SystemVerilog® Advanced Verification Using UVM

Engineer Explorer Series

Version 1.1

Lab Manual **February 2, 2012**

cādence®

Cadence Design Systems, Inc., 2655 Seely Avenue, San Jose, CA 95134, USA

Cadence Trademarks

Other Trademarks

Confidentiality Notice

Table of Contents

SystemVerilog Advanced Verification Using UVM

Introduction to the Labs

This section gives a high-level overview of the labs for the SystemVerilog Advanced Verification Using UVM course.

Software Releases

These exercises and code examples have been tested on the following releases:

- Incisive® Unified Simulator release (9.2 or later)
- Universal Verification Methodology (UVM) release (UVM 1.0 or later)

Lab Database Files

- In your top-level module, you should import the UVM package and include the UVM macros, as show below:

```
module top;
  // import the UVM library
  import uvm_pkg::*;
  // include the UVM macros
  `include “uvm_macros.svh”
  . . .
endmodule : top
```

- Your reusable files are located in the `<uvc>/sv/` directory.
- Non-reusable files are located in the `<uvc>/examples` or `<uvc>/tb` directory. Simulations are also run in this directory.

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.

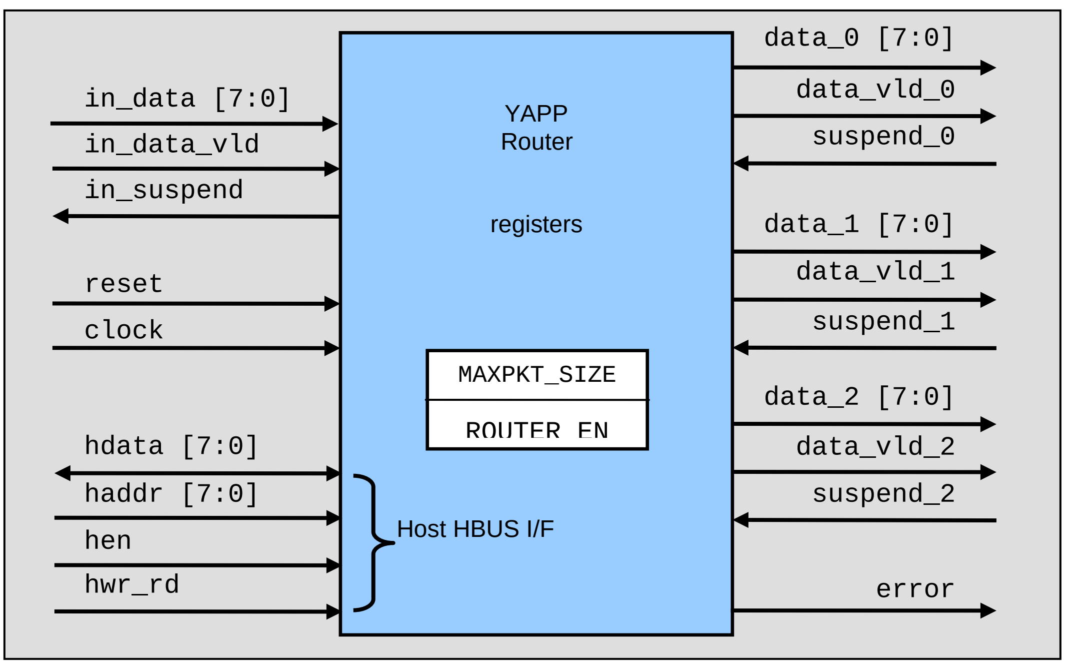
The project builds the environment from scratch so you will experience the full process

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

Packet Router Description

The packet 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 a host interface for programming registers that are described in the next section.

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



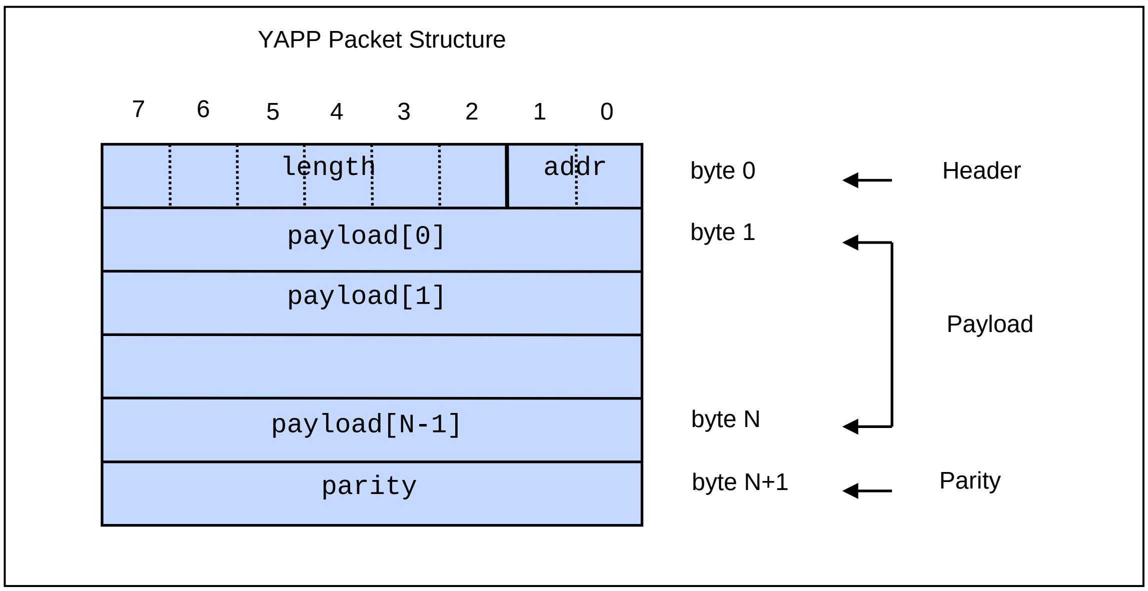
Packet Data Specification

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.

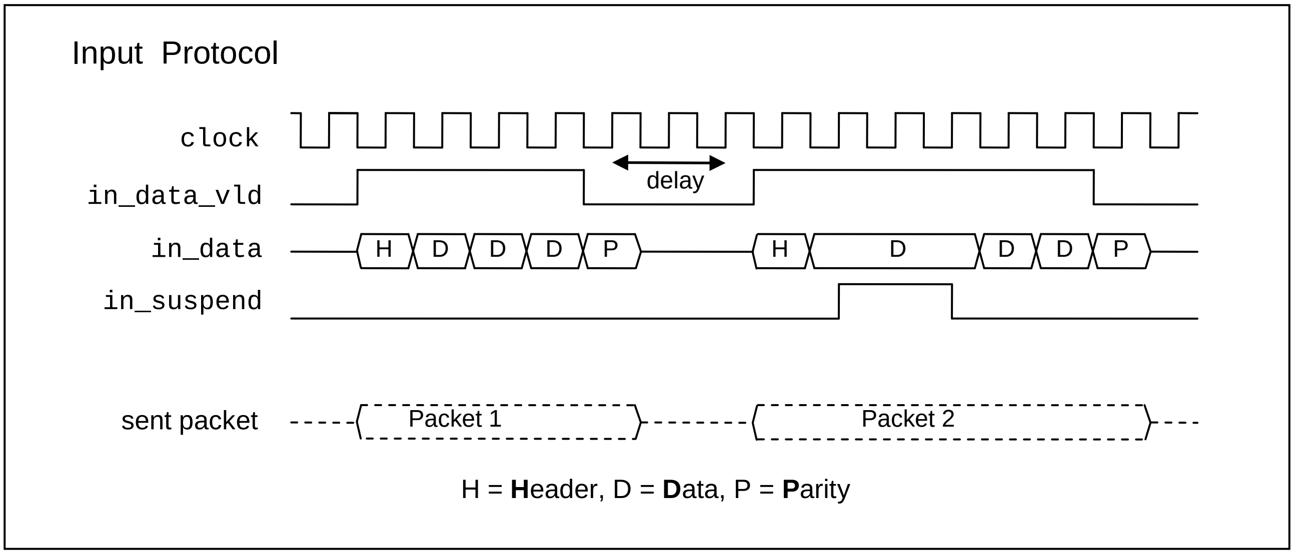
The parity should be a byte of even, bitwise parity, calculated over the header and payload bytes of the packet.



Input Port Protocol

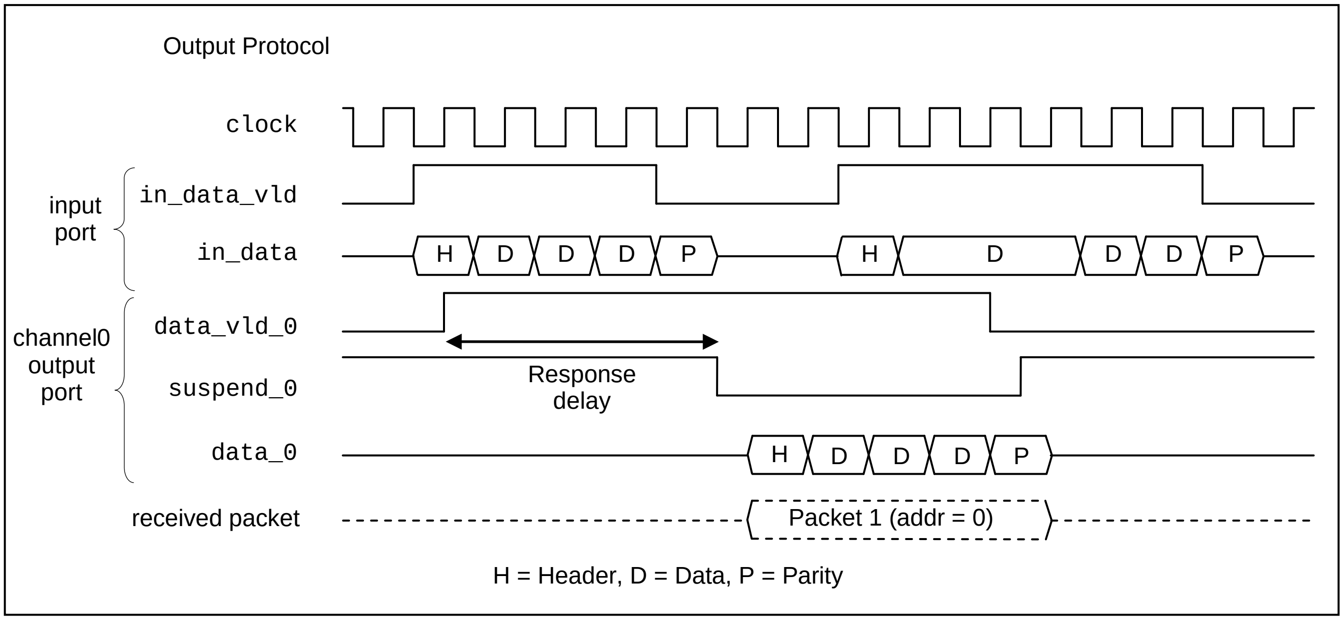
All input signals are active high and are to be driven on the **falling** edge of the clock. The `in_data_vld` signal must be asserted on the same clock when the first byte of a packet (the header byte), is driven onto the `in_data` bus. As the header byte contains the address, this tells the router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



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.



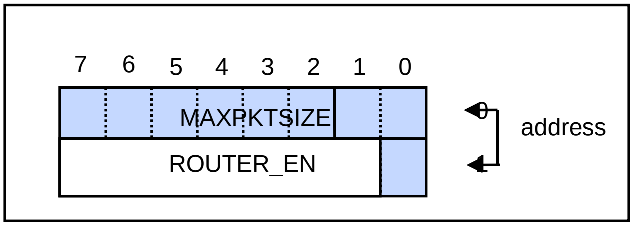
Packet Router DUT (Design Under Test) Registers

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

- MAXPKTSIZE – 8 bits – address 0   R/W - Reset value of ‘h3F
- ROUTER_EN - 1 bit   - address 1   R/W - Reset value of 1

If the input packet length is greater than the value of the MAXPKTSIZE register, the router drops the entire packet and the error flag is raised.

The ROUTER_EN register provides control of disabling the routing feature. Enabling or disabling the router during packet transmission will yield to unpredictable behavior.



Host Interface Port Protocol (HBUS)

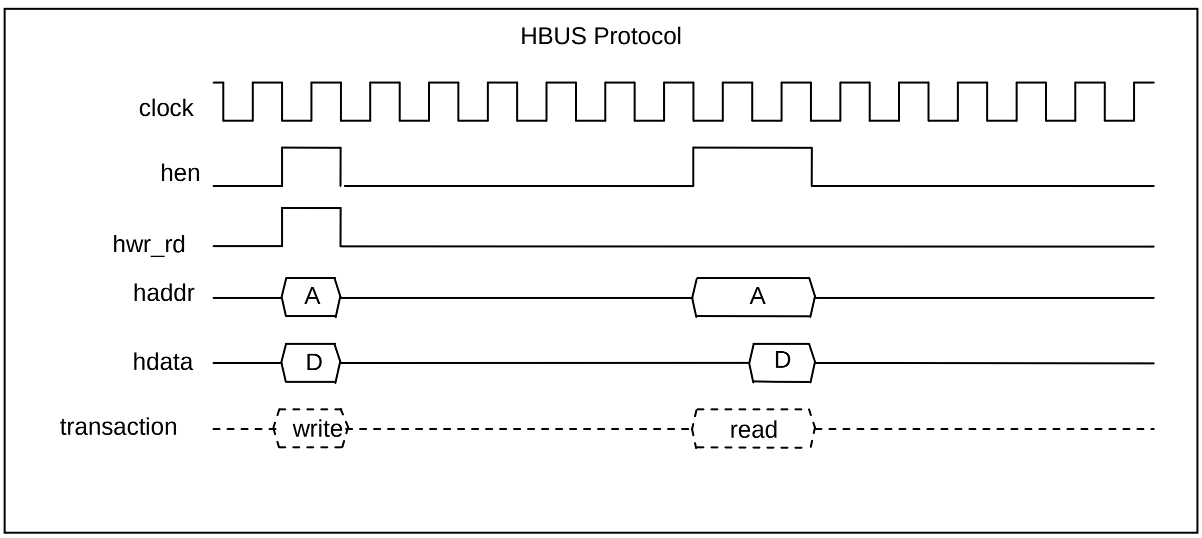
All input signals are active high and are to be driven on the **falling** edge of the clock. The host port provides synchronous read/write access to program the router.

A WRITE operation takes one clock cycle as follows:

- `hwr_rd` and `hen` must be 1. Data on `hdata` is then clocked on next rising clock edge in to 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 driver low after cycle 2 ends. This will cause the DUT to tri-state the `hdata` bus.



Before You Begin

Finding Your Files

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

`uvm_training_xx/uvm`	Top-level directory for the training
`labxx/`	Lab directories where you will be working
`hbus`	HBUS UVC and associated files
`channel`	Output channel UVC and associated files
`yapp`	YAPP input port UVC and associated files
`router`	Router module UVC and associated files
`router_rtl`	RTL Files for the YAPP Router DUT
`test_install`	Simple example to check your UVM installation
`uvm1-1_cdns`	UVM 1.1 library files with Cadence additions

Accessing UVM

UVM 1.1 is supported by IUS versions IES 10.2s11 or later *and* IES 9.2s35 or later.

UVM 1.1 may not be provided with the simulator, depending on the release version. Your instructor will tell you whether UVM is provided with your simulator release.

For simulator releases **not** including UVM, there are three options for accessing the UVM library:

- Use the library provided with your IES simulator installation.
- Use the library provided with the lab files.
- Download the library from `www.uvmworld.org`.

The first or second options are the best solution for IUS users, as the library contains extra Cadence 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. The install path contains the `src`, `docs` and `examples` directories: e.g. for the UVM library provided with the lab files

```
setenv UVMHOME <lab path>/uvm_training_1.1/uvm/uvm1-1_cdns
```

.

Testing Your UVM Installation

2. Change directory to `test_install`.

3. Run the test by executing the following command:

   ```
   irun –f run.f
   ```

4. In the test output, check the version number of the UVM library. You should see:

   ```
   -------------------------------------------------------
   UVM-1.1
   ...
   ```

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

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

Accessing Help Files

If you are using the UVM library provided with the lab files, you may wish to bookmark the following file in a web browser to give convenient access to the UVM HTML documentation

```
$UVMHOME/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/sv` directory.

Creating a Data Item

1. Review the YAPP packet data specification on pages 1-4 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 address, length, payload and parity properties to match the specification.

   c. Add a ``` `uvm_object_utils ``` macro block containing ``` `uvm_field_* ``` macros for every property.

   d. Add an UVM data constructor `new()`.

2. Add support for randomization of the packet:

   a. Declare the `length`, `address` 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_t`, 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/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:

   d. 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
   ```

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

   ```
   % irun –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.

End of Lab

Lab 2 Creating a Simple UVC

Objective: To create a simple UVM Verification Component (UVC, aka env) and print the topology.

You will be creating a simple driver, sequencer, monitor, agent and env for the YAPP input port of the router. You will focus on the transmit (TX) agent for this lab.

Work in the `lab02/sv` directory. Use the training slides for suggestions on implementing these components.

Creating the UVC

1. **First** - *copy* `yapp_packet.sv` and `yapp_pkg.sv` from `lab01/sv` into the `lab02/sv` directory.

   Also copy your `run.f` file from `lab01/tb` into the `lab02/tb` directory.

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 `` `uvm_component_utils `` macro and a UVM 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. Use an `` `uvm_info `` macro with a verbosity of `UVM_LOW`. Use the following code in the *string* portion of the macro (where `pkt` is the name of the input argument to the `send_to_dut()` task):

   ```
   $sformatf("Packet is \n%s", pkt.sprint())
   ```

   Note: `sprint()` is a sub-method of `print()`, which creates a print string, but does not write it to the output. We use `sprint()` in a report macro, rather than `print()` directly, to allow us to control the printing of the packet with verbosity settings.

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 `` `uvm_component_utils `` macro and a UVM 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 `` `uvm_component_utils `` macro and a UVM component constructor.

    c. Add a `run_phase()` task which prints an `uvm_info` message that 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 `` `uvm_component_utils `` macro and a UVM 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 will contain an `is_active` flag to control whether the agent is active or passive. Initialize this to 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 to construct the component hierarchy. Call `super.build_phase(phase)`, then construct the `monitor` which is always present. You will then conditionally construct the `driver` and `sequencer`, only if the `is_active` flag is set to `UVM_ACTIVE`.

    f. 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 environment (`yapp_env`) in the file **`yapp_env.sv`**.

    a. Extend from `uvm_env`. Remember environments do not have type parameters.

    b. Add a `` `uvm_component_utils `` macro and a UVM component constructor.

    c. Add a handle named `agent` of the `yapp_tx_agent` class.

    d. Construct the agent in a `build_phase()` method. Remember to call `super.build_phase(phase)` first.

    e. Implement a `run_phase()` task which prints the UVC (just call `print()`) ..

7. Create the UVC header file, **`yapp.svh`**:

   Include all of the files you created for this lab, together with the `yapp_packet.sv` copied from Lab01, **and** the supplied file `yapp_tx_seqs.sv`, 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"
   ```

   Don't forget to include the supplied file `yapp_tx_seqs.sv`. This file contains a sequence to help you verify your UVC. We will discuss sequences in detail later.

8. Edit the UVC package file, **`yapp_pkg.sv`**:

   Replace the include of the `yapp_packet.sv` file with an include for `yapp.svh`.

Checking the UVC Hierarchy

9. Move to the `lab02/tb` directory.

10. Edit the top-level module file: `top.sv` to check the UVC hierarchy:

    a. Import the YAPP package you created in step 8.

    b. Create a handle for the `yapp_env` component.

    c. In an `initial` block, construct the handle using the correct syntax.

    d. In a separate `initial` block, call a task `run_test()` without any arguments. This built-in method will run a default test (details described later).

11. Run a simulation using the following command:

    ```
    % irun –f run.f
    ```

    a. Look for the printed hierarchy. Does the hierarchy match your expectations?

b. Use the printed hierarchy to work out the full hierarchical pathname to your sequencer (e.g. `yapp.agent.sequencer`) and write it below.

   Sequencer pathname: ____

c. Use your printed hierarchy to find the value of the `yapp_tx_agent` `is_active` property.

   What is the value of the `is_active` variable when you printed the hierarchy?

Running a Simple Test

12. Your top-level module, `top.sv`, contains the following commented code:

```
uvm_config_wrapper::set(null, "<path>.run_phase",
                              "default_sequence",
                              yapp_5_packets::type_id::get());
```

This code sets the default sequence of the `run_phase` to the sequence `yapp_5_packets` defined in the `yapp_tx_seqs.sv` file.

a. Un-comment this code and move it into your `initial` block containing `run_test()`. Place the code *in front of* the `run_test()` call.

b. **Edit** the code to replace **`<path>`** with the hierarchical pathname to your sequencer as recorded above.

c. Check your initial block looks like this (although your pathname may be different):

```
initial begin
  uvm_config_wrapper::set(null, "yapp.tx_agent.sequencer.run_phase",
                                "default_sequence",
                                yapp_5_packets::type_id::get());
  run_test()
end
```

Note: In UVM 1.0 and later, a UVC will not create any stimulus without an explicit sequence. In this Lab we must use a configuration set call to execute a pre-supplied sequence which tells your UVC to generate YAPP packets.

Both sequences and configurations are covered in later modules of this course.

13. Run a simulation using the following command:

    ```
    % irun –f run.f
    ```

    Your UVC should now generate and print YAPP packets. Check the following:

    How many packets were generated?

    Is every field of the packet printed?

    Do you see good and bad parity packets?

14. Explore the verbosity options. Verbosity can be changed by adding the following option to the `irun` command line or compile file:

    ```
    +UVM_VERBOSITY=UVM_LOW
    ```

    If you have use the correct syntax and verbosity for `` `uvm_info ``, you will see different amounts of data printed when using different verbosity options. Try `UVM_NONE` and `UVM_FULL`. You should also note that the testbench is not re-compiled when you change verbosity.

15. Edit your `run.f` file to add the following compilation option:

    ```
    +SVSEED=random
    ```

    This sets a random value for the initial randomization seed of the simulation. Run the simulation again and you should see different packet data created for each run. The simulation reports the actual seed used for each simulation in the `irun.log` file.

End of Lab

Lab 3 Adding Test Classes and Exploring Phases

Objective: To use a test class to verify the simple YAPP UVC and to explore the built-in phases of *uvm_component*.

Create and work in `lab03/sv` and `lab03/tb` directory. For this lab, you will construct the YAPP UVC in a test class instead of in the top module.

1. **First** – *copy* your files from `lab02/` into `lab03/` , e.g. from the `uvm` directory, type:

   ```
   %  cp –r lab02/* lab03/
   ```

2. In the `lab03/sv/` directory, edit the **`yapp_env.sv`** file and **delete** the `run_phase()` task containing the `print()` method call.

3. In the `lab03/sv` directory, add a `start_of_simulation_phase()`method to your sequencer, driver, monitor, agent and environment components.

   The method should simply display a message indicating in which component the method is being called. (use `` `uvm_info `` with a verbosity of `UVM_HIGH`).

4. In the `lab03/tb` directory, create a test file, **`yapp_test_lib.sv`** as follows:

   a. Name the test class `base_test` and inherit from `uvm_test.`

   b. Add the `` `uvm_component_utils `` macro and a component constructor.

   c. Declare a handle for `yapp_env` inside the test.

   d. Add a `build_phase()` method containing `super.build_phase(phase)`.

   e. Construct the `yapp_env` instance in `build_phase`(). Add the constructor call after `super.build_phase(phase)`.

   f. From the `lab03/tb/top.sv` file, copy the `uvm_config_wrapper::set` code and paste it into the `build_phase()` method of `base_test` before `super.build_phase()`.

   g. Edit the `uvm_config_wrapper::set` code in `base_test` to replace `null` with `this`.

      When `set` is called from the `top` module, it must use `null`. When you move `set` into the test class, you must change `null` to `this,` which allows the simulator to resolve the sequencer hierarchical pathname from the test class scope.

h. Add an `end_of_elaboration_phase()` method to the test and use the `uvm_top.print_topology()` command to print the testbench hierarchy.

5. In the `lab03/tb` directory, modify the top-level module, `top.sv` as follows:

   a. Include `yapp_test_lib.sv` after the YAPP package import.

   b. **Delete** the `yapp_env` handle and constructor, as the environment is now created in the test class component.

   c. **Delete** the `uvm_config_wrapper::set` code, as the `set` call is now made in the test class.

Executing the Test

6. Run the test suite.

   a. Edit your `run.f` file to add the following simulation options:

   ```
   +UVM_TESTNAME=base_test
   +UVM_VERBOSITY=UVM_MEDIUM
   ```

   b. Run a simulation using the following command:

   ```
   irun –f run.f
   ```

7. Check the output from simulation as follows:

   a. Check the testbench hierarchy is correct.

   Does it match your expectations?

   Which test class is being executed?

   b. Check which `start_of_simulation_phase()` method was called first. Which is called last? Why? You will need to set the proper `+UVM_VERBOSITY` option to see the phase method messages.

8. (Optional) In the test library file `yapp_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`?

9. (Optional) Compile with the option `+UVM_TESTNAME=test2`.

You must edit the run file to change the UVM_TESTNAME option. Simply adding another UVM_TESTNAME option at the end of the command line will not over-ride the run file option.

Note that you can switch between base_test and test2 via the command-line option +UVM_TESTNAME, *without* re-compiling your test environment.

Check your printed hierarchy to make sure the correct test is being executed.

End of Lab

Lab 4 Using Factories

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

Create and work in `lab04/sv` and `lab04/tb` directories. For this lab, you will modify your existing files to use factory methods, and explore the benefits of configurations.

1. **First** – copy your YAPP files from `lab03/` into `lab04/`, e.g. from the `uvm` directory, type:

   ```
   %  cp –R lab03/* lab04/
   ```

   Note: You will be making many changes to files in this lab – so PLEASE *copy* the `sv` and `tb` directories from `lab03` to `lab04` to keep a working copy available!

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.

2. 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/yapp_test_lib.sv
   sv/yapp_env.sv
   sv/yapp_tx_agent.sv
   ```

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

Using Configurations and Overrides

4. In the `yapp_test_lib.sv` file, add a `check_phase()` phase method to `base_test` which contains the following call:

   ```
   check_config_usage();
   ```

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

5. 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 `tb/yapp_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.

6. Create a new configuration test in the file `yapp_test_lib.sv`.

a. Define a new test, `set_config_test` which extends from `base_test`.

b. In the `build_phase()` method, use the `set_config_int()` method to set the `is_active` bit to `UVM_PASSIVE` in the YAPP TX agent before building the `yapp_env` instance.

c. Make sure you are calling `uvm_top.print_topology()` from an `end_of_elaboration_phase()` method, either explicitly in the `set_config_test` class, or via inheritance from `base_test`.

d. 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.

e. You should get a configuration usage message reported.

Why do you get this?

How could you fix it?

End of Lab

Lab 5 Generating UVM Sequences

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

Create and work in `lab05/sv` and `lab05/tb` directories. 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/` into `lab05/`, e.g. from the `uvm` directory, type:

   ```
   %  cp –R lab04/* lab05/
   ```

   Note: You will be making many changes to files in this lab – so PLEASE *copy* the `sv` and `tb` directories from `lab04` to `lab05` to keep a working copy around!

2. In the `lab05/sv` directory, edit the sequences file, **`yapp_tx_seqs.sv`**, to add the sequences defined below in step 3 to 6 (and optionally steps 7 and 8).

   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 constructor and object utilities macro.

3. Create two new packet sequences with different constraints in the `body`.

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

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

4. Create a simple nested sequence.

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

5. Create a repeating address sequence.

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

6. 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).

7. (Optional) Create 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.

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

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

Running a Test Using a New Sequence

9. Create a new test in the file **`yapp_test_lib.sv`** from the `lab05/tb` directory:

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

   b. **Move** the `uvm_config_wrapper::set` code from `base_test` to the `build_phase()` method of `incr_payload_test`.

   c. Edit the `uvm_config_wrapper::set` to set the `run_phase` default sequence to `yapp_incr_payload_seq`.

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

   e. Run the test and verify the results. Setting verbosity to `UVM_FULL` will allow you to see which 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.

10. 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.

11. Create a new test in the file **`tb/yapp_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

12. 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.

    Why do you get violations?

    Answer:

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

    Answer:

    How could you fix these violations?

    Answer:

13. Run a simulation using the GUI. The simulation will now be stopped on a constraint violation and the constraints manager tool opened. This tool allows complex constraint violations to be more easily debugged.

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

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

End of Lab

Lab 6 Connecting to the DUT Using Virtual Interfaces

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

Work in the `lab06/sv` and `lab06/tb` directories. 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/` into `lab06/`.

   Note: You will be making many changes to files in this lab – so PLEASE *copy* the `sv` and `tb` directories from `lab05` to `lab06` to keep a working copy around!

2. Check the YAPP interface to the DUT, **`yapp_if.sv`**, which is supplied in the `lab6/sv` directory.

   Note that the interface has two input ports (`clock`, `reset`) and three DUT signals, `in_data`, `in_data_vld` and `in_suspend`.

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.svh`, before the include statements:

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

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

   a. You will to need to use a `collect_packets()` method to capture the packet data.

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

      Use the supplied code to update your monitor.

   b. Add a declaration for the virtual interface:

      ```
      virtual interface yapp_if vif;
      ```

   c. Add a `build_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. Passing this return value to an `assert` or `if` statement will help debug the virtual interface connection. For example:

      ```
      if (!yapp_vif_config::get(this,"","vif", vif))
        `uvm_fatal("NOVIF",{"vif not set for: ",get_full_name(),".vif"})
      ```

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

   a. You will need to use a `send_to_dut()` 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 monitor are provided in the file **`driver_example.sv`**. Check you understand the code.

      Use the supplied code to update your driver.

   b. Make sure you update the `get_and_drive()` method to pull down packets from the sequencer (see comments in code).

   c. Add a declaration for the virtual interface :

      ```
      virtual interface yapp_if vif;
      ```

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

Adding a Testbench

6. In the `lab06/tb` directory, create a top-level testbench, **`router_tb.sv`**.

   a. The `router_tb` extends from `uvm_component`. It contains a single instance of the `yapp_env`.

   b. The testbench requires an `` `uvm_component_utils `` macro, a constructor and `build_phase()`method. Create the `yapp_env` instance in `build_phase()`.

   c. Add the following configuration line to `build_phase()`method, before `super.build_phase()`, to enable transaction recording.

      ```
      set_config_int( "*", "recording_detail", 1);
      ```

7. Modify your tests to include the testbench layer, by editing **`yapp_test_lib.sv`**:

   a. Replace the `yapp_env` instance in `base_test` with a `router_tb` instance.

   b. Any hierarchical references to `yapp_env` need to include an additional reference to your `router_tb` handle, for example in configuration `set` pathnames.

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

      ```
      phase.phase_done.set_drain_time(this, 200ns);
      ```

The drain time will allow enough time for the packets to pass through the router design before the simulation ends.

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. **Rename** the test library file to **`router_test_lib.sv`**, as it will contain test classes for the entire router from this point on.

Initial Simulation Without the DUT

8. A top-level module, **`top_no_dut.sv`**, is provided for you in the `tb` directory. The module supplies the following functionality:
   - Declares the `clock`, `reset` and `in_suspend` signals; initializes them and generates the waveforms required.
   - Instantiates the `yapp_if` interface:

     ```
     yapp_if in0 (clock, reset);  // interface instance
     ```
   - Calls `run_test()` in an `initial` block to initiate the simulation.

   a. Use `set` to write the YAPP interface instance into the configuration database (Hint: Use your `typedef` from step 3). Use wildcards in the pathname to affect both monitor and driver with a single statement. Remember for a top module `set`, the context is `null`.

   b. Add an import for your YAPP UVC package.

   c. Add includes for `router_tb.sv` and `router_test_lib.sv`.

9. Update your `run.f` file:

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

10. Test your updates in simulation.

    a. Run a test to verify that things are working correctly. You may need the following `irun` default timescale option in your `run.f` file to avoid timescale errors:

    ```
    -timescale 1ns/100ps
    ```

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

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

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 your `top_no_dut.sv` module as follows:

    a. Instantiate the router module 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
    ```

    Hint –check the `yapp_router` module of the router code for the top level port list.

    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` via assign statements or port mapping. This will allow packets to pass through the device.

    d. Save your top module as `top_dut.sv`.

12. Run a test to make sure the router is correctly connected.. Remember to compile the `yapp_router.v` file from the `router_rtl` directory.

13. Run a simulation in GUI mode and use the SimVision waveform viewer to check packet data comes out on the right output channels.. If you have transaction recording in the monitor or driver, you can also view the transactions. Ask your trainer for guidance.

End of Lab

Lab 7 Integrating Multiple UVCs

Objective: To connect and configure the HBUS UVC and three output channel UVCs with the input port UVC and the *YAPP_router* design.

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

The HBUS UVC is provided. This will need to be connected to the HBUS port.

The Channel UVC is also provided. This is almost identical to the YAPP input UVC, except the Channel UVC must properly set/reset the `in_suspend` signal.

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

`hbus/sv`	HBUS UVC files
`channel/sv`	Channel UVC files
`yapp/sv`	YAPP input UVC (your files from `lab06`)
`router_rtl`	Router DUT
`lab07/tb`	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/`**`sv`** into `yapp/`**`sv`**.

   As the YAPP packet data item is used by both Channel and YAPP UVCs, you will need to make one change to your YAPP UVC.

   a. Create a package `yapp_pkt_pkg`. Cut the include for `yapp_packet.sv` in the file `yapp.svh` and paste the include into this package.

   b. Add an import for `yapp_pkt_pkg` at the top of `yapp.svh`.

2. We will still be working on the testbench, testclass, and top files. Copy these files from `lab06/`**`tb`** into `lab07/`**`tb`**.

   Work in the `lab07/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 `set_config_int` method to set the `has_rx` property of each Channel instance to `1`. The Channel UVC has both transmit and receive agents. For the router testing, we only need the receive agent.

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 `set_config_int` 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.

Test Library

5. Add a new test class, `simple_test`, in **`router_test_lib.sv`** as follows:

   a. Extend from `base_test` class.

   b. Use `set_type_override` and `uvm_config_wrapper::set` to create the following (copy from existing tests):

      - Set the YAPP UVC to create short YAPP packets.
      - Set the `run_phase` default sequence of the YAPP UVC to the `yapp_012_seq` sequence.
      - Set the `run_phase` default sequence of each Channel UVC to `channel_rx_resp_seq`. (Hint – the `docs` directory of the Channel UVC contains topology examples which can help in the configuration declaration).
      - Do not define a default sequence for the HBUS UVC.

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

Top Module

6. Update your top-level module to support the Channel and HBUS UVCs.

   a. Add an import for `yapp_pkt_pkg` before the YAPP UVC import statement.

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

c. Add interface instantiations for the `HBUS` and all three Channels.

   The `channel_if.sv` file can be found in the `channel/sv` directory. The `hbus_if.sv` file can be found in the `hbus/sv` directory.

d. 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** representation of this signal `hdata_w`, not the logic variable `hdata`.

   If you set the `suspend_0`, `suspend_1` and `suspend_2` router channel ports to `1'b0` via assign statements in the previous lab, remember to remove these statements.

e. Set the HBUS and Channel UVC virtual interfaces to the correct interface (Hint: the UVC header files contain `typedefs` for the channel and HBUS interfaces). Use wildcards in the pathname to update all UVC components with a single statement.

Running a Test

7. Run a simulation, remembering to compile the UVC packages and interfaces. Use the simulation output (`irun.log`) to help check the following:

   a. Sequencer settings are correct for all UVCs (use `UVM_FULL` verbosity).

   b. Configuration settings are correct for all UVC's (check the topology report carefully).

   c. Packets are passed correctly through the router and collected at the right channel.

Further Integration Testing (Optional)

8. Write a new YAPP sequence in the `yapp/sv/yapp_tx_seqs.sv` file to generate packets for **all** four channels. 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.

9. Create a new test, `test_uvc_integration`, in the `lab07/tb/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 in step 8.

   b. Set the `run_phase` default sequence of the HBUS UVC to set-up the router with `MAXPKTSIZE` = 20 and `ROUTER` enabled.

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.

10. 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.

End of Lab

Lab 8 Writing Multichannel Sequences and System-level Tests

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

For this lab, you will build and connect a virtual sequencer for the router, and create virtual sequences to co-ordinate the activity of the three router UVCs.

1. We will be working on the testbench, testclass and top files from lab07. Copy these files from `lab07/`**`tb`** into `lab08/`**`tb`**.

   Note: You will be making many changes to files in this lab, so PLEASE *copy* the `tb` directory from `lab07` to `lab08` to keep a working copy around!

   Work in the `lab08/tb` directory.

2. Create the virtual sequencer **`router_virtual_sequencer.sv`**.

   a. Add a component macro and constructor.

   b. Add the references for the HBUS and YAPP UVC sequencer classes. You could also add a reference to the Channel sequencer class, but since there is only one sequence for that sequencer, the handle can be omitted.

3. Create a virtual sequence file, **`router_virtual_seqs.sv`** and define a simple virtual sequence, `router_simple_vseq`, as follows:

   a. Add an object macro and constructor.

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

   c. Using the sequences defined in the YAPP and HBUS UVC sequence libraries (and optionally also the Channel), create a virtual 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 address 0, 1, 2, cycling the address.
      - 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`.

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

   Hint – examine a topology report to find the reference for the HBUS sequencer. Remember the connections for the virtual sequencer references are hierarchical references, not configuration instance names, therefore you cannot use wildcard characters in the connection.

5. Create a **`router_vtest_lib.sv`** test file to instantiate and build the virtual sequencer testbench.

   Tip: Copy or extend your test from the **`router_test_lib.sv`** created in lab07.

   Use configuration set methods, type overrides and the following checklist to correctly configure your testbench:

   a. Set a type override for short packets only.

   b. Set the default sequence of all output channel sequencers to `channel_rx_resp_seq`.

   c. Set the default sequence of the virtual sequencer to the `router_simple_vseq` sequence declared above.

   d. Do **not** set a default sequence for the YAPP or HBUS sequencer. Control is now solely from the virtual 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 virtual sequencer must be included before the testbench file, as the testbench creates an instance of the virtual sequencer.

7. Run a test and check your results. If you open the `irun.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 virtual sequence to clearly separate transactions on the different interfaces.

End of Lab

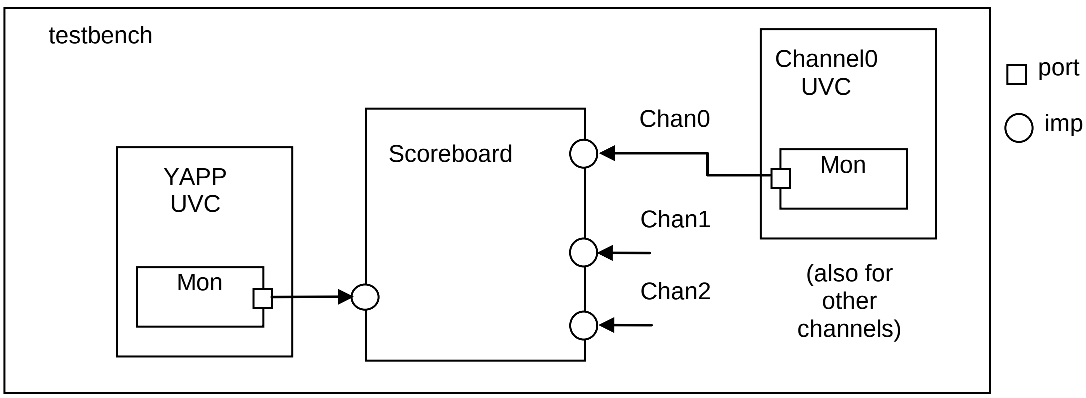
Lab 9A Creating a Scoreboard Using TLM

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.

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. We will be working with the testbench, testclass and top files from lab08. *Copy* these files from `lab08/`**`tb`** into `lab09a/`**`tb`**. Work in the `lab09a/tb` directory.

2. A TLM analysis port has already been implemented in the Channel UVC monitors. Check this to make sure you understand how it is written.

   Note that the Channel UVC has a common monitor, `channel_monitor.sv`, for both the TX and RX agents. The Channel monitor analysis port for collected YAPP packets is named `item_collected_port`.

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 `lab9a/sv` directory, create the scoreboard, **`router_scoreboard.sv`**.

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

   b. Add a constructor, and create analysis `imp` instances in the constructor.

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

   d. Use Channel `write()`implementations to pop packets from the appropriate queue and compare them to the channel packets.

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

   f. 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 `lab09a/tb` directory, update the top module to include the router scoreboard file.

6. In the `lab09a/tb` directory, modify the `router_tb.sv` as follows:

   a. Declare and build the scoreboard.

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

   c. Use the same multichannel sequences as in lab08 to test your scoreboard implementation, i.e. short packets with legal addresses, so that no packets are dropped by the router.

   d. Check that the simulation results are correct.

End of Lab

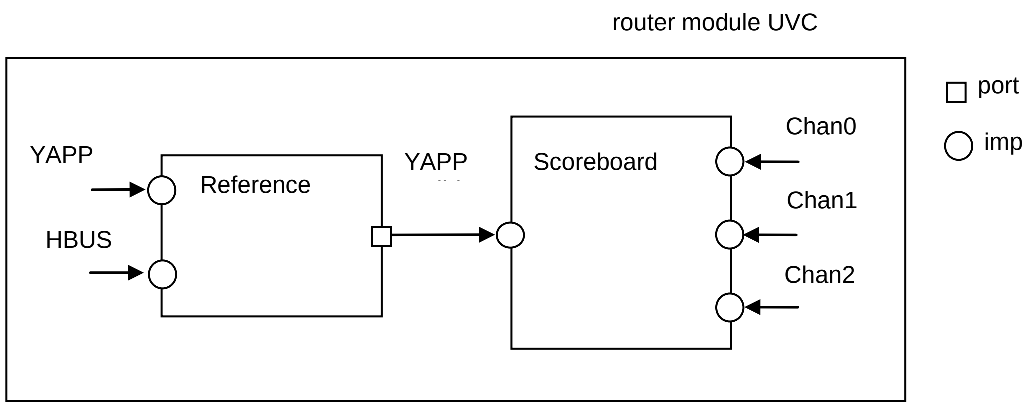
Lab 9B Router Module UVC (Optional)

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 `ENABLE` router settings so we know which packets are dropped. We can implement this in a separate router reference model component.

The router reference connects to the YAPP and HBUS UVC analysis ports and selectively passes on YAPP input packets to the scoreboard depending on the HBUS 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 lab09a. *Copy* these files from `lab09a` into `lab09b`. Work in the `lab09b` 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 `lab09b/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 `ENABLE` registers 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.

   Note in a real-world example, we would have to explore race conditions when the HBUS changes packet size during the processing of a packet. However there is a UVM_REG feature specifically for handling and verifying registers in a design. UVM_REG is covered in separate UVM training.

4. Create the router module environment, **`router_module_env.sv`**, in the `lab09b/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 file containing includes for the router module environment, reference and scoreboard files. Import this package into your 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.

End of Lab

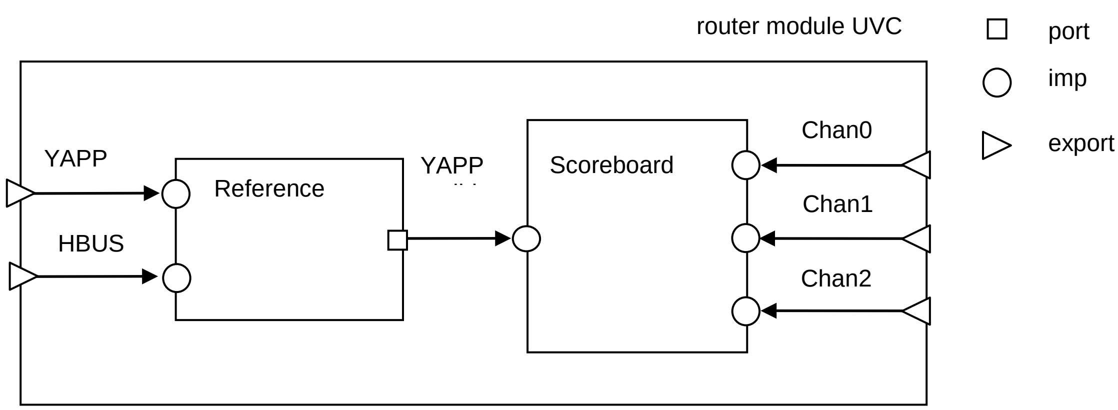
Lab 9C Further Work with TLM Connections (Optional)

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 monitor 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 10 Using TLM Analysis FIFOs (Optional)

Objective: To build a scoreboard using TLM Analysis FIFOs

This is an optional lab which 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 to complete Lab 9A at least (and preferably Lab 9B also) before attempting this lab. A full description for the scoreboard functionality is defined in Lab9A and Lab9B. 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 input, HBUS input and all three Channel outputs.

   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 get call to read from the YAPP analysis FIFO.
      - 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 appropriate Channel analysis FIFO.
      - 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 11 Writing Tests

Objective: To explore structuring tests and create a generation scheme with an UVM SystemVerilog verification environment.

To achieve this objective, use the yapp environment to create directed random tests.

Creating Virtual Sequences

1. Create a set of sequences that configures the device once, and sends packets with good and bad parity and bad size.

   Bad size is related to `max_packet_size`.

2. Change the `max_packet_size` on the fly and have the bad size sequence adjust dynamically.

Making a Virtual Sequences and Scoreboard

3. Send 20-30 packets to the device and disable the DUT by writing to the enable register.

4. Wait for a few cycles and enable the device.

5. Verify using the waveform viewer that the device is operating properly.

Writing a Test

6. Write a test in which the distribution of packet size is as follows:

   20: < (max_packet_size - 2)

   30: (max_packet_size -1)

   30: max_packet_size

   20: >max_packet_size

7. Generate packets of this distribution with all generated sequences.

Writing a Test (Another Method)

8. Write an API for writing simple tests, directed or random.

   For example, create a write and a read task for the host interface

   `write_hif(addr, data)`

   `read_hif(addr, output data)`

   Tip: Make this task invoke `` `uvm_do_with(...) `` inside a sequence.

9. From the `run_phase()` method of a test, call a series of `write_hif` and `read_hif` to test this simplified interface.

   Would this mechanism work for engineers who want to do directed testing or for designers that want a simple test writer interface?

End of Lab

Lab B Creating a Simple Functional Coverage Model

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 *dut*. Create buckets to detect MIN, MAX, BABY, TEENY, GROWNUP packets.
- REQ2: Ensure all addresses received a packet, including illegal address.
- REQ3: Ensure all size packets were sent to all addresses with parity errors, except for `addr3`.
- REQ4 (optional): Ensure that packets of different lengths are transmitted to all addresses with zero delay between transmissions.

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. Decide how to sample coverage for packets sent to the DUT.

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

   ```
   MIN = 1
   MAX = 63
   BABY in [2..10]
   TEENY in [11..40]
   GROWNUP in [41..62]
   ```

4. Run a simulation.

   You'll have to modify your `irun` file to enable functional coverage collection inside the simulator. Use the coverage options as described in the appendix and online tool documents.

5. Analyze coverage results in ICCR.

   By default, Incisive® Unified Simulator puts the coverage file `icc.fcov` in the directory `cov_work/design/test`.

   a. You can load this file in iccr by launching `iccr -gui` and loading the functional coverage file from the file menu.

   b. Select the **Functional** tab and see the distribution of packets observed.

6. 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.

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

   How can you work around the limitation that we cannot have ignores in the cross?

8. Run a simulation and analyze coverage results in ICCR.

9. **Optional**: Create a coverpoint for REQ4.

   a. Use a cross to model a transition, because the simulator currently doesn't support transitions on coverpoint.

      Tip: You will have to keep track of the previous packet.

   b. To implement back-to-back tracking you'll have to make sure that the delay is captured in the monitor.

      Tip: You can use guard expression to control sampling of a cross.

10. Run multiple simulations with random svseed and analyze coverage results in ICCR.

End of Lab

Lab C1 Locating Cadence Online Support Solutions

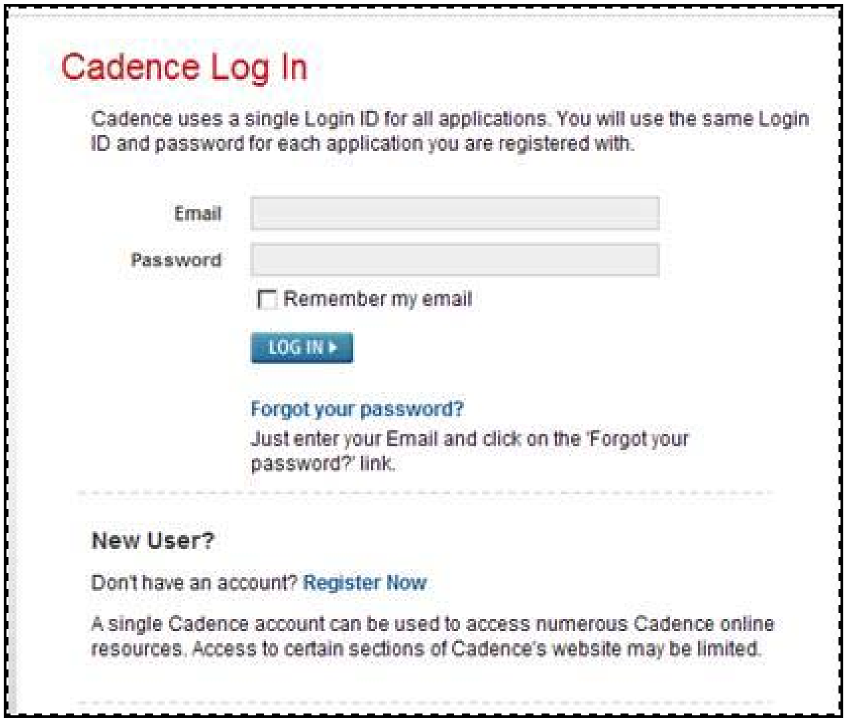
Objective: Log in to Cadence Online Support (COS) and search for information about a specific issue.

You can only complete this lab if you have access to the internet and a Cadence Online Support account. If you do not, your instructor might be able to perform a demo of this lab for the class.

1. In a web browser enter

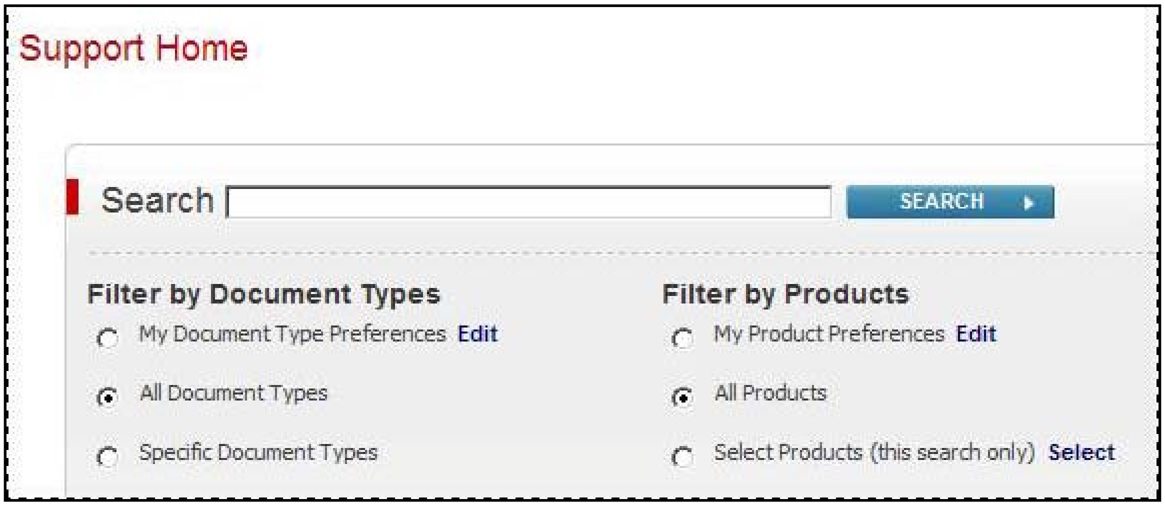
   http://support.cadence.com

2. Log in to Cadence Online Support with your email and password.



   The Support Home page appears.

3. In the Support Home page, make sure the following options are selected:
   - All Document Types
   - All Products



4. In the Search field enter

   `UVM`

   and click the **SEARCH** button. A window containing the search results opens.

5. From the search results, select any of the matches. This will open a window providing the information in the match title..

6. Close the solution window.

   Note: You can filter the search results by selecting specific document types or products which are listed to the left of the results.

End of Lab

Lab C2 Customizing Notification and Search Preferences

Objective: Set preferences so you can improve search results and receive email notification.

You can set product and other preferences for improved search results and email notification.

1. Click on the **My Account** link.

2. Click on the **Notification Preferences** tab.

3. On the Set Email Notification Preferences page:

   a. If you are interested in receiving email notifications, check **Send me email notifactions about new product releases**.

   b. Click on **Edit Product List** and select your products of interest.

   c. Specify your preferred email format.

   d. Select the document types your are interested in and the frequency of delivery.

   e. Click **Save**.

4. Click on the **Search Preferences** tab.

5. On the Set Search Preferences page:

   a. Click on **Use same product and document type preferences as my Notification Preferences**.

   b. Click **Save**.

End of Lab