SystemVerilog Accelerated Verification Using UVM

Engineer Explorer Series

Course Version 1.2.5

Lab Manual **Revision 3.0**

cadence®

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Table of Contents

SystemVerilog Accelerated Verification Using UVM

Project Overview

Throughout this training you will be developing a verification environment for a YAPP router design. These exercises will guide you through building the verification components required to verify the router design.

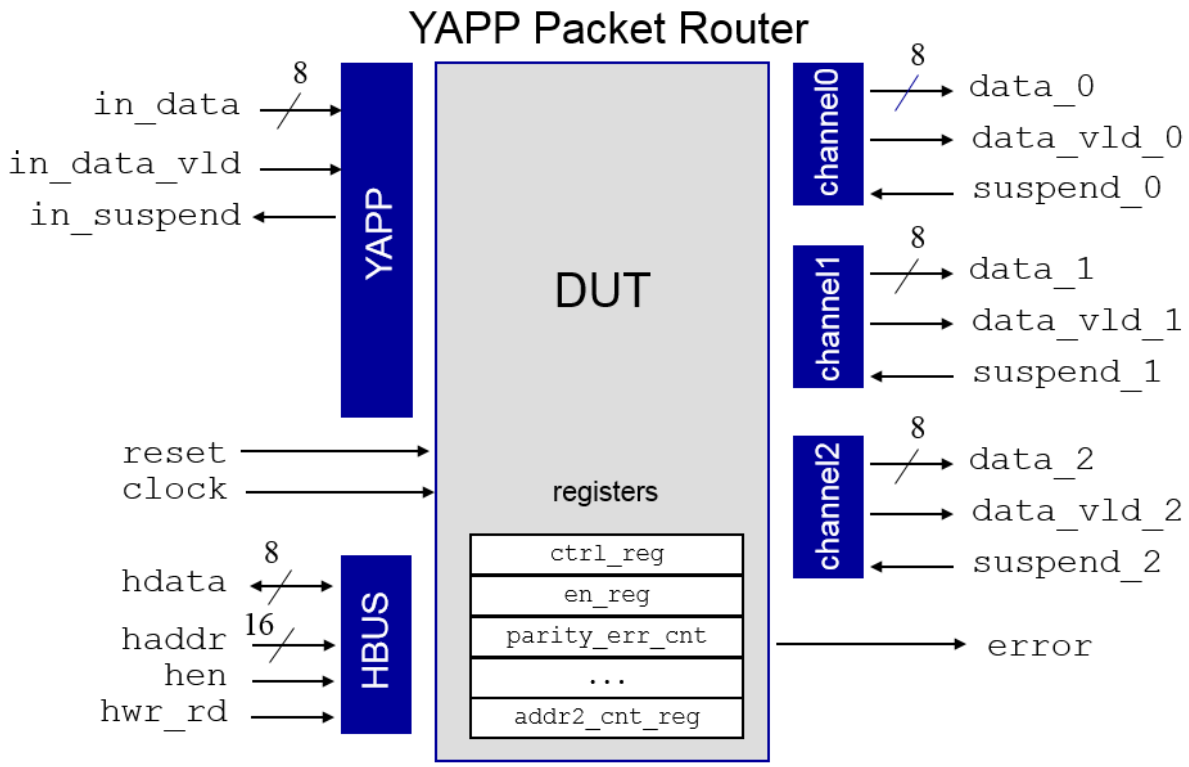
The project builds the environment from scratch so you will experience the full process.

You will be building one UVC component. The others will be provided for you later in the project.

YAPP Router Description

The YAPP router accepts data packets on a single input port, `in_data`, and routes the packets to one of three output channels: `channel0`, `channel1` or `channel2`. The input and output ports have slightly different signal protocols. The router also has an HBUS host interface for programming registers that are described in the next section.

High-Level Diagram – YAPP Router (Yet Another Packet Protocol)



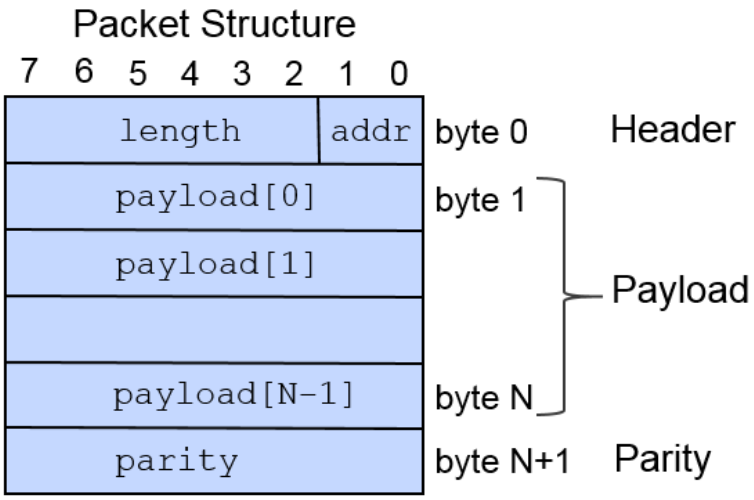
Packet Data Specification

A packet is a sequence of bytes with the first byte containing a header, the next variable set of bytes containing payload, and the last byte containing parity.

The header consists of a 2-bit address field and a 6-bit length field. The address field is used to determine which output channel the packet should be routed to, with the address *3* being illegal. The length field specifies the number of data bytes (payload).

A packet can have a minimum payload size of 1 byte and a maximum size of 63 bytes.

The parity should be a byte of even, bitwise parity, calculated over the header and payload bytes of the packet.

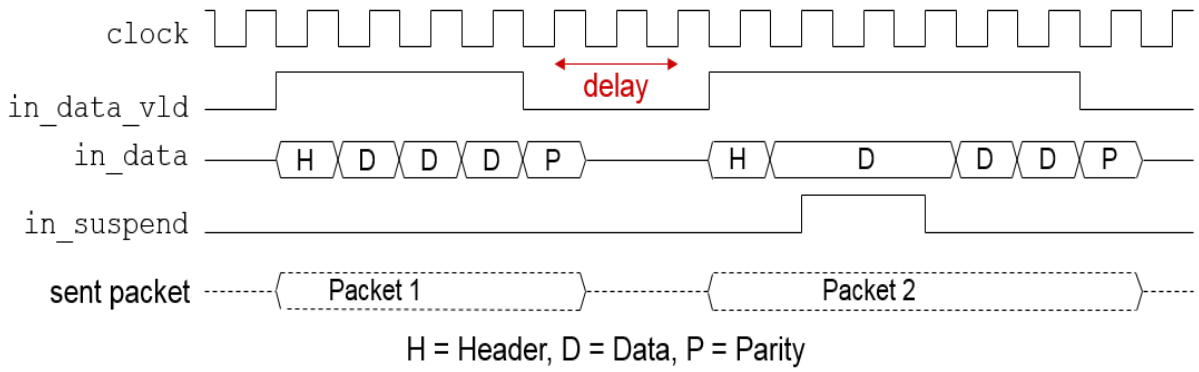


1<= N <= 63

Input Port Protocol

All input signals are active high and are to be driven on the **falling** edge of the clock. The `in_data_vld` signal must be asserted on the same clock when the first byte of a packet (the header byte) is driven onto the `in_data` bus. As the header byte contains the address, this tells the router to which output channel the packet needs to be routed. Each subsequent byte of data needs to be driven on the data bus with each new falling clock.

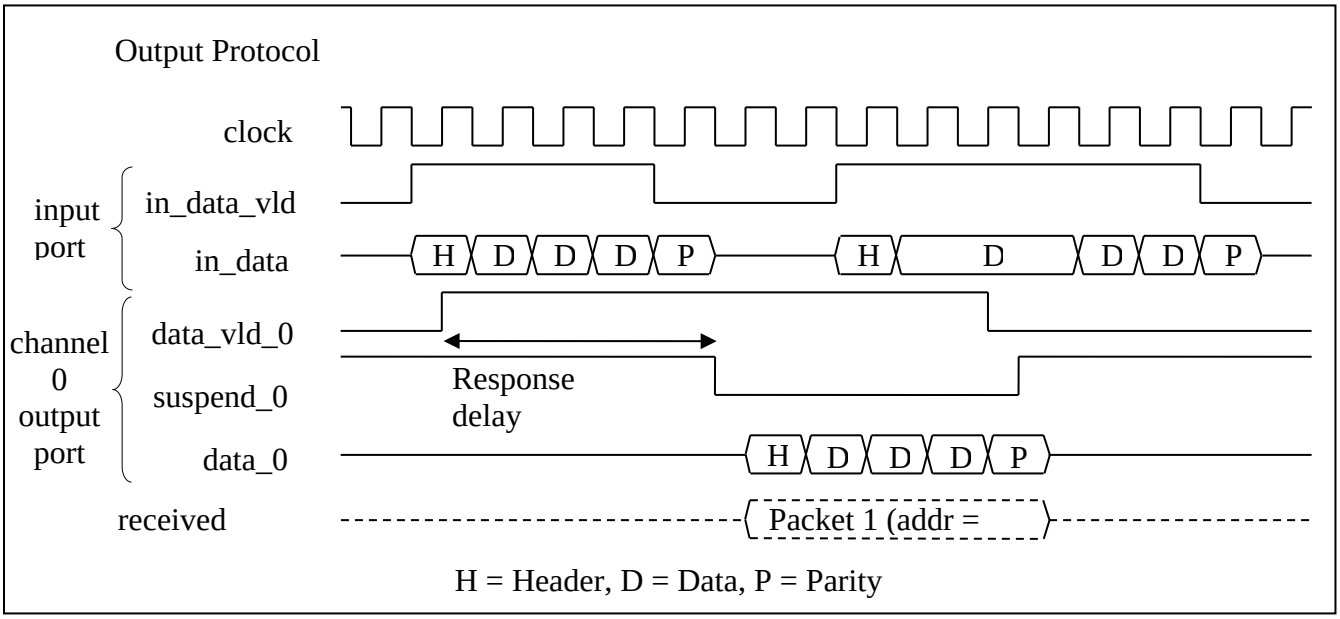
After the last payload byte has been driven, on the next falling clock, the `in_data_vld` signal must be de-asserted, and the packet parity byte needs to be driven. The input data cannot change while `in_suspend` signal is active (indicating FIFO full). The `error` signal asserts when a packet with bad parity is detected, within 1 to 10 cycles.



H = Header, D = Data, P = Parity

Output Port Protocol (Channel Ports)

All output signals are active high and are to be sampled on the **falling** edge of the clock. Each output port is internally buffered by a FIFO of depth 16 and a width of 1 byte. The router asserts the `data_vld_x` signal when valid data appears on the `data_x` output bus. The `suspend_x` input signal must then be de-asserted on the falling clock edge in which data is read from the `data_x` bus. As long the `suspend_x` signal remains inactive, the `data_x` bus drives a new valid packet byte on each rising clock edge.



Packet Router DUT Registers

The packet router contains internal registers that hold configuration information. These registers are accessed through the host interface port. Register characteristics are as follows:

address	register	reset	field	field name	policy	description
0x1000	ctrl_reg	0x3f	5:0	maxpktsize	RW	Maximum packet length
			7:6		RW	Unused
0x1001	en_reg	0x01	0	router_en	RW	Router enable
			1	parity_err_cnt_en	RW	Parity error count enable
			2	oversized_pkt_cnt_en	RW	Length error count enable
			3	[reserved]	RW	Not implemented
			4	addr0_cnt_en	RW	Address 0 packet count enable
			5	addr1_cnt_en	RW	Address 1 packet count enable
			6	addr2_cnt_en	RW	Address 2 packet count enable
			7	addr3_cnt_en	RW	Address 3 packet count enable
0x1004	parity_err_cnt_reg	0x00	7:0		RO	Packet parity error count
0x1005	oversized_pkt_cnt_reg	0x00	7:0		RO	Packet length error count
0x1006	addr3_cnt_reg	0x00	7:0		RO	Address 3 packet count
0x1009	addr0_cnt_reg	0x00	7:0		RO	Address 0 packet count
0x100a	addr1_cnt_reg	0x00	7:0		RO	Address 1 packet count
0x100b	addr2_cnt_reg	0x00	7:0		RO	Address 2 packet count

If the input packet length is greater than the `maxpktsize` field of the `ctrl_reg` register, then the router drops the entire packet.

The `router_en` field of the `en_reg` register controls the enabling and disabling of the router. A disabled router drops all packets. Enabling or disabling the router during packet transmission will yield to unpredictable behavior.

The router counters are enabled by individual bits in `en_reg`. The router specification says that if these bits are changed while the router is processing a packet, then the router behavior is undefined.

The router counters are defined as follows:

`parity_err_cnt_reg` – incremented when a bad parity packet is received

`oversized_pkt_cnt_reg` – incremented when packet with length greater than `maxpktsize` is received

`addr3_cnt_reg` – incremented when a packet with an illegal address (3) is received

`addr0_cnt_reg` – incremented when a packet with address 0 is received

`addr1_cnt_reg` – incremented when a packet with address 1 is received

`addr2_cnt_reg` – incremented when a packet with address 2 is received

Router Memories

The router contains two memory blocks as follows:

Start Address	Name	Size	Policy	Description
0x1010	yapp_pkt_mem	[0:63]	RO	Stores the bytes of the last packet received by the yapp router.
0x1100	yapp_mem	[0:255]	RW	“Scratch” memory

Host Interface Port Protocol (HBUS)

All input signals are active high and are to be driven on the **falling** edge of the clock. The host port provides synchronous read/write access to program the router.

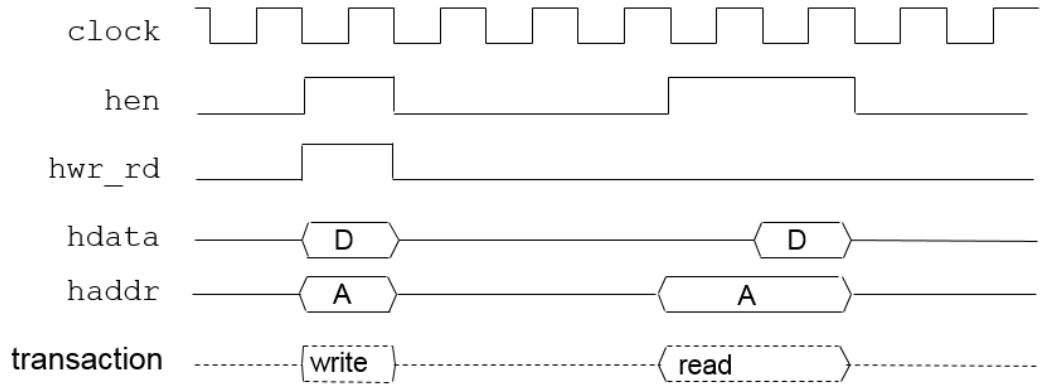
A WRITE operation takes one clock cycle as follows:

- `hwr_rd` and `hen` must be 1. Data on `hdata` is then clocked on the next rising clock edge into the register based on `haddr` decode.
- `hen` is driven to 0 in the next cycle.

A READ operation takes two clock cycles as follows:

- `hwr_rd` must be 0 and `hen` must be 1. In the first clock cycle, `haddr` is sampled and `hdata` is driven by the design under test (DUT) in the second clock cycle.
- `hen` is then driven low after cycle 2 ends. This will cause the DUT to tri-state the `hdata` bus.

The HBUS port provides synchronous read/write access to program the router.



A = Address, D = Data

Before You Begin

Finding Your Files

The UVM training lab database includes the following directories and files:

`uvma_training_`*`xx`*`/uvm`	Top-level directory for the training
`lab`*`xx`*`/`	Lab directories where you will be working
`hbus`	HBUS UVC
`channel`	Output channel UVC
`clock_and_reset`	Clock and Reset UVC (controls clock/reset)
`yapp`	YAPP input port UVC (created in labs)
`router`	Router module UVC (created in labs)
`router_rtl`	RTL Files for the YAPP Router DUT
`test_install`	Simple example to check your UVM installation
`uvm_register_modeling`	Files for standalone UVM Register Modeling lab

Accessing UVM

UVM 1.2 is supported by INCISIVE 15.2 and XCELIUM 17.04 onwards.

For all simulator releases, there are two options for accessing the UVM library:

- Use one of the libraries provided with your simulator installation.
- Download the library from `uvmworld.org`. (which redirects to `accellera.org`).

The first option is the best solution for Cadence® users, as the library contains extra Cadence® debug additions for UVM such as transaction recording.

Setting Up UVM

1. Set an environmental variable to the install path of the required UVM library release: e.g., for the UVM library provided with Cadence simulators:

   ```
   setenv UVMHOME `ncroot`/tools/methodology/UVM/CDNS-1.1d
   ```

   We currently recommend UVM1.1d for *training* due to transaction issues in UVM1.2.

Testing Your UVM Installation

2. Change the directory to `test_install`.

3. Run the test by executing the following command:

   ```
   xrun –f run.f
   ```

   (or `irun -f run.f` for INCISIVE users)

4. In the output log file (`xrun.log`), check the version number of the UVM library. You should see:

   ```
   ----------------------------------------------------------
   UVM-1.x
   ...
   ```

   Where `UVM-1.x` is the UVM library version you are using.

5. In the test output, check the installation test passed. You should see:

   ```
   UVM_INFO  ...  UVM TEST INSTALL PASSED!
   ```

Accessing Help Files

You may wish to bookmark the following file in a web browser to give convenient access to the UVM HTML documentation

```
$UVMHOME/sv/docs/html/index.html
```

Additional Notes

File names, component names, and instance names are suggested for many of the labs. You are not required to use these names but if you do not, you may need to edit code provided later in the week to match your path and instance names.

These labs do not include step-by-step instructions, and do not tell you exactly what you need to type.

Lab 1 Creating a Stimulus Model

Objective: To use the UVM class library to create the YAPP packet data item and explore the automation provided.

For this lab, start in the `lab01_data/sv` directory.

Creating a Data Item

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. (Optional) Edit the `top.sv` file to explore the UVM built-in automation: `copy()`, `clone()` and `compare()`. Also try printing using the table and tree printer options.

Lab 2 Creating Test and Testbench Components

Objective: To start the UVM hierarchy by building the test and testbench components.

For this lab, you will construct the test and testbench components of the UVM hierarchy.

1. **First** – create a new directory `lab02_test`, under the `uvm` directory, and copy your files from `lab01_data/` into `lab02_test/`, e.g., from the `uvm` directory, type:

   ```
   %  cp –r lab01_data/* lab02_test/
   ```

   Work in the `lab02_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 severity `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. (Optional) 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`?

   (Optional) 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.

End of Lab

Lab 3 Creating a Simple UVC

Objective: To the front end of a UVM Verification Component (UVC) and to explore the built-in phases of `uvm_component`.

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 lab.

1. **First** – copy your files from `lab02_test/` into `lab03_uvc/`, e.g., from the `uvm` directory, type:

   ```
   %  cp –R lab02_test/* lab03_uvc/
   ```

   Work in the `lab03_uvc/sv` directory. Use the training slides for suggestions on implementing these 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 an `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.

   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`**:

   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:

```
`include "yapp_packet.sv"
`include "yapp_tx_monitor.sv"
`include "yapp_tx_sequencer.sv"
`include "yapp_tx_seqs.sv"
`include "yapp_tx_driver.sv"
`include "yapp_tx_agent.sv"
`include "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 `lab03_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: Both sequences and configurations are covered in later modules of this course.

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 and you should see different packet data. The simulation reports the actual seed used for each simulation in the log file.

14. (Optional) 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, and then copy this generic method into every component.

15. (Optional) 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.

Lab 4 Using Factories

Objective: To create verification components and data using factory methods, and to implement test classes using configurations.

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

16. **First** – copy your YAPP files from `lab03_uvc/` into `lab04_factory/`, e.g., from the `uvm` directory, type:

```
%  cp –R lab03_uvc/* lab04_factory/
```

Work in the `lab04_factory` directory.

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.

17. Replace the `new()` constructor calls in the `build_phase()` methods by calls to the factory method `create()`. You will need to modify the following files:

```
tb/router_test_lib.sv
tb/router_tb.sv
sv/yapp_env.sv
sv/yapp_tx_agent.sv
```

18. Run your original test (`base_test`) to make sure the changes are working.

Using Configurations and Overrides

19. 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);
```

20. 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 a simulation using the new test, (`+UVM_TESTNAME=short_packet_test`), and check the correct packet type is created.

21. 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.

Lab 5 Generating UVM Sequences

Objective: To use the uvm_sequence mechanism to define a sequence library and to control execution of sequences.

For this lab, you will explore sequence writing and the objection mechanism for coordinating simulation time.

Creating Sequences

1. **First** – copy your YAPP files from `lab04_factory/` into `lab05_seq/`.

   Work in the `lab05_seq` directory.

2. In the `sv` directory, edit the sequences file, **`yapp_tx_seqs.sv`**, to add the sequences defined below in steps 3 to 7 (and optionally steps 8 and 9).

   For every sequence:

   a. Inherit the sequence from `yapp_base_seq` to use the objection mechanism.

   b. At the start of every sequence `body()`, add an `` `uvm_info `` call to print the sequence name. Use a verbosity of `UVM_LOW`.

   c. Remember to add a data constructor and an **object** utilities macro.

3. Create a single packet sequence with constraint:

   `yapp_1_seq` – single packet to address 1
   (addr==1)

4. Create a multi-packet sequence with different constraints for each packet:

   `yapp_012_seq` – three packets with incrementing addresses
   (addr ==0; addr==1; addr==2)

5. Create a nested sequence.

   `yapp_111_seq` – three packets to address 1
   (do `yapp_1_seq` three times)

6. Create a repeating address sequence.

   `yapp_repeat_addr_seq` – two packets to the same (random) address
   (addr==prev_addr: remember packet address cannot be 3)

   Hint: Use a random sequence property with constraint.

7. Create a sequence to generate a **single** packet with incrementing payload data.

   `yapp_incr_payload_seq` –
   Create a single packet to send.
   Randomize the packet.
   Set the payload values of the single packet to increment from 0 to (length -1).
   Update parity.
   Send the packet.
   Hint: Use `` `uvm_create `` and `` `uvm_send `` macros.

Running a Test Using a New Sequence

8. Create a new test in the file **`router_test_lib.sv`** from the `tb` directory:

   d. Call the test `incr_payload_test` and extend from `base_test`.

   e. Add a `uvm_config_wrapper::set` to set the `run_phase` default sequence to `yapp_incr_payload_seq`.

   f. Add a `set_type_override()` method to use the `short_yapp_packet` data type defined in Lab 4.

   g. Run the test and verify the results. Setting verbosity to `UVM_FULL` will allow you to see which default sequence is executed in the `run_phase()`.

Testing Your Sequences

You need to check that every one of your new sequences works correctly before we progress any further. There are several ways to do this, but the easiest is to create a single sequence which executes all the sequences you need to test.

9. Edit the sequences file, **`sv/yapp_tx_seqs.sv`**, to add the following sequence:

   `yapp_exhaustive_seq` – execute all sequences to test
   (Do all of your user-defined sequences).

   Remember to extend from the base sequence and add a data utility macro and constructor. Using meaningful names for the sequence instances will help in debug.

10. Create a new test in the file **`tb/router_test_lib.sv`**:

    a. Call the test `exhaustive_seq_test` and extend from `base_test`.

    b. Add a `uvm_config_wrapper::set` to set the `run_phase` default sequence to yapp_`exhaustive_seq`.

c. Add a `set_type_override()` method to use the `short_yapp_packet` data type defined in Lab04.

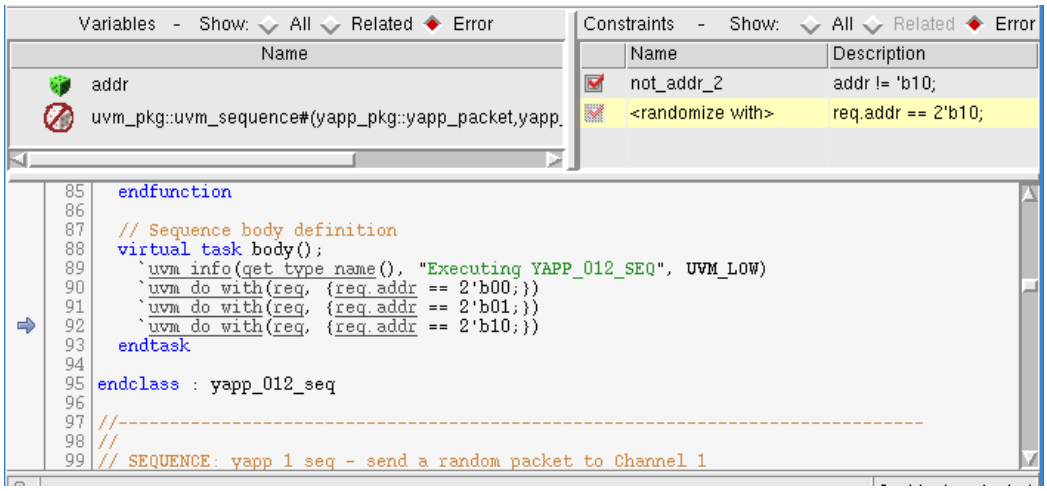
Testing Your Sequences and Fixing Randomization Errors

11. Run the test (`+UVM_TESTNAME=exhaustive_seq_test`) and check the results.

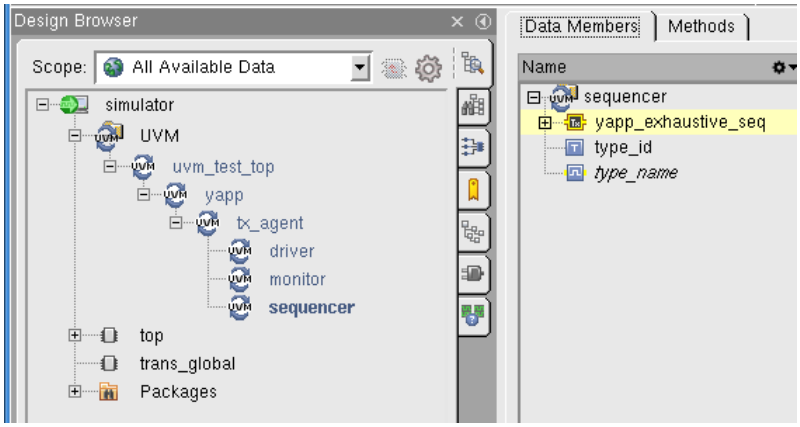
Note: You should get randomization **failures** for particular sequences due to constraint violations. Examine the simulation output carefully to see how the constraint violations are reported. Note that in batch mode, simulation is **NOT** stopped.

Using Constraint Manager and Transaction Debug Features

12. Run a simulation using the GUI. The simulation will now stop on a constraint violation and the Constraints Manager window opened. This allows complex constraint violations to be easily debugged.



13. In the Design Browser, navigate down the UVM hierarchy to your sequencer instance and find the sequence transaction (same name as your sequence). Add this to the waveform viewer.



Expand the transaction in the waveform viewer to see the sequence hierarchy and generated packets.

14. Use the Constraints Debugger and Sequence transaction to answer the following questions:

    Why do you get randomization violations?

    Answer: _____

    What happens to the packet when a constraint violation is found?

    Answer: _____

    How could you fix these violations?

    Answer: _____

15. **Fix your code** to make the simulation run without constraint violations.

    Hint: We will want to send short packets to every address.

    It is important that you fix the randomization violations before moving on to the next lab.

16. (Optional) Create an additional sequence to generate a random number of packets.

    `yapp_rnd_seq` –
    Declare a `rand int` property `count` in the sequence and generate a number of random packets according to the `count` value.
    Set a constraint inside the sequence to limit `count` to the range 1 to 10.
    Include the value of `count` in the sequence `` `uvm_info `` message.

17. (Optional) Create a nested sequence with a constraint.

    `six_yapp_seq` – do `yapp_rnd_seq` with `count` constrained to six.

18. (Optional) Add **`yapp_rnd_seq`** and **`six_yapp_seq`** to the **`yapp_exhaustive_seq`** and verify their behavior in simulation.

Lab 6 Connecting to the DUT Using Virtual Interfaces

Objective: To connect the YAPP UVC to the input ports of the DUT.

For this lab, you will connect your YAPP UVC to the RTL router Design Under Test (DUT) using interfaces and virtual interfaces.

Modifying the YAPP UVC

1. **First** – copy your YAPP files from `lab05_seq/` into `lab06_vif/`.

   Work in the `lab06_vif` directory.

2. Check the YAPP interface, **`yapp_if.sv`** in the `sv` directory. The interface has:
   - Two input ports (`clock`, `reset`)
   - Three DUT signals, `in_data`, `in_data_vld` and `in_suspend`.
   - Three methods, `yapp_reset()` and `send_to_dut()` for the driver, and `collect_packets()` for the monitor

3. Connecting the YAPP interface via the configuration database will be easier if you declare a `typedef` for the `uvm_config_db` with a `yapp_if` type parameter. This declaration has to be visible to your monitor, driver, and top-level module.

   Add the following declaration to `yapp_pkg.sv`, before the include statements:

   ```
   typedef uvm_config_db#(virtual yapp_if) yapp_vif_config;
   ```

4. Update your Monitor, **`yapp_tx_monitor.sv`**.

   a. Add a declaration for the virtual interface:

   ```
   virtual interface yapp_if vif;
   ```

   b. The monitor must call the `collect_packets()` interface method (using the virtual interface) to capture the packet data.

   The method call and *some* of the declarations for the monitor are provided in the file **`monitor_example.sv`**. Check you understand the code.

   Copy the code from the example into your monitor.

   c. Add a `connect_phase()` method containing a `yapp_vif_config::get` call to assign `vif` from the configuration database.

   Remember `get` returns bit 1 if it was successful. Test this return value in an `if` statement to help debug the virtual interface connection. For example:

```
if (!yapp_vif_config::get(this,"","vif", vif))
  `uvm_error("NOVIF","vif not set")
```

5. Update your Driver protocol, **`yapp_tx_driver.sv`**.

   a. Add a declaration for the virtual interface :

   ```
   virtual interface yapp_if vif;
   ```

   b. You need to call the `send_to_dut()` interface method to transmit packet data.

   You will also need to reset the input DUT signals when the reset is active using a `reset_signals()` method, and call this in the driver `run_phase()`.

   These methods and *some* of the declarations for the driver are provided in the file **`driver_example.sv`**. Check you understand the code.

   Use the supplied code to update your driver.

   c. Add a `connect_phase()` method containing a `yapp_vif_config::get` call to assign `vif` from the configuration database, as for the monitor.

Testing the DUT

6. Add a new test by editing **`router_test_lib.sv`**:

   d. Create a new test which sets the default sequence of the YAPP UVC to `yapp_012_seq`. Sending packets to all three output Channels will make the DUT connection test much easier.

   e. Add a `run_phase()` method to `base_test` which sets a drain time for the objection mechanism as follows:

   ```
   uvm_objection obj = phase.get_objection();
   obj.set_drain_time(this, 200ns);
   ```

   The drain time allows packets to pass through the router before simulation ends.

Initial Simulation Without the DUT

7. A top-level module, **`hw_top.sv`**, is provided for you in the `tb` directory. The module supplies the following functionality:
   - Instantiates the `clkgen` module to create the `clock` signal.
   - Declares the `reset` signal and generates the waveform required.
   - Instantiates the `yapp_if` interface:

   ```
   yapp_if in0 (clock, reset);
   ```

- Drives the `in_suspend` signal of the interface instance.

Check you understand the code in the module. No changes are required for this module at the moment.

8. In the `tb` directory, rename your UVM top level module **`top.sv`** to **`tb_top.sv`** and modify the file as follows:

   a. Change the module name to `tb_top`.

   b. Use a configuration `set` to write the YAPP interface instance into the config database as follows:

   - Use your `yapp_vif_config typedef` from step 3.
   - For a top module `set`, the context is `null`.
   - Use pathname wildcards to affect both monitor and driver with a single statement.
   - Use an absolute hierarchical pathname for the value to select the YAPP interface instance from the `hw_top` module.

9. Update your `run.f` file:

   a. Add the YAPP interface file from the `sv` directory to the list of files to be compiled. Remember interface files *must* be compiled, they cannot be included.

   b. Add **`clkgen.sv`** and **`hw_top.sv`** to the list of compilation files and change **`top.sv`** to **`tb_top.sv`**.

   c. You may need the following default timescale option in your `run.f` file to avoid timescale errors:

```
-timescale 1ns/1ns
```

10. Run a simulation using `yapp_012_test` to verify your updates are working correctly.

    a. Examine the simulation output carefully to check that the YAPP monitor is capturing correct packets according to the sequence being used.

    b. Run the simulation in GUI mode and use the waveform viewer to check your interface signals are correctly driven.

    Hint: The interface instance will appear in the Design Browser window and you can send the instance to the Waveform window to display all the interface signals.

Testing with the DUT

Now is the time to test your YAPP UVC with the actual router DUT, the model for which can be found in the `router_rtl` directory (at the same level as your lab directories).

With a few lines of extra code, the router DUT will function without any connections to the HBUS interface signals or the channel outputs. This will allow you to test your YAPP UVC connection to the DUT without having to use the HBUS or Channel UVCs.

11. Edit **`hw_top.sv`** as follows:

    a. Copy the DUT instance and port list from the file **`yapp_router_instance.txt`** and use *named mapping* to connect `clock`, `reset` and the YAPP interface signals. Leave the other ports unconnected. For example:

    ```
    .in_data(in0.in_data),   //connect YAPP data
    .data_0(),               // leave Chan0 data unconnected
    ```

    b. In the `initial` block, **remove** the following assignment to `in_suspend` as this will now be driven by the router DUT itself.

    ```
    in0.in_suspend <= 0;
    ```

    c. Set the `suspend_0`, `suspend_1` and `suspend_2` router channel ports to `1'b0` in the DUT instance port mapping. This allows packets to pass through the DUT.

12. Modify **`run.f`** to compile the **`yapp_router.sv`** file from the `router_rtl` directory.

13. Run a simulation in GUI mode to make sure the router is correctly connected.

    a. Use the SimVision waveform viewer to check packet data comes out on the right output channels, by adding the following signals from `top` to the waveform viewer:

    - `in_data` from the `in0` YAPP interface instance
    - `data_0`, `data_1` and `data_2` output ports from the `dut` module instance.

    You should see packet data going into the router on `in_data` and coming out on the correct channel data output port.

    b. You can also view the YAPP monitor and driver transactions. In the Design Browser window, navigate down the UVM hierarchy from `uvm_test_top` to your monitor and driver instances. Add the following named transactions to the waveform viewer:

    - `Monitor_YAPP_Packet`
    - `Driver_YAPP_Packet`

    These names are defined in the driver and monitor transaction method calls, e.g. in the driver `get_and_drive()` task

    ```
    begin_tr(req, "Driver_YAPP_Packet");
    ```

End of Lab

Lab 7 Integrating Multiple UVCs

Objective: To connect and configure the HBUS UVC, Clock and Reset UVC and three output Channel UVCs.

For this lab, you will connect the HBUS, Clock and Reset and Channel UVCs to the router DUT.

All three UVCs are provided. None of the UVCs use configuration objects.

These are the directories we will be using for this and subsequent labs:

`hbus/sv`	HBUS UVC files
`channel/sv`	Channel UVC files
`clock_reset/sv`	Clock and Reset UVC files
`yapp/sv`	YAPP input UVC (your files from `lab06_vif`)
`router_rtl`	Router DUT
`lab07_integ`	Your working directory for this lab

Setting Up the Directory Structure

1. **First** – your YAPP UVC is now complete enough to stand by itself. Copy your YAPP files from `lab06_vif/sv` into `yapp/sv`.

2. We will still be working on the testbench, testclass, and top files. Copy the files from `lab06_vif/tb` into `lab07_integ/tb`.

   Work in the `lab07_integ/tb` directory.

Testbench: Channel UVC

3. Update your testbench, **`router_tb.sv`**, to add the Channel UVCs.

   a. Add three handles of the Channel UVC (`channel_env`) and create the instances in the `build_phase()` method using factory calls.

   b. Use a configuration set method to set the `channel_id` property of each Channel instance. The Channel instance for address 0 should have a `channel_id` of 0, the Channel instance for address 1 should have a `channel_id` of 1and the Channel instance for address 2 should have a `channel_id` of 2. For example,

   ```
   uvm_config_int::set(this, "chan0", "channel_id", 0);
   ```

Testbench: HBUS UVC

4. Update your testbench, **`router_tb.sv`,** to add the HBUS UVC.

   a. Add a handle of the HBUS UVC (`hbus_env`) and create the instance in the `build_phase()` method using a factory call.

   b. Use configuration set methods to set the `num_masters` property of the HBUS UVC to `1`, **and** the `num_slaves` property to `0`. The HBUS UVC has both master and slave agents. For the router testing, we only need the master agent.

Testbench: Clock and Reset UVC

5. Update your testbench, **`router_tb.sv`,** to add a handle of the Clock and Reset UVC (`clock_and_reset_env`) and create the instance in the `build_phase()` method using a factory call. This UVC requires no configuration.

Hardware Top Module `hw_top`

6. Update **`hw_top.sv`** as follows:

   a. Add an interface instantiation for the Clock and Reset. The interface file can be found in the `clock_and_reset/sv` directory. Map the `clock`, `reset`, `run_clock` and `clock_period` interface ports to the local signals of the same name.

   b. Connect the `clkgen` module instance to the Clock and Reset interface instance by replacing the `run_clock` and `clock_period` literal port mappings with the local signals of the same name.

   c. Add interface instantiations for the HBUS and all three Channels. The interface files can be found in the `sv` directory of each UVC directory. Map the ports of the interfaces to the local `clock` and `reset` signals of the same name.

   d. As the `reset` will now be generated by the Clock and Reset UVC, delete the `initial` block which generates the `reset` waveform.

   e. Update the port mapping of the router instantiation to connect the Channel and HBUS interface signals.

      Warning – The HBUS interface contains a bi-directional signal `hdata`. When you connect the HBUS interface signals to the router DUT, you must use the **`wire`** net `hdata_w`, in the port mapping, not the `logic` variable `hdata`.

UVM Top Module `tb_top`

7. Update **`tb_top.sv`** as follows:

   a. Add imports for the Channel, Clock and Reset and HBUS UVC package files. The packages can be found in the `sv` directory of each UVC.

   b. Set the HBUS, Clock and Reset and Channel UVC virtual interfaces to the correct interface. (*Hint*: The UVC header files contain `typedefs` for each interface.)

      Use wildcards in the pathname to update all UVC components with a single statement.

      Use an absolute hierarchical pathname for the value to select the correct interface instance from the `hw_top` module.

Running Base Test

8. For **every** UVC (YAPP, Clock and reset, HBUS and Channel) add the following to your `run.f`.
   - An `incdir` reference to the UVC `sv` directory
   - UVC package filename.
   - UVC interface filename.

9. Run a simulation with `base_test` only. Check the topology report carefully to make sure all of your UVCs are instantiated and configured correctly. Copy the topology report into a new file for future reference.

Test Library

10. Add a new test class, `simple_test`, in **`router_test_lib.sv`** as follows (copy from existing tests). Sequencer pathnames can be read from the topology report.

    a. Set the YAPP UVC to create short YAPP packets with a `set_type_override`.

    b. Set the default sequence of the YAPP UVC to `yapp_012_seq`.

    c. Set the default sequence of each Channel UVC to `channel_rx_resp_seq`.
    Hint: you can set all three Channel UVCs with a single statement.

    d. Set the default sequence of the Clock and Reset UVC to `clk10_rst5_seq`.

    e. Do not define a default sequence for the HBUS UVC.

f. (Optional) Now might be a good time to clean up the test library and remove the older tests. Delete or comment out all the other test classes besides `base_test` and `simple_test`.

Running Simple Test

11. Run a simulation in GUI mode using `simple_test` and verify as follows:

    a. Add YAPP UVC monitor transactions to the waveform viewer.

    b. Add all three Channel UVC monitor transactions to the waveform viewer.

    c. Use the transactions to confirm packets are passed correctly through the router and collected at the right channel.

Further Integration Testing (Optional)

12. Write a new YAPP sequence in the `yapp/sv/yapp_tx_seqs.sv` file to generate packets for **all** four channels (including the illegal address 3). The packets should have incrementing payload sizes from 1 to 22 and parity distribution of 20% bad parity (88 packets in total).

    Hint: You could create packets using nested loops for address and payload.

13. Create a new test, `test_uvc_integration`, in the `router_test_lib.sv` file to perform the following:

    a. Set the `run_phase` default sequence of the YAPP UVC to the sequence created above.

    b. Set the `run_phase` default sequence of the HBUS UVC to set up the router with register field `maxpktsize` = 20 and enable the router (register field `router_en` = 1).

    Hint: There is a sequence defined for this in the HBUS master sequences `hbus_master_seqs.sv`.

    Hint: The hierarchical path name for the HBUS configuration setting can be read from the topology report.

14. Run a simulation and check the results to see that the three channels are properly addressed, that there is an error signal when parity is wrong, and that packets are dropped if bigger than `maxpktsize` or have illegal addresses.

Lab 8 Writing Multichannel Sequences and System-Level Tests

Objective: To build and connect multichannel sequences to your testbench.

For this lab, you will build and connect a multichannel sequencer for the router, and create multichannel sequences to coordinate the activity of the three router UVCs.

1. **First** – copy your files from `lab07_integ/tb` into `lab08_mcseq/tb`.

   Work in the `lab08_mcseq/tb` directory.

2. Create the multichannel sequencer component **`router_mcsequencer.sv`**.

   a. Add a component macro and constructor.

   b. Add the references for the HBUS and YAPP UVC sequencer classes.

   The Channel UVCs continuously execute a single response sequence, so they do not need to be controlled by the multichannel sequencer.

   The Clock and Reset UVC could be controlled by the multichannel sequencer if, for example, we wanted to initiate reset during packet transmission. However, for simplicity we'll leave Clock and Reset out of the multichannel sequencer.

3. Create a multichannel sequence library file, **`router_mcseqs_lib.sv`** and define a single multichannel sequence, `router_simple_mcseq`, as follows:

   a. Add an object macro and constructor.

   b. Add a `` `uvm_declare_p_sequencer `` macro to access the multichannel sequencer references.

   c. Using the sequences defined in the YAPP and HBUS UVC sequence libraries, create a multichannel sequence to:

   - Raise an objection on `starting_phase`.
   - Set the router to accept small packets (payload length < 21) and enable it.
   - Read the router `MAXPKTSIZE` register to make sure it has been correctly set.
   - Send six consecutive YAPP packets to addresses 0, 1, 2 using `yapp_012_seq`.
   - Set the router to accept large packets (payload length < 64).
   - Read the router `MAXPKTSIZE` register to make sure it has been correctly set.
   - Send a random sequence of six YAPP packets.
   - Drop the objection on `starting_phase`.

   There are pre-defined sequences in the UVC libraries for all the above operations.

4. Modify the **`router_tb.sv`** testbench to instantiate, build, and connect the multichannel sequencer.

   Hint: Examine a topology report to find the reference for the HBUS sequencer. Remember the connections for the multichannel sequencer references are hierarchical pathnames, not configuration instance name strings, therefore you cannot use wildcard characters in the connection.

5. Create a new test in **`router_test_lib.sv`** to achieve the following:

   a. Set a type override for short packets only.

   b. Set the default sequence of all output channel sequencers to `channel_rx_resp_seq` (copy from a previous test).

   c. Set the default sequence of the Clock and Reset sequencer to `clk10_rst5_seq` (copy from a previous test).

   d. Set the default sequence of the multichannel sequencer to the `router_simple_mcseq` sequence declared above.

   e. Do **not** set a default sequence for the YAPP or HBUS sequencer. Control is now solely from the multichannel sequencer.

6. Add `include` statements to your top module to reference the new files.

   Make sure the includes are in the correct order, for example the multichannel sequencer must be included before the testbench file, as the testbench creates an instance of the multichannel sequencer.

7. Run a test and check your results. If you open the simulator log file in an editor, you should be able to track packets through the router and see the HBUS read and write transactions.

   If necessary, you can insert extra delays between the YAPP and HBUS sequences in the multichannel sequence to clearly separate transactions on the different interfaces.

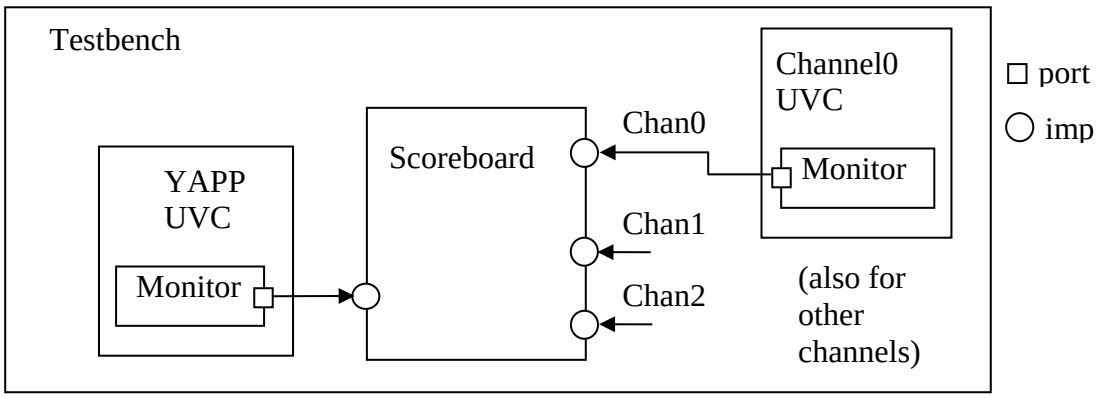
Lab 9A Creating a Scoreboard Using TLM

Objective: To build a scoreboard using TLM imp connectors.

For this lab, you will build and connect a scoreboard for the router, and create TLM analysis port connections to hook up the scoreboard to the UVCs.

The router Module UVC is a complex design, so this lab has been deliberately broken down into separate steps to build the UVC progressively.

The first step is to implement the scoreboard component itself and connect it up to the YAPP and Channel UVCs. For this part of the lab we assume all packets are sent to legal addresses with legal payload length, i.e., the router does **not** drop any packets.



1. Copy your files from `lab08_mcseq/tb` into `lab09_sba/tb`.

   Work in the `lab09_sba/tb` directory.

2. A TLM analysis port has already been implemented in the Channel UVC monitor for collected YAPP packets. This port is named `item_collected_port`. Check the Channel monitor in the file `channel/sv/channel_rx_monitor.sv` to make sure you understand how this port is used.

3. Modify `yapp/sv/yapp_tx_monitor.sv` to create an analysis port instance.

   a. Declare an analysis `port` object, parameterized to the correct type.

   b. Construct the analysis port in the monitor constructor.

   c. Call the port `write()` at the appropriate point.

4. In the `lab09_sba/sv` directory, create the scoreboard, **`router_scoreboard.sv`**.

   a. Extend from `uvm_scoreboard` and add a component utility macro and a constructor.

   b. As the YAPP and Channel UVCs use different packet types, you will need a custom comparison function to compare `yapp_packet` and `channel_packet` packets.

      You can use either simple Verilog comparison operators or the `uvm_comparer` class (see slides and reference material for details).

      A simple comparison operator is provided to you in the file `packet_compare.sv`, copy this into your scoreboard.

   c. Define four analysis `imp` objects (for the YAPP and three Channels) using `` `uvm_analysis_imp_decl `` macros and `uvm_analysis_imp_*` objects.

   d. Create analysis `imp` instances in the scoreboard constructor.

   e. Use the YAPP `write()` implementation to **clone** the packet and then push the packet to a queue. *Hint*: Use a queue for each address.

   f. Use Channel `write()` implementations to pop packets from the appropriate queue and compare them to the channel packets, using your custom comparer function.

   g. Add counters for the number of packets received, wrong packets (compare failed) and matched packets (compare passed).

   h. Add a `report_phase()` method to print the number of packets received, wrong packets, matched packets and number of packets left in the queues at the end of simulation.

5. In the `tb` directory, update the `tb_top` module to include the router scoreboard.

6. In the `tb` directory, modify the `router_tb.sv` as follows:

   a. Declare and build the scoreboard.

   b. Make the TLM connections between YAPP, Channel, and scoreboard.

   c. Use a test which generates a good number of legal YAPP packets, i.e. short packets with legal addresses, so that no packets are dropped by the router. We could use the Lab 8 multichannel sequence test or if you did not complete that lab, you could modify the Lab 5 exhaustive sequence test to add the Channel and Clock and Reset sequences.

   d. Check that the simulation results are correct, and debug as required.

7. Once you are happy that the scoreboard is working, check that it correctly reports mismatched packets. You can achieve this by commenting out the short YAPP packet type override in your test class which will allow the YAPP UVC to send oversized packets which will be dropped by the router and create mismatches in the scoreboard.

   Make sure that your HBUS sequencer is executing `hbus_small_packet_seq` to set the `MAXPKTSIZE` register to 20.

   Run a simulation with the type override removed and check the log file for the following:

   - Your YAPP UVC is generating packets of type `yapp_packet`.
   - The RTL router code generates `ROUTER DROPS PACKET` messages.
   - Your custom comparer function generates `uvm_error` messages when the comparison fails.
   - Packets are left in the scoreboard queues at the end of simulation.
   - Your scoreboard reports the number of received, mismatched and matched packets, as well as the number of packets left in the queues. Check the number of packets add up!

8. (Optional) Write your own custom comparer function which uses `uvm_comparer` methods instead of Verilog comparisons. Test your new implementation in simulation.

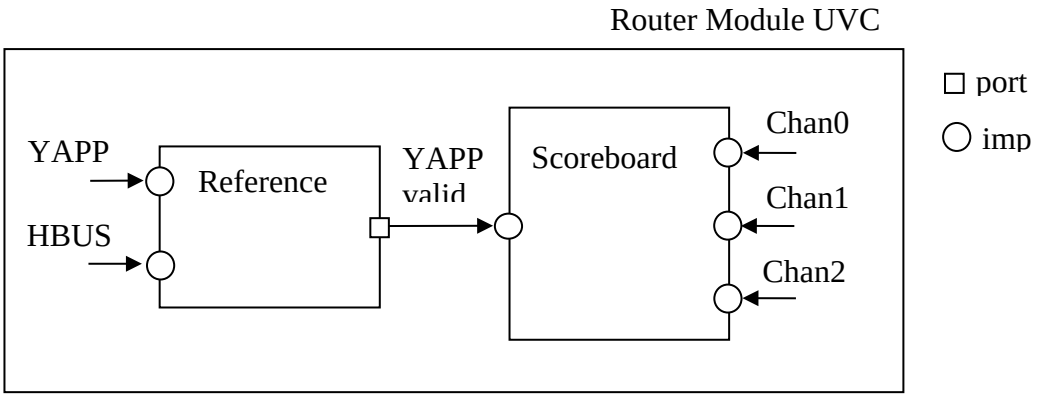
Lab 9B Router Module UVC

Objective: To create a module UVC for the router using the scoreboard.

In reality, the scoreboard will only be one part of a larger router module UVC. For example, the router UVC may also contain reference models and coverage. All these components will be enclosed in an `env` class.

In our example, we need to know the `maxpktsize` and `router_en` register field settings so we know which packets are dropped. We can implement this in a separate router reference model component.

The router reference model connects to the YAPP and HBUS UVC analysis ports and selectively passes on YAPP input packets to the scoreboard depending on the register settings. An `env` wrapper will instantiate both reference and scoreboard into a single router module UVC, as shown.



1. We will be working with the files from `lab09_sba`. *Copy* the directories from `lab09_sba` into `lab09_sbb`. Work in the `lab09_sbb` directory

2. A TLM analysis port has already been implemented in the HBUS UVC monitor.

   Note that the HBUS UVC has a common monitor, `hbus_monitor.sv`, for both the master and slave agents. The HBUS monitor analysis port for collected `hbus_transaction`'s is named `item_collected_port`.

   Check this to make sure you understand how it is written.

3. Create the router reference, **`router_reference.sv`**, in the `sv` directory.

   a. Extend from `uvm_component`.

b. Define two analysis `imp` objects for the YAPP and HBUS monitor analysis ports, using `` `uvm_analysis_imp_decl `` macros and `uvm_analysis_imp_*` objects. (Copy declarations from your scoreboard.) These are for input data to the reference.

c. Define one analysis port object for the valid YAPP packets. This is for output data to the scoreboard.

d. Define variables to mirror the `maxpktsize` and `router_en` register fields of the router and update these in the HBUS `write()` implementation.

e. In your YAPP `write()` implementation, forward the YAPP packets onto the scoreboard *only* if the packet is valid (router enabled; `maxpktsize` not exceeded; address valid). Keep a separate count of invalid packets dropped due to size, enable and address violations.

4. Create the router module environment, **`router_module_env.sv`**, in the `sv` directory.

   a. Declare and build the scoreboard and router reference components.

   b. Connect the “valid YAPP” analysis port of the reference model to the YAPP analysis imp of the scoreboard model.

5. In the `tb` directory, modify the `router_tb.sv`.

   a. Replace the scoreboard declaration and build with the router module.

   b. Modify the TLM connections for the YAPP and Channel analysis ports to allow for the `router_env` layer.

   c. Add a connection for the HBUS analysis port.

6. Create a `router_module.sv` package which includes the router module environment, reference and scoreboard files. Import this package into your UVM top module.

   a. Use the same multichannel sequences to test your scoreboard and system monitor implementation.

   b. Check the simulation results are correct and the scoreboard and monitor report the right number of packets.

7. The router module can now be used as a standalone UVC. Copy your router module UVC files to the `router` directory.

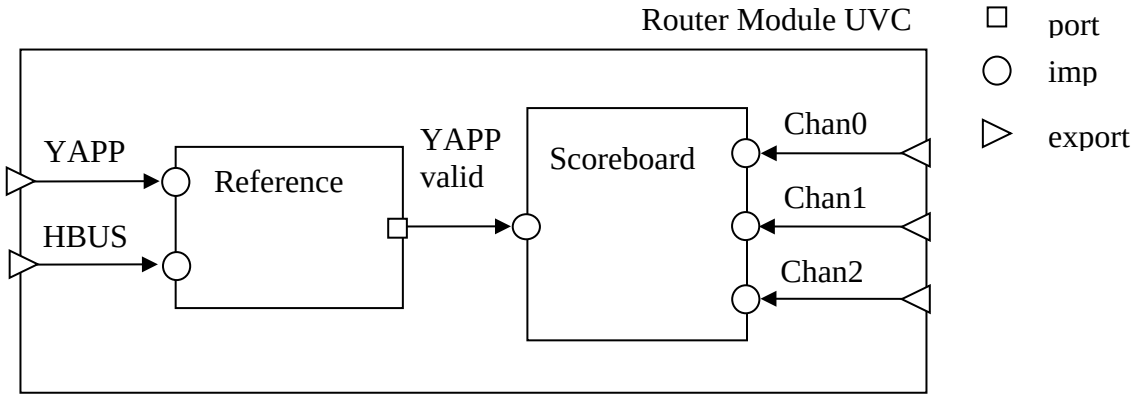
Lab 9C Using TLM Export Connectors (Optional)

Objective: To use export TLM connections with the router module UVC.

You must complete Labs 9A and 9B before starting this lab.

A module UVC does not have such a clearly defined architecture as an interface UVC. With an interface UVC, TLM analysis `port` objects are always defined in the monitor components. We know where to look in order to find these. With a module UVC, it can be more difficult to find declarations for the TLM analysis `imp` objects.

One technique is to extract all the TLM objects used in the module UVC to the top level environment. These top-level objects are then connected internally to the correct sub-component.



1. Modify your router module UVC to present the external TLM connections as top-level objects of the router module `env`. Note the following:
   - You will need to declare `export` objects in the Router Module environment for the `imp` objects of the monitor and scoreboard.
   - You must connect all the `export` objects to `imp` objects in the Router Module `connect` method.
   - The internal YAPP valid connection between reference and scoreboard does not need to be routed to the Router Module environment.
   - The testbench connections to the Router Module UVC will need to be modified.

2. Test your changes in simulation.

End of Lab

Lab 9D Using TLM Analysis FIFOs (Optional)

Objective: To build a scoreboard using TLM Analysis FIFOs.

This is an optional lab that you may like to try if your company uses TLM Analysis FIFO components, or you wish to explore alternative options for implementing scoreboards.

We recommend you complete Lab 9A at least (and preferably Lab 9B also) before attempting this lab. A full description for the scoreboard functionality is defined in Lab 9A and Lab 9B. Please refer to these labs for details.

The following guidance is intended to help you create a scoreboard implemented with TLM Analysis FIFOs:

1. Interface UVCs. The interface UVCs do not change. The UVC monitor analysis ports can be connected to either analysis FIFOs or analysis `imp` connectors. However, analysis FIFOs do not perform any cloning on input transactions. Therefore, you will need to check that the UVC monitors collect every transaction into a different instance to avoid overwriting data in the FIFOs.

2. Create a new scoreboard file with the following additions:

   a. Instantiate analysis FIFOs for the YAPP, HBUS and all three Channel connections.

   b. Instantiate `get` port connectors for each of the FIFO outputs and connect these to the `get_peek_export` connectors of the FIFO instantiations.

   c. In a run-phase task, check the packets as follows:

      - Use a blocking `get` call directly from the `get_peek_export` connector of the YAPP analysis FIFO instance to read input YAPP packets.
      - Discard the packet if the router is not enabled.
      - Discard the packet if it is not legal (length and address).
      - If the packet is valid, use a `get` call to read from the `get_peek_export` of the appropriate Channel analysis FIFO depending on the packet address.
      - Compare the packets and update any status counters.
      - In a concurrent forked method, read the HBUS analysis FIFO and on write operations, update local variables for maximum packet side and router enabled.

   d. In a check phase method, update any status counters and check the FIFOs are empty.

   e. In a report phase method, write out the status counters.

Lab 10 Creating a Simple Functional Coverage Model (Optional)

Objective: To understand where to implement cover groups in UVM architecture.

To achieve this objective, you need to create a coverage model for the input packet traffic to collect the following info:

- REQ1: Ensure all lengths of packets are sent into the router. Create buckets to detect minimum, maximum short, medium and large packets.
- REQ2: Ensure all addresses received a packet, including the illegal address.
- REQ3: Ensure all size packets were sent to all legal addresses with parity errors.

1. Create a covergroup in the **`yapp_tx_monitor.sv`**.

   Remember the syntax for a covergroup instantiated inside a *class* is different than one instantiated in a module. In a class, a covergroup instance is created by calling `new()` on the covergroup *name*. A separate covergroup variable is *not* required:

   ```
   function new (...);
     super.new(...);

     ...
     covergroup_name = new();
   endfunction: new;
   ```

2. Create a coverpoint for REQ1 to sample the length field and create bins to reflect the following ranges:

   ```
   MIN = 1
   MAX = 63
   SMALL in [2..10]
   MEDIUM in [11..40]
   LARGE in [41..62]
   ```

3. Sample coverage manually in the `collect_packets()` task, after the packet has been collected.

4. Run a simulation in GUI mode, using the following simulator coverage option:

   ```
   -coverage U
   ```

5. In the GUI, display coverage in Incisive Metrics Centre (IMC) by selecting

   ```
   Windows -> Tools -> Coverage
   ```

6. Display the coverage as follows:

   a. In the `Verification Hierarchy` pane, navigate down the Instances hierarchy under `uvm_pkg` to the YAPP monitor and select the instance.

   b. Right-click and select `Cover Group Analysis`.

   c. Select the `Items` of the covergroup to see coverage.

7. Create a coverpoint for REQ2 and REQ3. Create a coverpoint inside the covergroup created in step 1 for sampling address for REQ2 as follows:

   a. Create a legal address bin to verify that all addresses were sampled.

      If address 2 wasn't generated, then it needs to be reflected in this coverpoint.

   b. Create an illegal address bin that reflects how many packets were sent to address 3.

8. Create a cross inside the covergroup created in step 1 for coding REQ3 by creating a cross with the appropriate fields.

9. Run a simulation and analyze coverage results.

10. Modify your stimulus to achieve coverage of all requirements.

11. Run multiple simulations with random svseed and analyze coverage results.

Lab 11 Register Modeling in UVM

Objective: To verify the router register behavior using a UVM Register Model.

For reference, these are the registers and memory blocks of the YAPP Router design.

YAPP Router Registers

address	register	reset	field	field name	policy	description
0x1000	ctrl_reg	0x3f	5:0	maxpktsize	RW	Maximum packet length
			7:6		RW	Unused
0x1001	en_reg	0x01	0	router_en	RW	Router enable
			1	parity_err_cnt_en	RW	Parity error count enable
			2	oversized_pkt_cnt_en	RW	Oversized packet count enable
			3	[reserved]	RW	Not implemented
			4	addr0_cnt_en	RW	Address 0 packet count enable
			5	addr1_cnt_en	RW	Address 1 packet count enable
			6	addr2_cnt_en	RW	Address 2 packet count enable
			7	addr3_cnt_en	RW	Address 3 packet count enable
0x1004	parity_err_cnt_reg	0x00	7:0		RO	Parity error count
0x1005	oversized_pkt_cnt_reg	0x00	7:0		RO	Oversized packet count
0x1006	addr3_cnt_reg	0x00	7:0		RO	Address 2 packet count
0x1009	addr0_cnt_reg	0x00	7:0		RO	Address 0 packet count
0x100a	addr1_cnt_reg	0x00	7:0		RO	Address 1 packet count
0x100b	addr2_cnt_reg	0x00	7:0		RO	Address 2 packet count
0x100d	mem_size_reg	0x00	7:0		RO	Length of last packet

YAPP Router Memory Blocks

Start Address	Name	Size	Policy	Description
0x1010	yapp_pkt_mem	[0:63]	RO	Stores the last packet received.
0x1100	yapp_mem	[0:255]	RW	“Scratch” memory

Lab11A Generation

Objective: Create a UVM register reference model from an XML description.

In this lab, 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 **`lab11a_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? _____

Lab11B Integration

Objective: Integrate the register model into your UVM testbench.

In this lab, 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 `lab07_integ` onwards can be used. Copy your selected lab into the **`lab11b_rm_integ`** directory.

2. Copy the following register model files from `lab11a_rm_gen/reg_verifier_dir/uvmreg` into `lab11b_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 (Optional)

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
    ```

Lab11C Simulation

Objective: Create User-Defined Register Verification Stimulus.

In this lab, 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 **`lab11b_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. Simulate `reg_function_test` and check for correct behavior.

Automatic Checking on Read (optional)

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 (Optional)

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..