Reminder - the next SystemVerilog-BC Conference Call will start this morning at 9 AM PDT

The following information is for use in connecting to the list committee meetings (all times are West Coast):

19 August 9:00am-11:00am SV-BC
19 August 11:00am-1:00pm SV-EC

Toll Free Dial In Number: (877)233-7845
International Access/Caller Paid Dial In Number: (505)766-5458
PARTICIPANT CODE: 516134

SV-BC Minutes from the August 5th meeting (to be sent later this morning):

Next SystemVerilog-BC meeting: September 4th - Face-to-face meeting at Synopsys
Next IEEE Verilog Standards Group Meeting: August 26 - starts at 8:30 AM PDT - contact Mike McNamara for details.

Others -
16 September **CANCELLED** (Cliff will be unavailable to lead the call)
30 September
14, 28 October
11, 25 November
9, 16 December
6, 20 January
3, 17 February

NOTE: all emails will now only be sent to the sv-bc@eda.org reflector.

This is my list of attendees and voting status - please submit corrections:

```
(aaaa) Cliff Cummings       (Sunburst Design)  *
(aaaa) David Smith          (Synopsys)         *
(aa--) Heath Chambers       (HMC)              *
(aaaa) Karen Pieper         (Synopsys)         *
(aaa-) Kevin Cameron        (NSC)              *
(aa--) Medi Mohtashemi      (Synopsys)         *
(aa--) Paul Graham          (Cadence)          *
(aaaa) Peter Flake          (Co-Design) (Henry Cox)       *
(aaaa) Simon Davidmann      (Co-Design)        *
(aaaa) Stefen Boyd          (Boyd Technology)  *
(aaa-) Steven Sharp         (Cadence)          *
(a---) Rog Armoni           (Intel)            *
(-aaa) Dave Kelf            (Co-Design)        *
(-aaa) Dennis Brophy        (Model Technology) *
(-a--) Kurt Takara          (Zero-In) sv-bc reflector??
(-aa-) Mike McNamara        (Verisity)         *
(-aaa) Tom Fitzpatrick      (Co-Design)        *
(--aa) Vasisilios Gerousis  (Seimens)          *
(---a) Francoise Martinolle (Cadence)          *
```

* indicates eligible to vote on consensus issues

Steve Sharp will be on the call this morning to pick up discussion of Cadence issues, starting with section 3) as noted in the email message:

+ From: Stefen Boyd <stefen@boyd.com>
+ Date: Mon, 08 Jul 2002 11:58:07 -0700
+ Subject: SV-BC: Minutes from July 8, 2002 meeting

The pertinent sections of the July 8th email is reproduced below for committee convenience.

==========
Additional detail for the issues discussed in the System Verilog 3.1 Kickoff Meeting
==========
Section 2 - (discussed at the July 22nd SystemVerilog-BC Committee meeting)
==========
Section 3 - Data packing issue (Kevin/NSC) [Basic]
- it is impossible to implement "union" from the current LRM description
- there are many ways to do it which are not compatible
- encoding of logic types is a factor, and "big-endian" vs. "little-endian"
- unions should have either all logic or all bit as the base type of all elements
- if packing is defined then 'packed' union syntax is redundant
- may be desirable to state the packing/alignment explicitly for software compatibility
[See email on the reflector archive for details from Kevin.]
==========
Section 3 - Type use before definition (Steve, Paul/Cadence) [Basic]
- forces type checking to be post-elaboration
- cause unnecessary complication of analysis, particularly separate analysis
- useful only with pointer types
[This has been discussed before - no additional detail is involved.]
==========
Section 3.1 - Parameterized data types (Stuart/Cadence) [Basic]
- nice, but difficult to use because they cannot be resolved until elaboration
- (can we improve this to better support separate compilation?)
It's nice to be able to use parameters to specify data types (similar to C++ class templates), but in Verilog parameters cannot be resolved until elaboration. The end result is that for such parameterized modules almost no error-checking or code generation can be done until after elaboration. This is painful for the user (due to very late error checking) and complicates tool implementation.
==========
Section 3.4.1 - Issues with Time data type (Steve/Cadence) [Basic]
- need to detail the rules for mixed expressions, scaling, etc.
The specification of the time data type (section 3.4.1) is extremely unclear. The examples are not particularly helpful, since they only involve simple time literals used as delays. There is no specification of what happens when a time literal is assigned to a time variable, or an integer or real value is assigned to a time variable, or a time value is assigned to an integer or real variable, or time and integer values are combined in an expression. It is so unclear that it is hard to even formulate a general question, so I will ask about examples. The answers will help clarify things, but what is really needed is a specification of the underlying rules that produce those answers.
a. If I have the following code:
`timescale 1ns/10ps
time t;
initial

```
begin
t = 1;
#(t) $display(t);
end
```
What is the numerical value in the variable t? In normal Verilog, a time variable is just a 64-bit vector, so it would be 1. But the specification seems to indicate that a time variable is this new time data type, so it needs to be scaled. I would guess that this means multiplication by the ratio between the time unit and the time precision, or 100. Does this mean that t gets the scaled value of 100? What if some other module has a precision of 1ps, so that the overall time step for the simulation is 1ps? Does t still get 100 based on the local precision, or 1000 based on the smallest precision? What is the delay produced by the #(t)?

b. If I have the following code:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 1ns;
#(t) $display(t);
end
```
what is the numerical value in the variable t? Presumably it is 100, since that is the multiple of the time step. Again, does this depend on the local precision or the overall precision? What is the delay produced by the #(t)? One would hope it is 1ns.

c. If I have the following code:

```
`timescale 1ns/10ps
integer i;
initial
begin
i = 1ns;
#(i) $display(i);
end
```
Is this legal? If so, what is the numerical value of the integer variable i? Does this produce a delay of 1ns?

d. If I have the following code:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 1ns + 10ps;
#(t) $display(t);
#(1ns + 10ps) $display($time);
end
```
What is the value of t? Presumably both time literals get scaled and then added, so the result is 101.

e. If I have the following code:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 1ns + 1;
#(t) $display(t);
#(1ns + 1) $display($time);
end
```
How are mixed expressions like this handled? Note that an integer expression

would fully evaluate as an integer and then be scaled for the delay. But a time literal needs to be scaled to the time unit first, and can't be re-scaled again later. How and when are these scalings done?
f. If I have the following code:

```
`timescale 1ns/10ps
time t1,t2;
initial
begin
t1 = 0.1 + 0.1;
#(t) $display(t);
end
```

What is the value of t? Presumably it is 20, and the #(t) delays by 200ps. But are those answers obtained by adding and then scaling, or scaling and then adding? Some further questions will expose differences between those.
g. If I have the following code:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 0.004 + 0.004;
#(t) $display(t);
end
```

What is the value of t? Are the two reals scaled and truncated to 0, and then added to get 0? Or are they added, then scaled and truncated, producing 1?
h. One possible rule is that operands are all scaled and then combined. But this doesn't work for all types of operands. For example:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 1ns << 1;
#(t) $display(t);
end
```

Clearly scaling the 1 by 100 before the shift is undesirable. Perhaps it only scales operands that are not self-determined. And what happens if we try to shift by 1ns? Is this legal? There are no specifications of where times are legal.
i. Even scaling only operands that are not self-determined doesn't always work properly. Whenever dimensional analysis would result in units that are not times, this scaling falls apart. For example:

```
`timescale 1ns/10ps
time t;
initial
begin
t = 1ns + 1 * 1;
#(t) $display(t);
end
```

What is the value of t here? If both integer literals get scaled, then the scaling factor gets multiplied in twice. So apparently scaling needs to wait until the integer sub-expression is actually combined with a time (unlike the other integer type conversions in Verilog, which are applied to operands before any operations). But this is not specified anywhere.

==========

Section 3.6 - Implications of Enum type I/O (Steve/Cadence) [Basic]
- need to detail what is expected of Verilog I/O routines to support this
- [also vcd enhancements]

In section 3.6, it is stated that enumerated types can be displayed using the enumerated names. Is this supposed to imply some capability of the Verilog I/O routines? If so, these capabilities need to be defined. If it is just a speculation about what the user interface to some tool might allow during debugging, then this should be made plain.

==========

Section 3.7 - Definition of "masked" and "unmasked" (Steve/Cadence) [Basic/Doc]

- apparently not defined?

Section 3.7 refers to structure members as being masked or unmasked. These terms do not appear to be defined anywhere, and appear to refer to 4-state and 2-state values. The terminology should be fixed.

==========

Section 3.7 - Size requirement(?) on members of a packed union (Steve/Cadence) [Basic/Doc]

- should say "must be the same size", not "are the same size" (?)

The text in section 3.7 stating that members of a packed union "are" the same size should be changed to state that they "must be" the same size, if that is the intent.

==========

Section 3.7 - Passing large structs/arrays (Stuart/Cadence) [Basic, System]

- can this be done by reference instead of value (which would be inefficient)?

Is there a way to pass large structs or arrays to functions/tasks by pointer or as const references, rather than by value? If not, this can be extremely inefficient for large structs or arrays.

==========

Section 3.8 - Conversion of shortreals to 32 bits (Steve/Cadence) [Basic/Doc]

- "bit pattern is preserved" is inconsistent with other conversions
- should use $realtobits if the intent is to transfer the bit pattern

In section 3.8, it is stated that when a shortreal is converted to 32 bits, the bit pattern is preserved (instead of rounding). This is not consistent with the rest of Verilog. If a shortreal converts to int by rounding, then it should convert to 32 bits by rounding. The int type is equivalent to 32 bits. Also, a real converts to a reg by rounding, so a shortreal should do the same. Nothing is said about real transferring its bit pattern when converted to bits, just shortreal. Since the statement is made in the section on casting, it implies that this happens only when casting, and that a conversion on assigning to a variable of that type would behave differently. This statement in the document is a bizarre exception to all related situations. If a real needs its bit pattern transferred to a vector, Verilog provides $realtobits for this purpose. A similar function could be provided for shortreal. Alternately, a union of a shortreal and a bit vector, or a struct or !
union containing just a shortreal, could be used for conversion.

==========

Section 4.2 - Packed array of signed (Erich/Cadence) [Basic]

- conflict between signed elements and signed whole array

This has been partially addressed in draft 9, but there is still an issue.
Draft 9 says
"If a packed array is declared as signed, then the
array viewed as a single vector shall be signed.
A part-select of a packed array shall be unsigned."
But if a part-select of a signed packed array includes the MSB, shouldn't that part-select be signed? In particular, if the part-select selects all the bits of a signed packed array, shouldn't it be considered signed, since the array as a whole is considered signed?

==========

Section 5.3 Constant expression (Paul/Cadence) [Basic]

- need to define precisely what can/cannot appear in a constant expression
[This has been discussed before - no additional detail is involved.]
==========
Section 6.1 Attribute syntax (Paul/Cadence) [Basic/BNF]
- should factor syntax to improve readability
[This has been discussed before - no additional detail is involved.]
==========
Section 9 - Process execution efficiency when calling C (Kevin/NSC) [C/C++ Intf]
- LRM doesn't say much about calling C from Verilog processes
- should require that C functions can't suspend
==========
Section 9.1 - Interleaving of exection (Stuart/Cadence) [Basic]
- allowing arbitrary interruption is error-prone
- should only allow interruption at synchronization points
Why allow a thread of execution to be interrupted by another thread between any statements? Why not only at synchronization points such as wait (#) or blocking assignments? The current proposal results in a modeling style that is very error prone and provides no benefit in terms of performance or functionality over only allowing interruption at synchronization points.
==========
Section 9.1-Related - Verilog 2001 - Scheduling Algorithm (Shalom/Motorola) [Basic]
- allows interleaving of processes that need to be atomic
- conflicts with requirement that non-blocking assignments execute in order of appearance
- potential problem with scheduling of PLI calls
==========
Section 9.1 - Issues with dynamic processes (Stuart/Cadence) [Basic]
- need the ability to suspend/resume/abort child processes
- need process handles to support this
Ability to use dynamic processes seems incomplete without ability to specify suspend/resume of child processes, and ability to abort child processes. These are needed to model pipelines with stalls, operating systems, etc. The current proposal doesn't seem to have a clean way to return process handles that will be needed once suspend/resume/abort etc are supported in future.
==========
Section 13 - Interfaces vs. Modules (Stuart/Cadence) [Basic, System]
- interfaces and modules are almost the same
- should make them so and simplify definition
I see no reason why "interfaces" and "modules" aren't exactly the same thing. Both can have ports, both can export tasks, both can be parameterized, both represent hierarchical structures, both can have "always" blocks and processes, etc. Making "interfaces" and "modules" be exactly equivalent would go a long way towards simplifying SystemVerilog.
==========
Section 13.1 - Interfaces restrictions (Stuart/Cadence) [Basic, System]
- should allow an interface to contain other modules
- would allow wrapping of a module with an interface
Why can't an interface contain other modules? This would allow existing module implementations to be "wrapped" within an interface implementation and then instantiated just like any other interface. For example, I might have an RTL implementation of a FIFO. I might want to embed this within an interface implementation that implements write() and read() tasks.
==========
Section 13.1 - Scheduling issues (Stuart/Cadence) [Basic, System]
- interfaces allow non-deterministic behavior due to scheduling order

- need ability to control scheduling order
Interfaces provide plenty of opportunities for users to create non-deterministic designs (i.e. designs whose behavior is dependent on the scheduling order that a particular simulator happens to use.) What are the rules and the capabilities that System Verilog will provide to enable users to create interfaces that do NOT result in non-deterministic designs? For example: Can the user model the equivalent of non-blocking delays & delta cycles within interfaces? Can the user do the equivalent of signal resolution for multiple processes that simultaneously update the state of an interface?

==========

Section 13.2.3 - Interface usage issues (Stuart/Cadence) [Basic, System]
- need to be able to specify and enforce rules about interface usage
- e.g., use of either Read or Write operation but not both
- e.g., limits on number of modules/processes invoking a given interface operation
- should be possible to define such rules in the interface itself without changing other code
- need to check rules in a way that allows for separate compilation
- need to be able to specify that some tasks/members of an interface are private
"Generic" interfaces allow a module to be attached to different interfaces, but they do not convey to the reader or allow tools to enforce that the module must be connected to an interface that implements a specific set of tasks or has specific members. As a simple example, there may be several different interface implementations of a FIFO -- one might gather statistics, another might not. In both cases modules that might be connected to such FIFOs should have an interface declaration that clearly specifies the set of FIFO operations that must be supported. E.g. write() but NOT read(), or vice-versa. Given the current proposed scheme, it can only be determined at elaboration-time whether an interface binding in a particular case is legal or not. Instead I believe it ought to be possible to determine this legality at the time a single module or interface is compiled.
It should be possible to do port registration checks for interfaces. For example, a user might want to implement a FIFO interface that enforces that it has exactly one module or process that writes to it, and exactly one module or process that reads from it. It should be possible for the designer to create such checks only within the interface, without forcing the code within modules attached to the interface to be modified.
In SystemVerilog today all tasks and members of an interface are available to be used by modules attached to the interface. This will inevitably lead to designs or models that use tasks or members of an interface that were never intended to be used externally. Therefore I believe that it should be possible to make certain tasks and members private so that they cannot be accessed externally.

==========

Section 13.4 - Modports issues (Stuart/Cadence) [Basic, System]
- allow modports to be declared outside of interface/module, for reuse
- allow (modules and) interfaces to specify which modport(s) they implement
- import/export is confusing and unnecessary
It appears that Modports can only be declared within an interface. Why not allow modports to be declared outside of interfaces and modules, so that more than one interface implementation can implement the same modport? This would be a less error-prone solution than "generic interfaces".
The use of "import/export" within Modports is confusing and unnecessary. Instead I believe that modports should be declared separately from interfaces, and the tasks that a modport makes available should be declared within the modport. Then when an interface is to implement the tasks of a modport, it explicitly indicates this via a "implements mod_port_name" construct. Similarly, since I

believe that modules and interfaces should be equivalent, a module that wishes to make tasks available to other modules or interfaces will simply specify that it "implements some_mod_port_name".
Modules and interfaces should be able to implement multiple modports.

==========

Section 13.5.4 - Issue with extern forkjoin task (Stuart/Cadence) [Basic, System]

- not necessary (at least for the example)
- unnecessarily inefficient - there are better methods

"Extern forkjoin tasks" capability is highly specialized and unnecessarily inefficient. A much better approach to modeling the example within this section is to have the simple_bus be aware of the address map for the slaves attached to the bus -- then only the slave responsible for a particular address needs to be activated for any request. This would allow us to get rid of the "extern forkjoin task" construct altogether and have a much more straightforward and efficient solution.