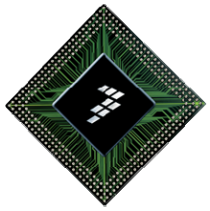
FSL SystemVerilog Requirements

- **2010.02**



freescale™ semiconductor

Overview

- Requirements on basic constructs and types
- Requirements on assertions
- Requirements on external capabilities
- Requirements on hierarchy
- Requirements for AMS
- High-level problems (solutions not yet proposed)

Requirements on basic constructs and types

- Enhancement to data declarations
- Unpacked arrays of size zero
- Enhancement to enums
- Aspect-oriented features
- Nontrivial covergroup shapes and filtering

Enhancement to Data Declarations

- PROBLEM: Users are currently required to declare data objects at the top of blocks. Code is more maintainable when data objects can be declared just before use.

- PROPOSAL:
  - Allow data objects to be declared interspersed with procedural statements.

Example

- EXAMPLE:

```
function void do_something();
  do_something_else();
  int return_value = do_one_more_thing();
endfunction
```

Unpacked arrays of size zero

- PROBLEM: Parameterized designs need a convenient way for certain constructs to "go away" when not needed. This capability is required in module ports. One approach is to use unpacked arrays of parameterized size.
  - LRM definition is not clear about arrays of unpacked size zero: `fooType foo[0]` is defined as `fooType foo[0:-1]`.

- PROPOSAL:
  - Clarify that `fooType foo[0]` is a size zero array.
  - Disallow `fooType foo[n]` for negative n.

Example

```
interface simple_bus;
   logic req, gnt;
endinterface: simple_bus
module #(parameter N) memMod(
   simple_bus a[N], // array of interfaces
   input logic clk
);
   for (genvar i=0; i<N; i++)
      always @(posedge clk) a[i].gnt <= a[i].req;
endmodule
module top;
   parameter M = 0;
   logic clk = 0;
   simple_bus sb_intf_arr[M];
   memMod #(M) mem(sb_intf_arr, clk);
endmodule
```

Enhancement to enums

- PROBLEM: enums cannot be extended like classes. As a result:
  - User needs to take care that enum values do not overlap
  - User needs to conditionally cast values
  - User needs to conditionally use the .name() enum method.
- PROPOSAL:
  - Allow reference to an enum inside the definition of a new enum
- EXAMPLE:
  - Extension of an enum defined in class A inside the class B.

```
class A;
 typedef enum { READ=0, WRITE=1} AType;

class B;
 typedef enum { A::AType, NOACC} Btype;
```

New aspect-oriented features

- PROBLEM:
  - Even if some aspect-orientated programming (AOP) features can be implemented using object-orientated programming (OOP) techniques, it is very helpful in some cases to have AOP within an OOP language. Adding AOP to SV can:
    - Enable fast fix path for late testbench changes
    - Enable hot fixes on provided infrastructure

- PROPOSAL:
  - Add new language constructs to support aspect-oriented programming

Manipulate class

- EXAMPLE:
  - Manipulation of a class without explicit extension class declaration:

```
class BusDrv;
  protected task execute();
    ..
  endtask : execute
endclass : BusDrv
//AOP
extends MyBusDrv(BusDrv); // Aspect called MyBusDrv
 event exec_done; // new public event for class BusDrv
 task run(); // new public method for class BusDrv
    execute();
   ->  exec_done ;
  endtask : run
endextends
```

Enhancement to covergroups

- PROBLEM: Covergroups do not allow easy specification of nontrivial and joint conditions on the bins of coverpoints and crosses.
  - Conditions on a coverpoint that are not simple enumerations
  - Joint conditions for exclusion from the coverage space (a.k.a., "filtering out")
  - Joint conditions for definition of a bin

- PROPOSAL (Mantis 2506):
  - Allow specification of types for coverpoints; infer types for crosses.
  - Allow "with" expression to specify nontrivial conditions on bins of a coverpoint or cross.
    - Use coverpoint name or "item" in a coverpoint.
    - Use coverpoint names in a cross.
  - Allow generation of bin definitions as sets, represented by arrays, via functions.
  - Use "dist" to specify the value selection policies for bins.

Covergroup bin definition (Mantis 2506)

- EXAMPLE: Cross logic [2:0] a with logic [2:0] b, but exclude pairs for which a + b > 8.

```
logic [2:0] a,b;
covergroup foo;
  X: cross a,b {
    ignore_bins ig_bin = X with(a+b > 8);
  }
endgroup
```

Covergroup bin definition (Mantis 2506)

- EXAMPLE: Create a coverpoint with a bin for each Fibonacci number from the initial Fibonacci generators 1, 1 that is greater than or equal to MIN and less than or equal to MAX and that can be stored in the integral type fibEltType.

```
typedef fibEltType fibQType [$];
function fibQType fibGen(fibEltType low, high);
   // write your fib generator here
endfunction
fibEltType exp;
covergroup fibCG @(posedge clk);
   fibEltType fibCP: coverpoint exp iff (!reset) {
      bins data[] = <fibGen(MIN, MAX)>;
   }
endgroup
```

Requirements on assertions

- First class local variables
- Alignment with PSL local variables

First class local variables

- PROBLEM: Local variables are not a first class language construct in SVA. They can be used only within named sequences and properties.
  - Simple assertions must instantiate helper properties or sequences that are declared solely for the purpose of enabling use of local variables.
  - Local variable scope is generally controllable at the granularity of declared sequences and properties only, not at the finer level of subexpressions.

- PROPOSAL:
  - Allow local variables to be declared at the beginning of a parenthesized sequence or property expression.
  - The scope of such a local variable is the parenthesized expression in which it is declared, subject to the existing rules of local variable flow.
  - The foundations for this enhancement have already been lain in the formal semantics of IEEE 1800 2009.

First class local variables

- EXAMPLE: When **`start`** occurs, capture **`dataIn`** and **`tagIn`**. Check that at the next occurrence of **`complete`** together with **`tagOut`** equal to the captured value of **`tagIn`**, **`dataOut`** equals the captured value of **`dataIn`**.

```
assert property
(
   start |->
   (
      dataType l_data = dataIn;
      tagType l_tag = tagIn;
      (complete && tagOut == l_tag)[->1]
      |-> dataOut l_data
   )
);
```

Alignment with PSL local variables

- PROBLEM: IEEE 1850 PSL is poised to introduce local variables with different definitions of scoping and semantics than SVA.
  - Claims have been made (FMCAD 2008) that the PSL approach fixes significant problems that exist with the SVA approach to local variables.

- PROPOSAL:
  - Study the technical differences between the new PSL approach to local variables and the longstanding SVA approach.
  - Analyze the cost/benefit tradeoffs associated with changing SVA local variables.
  - Determine what changes may be worthwhile and feasible in the next revision of SystemVerilog.

Requirements on external capabilities

- DPI/VPI interaction
- Plusarg access
- Management of random seeds

Enhancement and clarification of DPI/VPI(PLI) interaction

- PROBLEM: Interaction between the two main SystemVerilog API's DPI and VPI is not defined. This will result in vendor specific solutions outside the standard.

- PROPOSAL
  - Identify the possible and allowed interactions
  - Specify the supported interactions between DPI and VPI
  - Clarify the correct handling of those interactions

DPI example of allowed flow

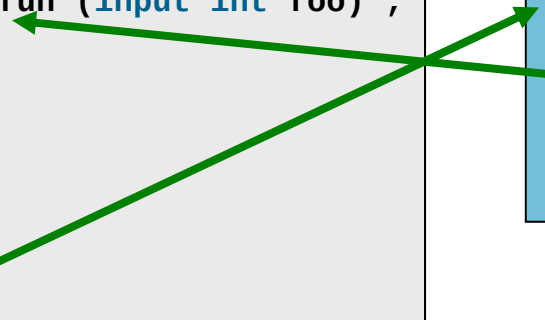
- EXAMPLE: This is what is working and supported. The SV calls “C”, which later calls back the SV. (The example is simplified since you also need to set the scope.)

SystemVerilog Code

```
import "DPI" function void call_c_func();

module test ;

export "DPI" function sv_export_fun ;
function void sv_export_fun (input int foo) ;
begin
…
end
endfunction

    ↓
initial
   call_c_func();

endmodule
```

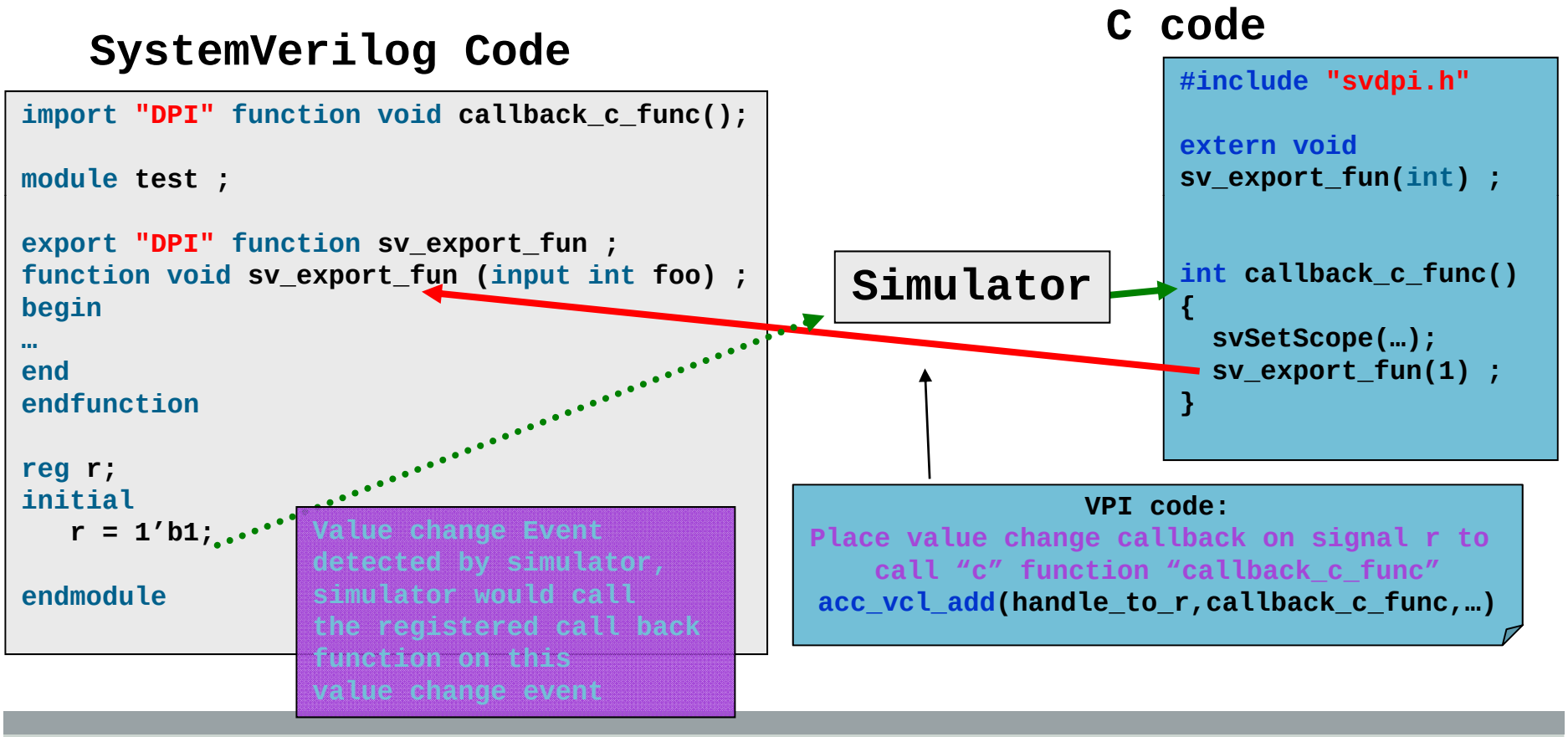
C code

```
#include "svdpi.h"

extern void sv_export_fun(int) ;


int call_c_func()
{
  …
  svSetScope(…);
  sv_export_fun(1) ;
}
```



DPI example of desired additional flow

- EXAMPLE: This is the desired flow which isn't explicitly allowed (the red arrow indicates what isn't standard).



freescale semiconductor

LRM text of relevance

- 3’rd paragraph of Section 35.5.3 (SV-2009):

  “A foreign language subroutine supported through some other interface (a VPI callback for example), can also make a call to svSetScope or to other DPI scope-related APIs. This foreign language subroutine can also call an export subroutine declared in a specific instantiated scope by first making a call to svSetScope. The behavior of the DPI scope-related APIs and invocation of DPI export subroutines will be simulator defined and is beyond the scope of the DPI specification.”

- The suggested solution is to identify the possible allowed interaction and explicitly state it.

Enhancement to plusarg access

- PROBLEM: Currently, user code must know the name of all plusargs in order to access them (with $test$plusargs or $value$plusargs). This restricts the ability to do more complex plusarg parsing.

- PROPOSAL:
  - Add a new system task that returns all runtime command line switches not recognized as simulator switches.
    - Removes switches intended specifically for the simulator.
    - Allows user code access to the entire set of command line plusargs.

Example

- EXAMPLE: Get all provided plusargs.

```
string commandline;
string plusargs[$];

commandline = $all$plusargs();
tokenize(commandline, plusargs);
```

Enhancement to random seed management

- PROBLEM:
  - SystemVerilog does not allow control of the random seeds per thread
- PROPOSAL:
  - Provide a system call to set a unique seed for all the threads in the simulation.
- EXAMPLE:
  - Suppose drivers that start driving a random transaction in the **initial** block. Each initial thread has its own random number generator (RNG), using its own seed. We would like having a way for setting the seed for ALL for these RNGs from a single point, such as the stimulus or an **initial** block from the testbench.

Example

- module drv(...);

```
  ...
  initial begin
      ...
      @(posedge rst_b);
      drive_transaction($urandom());
  end
```

- endmodule

- module testbench;

```
  drv drv1(...);
  drv drv2(...);
  initial begin
      $set_all_seeds(seed);    // Set the seeds of all threads (drv1, drv2, etc)
  end
```

- endmodule

Requirements on hierarchy

- Module instance handles
- Detecting bound port connections

Visibility of module instance handles

- PROBLEM: Allow visibility of module instance handles.

- PROPOSAL
  - Provide a way to manipulate handles to module instances to allow set/get module inputs and call module tasks by indirection

Problem/proposal description

- Several instances I1, I2, …, In of module M with tasks T1, T2, …, Tm
- want to call tasks, access signals, vars, randomizing the instance, like:
  - `I[$urandom_range(1, n)]->T5; // I[1..n] is an array of instance handles`
- Proposal: create an “instance handle” type and respective “getter”

```
HANDLE_T inst[1:n]; // array of instance handles

initial begin
  reg signal_val;
  int instance_number;

  for(int i=1; i <= n; i++) { // get the handles of all M instances
    inst[i] = get_instance_handle($psprintf(“top.modMinst%d”, i));
  }

  while (1) {
    instance_number = $urandom_range(1, n); // randomize instance
    inst[instance_number]->T1($urandom, $urandom);
    inst[instance_number]->T2($urandom);
    signal_val = inst[instance_number]->some_signal;
  }
end
```

Current problem workarounds

- Workaround 1: use “case”
  - case ($urandom_range(1,n)) 1: i1path.T5(); 2: i2path.()T5; … n: inpath.T5; endcase
  - cannot parameterize number of instances (n)
  - have to do one case for each task T1, …, Tm: hard to maintain
- Workaround 2: use DPI to handle the indirection
  - possible to parameterize number of instances (n)
  - have to maintain testbench C code, one wrapper for each task.
  - see code at next slide

Workaround using DPI

```
// System Verilog code

import “DPI” context routeT1(input int instance, int parm1, int parm2);
import “DPI” context routeT2(input int instance, int parm1); // one for each task

initial repeat forever
routeT1($urandom_range(1,n), p1, p2);

module M;
  export “DPI” T1(int parm1, int parm2);
  task T1(int parm1, int parm2);
  // …
  endtask
endmodule

// C code
#include “svdpi.h”
svScope inst_scope[n]; // array of scope pointers, initialized elsewhere
extern T1(int parm1, int parm2); // one for each task

void routeT1(int instance, int p1, int p2) { // one for each task
  svSetScope(inst_scope[instance]);
  T1(p1, p2);
}
```

New feature $sv_is_bound

- PROBLEM: It is not possible from within SystemVerilog code to determine whether an interface port is connected. This creates the following problem, especially for verification IP with optional signals on the interface:
  - Unused interface inputs need to be tied to the deasserted state.
  - Constraints for the unused functions need to be disabled to avoid randomization in unused random space
  - Functional coverage related to unused function needs to be removed in post processing .
- $isunknown system function does not address the problem, since it just checks for x and z on the line, which can be legal states at the interface.

- PROPOSAL:
  - Add a new system function to detect signals that are not connected.

New feature $sv_is_bound

- EXAMPLE:
  - Use on a verification IP Driver with optional xfr_err and test_acc signal.

```
class BusDrv;
constraint mode_constraint {
     mode inside {USR,SPRV,TEST,USR|TEST}
       iff ($sv_is_bound(Port.test_acc));
     mode inside {USR,SPRV}
       iff (!$sv_is_bound(Port.test_acc));
  }

 if ($sv_is_bound(Port.xfr_err))
    xfr_err = Port.initiator_cb.xfr_err;
 else
    xfr_err = 0;
```

New feature $sv_is_bound

- EXAMPLE:
  - Use on a verification IP Monitor with optional xfr_err and test_acc signal.

```
class BusMon;
covergroup acc_modes ( )@(sample_observer) ;
   coverpoint mode {
      bins s_usr   = {1}
           iff ((cmd == READ) || (cmd == WRITE) );
      bins s_supv  = {2}
           iff ( (cmd == READ) || (cmd == WRITE) );
      bins s_test  = {4}
           iff ( (cmd == READ) || (cmd == WRITE) );
      ignore_bins test_i = {4}
           iff (!$sv_is_bound(Port.test_acc));
   }
endgroup : acc_modes
```

Requirements for AMS

- Scale factors for real constants
- Mixed-signal assertions features
- Reals in covergroups

Scale factors for real constants

- PROBLEM: SystemVerilog does not provide support for scale factors on real constants.
  - Scale factors are a convenient shorthand when working with real constants.
  - Adding scale factors to SystemVerilog aids with SV-VAMS integration.
- PROPOSAL:
  - Add Verilog-AMS compatible scale factors to SystemVerilog.

```
real r1 = 1.6M; // M = 1e6
real r2 = 1.2m; // m = 1e-3
real r3 = 1.8n; // n = 1e-9
```

Mixed-signal assertions features

- PROBLEM: SVAs do not provide support for assertions involving real variables or continuous time, which are necessary for mixed-signal assertions.
  - If V(**a**) < 10.5 mV and **b** is true, then **c** is true.
  - When **a** falls, **b** remains low for at least 5.2 ms.
- PROPOSAL (work being done by ASVA committee):
  - Real values in SVAs
  - Real-time regular expression constructs
  - Ability to access analog and digital values in ASVA containers (modules, checkers, etc.)

Mixed-signal assertion features

- EXAMPLE: If V(**a**) < 10.5 mV and **b** is true, then **c** is true.

```
assert property (
  (V(a) < 10.5m) & b |-> c
);
```

- EXAMPLE: When **a** falls, **b** remains low for at least 5.2 ms.

```
assert property (
  $fell(a) |-> !b[*5.2m]
);
```

Reals in covergroups

- PROBLEM: Covergroups do not support real values.
  - Collecting coverage on real-valued signals requires specification of real bins in covergroups.

- PROPOSAL:
  - Allow real values to be specified in covergroups.
  - Extend bin syntax to allow real intervals to map into bins.

Reals in covergroups

- EXAMPLE: Create a coverpoint with bins for various ranges of a real variable.

```
real r1;
covergroup realCG @(posedge clk);
   realCP: coverpoint r1 iff (!reset) {
      bins i0 = { r1 < 1.0 };
      bins i1 = { r1 >= 1.0 & r1 < 2.5 };
      bins i3 = { r1 >= 2.5 };
   }
endgroup
```

- alternatively

```
      bins i0 = { item in (-inf,1.0) };
      bins i1 = { item in [1.0,2.5) };
      bins i3 = { item in [2.5,inf) };
```

High-level problems (solutions not yet proposed)

- PROBLEM: Enable assertions to be used in classes.
- PROBLEM: Facilitate multiple inheritance for classes.
- PROBLEM: Provide dynamic memory consumption debug constructs.
- PROBLEM: Adopt the (upcoming) UVM macros into SystemVerilog.
- PROBLEM: Form new technical committee for synthesizable SystemVerilog.
- PROBLEM: Improve interaction between SystemVerilog and C/C++ from both a data structure perspective and a method calling perspective.