



Digital Design Verification

**Lab Manual # 38 – Creating YAPP Interface UVM**

**Release: 1.0 Date: 6-Aug-2024**

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

***Copyrights ©,*** *NUST Chip Design Centre (NCDC). All Rights Reserved. This document is prepared by NCDC and is for intended recipients only. It is not allowed to copy, modify, distribute or share, in part or full, without the consent of NCDC officials.*

**Revision History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Revision Number** | **Revision Date** | **Revision By** | **Nature of Revision** | **Approved By** |
| 1.0 | 6/08/2024 | Amber Khan | Complete Manual | - |

# Contents

[Contents 2](#_bookmark0)

[Objective 3](#_bookmark1)

[Tools 3](#_bookmark2)

[Instructions for Lab Tasks 3](#_bookmark3)

[Task 1: Creating a Simple UVC 4](#_bookmark4)

[Task 2: Using Factories 6](#_bookmark5)

# Objective

The objectives of this lab are

* To the front end of a UVM Verification Component (UVC) and to explore the built-in phases of **uvm\_component**.
* To create verification components and data using factory methods, and to implement test classes using configurations.

# 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\_lab10/

## ├── task1\_uvc/

|  |  |  |  |
| --- | --- | --- | --- |
| │  │ | ├── | tb/  ├── |  |
|  | file.f |
| │  │  │  │  │  │  │  │  │  │  │  │ | ├── | ├──  ├──  ├──  sv/  ├──  ├──  ├──  ├──  ├──  ├──  ├──  ├── | top.sv router\_tb.sv router\_test\_lib.sv  yapp\_pkj.sv yapp\_packet.sv yapp\_tx\_env.sv yapp\_tx\_agent.sv yapp\_tx\_driver.sv yapp\_tx\_monitor.sv yapp\_tx\_sequencer.sv  yapp\_tx\_seqs.sv |
| ├── | task2\_factory/ | | |
| │ | ├── tb/ | |  |
| │ |  | ├── | file.f |
| │ |  | ├── | top.sv |
| │ |  | ├── | router\_tb.sv |
| │ |  | ├── router\_test\_lib.sv | |
| │ | ├── sv/ | |  |
| │ |  | ├── | yapp\_pkj.sv |
| │ |  | ├── | yapp\_packet.sv |
| │ |  | ├── | yapp\_tx\_env.sv |
| │ |  | ├── | yapp\_tx\_agent.sv |
| │ |  | ├── | yapp\_tx\_driver.sv |
| │ |  | ├── yapp\_tx\_monitor.sv | |
| │ |  | ├── yapp\_tx\_sequencer.sv | |
| │ |  | ├── | yapp\_tx\_seqs.sv |

# Task 1: Creating a Simple UVC

You will be creating the driver, sequencer, monitor, agent and env for the UVC to drive the YAPP input port of the router. You will focus on the transmit (TX) agent for this task.

1. First – copy your files from task2\_test/ into task3\_uvc/, e.g., from the lab9 directory, type:

cp –R task2\_test/\* task3\_uvc/

Work in the task3\_uvc/sv directory, implementing the UVC components.

### Creating the UVC

1. Create the yapp\_tx\_driver in the file yapp\_tx\_driver.sv.
   1. Use uvm\_driver as the base class and add a yapp\_packet type parameter.
   2. Add a component utility macro and a component constructor.
   3. Add a run\_phase() task. Use a forever loop to get and send packets, using the seq\_item\_port prefix to access the communication methods (get\_next\_item(), item\_done() ).
   4. Add a send\_to\_dut() task. For the moment, this task should just print the packet:
      * Add an `uvm\_info macro with a verbosity of UVM\_LOW.
      * Use the following code in the message portion of the macro (where <arg> is the argument name of the send\_to\_dut()task):

$sformatf("Packet is \n%s", <arg>.sprint())

**Note:** sprint()creates the print string, but does not write it to the output.

* 1. Add a #10ns delay in send\_to\_dut(). This will enable easier debugging.

1. Create the yapp\_tx\_sequencer in the file, yapp\_tx\_sequencer.sv.
   1. Use uvm\_sequencer as the base class and add a type parameter.
   2. Add a component utility macro and a component constructor
2. Create the yapp\_tx\_monitor in the file yapp\_tx\_monitor.sv:
   1. Extend from uvm\_monitor. Remember monitors do not have type parameters.
   2. Add a component utility macro and a component constructor.
   3. Add a run\_phase()task which displays a uvm\_info message of verbosity UVM\_LOW

saying you are in the monitor.

1. Create the yapp\_tx\_agent in the file yapp\_tx\_agent.sv.
   1. Extend from uvm\_agent. Remember agents do not have type parameters.
   2. Add a component utility macro and a component constructor.
   3. The agent will contain instances of the yapp\_tx\_monitor, yapp\_tx\_driver and yapp\_tx\_sequencer components. Declare handles for these and name them monitor, driver, and sequencer, respectively.
   4. The agent contains a built-in is\_active flag (inherited from uvm\_agent) to control whether the agent is active or passive. It is initialized to UVM\_ACTIVE:

// uvm\_active\_passive\_enum is\_active = UVM\_ACTIVE;

Add a field macro for is\_active within the component utilities block and set flag to

UVM\_ALL\_ON.

* 1. Add a build\_phase() method calling super.build\_phase(phase),
  2. In the build phase method, construct the driver, sequencer and monitor instances. Remember the monitor is always constructed, but the driver and sequencer are only constructed if the is\_active flag is set to UVM\_ACTIVE.
  3. Add a connect\_phase() method. Conditionally connect the seq\_item\_export of the sequencer and the seq\_item\_port of the driver, based on the is\_active flag.

1. Create and implement the UVC top level (yapp\_env) in the file yapp\_env.sv.
   1. Extend from uvm\_env. Remember uvm\_env does not have type parameters.
   2. Add a component utility macro and a component constructor.
   3. Add a handle for the yapp\_tx\_agent class.
   4. Construct the agent in a build\_phase() method. Remember to call

super.build\_phase(phase) first.

1. Edit the UVC package file, yapp\_pkg.sv in the task1\_uvc directory:
   1. Add includes for all of the files you created for this lab, together with the supplied file

yapp\_tx\_seqs.sv, in the correct order as follows:

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`include "sv/yapp\_packet.sv"

`include "sv/yapp\_tx\_monitor.sv"

`include "sv/yapp\_tx\_sequencer.sv"

`include "sv/yapp\_tx\_seqs.sv"

`include "sv/yapp\_tx\_driver.sv"

`include "sv/yapp\_tx\_agent.sv"

`include "sv/yapp\_env.sv"

### Instantiate the YAPP UVC

1. Modify the testbench (router\_tb.sv) to declare a handle for the YAPP UVC class
2. Create an instance of the handle in build\_phase().

### Checking the UVC Hierarchy

1. In the task1\_uvc/tb directory, run a simulation using the base\_test test class:
   1. Find the topology print.

*Does the hierarchy match your expectations?*

**Answer**:

* 1. Use the topology print to find the full hierarchical pathname from your test class to your UVC sequencer (e.g., tb.yapp.agent.sequencer) and write it below.

### Sequencer pathname:

* 1. Use your topology to find the value of the is\_active property of the YAPP agent. *What is the value of the is\_active variable when you printed the hierarchy?* **Answer**:

### Running a Simple Sequence

1. Open the file sv/yapp\_tx\_seqs.sv and find the sequence yapp\_5\_packets, which generates five randomized YAPP packets.

In the comment block of this sequence is a test class configuration template to set a UVC sequencer to execute this sequence.

uvm\_config\_wrapper::set(this, "<path>.run\_phase",

"default\_sequence", yapp\_5\_packets::get\_type());

* 1. Copy this code and paste it into the build phase method of the base\_test class in

tb/router\_test\_lib.sv, before the construction of the testbench handle.

* 1. **Edit** the configuration code to replace **<path>** with the hierarchical pathname to your sequencer from the test class as recorded above.

Note: We will work on sequences and configurations in later labs in detail.

1. Run a simulation using the base\_test test class:

Your UVC should now generate and print YAPP packets. Check the correct number of packets are printed and every packet field is printed.

1. Add the following compilation option to the end of you command line:

+SVSEED=random

This sets a random value for the initial randomization seed of the simulation. Re-run the Simulation(**do not recompile**) and you should see different packet data. The simulator reports the actual seed used for each simulation in the simulation log file.

1. Add a start\_of\_simulation\_phase()method to your sequencer, driver, monitor, agent, environment and testbench components.

The method should simply report a message indicating in the component from which the method is called (use `uvm\_info with a verbosity of UVM\_HIGH).

**Hint:** You can write a generic method which uses get\_type\_name() to print the component name, add string “Running Simulation …” etc, then copy this generic method into every component.

1. Run a simulation with base\_test and check which start\_of\_simulation\_phase()

method was called first. Which is called last?

Why? You will need to set the right +UVM\_VERBOSITY option to see the phase method messages.

# Task 2: Using Factories

For this lab, you will modify our existing files to use factory methods, and explore the benefits of configurations.

### Using the Factory

The first step is to use the factory methods to allow configuration and test control from above without changing the sub-components.

1. First – copy your YAPP files from task1\_uvc/ into task2\_factory/, e.g., from the lab10 directory, type:

cp –R task1\_uvc/\* task2\_factory/

Work in the task2\_factory directory.

1. Make sure you are using factory method create() and not the new() constructor calls in the build\_phase().
2. In the router\_test\_lib.sv file, modify base\_test as follows:
   1. Add a check\_phase() phase method which contains the following call:

check\_config\_usage();

This will help debug configuration errors by reporting any unmatched settings.

* 1. Add the following line to build\_phase()to enable transaction recording:

uvm\_config\_int::set( this, "\*", "recording\_detail", 1);

1. Create a new short packet test as follows:
   1. Define a new packet type, short\_yapp\_packet, which extends from yapp\_packet. Add this subclass definition to the end of your sv/yapp\_packet.sv file.
   2. Add an object constructor and utility macro.
   3. Add a constraint in short\_yapp\_packet to limit packet length to less than 15.
   4. Add a constraint in short\_yapp\_packet to exclude an address value of 2.
   5. Define a new test, short\_packet\_test, in the file router\_test\_lib.sv. Extend this from base\_test.
   6. In the build\_phase() method of short\_packet\_test, use a

set\_type\_override method to change the packet type to short\_yapp\_packet.

* 1. Run the simulation using the new test, (+UVM\_TESTNAME=short\_packet\_test), and check the correct packet type is created.

1. Create a new configuration test in the file router\_test\_lib.sv.
   1. Define a new test, set\_config\_test, which extends from base\_test.
   2. In the build\_phase() method, use a configuration method to set the is\_active property of the YAPP TX agent to UVM\_PASSIVE. Remember to call the configuration method before building the yapp\_env instance.
   3. Run a simulation using the set\_config\_test test class (UVM\_TESTNAME=set\_config\_test) and check the topology print to ensure your design is correctly configured.
   4. You should get a configuration usage report from check\_config\_usage(). Why do you get this?

Answer:

Although the configuration report maybe expected, it is good practice to minimize the number of reports where possible.

Edit your test classes so that no configuration mismatch messages are reported, but all tests still work as required. Check your changes in simulation.