functional formal verification (FV) environment of the DUV

A nondeterministic signal assumes all values in FV, whereas, if used in a simulation model, a simulation driver would chose a random value from the legal value set. Non-determinism is a powerful means to specify FV drivers.

Figure 5 shows how to use non-determinism to specify all possible inputs to the ARB DUV. The flowchart specification makes the two random choices of the driver. First, a random number 1 to n is chosen as the cycle delay before generating a new cmd signal value of 1. Second, there is a random selection of eight choices for all the possible bit-pattern for the req[] vector.

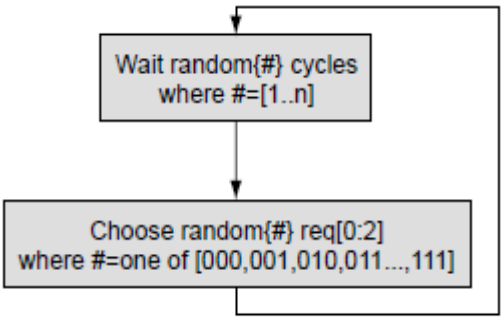


Fig.5 Driver flowchart

10. Complete the VHDL code of the Driver entity given in driver_tobe_completed.vhd.
    Note: To generate a random number in VHDL, use the UNIFORM procedure; UNIFORM (seed 1, seed 2, rand). [12 pts]
11. Add in the driver component a VHDL process that will display on txt file "Display.txt" during simulation the values of the vector req whenever cmd goes to '1'. Then add the driver component to the testbench (arb_tb2.vhd), simulate the arbiter design and observe the evolution of the signals N1, N2, and N3 by writing them in the same file "Display.txt"; thus, it will be possible to determine empirically the max and min values of the registers Ni. [12 pts]
    Tips: For this purpose, use a process that integrates the write, and writeline functions.

4) Verification using a fuzzer (genetic algorithm)

A fuzzer (or genetic algorithm) is an algorithm for detecting errors in a system. It generates random sequences and modifies them according to the results obtained in order to get as close as possible to a potential error. For example, we can check if n1, n2 and n3 do not exceed a certain very low value.

The fuzzer generates valid inputs randomly and keeps the best inputs (inputs with the lowest scores: minimum of n1, n2 and n3 at the end of the simulation). With those inputs, the Genetic Algorithm generates new inputs and compares the new ones with the old ones. It keeps then the 10 bests inputs and do it again in order to have the lowest scores:

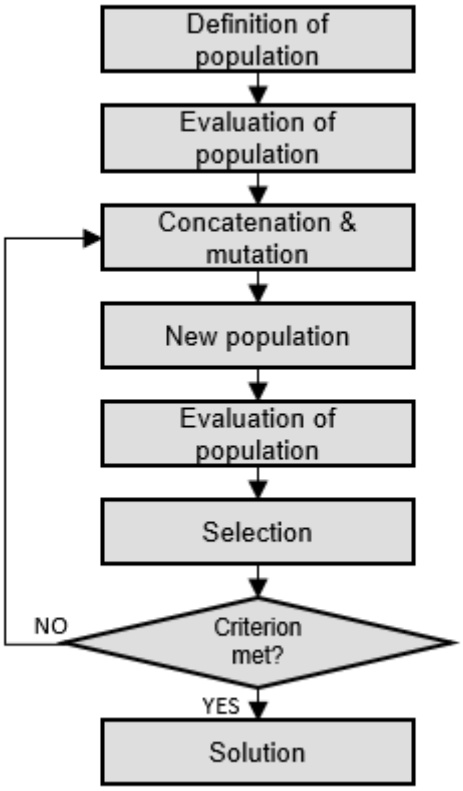


Fig.6 Genetic Algorithm

Before using the genetic algorithm. You will need to know how the specific testbench related to the fuzzer works. As illustrated in Fig. 7, it reads a number of sequences from a file (sequences.txt) and write the results on another file (results.txt).

12. **Create the file sequences.txt with valid sequences [4]**.

    The sequences have to be a list of 4 bits (cmd&req). You can also put multiple sequences and separate them with "Done" like this:

    1111
    0000
    1100
    0000
    0000
    Done
    1100
    0000
    1011na
    Done
    (without a new line here)

    Simulate the arb_tb_fuzzer and see the results [4].
    You can use "run all" in order to run until the end of the file.
    Check if the results obtained in results.txt are the same as in the chronograms.

To use the given Python script genetic_algorithm.py, you have to modify this file and modelsim_command.do:

- {lib} must be modified in **genetic_algorithm.py** (line 11&12) and **modelsim_command.do** (line 1) by the library where arb and testbench are compiled (example: work).
- {dir} must be modified in **genetic_algorithm.py** (line 11&12) by the directory where the arb.vhd and arb_tb_fuzzer.vhd are located (example: /home/user/work).

This script will run a genetic algorithm and simulate arb_tb_fuzzer.vhd with modelsim_command.do.

The second file modelsim_command.do contains all the instructions needed to simulate, run and stop the simulation. This is helpful to work on batch mode without GUI**.**

The following figure shows how the Python script genetic_algorithm.py is connected to the testbench:

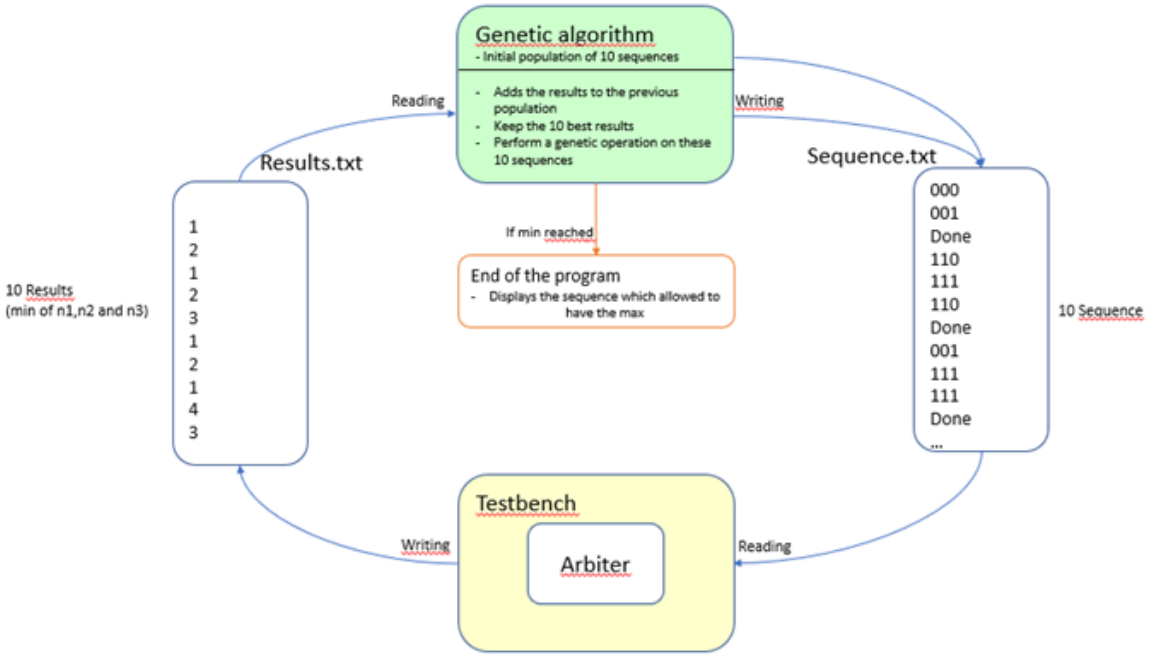


Fig 7: How python script works

13. Execute the script with the command: **python3 Genetic_algorithm.py [4]**

If a sequence is displayed, that means that the value wanted has been reached by this sequence.

In order to verify if the generated sequence is good, copy the sequence displayed and put it on the file sequences.txt.

14. Simulate the sequence displayed [4]
15. Modify the value MIN_WANTED in order to see if n1, n2 or n3 reach this value. Is the chosen value is reached? What does it mean? Is this a problem and why? [8]
16. Modify the arbiter to avoid the problem, and test again to see if the python script still finds a problem [8].

5) Optional: Security verification using a fuzzer (genetic algorithm)

The same tool can be used to perform security verification. The first step consists in defining security properties for the arbiter design. We can assume the following security property:

"When the input command is not valid, no requestor should obtain the resource"

17. Modify the previous input protocol checker component to verify this additional property [8]
18. Modify the python script to generate not valid input sequences and use this script to verify if the security property fails or holds [20].

6) Optional part: Using the OVL Library

The Open Verification Library (OVL) library is a library of HDL building blocks that a designer can embed as an instantiated component into an HDL specification and connect to a set of signals that provide the context for the assertion check. We assume a VHDL implementation where we separate the DUV from the property checker code.

The property (1) of the ARB is described in the form of an OVL assertion in arb.vhd. The main assertion to specify this property is the ovl_cycle_sequence assertion. You will find in annex an excerpt describing this OVL assertion.

library accellera_ovl_vhdl;
use accellera_ovl_vhdl.std_ovl_components.all;

To compile the OVL assertion, you must add the following command in the Transcript of the Modelsim.

vmap accellera_ovl_vhdl /tp/xes5eis/xes5eisvbe/TP_PSL_ESISAR/Annexe/ovl

18. Using the OVL assertion described in the annex, simulate your arbiter design [12 pts].

7) Using the PSL Library

Property Specification Language (PSL) belongs to the class of time domain specific languages. It is independent of any other language or HDL and its single purpose is property specification. There are two modes to use PSL. First, it can be used stand-alone, which means

that the verifier can group all PSL specifications into their own files. Obviously, the verifier has to have a way to link these files to the DUV. This is generally proposes by the simulator tools (and will be explained for QuestaSim in the following). The second mode to use PSL is the embedded mode. In embedded mode, designers write PSL properties and assertions directly into the HDL files of the DUV. In this LAB, we use the first mode.

19. Complete the PSL instructions in the file arb_tb_tobe_completed.psl corresponding to the 3 main properties. Inject faults to detect the errors.[24 pts]

Notes:

- The verification of these PSL properties must be performed using the ModelSim simulator (associated with Questa) installed on the CIME server.
- Use the command gedit arb_tb.psl to write the assertions in the psl file. In order to associate the file psl (arb_tb.psl) with the arb_tb2.vhd testbench (reuse the OVL testbench), it is necessary to add this file in the compilation options of arb_tb2.vhd. Right click on the testbench *file/Compile properties/select the PSL file*.
- It is possible to add the assertions directly in the chronogram (rather than observing the reports generated in the console) by going to *view/coverage/assertions* and then adding the assertions to the chronogram as you would for signals.
- The built-in function prev() takes an expression of any type as argument and returns a previous value of that expression. With a single argument, the built-in function prev() gives the value of the expression in the previous cycle, with respect to the clock of its context. If a second argument is specified and has the value i, the built-in function rev() gives the value of the expression in the i^{th} previous cycle, with respect to the clock of its context.

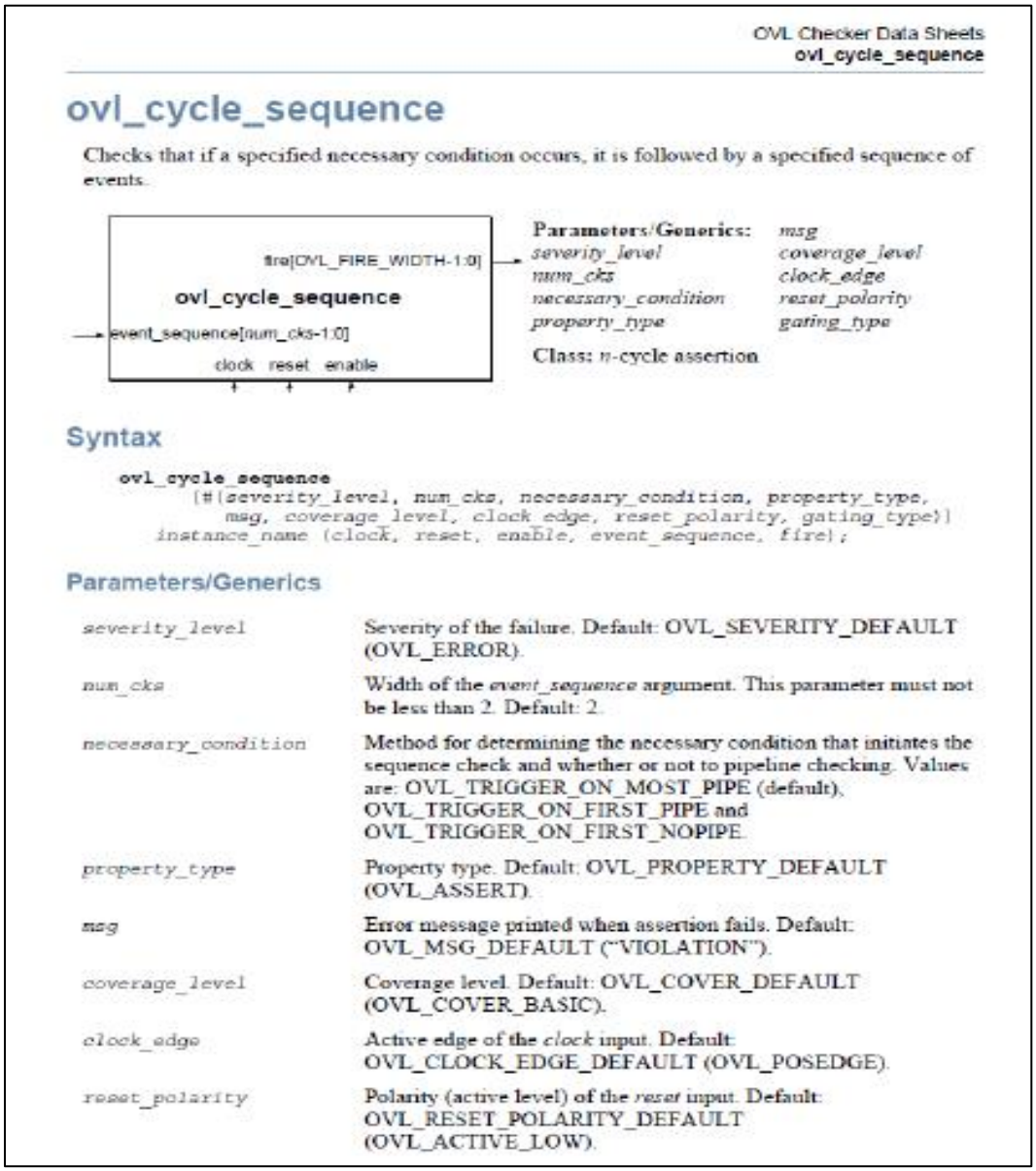
8) Fault injection in the arbiter design

Fault injection can be used to evaluate the robustness of circuits. You will apply Single Event Upset (SEU) fault injection in the arbiter design in order to propose a countermeasure to improve the design robustness.

20. Apply a simulation-based fault injection using the SEU fault model in the *gnt_temp register* at 50 ns. This fault attack aims to disrupt the specification of the arbiter affecting the one-hot specification of the gnt signal [6pts].
21. Propose a simple countermeasure to detect this fault injection. An error signal must be activated when the attack is detected. [6pts]

Annex

List of files associated with this TP: arb_tobe_completed.vhd, arb_tb1_tobe_completed.vhd, property_checker_tobe_completed.vhd, protocol_checker_tobe_completed.vhd, Simple_arb.vhd, driver_tobe_completed.vhd, arb_tb2_tobe_completed.vhd, arb_tb_tobe_completed.psl.



ovl_cycle_sequence

Checks that if a specified necessary condition occurs, it is followed by a specified sequence of events.

Parameters/Generics:
severity_level
num_cks
necessary_condition
property_type
msg
coverage_level
clock_edge
reset_polarity
gating_type

Class: *n*-cycle assertion

Syntax

```
ovl_cycle_sequence
      [#(severity_level, num_cks, necessary_condition, property_type,
         msg, coverage_level, clock_edge, reset_polarity, gating_type)]
   instance_name (clock, reset, enable, event_sequence, fire);
```

Parameters/Generics

severity_level	Severity of the failure. Default: OVL_SEVERITY_DEFAULT (OVL_ERROR).
num_cks	Width of the *event_sequence* argument. This parameter must not be less than 2. Default: 2.
necessary_condition	Method for determining the necessary condition that initiates the sequence check and whether or not to pipeline checking. Values are: OVL_TRIGGER_ON_MOST_PIPE (default), OVL_TRIGGER_ON_FIRST_PIPE and OVL_TRIGGER_ON_FIRST_NOPIPE.
property_type	Property type. Default: OVL_PROPERTY_DEFAULT (OVL_ASSERT).
msg	Error message printed when assertion fails. Default: OVL_MSG_DEFAULT ("VIOLATION").
coverage_level	Coverage level. Default: OVL_COVER_DEFAULT (OVL_COVER_BASIC).
clock_edge	Active edge of the *clock* input. Default: OVL_CLOCK_EDGE_DEFAULT (OVL_POSEDGE).
reset_polarity	Polarity (active level) of the *reset* input. Default: OVL_RESET_POLARITY_DEFAULT (OVL_ACTIVE_LOW).

OVL Checker Data Sheets
ovl_cycle_sequence

gating_type	Gating behavior of the checker when *enable* is FALSE. Default: OVL_GATING_TYPE_DEFAULT (OVL_GATE_CLOCK).

Ports

clock	Clock event for the assertion.
reset	Synchronous reset signal indicating completed initialization.
enable	Enable signal for *clock*, if *gating_type* = OVL_GATE_CLOCK (the default gating type) or *reset* (if *gating_type* = OVL_GATE_RESET). Ignored if *gating_type* is OVL_NONE.
event_sequence [*num_cks*-1:0]	Expression that is a concatenation where each bit represents an event.
fire [OVL_FIRE_WIDTH-1:0]	Fire output. Assertion failure when *fire*[0] is TRUE. X/Z check failure when *fire*[1] is TRUE. Cover event when *fire*[2] is TRUE.

Description

The ovl_cycle_sequence assertion checker checks the expression *event_sequence* at the active edge of *clock* to identify whether or not the bits in *event_sequence* assert sequentially on successive active edges of *clock*. For example, the following series of 4-bit values (where *b* is any bit value) is a valid sequence:

```
1bbb -> b1bb -> bb1b -> bbb1
```

This series corresponds to the following series of events on successive active edges of *clock*:

cycle 1	event_sequence[3] == 1
cycle 2	event_sequence[2] == 1
cycle 3	event_sequence[1] == 1
cycle 4	event_sequence[0] == 1

The checker also has the ability to pipeline its analysis. Here, one or more new sequences can be initiated and recognized while a sequence is in progress. For example, the following series of 4-bit values (where *b* is any bit value) constitutes two overlapping valid sequences:

```
1bbb -> b1bb -> 1b1b -> b1b1 -> bb1b -> bbb1
```