PARADIGM® WORKS

Using Specman Elite to Verify a 6-Port Switch ASIC

Timothy J. Holt
Paradigm Works
tim.holt@paradigm-works.com

Tom Franco
Narad Networks
francot@naradnetworks.com

Robert J. Ionta
Paradigm Works
bob.ionta@paradigm-works.com

Jack Collins
Narad Networks
collinsj@naradnetworks.com

Abstract

This paper will show how Specman Elite was effectively used to verify a 6-port switch ASIC in a critical time-to-market situation. The goals of the project were to 1) develop a test environment quickly in order to get immediate test results, 2) build a reusable, maintainable environment that could be used for different versions of the ASIC, and 3) make effective use of directed-random stimulus in the testbench. The system under test was a 6-port switch ASIC. There was much complex functionality to test on this ASIC including flow control, upstream traffic arbitration, a proprietary physical layer, and error detection/recovery. The testbench consisted of multiple port transactors, a microprocessor transactor, scoreboards for the port and the microprocessor interfaces, several internal monitors and checkers, and a packet generator. The project concluded very successfully with a first-pass working ASIC.

1. Introduction

This paper describes an ASIC verification project that used Specman Elite (*e*) as the primary verification language. The system under test is a 6-port switch ASIC that was coded using the Verilog HDL language. The goals of the verification project (aside from producing a first-pass successful ASIC) were to 1) develop a test environment quickly in order to get immediate test results, 2) build a reusable, maintainable environment that could be used for different versions of the ASIC, and 3) make effective use of directed-random stimulus in the testbench. Specman *e* was chosen as the verification language since it appeared capable of meeting these goals. The success of the project depended on the efficient use of a revision control system, a thorough written test plan, and realistic completion criteria (which were established at the outset of the project).

In general, downstream traffic enters the Switch ASIC from the upstream port at speeds up to 1 Gbps. This traffic is switched to any of the five egress ports and/or a processor port. Four egress ports can be used at 1 Gbps, or five ports can be used when a single port is at 1 Gbps and the other four ports are running at 100 Mbps. In the upstream direction, traffic enters the Switch ASIC from four or five ports depending on the configuration. The traffic is aggregated and prioritized and sent upstream on the single upstream egress port. Traffic can be monitored by the onboard microprocessor, which can additionally insert traffic into the upstream path.

2. System under test

The system under test is called the Switch ASIC. It is a 6-port device that supports bidirectional data streams running at 1 Gbps at each port. The main functions performed by the Switch ASIC are as follows:

- Switches downstream packets to the output port for which they are destined;
- Provides a large amount of downstream packet buffering to accommodate traffic shaping;
- Allows the replacement of the upstream physical layer port with the C5-DCP, a network processor, over a modified GMII port;
- In the upstream direction it supports the multiplexing of five traffic streams each carrying 1 Gbps onto a 1-Gbps output port. It can also be configured to support multiplexing of four 100-Mbps input ports and a 1-Gbps input port onto a 1-Gbps output port;
- Provides on-chip queuing based on four priorities (QoS classes) in the upstream direction, enabling services requiring low latency, jitter, and packet loss;
- Minimizes packet loss by implementing priority-based flow control in the upstream direction;
- Incorporates intelligence into the network by integrating a PowerPC processor on the board, allowing for the creation of managed networks;
- Implements a weighted round robin (WRR) scheme to ensure that unsubscribed bit rate (UBR) bandwidth is shared fairly between users;
- Transports standard link layer Ethernet packets across the Narad physical layer.

3. Verification tools and testbench

The following verification tools were used on this project:

- Specman Elite;
- VCS;
- Covermeter;
- Virsim;
- LSF (job batch system);
- CPU farm (16 processors);
- Perl-based run/regress scripts;
- CVS for source code control;
- JitterBug for bug tracking.

Most of these tools were chosen based on price and industry standards. (There is little reason to spend money on tools that can be used for free: Perl, CVS, and JitterBug.)

The testbench itself consists of the following modular components:

- Port transactor (×6);
- Microprocessor transactor;
- GMII transactor;
- Port/microprocessor scoreboards (×7);
- Internal monitors/checkers (upstream, downstream, dropped packets, arbitration);
- Testbench ASIC configuration file;
- Packet generator;
- Top-level testbench code.

The port transactor (bus functional model) is responsible for generating packets and driving them onto the port interface of the ASIC. Every packet generated is placed on the scoreboard for that port for later checking. The port transactor also receives packets from the ASIC and puts these on the scoreboard to be checked. The upstream port transactor is slightly different from the downstream port transactor because of flow control issues. The upstream scoreboard is also a little different from the downstream scoreboards because packets headed upstream act differently than packets headed downstream (routing downstream, aggregating traffic upstream).

The microprocessor transactor is responsible for sending and receiving packets just as the port transactor does. It also implements the ASIC configuration and setup routines, contains an interrupt monitor for checking and responding to interrupts, and is available to the test for accessing the ASIC.

The scoreboards receive transmit and receive packets from each port and microprocessor transactor. Since there is source port information embedded in the data of the packet, the scoreboard is able to determine where the packet came from and can check every byte of data. The scoreboard also receives information from each of the internal monitors and checkers to deal with dropped or stalled packets. Of course, any packets that are left over in the scoreboard at the end of simulation might indicate an error.

The testbench and ASIC configuration code is based on a single configuration file that contains information about every ASIC and testbench configuration. New configurations are simple to add since only the differences between the base configuration and the new configuration need to be specified. A Perl script is used to parse the file and generate the appropriate *e* file to include at the top level of the testbench. This *e* file contains a set of defines for each ASIC configuration register, in addition to some testbench defines. Each ASIC register definition contains read/write, hard reset, soft reset, and default value information. There are four basic configurations of the Switch ASIC defined. These configurations are different combinations of port speeds and numbers of active ports.

The packet generator is essentially a *struct* that defines an Ethernet packet with the additional physical layer and proprietary embedded control information. Packets are generated on the fly by the transactors based on default constraints or constraints imposed by the test itself.

The *unit* concept is used to create all of the transactors and checkers, as well as the top level of the testbench. Each of the transactors, checkers, and monitors is instantiated at the top level. The *hdl_path* of that particular entity is set at the top level. Therefore, each low-level unit does not need to know anything about how it actually fits into the hierarchy of the Verilog environment.

Most of the tests developed for this testbench are directed-random tests. The user specifies the number of packets to be generated on each port (via a constraint), and also specifies any other constraints to be modified or added. The most simplistic test is a mere dozen lines of code that essentially specify the number of packets per port. Since Specman allows the user to override or modify existing constraints or to add new constraints, it is easy for the test writer to modify the behavior of the testbench. The more tightly everything is constrained, the more directed the test becomes. The timing and content of every packet can be controlled by the test. This, combined with easy access to the RTL itself and to the microprocessor interface, allows for completely directed tests if desired.

Figures 1 and 2 illustrate how the Switch ASIC fits into the testbench in this environment.

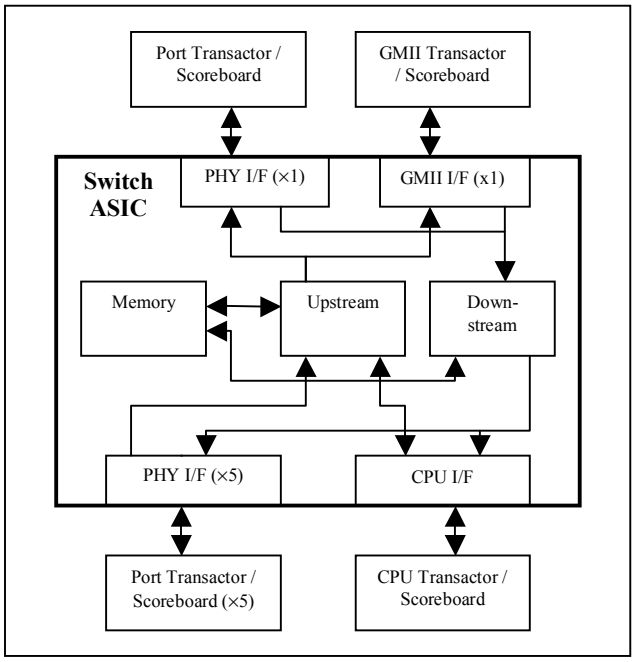

Figure 1 — Testbench/DUT block diagram

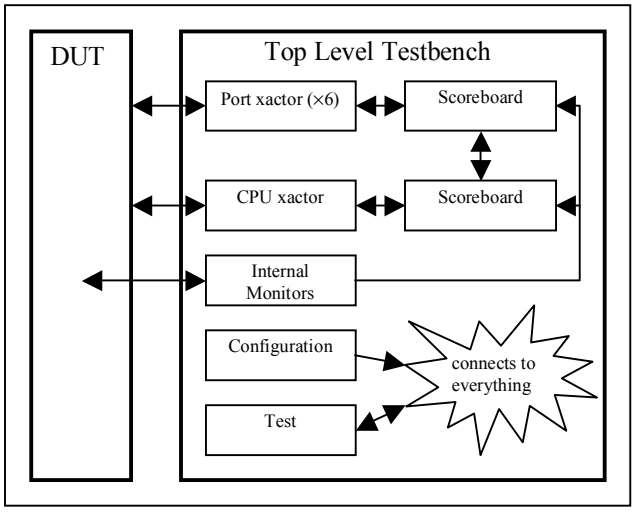

Figure 2 — Testbench

4. Interesting items to test

There are several interesting features of the Switch ASIC that required significant testing. These include flow control, arbitration, the physical layer, and error detection/recovery.

The Switch ASIC implements flow control in the downstream direction (i.e., upstream ports are allowed to flow control downstream ports). This is necessary because there can be 4 Gbps of data heading upstream into a 1-Gbps pipe. We needed to test the ASIC's response to flow control from upstream and also verify that flow control information heading downstream was generated properly.

The Switch ASIC implements a complex arbitration scheme for traffic headed upstream. Packets of different QoS classes are given different priorities, and only the highest priority traffic is allowed to travel upstream through the ASIC. The complete arbitration mechanism is fairly complex and required a significant amount of code to model.

The physical layer is the next major item to be tested. The Switch ASIC implements a complex framing mechanism on the physical layer. Ethernet packets are packed into frames containing carrier, symbol synchronization, SFD, and payload. In addition to the Ethernet payload, there is an additional small payload on every frame that contains control information for the network. On transmit, the physical layer interface takes the framed data, scrambles it, breaks the bytes into 16-QAM symbols, then scrambles the signs of the symbols before passing the symbols on to the analog portion of the transceiver. The converse of this happens on the receive path.

Finally, the Switch ASIC implements several mechanisms for error detection and recovery, including both physical line errors and internal RAM errors.

5. Packet generation and flow control

The standard packet in this environment is an Ethernet packet with an additional 2-byte header (length and CRC), an additional N-byte embedded control field, and a recalculated CRC. None of the Ethernet data is actually used by the Switch ASIC. All switching is done based on the N-byte embedded control field. The packet is defined by a *struct*, and all of the *struct* fields can be constrained by either the testbench or by the test itself. In addition to the data items that are part of the physical packet, there are many items that are used only for control of the packet generation and randomization. For example, the packet structure contains switches that allow generation of length/header CRC errors, bad length errors, and data CRC errors.

Since frames are streamed constantly from the ASIC, they may or may not contain valid packets. They do, however, always contain valid out-of-band (OoB) data every M bytes. Contained in this OoB data is the flow control information. The OoB data is defined by another *struct* that can be easily constrained by the testbench (or test). The frame itself is not defined by any structure but is created on the fly by the port transactor as it generates packets to send to the ASIC.

The port transmit transactor is responsible for actually generating the packets, framing them, converting the byte stream to symbols, and then sending them to the ASIC. The port receive transactor does the converse: it grabs

data off of the wire, converts the symbols back to a byte stream, deframes, and then repacketizes the data. There are a few things that the test can control in the transactor. These are the interpacket gap (IPG), flow control, and the interface speed (1G/100M). The latter is actually more under the control of the testbench configuration routines.

The constraints used on the packet, the OoB data, and the port transactor are typically *soft* constraints, which means that they can be easily overridden by the test. Many of these constraints are also tied to the speed of the port (1G/100M). The test or testbench may also want to control these constraints on a port-by-port basis (e.g., different traffic patterns on each port). This is accomplished by using *e*'s implication operator in the constraint generation, (e.g. *keep port == 6 => soft bad_crc == FALSE;*).

The flow control byte is defined as part of the OoB data structure. The Switch ASIC enables flow control by transmitting flow control bits in the OoB data to a downstream node. The port receive transactor is responsible for grabbing the flow control information from outgoing frames from the Switch ASIC and passing this information to the port transmit transactor. If flow control is set for a particular queue, then no traffic will be generated for that queue (QoS value). When the port transmit transactor is emulating the upstream port, it can set flow control to the Switch ASIC. Using constraints and extensions of existing data structures allowed us to easily add functions that tie together the IPG and flow control properties of the transactor to the embedded control field in the packet.

6. Error generation

In general, there are several methods of error injection in any test environment. In this environment we mainly use three of those methods:

- Allow constraints to be modified by the test or testbench (e.g., *keep length < 64*).
- Add an enable/weight function for a particular error (e.g., *keep crc_error == select [20:TRUE; 80:FALSE;]*).
- Extend an existing structure definition to modify existing functions or to add a *post_generate()* method.

Specific error mechanisms used in this environment are as follows. There are flags in the packet definition to enable CRC errors (both Ethernet and length/header) and bad length errors. Runts and oversized packets are generated by simply adjusting the length constraint on the packet. Runt packets can cause problems with the CRC generation and packing functions of the packet structure. Therefore, additional code in the packet structure is needed to compensate for this. Errors on the embedded control field of the packet are generated by extending the *post_generate()* method of the packet structure. Bad flow control and IPG is handled by constraints in the port transactor. Finally, there is always the option of monitoring and forcing a HDL net to achieve the error condition. This was done in our environment mainly to inject internal RAM errors.

Many errors and other combinations of parameters are only valid on certain ports (e.g., the microprocessor port will never have CRC errors because it is considered an on-board port). Specman's implication operator is ideal to use for this condition. It allows us to modify constraints only on specific ports or only after other constraints have been met.

7. Internal monitors

There are several internal chip monitors in this testbench, including:

- Upstream QoS;
- Upstream packet drop;
- Downstream arbitration;
- Downstream routing;
- Downstream packet drop.

The rationale behind peeking into the ASIC is simple. In this particular system it is difficult to predict which port a packet should come out of the chip on and when it should actually come out. In order to predict this, we would have to model all of the internal pipeline delays throughout the entire ASIC. Instead, by monitoring internal interfaces we can correlate between packets seen internally and the packets on the external interface. Unique identifiers within each packet are monitored as the packet traverses the logic. Logs containing these identifiers assist with isolating missing packets rather than always looking at waveforms. Specman is not necessarily any better at doing these tasks than any other high-level language would have been, except for its ability to easily monitor any HDL net by simply quoting the signal name.

The upstream QoS monitor receives the data from the upstream memory RTL interface and checks the QoS functions: on-chip queuing based on four QoS classes and the WRR scheme that handles the lowest priority UBR bandwidth.

The downstream arbitration monitor is written in C. Specman allows easy access to C routines, thus easing any potential problems. The downstream arbitration monitor is written this way because C is in general much faster than *e* and we wanted to test the downstream arbiter in a stand-alone setting before the remainder of the chip was ready. The drawback to integrating C is the lack of visibility into the C code with the Specman debug tools.

C debuggers were not attempted simultaneously with the Specman toolset. The goal of this monitor is to predict at a cycle-accurate level what the arbiter should be doing. The inputs and outputs of the arbiter are monitored, the inputs are correlated and checked against traffic sent into the Switch ASIC, the prediction is made based on these inputs, and finally the outputs of the monitor are checked.

The downstream routing monitor is used to verify that the downstream packet gets routed to the correct destination port. It has its own model of the routing table and verifies that the ASIC does the correct thing. Additionally, it is used to help the external scoreboards check their data. The scoreboard simply asks the downstream routing monitor if this particular packet should have ended up at this port before doing the data checking.

The drop packet monitors (both upstream and downstream) are used only to help the packet scoreboards do their jobs. Predicting which packet would be dropped (either upstream or downstream) was deemed too complicated even when looking at the internals of the ASIC. The packet scoreboards, however, still need to know which packets are being dropped. If there is a problem in the chip where too many packets are being dropped, it would be detected by the external bandwidth monitors that sat on each of the ports.

8. Meeting our goals

As stated earlier, the goals of the project (in addition to having working silicon at the end) were to 1) develop a test environment quickly in order to get immediate test results, 2) build a reusable, maintainable environment that could be used for different versions of the ASIC, and 3) make effective use of directed-random stimulus in the testbench.

The first goal was handily met. We had a basic, useful testbench up and running in 4 weeks. This testbench included most of the major pieces of functionality in the testbench (port transactors, microprocessor transactor, packet generation, drop packet monitor, upstream and downstream monitors, and scoreboards). This accomplishment would not have been possible without the use of a high-level verification language. Even using C or C++, we would had to have written a large number of PLI access routines in order to access the ASIC. With Specman, this is all taken care of behind the scenes so that we could focus on writing the transactors and checkers.

The second goal of the project was easily met as well. All of the testbench pieces are fairly modular and can be pulled from the testbench with little effort. This was a significant bonus when it came time to put together an environment for the FPGA port of the Switch ASIC. The FPGA version of the device had multiple FPGAs, different clock speeds, different number of ports, and different size memories. Because of the modular nature of the Specman components in the testbench, we were able to pull together this environment in short order.

Our third goal, to make effective use of directed-random stimulus, was also met. Verifying today's ASICs is a significant challenge, and directed testing often falls far short. Random testing is necessary to hit all of the nasty corner cases in the design that the verification engineers cannot think of. Using Specman not only allowed us to easily generate random stimulus, but also enabled us to direct this stimulus towards many specific corner cases that we did think of.

There were several issues that were not addressed due to resource constraints on the project. The major issue was module level testing on portions of the ASIC. Specifically, we wanted to test the MAC/PHY block, the upstream data path, the downstream data path, and the upstream arbiter. Given the environment that was created, this could have been easily accomplished. These modules each have relatively generic MII-like interfaces that the external port transactors and packet generators would have hooked up to easily. Because of the portable nature of our Specman code, it would have been simple to create module level testbenches for these items.

We had also planned to use Specman's function coverage feature to improve, or even direct, some of the tests we had written. This turned out to be a non-trivial task, and time constraints prevented its proper execution.

Lastly, we wanted to make performance enhancements in the testbench. This would have involved recoding many of our *e* functions to improve speed. According to one of the profilers that we used, we were seeing more than 85% of the CPU time of the run being taken by Specman. Again, because of schedule constraints these enhancements were not made.

This last issue is probably the most significant drawback to using Specman *e* as a verification language. Specman is great to get a testbench up and running in a short amount of time. This benefit has a price, however, in that run times can get very long.

After the project ended, the performance was enhanced by utilizing performance tools provided by Verisity to target code bottlenecks. The first target was to rewrite complicated keyword *events* into *time-consuming methods*. This improved performance 2 times over the initial coding schemes. Other commands that were deemed too inefficient were list commands such as *pop0()*. This actually pops the first item off the list but then shifts all the other items up in the list to cover the missing location. The *gen* command was found to be very inefficient when used on large *structs*. It was found to be much more effective to *gen* the *struct* once, and then any item that required on-the-fly randomizing later was *gen'ed* individually at the time it was actually needed by the simulation.

9. Conclusion

The project concluded very successfully. It took 6 months from verification concept to ASIC tape-out. We had a basic, useful testbench up and running in 1 month. Verification was completed only 1 month behind the original, extremely aggressive schedule. We ran more than 30,000 jobs via LSF and attained line coverage of more than 95% in less than 4 months. Towards the end of the project we had a regression of over 100 tests, which took approximately 500 CPU hours to complete. It was easy to port the environment from the ASIC testbench to several different FPGA testbenches. Finally, the most important goal was met: the ASIC worked correctly the first time.

The team communicated very well and made quick decisions on methodology. Code was leveraged very heavily between team members, and as a result, less time was spent trying to reinvent the wheel if someone had already developed code for a similar item. As with any language, there seems to be a small subset of the *e* language that gets the job done and thus speeds up the learning curve for a new team member to become productive. Specman *e* was a critical element in the timeliness and success of this project.