# Essential SystemVerilog for UVM

**Lab Manual**

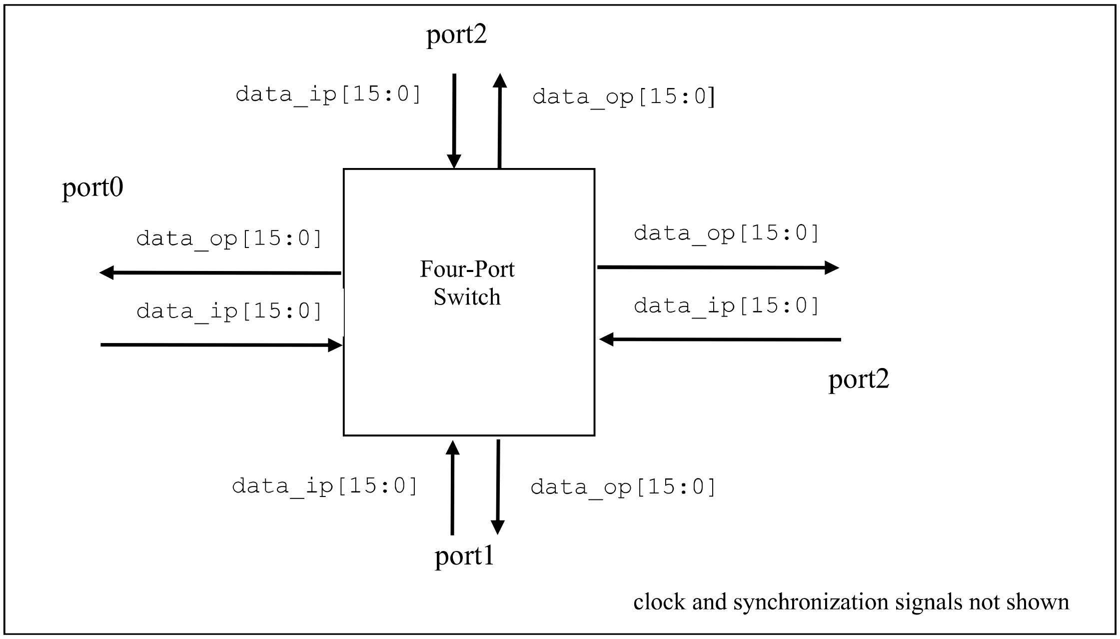
# Table of Contents

## Essential SystemVerilog for UVM

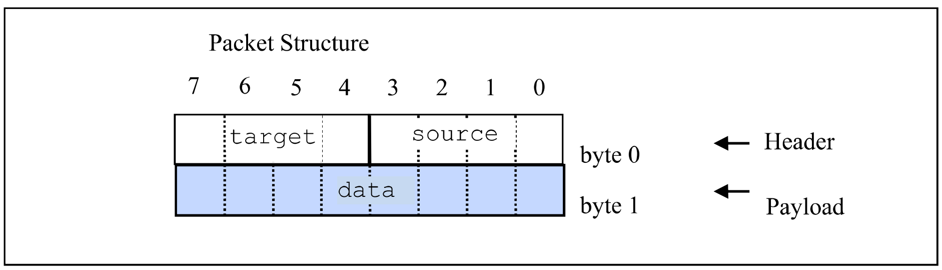
## Four-Port Switch Project Overview

In this course you will be developing verification components for a 4-port switch design.

A simplified packet switch design has four ports. The switch receives data packets on a port and transmits the packet to one or more of the four ports depending on the packet data.




The switch data packet has the following format:




Where:

- Source is the address of the port where the packet was sent *into* the switch. Source has a 4-bit, one-hot encoding as follows:

| Port0 | Port1 | Port2 | Port3 |
|---|---|---|---|
| 4’b0001 | 4’b0010 | 4’b0100 | 4’b1000 |

- Target is the 4-bit address of the port(s) to which the packet is being sent (more information below).
- Data is an 8-bit payload for the packet.

There are three types of packet, based on the value of target.

1. A **Single** packet is sent to **one** port. Its characteristics are:

   a. One bit set in target, representing the single destination port address of the packet, using the same encoding as the source field.

   | Port0 | Port1 | Port2 | Port3 |
   |---|---|---|---|
   | `4'b0001` | `4'b0010` | `4'b0100` | `4'b1000` |

   b. Single packets are not allowed to be transmitted to the same port from which they originated, i.e., target cannot equal source.

2. A **Multicast** packet is sent to between **two** and **three** ports. Its characteristics are:

   a. Either 2 or 3 bits set in target, representing the multiple destination ports for the packet. For example, a target of `4'b0110` represents a packet to be sent to both port 1 and port 2.

   b. Multicast packets are not allowed to be transmitted to the same port from which they originated, i.e., the target cannot contain the same bit set as the source.

3. A **Broadcast** packet is sent to all ports, including the source port. Its characteristics are:

   a. All bits are set in target, i.e. `4'b1111`.

# Lab 1 Simple Data Class Declaration

## Objective: To declare the Switch packet properties and methods.

### Creating the Basic Packet

The first step is to create a class for the basic packet properties and methods.

1. Work in the `essential_sv/lab1_intro` directory.

2. Edit the file `packet_data.sv` as follows:

   a. Create a class `packet` containing the following declarations:

      - Local string property `name` for the instance name.
      - Bit array properties `target`, `source` and `data`.

   b. To help in debug, we will add a property to identify the packet type.

      - Declare an `enum` type above the class declaration with the values `ANY`, `SINGLE`, `MULTICAST`, `BROADCAST`.
      - Declare a property `ptype` of the above enumerate type.

   c. Define a constructor to set the following properties.

      - Instance `name` set via a constructor argument.
      - `source` set by a constructor argument. The argument should be an integer representing the port number where the packet is sent into the switch (0,1,2,3). Convert this number into the 1-hot encoding required by `source`.
      - `ptype` set to `ANY`.

   d. Declare a method `gettype()` to return `ptype` as a `string`. Hint: use the enumeration method `name()`.

   e. Declare a method `getname()` to return the instance name as a `string`.

   f. Add a `print` method to display **all** `packet` properties using a print policy.

      - Declare an `enum` above the class declaration to define Hex, Decimal or Binary printing formats.
      - Pass an argument of the `enum` type to the `print` method with a default value.

      Hint: use `gettype()` to print `ptype`.

### Initial Packet Testing

3. Create a package in a file **`packet_pkg.sv`**. Include **`packet_data.sv`** into the package.

4. Create a module in the file **`packet_test.sv`** and add the following:

   a. Import the packet package.

   b. Declare a handle on the `packet` class.

   c. In an `initial` block:

      - Create the `packet` instance, passing the appropriate constructor arguments.
      - Assign `data` and `target` properties to non-zero values
      - Print the instance using different print policies.

5. Create a run file **`run.f`** and add **`packet_pkg.sv`** and **`packet_test.sv`** for compilation.

6. Run a simulation as follows, checking your results and debugging where necessary.

```
xrun -f run.f
```

End of Lab

# Lab 2 Simple Randomization and Constraints

## Objective: To add support for constrained randomization of the packet.

For this lab we will work in the directory `lab2_rand`. Create this directory.

1. Copy your `sv` and **`run.f`** files from `lab1_intro` to `lab2_rand`.

2. Edit **`packet_data.sv`** and declare the `target` and `data` properties of the `packet` class as `rand`. Remember `source` is set via a constructor argument according to the originating port of the packet and so is not randomized.

3. Add a constraint to the `packet` class, defining `target` cannot be zero.

4. Add a constraint to the `packet` class defining `target` cannot contain the same bit set as `source`. This will apply to single and multicast packets. We will override this constraint later for broadcast packets.

   Note that just constraining `target` to be not equal to `source` is insufficient. If `target` is `4'b0101` and `source` is `4'b0001`, the properties are not equal, but this is still an illegal packet. Hint: You could use a bit-wise operator for the constraint.

### Random Packet Testing

5. Edit the file **`packet_test.sv`** to randomize and print the `packet` instance 10 times.

6. Run a simulation as before. Check your results as follows and debug where necessary:
   - Check `target` is not zero
   - Check `target` and `source` do not have the same bit set.

7. Force a constraint violation by adding another randomization of the packet instance with an *inline* constraint to create a broadcast packet (`target == 4'b1111`).

   Print the packet after randomization.

8. Run a simulation as before. You should see a randomization failure.

   What happens to the packet properties when randomization fails?

## Conditional Constraints (Optional)

9. Declare the `ptype` property as `rand`.

10. Add a conditional constraint on `target` depending on the value of `ptype` as follows:
    - If `ptype` is `SINGLE`, `target` has only one bit set, i.e. is 1,2,4 or 8.
    - If `ptype` is `MULTICAST`, `target` has 2-3 bits set, i.e. is 3, 5-7 or 9-14.
    - If `ptype` is `BROADCAST`, `target` has all bits set, i.e. is 15.

    Hint: use `inside` operators in the constraints.

11. Modify the randomization calls to add an inline constraint preventing `ptype` from being `ANY`.

12. Run a simulation with 10 randomizations as before. Check your results carefully and debug where necessary.

13. You could also experiment with adding a constraint to `solve ptype before target`, although you may need many more randomizations to see a noticeable effect.

End of Lab

# Lab 3 Static Properties and Methods (Optional)

## Objective: To add static properties and methods to the packet class.

This lab is optional and is not required for the creation of the 4-port switch test environment.

For this lab we will work in the directory `lab3_static`. Create this directory.

1. Copy your `sv` and **`run.f`** files from `lab2_rand` to `lab3_static`.

2. We will use static properties and methods to count the number of packet instances constructed, and to add the option of assigning a unique `data` value to help in debug.

   Edit **`packet_data.sv`** as follows:

   a. Add a `static int` property `pktcount` and a normal (dynamic) `int` property `tag` to the `packet` class.

   b. In the `packet` constructor, increment `pktcount` and assign to `tag`.

   c. Declare an `enum` type above the class declaration with the values `UNIDED` and `IDED`. Declare `UNIDED` first so it is the default value.

   d. Add a packet property `tagmode` of the above `enum` type.

   e. Add `post_randomize()` which assigns `tag` to `data` if `tagmode` is `IDED`. This makes the `tag` information part of the packet so it can pass through a DUT.

   f. Add a `static` method `getcount()` to return `pktcount`.

   g. Add `pktcount`, `tag` and `tagmode` to the `print` function.

3. Edit the file **`packet_test.sv`** as follows:

   a. Comment out the existing code.

   b. Call the static method `getcount()` and display `pktcount`.

   c. Create several instances of `UNIDED` and `IDED` packets, randomize and print.

   d. Call the static method `getcount()` and display `pktcount`.

4. Run a simulation. Check the printed values of `pktcount`, `tag`, `data` and `tagmode` carefully and debug where necessary.

End of Lab

# Lab 4 Inheritance and Polymorphism

## Objective: To define and use packet subclasses.

For this lab we will work in the directory `lab4_extend`. Create this directory.

1. Copy your `sv` and **`run.f`** files from `lab2_rand` to `lab4_extend`.

2. Edit **`packet_data.sv`** to add the following `packet` subclasses:

   Hint: use `inside` operators in the constraints.

   a. `psingle`, with a constraint that `target` has only one bit set, i.e. is 1,2,4 or 8. Define a constructor to pass the instance name and source to `packet`. Set `ptype` to `SINGLE` in the constructor.

   b. `pmulticast`, with a constraint that `target` has 2-3 bits set, i.e. is 3, 5-7 or 9-14. Define a constructor to pass the instance name and source to `packet`. Set `ptype` to `MULTICAST` in the constructor.

   c. `pbroadcast`, with a constraint that `target` has all bits set. Choose the constraint name carefully so it overrides any conflicting constraints in `packet`. Define a constructor to pass the instance name and source to `packet`. Set `ptype` to `BROADCAST` in the constructor.

3. Edit **`packet_test.sv`** as follows:

   a. Declare a 16-element `packet` array and handles for the single, multicast and broadcast subclasses

   b. Use a `randcase` construct inside a `foreach` loop to randomly create instances of the `packet` subclasses in each element of the `packet` array. Assign a unique instance name for each element and randomize the instance before copying to the array. For example:

```
foreach (parray [i])
  randcase
    1 : begin
        // construct single instance with unique name
        // randomize instance
        // copy instance to next element of packet array
        end
    1 : // repeat for all subclasses
```

4. Use a second `foreach` loop after the first to `print` each array element.

5. The testbench contains a `void` function named `validate` to check packets meet the specification. Call `validate` on every array element. You may need to modify the function to match property names and types used in your packet.

6. Run a simulation and use `print` and `validate` to check packet data, types, instance names and constraints are correct.

   If you do not see any errors from `validate`, then your packets are probably correct, but you should break several packet instances to confirm `validate` is correctly checking packets.

End of Lab

# Lab 5 Building a Component Hierarchy

**Objective: To build the Verification Component (VC) infrastructure using aggregate classes.**

---

For this lab we will work in the directory `lab5_component`.

1. Copy your `sv` and **`run.f`** files from `lab4_extend` to `lab5_component`.

2. The file **`component_base.sv`** declares a `component_base` class from which your Verification Components (VCs) will extend. The class implements:

   a. Protected instance `name` with an access method `getname()`.

   b. `parent` pointer.

   c. Constructor which sets `parent` and `name`.

   d. `pathname` method to generate a hierarchal path of instance names.

   e. Simple `print()` method.

3. Declare a `sequencer` class in a new file, extending from `component_base`. The sequencer should just contain a constructor. Hint copy from `component_base`.

4. Declare a `driver` class in a new file, extended from `component_base`, and containing a constructor. The driver should also contain:

   a. A handle on the `sequencer` class.

   b. A `run` task displaying `pathname()` directly **and** off the `sequencer` handle.

5. Declare a `monitor` class in a new file, extended from `component_base` and containing a constructor. The monitor should also contain:

   a. A `run` task displaying `pathname()`.

6. Declare an `agent` class in a new file, extended from `component_base`. The agent should contain:

   a. Handles for the `sequencer`, `driver` and `monitor` classes.

   b. A constructor which constructs the sequencer, driver and monitor instances.

   c. A constructor assignment of the driver sequencer handle to the sequencer instance.

7. Declare a `packet_vc` class in a new file, extended from `component_base`. This class should contain:

    a. A handle for the `agent` class.

    b. A constructor which constructs the agent instance.

    c. A `run` task which calls the `run` tasks of the driver and monitor instances.

8. Edit the **`packet_pkg.sv`** package to include the component base, driver, monitor, sequencer, agent and packet VC files in the correct order.

9. In the **`packet_test.sv`** file, comment out the existing stimulus (do not delete – you will need some of this code in the next lab). Add the following:

    a. A handle on the `packet_vc` class.

    b. An `initial` block which:

        - Constructs the `packet_vc` instance. Remember the `parent` of the topmost component should be set to `null`.
        - Calls the `run` task of the `packet_vc` instance.

10. Run a simulation and debug where necessary. You should see displayed pathnames for the driver, monitor and sequencer instances (the sequencer pathname is called in the driver from its sequencer handle).

End of Lab

# Lab 6 Completing the Verification Component.

## Objective: To complete the Verification Component and connect to the DUT.

For this lab we will work in the directory `lab6_vc`.

1. Copy your `sv` and **`run.f`** files from `lab5_component` to `lab6_vc`.

2. The `lab6_vc` directory contains a **`port_if.sv`** interface file containing signal declarations and stimulus tasks for the switch ports. The interface needs visibility of your class declarations. Edit the `import` statement so it references the name of your package.

   Note the name of the interface (`port_if`).

3. Edit the `sequencer` class as follow.

   a. Add an `int` property `portno`. This will be set in configuration and used to assign `source` for packets generated by the sequencer.

   b. Add a `void` function `get_next_item()` with an output argument of `packet`. Use a `randcase` to randomly construct and randomize an instance of a Single, Multicast or Broadcast packet.

      Hint: copy from the **`packet_test.sv`** file of `lab4_extend`.

      In the packet instance constructor calls, assign the `source` argument from `portno`. Output the instance via the `packet` output argument.

4. Edit the `driver` class as follows:

   a. Add a virtual interface property for `port_if`.

   b. Add a handle on `packet`.

   c. Edit the `run` task as follows:

      - Call `get_next_item()` from the sequencer handle with your `packet` handle as the output argument.
      - Print the packet.
      - Call the interface method `drive_packet()` from the virtual interface with your `packet` handle as the input argument.
      - Add an input `int` argument named `runs` for the number of packets to generate.
      - Wrap the `get_next_item()`, `print` and `drive_packet()` lines in a `repeat` or `for` loop to execute the lines `runs` number of times.

5. Edit the `monitor` class as follows:

   a. Add a virtual interface property for `port_if`.

   b. Add a handle on `packet`.

   c. Edit the `run` task as follows:

      - Call the interface method `collect_packet()` from the virtual interface with your `packet` handle as the output argument.
      - Print the packet.
      - Wrap the `collect_packet()` and `print` lines in a `forever` loop.

6. Edit the `packet_vc` class as follows:

   a. Add a `configure` method. The method has two input arguments:

      - A `port_if` virtual interface. Assign this to the driver and monitor virtual interface properties via a hierarchical pathname.
      - An `int portno`. Assign this to the sequencer `portno` property via a hierarchical pathname.

   b. Edit the `run` task as follows:

      - Add an input `int` argument named `runs`.
      - Wrap the monitor `run` call in `fork-join_none`. As the monitor `run` contains a `forever` loop, this needs to be called in a separate concurrent process.
      - Pass the `runs` argument to the driver `run` task.

7. A module for initial testing of the VC is provided in the file **`vc_test.sv`**. This module:

      - Declares an instance of the `port_if` interface named `port0`.
      - Generates `clock` and `reset` signals.

   a. Check the `import` statement matches your packet package name.

   b. Declare a handle on your `packet_vc` class.

   c. Add the following to the existing `initial` block *after* the `reset` signal assignments and *before* the `$finish` call:

      - Construct an instance of the `packet_vc` class.
      - Configure the instance with `port0` interface and port number arguments.
      - Call the `packet_vc` `run` task with a `runs` argument of 3.

8. Update **`run.f`** to compile **`vc_test.sv`** instead of **`packet_test.sv`**.

9. Run a simulation and debug where necessary. You should see a print from the driver as a packet is sent to the interface and a print from the monitor as it detects the packet on the interface signals. These packets should match.

   Once you are happy with the results, you can move on to testing your Verification Component with the DUT.

## Testing with the DUT

10. A module for the DUT test is provided in the file **`switch_test.sv`**. Compared to **`vc_test.sv`**, this module adds the following functionality:
    - Declares all 4 instances of the `port_if` interface.
    - Instantiates the DUT and connects the interface instances.
    - Calls the interface `monitor` method on all 4 interface instances to capture port output data. These are called inside a `fork-join` for concurrent execution.

    a. Check the `import` statement matches your packet package name.

    b. Declare a handle on your `packet_vc` class.

    c. Copy your `packet_vc` constructor, configuration and run calls from **`vc_test.sv`** and paste into the existing `initial` block *after* the `reset` assignments and *before* `$finish`. Start with a `runs` count of 1 to begin with.

    d. Add a large delay *after* your `vc run` call and *before* the `$finish` to allow propagation of packets through the switch, e.g. `#500ns;`

11. Update **`run.f`** to compile **`switch_test.sv`** instead of **`vc_test.sv`** and add **`switch_port.sv`**.

12. Run a simulation and debug where necessary. Read the log file carefully. As well as a print from the driver and monitor of your VC, you should see prints from the output monitors for all four ports. This should allow you to track a packet through the switch, using the `data` field to identify packets.

    Edit your files if necessary, to allow easier debug. For example:
    - Comment out the driver packet print and just use the VC monitor print.
    - Edit the sequencer to send a specific packet type rather than random packets.

### Driving All Four Ports

Once we have a single VC working correctly, it should be simple to connect a VC to all four ports. Modify **`switch_test.sv`** as follows:

13. Declare handles and create instances for the other 3 ports of the switch. Each instance will have a `parent` of `null`.

14. Configure each instance to connect to a different interface instance, with a matching `portno`.

15. Add run calls for your new VC instances and wrap all 4 VC run calls in a `fork-join` to execute them concurrently. You should start with a `runs` count of 1 for each instance.

16. Run a simulation and debug where necessary. Read the log file carefully to track packets through the switch.

End of Lab