March 11 - 12, 2002

Using Specman Elite to Verify a 6-port Switch ASIC

by

Timothy J. Holt

Principal Consulting Engineer

Paradigm Works

Introduction

- Case study of an ASIC verification project using Specman Elite (*e*) as primary verification language
- System under test is a 6-port switch ASIC
- Goals of verification project
  - First pass successful ASIC
  - Quick development of initial test environment
  - Build a reusable and maintainable test environment
  - Easily allow both random and directed testing

Device Under Test

- Six-port switch ASIC
- Switches downstream packets to output ports
- Provides downstream packet buffering
- Five-to-one multiplexing of upstream data streams
- Upstream queuing based on four priorities (QoS classes)
- Priority-based upstream flow control to minimize drops
- Microprocessor interface can be used for traffic management
- Transports standard Link Layer Ethernet packets across the Narad physical layer

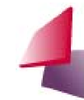
Verification Tools

- 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

Testbench Components

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

Testbench/DUT Block Diagram



Testbench Block Diagram

Interesting Items to Test

- Flow control
  - Throttling 5G heading upstream into a 1G pipe
- Arbitration
  - Packets of different QoS classes given priorities
  - Complex – modeling required significant code
- Physical Layer
  - Complex framing mechanism over the physical layer
  - Required significant testing
- Error detection and recovery
  - Physical line errors
  - RAM errors

Packet Generation

- Packet Definition
  - Length Header (w/ CRC), SA, DA, N-byte Embedded Control, Ethernet Data, Recalculated CRC
- Transactor Definition
  - IPG, Flow Control, Speed
- Most constraints are soft and can be modified by test
- Added functions that tie IPG and Flow Control of transactor to the Control field of the packet
- Many constraints are tied to the speed of the port
- Use the *e* implication operation (=>) to control the packet and transactor on a port-by-port basis

Generic Error Generation

- Allow constraint to be modified

```
keep length < 64;
```

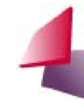
- Add enable and weight function for a particular error

```
keep crc_error == select {
        20 : TRUE;
        80 : FALSE;
};
```

- Extend packet or transactor definition
  - Modify existing functions
  - Add new functions
  - Extend *post_generate()* method
- Force HDL signal directly

Specific Error Mechanisms

- Flags in packet definition to enable CRC errors and Bad Length Errors
- Runts and Oversized generated by adjusting Length constraint
- Bad Embedded Control fields generated by extending *post_generate()* method
- Bad flow control and IPG handled by constraints in port transactor
- Internal RAM errors created by forcing HDL signals



Basic Error Test

```
<'
extend asic_dut_env {
   keep for each (p) in rfic_drivers {
      p.num_packets == (index == 0 ? (50 * PACKET_MULT)  : (10 * PACKET_MULT) );
   };
};

extend rfic_drvr {
   keep soft pkt_delay == select {
      50 : 0;
      50 : [1..100];
   };
};

extend  oob {
   keep soft bad_oob_crc == select {
      80 : FALSE;
      10 : TRUE;
   };
};

extend n_packet {  // Don't allow NULL packets to contain errors
   keep length == 6 => bad_crc == FALSE;
   keep length == 6 => bad_len_crc == FALSE;
};
```

Basic Error Test

```
extend  n_packet {
   keep soft rand_patt == TRUE;

   keep soft length == select {
      10 : [28..(`MIN_PKT-1)];    // runt packets
      50 : [`MIN_PKT..110];       // small packets
      10 : [110..(`MAX_PKT-1)];   // large packets
       1 : [`MAX_PKT];            // Max packets
      10 : [(`MAX_PKT+1)..1600]; // Too Large packets
   };

   keep soft bad_crc == select {
      80 : FALSE;
      10 : TRUE;
   };

   keep soft bad_len_crc == select {
      80 : FALSE;
      10 : TRUE;
   };
};

'>
```

Internal Monitors

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

Integrating C code into a Specman environment

- Downstream Arbitration Monitor written in C
- Monitor written to test Arbiter in a module level environment
- Inputs to monitor are HDL signals passed through Specman
- Output of monitor is the predicted result which is checked in the calling *e* routine
- C code integration process is as easy as adding some hooks to the makefile

What didn't we do?

- Specman functional coverage
- Module level testing (MAC/PHY, Upstream, Downstream, Arbiter) using existing packet generation transactors.
- Save/Restart for batch mode
- Performance enhancements (recoding some *e* functions to improve speed)
  - Accomplished after chip tape-out using Specman's performance tools
  - Rewrote *events* into *time consuming methods*
  - Removed some list commands, e.g. *pop0()*
  - Used fewer *gen* commands

Conclusion

- 6 months from Verification Concept to ASIC tape-out
- Had basic, useful testbench up and running in 1 month
- Finished within 1 month of original, aggressive schedule
- Over 30,000 jobs run via LSF
- Regression of over 100 tests took 500 CPU hours
- Line coverage of over 95% in 4 months
- Easy port of environment from ASIC to FPGA version of chip (different speeds, multiple FPGAs, different size memory, different number of ports)
- Specman *e* was critical to timeliness and success of project