# CSE 591: Advanced Hardware Design & Verification

## *LAB #03 : Verification of 4-port Switch*

Due on May 3rd, before 11:59 pm

## 1 Contents

## 1. Overview of Project

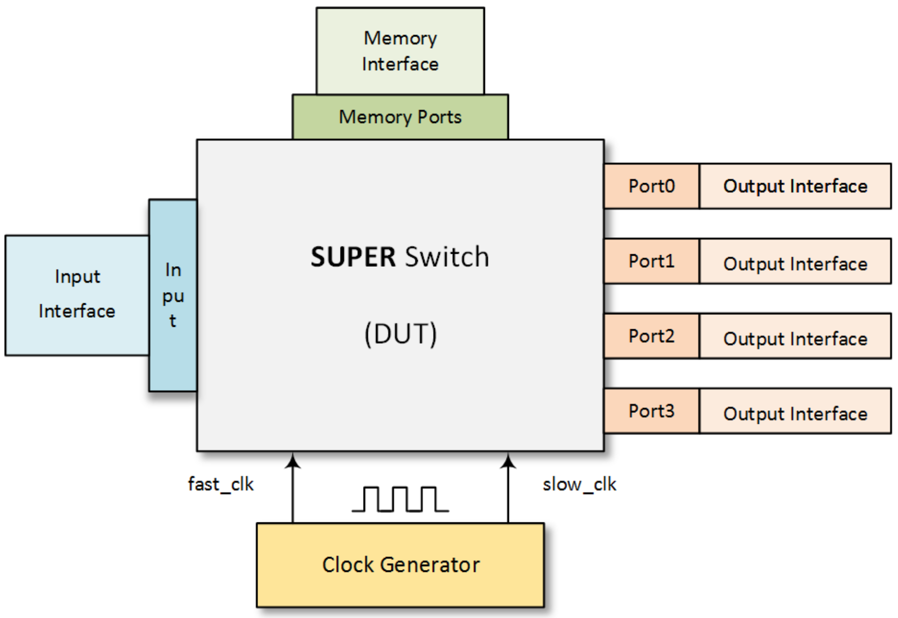
The following sections describe the latest addition to SimpleTech's product line, the "super" switch. Our customers have demanded the following features be added to the design.

- The ability to stall the producer port (input interface)
- The ability to read back the contents programmed into the memory interface.
- The ability to send the data in at a different clock frequency.
- The ability to output the data to a slower consumer clock.
- The ability for the consumer to stall the "super" switch.
- Update the port names.

## 2. Design Specification

Here at SimpleTech, we are proud to announce "super" switch. The "super" switch uses that same great packet based protocol you are familiar with from our original "switch" product offering.

The original Switch drives the incoming packet, which comes from the input port, to output ports based on the address contained in the packet. The "super" switch not only forwards packets, it allows them to come in at a much faster rate than the output interface can actually consume them. The outputs are now allowed to operate at a lower clock frequency, effectively making the switch a fast asynchronous data aggregator.



**Note:** For this design we will be using a clock frequency of 100MHz for the fast_clk and 3MHz for the slow_clk.

In addition, the "super" switch can stall the producer. When stalled, the bytes on the "data" interface are ignored. So you better make sure the ***data_stall*** output from our super switch is not asserted when you send data, or it will be ignored. In addition, we have improved the names of our switch based on customer feedback.

The switch has a one input port from which the packet enters. It has four output ports where the packet is driven out.

### 1.1 Packet Format

The packet format is backwards compatible with the switch specification. A packet contains 3 parts. They are Header, data and frame check sequence. Packet width is 8-bits and the length of the packet can be between 4-bytes to 256-bytes.

### 1.1.1 Packet Header

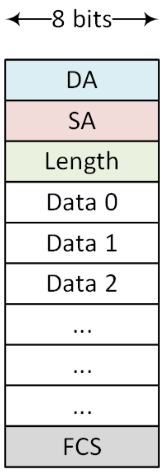
Packet header contains three fields DA, SA and length.

- **DA**: Destination address of the packet is of 8 bits. The switch drives the packet to respective ports based on this destination address of the packets. Each output port has 8-bit unique port address. If the destination address of the packet matches the port address, then switch drives the packet to the output port.

- **SA**: Source address of the packet from where it originate. It is 8 bits.

- **Length**: Length of the data is of 8 bits and from 0 to 255. Length is measured in terms of bytes.
  If Length = 0, it means data length is 0 bytes
  If Length = 1, it means data length is 1 bytes
  ...
  If Length = 255, it means data length is 255 bytes

- **Data**: Data should be in terms of bytes and can take anything.
- **FCS**: Frame Check Sequence. This field contains the security check of the packet. It is calculated over the header and data.



## 1.2 Configuration

Switch has four output ports. These output port addresses have to be configured to a unique address. Switch matches the ***DA*** field of the packet with this configured port address and sends the packet on to that port. Switch contains a memory. This memory has 4 locations, each can store 8 bits. To configure the switch port address, memory write operation has to be done using memory interface. Memory address (0, 1, 2, 3) contains the address of port (0, 1, 2, 3) respectively.

## 1.3 Interface Specification

The Switch has one input Interface, from where the packet enters and 4 output interfaces from where the packets come out and one memory interface through which the port address can be configured. Switch also has a clock and asynchronous reset signal.

## 1.4 Memory Interface

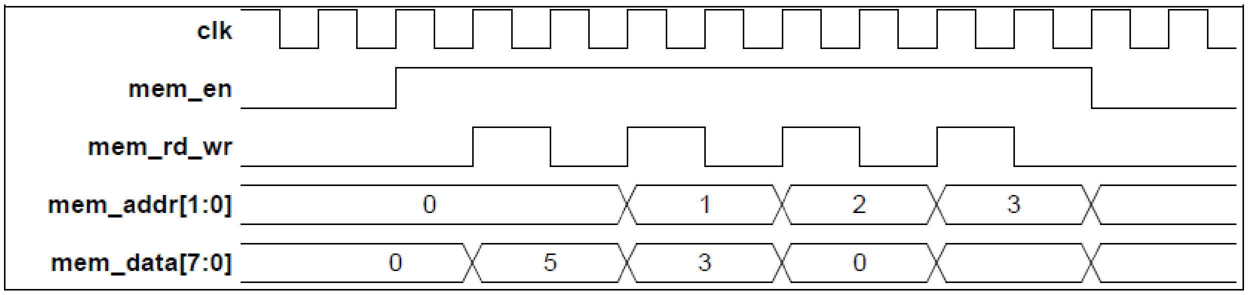
Output port address are configured through the memory interface. It accepts 8-bit data to be written to the memory. It has 8-bit address inputs. Address 0, 1, 2, 3 contains the address of port 0, port 1, port 2, and port 3 respectively.

There are 6 input signals to memory interface. They are:

```
input              slow_clk;          // Slow Clock
input              mem_en;            // Memory Interface Enable
input              mem_rd_wr;         // 1 = Read; 0 = Write
input              mem_rstn;
input        [1:0] mem_addr;
input        [7:0] mem_wdata;
output       [7:0] mem_rdata;
```

All the signals are active high and are synchronous to the positive edge of clock signal. The reset is asynchronous and active low. To configure a port address:

1. Assert the **mem_en** signal.
2. Assert the **mem_rd_wr** signal for a write (1 indicates a read while 0 indicates a write).
3. Drive the port number (0 or 1 or 2 or 3) on the **mem_addr** signal.
4. Drive the 8-bit port address on to **mem_rdata** signal.



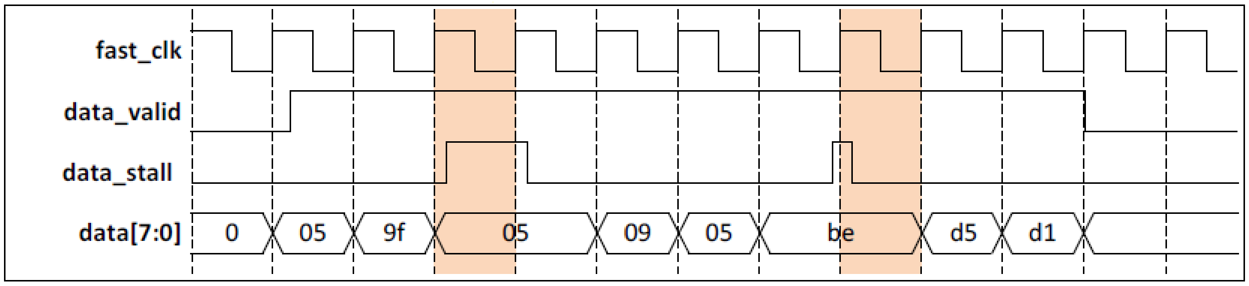
## 1.5 Input Port

Packets are sent into the switch using input port. All the signals are active high and are synchronous to the positive edge of clock signal.

```
input                  fast_clk;          // Fast Clock
input            [7:0] data;
output                 data_stall;
```

To send the packet in to switch:

1. Assert the **data_valid** signal.
2. Send the packet on the data signal byte by byte. Data is only captured if **data_stall** is low. So the current byte being sent should be held until **data_stall** goes low.
3. After sending all the data bytes, de-assert the **data_valid** signal.
4. There should be at least 3 clock cycles difference between packets (this is a requirement that needs to be followed even if your design doesn't require it).

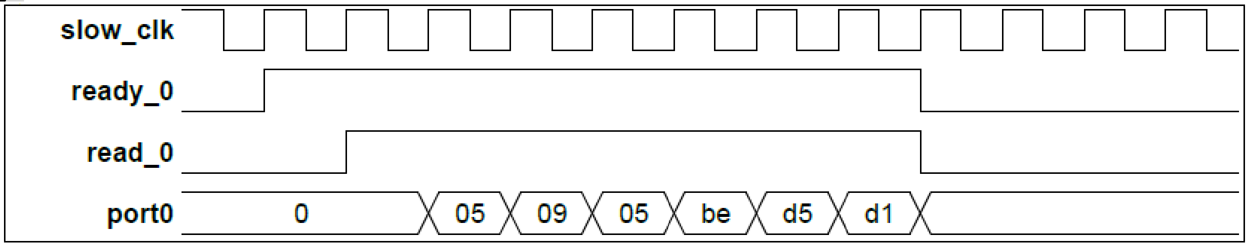


## 1.6 Output Port

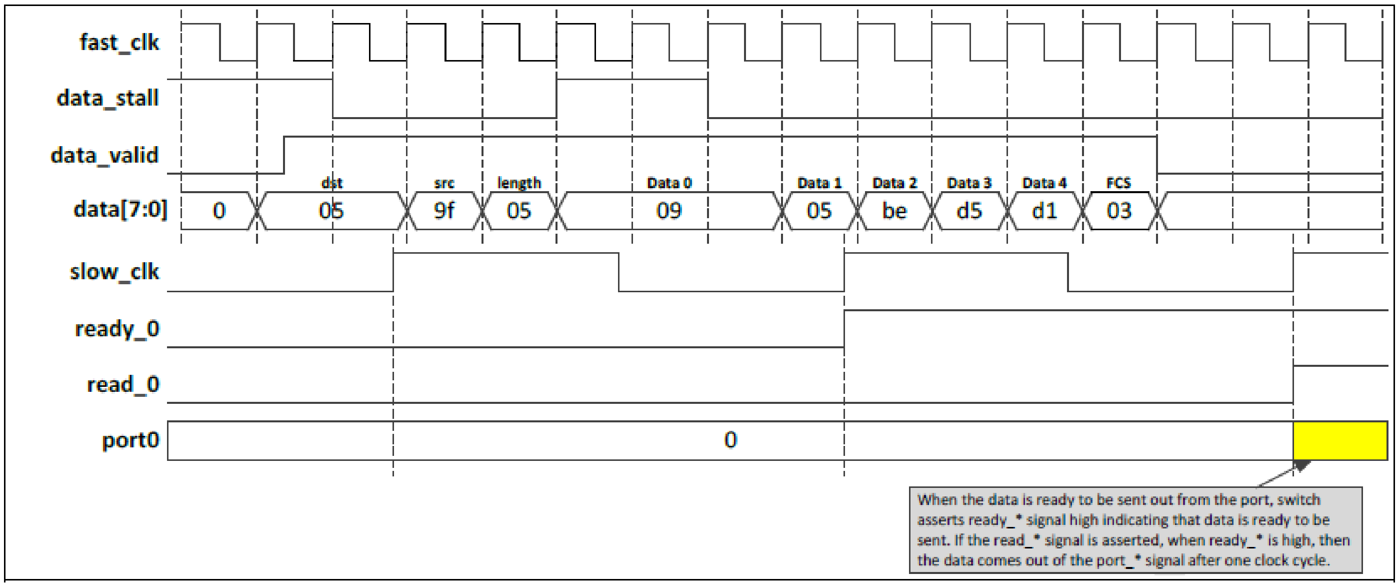
Switch sends the packets out using the output ports. There are 4 ports, each having data, ready and read signals. All the signals are active high and are synchronous to the positive edge of clock signal. The signal list is:

```
input                 read_0;
input                 read_1;
input                 read_2;
input                 read_3;

output                ready_0;
output                ready_1;
output                ready_2;
output                ready_3;

output          [7:0] port0;
output          [7:0] port1;
output          [7:0] port2;
output          [7:0] port3;
```

When the data is ready to be sent out from the port, switch asserts **ready_*** signal high indicating that data is ready to be sent. If the **read_*** signal is asserted, when **ready_*** is high, then the data comes out of the **port_*** signal after one clock cycle.



Here is an example of one possible configuration:



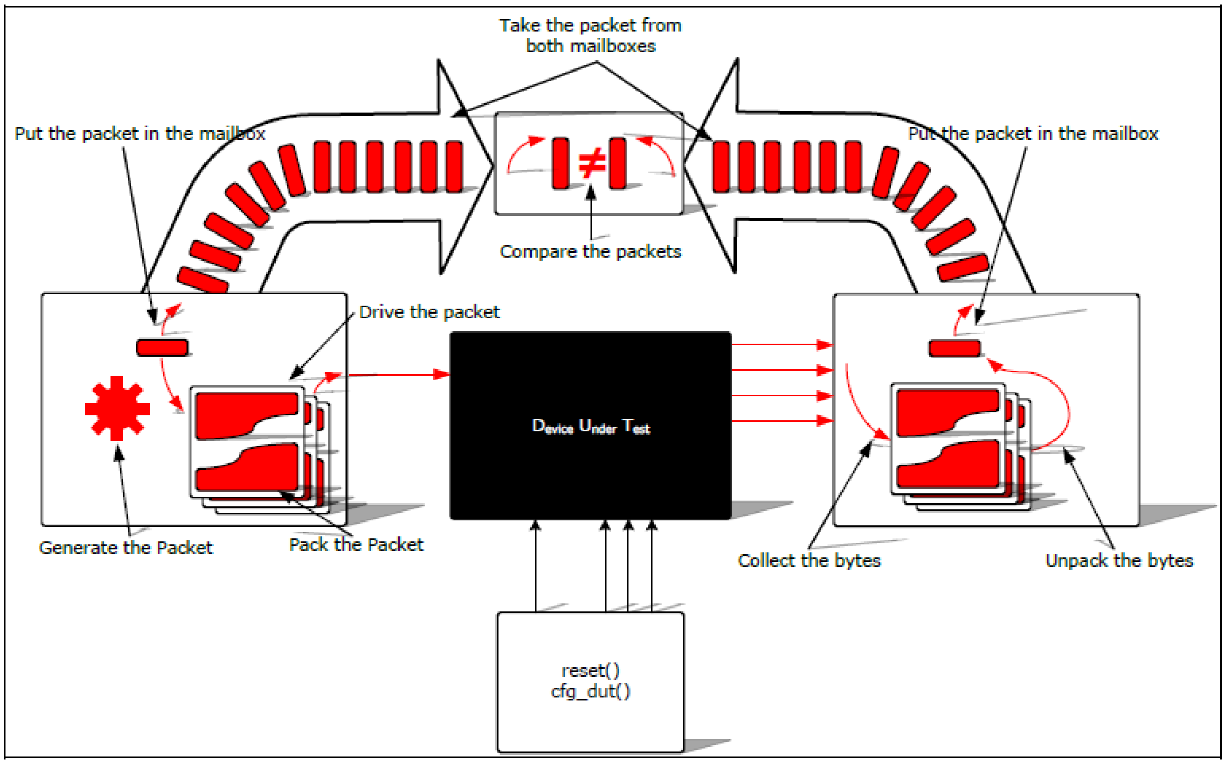
When the data is ready to be sent out from the port, switch asserts ready_* signal high indicating that data is ready to be sent. If the read_* signal is asserted, when ready_* is high, then the data comes out of the port_* signal after one clock cycle.

## 2 Verification of Switch

This section describes the Verification Plan for Switch. The Verification Plan is based on System Verilog Hardware Verification Language. The methodology used for Verification is Constraint random coverage driven verification.

As a verification team lead at SimpleTech's front-end division, you are required to develop the Verification Plan, construct the TestBench and develop testcases to completely verify the **Super** Switch before deadline.

The testbench will be similar to Lab #02, but more focus will be given on creating a constrained random environment. The class



### 2.1 Feature Extraction

This section contains list of all the features to be verified.

| TC # | Test Name | Description |
|---|---|---|
| 1 | Configuration | Configure all the 4 port address with unique values. |
| 2 | Packet DA | DA field of packet should be any of the port address. All the 4 port address should be used. |
| 3 | Packet payload | Length can be from 0 to 255. Send packets with all the lengths. |
| 4 | Length | Length field contains length of the payload. Send Packet with correct length field and incorrect length fields. |
| 5 | FCS | Good FCS: Send packet with good FCS. Bad FCS: Send packet with corrupted FCS. |

## 2.2 Stimulus Generation Plan

The stimulus…

- Packet DA: Generate packet DA with the configured address.
- Payload length: generate payload length ranging from 2 to 255.
- Correct or Incorrect Length field.
- Generate good and bad FCS.

## 2.3 Coverage Plan

- Cover all the port address configurations.
- Cover all the packet lengths.
- Cover all correct and incorrect length fields.
- Cover good and bad FCS.
- Cover all the above combinations.

### 2.3.1 Stall Input Interface Port Addresses

Now that we support stalling, update the drivers and receivers to insert random stalls of random length.

### 2.3.2 Randomize Port Addresses

Currently only four fixed values are used to set the port routing addresses. I would add randomization of these values, which would need to be constrained such that duplicate values were not used for two different ports.

### 2.3.3 Generate a mixture of good and bad packets

Presently only legal port addresses are generated. I would modify the test to generate a mixture of packets with legal port addresses that would get routed and illegal port addresses that would get dropped.

### 2.3.4 Predict DUT Behavior

In order to properly compare output results of the DUT, we need to account for any packets that get dropped due to illegal port addresses. This would entail creating a behavioral model of the DUT's functionality that is aware of the currently programmed port addresses. Packets being input to the DUT would also be input to this Predictor module; it would then determine whether to drop the packet or to route it to an output.

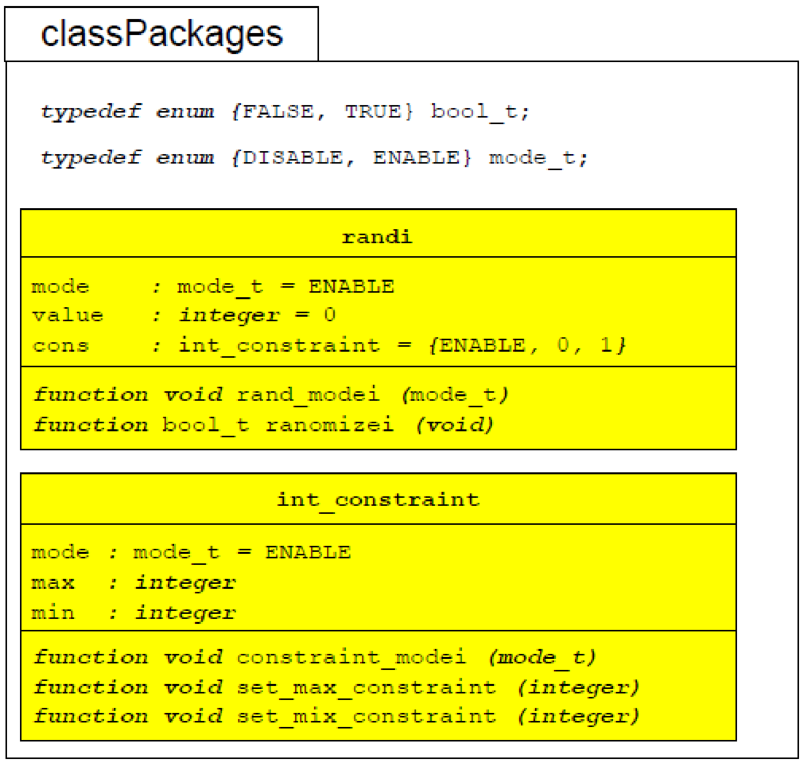
### 2.3.5 Separate receive mailboxes

With the existing test environment, when a packet is output by the DUT the four receivers all put the output packets in the same mailbox. This set up would not detect the case where a packet is routed in its entirety but output by the wrong port. I would add a separate Scoreboard for each output port so that we can verify packet routing is done correctly. The Predictor mentioned above would determine which Scoreboard to write a packet to.

### 2.3.6 Examine Scoreboard at end of the simulation

If an input packet is placed in the Scoreboard but is never received, an error will not be counted. At the end of the simulation I would examine the mailbox, the Driver writes into, to ensure it is empty. Otherwise I would count these packets as errors.

Since the `randomize` feature is not available in our educational tool version, the randomization class is provided such that the objects of this class can be randomized.


classPackages
typedef enum {FALSE, TRUE} bool_t;
typedef enum {DISABLE, ENABLE} mode_t;
randi
mode : mode_t = ENABLE
value : integer = 0
cons : int_constraint = {ENABLE, 0, 1}
function void rand_modei (mode_t)
function bool_t ranomizei (void)
int_constraint
mode : mode_t = ENABLE
max : integer
min : integer
function void constraint_modei (mode_t)
function void set_max_constraint (integer)
function void set_mix_constraint (integer)

### *2.3.6.1 randi:*

#### 2.3.6.1.1 Class Properties:

- ***mode***: This value is an enumerated data-type whose value indicates whether or not generation of random values is enabled.
- ***value***: This is the current state of the random value contained in this class. It is updated upon every call to randomizei provided the mode is set to ENABLE.
- ***cons***: Of type int_constraint. Is used to capture the individual constraints on the integer being generated.

#### 2.3.6.1.2 Class Methods:

- **`rand_modei`**: This function is used to ENABLE or DISABLE the generation of random values.
- **`randomize_i`**: This function is used to generate random values for its property value. If the current mode is set to ENABLE. Otherwise, the value is left as is. If the mode of the object of type int_constraint is set to ENABLE, the function will try to generate a random value that fits within the constraints specified by that object. It will attempt to do so (how it does so is up to you) for some number of times (you decide what is a good limit). If the mode is disabled for the object of type int_constraint, any random value generated will be accepted. If the function successfully generates a random value, TRUE is returned. If it fails, FALSE will be returned.

### *2.3.6.2 int_constraint:*

#### 2.3.6.2.1 Class Properties:

- ***mode***: This property is an enumerated data-type whose value indicates whether or not generation of constrained random values is enabled.
- ***max***: This property indicates the maximum value that can be randomly generated, i.e. value <= max, when the mode is set to ENABLE.
- ***min***: This property indicates the minimum value that can be randomly generated, i.e. value >= min, when the mode is set to ENABLE.

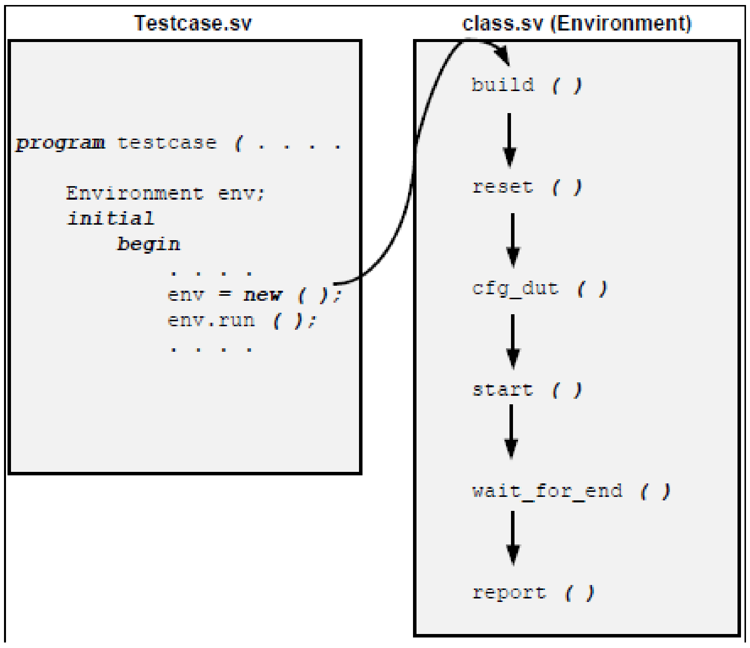
#### 2.3.6.2.2 Class Methods:

- ***constraint_modei***: This function is used to ENABLE or DISABLE the constraints on randomly generated values.
- ***set_max_constraint***: This is a method that changes the value of the property max.
- ***set_min_constraint***: This is a method that changes the value of the property min.

## 2.4 **`Environment`** Class

The **`Environment`** class is a base class used to implement verification environments. Testcase contains the instance of the environment class and has access to all the public declaration of environment class. In environment class, we will formalize the simulation steps using virtual methods. The methods are used to control the execution of the simulation. The following methods are going to be defined in environment class: new( ): The constructor method. It will be empty for the moment.

1. **`build()`**: In this method , all the objects like driver, monitor etc are constructed. Currently this method is empty as we did not develop any other component.
2. **`reset()`**: in this method we will reset the DUT.
3. **`cfg_dut()`**: In this method, we will configure the DUT output port address.
4. **`start()`**: in this method, we will call the methods which are declared in the other components like driver and monitor.
5. **`wait_for_end()`**: This method is used to wait for the end of the simulation. Waits until all the required operations in other components are done.
6. **`report()`**: This method is used to print the TestPass and TestFail status of the simulation, based on the error count..
7. **`run()`**: This method calls all the above declared methods in a sequence order. The testcase calls this method, to start the simulation.



### 2.5 Packet

We will define a packet and then test it whether it is generating as expected. The **`Packet`** is modeled using class. **`Packet`** class should be able to generate all possible packet types randomly. **`Packet`** class should also implement required methods like **`pack( )`**, **`unpack( )`**, **`compare( )`** and **`display( )`** methods. We will write the packet class in class.sv file. **`Packet`** class variables and constraints have been derived from stimulus generation plan. All fields of the **`Packet`** class should be of type **`randi`**, so that they can be randomized accordingly.

## 3 Lab Submission:

Your lab must be in the following format:

1. It must be contained in a zip file.
2. The contents of the zip file must be exactly as follows:
   **LAB3_<First_Name>.<Last_Name>** - Unzipped folder
   i. tb/class.sv
   ii. tb/interface.sv
   iii. tb/testcase.sv
   iv. tb/top.sv
3. The zip file must be renamed (change extension from **.zip** to **.piz**) exactly as follows:
   **LAB3_<First_Name>.<Last_Name>.piz**
4. Your RTL may not look similar to anyone else's RTL, i.e. NO CHEATING!
5. It must be e-mailed to the following e-mail accounts: ***Kyle.Gilsdorf@asu.edu***