Lab 8
Creating YAPP Interface UVC

Module ID : CX-301

Design Verification

Instructor : Dr. Abid Rafique

Version 1.1

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	Jan, 2024	Qamar Moavia
1.1	Modified with new exercises	Feb, 2025	Qamar Moavia

Table of Contents

Objectives

By the end of this lab, students will be able:

- 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
- Synopsys VCS

Instructions for Lab Tasks

The required files for this lab can be found on the

```
shared_folder/CX-301-DesignVerification/Labs/Lab8
```

The submission must follow the hierarchy below, with the folder named and the file names exactly as listed below.

```
./Lab8/
        ├── task1_uvc/
        │   ├── 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
        ├── 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
```

Along with that you also need to upload your solution on the github as well, and share the link.

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 `lab7` directory, type:

   ```
   cp -R task2_test/* task3_uvc/
   ```

   Work in the `task3_uvc/sv` directory, implementing the UVC components.

Creating the UVC

2. Create the `yapp_tx_driver` in the file `yapp_tx_driver.sv`.
   a. Use `uvm_driver` as the base class and add a `yapp_packet` type parameter.
   b. Add a component utility macro and a component constructor.
   c. 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()`).

d. 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.

e. Add a `#10ns` delay in `send_to_dut()`. This will enable easier debugging

3. Create the `yapp_tx_sequencer` in the file, `yapp_tx_sequencer.sv`.
   a. Use `uvm_sequencer` as the base class and add a type parameter.
   b. Add a component utility macro and a component constructor

4. Create the `yapp_tx_monitor` in the file `yapp_tx_monitor.sv`:
   a. Extend from `uvm_monitor`. Remember monitors do not have type parameters.
   b. Add a component utility macro and a component constructor.
   c. Add a `run_phase()` task which displays a `uvm_info` message of verbosity UVM_LOW saying you are in the monitor.

5. Create the `yapp_tx_agent` in the file `yapp_tx_agent.sv`.
   a. Extend from `uvm_agent`. Remember agents do not have type parameters.
   b. Add a component utility macro and a component constructor.
   c. 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.

d. 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`.
e. Add a `build_phase()` method calling `super.build_phase(phase)`,
f. 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`.
g. 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.

6. Create and implement the UVC top level (`yapp_env`) in the file `yapp_env.sv`.
   a. Extend from `uvm_env`. Remember `uvm_env` does not have type parameters.
   b. Add a component utility macro and a component constructor.
   c. Add a handle for the `yapp_tx_agent` class.
   d. Construct the agent in a `build_phase()` method. Remember to call `super.build_phase(phase)` first.

7. Edit the UVC package file, `yapp_pkg.sv` in the `task1_uvc` directory:
   a. 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

8. Modify the testbench (`router_tb.sv`) to declare a handle for the YAPP UVC class
9. Create an instance of the handle in `build_phase()`.

Checking the UVC Hierarchy

10. In the `task1_uvc/tb` directory, run a simulation using the `base_test` test class:
    a. Find the topology print.

    Does the hierarchy match your expectations?

    Answer: _____

    b. 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: _____

    c. 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

11. 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());
```

    a. 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.
    b. 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.

12. 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.

13. Add the following compilation option to the end of your 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.

14. 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.

15. 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 `lab8` directory, type:
   ```
   cp -R task1_uvc/* task2_factory/
   ```
   Work in the `task2_factory` directory.

   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:
   a. 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.
   b. Add the following line to `build_phase()` to enable transaction recording:
   ```
   uvm_config_int::set( this, "*", "recording_detail", 1);
   ```
3. Create a new short packet test as follows:
   a. 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.
   b. Add an object constructor and utility macro.

c. Add a constraint in `short_yapp_packet` to limit packet length to less than 15.
d. Add a constraint in `short_yapp_packet` to exclude an address value of 2.
e. Define a new test, `short_packet_test`, in the file `router_test_lib.sv`. Extend this from base_test.
f. In the `build_phase()` method of `short_packet_test`, use a `set_type_override` method to change the packet type to `short_yapp_packet`.
g. Run the simulation using the new test, (`+UVM_TESTNAME=short_packet_test`), and check the correct packet type is created.

4. Create a new configuration test in the file `router_test_lib.sv`.
   a. Define a new test, `set_config_test`, which extends from `base_test`.
   b. 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.
   c. 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.
   d. 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.

Good Luck 🙂