USING SYSTEMVERILOG NOW WITH DPI

Rich Edelman, Mentor Graphics, San Jose, CA (rich_edelman@mentor.com)
Doug Warmke, Mentor Graphics, San Jose, CA (doug_warmke@mentor.com)

Abstract

In this paper, we describe some applications of the SystemVerilog 3.1a [1] Direct Programming Interface (DPI) which increase verification efficiency and can be added to existing verification methodologies. In particular we will focus on methodologies that already use C as part of the test environment or methodologies targeting hardware emulation or acceleration with C tests reused across abstraction levels.

Introduction

In this paper we'll demonstrate a number of ways that SystemVerilog DPI can be used to increase verification efficiency and productivity by allowing easy integration of new or existing C code with existing Verilog code, and by allowing test re-use across modeling abstraction levels.

For this paper, we consider SystemVerilog a superset of Verilog. Techniques are described that add SystemVerilog DPI constructs to enable more productive verification.

Using some small examples and pseudocode, we will demonstrate verification techniques using SystemVerilog DPI; passing data and returning data between C and SystemVerilog, and using time-consuming C tasks.

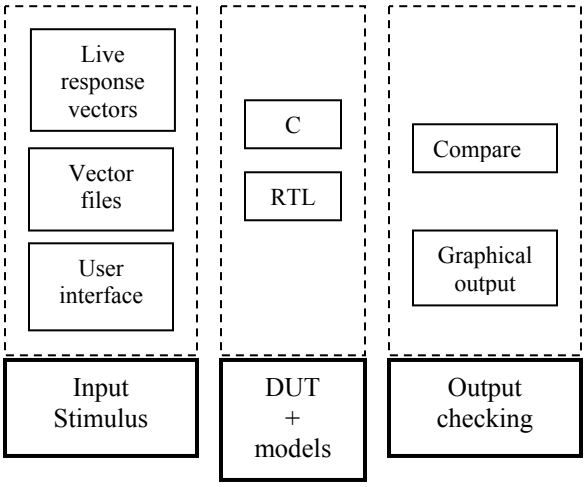


Figure 1 - General block diagram

Eight ways of using DPI to achieve various verification improvements are listed, with example C and Verilog snippets, and any issues with each usage. Complete examples are available from the authors.

We cover traditional techniques for simulation checking such as reading inputs and expected outputs from a file and using a live model to compare DUT results. Additionally, we'll cover using C code to write tests – including a graphical user interface, simple function calls to hardware routines, and using standard C libraries as part of the test harness. Such libraries can be used either for golden results calculations or as utilities like timers. In **Figure 1**, each block can be modeled in C or SystemVerilog, with the communication between blocks modeled as a function call interface.

Direct Programming Interface

SystemVerilog Direct Programming Interface (DPI) 3.1a defines a function call interface between C and Verilog. This interface has evolved from contributions to the Accellera standards committee including the Synopsys VCS DirectC Interface and the Co-Design Cblend interface. Additional standardization is currently in progress with the IEEE P1800 standard.

The well-known Verilog PLI and VPI interfaces are not function call interfaces, but rather are closely related to simulator internals. PLI and VPI are important interfaces that allow powerful applications to be created, but they can be hard to use. SystemVerilog supports the PLI, VPI and DPI interfaces. [4] provides a broad discussion of the relationship between PLI, VPI and DPI, and the advantages and disadvantages of each. The remainder of this paper will only discuss DPI.

DPI is designed to be easy to use, and to be fast. With DPI, C code is compiled into a shared object. The shared object is loaded during simulation, and provides a collection of *imported* interface functions that are indistinguishable from traditional Verilog tasks and functions. In addition to the *imported* interface functions, Verilog supplies a collection of *exported* interface functions that the C code may call. To the calling C code, these exported Verilog interface functions are indistinguishable from other C functions.

Datatypes

SystemVerilog DPI passes information across the SystemVerilog/C function call boundary by using well-known C datatypes, like *char, short, int, long long, float,*

double or aggregates of those types. Four-state SystemVerilog types are also available.

Table 1 - SystemVerilog DPI - Scalar - Small Values

C Data Type	SystemVerilog Data Type
char	byte
short	shortint
int	int
long long	longint
float	shortreal
double	real
void *	chandle
const char *	string
unsigned int	bit
unsigned int	logic

In general C is good at manipulating two-state data, and SystemVerilog is good at manipulating four-state data (i.e. scalars and vectors of **logic** data type). Each language is best used where it is good. Use C for two-state data, and only pass data back and forth with C that is of a two-state data type. A SystemVerilog compiler can convert between two and four state data very efficiently.

C can be used to manipulate four-state data, but additional conversions are required on the C side. These macros, access functions and conversion routines can be found in the SystemVerilog LRM, and are beyond the scope of this paper.

Function Arguments

In SystemVerilog, tasks and functions can have *input*, *inout*, or *output* arguments. This idea is carried over to DPI, where *inputs* are passed to and from C as values, (unless they are "large", such as a struct or array, in which case they are passed by const reference), and *inouts* and *outputs* are passed back and forth as references. Additionally, values can be passed back and forth via the return value of a function.

The C code below is *callable* from SystemVerilog. It will be passed an integer, and two addresses of integers.

```
SV : import "DPI" task t_impC(
       input int i, inout int io, output int o);
 C : int t_impC(int i, int *io, int *o);
```

The C code below *calls* SystemVerilog; passing "5", the address of "io" and the address of "o".

```
SV : export "DPI" task t_expVL;
     task t_expC(
       input int i, inout int io, output int o);
       ...
     endtask
C : c_code() {
      int io, o;
      ...
      t_expC(5, &io, &o);
      ...
    }
```

Imports and Exports

The *import* and *export* statements define the SystemVerilog and C interface functions. Import is used to "import" C code (interface functions) into SystemVerilog. Once imported, the C function is callable by SystemVerilog like any regular task or function.

Export is used to "export" SystemVerilog tasks or functions to C. Once exported, the SystemVerilog tasks or functions are callable by C like any regular C function.

In Figure 2 SystemVerilog task *vl_task()* is defined to have two input parameters, *inp1* and *inp2,* and return an output parameter *result.* Line 4 *exports* the task interface, thus making it available as an entry point for the C function call in Figure 3, Line 4.

```
 1  module top;
 2   import "DPI" context task c_test(
 3     output int c_result);
 4   export "DPI" task vl_task;
 5   task vl_task(
 6    input int inp1,
 7    input int inp2,
 8    output int result);
 9    result = inp1 + inp2;
10   endtask

11   initial begin
12     c_test(answer);
13     $display("VL: Answer=%0d", answer);
14   end
15  endmodule
```

Figure 2 - Exported SystemVerilog Code

In Figure 3 function *c_test()* is defined to return a parameter value, and a return code. At line 4, function *vl_task()* is called with three arguments – two integers and the address of an integer. These arguments correspond to the arguments in the task definition in Figure 2, lines 5-8.

In this paper, Verilog tasks and functions will be referred to variously as tasks or functions. Tasks and functions called directly from C will be referred to as interface tasks and functions. Tasks and functions have different rules of usage

in the DPI standard, but for this paper they are interchangeable.

In order for C to communicate with SystemVerilog it simply calls a function. From the C point of view, it has called a function, and it expects a normal return value (if one is defined), along with values for any returned arguments.

```
1  int
2  c_test(int *c_answer) {
3   int inp1 = 10, inp2 = 20, vl_answer;
4   vl_task(inp1, inp2, &vl_answer);
5   print(" C: Answer = %d\n", vl_answer);
6   *c_answer = vl_answer * 2;
7   return 0;
8  }
```

Figure 3 - Imported C Code

It is important to note that with a few limitations, C functions and SystemVerilog tasks and functions become interchangeable, and can be used as replacements for each other.

Time Consuming C Code

SystemVerilog tasks can consume time. This is a powerful mechanism since a C call to a time-consuming SystemVerilog task will itself consume time. Using this method, it is possible to write C tests that call back into SystemVerilog to wait for time, wait for events or wait for variable changes (like a clock edge).

Figure 5 shows a C routine that calls a SystemVerilog task *hw()*, which in Figure 4 will wait for a positive clock edge, then a negative clock, and finally for an additional 100 simulation units.

```
module DUT(reg clk, …);
  reg [31:0]dut_x;
  wire [31:0]dut_cnt;
  export "DPI" task hw;
  task hw(input int x, output int y)
    @(posedge clk);
    dut_x = x;
    @(negedge clk);
    y = dut_cnt;
    #100 ;
  endtask
  ...
endmodule
```

Figure 4 - DUT for C tests that consume time

```
int
c_test() {
  int x = 5, y ;
  hw(x, &y);
  return 0;
}
```

Figure 5 - C code that will consume time

In this example, when the C code calls *hw()*, and *hw()* waits for the clock, its execution is suspended until the clock changes. At this time, SystemVerilog may begin execution of another thread.

Threads

Threads are the way the SystemVerilog manages parallel execution. A DPI programmer needs to be aware of the threaded environment, since the imported C code may be called from multiple threads.

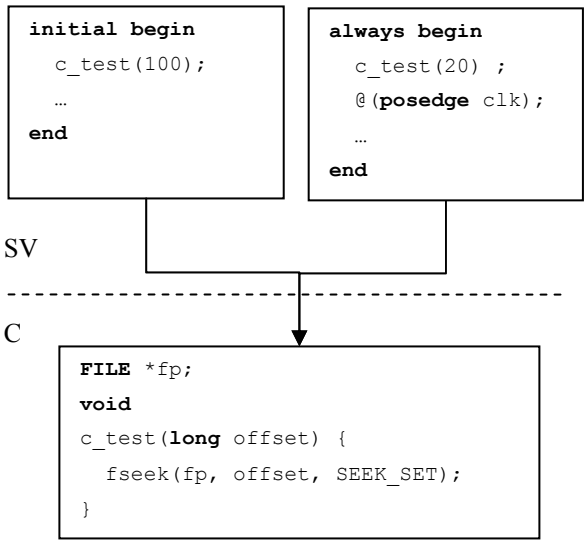


Figure 6 - Threaded software

Specifically, a DPI programmer needs to be aware of traditional "thread-safety" issues. That is, all C and SystemVerilog tasks and functions that are called from multiple threads must be written in a reentrant style. There is nothing special about SystemVerilog threads in this regard. It is important to be aware of this thread-safety issue. Threaded programming is not the same as non-threaded programming.

Thread safety issues usually arise when a "singleton resource" like a static variable, an I/O device or global data becomes a shared resource because there are two occurrences of a function call active at the same time.

Note that DPI C code must only be called from within SystemVerilog procedural code. It is an error to call DPI code from within PLI callbacks, or other custom threading systems available in the C programming environment (e.g. pthreads or QuickThreads).

Figure 6 shows two threads started in SystemVerilog. Each thread calls the C function c_test().

The C function c_test() uses a global variable to keep track of an open file. In this example, the function c_test() will be called twice. Once each from the two SystemVerilog threads. One starts from the *initial* block, the other starts from the *always* block. The file pointer will be positioned to either 100 or 20. This is not the intended behavior. This code needs to be rewritten in order to serialize the file positioning instructions.

Creating a Task / Function Interface

An existing Verilog DUT can interact with C using SystemVerilog DPI, but you must first create the required SystemVerilog task/function interfaces. For example, a *read* may be executed from C by calling a read task in SystemVerilog. The read task must perform the appropriate steps in order to actually execute the read from the hardware.

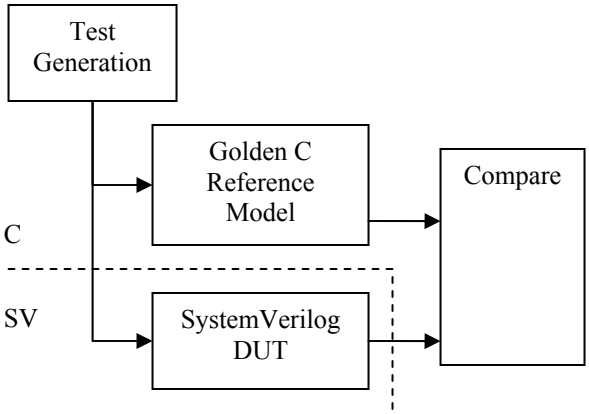
Once created, SystemVerilog tasks and functions are exported for possible use as entry points by C. These entry points become an additional interface to the DUT.

See Figure 4 for an example of wrapping a lower level "pin wiggling" interface with a task interface. *hw()* is implemented by assigning the task argument *x* to the DUT pins, and running a clock cycle, and then returning the *dut_cnt* as a task output argument.

With a task/function interface defined for the DUT, we can write complex C code that models our simulation environment, or drives tests. The C code thinks it is calling a collection of functions that perform certain jobs. It has no idea that the underlying function implementation is actually modeled in a hardware description language and is simulating real hardware behavior, including timing.

C Code as Golden Reference

During simulation, verification can be performed against a known good *golden* result. The RTL and the golden C model operate in parallel – and have their inputs connected to the same test inputs. The outputs of the RTL model and the C model are connected to a compare function. As each new output is generated by the DUT, the result is compared with a golden output generated from the golden C reference model.



Pseudo Code for C as golden model

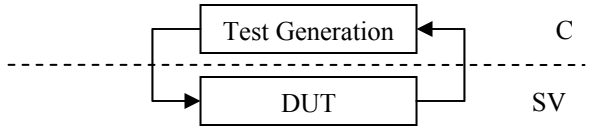
```
forall (inputs) {
  C_func(inputs, &c_results)
  VL_func(inputs, &vl_results)
  if (c_results != vl_results) {
    error()
  }
}
```

Figure 7 - Golden reference and compare

See the Appendix for a complete C and SystemVerilog example of a golden C model.

C Code as Testbench Stimulus – Live vectors

A reactive testbench is smart, and can "drive" the test to certain interesting points. Typical reactive testbenches generate random or directed random stimulus using a heuristic until a desired coverage metric is reached.



Pseudocode for C Test Generation

```
until_done() {
  Apply inputs to DUT via task call.
  Run clocks in DUT, wait for results.
  Collect results via task call from DUT.
  Based on results, generate new inputs.
}
```

Figure 8 - Test generation feedback loop

Once the coverage metric is reached, the heuristic can be changed to avoid unproductive simulation cycles. [2] describes testbench architectures, examples and techniques for reactive testbench usage.

Any test which produces a measure like coverage, or congestion, or error packets can be used in a reactive testbench. As the measure reaches a desired level, the test is changed to achieve a higher measure.

C Code as Testbench Stimulus – Software

A system is described in C; calling functions to perform various operations. The underlying functions can be implemented as C or as SystemVerilog tasks or functions that interact with hardware models. The C system software doesn't have any indication that the function calls are implemented in SystemVerilog – the hardware implementation is transparent to the running software.

```
task pci_transaction(inout pcicmd_t cmd);
  // Perform specified bus transaction
  case (cmd.cb_e)
    `PCI_CFGREAD: begin
      pci_frame = 1'b0;
      @(posedge pci_clk);
      pci_irdy = 1'b0;
      @(posedge pci_clk);
      ...
      cmd.databuffer = pci_data;
    end
    `PCI_CFGWRITE: begin
      ...
   endcase
endtask
```

Figure 9 – PCI SV configuration task

For example a design under test has a well-defined pin interface, and we have written Verilog tasks (i.e. `pci_transaction()`, below) to manipulate this interface. Our testbench is written in C, and calls the Verilog tasks to manipulate the DUT.

```
void
c_test() {
  struct pcicmd_t cmd;
  cmd.addr = 0x30;
  cmd.cb_e = PCI_CFGREAD;
  pci_transaction(&cmd);
}
```

Figure 10 – PCI C configuration code

Using the defined hardware task interface, we can write software on the C side, ranging from assembly style register or memory manipulation, to configuration, to complete software running on a virtual hardware interface.

C Code as User Interface

Some verification tasks require a graphical user interface (GUI). Such a GUI can be used either to drive the simulation directly, or as an indirect interface that allows monitoring of the simulation with enough detail to understand progress, but at a high enough level to avoid performance problems. For example, a GUI progress bar could be created that reports the current coverage metric, updated every 100,000 clock cycles.

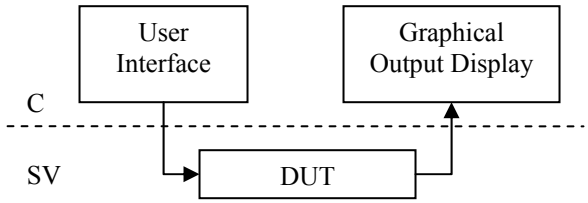


Figure 11 - User interface input and graphical output

Many tests require some user interaction to configure. A user interface with pull-down menus, and other ways to change configuration can be used with SystemVerilog DPI. Once the configuration parameters are set via the user interface, the hardware is configured, and the test can begin.

```
int c_test() {
  while(!feof(stdin)) {
    fscanf(stdin, "%d %d", &x, &y);
    newimage(x, y) ;
  }
  return 0;
}
int c_draw(int x, int y, int color) {
  //… X Windows Drawing Commands…
  return 0;
}
```

Figure 12 - Using C as a user interface

Figure 12 illustrates a C routine that reads a pair of coordinates, *x* and *y* from *stdin*. The coordinates read are passed to the hardware routine, *newimage()* in Figure 13 for algorithmic image generation. As the new pixels are calculated by the hardware, they are passed to the C drawing routine, *c_draw()*.

For graphical output, data can be drawn directly into any common drawing package, such as *EGGX*, *X11* or *Tcl/Tk*.

```
module DUT(…);
  import "DPI" task c_test();
  import "DPI" task c_draw(
                    input int x, y, color);
  ...
  int pixel_x, pixel_y, pixel_color;
  export "DPI" task newcalc;
  task newimage(input int x, y);
    // Setup new image calculation by
    //   assigning x, y to control
    //   registers. Run clocks,
    //   generating 640x480 pixels.
  endtask;
  always (...) begin
    @(posedge pixel_done);
    c_draw(pixel_x, pixel_y, pixel_color);
    ...
  end
```

Figure 13 – DUT with C user interface

C Code as External Model

Using C to model external hardware components can allow models to be developed quickly. Architecture exploration can be done with less time and effort.

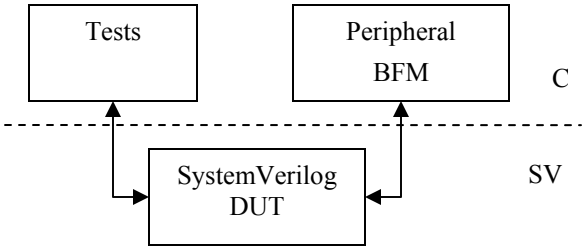


Figure 14 - Modeling external peripherals with C

C can be used to model a complex BFM that provides enough functionality to allow further testing with minimal effort.

```
int mem[4096];
int c_store(int addr, int data) {
  mem[addr & 0x7ff] = data;
  return 0;
}
int c_retrieve(int addr) {
  return mem[addr & 0x07ff];
}
```

Figure 15 - Memory modeled in C

```
module DUT(…);
  import "DPI" task c_store(
                    input int addr, data);
  import "DPI" function int c_retrieve(
                    input int addr);
  int addr, data;
  // Run tests.
  ...
  always (...) begin
    data = c_retrieve(addr) + 1;
    @(posedge ready);
    c_store(addr, data);
    ...
  end
```

Figure 16 - DUT using external C memory model

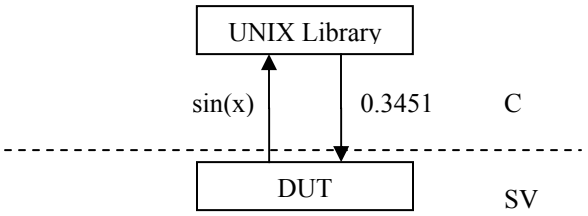
With the BFM implemented, experiments with memory architectures and bus configurations can be built.

Figure 15 demonstrates a simple C implementation of a memory modelled with READ and WRITE as *c_retrieve()* and *c_store()*.

C Code as Utility Library

With DPI, C code can be called directly from Verilog. Such calls include standard C library utilities, like socket(), gettimer(), et al. The C functions must support DPI datatypes, in order to be called, but this requirement is not a serious one, since the DPI datatypes include *char, short, int, long long, float, double, struct,* and *arrays*. Additional SystemVerilog datatypes, including bit and logic vectors, (packed vectors) are not important for standard C library calls. C libraries are not aware of these simulation specific datatypes.

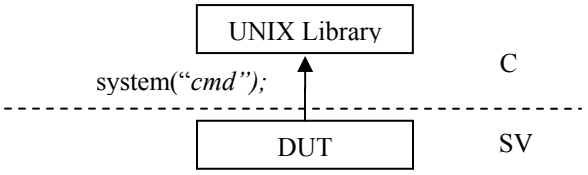
Any C library adhering to normal compilation rules and supporting the SystemVerilog DPI datatypes can be called as a DPI import; for example, *log(), exp(), sin(), cos()* and *getrusage()* can be used to add functionality and improve verification power by directly using math functions, or collecting process resource usage.




```
module DUT(...);
  import "DPI" real sin(input real a);
  ...
  real x, y;
  always (…) begin
    y = sin(x);
    ...
  end
```

Figure 17 - Using the std C math library from SV

Using standard C library calls such as *system()*, can add improved functionality to SystemVerilog models.




```
module DUT(...);
  import "DPI" function
         int system(input string s);
  initial begin
    system("cp /tmp/f1 /tmp/f2");
    ...
  end
```

Figure 18 – SV Calling UNIX *system()*

Additionally, simulation can produce a stream of data that is processed by an external reader.

For example, SystemVerilog DPI can implement the equivalent of the UNIX *system()* call – which can be used to launch an external graphics viewer, such as *xv* or *IrfanView*. The stream of data from the simulation can be viewed immediately by the external viewer program.

C Code As Testbench Stimulus - Vector File

Some existing methodologies read inputs and expected outputs from a compressed text file. Inputs are applied to the DUT, and expected outputs and actual outputs are compared after a fixed number of clocks or time period. Verilog has been used for this kind of testbench for years; by using C, this methodology can be made more flexible.

```
int
c_test() {
  // Open compressed file using 'zlib'
  byte_stream =
     open_compressed("vecfile.txt.gz");
  while(1) {
   readvec(byte_stream, 1024, inputs);
   readvec(byte_stream, 1024,
                    expected_outputs);
   // Convert 01xz to legal SV 4-state
   ...
   hw_apply(inputs, expected_outputs);
  }
}
```

Figure 19 - C code opening compressed vector file

```
module DUT (...);
  // in1,in2,... are input pins to the
  //  chip under test - not defined here.
  task hw_apply(
      input logic [1024]inp_vec,
      input logic [1024]exp_vec));
    //Apply inputs.
    {in1,in2,...} = inp_vec;
    @(posedge clk);
    //Run half a clock
    @(negedge clk);
    //Compare
    if ({in1,in2,...} != exp_vec) begin
      ...
    end
  endtask
  initial begin
    c_test();
  end
  ...
endmodule
```

```
                    uncompressed vecfile.txt
1011001001...  1000001000000111000000...
1011001000...  1000001000011111000000...
...
```

Figure 20 - SV DUT and uncompressed test vectors

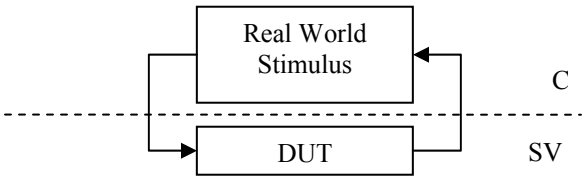
In the *c_test()* above, *zlib* is used to create a stream of bytes from a compressed vector file. The byte stream is "read" as

lines of input and expected outputs. The vectors are converted to four-state logic by DPI conversion routines and at each clock cycle, an input is applied to the hardware via the *hw_apply()* task. The actual outputs are sampled on the negative clock, then compared to the expected outputs. The DPI logic conversion routines are beyond the scope of this paper.

C Code as Testbench Stimulus - Real world stimulus

The testbench stimulus for many hardware designs is actually "real" data; JPEG or MP3 decoders can directly use JPEG or MP3 files as stimulus. Network traffic can be used directly by network infrastructure hardware models.

With SystemVerilog DPI, the SystemVerilog DUT can call a C function to read the next byte or line from the stimulus, thereby reading the JPEG or MP3.



Pseudocode for C

```
Read MP3 file, creating a byte stream.
until_done() {
  Supply byte stream to DUT via task call.
  Run clocks, wait for results.
}
```

There are many such standard formats supported by open source software readers and writers. SystemVerilog DPI enables these tools to be leveraged and pulled into the verification methodology with little effort – providing a way to supply "real world" stimulus to the DUT.

Re-using Tests Across Abstraction Levels

When using a SystemVerilog DPI function call interface, as lower level implementations are synthesized or otherwise refined, high level tests can be reused with the addition of a function call adapter layer to step-up or step-down the communication between function call parameters and pin wiggles.

For example, modeling can begin in C. Models are then migrated to SystemVerilog behavioral code, then to SystemVerilog RTL, Gates, and perhaps even emulated gates. At each stage the same high level tests can be run. These *transactors* are beyond the scope of this paper.

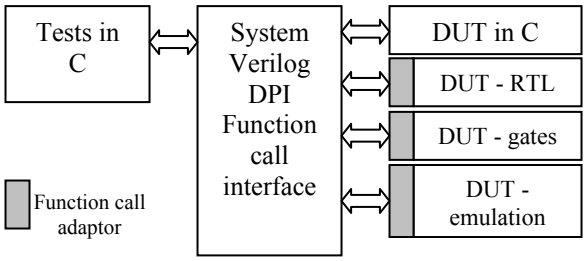


Figure 21 - Re-using high level tests

Conclusions

SystemVerilog and the Direct Programming Interface can be used to augment existing Verilog verification environments, especially ones already using C.

We have demonstrated techniques for writing testbenches in C, and then re-using them as the DUT is refined into RTL and gates. We have demonstrated traditional vector file input with expected outputs along with golden model comparison. Additionally, we have shown techniques to connect a graphical user interface to a DUT.

Finally, these techniques can be used together. For example a user interface can drive the DUT and the golden model, and then a comparison can be performed.

SystemVerilog DPI provides powerful techniques to improve the efficiency and productivity of existing verification processes.

References

1. SystemVerilog 3.1a LRM (www.systemverilog.org)
2. Janick Bergeron, "Writing Testbenches – Functional Verification of HDL Models", Kluwer Academic Publishing (February 2003).
3. Cliff Cummings, "SystemVerilog Basic Training", (systemverilog.org/pdf/SV_Symposium_2003.pdf)
4. Stuart Sutherland, "The Verilog PLI Is Dead (maybe) Long Live The System Verilog DPI!", SNUG San Jose 2004 (sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf)
5. Stuart Sutherland, "Integrating SystemC Models with Verilog and SystemVerilog Models Using the SystemVerilog Direct Programming Interface", SNUG Europe 2004 (sutherland-hdl.com/papers/2004-SNUG-paper_Verilog_PLI_versus_SystemVerilog_DPI.pdf)
6. DAC2003 DPI Overview (systemverilog.org/pdf/4_DPIOverview.pdf)

Appendix

Full Example for "C as GOLDEN model"

ccode.c

```
#include <stdio.h>
#include "svdpi.h"
#include "dpiheader.h"
#define MAX 1000

int
c_test() {
    int inp1, inp2, c_answer, vl_answer;

    vpi_printf("Running\n");
    for(inp1 = 0; inp1 < MAX; inp1++) {
      for(inp2 = 0; inp2 < MAX; inp2++) {

        c_answer = inp1 * inp2;
        vl_task(inp1, inp2, &vl_answer);

        if (c_answer != vl_answer) {
          vpi_printf(
"Error: MISMATCH (%d, %d) vl<%d> != c<%d>\n",
            inp1, inp2, vl_answer, c_answer);
        }
      }
    }
    vpi_printf("...done.\n");
    return 0;
}
```

vlcode.v

```
module top;
  reg clk;

  initial begin
      c_test();
      $finish;
  end

  import "DPI" context task c_test();
  export "DPI" task vl_task;
  task vl_task(
    input int inp1,
    input int inp2,
    output int result);
      int n;
      result = 0;
      @(posedge clk);
      for(n = 0; n < 32; n++) begin
        if (inp2 & 1'b1) begin
           result += inp1;
        end
        inp1 <<= 1;
        inp2 >>= 1;
        @(posedge clk);
      end
  endtask

  always begin
    clk = 0; #50; clk = 1; #50;
  end
endmodule
```

EXECUTION FLOW

1 Simulation in **initial** block
2 Call to C code *c_test()*.
3 Call to VL code *vl_task()*.
4 Wait for a clk edge.
5 Multiply, then wait for a clk edge.
6 Return results via **output** argument.
7 Check results, return to main *c_test()*
8 Return to SystemVerilog. Test complete.
9 *$finish(). Exit.*

```
# Using ModelSim 6.0
vlib work
vlog -dpiheader dpiheader.h -sv vlcode.v
gcc -fPIC -shared -I$(MTI_HOME)/include -I. -o ccode.so ccode.c
vsim -c -sv_lib ccode top -do "run -all; quit -f"
```