## *Issue 1: Organization of an unpacked array of packed arrays.*

### Scope

The current definition of organization of unpacked array of packed arrays does not appear aligned either with the examples given in the standard or the organization of packed arrays themselves. This section explains the understanding and why this conflict appears.

### Explanation

The explanation below builds up from the contents of the standard to converge at a uniform conclusion. It points a contradiction and tries to arrive at an inference that will address the conflict identified.

**Section F.1**

```
/* common type for 'bit' and 'logic' scalars */
typedef unsigned char svScalar;

typedef svScalar svBit;   /* scalar */
typedef svScalar svLogic;  /* scalar */

/* Canonical representation of packed arrays */
/* 2-state and 4-state vectors, modeled upon PLI's avalue/bvalue */
#define SV_CANONICAL_SIZE(WIDTH) ((WIDTH)+31)>>5)

typedef unsigned int svBitVec32; /*  (a chunk of) packed bit array */
typedef struct {unsigned int ; unsigned int d; } svLogicVec32; /* (a chunk of) packed
logic array */
```

**Section E.9.2**

Some macros that are allowed to be vendor specific representation are as follows:

```
#define SV_BIT_PACKED_ARRAY(WIDTH, NAME) …
#define SV_LOGIC_PACKED_ARRAY(WIDTH, NAME) …
```

A possible definition for this could be

```
#define SV_LOGIC_PACKED_ARRAY(WIDTH, NAME) \
                svLogicVec32 NAME[SV_CANONICAL_SIZE(WIDTH)]
```

Section E.9.4 Example 3 – source-level compatible application

**System Verilog:**

```
  Typedef struct {int a; bit [6:1][1:8] b [65:2]; int c; } triple;
       // troublesome mix of C types and packed arrays
   import "DPI" function void foo (input triple i);
```

**C:**

```
typedef struct {
      Int  a;
      Sv_BIT_PACKED_ARRAY(6*8, b) [64]; /* implementation specific
                                           representation */
       int   c;
 } triple;
```

The definition for b in C will translate into:

```
svBitVec32 b(2)(64);
```

This is an inference from the application of the example macro given above:

```
Sv_BIT_PACKED_ARRAY(6*8, b)[64]  =>
        SvBitVec32 b[SV_CANONICAL_SIZE(48)][64] =>
        SvBitVec32 b[SV_CANONICAL_SIZE(48)][64] =>
        svBitVec32 b[(48+31)>>5][64]
        svBitVec32 b[2][64]
```

This results in the following data organization for the packed array:

| | Col 0 | Col 1 | Col 2 | | | | Col 61 | Col 62 | Col 63 |
|---|---|---|---|---|---|---|---|---|---|
| Row 0 | | | | | | | | | |
| Row 1 | | | | | | | | | |

In the above figure, Row 0 and Row 1 represent the two words it will take to contain the 48 bits of a 48 bit vector as per the canonical size definition. The columns as per this definition represent the dimensions of the unpacked array containing the packed arrays. From this organization, if a user had to pick the packed element indexed at position 32 then the following will have to be done:

```
Elem[0] = B[0][31];
Elem[1] = B[1][31];
```

A direct indexing into the data structure to access a linearly organized memory will not be possible.

Going by the above definition as per the macro, we have ended up with a column major ordering for the unpacked array of 64 packed arrays of forty eight elements each. As per the above ordering, the unpacked portion of the arrays is changing faster than the packed portion (Note that 64 coming as a later dimension means each of the two rows contains an element of the packed array.)

The organization logically should have been of a packed array element of an unpacked array following the previous packed element. In such a scenario the organization would appear something as follows:

Sv_BIT_PACKED_ARRAY(6*8, b)[64] =>
svBitVec32 b[64][SV_CANONICAL_SIZE(48)] =>
svBitVec32 b[64][(48+31)>>5] =>
svBitVec32 b[64][2]

This will result in the following data organization for the packed array:

| | Col 0 | Col 1 |
|---|---|---|
| Row 0 | | |
| Row 1 | | |
| Row 2 | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| Row 61 | | |
| Row 62 | | |
| Row 63 | | |

In the above organization you have the packed array elements placed consecutively and can be picked by accessing the index I of the unpacked array portion.

In the above figure, Col 0 and Col 1 represent the two words it will take to contain the 48 bits of a 48 bit vector as per the canonical size definition. The rows as per this definition represent the dimensions of the unpacked array containing the packed arrays. From this organization, if a user had to pick the packed element indexed at position 32 then the following will have to be done:

Elem = &B[31];

A direct indexing into the data structure to access a linearly organized memory is possible in this case. This kind of an organization is what appears to be implied in the examples provided in the standard. But the macros and the definitions are running counter-intuitive to this organization.

Now I will extending this case further to the example given for the structure definition in E.9.4. The piece of C code that is being used to access the packed array elements of the unpacked array in the structure is the following:

```
Void foo(const triple *i)
{
int j;
/* canonical representation */
svBitVec32 arr[SV_CANONICAL_SIZE(6*8)]; /* 6*8 packed bits */
…
```

```
…
for (j=0; j<64;j++) {
        svGetBitVec32(arr, (svBitPackedArrRef)&(i->b[j]), 6*8);
…
}
```

What I understand from the above code is that:

- System Verilog passed memory organization will confirm to the C layout of the structure
- The C layout expects an array with 64 rows, each containing an element of 48 bits

Now, if we go by the definition of macros as specified currently in the standard, the layout of the structure as per the definition given in the standard will be:

| **integer a** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Array B** | Col 0 | Col 1 | | | | | Col 61 | Col 62 | Col 63 |
| Row 0 | | | | | | | | | |
| Row 1 | | | | | | | | | |
| **Integer c** | | | | | | | | | |

This layout certainly does not confirm to what the C example provided above is expecting.

Case 2 is that of the following data organization:

| **Integer a** | | |
|---|---|---|
| **Array b** | Col 0 | Col 1 |
| Row 0 | | |
| Row 1 | | |
| | | |
| | | |
| | | |

| | | |
|---|---|---|
| | | |
| | | |
| | | |
| | | |
| Row 61 | | |
| Row 62 | | |
| Row 63 | | |
| **Integer c** | | |

With this kind of data organization the C code specified above will be able to access each of the 64 rows containing a packed array each with 48 bits.

## Inference on Organization of Unpacked arrays of Packed Arrays

Going with the above example I think that in the C definition of an unpacked array of packed arrays, the packed dimension should be the rightmost definition of the C data type. The unpacked dimensions should be the first ones to appear on the C data type definition. This would mean that the organization will move away from a column major organization of the data to a row major organization.

This appears to be the understanding of the examples given in the standard and also will accurately represent the data organization of the packed arrays, as packed dimensions are the most closely associated in memory layout also.