Ethernet over SONET (EoS) Device Verification, Leveraging SUPERLOG™

Stefen Boyd

Boyd Technology, Inc.
Email: stefen@boyd.com
Tel: 408 739 2693

HDLCon 2001
Santa Clara, CA
March 2nd, 2001

Abstract
The complexity of electronic device verification continues to expand exponentially with device size. This trend has caused the employment of a diverse range of techniques, targeted at minimizing the productivity impact of the verification phase within the overall project cycle. This paper proposes a verification environment architecture for a typical networking device, employing a case study to illustrate good verification practices. Components of the SUPERLOG language are leveraged to demonstrate the power of the capability when utilized for these applications.

Introduction
Choosing an effective verification methodology is a critical element of hardware design projects, and given the complexity of modern electronics design, this task is fraught with difficulties. This paper describes an environment for verification of a device that encapsulates Ethernet packets in SONET frames. Relevant methodology is highlighted and examples are used to show SUPERLOG implementations for stimulus generation, output checking and verification models.

The Verification Task
The 'Design Under Test' (DUT) is an Ethernet over SONET (EoS) device, as shown in Figure 1. This design was chosen as it represents a typical level of device complexity, and includes standard interfaces often utilized in the networking industry.

The DUT has two primary interfaces. The interface for the interconnection of Physical Layer (PHY) devices to Link Layer devices (PL3) will be referenced as the Local Area Network (LAN). The other interface will be referenced as the Wide Area Network (WAN) since it connects to OC-24 SONET transceivers. Ethernet packets are transferred between the interfaces, and either encapsulated into (LAN to WAN) or stripped from (WAN to LAN) SONET frames. The Ethernet and SONET ports have a one-to-one mapping. In a system, the LAN interface would be connected to an Ethernet PHY such as the PMC-Sierra S/UNI 2XGE. A local processor bus is also provided for system firmware diagnostics and to enable device administration, for example the handling of error conditions.

The proposed verification system will focus on the top level testing of this device, as supposed to individual block level tests. At this level, testing will focus on the correct operation of the entire device within the total system. The key difference between this level and block testing, apart from the amount of design data, is the interface specifications; the top level tests tending to utilize standard interfaces.

If the environment is constructed correctly, designs with different applications can leverage its components. A whole class of devices that transport data from one interface type to another could utilize this environment with a little

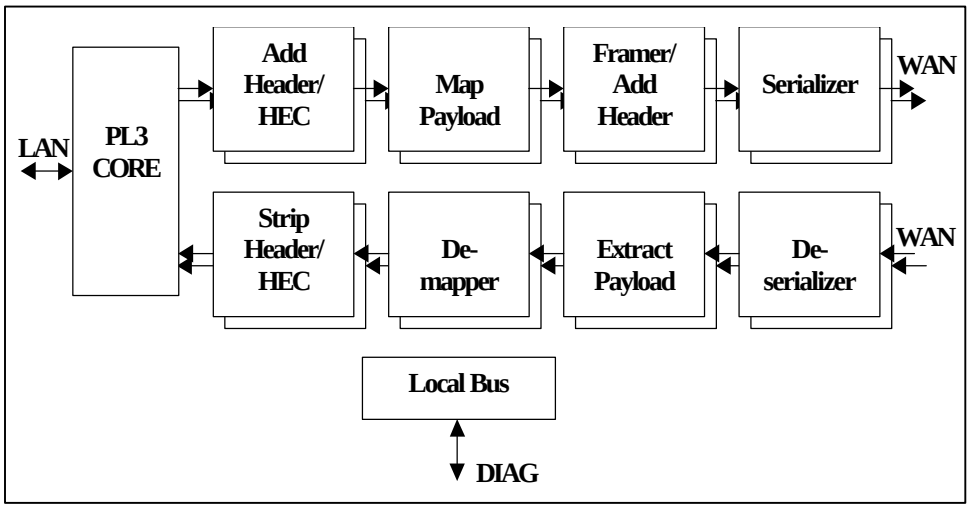

Figure 1 – A Block Diagram of the Device Under Test

additional complexity, enabling the verification of many networking systems.

Environment Design Goals and Objectives

The primary goal of this environment is to provide the maximum test coverage in as easy to use a fashion as possible, enabling a reduction in the writing and debugging of complex tests. This has the benefits of easier maintenance, a more robust and higher quality environment, and most importantly, greater testing efficiency. Key objectives that need to be observed are:

- Organization. A clear approach to the organization of the environment components makes it easier to understand and allows a greater level of extensibility, important to provide for the changing needs of this DUT or for future applications.
- Reuse. A test environment often gets re-applied to new designs, providing a known validation procedure to unknown, new design code. An ability to apply old tests and prune out redundant code is important for this purpose.
- Stimulus Generation. Since the data has few dependencies, random transactions may be used to stress the DUT, hunting down the most complex of corner cases. This device type lends itself well to directed stimulus, where a packet generation model drives different inputs, using parameters selected in the testcase to provide content specification. These may easily be extended into random generation.
- Output Validation. Self-validating testbenches provide a high degree of automation, and the more completely the environment checks the output, the greater degree of confidence achieved. To achieve this, the same Ethernet packet data used to stimulate is passed to Ethernet checkers that verify the integrity of data passing through the DUT.
- Efficiency. This may be measured both in term of man hours and tool execution. Taking advantage of available levels of abstraction improves both, and this is relatively easy, as synthesis of the testbench is not required. SUPERLOG provides multiple constructs that enable abstraction to be utilized.

The Verification Environment

The structure of the EoS verification environment closely follows that of the DUT.

The environment is built around models that interact with the DUT at each of the three interfaces. Since the device is used for EoS, there are Ethernet packet generator and checker models on both the LAN and WAN interfaces of the DUT.

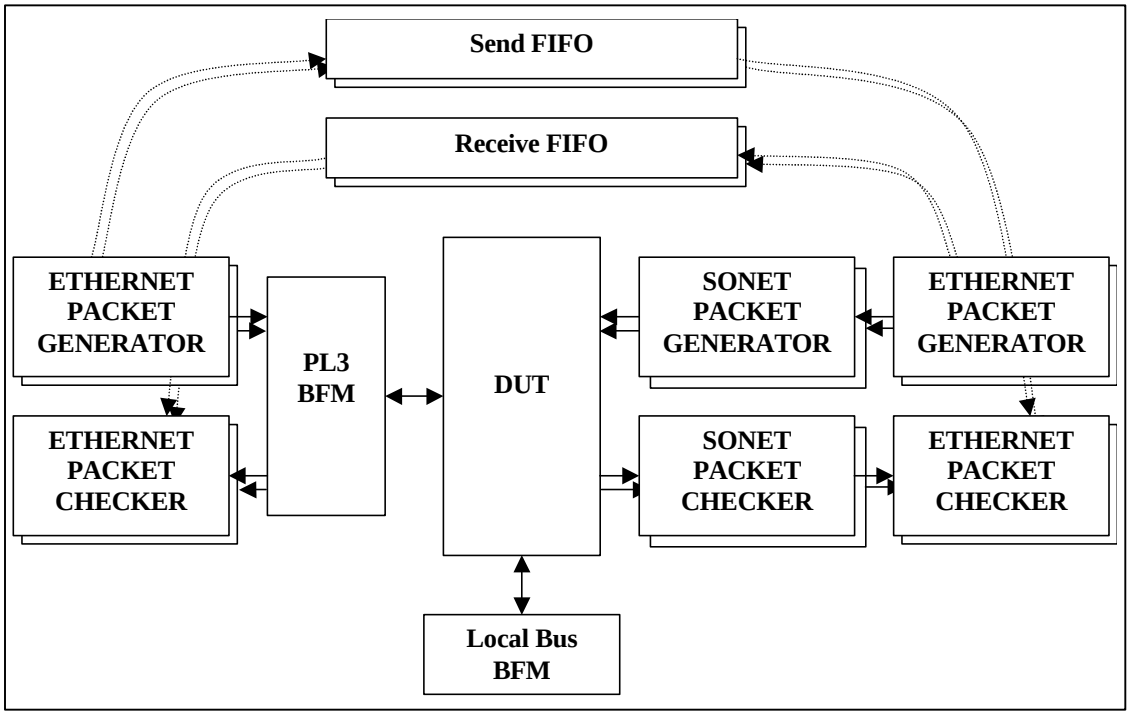

Figure 2 – EoS Validation Models and Data

```
sample_test sample_test_inst;
module sample_test;
  initial
    if ($test$plusargs("sample_test")) begin
      top.activate();       // initialize design
      // initialize parameter lists
      top.eth1.ctls.length = {16, 256, 10, 1024};
      top.eth1.ctls.count = 15;
      top.eth1.ctls.delay = {0, 1, 5, 500, 10000, 100, 100000};
      top.run();            // execute test
    end
endmodule: sample_test
```

Figure 3 – Example Test

The fundamental data structure flowing between the LAN and WAN interfaces of the design is an Ethernet packet, so our test environment will use Ethernet packets as the basis for stimulating and checking the output. The header, extended header and HEC included into the SONET frame will need to be checked for self-consistency. The content of each packet from LAN to WAN or vice versa is placed into a FIFO when it is generated.

The Ethernet packet checker uses that same FIFO to ensure the packet traversed the design without error. Note that there are two Ethernet generators and checkers on the LAN side. Even though they travel over a single interface to the DUT, this allows easier generation and checking of the packets coming from or going to the Ethernet PHY ports.

Synchronization is achieved throughout the blocks using control signals between the models, and a relatively simple set of tasks control starting, stopping, status display and completion for each test. The control signals are included within a simple interface structure utilized by all the models and the tasks are in the testbench module. Other SUPERLOG enhancements are used throughout the environment to provide advanced messaging and code consistency checking. These can be seen in examples as the print task and ":label" usage at the end of modules, tasks, etc.

The environment is structured such that the tests themselves are separated from the testbench components. This improves maintenance as testbench change can be separated from the tests, reducing the modification ripple affect. This also helps the engineers see the "forest for the trees" as they analyze their tests.

A key aspect of the testbench has to be its "modularity." Although the initial testbench has been specified for use with two WAN and LAN ports, it must be flexible enough to handle multiple numbers of ports, as well as adapt to changing interface requirements. This requirement can be met by using a set of models to describe interface detail and control those models from the tests. This allows for the substitution of models if the interface spec changes, with a minimal impact on the higher-level transaction tests that are driving the models.

Another important benefit of model usage is the abstraction it provides to tests. Tests only need to specifically control behavior that is different than the default. A test that only creates different Ethernet packet types could run without modification even if the PL3 interface was completely replaced since it would not access the PL3 BFM at all. An integrated Ethernet PHY could be used to replace the PL3 core with minimal impact on tests. Also, since the test only configures the desired behavior, the models can be used from many different sources. Not only can Verilog models be used, C and C++ models can be integrated into the environment without using a PLI interface. Systemsim's™ CBlend™ technology makes calling these C and C++ models as easy as calling a Verilog model.

Test Generation

Given the hierarchical nature of the environment, the tests will center on the parameter settings required to produce the desired effect, with the rest of the environment providing the structures necessary to transform these parameters into actual behavior. For example a test could set a sequence of say 15 Ethernet packets, with specific restrictions on data lengths and inter-packet delay. The resulting test fragment (shown in figure 3), simply needs to pass this information into the respective model structures, with the models doing the rest. This method provides clear concise tests, which are easy to write and maintain.

```
typedef bit [6*8-1:0] eth_addr_type;
typedef bit [2*8-1:0] eth_len_type;
typedef bit [3*8-1:0] eth_fcs_type;
typedef struct {
  eth_addr_type dest_addr, source_addr;
  eth_len_type  length_type;
  byte data[0:$];             // variable length field
  byte pad [0:$];
  eth_fcs_type  fcs;
} eth_pkt;
typedef ref eth_pkt ep_ref;
```

Figure 4 – Ethernet Packet Structure

There are a number of simple rules to be followed when creating tests:

- The test name should be unique. The test is instantiated as a top-level module. The testbench is instantiated as "top" and contains the tasks called by the test. Both use SUPERLOG's explicit instantiation of top-level modules.
- The test is a module with no ports. With the interactions loaded into models, no input/output is required from the test, just value settings.
- Optionally a plusarg can be used to activate the test. This can save recompiling in compiled code simulation just to rerun certain tests.
- The test is structured as an initial block, with the activate() task used to initialize the design and perform any set up, followed by the parameter settings, and finally the run() task which executes the test.

Note two SUPERLOG features used in this example. First, SUPERLOG has the ability to initialize an unbounded array of values as a list (similar syntax to concatenation of bits in a vector). Second, a label can be placed at the end of constructs such as modules and tasks. Unlike a comment, this label is syntax checked to ensure that it matches the declared name.
The basic Ethernet datastructure used throughout the environment is shown in figure 4. This structure is referenced by the FIFOs and utilized by the generators and checkers to transfer packets. The definition is similar to a structure definition in C except that the data and pad fields have a range of [0:$]. The '$' in the range specifies an unbounded variable sized array. Using typedefs for the fields is good practice as they can be shared with other structure definitions for improved readability.

Default values are provided to portions of the environment that are retained and used if the test does not overwrite them. This simplifies test requirements and also aids test creation. Note the typedef of ep_ref. This creates a pointer to an eth_pkt that is used in much of the environment.

If error conditions need to be inserted into the simulation, they may be controlled in a similar fashion to other stimulus. The environment can contain additional parameters to drive errors, and these errors are then set up using the same test structure as before. This facilitates cross environment communication, as it is likely that an error introduced in one part of the environment will require checking in another, for example a CRC fault in a generated packet may well result in the packet dropped from the list to be checked. Additional interfaces, such as the diagnostic link, will require checking for the dropped packet.

Output Checking

There are various methods that are commonly employed to check test output, some better than others. In complex test environments such as this, it is key that the environment provides checks which are run during simulation, as this is the only way that rigorous validation of results will occur, especially after the design has gone through numerous iterations and the test re-run countless times.

In this environment, one of the major tests is the correct transmission of packets through the DUT. As packets are generated, a pointer to the packet is passed to the FIFO, which is then used by the checker at the other end to ensure that the same data is transmitted from DUT. The FIFOs are implemented using a SUPERLOG interface, which allows the queue datastructure to be packaged with transaction tasks, and the whole

```
typedef enum { NONE,
               BAD_FCS} error_type;
typedef enum { RANDOM,
               PATTERN_A5,
               PATTERN_5A,
               INCREMENT,
               USR_DATA} data_gen_type;
typedef struct {
  error_type       error_type[0:$];
  data_gen_type    data_gen[0:$];
  byte data[0:$];
  eth_addr_type    dest_addr[0:$],
                   source_addr[0:$];
  eth_len_type     length[0:$];
  // pad and fcs are auto-generated
  bit  [31:0]      delay[0:$];  // cycles between packets
  int              count;
} eth_gen_ctl;
```

Figure 5 – Ethernet Packet Generator Control Structure

ensemble then referenced in the environment, much like a C++ class. SUPERLOG queues are used to facilitate the datastructure, allowing flexibility in the size and handling of the FIFO. Parameters are also used to set the type of queue, such that the structure may be utilized for different situations.

Protocol Monitors are very effective mechanisms for quickly catching errors and aiding debug. While serial interfaces can be verified almost completely by checking the data on the interface, bus interfaces, such as PL3, often have more complex requirements. SUPERLOG provides an effective assertion mechanism, which may be used to construct monitors of this sort. Used with syntax for branching and parallel checks, a monitor can be quickly constructed using easy to understand Verilog syntax. Since this aspect of SUPERLOG has not yet been publicly released, the PL3 protocol monitor is not included in this paper.

Ethernet Generator and Checker

To illustrate the characteristics that should be part of all the models in the environment, the Ethernet models will be examined. They illustrate desirable verification model features. The Ethernet generator demonstrates how the model supports the parameterized generation of packets to provide directed testing. Not only does the Ethernet checker use a number of interesting SUPERLOG constructs, it highlights the benefits of SUPERLOG interfaces. These can be seen by their impact on simplifying the checker and the efficiency gained when used for model-to-model communication.

The Ethernet Generator is the primary source of stimulus data for the environment. It creates new eth_pkt structures and populates the fields based on the parameter settings given by each test. The structure used to define the packets, shown in figure 5, need not have values for every field. Empty fields imply default behavior. Each generator has a default eth_pkt that is modified so the unassigned fields remain valid. Since other models use the same type of controls, tests need only set parameters where non-default behavior is required. Note that the use of unbounded arrays (‘[0:$]’) is what supports the list-style initialization shown in the sample testcase.

Each LAN/WAN port pair has a generator instance, FIFO, and checker in each direction. Packets are placed into the corresponding FIFO by the generator for use by the checker. Although the body of the generator is not shown, it uses the same env_ctl interface to start or abort packet generation and to indicate to the environment that it has completed packet generation.

The Ethernet Checker (figure 6) will ensure that Ethernet packets traverse the DUT without modification. This model is a little simplistic as the final model should be able to cut down on spurious error messages by recovering from trivial errors, such as picking up the correct comparison again after a dropped packet error.

```
module eth_chk(interface dut_out,
               interface checker_pkts,
               interface env_ctl);
  bit active = 0,
      exp_pkts = 0;
  ep_ref dut_ep, check_ep;
  assign env_ctl.complete = !exp_pkts;

  always @(env_ctl.start) active = 1;
  always @(env_ctl.done) begin
    active = 0;
    disable CHK_BLK;
  end
  always begin: CHK_BLK
    dut_out.get_pkt(dut_ep);
    if (active) begin
      if (!checker_pkts.empty()) begin
        checker_pkts.pop_data(check_ep);
        if (notequal(dut_ep, check_ep))
          p.print(p.ERROR,
                  "%s %m: %s\n%s%s%s",
                  "Data miscompare in",
                  "Expected packet:",
                  p.struct_prettyp(check_ep),
                  "Actual packet:",
                  p.struct_prettyp(dut_ep));
      end else begin
        p.print(p.ERROR,
                "%s %m: %s\n%s%s%s",
                "Unexpected packet in",
                "Recieved packet:",
                p.struct_prettyp(dut_ep));
      end
    end
  end: CHK_BLK

  function bit notequal(ep_ref dut_ep,
                        ep_ref check_ep);
    // compare fields of ep_ref structures
  endfunction: notequal

endmodule: eth_chk
```

Figure 6 - Ethernet Packet Checker

The also needs to set exp_pkts to 0 when there were no more packets to check.

The env_ctl interface is simply a collection of two events and a wand. The checker uses the start and done events to enable and disable checking. Not only does the done event prevent a new check from starting, it disables a pending check. This would only be used to abort a test. Completion is indicated to the environment and other models by driving the complete wand. This leverages low-level Verilog wire resolution to indicate a 1 only when all of the models have finished generation and checking.

Packets are received from the DUT through the dut_out interface. This is not a unique interface, but one of two interfaces that provide the same get_pkt() task. The interface provides abstraction of PL3 or SONET to the checker by giving a single common access method. It is inside that interface that the eth_pkt structure is added to other encapsulated packets in a SONET frame, or is passed to a PL3 BFM using the BFM task

calls. So long as there is a get_pkt() task that returns a pointer to a populated ethernet packet structure, the interface can communicate with any kind of model (Verilog, C, C++) through task calls, or even the DUT by causing the appropriate signal transitions on bus or serial connections.

Every packet received from the DUT will be checked against the packet placed in the FIFO by the generator so long as the checker is active. Packets that are unexpected (because the FIFO is empty) or are incorrect will cause an error to be generated. Note that the error messages use a print function that is in a top-level module instantiated as 'p.' It is good practice to standardize the messages and it allows pass and failure statistics to be generated simply by the number of errors displayed. This task also takes advantage of SUPERLOG's ability to pass variable numbers of arguments of any type to a task.
Unlike the interface used for the dut output, the FIFO (checker_pkts) interface will be identical for all instances of the checker. Although it is identical in this environment, another project might substitute a different interface to to better represent the operation of it's DUT. So long as empty() and pop_data() tasks are provided by the new interface, the Ethernet checker can be used without modification. This is true even if the interface is no longer a FIFO, but instead implements complex quality of service algorithms by calling a C model of the design!

Summary
This paper provides a perspective on some good techniques, using a practical case study. Guidelines were given for the architecture of an environment that is capable of verifying many types of network devices. A scalable and modular environment was presented that can be adapted to other networking designs and embrace new verification features as they are added to the SUPERLOG language. Finally, significant portions of the environment have been shown in examples to illustrate features that are currently available in the language.

References

1. T1X1.5/99-268r1 Generic Format for Carrying Ethernet MAC Frames over SONET, October 4-8, 1999.
2. Flake, SUPERLOG 2000 Language Definition G3, October 2000.