
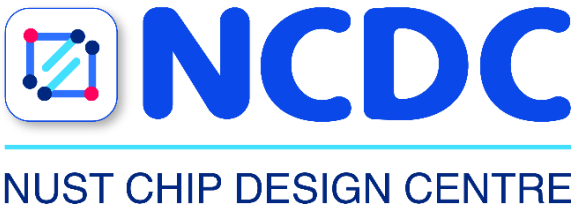


# Digital Design Verification

## Lab Manual # 43– Register Level Modeling

**Release: 1.0**

**Date: 20-Aug-2024**

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

## Revision History

| Revision Number | Revision Date | Revision By | Nature of Revision | Approved By |
|---|---|---|---|---|
| 1.0 | 20/08/2024 | Saad Khan | Complete Manual | - |

## Contents

## Objective

The objective of this lab is

## 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_lab15/
                ├── task1_rm_gen/
                │   ├── reg_verifier_dir/
                │   ├── README
                │   ├── yapp_router_reg.xml
                ├── task2_rm_integ/
                │   ├── sv/
                │   ├── tb/
                ├── router_rtl
                ├── yapp
                ├── hbus
                ├── channel
                ├── router_module_uvc  /* ignore in case its in the sv directory */
                ├── clk_and_reset
```

## Task 1: Generation

In this task, you will generate the Register Model using Cadence's reg_verifier tool and execute a quick test to verify the model is correct.
Work in the directory lab15/task1_rm_gen:

1. View the IP-XACT XML register description file: yapp_router_regs.xml.
   a. You are not expected to understand the file structure or syntax, but it is useful to be able to check basic information.
      What is the access policy of the control register (ctrl_reg)?
      Answer: _____
      What is the access policy of the address 0 register (addr0_cnt_reg)?
      Answer: _____

2. View the reg_verifier command line in the file README.txt:

| | |
|---|---|
| **reg_verifier** | |
| **-domain uvmreg** | Create a UVM register model |
| **-top yapp_router_regs.xml** | Input IP-XACT file |
| **-dut yapp_router_regs** | Top component name in IP-XACT file |
| **-out_file yapp_router_regs** | Output filename |
| **-quicktest** | Generate quick test |
| **-cov** | Generate coverage code |
| **-pkg yapp_router_reg_pkg** | Package name |

3. Use copy-paste to execute the reg_verifier command and create the register model. The files are generated in the subdirectory reg_verifier_dir/uvmreg.
   a. Change directory to reg_verifier_dir/uvmreg.
   b. Following files are generated by reg_verifier:

| | |
|---|---|
| **yapp_router_regs_config.dat** | Configuration information |
| **yapp_router_regs_hdlpaths.dat** | Path information for backddoor access |
| **yapp_router_regs_rdb.sv** | Register Model |
| **cdns_uvmreg_utils_pkg.sv** | Cadence utility package |
| **quicktest.sv** | UVM test to verify model |

4. The quicktest option creates a test to create, print and reset the register model. We can edit this test to extract more model information. Edit quicktest.sv as follows:
   a. In the run phase method of class qt_test, move model.print(); to after model.reset();. This allows us to print and check register reset values.
   b. Add the following line after the moved model.print(); line:

```
model.default_map.print();
```

      This will print the address map for the HBUS interface. This is the only interface to the DUT registers, and so can be accessed through the default name of default_map.

5. Run the test as follows:

```
make run_test
```

6. View the register model information in the simulator log file.
   a. Note the type of the model – yapp_router_regs_t. You will need to create a handle of this type to integrate the register model into your testbench.

b. The model print shows the hierarchy of a register block (router_yapp_regs) containing registers (e.g. en_reg) which contain fields (e.g. router_en). The model also contains the two memories. Use the model print to answer the following:
What is the reset value of the plen field of ctrl_reg? _____
What is the size of the yapp_pkt_mem? _____
What is the access policy of addr3_cnt_reg? _____

c. The address map print (uvm_reg_map) shows the register addresses and memory starting addresses for access via the HBUS interface. Use the map print to answer the following:
What is the address of mem_size_reg? _____
What is the starting address of the packet memory? _____

## Task 2: Integration

In this task, you will:

- Instantiate the Register Model module in your testbench.
- Run a simple test using a built-in register sequence.

1. We need a working set of lab files to integrate the register model. Use your latest completed lab – any lab from multiple UVC integration onwards can be used. Copy your selected lab into the lab15/task2_rm_integ directory.

2. Copy the following register model files from

   lab15/task1_rm_gen/reg_verifier_dir/uvmreg into lab15/task2_rm_integ/tb:
   ```
   yapp_router_regs_config.dat
   yapp_router_regs_hdlpaths.dat
   yapp_router_regs_rdb.sv
   cdns_uvmreg_utils_pkg.sv
   ```

3. Integrate the register model and adapter into the testbench (router_tb.sv) as follows

   (Hint some code is provided in the file rm_integration.txt):

   a. Add local handles for the register model and HBUS adapter (from the HBUS UVC):
   ```
   yapp_router_regs_t yapp_rm;
   hbus_reg_adapter reg2hbus;
   ```

   b. Add a field automation macro for the register model to the component utility:
   ```
   `uvm_field_object(yapp_rm, UVM_ALL_ON)
   ```

   c. In the build phase, instantiate and configure the register model as follows:

   - Create the register model instance
   - Call the methods build and lock_model on the instance to build the hierarchy, lock the model and create the address map.
   - Then set the topmost hierarchical pathname for backdoor access to the DUT: yapp_rm.set_hdl_path_root("hw_top.dut");
   - Set auto (implicit) prediction for the model using the following code: yapp_rm.default_map.set_auto_predict(1);

   d. Finally, in build phase, create the HBUS adapter instance.

e. In the connect phase, set the sequencer and adapter for the model address map:

```
yapp_rm.default_map.set_sequencer(
      hbus.masters[0].sequencer, reg2hbus);
```

Where hbus is the instantiation name for the HBUS UVC in the testbench. Make this name to match your instantiation if it is different.

4. The register model package is in the file yapp_router_regs_rdb.sv. Check the package name in the file and import the package into tb_top.sv before referencing the router testbench.
5. Finally, you need to add the following register model files to your run.f file.

```
cdns_uvmreg_utils_pkg.sv
yapp_router_regs_rdb.sv
```

Note that you do not need to compile the config and hdlpaths dat files. However, they must be in the tb directory as they are read by the register model package.

6. Copy the uvm_reset_test class from the file uvm_reset_test.sv to the end of router_test_lib.sv file. Note that the reset test:
   - Creates an instance of the built-in sequence uvm_reg_hw_reset_seq.
   - Sets the model property of the sequence instance via a hierarchical pathname.
   - Uses a start method call to execute the sequence.

   a. Find the following line in uvm_reset_test and update the testbench (tb) and model (yapp_rm) instance names to match your instances: reset_seq.model = tb.yapp_rm;
   b. Copy a default sequence setting for the clock and reset UVC from another test into the uvm_reset_test class build phase.
7. Edit run.f file to change UVM_TESTNAME to uvm_reset_test and run a simulation. The reset sequence:
   - Resets the register model.
   - Reads all the registers in the DUT
   - Compares the value read with the expected reset value from the register model.

   Carefully check the simulation output to confirm:
   - The register model is printed as part of the testbench hierarchy.
   - There are no errors and any warnings are understood.

### Testing the Memory

There is a built-in register sequence to test memory, uvm_mem_walk_seq, which executes a "walking-ones" algorithm. The sequence will automatically test all read-write memories in a register model. We can only use this to test the yapp_mem, as the yapp_pkt_mem is read-only.

8. Create the memory test by modifying the file router_test_lib.sv as follows:
   a. Create a new test by copying uvm_reset_test and rename the test to uvm_mem_walk_test. Remember to update the utility macro argument.
   b. Change all occurrences of uvm_reg_hw_reset_seq, in the uvm_mem_walk_test test to uvm_mem_walk_seq.
   c. Change the sequence handle name to something more meaningful.
9. Select the memory test by editing the run.f file.
10. Re-run the simulation. Check the log carefully to make sure there are no errors.

    The HBUS transactions should cover the whole address space of yapp_mem, from 'h1100 to 'h11ff.

    For a 256 location memory, the test should result in 511 write and 255 read operations. Check you have the correct number of HBUS transactions reported in the summary.

11. There is an option to inject an error into the design. Re-run the simulation with the following command and check the error is detected by the test:

```
xrun -f run.f -define INJECT_ERROR
```

## Task 3: Simulation

In this task, you will:

- Use the register access methods to verify the accessibility and then the functionality of the router registers.

For simplicity, work in the directory lab15/task2_rm_integ/tb.

**Access Verification**

First we will test basic access for selected registers.

1. Create a new test in router_test_lib.sv, named reg_access_test by copying and modifying uvm_reset_test.
2. Declare a convenience handle for the register block (of type yapp_regs_c) and assign the handle to register block instance using a hierarchical pathname. Use the topology report from the previous lab to find the pathname. For example:

```
...
tb                    router_tb ...          \\ testbench
  yapp_rm             yapp_router_regs_... \\ register model
   router_yapp_regs   yapp_regs_c ...        \\ register block
     addr0_cnt_reg    addr0_cnt_reg_c...     \\ registers
...
```

3. Add register access calls to the test run phase to verify selected registers as follows:

a. Select one RW register and test as follows:

- Front-door write a unique value.
- Peek and check the DUT value matches the written value.
- Poke a new value.
- Front-door read the new value and check it matches.

b. Select one RO register and test as follows:

- Poke a unique value.
- Front-door read and check the value matches.
- Front-door write a new value.
- Peek and check the DUT value has not changed.

c. Use reports with verbosity UVM_NONE to document each access.

4. Simulate reg_access_test with the -access rwc option (to allow back-door access) and check the results. What happens when you write to a RO register?

   Note both en_reg and ctrl_reg contain reserved bits. The behavior of reserved bits in the router is undefined. Also, the mem_size_reg only processes the bottom 6 bits. This affects the values which can be written to and read from these registers.
   In real life we would test all the registers by using introspection methods to create queues of RW and RO registers, and then executing the methods on every queue element.

**Functional Verification.**

To check the behavior of the registers, we will need to execute YAPP transactions in the test class.

5. Create a new test in router_test_lib.sv, named reg_function_test by copying and modifying reg_access_test.

Edit the test as follows:

a. Declare a handle of the YAPP sequencer type and in the connect phase, assign the handle to your YAPP UVC sequencer instance using a hierarchical pathname.
b. Declare a handle of your YAPP 012 sequence (which sends a packet to each channel) and create an instance in the build phase.
c. Also in the build phase, add a default sequence setting for the Channel UVCs (to channel_rx_resp_seq) by copying from a previous test.
d. In the run phase, create the following stimulus (all register access should be front door unless specified otherwise):

- Use write to set only the router enable bit in en_reg.
- Read the enable register to check the value.
- Execute the YAPP 012 sequence instance using a start call. Start syntax is:
  <sequence instance>.start(sequencer handle);
- Read all four address counter registers (addr0_cnt_reg to addr3_cnt_reg) and check they have not been incremented.
- Set all the enable bits by writing 8'hff to en_reg.
- Execute the YAPP 012 sequence instance twice using a start call.
- Read all four address counter registers (addr0_cnt_reg to addr3_cnt_reg) and check they have been incremented correctly.
- Also use reads to check the parity error and oversized packet counters.

6. Finally, simulate reg_function_test and check for correct behavior.

**Automatic Checking on Read**

Register verification can be simplified by enabling check-on-read, where a value read from the DUT is automatically checked against the register model value. However for RO registers, we will need to use manual prediction to set expected values into the model.

7. Enable automatic checking of read values against the mirrored value in the register model, by calling the following method at the start of the run_phase():
   <tb instance>.yapp_rm.default_map.set_check_on_read(1);
8. Re-simulate. You should see errors on reading the RO counters, as the read DUT value does not match the register model value.
9. Use predict calls to assign the register model mirrored values with the expected results for the counters before reading the DUT register.
10. Re-simulate and check for correct behavior.

**Register Introspection**

Carrying out repeated operations on individual registers is obviously inefficient and time-consuming.

The introspection methods allow us to extract lists (queues) of registers with common characteristics, directly from the Register Model. For example, a queue of Read-Only registers or all registers in a certain address range. We can then carry out operations on every element of the queue.

11. Use introspection methods and array selection operators to create:
    a. A queue of all the RW registers.
    b. A queue of all the RO registers.

Use methods to print the names of registers in the queues to check queue contents..