Agenda

Introduction:
SystemVerilog Motivation
Vassilios Gerousis, Infineon Technologies
Accellera Technical Committee Chair

Session 1:
SystemVerilog for Design

Language Tutorial
Johny Srouji, Intel

User Experience
Matt Maidment, Intel

Session 2:
SystemVerilog for Verification

Language Tutorial
Tom Fitzpatrick, Synopsys

User Experience
Faisal Haque, Verification Central

Lunch: 12:15 – 1:00pm

Session 3: SystemVerilog Assertions

Language Tutorial
Bassam Tabbara, Novas Software

Technology and User Experience
Alon Flaisher, Intel

Using SystemVerilog Assertions and Testbench Together
Jon Michelson, Verification Central

Session 4: SystemVerilog APIs
Doug Warmke, Model Technology

Session 5: SystemVerilog Momentum

Verilog2001 to SystemVerilog
Stuart Sutherland, Sutherland HDL

SystemVerilog Industry Support
Vassilios Gerousis, Infineon

End: 5:00pm

Introducing the new SystemVerilog 3.1 C Interfaces

Doug Warmke
Model Technology Inc.
Material prepared together with
Joao Geada, Synopsys,
Michael Rohleder, Motorola
John Stickley, Mentor Graphics

Outline

- Reasons behind SystemVerilog C API's
- How the standard was developed
- The enhanced SV 3.1 APIs
  - Direct Programming Interface (DPI)
  - Consistent method for loading user C code
  - VPI extensions for Assertions
  - VPI extensions for Coverage
- How it all works: packet router example
- Open Issues and Further Plans

Why SV3.1 Needs New C Interfaces

- Users have long needed a simple way of invoking foreign functions from Verilog and getting results back
- Many users need to call SV functions from C code
- VPI and PLI are not easy interfaces to use
  - Even trivial usage requires detailed knowledge
  - Many users do not need the sophisticated capabilities provided by VPI/PLI.
- SystemVerilog includes assertions. These are a significant addition to the language and were not addressed by any prior Verilog API
- Coverage driven tests have become a well established practice, but no standard mechanism was available to implement such testbenches

How the SystemVerilog C Interfaces Were Developed

- DPI and VPI extensions are based on production proven donations from Synopsys
  - DirectC interface
  - Assertions
  - Coverage
- The SV-CC committee accepted those donations and integrated them into the framework of the SV language
- Foreign code loading mechanism proposed by Motorola

SystemVerilog C Committee

- Representatives of users and vendors
- All major EDA companies are represented

Regularly attending members:

John Amouroux, Mentor
Kevin Cameron, National
João Geada, Synopsys
Ghassan Khoory, Synopsys, *Co-Chair*
Andrzej Litwiniuk, Synopsys
Francoise Martinole, Cadence
Swapanjit Mittra, SGI, *Chair*
Michael Rohleder, Motorola
John Stickley, Mentor
Stuart Swan, Cadence
Bassam Tabbara, Novas
Doug Warmke, Mentor

Other contributing members:

Simon Davidmann, Synopsys
Joe Daniels, LRM Editor
Peter Flake, Synopsys
Emerald Holzwarth, Mentor
Tayung Liu, Novas
Michael McNamara, Verisity
Darryl Parham, Sun
Tarak Parikh, @HDL
Alain Reynaud, Tensilica
Kurt Takara, 0-in
Yatin Trivedi, ASIC Group, *Past Chair*

Overview of DPI

- DPI is a natural inter-language function call interface between SystemVerilog and C/C++
  - Standard allows for other foreign languages in the future
  - DPI relies on C function call conventions and semantics
- Golden Principle of DPI: On each side, the calls look and behave the same as native function calls for that language
- Binary or source code compatible
  - Binary compatible in absence of packed data types (svdpi.h)
  - Source compatible otherwise (svdpi_src.h)

DPI - Declaration Syntax

- Import functions (C functions called from SV):

```
import "DPI" [<dpi_import_property>]
  [c_identifier=] <dpi_function_prototype>;
```

- Export functions (SV functions called from C):

```
export "DPI" [c_identifier=] <dpi_function_identifier>;
```

- Explanation of terms
  - <dpi_function_prototype> same as a native function declaration
  - <dpi_function_identifier> simple name of native function
  - <dpi_import_property> -> *pure* or *context* (more later)
  - c_identifier= is an optional C linkage name
- Declarative Scopes of DPI functions
  - Import declarations -> same scope rules as native SV functions
  - Think of import functions as *native functions implemented in C*
  - Duplicate c_identifiers are not permitted anywhere in the design
    - Import declarations are *not* simple function prototypes
  - Export declarations -> same scope as function definition

Example Import Declarations

```
// The following defines a queue facility implemented in C code.
// SystemVerilog code makes use of it via import functions.
module queuePackage();

// Abstract data structure: queue
import "DPI" function chandle newQueue(input string queueName);
```

```
// The following import function uses the same C function for
// implementation as the prior example, but has a different SV
// name and provides a default value for the argument.
import "DPI" newQueue=
    function chandle newAnonQueue(input string s = null);
```

```
// Functions to round out the queue's interface
import "DPI" function chandle newElem(bit [15:0]);
import "DPI" function void enqueue(chandle queue, chandle elem);
import "DPI" function chandle dequeue(chandle queue);

// More module body items here. Any sequential code in the design
// may use the import functions declared above, just as if they
// were native SV function declarations.

endmodule
```

Example Export Declaration

```
interface ethPort( ... );
  ...

  typedef struct {
      int unsigned packetType;
      int unsigned length;
      longint unsigned dest_addr;
      longint unsigned src_addr;
  } etherHeaderT;

  // The C code will name this export function “SendPacketHeader”
  export "DPI" SendPacketHeader=handlePacketHeader;

  // Returns 32 bit CRC; callable from C code
  function int unsigned handlePacketHeader(
      input etherHeaderT header);
      ...
      return computedCRC;
  endfunction
  ...

endinterface
```

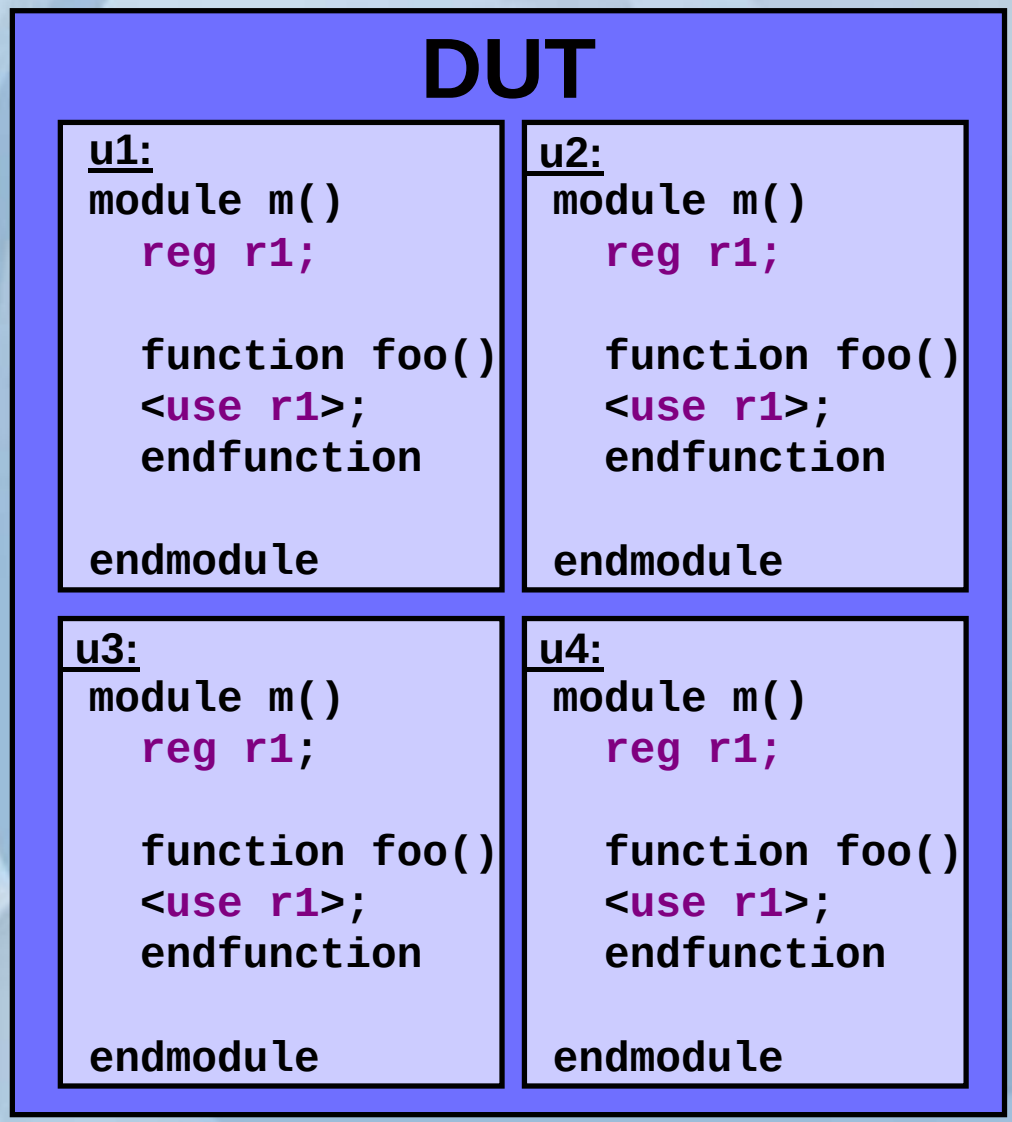
Basics of DPI Programming

- Formal arguments: input, inout, output + return value
  - output arguments are uninitialized
  - passed by value or reference, depending on direction and type
- No DPI functions may contain delay or event controls; thus they complete instantly and consume zero time
- Changes to function arguments become effective when simulation control returns to SV side
- Memory ownership: Each side is responsible for its allocated memory
- Use of *ref* keyword in actual arguments to **import** function calls is not allowed

Import Function Properties

- The *pure* property:
  - is useful for compiler optimizations
  - has no side effects or state (I/O, global variables, PLI/VPI)
  - result depends solely on inputs, optimizer might cache results
- The *context* property:
  - is useful when modeling system components
  - works with data specific to the enclosing module instance
  - is mandatory when PLI/VPI calls are used within the function
  - is mandatory when an **import** function calls an **export** function
- Free functions (non-context): no relation to instance-specific data
  - Useful for doing calculations, i/o operations, numerical work, etc.
- Context import functions are bound to a particular SV instance
- All export functions are “context” functions

What Does “Context” Mean?



One definition of module m

\+ One declaration of function foo

\+ Four different instances of module m

= Function foo runs in four different *contexts*

(each call to foo uses a different r1)

DPI Context Functions

- Simulator keeps track of context during function calls
  - Exact same as native SV function calls
  - The terms *context* and *scope* are used equivalently here
- Allows interleaved call chains, e.g. SV-C-SV-C
- Context is needed for C to call SV export functions
  - Simulator sets default context at each context import function call
  - User can override default context using svSetScope()
- User data storage is available for each scope
  - Similar to userData concept of VPI, but slightly more powerful
    - Multiple independent DPI apps can store user data with no clash
  - Can store instance-specific data for fast runtime retrieval by context import functions
    - Useful for avoiding runtime hashing in C code

Argument Passing in DPI

- Supports most SV data types
- Value passing requires matching type definitions
  - user's responsibility
  - packed types: arrays (defined), structures, unions
  - arrays (see next slide)
- Function result types are restricted to small values and packed bit arrays up to 32 bits
- Usage of packed types might prohibit binary compatibility

SV type	C type
byte	char
shortint	short int
int	Int (32-bit)
longint	long long
real	double
shortreal	float
chandle	void*
string	char*
bit	(abstract)
enum	int
logic	avalue/bvalue
packed array	(abstract)
unpacked array	(abstract)

Choosing DPI argument types

- C types, such as *int* and *double* (scalars and composites)
  - Think of these as “software types”
  - Efficient for performance
  - Straightforward to use (“API-less” on C side)
  - May require more cumbersome programming on SV side
- Non-C types, *bit* and *logic* (scalars and composites)
  - Think of these as “hardware types”: wire, reg, composites thereof)
  - Convenient for interfacing to legacy Verilog
  - Convenient for interfacing to hardware constructs
  - Requires more cumbersome programming on C side
  - Binary and source compatibility issues
  - May degrade performance in some cases

=> All things being equal, prefer C types

DPI Array Arguments

- There are three types of array to consider
  - Packed array (elements of SV types “bit” or “logic”)
  - Unpacked array (elements of C-compatible types)
  - Open array (array bounds not statically known to C)

- Arrays use normalized ranges for the packed [n-1:0] and the unpacked part [0:n-1]

  For example, if SV code defines an array as follows:

```
logic [2:3][1:3][2:0] b [1:10][31:0];
```

  Then C code would see it as defined like this:

```
logic [17:0] b [0:9][0:31];
```

Open Array Arguments

- Open Array arguments have an unspecified range for at least one dimension
  - Good for generic programming, since C language doesn't have concept of parameterizable arguments
  - Denoted by using dynamic array syntax [] in the function declaration
  - Elements can be accessed in C using the same range indexing that is used for the SV actual argument
  - Query functions are provided to determine array info
  - Library functions are provided for accessing the array
- Examples:

```
logic [] \1x3 [3:1];
bit [] unsized_array [];
```

Some Example Uses of DPI

- Value calculations done in C
  - FFT, other numerical or crunching work
- Complex I/O processing done in C
  - Stimulus-fetching socket, custom file i/o, etc.
- Test executives running in C
  - Call export functions to kick design into action; rely on import functions for response
- Complex multi-language modeling
  - Connect to SystemC or other multi-threaded environments running a portion of the verification

Consistent Load of User C Code

- Only applies to DPI functions, PLI/VPI not supported (yet)
- All functions must be provided within a shared library
  - User is responsible for compilation and linking of this library
  - SV application is responsible for loading and integration of this library
- Libraries can be specified by switch or in a bootstrap file
  - -sv_lib <filename w/o ext>
  - -sv_liblist <bootstrap>
  - extension is OS dependent; to be determined by the SV application
- Uses relative pathnames
  - -sv_root defines prefix

```
#!SV_LIBRARIES
# Bootstrap file
# containing names
# of libraries to
# be included
function_set1
common/clib2
myclib
```

VPI Extensions for Assertions

- Permits 3rd party assertion debug applications
  - Usable across all SV implementations
- Permits users to develop custom assertion control, response, and reporting mechanisms

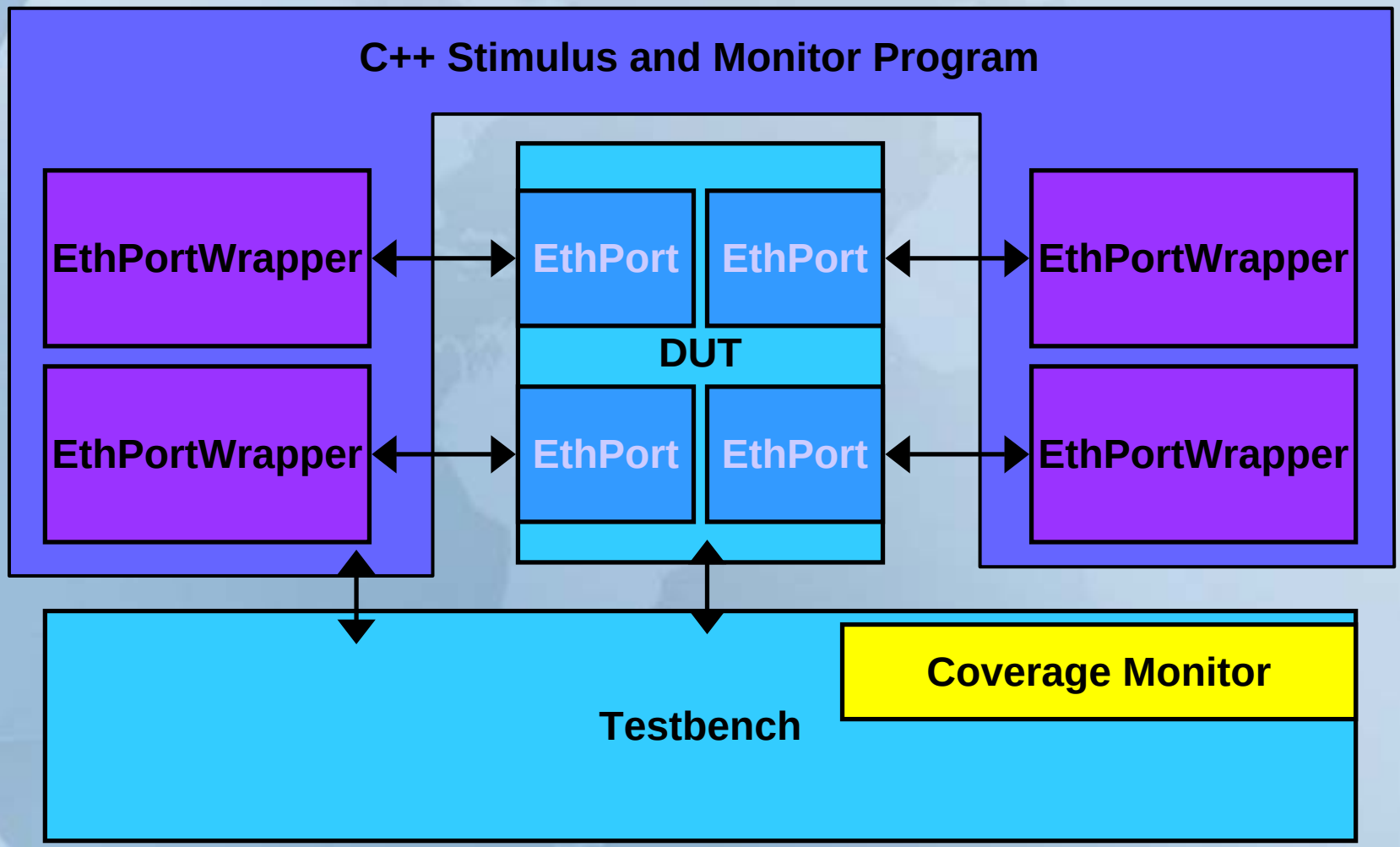
VPI for Assertions: Overview

- Iterate over all assertions in an instance or the design
- Put callbacks on an assertion
  - Success
  - Failure
  - Step
- Obtain information about an assertion
  - Location of assertion definition in source code
  - Signals/expressions referenced in assertion
  - Clocking signal/expression used in assertion
  - Assertion name, directive and related instance, module
- Control assertions
  - Reset: discard all current attempts, leave assertion enabled
  - Disable: stop any new attempts from starting
  - Enable: restart a stopped assertion

Coverage Extensions

- Standardized definition for a number of coverage types
  - Statement, toggle, FSM state and assertion coverage defined
  - For these coverages, coverage data has same semantics across all implementations
- Defines 5 system tasks to control coverage and to obtain “realtime” coverage data from the simulator
  - $coverage_control, $coverage_get_max, $coverage_get, $coverage_merge, $coverage_save
  - Interface designed to be extensible to future coverage metrics without perturbing existing usage
  - Coverage controls permit coverage to be started, stopped or queried for a specific metric in a specific hierarchy of the design
- VPI extensions for coverage provide same capabilities as the system tasks above, plus additional “fine-grain” coverage query
  - Coverage can be obtained from a statement handle, FSM handle, FSM state handle, signal handle, assertion handle

Ethernet Packet Router Example



Example developed by John Stickley, Mentor Graphics

C++ Side: SystemC Testbench

```
 1  SC_MODULE(TestBench) {
 2      private:
 3          EthPortWrapper* context1;
 4          EthPortWrapper* context2;
 5          EthPortWrapper* context3;
 6          EthPortWrapper* context4;
 7          int numOutputs;
 8
 9          void testThread(); // Main test driver thread.
10      public:
11          SC_CTOR(System) : numOutputs(0) {
12
13              SC_THREAD(testThread);
14              sensitive << UTick;
15
16              // Construct 4 instances of reusable EthPortWrapper
17              // class for each of 4 different HDL module instances.
18              context1 = new EthPortWrapper("c1"); context1->Bind("top.u1", this);
19              context2 = new EthPortWrapper("c2"); context2->Bind("top.u2", this);
20              context3 = new EthPortWrapper("c3"); context3->Bind("top.u3", this);
21              context4 = new EthPortWrapper("c4"); context4->Bind("top.u4", this);
22          }
23          void BumpNumOutputs() { numOutputs++; }
24  };
25
26  void TestBench::testThread() {
27      // Now run a test that sends random packets to each input port.
28      context1->PutPacket(generateRandomPayload());
29      context2->PutPacket(generateRandomPayload());
30      context3->PutPacket(generateRandomPayload());
31      context4->PutPacket(generateRandomPayload());
32
33      while (numOutputs < 4) // Wait until all 4 packets have been received.
34          sc_wait();
35  }
```

C++: SystemC EthPortWrapper

```
 1  #include “svc.h”
 2
 3  SC_MODULE(EthPortWrapper) {
 4      private:
 5          svScope myContext;
 6          sc_module* myParent;
 7      public:
 8          SC_CTOR(EthPortWrapper) : svContext(0), myParent(0) { }
 9          void Bind(const char* hdlPath, sc_module* parent);
10          void PutPacket(vec32* packet);
11
12      friend void HandleOutputPacket(svHandle context,
13          int portID, vec32* payload);
14  };
15
16  void EthPortWrapper::Bind(const char* svInstancePath, sc_module* parent) {
17      myParent = parent;
18      myContext = svGetScopeFromName(svInstancePath);
19      svPutUserData(myContext, (void*)&HandleOutputPacket, (void*)this);
20  }
21
22  void EthPortWrapper::PutPacket(vec32* packet) {
23      svSetScope(myContext);
24      PutPacket(packet); // Call SV function.
25  }
26
27  void HandleOutputPacket(int portID, vec32* payload) {
28      svScope myContext = svGetScope();
29
30      // Cast stored data into a C++ object pointer
31      EthPortWrapper* me = (EthPortWrapper*)svGetUserData(myContext,
32           (void*)&HandleOutputPacket);
33
34      // Let top level know another packet received.
35      me->myParent->BumpNumOutputs();
36
37      printf("Received output on port on port %\n", portID);
38      me->DumpPayload(payload);
39  }
```

SV-side: SV EthPort Module #1

```
 1
 2  module EthPort(
 3      input [7:0] MiiOutData,
 4      input MiiOutEnable,
 5      input MiiOutError,
 6      input clk, reset,
 7      output bit [7:0] MiiInData,
 8      output bit MiiInEnable,
 9      output bit MiiInError);
10
11      import "DPI" context void HandleOutputPacket(
12          input integer portID,
13          input bit [1439:0] payload);
14
15      export "DPI" void PutPacket;
16
17      bit inputPacketReceivedFlag;
18      bit [1499:0] inputPacketData;
19
20      //
21      // This export function is called by the C side
22      // to send packets into the simulation.
23      //
24      function void PutPacket(input bit [1499:0] packet)
25          inputPacketData = packet;
26          inputPacketReceivedFlag = 1;
27      endfunction
28
```

SV side: SV EthPort module #2

```
29      always @(posedge clk) begin      // input packet FSM
30          if (reset) begin
31              ...
32          end
33          else begin
34              if (instate == READY) begin
35                  if (inputPacketReceived)  // Flag set by C call to export func.
36                      instate <= PROCESS_INPUT_PACKET;
37              end
38              else if (instate == PROCESS_INPUT_PACKET) begin
39                  // Start processing inputPacketData byte by byte ...
40              end
41          end
42      end
43
44      always @(posedge clk) begin     // output packet FSM
45          if (reset) begin
46              ...
47          end
48          else begin
49              if (outstate == READY) begin
50                  if (MiiOutEnable)
51                      outstate <= PROCESS_OUTPUT_PACKET;
52              end
53              else if (outstate == PROCESS_OUTPUT_PACKET) begin
54                  // Start assembling output packet byte by byte ...
55                  ...
56                  // Make call to C side to handle the assembled packet.
57                  HandleOutputPacket(myPortID, outPacketVector);
58              end
59          end
60      end
61  endmodule
```

SV side: Coverage monitor

```
module coverage_monitor(input clk)
  int cov = 0, new_cov = 0, no_improvement = 0;

  always @(posedge clk) begin
     // count clocks and trigger coverage monitor when appropriate
  end

  always @(sample_coverage) begin
    // get the current FSM state coverage in the DUT and all instances below
    new_cov = $coverage_get(`SV_COV_FSM_STATE,
                    `SV_HIER, "DUT");
    if (new_cov <= cov) begin
          // no coverage improvement
          no_improvement++
          if (no_improvement == 3) $finish();
    end
    else begin
          // coverage still increasing. Good!
          cov = new_cov;
    end
  end
endmodule
```

Open Issues and Further Plans

- Extend VPI object model to support the complete SV type system
  - extend VPI to cover all new elements of SystemVerilog
- Additional callback functions to match enhanced scheduling semantics
- Further enhancements to loading/linking
  - inclusion of source code, uniform PLI/VPI registration
- Extending DPI to handle SV tasks
- All driven by experiences and user requests/needs