AEDVICES Consulting Regus Stop&Work Centr’Alp 196C Rue du Rocher de Lorzier

38346 Moirans Grenoble – France

[trainings@aedvices.com](mailto:trainings@aedvices.com)

Application Engineering, Design & Verification in ICs and Embedded Systems [http://www.aedvices.com](http://www.aedvices.com/)

# Training on

**IP & SoC Functional Verification Methodology**

***Using UVM***

# LAB

**Test Sequences & Coverage**

**Objectives**

This lab goes through the concept of sequences.

* Goals:
  + Understand the concept of UVM sequences
  + Implement directed test sequences
  + Implement random test sequences
* Design Example:
  + UART 16550 (opencores.org)

# Introduction

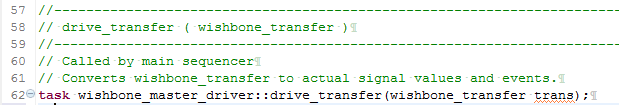
The design under test (DUT) is a UART 16550 from opencores.org.

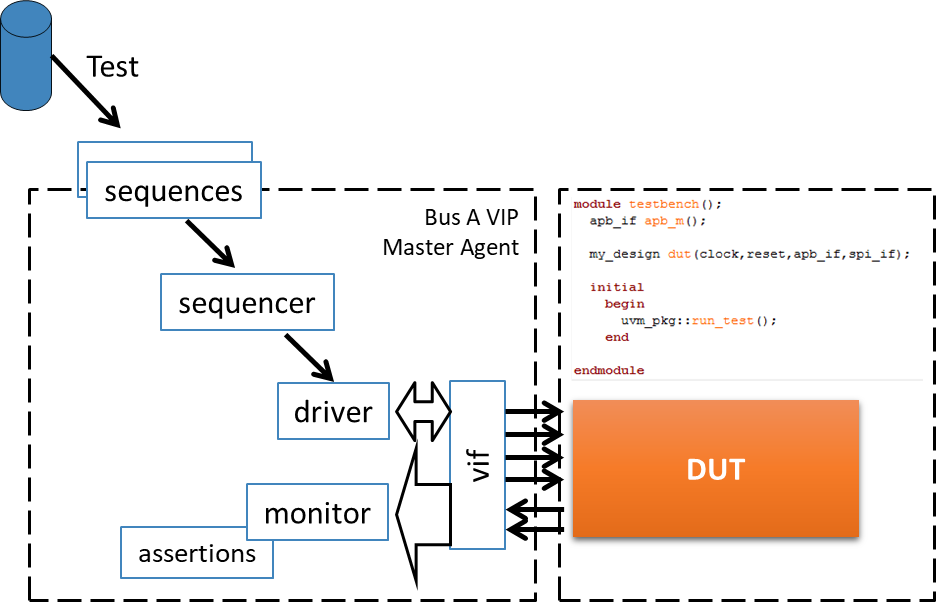
The goal of this design is to perform serial transfers on the Tx line and receive Rx transfers from the Rx line, given registers programmable thru a APB interface.

The testbench includes a APB Verification IP which main driver class is defined and implemented in the following two files:

* ambersoc\aedvices\vip\APB\src\sv\APB\_master\_driver.svh (declaration)
* ambersoc\aedvices\vip\APB\src\sv\APB\_master\_driver.sv (implementation)

The SystemVerilog task **drive\_trans()** is responsible of driving the APB bus interface from generated transactions of the test sequences.





The goal of this lab is to implement test sequences to exercise the different functionalities of the UART.

# Instructions

Follow instructions given in “aedv\_training\_labs\_intructions\_for\_questa.pdf”. Open the file “<SANDBOX>/labs-Xdays/labNN-uvm\_sequences/lab.sv” Select



# Part 1 – Sequences

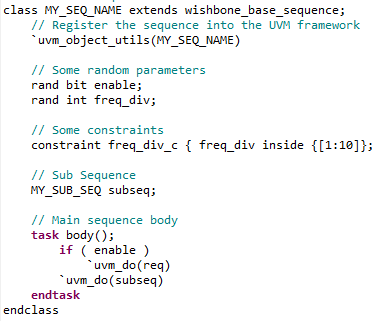
## Implement a random init sequence

The provided test is a “directed” test and only simulate one specific configuration.

In order to automate different test for different configurations, we want to implement an init sequence which randomly choose the following configuration fields:

* Interrupt enable or disable ( register IER bit 0 )
* Clock divider set to random values between 1 to 0x10 ( register DIVISOR\_LSB )
* Parity even or odd
* Char length set to 5 to 8 randomly

For this, we will create a new sequence named “lab4\_uart\_init\_seq” based on the following template



Instructions:

* Search for LAB-TODO-STEP-1-a
* Create a new class “lab4\_uart\_init\_seq” based on the above template
  + Add random parameters for each of the following
    - Interrupt enable or disable
    - Clock divider value
    - Parity even or odd
    - Char length
  + Search for LAB-TODO-STEP-1-b

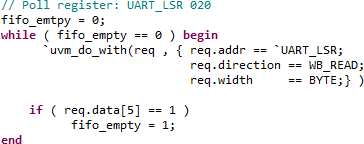
From the existing lab4\_test\_sequence, cut and paste the init part doing the register accesses (between the comments “UART Init start” and “Transmit Data”) to the newly created lab4\_uart\_init\_seq class.

* + Modify the copied register values accordingly using the random parameters.
  + Add a constraint to force the parity to be set to even when the width is set to 8
* In “lab4\_test\_sequence”
  + Search for LAB-TODO-STEP-1-c
  + Instantiate the new init sequence
  + Search for LAB-TODO-STEP-1-d
  + Call the init sequence

`uvm\_do(init\_seq)

## More Random Testing

Exercise:



* Search for LAB-TODO-STEP-2-a
* Create a sequence A which
  + Performs a random number (between 10 and 100) of write to the transmit buffer of random data. Cut and paste the template found on LAB-TODO- STEP-2-b

Note: After writing to the Tx buffer, poll check the transmit FIFO status register:

* Search for LAB-TODO-STEP-2-c
* Instantiate the test sequence
* LAB-TODO-STEP-2-d
* Use uvm\_do to call the newly created test sequence
* Search for LAB-TODO-STEP-3-a
* Create a sequence B which
  + Performs a random number (between 10 and 100) of write to the transmit buffer of incremental data (start with a random number, then increment)
* Search for LAB-TODO-STEP-4-a
* Create a sequence C which performs
  + The init sequence with only 8 bit char width
  + Select randomly one of the above two random sequences A or B ( hint: use randcase )
* Create a test sequence which randomly selects one of the above sequences ( A, B , C , init ) in a loop of 10 to 20 subsequences (see slides).
* Search for LAB-TODO-STEP-4-b
* Make the test to call your new test sequences

## Constraining Delays using uvm\_do

Exercise:

* In one of the previous random sequences, add a “delay” random variables that take a random value between 1 and 20.
* Modify the uvm\_do\_with to pass this delay to all the generated transaction ( at least in the polling loop)

`uvm\_do\_with( req , { req. delay == local:: delay; } ) Rerun a test which runs the sequence. What do you observe?

# Part 2 – Code and Functional Coverage

In this lab we have built sequences that exercise the DUT (design under test – the UART).

We now need to check how well the sequences cover or design.

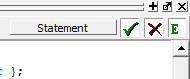
# Cover coverage

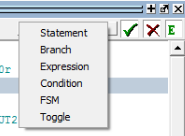
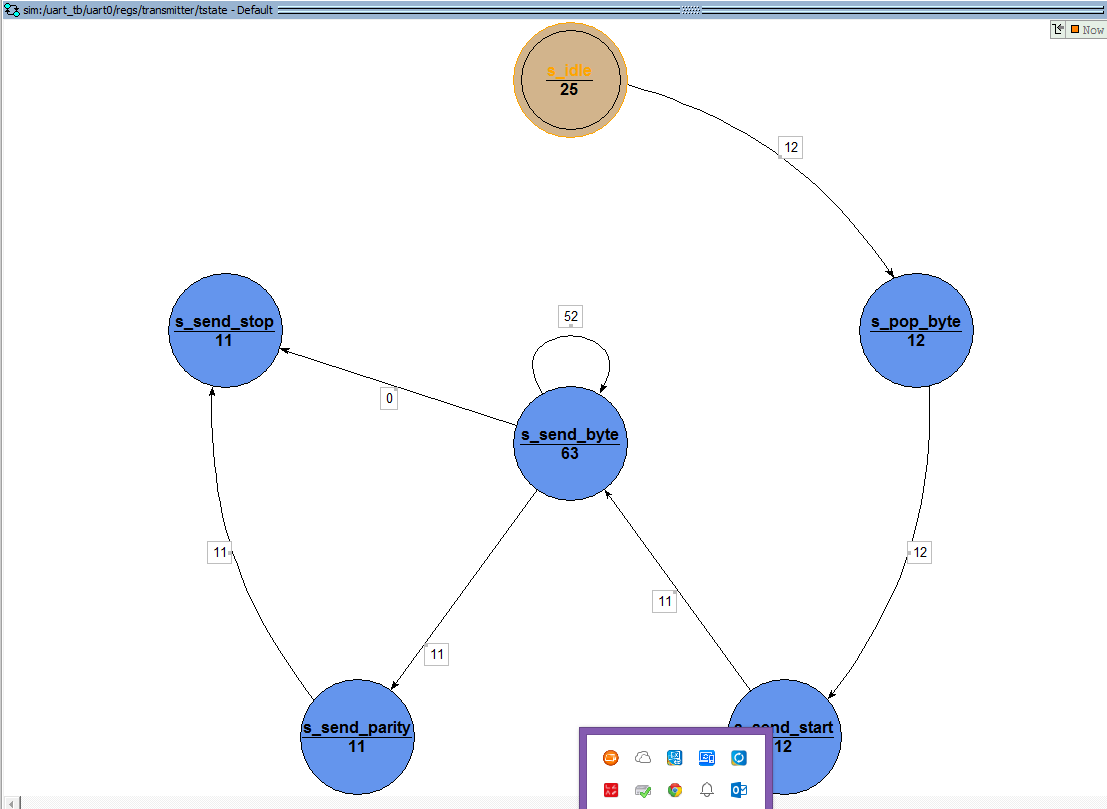
Simulate your main Lab 2 sequences.

Then open the RTL instances “UART0”, “WB\_INTERFACE”, “REGS”, and “DBG”.



* Open the RTL files of the DUT
* **Explain the lines which are not covered (globaly, don’t explain each line).**
  + Note: You can also open the Analysis window using View->Coverage->Code Coverage Analysis
  + Note: You can also sort/filter coverage holes in the analysis window:



* In the analysis window, open the toggle coverage.   
  
* **Explain (globaly, not for each signal).**
* Now look at the regs/transmitter and the regs/receiver FSM.  
  

***Note: you may have a different vision of this same FSM***

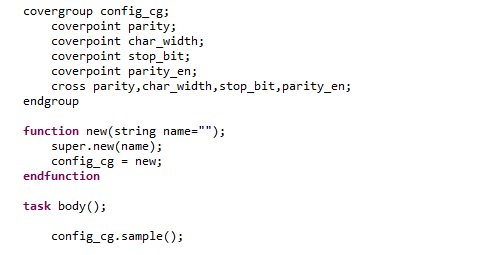
* **Explain what is not covered in the transmitter FSM.**

**Explain globally the receiver FSM coverage**

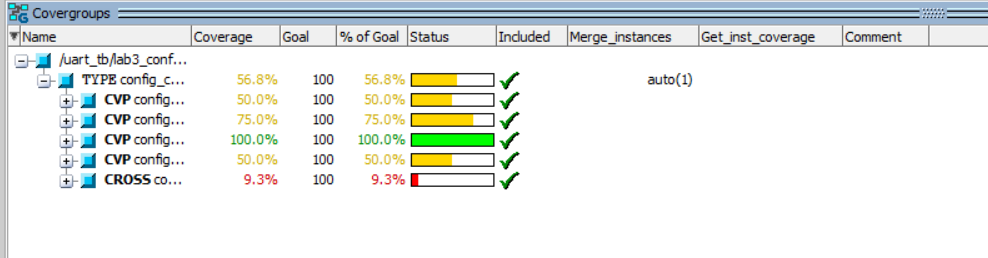
# Functional Coverage

We will implement coverage groups to add further functional information of the targeted feature.

In the config sequence (of lab2), add the following code (use the same name for the coverpoint as the config variables already used):



* Rerun test.
* Open the cover group window and analyse coverage



* Optional: Now add coverage that says:
  + Cover data values with upper bit set or not
  + Cross with char width.
  + Data bins of [0:10] , [245:255], others

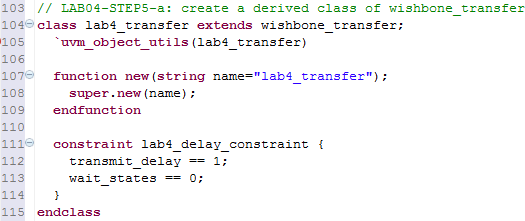
**Optional : Constraining Delays using derived class and the UVM factory**

We now want to over constrain all transactions. However we don’t want to modify the base APB\_transfer class nor each of the `uvm\_do call.

The concept of the factory is an Object Oriented Programming pattern which allow to register a new class as the one to use for a specific object. It allows to dynamically choose the type of an object at its creation time.

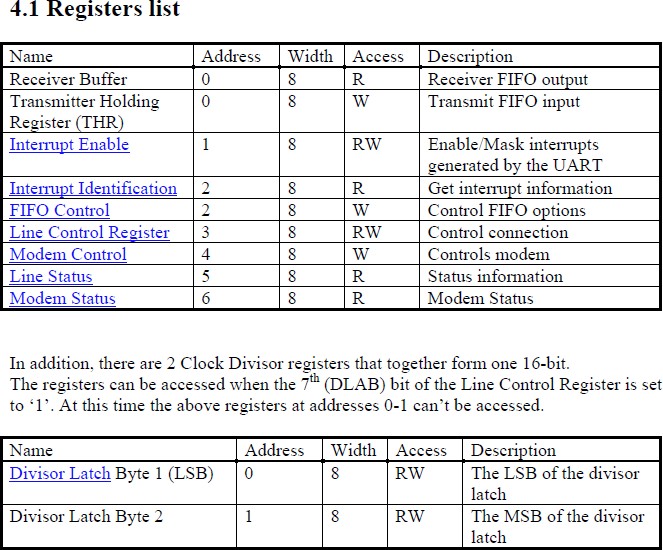
The UVM class libraries implement the factory thru the registering macros `uvm\_object\_utils and

`uvm\_component\_utils which register each class in the factory, and the factory object which allows to replace any instance.

We will use this mechanism to replace the base APB\_transfer class by our own implementation.

|  |  |
| --- | --- |
| Exercise:   * Search for LAB-TODO-STEP5-a * Implement a new class “lab\_transfer” which forces the delay to be set to 1   -  -   * Rerun the test. Does it constrain the transaction delays? * Search for * Add the following line in the build\_phase of the test. | |
|  | factory.set\_type\_override\_by\_type(apb\_transfer::get\_type(),la b\_apb\_transfer::get\_type()); |
| - Rerun the test. Does it constraint the transaction delays? | |

# Appendix A: UART registers

Note: Full detailed registers are described in the UART specification under opencores/docs