SV-CC Proposal for Consideration: VPI Standard (1800-2005) Changes for Packed-Arrays and Related Properties & Methods

C. Berking 11/17/2006

Introduction

The following proposal had begun as several loosely-related proposals for improvements to the VPI data model to handle packed-arrays, and eliminate related limitations of the current model. It became clear very quickly that these improvements would not be fully appreciated without describing them as a **set** of features, since their implications are closely interrelated, and best understood together, i.e. as a system. It is therefore suggested that you consider the proposal in its entirety before raising specific concerns.

Given the new complexities present in SV designs, features have been chosen to facilitate efficient application design. This is the critical factor that should motivate creation of additional objects, methods and properties when compatibility is not affected.

Glossary of essential terms

Slice An indexed contiguous part of a packed-array that includes one or more dimensions itself, i.e.,from declaration "logic [1:0][2:5][6:8] myvar", a slice could be myvar[1], or myvar[0][2:3], etc. Note that this is not to be confused with the proposed 'vpiSlice' object, which is intended to represent only the latter form.
Subpart A **slice** with no part-select-like ranges specified, i.e. the first form described above, 'myvar[1]'.
Element The smallest unit of an array that has a datatype consistent with its declaration, i.e. with either all explicit unpacked ranges removed (if array is unpacked) or all explicit packed ranges removed. Another way of saying this is the **subpart** of an array with the fewest number of dimensions of its own whose vpiParent is still the array.

A) The Case For New Objects to Represent Packed-Arrays of Structs and Unions

The current VPI model has no separate objects to designate packed-arrays of variable-width net/variable objects. The implications of this are that the object net/variable types themselves must represent both scalar and/or multi-dimensional vector (packed-array) forms. For example:

```
logic logicElem;
logic [1:0] logic1dim;
logic [1:0][2:5] logic2dim;

bit bitElem;
bit [1:0] bit1dim;
bit [1:0][2:5] bit2dim;

struct packed { bit [7:0] b1; logic [2:0] r1 } structElem;
struct packed { bit [7:0] b1; logic [2:0] r1 } [1:0] struct1dim;
struct packed { bit [7:0] b1; logic [2:0] r1 } [1:0][2:5] struct2dim;
```

There is no alternative in 1800-2005 to representing the above declarations as follows:

logicElem, logic1dim, logic2dim:	vpiLogicVar (vpiReg)
bitElem, bit1dim, bit2dim:	vpiBitVar
structElem, struct1dim, struct2dim:	vpiStructVar

It has been suggested in Mantis item #01230 that all the above forms except "structElem" be represented as either vpiBitVar or vpiLogicVar (and presumably vpiLogicNet) types, thus preserving the "vector-like" qualities of all three object types.

This works fine for vpiLogicVar (and vpiLogicNet), and vpiBitVar, but the system breaks down when applied to vpiStructVar's (and similarly for vpiUnionVar, vpiStructNet, vpiUnionNet). Consider for these types, that:

1. The bottom-most range (rightmost packed-dimension) is *implicit*, unlike vpiBit/Logic types. For example 'struct1dim[1][0]' is a bit-select, but like 'integer' and 'byte' types, there is no explicit range for index '[0]' (does 'struct1dim' have 2 ranges or 1 ?).
2. Representing these as vpiBit/Logic type objects would be inconsistent with their declarations. This means, for example, that applications that need to characterize packed-arrays as such would be *forced* to extract the **element** type of each vpiBit/Logic object to do so (every vpiReg would be a potential packed-array of packed-structs !). This has the potential complicate the upgrading of older 1364-compatible applications, and slow down both new and old applications, at the very least.

The following new object types should be considered for adding to the standard to represent packed-arrays of struct and union variable and net types:

vpiPackedArrayVar
vpiPackedArrayNet

And their corresponding typespec object type:

vpiPackedArrayTypespec

Most existing transitions and properties that apply to vpiArrayVar and vpiArrayNet would apply similarly to these new types, with some implications as follows:

vpiSize — This property would return the total *number of bits* of the packed-array, rather than the number of elements. It is deemed slightly more important to be consistent with the other packed-array types (vpiLogicVar, vpiBitVar, and vpiLogicNet), rather than with vpiArrayVar, for which vpiSize indicates the number of **element**s.

vpiVector — This property will always return '1' (true) for the new packed-array types (and thus vpiScalar would return '0' (false) accordingly).

vpiReg	This transition would allow iteration over the **element**s of the packed-array, just as it does with an unpacked-array.
vpiArrayMember	This property will return '1' (true) for all **element**s of the packed-array objects.
vpiMember	This transition would *not* apply to the new packed-array objects directly, but would clearly be applicable to an **element.**

Changes to the behavior of other transitions and new properties applicable to these new types are described in the following sections.

B) The Case For Clarifying and Extending vpiElemTypespec

Getting typespec handles for packed array **element**s is problematic. The existing vpiElemTypespec transition (see 27.17) would appear to be the correct method to use for this, however this is only legal for unpacked array typespecs (vpiArrayTypespec). Moreover, it's function is not clearly defined; the data model has no detail information as to what an "Elem" means. It is suggested that this should be clearly defined as **element**, as defined in the "Terminology" section. This would imply that vpiElemTypespec would extract the typespec handle for the **element** of an array in one step, given an array typespec handle. Furthermore, this should be extended to operate on packed-arrays as well, allowing applications to extract bit-level vpiLogicTypespec, or vpiBitTypespec handles, respectively, from vpiLogicTypespec or vpiBitTypespec handles representing packed-arrays, and vpiStruct/UnionTypespec handles from vpiPackedArrayTypespecs (see table below). Note that the choice of what is meant by **element** avoids bypassing struct/union levels for the new packed-array typespec object, i.e. a bit-level typespec should not be returned for an **element** of a packed-array of struct/union objects!

Multi-Dim Object	**Typespec**	**Element**	**vpiElemTypespec**
	..... Unpacked Array Types		
vpiArrayVar	vpiArrayTypespec	< variable object >	<variable typespec>
vpiArrayNet	vpiArrayTypespec	< net object >	< net typespec >
	..... Packed Array Types		
vpiLogicVar	vpiLogicTypespec	vpiLogicVar	vpiLogicTypespec
vpiLogicNet	vpiLogicTypespec	vpiLogicNet	vpiLogicTypespec
vpiBitVar	vpiBitTypespec	vpiBitVar	vpiBitTypespec
vpiPackedArrayVar	vpiPackedArrayTypespec	vpiStructVar	vpiStructTypespec
		vpiUnionVar	vpiUnionTypespec
vpiPackedArrayNet	vpiPackedArrayTypespec	vpiStructNet	vpiStructTypespec
		vpiUnionNet	vpiUnionTypespec

C) The Case For Adding New Subpart Methods, Especially vpiSubTypespec.

SystemVerilog allows partial indexing of packed-arrays to denote **subpart**s, as we have defined them here. For example:

```
logic [1:0][2:5][6:8] myvar;

initial
   myvar[1] = 0;
```

The vpiLogicVar packed-array 'myvar' is indexed only in the first range, implying that 'myvar[1]' is actually a 12-bit vector **subpart** with two ranges "below" this level. In terms of its properties as a design object, it is itself indistinguishable from a two-dimensional vector, thus it is consistent in this regard to call this a vpiLogicVar.

Characterizing datatypes for multidimensional objects (packed or unpacked) becomes inaccurate, or at best difficult, when their datatypes are constructed incrementally using typedefs from different sources. For example, if the above 'myvar' declaration was instead constructed as follows:

```
package P;
   typedef  logic [2:5][6:8] mytype;
endpackage

module m;
 import P::*
 mytype [1:0] myvar;
```

The typespec handle for 'myvar' is of type vpiLogicTypespec, but we cannot accurately represent its vpiName property (the name of the typedef used to define it, or NULL) as 'mytype' here, since this is not complete (the explicit range "[1:0]" is not included in the *definition* for "mytype"). The vpiLogicTypespec handle cannot be decompiled to indicate this form of incrementally constructed type without portraying the 'mytype' typedef inaccurately, or ignoring the fact that a typedef was used here.

vpiSubTypespec (or vpiSubpartTypespec if preferred) is proposed to remedy this. This method allows incremental unwinding of multi-dimensional typespecs, such that compound datatype definitions of this type can be accurately decompiled. This allows VPI applications to decompile such declarations accurately in terms of the way they were originally specified.

For example, applying the vpiSubTypespec method progressively to a vpiLogicTypespec handle for 'myvar' above, we get:

	Ranges	vpiSize	vpiName

TSH0: Initial typespec for ‘myvar’	[1:0][2:5][6:8]	24	NULL
TSH1: vpi_handle(vpiSubTypespec, TSH0)	[2:5][6:8]	12	‘mytype’
TSH2: vpi_handle(vpiSubTypespec, TSH1)	[6:8]	3	NULL
TSH3: vpi_handle(vpiSubTypespec, TSH2)	-- none --	1	NULL

Similarly, the vpiSubTypespec method should allow incremental unwinding of *unpacked* array object datatypes too.

Optionally, to allow full VPI access to **subpart**s of *objects* (that do not appear as such in the design), an analogous method vpiSubPart could be added.
This would allow “incremental” indexing of packed-array (and unpacked array) objects. It would operate much like vpiReg iterations do for vpiArrayVar’s, except that it would iterate over only one index (dimension level) of the reference object for a given iteration.

D) New Property Recommendations

Given the proliferation of new objects, and their possible multi-dimensional combinations, it is surprising that there is no property that relates directly to dimensionality. It seems grossly inefficient to require the application developer to create a vpiRange iteration and their respective handles to determine if there is one or more dimensions, or worse, determine how many.

The following two new properties are proposed to address this need:

vpiArrayDims Returns number of *unpacked* dimensions of vpiArrayVar, vpiArrayNet, or vpiArrayTypespec

vpiPackedDims Returns number of packed dimensions of any packed-array object (vpiLogicVar, vpiLogicNet, vpiBitVar, vpiPackedArrayVar, or vpiPackedArrayNet) or typespec (vpiLogicTypespec, vpiBitTypespec, or vpiPackedArrayTypespec). In addition, it is allowed on *unpacked* array objects (listed above) as well, in order to characterize packed-dimensions *underneath* unpacked ones.

Examples:

```
bit [1:0][2:5] mybit;                // vpiPackedDims == 2

bit [1:0][2:5] mybundles [3:0];   //  vpiArrayDims == 1, and vpiPackedDims == 2

struct packed {
    bit [7:0] b1;
    logic [2:0] r1 } [1:0][2:5] struct2dim;    // vpiPackedDims == 2
```

Note that vpiPackedDims for ‘struct2dim’ does *not* include the *implicit* vector dimension of the struct itself.