# Digital Design Verification

## Lab Manual # 37 – Creating a Stimulus Model, Test and Testbench Components

**Release: 1.0**

**Date: 5-Aug-2024**

**NUST Chip Design Centre (NCDC), Islamabad, Pakistan**

## Revision History

| Revision Number | Revision Date | Revision By | Nature of Revision | Approved By |
|---|---|---|---|---|
| 1.0 | 5/08/2024 | Amber Khan | Complete Manual | - |

## Contents

## Objective

The objectives of this lab are

- To use the UVM class library to create the YAPP packet data item and explore the automation provided.
- To start the UVM hierarchy by building the test and testbench components.

## Tools

- SystemVerilog
- Cadence Xcelium

## Instructions for Lab Tasks

The submission must follow the hierarchy below, with the folder named after the student (no spaces), and the file names exactly as listed below.

```
./student_name_lab9/
                  ├── task1_data/
                  │     ├── tb/
                  │           ├── file.f
                  │           ├── top.sv
                  │     ├── sv/
                  │           ├── yapp_pkj.sv
                  │           ├── yapp_packet.sv
                  ├── task2_test/
                  │     ├── tb/
                  │           ├── file.f
                  │           ├── top.sv
                  │           ├── router_tb.sv
                  │           ├── router_test_lib.sv
                  │     ├── sv/
                  │           ├── yapp_pkj.sv
                  │           ├── yapp_packet.sv
```

## Task 1: Creating Data Item and a Simple Test

In this task, you will learn to use UVM class library to create the YAPP packet data item. Moreover we will also explore the automation provided.

### Creating a Data Item

Working in the `lab9/task1_data/sv` directory perform the following:

1. Review the YAPP packet data specification and create your packet data class (`yapp_packet`) in a file `yapp_packet.sv`.
   a. Use `uvm_sequence_item` as the base class.
   b. Declare `addr`, `length`, `payload` and `parity` properties to match the specification.
   c. Add a `` `uvm_object_utils `` macro block containing field macros for every property.
   d. Add a UVM data constructor `new()`.
2. Add support for randomization of the packet:
   a. Declare the `length`, `addr` and `payload` properties as `rand`.
   b. Create a method `calc_parity()` to calculate and return correct packet `parity`:
   `function bit [7:0] calc_parity();`

c. Declare an enumeration type, `parity_type_e`, outside the `yapp_packet` class, with the values `GOOD_PARITY` and `BAD_PARITY`. Create a property `parity_type` as an abstract control knob for controlling parity and declare this property as `rand`.

d. Create a method `set_parity()` to assign the `parity` property:

```
function void set_parity();
```

If `parity_type` has the value `GOOD_PARITY`, assign parity using the `calc_parity()` method. Otherwise assign an incorrect `parity` value.

e. Add a `post_randomize()` method to call `set_parity()`.

f. Add a constraint for valid address.

g. Add a constraint for packet length and constrain payload size to be equal to length

h. Add a constraint for `parity_type` with a distribution of `5:1` in favor of good parity

i. Add another randomized control knob, `packet_delay`, of type int, This will be used to insert clock cycle delays when transmitting a packet. Constrain `packet_delay` to be inside the range 1 to 20.

3. Create a package named yapp_pkg in a file `yapp_pkg.sv`.
   a. First import the UVM package and include the UVM macro file:

```
import uvm_pkg::*;
`include "uvm_macros.svh"
```

   b. Then add a `include for yapp_packet.sv.

## Creating a Simple Test

4. Move to the lab01_data/tb directory.
5. Modify the top-level test module (top.sv):
   a. Import the UVM library and include the UVM macros file.
   b. Import the YAPP package (yapp_pkg) and create an instance of the YAPP packet.
   c. Using a loop in an initial block, generate five random packets and use the UVM print() method to display the results.
6. Modify the run file and simulate:
   a. Add the following lines to your run.f file:

```
-incdir ../sv // include directory for sv files
../sv/yapp_pkg.sv // compile YAPP package
top.sv // compile top level module
```

   b. Compile, simulate and check your results using the following command:

```
% xrun -f run.f
```

7. Edit the `top.sv` file to explore the UVM built-in automation: `copy()`, `clone()` and `compare()`. Also try printing using the table, tree and line printer options by adding one of the following argument to the print method:

```
uvm_default_table_printer
uvm_default_tree_printer
uvm_default_line_printer
```

**Note:** The default argument for the **print()** function is the table format.

## Task 2: Creating Test and Testbench Components

In this task, you will construct the test and testbench components of the UVM hierarchy.

1. First – create a new directory `task2_test` under the `lab9` directory, and copy your files from `task1_data/` into `task2_test/`, e.g., from the `lab9` directory, type:

   ```
   cp -r task1_data/* task2_test/
   ```

   Work in the `task2_test` directory.
2. In the `tb` directory, create a testbench in the file, `router_tb.sv` as follows:
   a. Extend your testbench from `uvm_env`.
   b. Add the `` `uvm_component_utils `` macro.
   c. Add a component constructor with `name` and `parent` arguments.
   d. Add a `build_phase()` method containing `super.build_phase(phase)`.
   e. In the build phase method, add a `` `uvm_info `` report of verbosity `UVM_HIGH` to display that the build phase of the testbench is being executed.
3. In the `tb` directory, create a test in the file, `router_test_lib.sv` as follows:
   a. Name the test class `base_test` and inherit from `uvm_test`.
   b. Add the `` `uvm_component_utils `` macro.
   c. Add a component constructor with `name` and `parent` arguments.
   d. Add a `build_phase()` method containing `super.build_phase(phase)`.
   e. Add a handle for the testbench class and construct an instance in `build_phase()`

   Add the testbench constructor call after `super.build_phase(phase)`.

   f. In the build phase method, add a `` `uvm_info `` report of verbosity `UVM_HIGH` to display that the build phase of the test is being executed.
   g. Add an `end_of_elaboration_phase()` method to the test and use the `uvm_top.print_topology()` command to print the UVM hierarchy.
4. In the `tb` directory, modify the top-level module, `top.sv` as follows:
   a. Add an include for `router_tb.sv` after the YAPP package import.
   b. Add an include for `router_test_lib.sv` after the testbench include. The order of the includes is important as the test references the testbench.
   c. Remove your existing `yapp_packet` randomization and print code.
   d. Add an initial block which calls `run_test()` to initiate phasing

### Executing the Test

5. Run a simulation with the following options added to the `run.f` file:

   ```
   +UVM_TESTNAME=base_test
   +UVM_VERBOSITY=UVM_HIGH
   ```

   Check the output from simulation and answer the following questions:

   *Does the printed topology match your expectations for the UVM hierarchy?*

   **Answer:** _______________

   *Which test class is being executed?*

   **Answer:** _______________

   *Do you see build phase reports from both test and testbench?*

   **Answer:** _______________

6. Change verbosity settings by editing the run.f option as follows:
   ```
   +UVM_VERBOSITY=UVM_LOW
   ```
   You will see different amounts of data printed when using different verbosity options.
   **Note** that the testbench is not re-compiled when you change verbosity.
7. In the test file `router_test_lib.sv`, create a second test named test2.
   Extend your test2 class from base_test. What is the minimum amount of code for `test2`, given that we are inheriting from `base_test`?
   Compile with the option `+UVM_TESTNAME=test2`, and check your topology print to make sure the correct test is being executed.
   Note that you can switch between `base_test` and `test2` via the command-line option `+UVM_TESTNAME`, without re-compiling your test environment.