How to Verify a Switch:

I. Components

A. Packet Generator

1. Network system
2. Packet types

a) Destination class

(1) Unicast

Need the destination system to get the address

(2) Multicast

(3) Broadcast

b) Layers

(1) Layer 2

(2) Layer 3

(3) Layer 4

3. Packet sizes

   64-1522 bytes (1519-1522 must be tagged)

4. Untagged vs tagged
5. Errors

a) Depends on network type

b) Set a field with needed errors

(Interface will actually generate most)

B. Port

1. Controls access to the packet generators
2. Stores a list of expected packets

   Scoreboard

3. Checks that received packets are in expected list

    a) Finds all CRCs that match

    b) Checks addresses (may only be checked by bytes)

    c) Checks the rest of the packet byte-by-byte

C. Interface

1. Base type

   The rest of the types use this as the main building block

2. Types

   Must have one for each type supported by the switch

    a) 10 BT

    Usually based on the 100 BT type running in 10 BT mode.

    b) 100 BT

    (1) MII

    (2) SMII

    c) Gigabit

    (1) PCS

    (2) GMII

3. Transmit/receive packets based on type of interface

4. Half/full duplex

5. Produce most errors

    a) Watch for incompatible error types

    b) Giant: adding bytes

    c) Undersize: stop after length

D. Packet Router

1. Modify Packet (if needed)
2. Determine and Place Packets on Destination Ports
    - **a) Addresses**
    - **b) Cast of Packet**
    - **c) Type of Layer 3 Packet**
        - (1) IP
        - (2) IPX
        - (3) Appletalk
        - (4) etc.
    - **d) Type of Layer 4 Packet**
    - **e) Forwarding Tables**
3. Can be the Most Complex Piece

Some background.
I worked for a large company for 6.5 years with the last 3.5 being in ASIC verification. I left the company at the end of December 1999 to do some teaching and to start my own consulting business.

For the previous two projects, we had been enhancing a homegrown simulation environment that used Verilog and Perl (pre and post processing) to test the switch chips. We had gotten to the limits of the old environment and were looking at rewriting it using C and PLI. We were told that we did have time to investigate exactly one new tool. We chose Specman after talking to another division who had looked into both Specman and Vera, had an onsite demo, and I worked with a Verisity FAE to do a proof on our previous project.
We saved a significant amount of time by not having to worry about memory management issues (garbage collection), the low-level PLI routines, how to get randomness that was repeatable, and various other headache issues that Specman takes care of automatically. We were then able to use that extra time in learning ‘e’ and creating a fairly robust environment that should be mostly reusable with only minor tweaks.

The old environment used a very static method of generating packets and supplying them to the switch chips. A test file would be written and then a Perl script would be run to generate three sets of files. The first set contained control information for the central environment parser to tell things when to

happen (e.g. register initialization, register checking, sending packets, etc.). The second set contained the packets that needed to be transmitted on each port (one file per port that could be very large depending on the number and size of packets being sent from that port). The third set of files contained the packets that were expected on each port (one file per port that could be even larger that the transmit files depending on how many and what size of packets were destined for the port). Each file was loaded into a Verilog memory for runtime processing by Verilog environment modules. This was taking up a huge amount of memory and consequently causing the simulations to run much slower (i.e. we had to allocate enough memory to hold the maximum size files of a given type). The control information was very limited in that is was primarily a one-way communication path, did not allow branching on conditions, and not have any looping capability (i.e. straight-line code).

We were looking for a way to reduce memory usage, give us some interaction between the environment and the switch under test (e.g. events in the switch could affect what packets were sent or what registers were checked), and allow us the full capability of branching and looping constructs.

We chose a design that would allow us to do the following:

1. Use a small template to generate packets.
2. Only one transmit packet per port would be kept around at a time.
3. Only outstanding packets (ones that hadn't been verified yet) would be kept on the destination ports.
4. Test writers could use all the features of 'e' to write directed tests.
5. Easily set up random tests.

Figure 1: Top Level Block Diagram

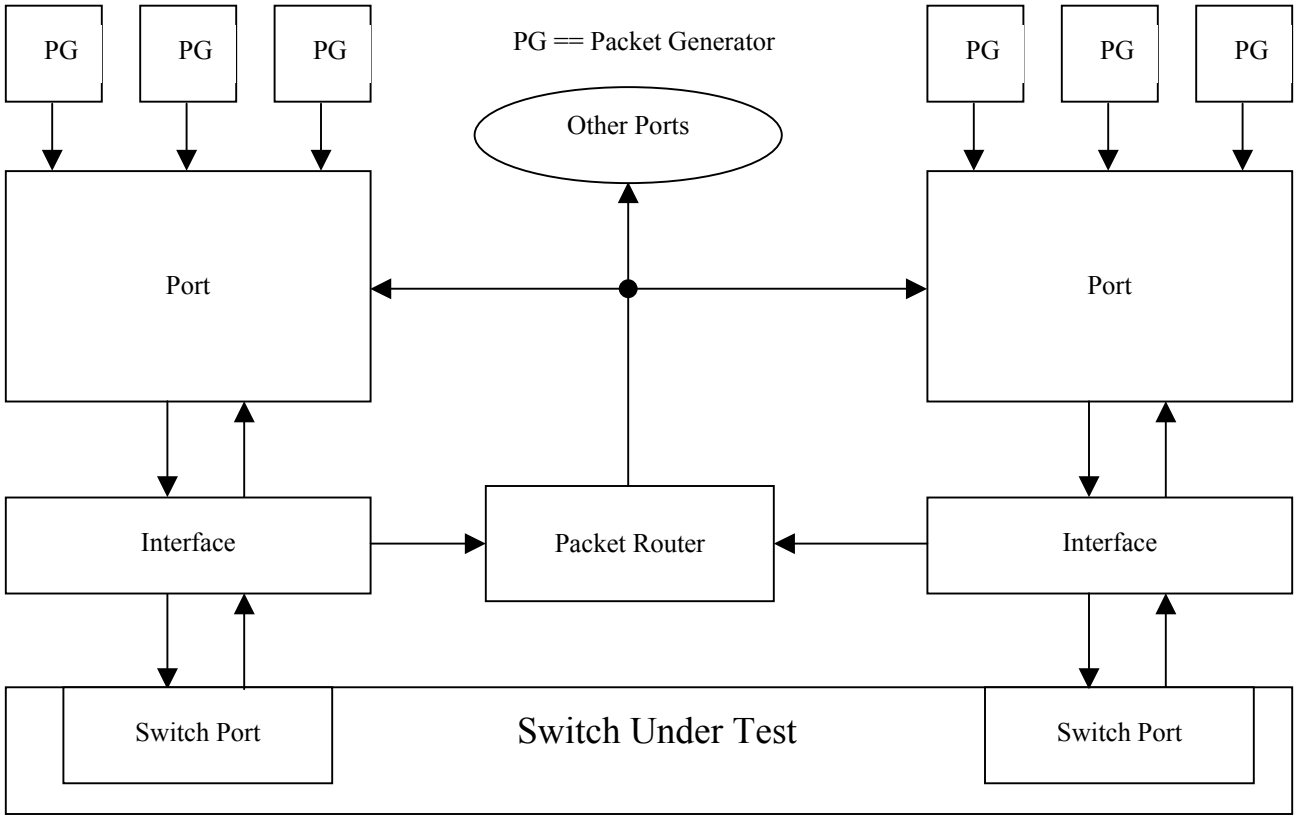

Figure 1 is the top level block diagram showing the verification environment and the switch under test.

TX:

1. The packet generators give a packet to the port when requested.
2. The port gives the packet to the interface.
3. The interface sends the packet to the switch port (generating any needed errors) and then to the packet router when finished.
4. The packet router modifies the packet as necessary and places it on the scoreboard for each port that is supposed to receive the packet. Depending if there were errors and what type, the packet router may not place the packet on any scoreboards.

RX:

1. The interface receives the packet from the switch and passes it to the port.
2. The port finds the matching packet on the scoreboard and verifies that they are identical.
3. The port removes the packet from the scoreboard and announces that it has received the packet (prints out some packet information for visual tracking).

Figure 2: Packet Generator

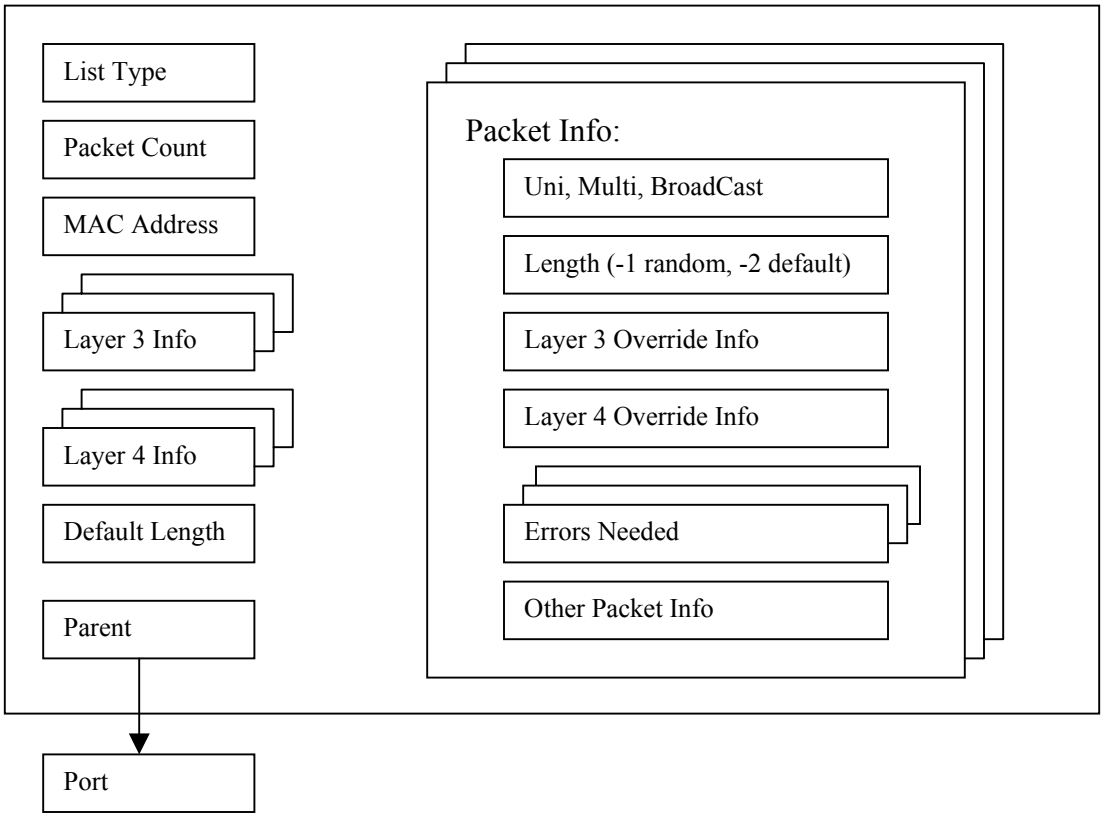

Figure 2 is the block diagram for the Packet Generator. At a minimum, it contains:

1. MAC address, Layer 3 information, and Layer 4 information of the network device it represents. Default length for packets (-1 indicates random and is the default value).
2. Pointer to its parent port.
3. List of packet information structures that tells the packet generator about the next packet.

The "List Type" field is used to tell how the "Packet Info" list is to be used.

1. It can be ignored and completely random packets generated up to the "Packet Count".
2. It can be treated as a single template for all packets up to the "Packet Count".
3. It can be used as a list of templates and the "Packet Count" field is ignored.

The packet information structure doesn't have to have all of the fields listed filled in (only cast and length are required). Any fields that are empty use the default information for the packet generator or don't get created for the packet.

This design allowed us to reduce the amount of space needed for storing packets that need to be transmitted (i.e. only one full packet per port is kept around at a time and the rest are stored as small template structures). This also allowed us to easily specify random packets (size, type, etc.).

Figure 3: Port

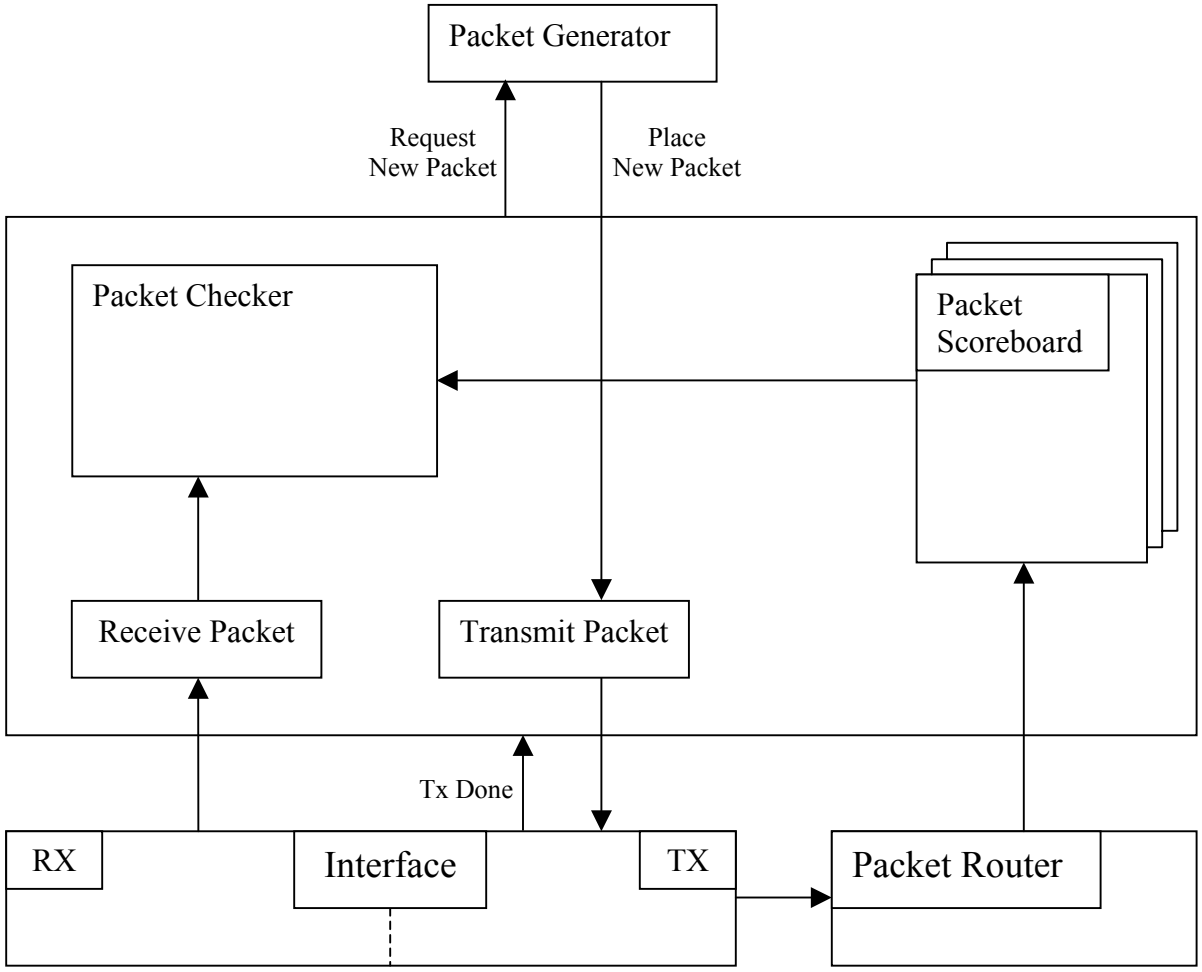

Figure 3 is the block diagram for the port. It contains the following:

1. A transmit packet.
2. A receive packet
3. The expected packet scoreboard
4. The packet checker
5. Some control logic to request new packets from the Packet Generator.
6. There are also some fields (not shown) that are used for configuration purposes and are inherited by the Interface.

The expected Packet Scoreboard allowed us to reduce memory use for storing expected packets since packets are placed on it only after an Interface has successfully transmitted the packet to the switch; and as soon as a packet is verified, the packet is removed from the scoreboard.

Another item that saved us space is the fact that we only had to store one transmit packet per Port and the address of that packet is passed to the Interface rather than the entire packet.

Figure 4: Interface

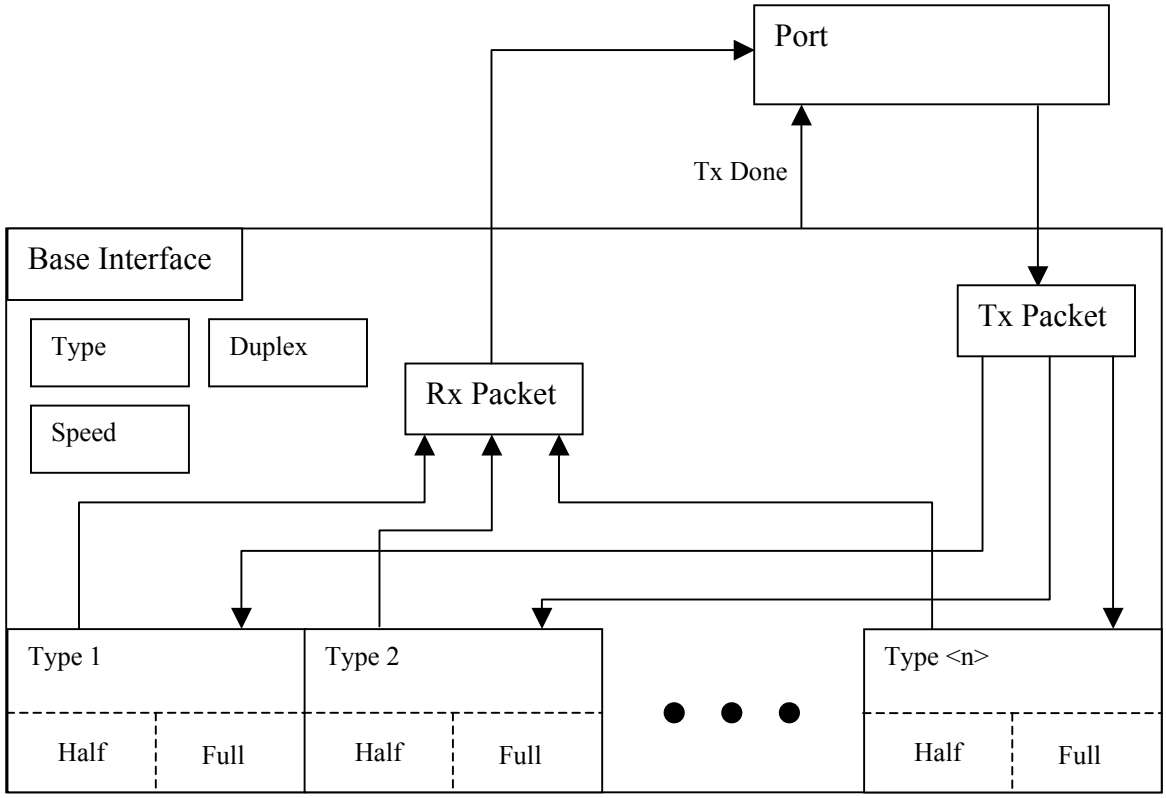

Figure 4 is the block diagram for the Interface. The Interface contains:

1. A base interface that contains all of the common pieces.
   a. The type.
   b. The duplex (half or full).
   c. The speed (10 Mbps, 100 Mbps, 1 Gbps, 10 Gbps, etc.).
   d. A pointer to the Tx Packet.
   e. Space for the Rx Packet.
2. A sub interface for each type (uses the when statement to create). Each sub interface can have a sub section for half and full duplex and for different speeds (speeds are normally handled by clocks coming from the switch or by if statements rather than when statements). The sub interface is what has all of the protocol code for the interface type (e.g. MII, SMII, GMII, etc.).
3. An error generator (this may be split between the base interface and the sub interfaces depending if there are any error types that are specific to an interface type).

This allowed us to easily substitute in different card and port types in the switch to test out all of the different combinations.

Figure 5 Packet Router

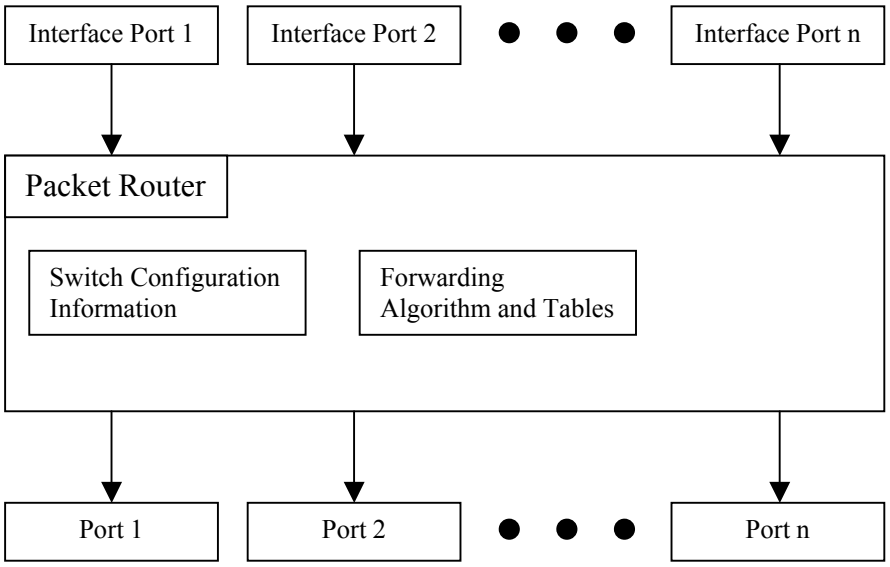

Figure 5 is the block diagram of the Packet Router. It contains the following:

1. Switch configuration information.
2. Forwarding algorithms and tables.

The interface calls the Packet Router when successfully finished with transmitting a packet. The Packet Router then uses the switch configuration, forwarding algorithms and tables, and the data in the packet to modify and determine destination port(s) to place a copy on the receive scoreboard(s).