Introduction

SystemVerilog is both a design and verification language. For this reason, VPI has been recently extended to handle coverage and assertion data, and there is a proposal to update it with the new data types and testbench facilities of SystemVerilog that are not covered in the original VPI. The wealth of design and verification data access mechanisms in VPI makes it an ideal vehicle for tool integration. VPI can potentially replace today's arcane, inefficient, and error-prone file based data exchanges with a new mechanism for tool interface likely enabling innovative flows. A single access API eases the interoperability investments for both vendors and users. Reducing interoperability barriers allows vendors to focus on tool enhancements by adding proprietary extensions. Users, on the other hand, will be able to create integrated design flows from a multitude of best-in-class vendors spanning the realms of design and verification (e.g. simulators, debuggers, formal tools, coverage or testbench tools and so on).

Goal

The purpose of this donation is to enhance the SystemVerilog design data dumping facilities. We propose to extend VPI with read/write dumping facilities so that it acts as an extended API for data access tool interaction irrespective of whether this data is in memory or a file (i.e. whether in host memory or in persistent form as in file) and also irrespective of the tool the user is interacting with.

To that end, Novas is donating several technology elements from its read/write API of the Novas proprietary dump format (Fast File Signal DataBase: FSDB).

In the first part of this document we describe the original donation extracted from the FSDB reader/writer documentation. In the second part we describe in brief how we propose to enhance the current VPI by merging the aforementioned ideas and facilities into it.

PART I: Original Technology Donation[1]

Data Dump Reader

The dump file may contain two kinds of information:

1) Design hierarchies: which include dumped scopes and variables (and their data types) and represents their inter-relationship. This is sometimes referred to as the “tree” portion.

2) Value changes: of the stored variables throughout the dump time ranges.

[1] Note that we primarily describe here the facilities that do not replicate any similar VPI functionality. Where appropriate, we will point out the areas where VPI functionality exits and can be reused with a slight update (not discussed here). **We will also where appropriate remove any Novas terminology and replace it with a generic form.**

The Reader tells the application the design hierarchies via tree callback function.

The Reader tells the application the value changes via a Value-Change (VC) traverse handle, which provides many functions to traverse the value changes back and forth.

Note that after the design hierarchies have been traversed in a tree callback function, the Reader does not keep the design hierarchies with a well-defined data structure in memory; it is up to the application to keep this information.

Note that the application can create more than one value-change traverse handle for the same variable, thus providing different views of the value changes. Each value-change traverse handle has an internal index, which is used to point to the current time and value change. In fact, the value-change traverse is done by increasing or decreasing this internal index.

The Flow of Reading of the dump file is as follows:

- Open the dump file.
- Traverse the design hierarchies.
- **Select variables we are interested in.**
- **Load variables' value changes onto memory.**
- **Create value-change traverse handle for variables.**
- **Traverse the value changes via traverse functions.**
- Close the dump file.

Opening/Closing the dump file, and traversing the design hierarchies are facilities already found in the VPI and will not be covered here[2].

Load Variables' Value Changes onto Memory

Once the variables (the application is interested in) have been selected (by addition to the LoadSignalList), the value changes of these variables are loaded onto memory by calling ddrLoadSignals(). Note that value changes of unselected variables are not loaded onto memory.

Traversing the Value Changes

Traversing value changes is done via value-change traverse class defined by data dump Reader. The class (ddrVCTrvs) contains numerous member functions used to traverse the value changes back and forth. In tree callback function, the application knows the vpihandle of each variable. These handles are very important because when the application needs to traverse the value changes of a variable, the original handle is the key to create a value-change traverse object associated with it. By supplying the vpi

[2] As appropriate we will share our expertise as part of the committee work to update those mechanisms if needed.

handle, the application calls the API, ddrCreateVCTrvsHdl, to create a traverse object handle and points the pointer at it; such a handle is called a "traverse handle." After that, the application can call the member functions with that traverse handle, such as ddrGotoXTag, ddrGotoNextVC, ddrGetVC, and so on, to traverse the value changes back and forth. Here is a sample code segment.

```
ddrObject *obj;
vpihandle var_handle;
ddrVCTrvsHdl vc_trvs_hdl;
byte_T *vc;
unsigned64 time;
… …
vc_trvs_hdl = obj->ddrCreateVCTraverseHndl(var_idcode);
vc_trvs_hdl->ddrGetMinXTag((void*)&time);
vc_trvs_hdl->ddrGotoXTag(&time);
for ( ; ; ) {
        if (DD_RC_SUCCESS != vc_trvs_hdl->ddrGotoNextVC())
                break;
        vc_trvs_hdl->ddrGetXTag((void*)&time);   // Time of VC
        vc_trvs_hdl->ddrGetVC((void*)&vc);       // VC data
}
```

The code segment creates a value-change (VC) traverse object associated with a variable, whose handle is represented by var_handle, and creates a traverse handle, vc_trvs_hdl. With this traverse handle, it first calls ddrGetMinXTag to query the minimum time where the value has changed, then it moves the internal index to that time by calling ddrGotoXTag; and, finally, it calls ddrGotoNextVC to move the internal index forward repeatedly until there is no value change behind. ddrGetXTag gets the actual time where this VC (data obtained by ddrGetVC) happens (note the "hold value" semantics between VCs for HDL variables in general, this is not the case for some other variables e.g. assertion variables. See the Section on "Jump Behavior" for more details).
The traverse handle is a pointer that points to a traverse object, and the value-changes are traversed via the member functions of the traverse object. Here is a short description of each member functions. More details are in the function list section.

- ddrGotoXTag: moves the internal index to the proper position, based on the time specified by the application.
- ddrGotoPrevVC: moves the internal index to the previous time and value change.
- ddrGotoNextVC: moves the internal index to the next time and value change.
- ddrGetXTag: returns the current time, based on the internal index.
- ddrGetVC: returns the current value change, based on the internal index.
- ddrGetMinXTag: returns the minimum time where value change occurs.
- ddrGetMaxXTag: returns the maximum time where value change occurs.
- ddrHasIncoreVC: tell the application if this variable has value changes.

- ddrFree: is used to free the data structures of the traverse handle itself. The value changes of the associated variable are not freed.

Jump Behavior

Jump behavior refers to the behavior of ddrGotoXTag(). The application specifies a time to which it would like to jump, but the specified time may or not have value changes. Data Dump Reader will align the return time if necessary. In Figure 1 below, the whole simulation run is from time 0 to time 65, and a variable has value change at time 0, 15 and 50.

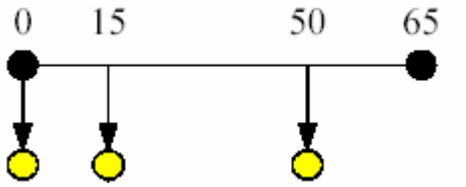

Figure 1 Jump Behavior Illustration

If we create a value-change traverse handle associated with this variable and try to jump to a different time, the result will be determined as follows:

- Jump to 10; return time is 0.
- Jump to 15; return time is 15.
- Jump to 65; return time is 50.
- Jump to 30; return time is 15.
- Jump to (-1); return time is 0.
- Jump to 50; return time is 50.

If the jump time has a value change, then the internal index of the traverse handle will point to that time. Therefore, the return time is exactly the same as the jump time. If the jump time does not have a value change, and if the jump time is not less than the minimum time of the whole trace[3] run, then the return time is aligned forward. If the jump time is less than the minimum time, then the return time will be the minimum time.

Reader Function List

NOTE: In the following "ddvarhandle" is any unique var handle it can be replaced by "vpihandle"

ddRC ddrObject::ddrAddToSignalList(ddvarhandle vh);

Purpose: Add signal we are interested in to signal list. The chosen scheme is based on variable handle since it identifies a unique signal.

Arguments:

vh: the var handle of the signal that application is interested in.

Return Value

[3] The word trace can be replaced by "simulation", we use trace here for generality since a dump file can be generated by several tools.

DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: ddObject -> ddrAddToSignalList(handle);

ddRC ddrObject::ddrResetSignalList();

Purpose: Reset signal list, i.e. empties signal list so that it contains no signals.
Arguments:
None.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: ddObject -> ddrResetSignalList();

bool_T ddrObject::ddrIsInSignalList(ddvarhandle vh);

Purpose: Check the signal to see if it is in signal list or not, while the signal is specified by its handle.
Arguments:
Var handle which identifies a signal.
Return Value
TRUE if the signal is in signal list, FALSE otherwise.
Example: ddObject -> ddrIsInSignalList(handle);

ddRC ddrObject::ddrLoadSignals();

Purpose: Load the value changes of the signals that are in signal list onto memory. That means if a signal is not added into signal list, then ddrLoadSignals won't load its value changes onto memory.
Arguments:
None.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: ddObject -> ddrLoadSignals();

ddRC ddrObject::ddrUnloadSignals();

Purpose: Unload the value changes of the signals from memory, which means all in-core value changes will be unloaded. Note that signals have a reference count. If a signal's reference count is not zero, its value changes will not be unloaded.
Arguments:
None.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: ddObject -> ddrUnloadSignals();

ddrVCTrvsHdl ddrObject::ddrCreateVCTrvsHdl(ddvarhandle vh);

Purpose: Create a traverse handle associated with the signal which is specified by its handle. Application can traverse the value changes of the signal via its traverse handle, and there could be multiple traverse handles associated with a signal.
Arguments:

vh: the handle which identifies the signal.
Return Value
Non-null if successful, null otherwise.
Example: vc_traverse_handle = ddObject -> ddrCreateVCTrvsHdl(handle);

ddRC ddrVCVCTrvs::ddrGotoXTag(unsigned64 *xtag);

Purpose: Try to move value change traverse index to xtag, if there is no value change at xtag, then the value change traverse index is aligned based on the jump behavior defined by DD reader, and the xtag will be updated based on the aligned traverse index. For details, please refer to DD reader document, look for jump behavior section. If there is a value change occurring at xtag, then value change traverse index is move to that xtag.
Arguments:
*xtag: a pointer which represents the xtag where we want to go.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.

ddRC ddrVCVCTrvs::ddrGetXTag(unsigned64 *ret_xtag);

Purpose: Get xtag based on value change traverse index. Usually, this API is called after ddrGotoXTag is made successfully.
Arguments:
*ret_xtag: a pointer representing returned xtag.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGetXTag(&time);

ddRC ddrVCVCTrvs::ddrGetVC(byte_T **ret_vc);

Purpose: Get value change based on value change traverse index.
Arguments:
**ret_vc: Pointer to pointer to byte_T representing the value change.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGetVC(&vc_ptr);
Note: You can get the value that is specially associated with an assertion variable (see IsAssertionVar) by using structure ddrAssertionVarValue[4] for example:

```
struct ddrAssertionVarValue {
     unsigned64 begin_xtag; // begin time
     unsigned64 end_xtag; // end time
     str_T result; // result: success, failure, ...
};
```

ddRC ddrVCVCTrvs::ddrGotoNextVC();

Purpose: Move value change traverse index to the next value change, if there is no value-change behind value change traverse index, then calling of this API will fail.

[4] This is just a placeholder the actual data will be that of the VPI variable of object type **vpiAssertion**

Arguments:
None.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGotoNextVC();

ddRC ddrVCVCTrvs::ddrGotoPrevVC();

Purpose: Move value change traverse index to the previous value change. If there is no value-change before value change traverse index, then the calling of this API will return fail.
Arguments:
None.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGotoPrevVC();

bool_T ddrVCVCTrvs::ddrHasIncoreVC();

Purpose: Check the signal to see if it has value change or not. Usually, this API is used when the data file is still growing or in processing.
Arguments:
None.
Return Value
TRUE if the signal has value change, FALSE otherwise.
Example: vc_traverse_handle -> ddrHasIncoreVC();

ddRC ddrVCVCTrvs::ddrGetMinXTag(unsigned64 *ret_xtag);

Purpose: Get the xtag of the first value change. Since simulation xtag is in ascending order and the first value change of each signal might occur at different xtag, so ddrGetMinXTag is used to get the xtag of the first value change, and the xtag would be the minimum one for that signal.
Arguments:
*ret_xtag: pointer representing the xtag of the first value change.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGetMinXTag(&time))

ddRC ddrVCVCTrvs::ddrGetMaxXTag(unsigned64 *ret_xtag);

Purpose: Get the xtag of the last value change. Since the time xtag is in ascending order and usually the last value change of each signal occurs at different xtags, so ddrGetMaxXTag is used to get the xtag of the last value change, and the xtag would be the maximum one for that signal.
Arguments:
*ret_xtag: pointer representing the xtag of the first value change.
Return Value
DD_RC_SUCCESS if successful, DD_RC_FAILURE otherwise.
Example: vc_traverse_handle -> ddrGetMaxXTag(&time))

void ddrVCVCTrvs::ddrFree();

Purpose: Free the value change traverse object. When the traverse object is freed, the reference count of the signal it is associated with will be decremented by one. Only when the reference count of a signal is zero could its value changes be unloaded by *ddrObject::ddrUnloadSignals()*. Hence it is import to free a traverse object when it is no longer in use. Note that this routine does not free the value changes associated with the signal. It just frees the traverse object itself. User must call *ddrObject::ddrUnloadSignals()* to actually unload a signal's value changes
Arguments:
None.
Return Value
None.
Example: vc_traverse_handle -> ddrFree();

bool_T ddrVCVCTrvs::ddrIsItAssertionVar();
Purpose: Check to see if the var that the ddrVCVCTrvs currently traversing is an assertion var or not.
Arguments: none
Return Value: TRUE if the signal is an assertion var. FALSE otherwise.

Dump Data Writer

A dump file may contain two kinds of information: design hierarchies, and value changes. The design hierarchies include the hierarchies between each scope (design entity) and the variables that each scope holds. The value changes are the "waveform" data. Each value change includes a time and a value. In order to write information into the dump file, Writer supports two kinds of creation mode: tree creation mode and value change creation mode. Under tree creation mode, the application calls Writer APIs to create the design hierarchies, variables and their data types. Under value change creation mode, the application creates the VC data. A handle for a variable can be used to write out the data. A variable may have more than one name i.e. an "alias." For example, a variable may be called "bus" in scope "top," and "data_bus" in scope "system." But both "bus" and "data_bus" refer to the same variable that has a unique handle. The application can create an alias to a variable by calling variable creation API with the different name but the same handle.

A dump file may contain none, one, or multiple design hierarchies. The design hierarchy is like a multi-tree. A node corresponds to a scope of the design hierarchy. A scope can contain scopes and variables. The capability that a scope can contain scopes recursively builds the design hierarchy. In order to traverse the design hierarchy, we must introduce the "current scope" concept: meaning "which node we are at now" in the writing process. Some of the tree creation APIs can move the "current scope" to another one so that the application can describe and build the design hierarchy. The "current scope" also determines which scope the newly created variables belong to: if a variable is created, then it belongs to the "current scope". The writer is best built around a **time-based scheme**: the application stops at a specific time where value changes occurred, then it figures out what variables have value changes at that specific time. Finally, it creates the

value changes of those variables at that specific time. The same steps repeat until it reaches the maximum time of the run.

The value change creation is composed of 2 steps:

- Tag time creation
- Value creation at that tag

The time is created by calling ddwCreateXCoorByHnL, while the value is created by calling ddwCreateVarValByHndl API.

Writer Function List

bool_T ddwBeginTree(ddwObject *obj);

Purpose: Begin of tree creation and sets current scope to root scope
Arguments:
*obj: Pointer to DD writer object
Return Value TRUE on success; FALSE on failure

void ddwEndTree(ddwObject *obj);

Purpose: End of tree creation. It closes a tree creation.

void ddwCreateScope(ddwObject *obj, byte_T type, str_T name);

Purpose: Create a sub-scope under current scope and then current scope goes to the newly created sub-scope.
Arguments:
*obj: Pointer to DD writer object
type: Scope type
name: Pointer to string which represents scope name
Return Value none

void ddwCreateUpscope(ddwObject *obj);

Purpose: Make current scope go to its parent scope.
Arguments:
*obj: Pointer to DD writer object
Return Value none

ddRC ddwCreateVarByHndl(ddwObject *obj, ddwVarData *data);
ddRC ddwCreateAssertionVarByHndl(ddwObject *obj, ddwAssertionVarData *data);

Purpose: Create a var by handle.
Arguments:
*obj: Pointer to DD writer object.
*data: A pointer to var data
Return DD_RC_SUCCESS/FAIL

void ddwCreateXCoorByHnL(ddwObject *obj, unsigned64 time);

Purpose: Create an xtag where a value change has occurred. Time is a 64-bit unsigned integer.
Arguments:
*obj: Pointer to DD writer object
time: 64-bit unsigned integer
Return Value none

PART II: How to Integrate into VPI

Note: The assertion portion has already been partially integrated into the Assertion VPI extension.

Reader:

The idea is to integrate into VPI using the same flavor and shape of the VPI routines. So the donation routines will be added as new types and/or object properties into VPI instead of adding new methods. So the integration outline is as follows[5]:

- A new set of **#define** would be added to accommodate the new types or object properties.
- The VC traverse handle would be obtained appropriately from the original variable handle using **vpi_handle()** method.
- The time traversal APIs would be integrated as additions to **vpi_control()** e.g. Next, Prev and so on.
- The "get" APIs (value/current, min, or max time) would be added into the **vpi_get()**, and **vpi_get_value()** methods.
- Dump object or file would be obtained by extensions to the **systfs** as in extending **$dumpvars**/**$readvars** to get any special information (along the same strategy followed in Verilog 2001 for dump enhancements to Verilog).

Writer:

VPI already has the basic concept of **Value Change** callbacks; incorporating writer calls into those should accomplish the value change writing in a seamless way.

Conclusion

In summary, we propose here a set of enhancements donated by Novas Software, Inc. to Accellera as part of our continuing involvement in the SystemVerilog standardization effort. We envision that by working closely with the other user and vendor members of the SV committees (SV-CC in particular) we will further improve SystemVerilog by adding a data dump component.

[5] Section 27 in the SystemVerilog 3.1 LRM on the Assertion API serves as a model of this strategy.