"System Verilog Tagged Unions and Pattern Matching"

(An extension to System Verilog 3.1 proposed to Accellera)

Bluespec, Inc.

Contact:
Rishiyur S. Nikhil, CTO, Bluespec, Inc.
c/o Sandburst Corp.,
600 Federal St., Andover, MA 01810, USA
Email: nikhil AT bluespec.com
Phone: +1 (978) 689 1679

Original: September 9, 2003
Revision: October 3, 2003

Abstract

System Verilog has structure and union types. We propose related extensions: a qualification on unions for "Tagged unions", together with pattern matching on tagged unions and structures. These yield the following benefits: (1) complete type-safety (which ordinary unions lack), (2) greater brevity, (3) a more "visual" (*i.e.,* readable) way of programming with unions and structures, (4) direct expressions to build tagged union values, usable in arbitrary expression contexts, (5) zero implementation overhead, and (6) customizable bit representations. These properties raise the level of programming with structures and unions, thereby eliminating a number of common errors and making programs more readable. The proposed constructs are synthesizable, and the underlying ideas have been used in some languages for many years.

October 3 Revision Notes: Simplifies the proposal, integrates better into the language, simplifies parsing. Tagged unions are now just a qualification on unions.

Contents

The actual extensions to the LRM are quite small (BNF changes: 2 lines changed, 3 lines added) and are described in a few pages in the Appendices.

Most of this document is about rationale and motivation (intended for the Accellera committees and not part of the LRM). It describes the extension, gives examples, compares it to existing constructs, and discusses implementation issues.

1 Background concepts: tagged and untagged unions

Mathematics has the concepts of *untagged* unions ($A + B$, or $A \cup B$) and *tagged unions* ($A \oplus B$). The latter are also called *discriminated unions* or *discriminated sums*. The difference is the following.

In untagged unions, when a component value (of type A or B) is injected into a union value (of type $A+B$), it loses its identity, *i.e.,* there is no way to know which summand it came from. This loss of information is the source of type-loopholes, because it is possible to inject an A value into an $A + B$ value and then to project it out as a B value, thereby misinterpreting the representation (bits).

In tagged unions, a union value (of type $A \oplus B$) always has a *tag* that "remembers" which summand it came from, so that it is possible to examine a union value to determine which summand it came from. This allows correct (type-safe) projection of the contained value back into the summand domains.

The **union** construct in System Verilog (and C/C++) corresponds to untagged unions. This proposal introduces tagged unions.

2 Motivating Examples

Example 1: Imagine that we are designing hardware in which some processor has two kinds of instructions: Add and Jump. An Add instruction contains three 5-bit register names (two sources and a destination). A Jump instruction is either unconditional and contains a 10-bit immedate address offset, or conditional and contains a register name (for destination address) and a 2-bit condition-code register name.

Note that the certain members are meaningful only in certain contexts. For example, the destination register member is meaningful only in an Add instruction. A condition-code register member is meaningful only in a conditional Jump instruction.

Example 2: Imagine that we are designing hardware in which we wish to represent an integer together with a "valid" bit.

Note that the integer member is meaningful only if the valid bit is set.

Both these examples can be expressed using existing struct and union notations. We will show how they can be improved significantly using tagged unions. The primary benefit is type-safety (a verification benefit) because members can be examined/assigned only when they are meaningful. Additional benefits include greater brevity and a more "visual" (readable) notation.

3 Background: SystemVerilog 3.1 structs and unions

In this section we recap some features of existing structs and unions in System Verilog 3.1, which will be improved/fixed in the proposed tagged unions introduced in § 4.

Members of structs and unions are set and accessed only using traditional "dot-notation" (*structure.member*). The tagged union proposal improves this with pattern matching and tagged union expressions.

There are several potential non-orthogonalities in the facilities for struct and union "values", *i.e.,* some facilities are missing, and some can be used only in limited contexts:

- An entire struct value can be transferred in an assignment, or in argument- and result-passing. But the LRM seems to be silent on whether union values can be similarly passed (LRM § 3.11), possibly because unions are untagged and so union values are considered to be the same as summand values (so, presumably, a union inherits its assignment semantics from the summands).
- There are two very similar but separately described constructs for creating structure values: "structure literals" (LRM § 2.8) and "structure expressions" (LRM § 7.13).
- There do not seem to be analogous union values (union literals or union expressions), perhaps because unions are untagged, so a summand literal/expression can be used directly.
- The syntax of structure expressions (LRM § 7.13) overlaps with the syntax of bit-concatenation (braces and commas). Hence structure expressions can be used only in limited contexts (*e.g.,* as the top-level of a right-hand side of an assignment to a structure variable, see LRM § 7.13).

The **packed** qualifier allows structs and unions to be considered as bit-vectors. But in a packed union, all members must be packed elements with the same size (we drop this restriction in tagged unions).

3.1 Simulating tags using existing structs and unions

Tagged unions can be simulated by manual coding with existing structs and unions, but this is not type-safe, and they are generally not as concise or visually obvious.

Here are some definitions for Example 1.

```
typedef enum { A, J } Opcode;
typedef enum { JC, JU } JumpOpcode;

typedef struct {
    Opcode  op;
    union {
        struct {
            bit [4:0] reg1;
            bit [4:0] reg2;
            bit [4:0] regd;
        } A_operands;
        struct {
            JumpOpcode jop;
            union {
                bit [9:0] JU_operand;
                struct {
                    bit [1:0] cc;
                    bit [4:0] addr;
                } JC_operands;
            } J_suboperands;
        } J_operands;
    } operands;
} Instr;
```

Note that the `op` member acts as a "tag" which indicates how to intepret the remaining bits, *i.e.,* as add operands or as jump operands. Similarly, the `jop` member acts as another tag which indicates whether to interpret the remaining members as unconditional or condition jump operands.

A typical usage, employing dot-notation to access components:

```
Instr instr;

...
case (instr.op)
  A: rf [instr.operands.A_operands.regd] =
         rf [instr.operands.A_operands.reg1] +
         rf [instr.operands.A_operands.reg2];
  J: case (instr.operands.J_operands.jop)
       JU: pc = pc + instr.operands.J_operands.J_suboperands.JU_operand;
       JC: if (cf [instr.operands.J_operands.J_suboperands.JC_operands.cc])
              pc = instr.operands.J_operands.J_suboperands.JC_operands.addr;
     endcase
endcase
```

where `rf` is the main register file and `cf` is the condition-code register file. Often, programmers use macros to abbreviate such multi-level dot-selections.

Lack of type-safety: Extracting a union member opens a type-loophole. For example, we can set the tag to "A" and assign "J" members:

```
Instr instr;

instr.op                        = A;
instr.operands.J_operands.jop = JC;    // meaningless when op == A
```

Or, when the tag is "A" we can still examine "J" members:

```
case (instr.op)
    A : ...
              instr.operands.J_operands.jop ...    // meaningless since tag is A
endcase
```

This lack of type-safety introduces a verification obligation to ensure that members are only used meaningfully.

Note: very occasionally, this kind of type loophole, or "type laundering", is exactly what the designer wants, because he *intends* to view the same bits in two different ways. It might be better to make those (dangerous) situations clearly visible using an explicit 'cast' operation.

4 Proposed Extensions: Tagged Unions and Pattern Matching

We propose to add a new construct called a "Tagged Union" type, together with an extension to case-statements called "Pattern Matching", and an extension to expressions for constructing tagged union values. With these, our examples can be rendered:

- with complete type-safety (so, simpler verification),
- with greater brevity, and
- with a more visually apparent notation

These properties raise the level of programming with structures and unions and thereby eliminate a number of common errors and make programs more readable.

4.1 Tagged union types

We extend the syntax of types to include tagged unions. The keyword "`tagged`" can prefix a "`union`" type, making it a tagged union. The identifier for each summand can now be regarded as a tag for that summand.

Here is our 2-instruction example, using a tagged union:

```
typedef tagged union {
    struct {
        bit [4:0] reg1;
        bit [4:0] reg2;
        bit [4:0] regd;
    } A;
    tagged union {
        bit [9:0] JU;
        struct {
             bit [1:0] cc;
             bit [9:0] addr;
        } JC;
    } J;
} Instr;
```

Here, `Instr` is declared as a new tagged union type. `A` and `J` are now also *tags* for the summands. The representation of a tagged union value always contains enough additional bits to contain the tags. In this example, the tag occupies 1 bit and plays the role of the opcode (A or J). Similarly, `JU` and `JC` are tags for the nested union.

The scope of the identifiers defined in a tagged union is exactly the same as in ordinary unions.

Note that the original version using ordinary unions required several extra levels of nesting, with several corresponding extra member names. The orginal opcode members are not necessary in the tagged union because the tags now carry that information.

4.2 Tagged union expressions

We extend the syntax of expressions to include tagged union expressions (or tagged union literals). These are expressions that evaluate to tagged union values, and can be used in any expression context. A tagged union expression consists of the keyword "`tagged`" followed by a tag identifier, optionally followed by an expression for the the corresponding summand.

expression ::=
    ...
  | **tagged** identifier expression
  | **tagged** identifier ()
  | **tagged** identifier

The first form is used in most situations, and of course the expression must have the correct type for the summand represented by the tag identifier. The second and third forms are used only when the tag identifier is for a summand of `void` type (this is similar to the syntax for tasks and functions with void arguments).

Examples of tagged union expressions, evaluating to tagged union values:

```
tagged A { e1, e2, ed }                          // struct members by position
tagged A { reg2:e2, regd:ed, reg1:e1 }           // by name
tagged J (tagged JU eOffset)
tagged J (tagged JC { eC, eAddr })               // inner struct by position
tagged J (tagged JC { cc:eC, addr:eAddr })       // by name
```

Here, the expressions in braces are structure expressions using the standard syntax.

4.3 Pattern Matching

We extend the syntax of case statements to include structure and tagged union patterns on the left-hand-side of each case item. Here is our example, again:

```
Instr instr;

...
case (instr)
  tagged A {r1,r2,rd} : rf[rd] = rf[r1] + rf[r2];
  tagged J j          : case (j)
                          tagged JU  a   : pc = pc + a;
                          tagged JC {c,a}: if (cf[c]) pc = a;
                        endcase
endcase
```

Here, the pattern "`tagged A {r1,r2,rd}`" will "match" tagged union values that have tag A, and in that case implictly declares and binds the variables `r1`, `r2` and `rd` to the values of the members `reg1`, `reg2` and `regd`, respectively. `r1` is implicitly declared to be of type `bit [4:0]`, and similarly for `r2` and `rd`. These variables can then be used in the statement after the "`:`", *i.e.,* the scope of these declarations is the RHS of the same case item. It is a type checking error if instead of `{r1,r2,rd}` we had a pattern that could not match the summand corresponding to tag "`A`" (*e.g.,* a 4-member structure or something that was not a structure).

Similarly, the pattern "`J j`" matches only tagged union values that have tag J, and in that case implicitly declares and binds the variable `j` to the nested tagged union value. The nested case statement further discriminates `j` between `JU` and `JC`.

Patterns can be nested, so the above example can also be written directly using a single case statement as follows:

```
case (instr)
  tagged A {r1,r2,rd}       : rf[rd] = rf[r1] + rf[r2];
  tagged J (tagged JU a)    : pc = pc + a;
  tagged J (tagged JC {c,a}): if (cf[c]) pc = a;
endcase
```

Observe the substantial increase in brevity, and the more "visual" access to the members of the structures due to pattern matching. In particular, tagged union expressions, which are used to build tagged union values, look exactly the same as tagged union patterns, which are used to deconstruct such values. They both suggest the "layout" of the structure.

The above example used "positional" pattern matching. When the summand is a struct, pattern matching can also be done by name (in which case the ordering of the members is not relevant):

```
case (instr)
  tagged A {reg2:r2,regd:rd,reg1:r1} : rf[rd] = rf[r1] + rf[r2];
  tagged J (tagged JC {cc:c,addr:a}) : if (cf[c]) pc = a;
endcase
```

Further, when done by name, we can omit members that are not of interest in a particular case item.

In a case statement, pattern matching is attempted sequentially, from first case item to last case item. In particular, if more than one pattern matches the case value, the first one is selected.

Patterns can also be used with ordinary structs, and with integral and string values. Example:

```
case (instr)
  tagged A a: case (a)
                {r1,r2,0}  : ; // No op
                {r1,r2,rd} : rf[rd] = rf[r1] + rf[r2];
              endcase
  ...
endcase
```

Here, "a" is bound to the structure inside an "A" tagged union. This, in turn, is matched in the inner case statement with one of two structure patterns. The first structure pattern matches if the destination register is Register 0 (traditionally a "throw away the value" register). The second structure pattern matches all the remaining possibilities.

The set of patterns in a case statement need not be exhaustive. The usual **default** mechanism can be used as a final catch-all if all patterns fail.

Note: there are well-known techniques in the literature describing how to compile this kind of pattern matching into deterministic decision trees avoiding repeated tests, exploiting mutual exclusion, etc.

4.3.1 Optional extension: pattern matching in if statements

An option to this proposal is to extend if statements so that the predicate uses pattern matching:

```
if (e matches tagged J (tagged JC {cc:c,addr:a}))
    ...     // c and a can be used here
else
    ...
```

Here, "`matches`" is a new keyword. If the value of expression `e` matches the pattern, the then-arm is executed, otherwise the else-arm is executed. The scope of the variables `c` and `a` is the then-arm, and they will be bound to the values of the corresponding members.

This example would be much more verbose if written using ordinary struct-and-union notation and without the implicit declaration of pattern variables:

```
x = e;
if ((x.op == J) &&
    (x.operands.J_operands.jop == JC))
  begin
    bit [1:0] c;
    bit [4:0] a;
    c = x.operands.J_operands.J_suboperands.cc;
    a = x.operands.J_operands.J_suboperands.addr;
    ...     // c and a can be used here
  end
else
    ...
```

(this relies on the sequential evaluation of the `&&` operator, since its right operand is meaningful only if the left operand is true).

4.4 Type-safety and verification

All these examples are completely type-safe because the compiler ensures that the `reg1`, `reg2` and `regd` members can be accessed only in the A case, and the `cc` and `addr` members can be accessed only in the J/JC case. It is syntactically impossible, for example, to access the `reg1` member in the J case.

This type-safety directly impacts verification. If the same functionality were manually coded using existing structs and unions, then we are left with a verification obligation to ensure that members are accessed/updated meaningfully with respect to the tags. Here, that obligation is discharged automatically by static type-checking based on the syntactic structure.

Note: by 'type-safe' we mean that it is impossible to *misinterpret* or "launder" the bits in a tagged union, the way it is possible to do in an ordinary (untagged) union. Tagged unions can still, of course, raise run-time errors, but these situations are can now be reported, and are controllable.

4.5 Dot notation to select and assign members type-safely

"Dot notation" can still be used to access and assign members of a tagged union:

```
x            = instr.A.reg1

instr.A.reg2 = eNew;
```

but these will now be completely type-safe, because they are defined to be equivalent to:

```
case (instr)
    tagged A {r1,r2,rd} : x = r1;
    default             : $error (....);
endcase

case (instr)
    tagged A {r1,r2,rd} : instr = tagged A {eNew,r2,rd};
    default             : $error (....);
endcase
```

i.e., the access or assignment is only allowed if the tag has the correct value.

4.6 Packed representations (canonical)

Tagged unions can be packed, using the **packed** keyword, just like structs and unions, provided their components are packed items.

Tagged unions have a simple, orthogonal and transparent canonical representation in bits (non-canonical, or custom, representations, which are also useful in real hardware, are discussed in § 4.8). The canonical representation of a tagged union value is:

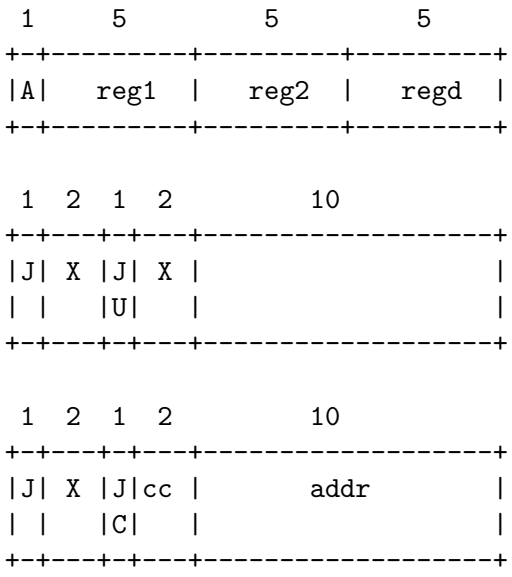
```
+-----+-------+--------+--------+-----+--------+
| tag | XXXXX | field1 | field2 | ... | fieldN |
+-----+-------+--------+--------+-----+--------+
```

where the `fieldJ`'s are appropriate for the `tag`.

Unlike the requirement for unions, the tagged union summands do *not* have to be of the same size. A tagged union value, no matter what the current tag value, has a fixed, definite size which is easily and transparently apparent from the summand sizes.

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the summands.
- The size of the tag is the minimum number of bits needed to represent the number of summands.
- The tag bits are always left-justified (*i.e.,* towards the most-significant bits).
- For each summand, the summand bits are always right-justified (*i.e.,* towards the least-significant bits).
- The bits between the tag bits and the summand bits are undefined (the "XXX"s in the figure above). In the extreme case of a summand of `void` type, only the tag is significant and all the remaining bits are undefined.

Following these principles, the representations for Example 1 is directly evident from its type declaration:



These representation choices make tagged union values synthesizable with efficient circuits (tags and fields are always at known bit positions).

Note to committee: The particular choice of the tag located at the upper-order bits and the summand contents right-justified at the lower-order bits is arbitrary. But it should be specified, so that there is no ambiguity about representation.

4.7 Zero implementation overhead

Tagged union values have zero overhead with respect to the number of bits in their representation, compared to manually coding the same functionality explicitly with structs and unions. The struct-and-union version of the `Instr` structure is represented in 16 bits: 1 bit for the opcode and 15 bits for the largest summand. The tagged union version is also represented in 16 bits: 1 bit for the tag and 15 bits for the largest summand.

Note: in the limit case of a tagged union with just one summand, 0 bits are needed for the tag, i.e., there is no representation overhead at all. This is sometimes useful in place of an ordinary struct, allowing the use of pattern matching to access the members instead of dot notation.

Similarly, the circuits produced for case statements with tagged union values are exactly the same as those that would be produced if the same functionality were coded manually using unions and structs.

4.8 Non-canonical (custom) representations

Sometimes, the designer wants a non-canonical (usually non-orthogonal) representation. For example, processor instruction encodings are typically full of special tricks to fit an instruction into the shortest possible instruction word, where the packing is highly dependent on the particular opcode, sub-opcodes, and so on.

To support non-canoncial encodings, the designer can customize the representation of a tagged union by defining two methods **pack** and **unpack** associated with the tagged union type. This is allowed only for packed tagged unions. For example,

```
function Instr      Instr.unpack (Bit [n:0] bits);
function Bit [n:0]  Instr.pack   (Instr instr);
```

If the designer provides these function definitions in the same scope as the tagged union typedef for `Instr`, the compiler will automatically use these functions to convert a literal tagged union into its packed bit-representation, and to unpack the members out of a packed tagged union value in a pattern-match.

With this mechanism, the designer can choose any preferred representation, which sometimes can be more efficient than the canonical representation. For example, suppose we want to represent something that is:

- Either a 32b byte pointer, but which is always word-aligned, or
- a 31b immediate integer value.

(This is a standard representation in garbage-collected languages.) The tagged union definition for this might be:

```
typedef tagged union {
    bit [31:0]  Ptr;
    bit [30:0]  Immed31;
} PtrOrImmed;
```

The canonical (orthogonal) representation would take 33 bits (32 bits for the pointer, plus 1 bit for the tag). However, by defining an explicit **pack** and **unpack**, we can represent it in 32 bits, exploiting the fact that `Ptr` LSBs are always zero due to word-alignment. We can represent `Ptr` summands just using their 32 bits (LSB always 0), and `Immed31` summands as `{31b_value, 1'b1}`, with LSB always 1.

To get a non-canonical representation, the designer simply provides definitions for **pack** and **unpack**. Except for this, the tagged unions are used in expressions and pattern matching in the normal way. It is straightforward for the compiler automatically to insert the pack/unpack routines wherever necessary while generating code for tagged union expressions and pattern matching. Again, this automation removes another potential source of programming errors.

5 Example 2

Example 2 (an integer together with a valid bit) can be expressed as follows:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;
```

The representation will have 33 bits: 32 bits for the int, plus 1 bit for the tag (but note, following the discussion on non-canonical representations in § 4.8, if a particular application does not use all possible integer values, `VInt` can be represented in 32 bits by using one of the unused values to encode the invalid summand).

A valid `VInt` value is constucted by the expression:

```
tagged Valid e
```

where `e` is an integer expression. The "invalid" `VInt` value is constructed by:

```
tagged Invalid ()
```

or by:

```
tagged Invalid
```

A `VInt` value `v` can be examined using pattern matching:

```
case (v)
  tagged Invalid  : $display ("Is invalid");
  tagged Valid  x : $display ("Valid with value %d", x);
endcase
```

Note: again, it is syntactically impossible to extract an int value from a `VInt` value that has the `Invalid` tag.

6 Maturity of the proposed constructs

All these constructs have been implemented and well-tested for over a decade in many high-level programming languages, including Haskell and SML. There are plenty of of papers in the literature on how to implement tagged unions and pattern matching efficiently.

Tagged unions have also been implemented and used in the Hardware Description Language Bluespec for over 3 years. Tagged unions, tagged union expressions and pattern matching are eminently synthesizable into efficient hardware.

Although the ideas behind tagged unions and pattern matching are very mature from previous languages, to our knowledge this is the first time they are being cast into syntax that is consistent with System Verilog.

7 Some comments on the relationship to existing constructs

A tagged union in which each summand has the `void` type is equivalent to an enumeration. Example:

```
typedef enum { red, yellow, green } Colors;
```

is equivalent to:

```
typedef tagged union {
    void   red;
    void   yellow;
    void   green;
} Colors;
```

The representation needs exactly the same number of bits (2, in this example).

A tagged union with a single summand is equivalent to a struct. Example:

```
typedef struct {
    byte  b;
    int   i;
} s;
```

is equivalent to:

```
typedef tagged union {
    struct {
        byte  b;
        int   i;
    } T;
} s;
```

and is represented in exactly the same number of bits (because a single tag can always be represented in 0 bits).

Thus, in principle, tagged unions could subsume enumerations, structs and unions. However, since enumerations, structs and unions have a long tradition in C/C++ and are familiar to legions of programmers, we do not propose replacing them with tagged unions; we simply propose tagged unions and pattern matching as extensions that provide a new opportunity for type-safety, brevity, and a more visual style of programming.

A Additions to LRM Text

In the syntax box at the top of Section 3.11, prefix both the `union` *keywords with the optional keyword* `tagged`.

data_type ::= *// from Annex A.2.2.1*
  ...
| [**tagged**] **union packed** [signing] { { struct_union_member } } { packed_dimension }
  ...
| [**tagged**] **union** [signing] { { struct_union_member } }

At the end of Section 3.11, add the following text.

Tagged unions capture certain common usages of structs and unions. Together with tagged union expressions (Section 7.13+) and pattern matching (Section 8.4.1), they providing additional type-safety, brevity and readability.

In tagged unions, each struct_union_member is also known as a *summand*, and the identifiers declared by the summands are called *tags*. Conceptually, a tagged union is a like a union, but it also contains the tag itself to remember which summand the value came from. This, in turn, allows the contained value to be extracted with type-safety.

Example: an integer together with a valid bit:

```
typedef tagged union {
    void  Invalid;
    int   Valid;
} VInt;
```

Example: two kinds of instructions in a processor: Add and Jump. An Add instruction contains three 5-bit register names (two sources and a destination). A Jump instruction is either unconditional and contains a 10-bit immedate address offset, or conditional and contains a register name (for destination address) and a 2-bit condition-code register name.

```
typedef tagged union {
    struct {
        bit [4:0] reg1;
        bit [4:0] reg2;
        bit [4:0] regd;
    } A;
    tagged union {
        bit [9:0] JU;
        struct {
             bit [1:0] cc;
             bit [9:0] addr;
        } JC;
    } J;
} Instr;
```

In a packed tagged union, all summand types must also be packed types. The (standard) representation for a packed tagged union is the following.

- The size is always equal to the number of bits needed to represent the tag plus the maximum of the sizes of the summands.
- The size of the tag is the minimum number of bits needed to represent the number of summands.
- The tag bits are always left-justified (*i.e.,* towards the most-significant bits).
- For each summand, the summand bits are always right-justified (*i.e.,* towards the least-significant bits).
- The bits between the tag bits and the summand bits are undefined. In the extreme case of a summand of `void` type, only the tag is significant and all the remaining bits are undefined.

For greater control on representation of a tagged union type T (where the standard representation does not suffice), the representation can be customized simply by defining the following functions in the same scope as `T`'s declaration:

```
function T          T.unpack (Bit [n:0] bits);
function Bit [n:0]  T.pack   (T t);
```

These functions can perform arbitrary packing and unpacking to and from bits. Note, the type definition for T, and its uses in expressions and patterns are unaffected by this; this mechanism is simply a hook to customize the representation. Such non-standard representations may be necessary either to meet external representation requirements or to obtain a more efficient representation than the standard one (such as Huffman encoding).

The following is a new section to be added after Section 7.13:

Section 7.13+ Tagged union expressions

expression ::= *from Annex A.8.3*
  ...
  | tagged_union_expression

tagged_union_expression ::=
  tagged identifier expression
  | **tagged** identifier ()
  | **tagged** identifier

A tagged union expression (packed or unpacked) is built from the keyword `tagged` followed by a tag identifier optionally followed by an expression representing the value of the summand for that tag.

Examples (the expressions in braces are structure expressions (Section 7.13):

```
tagged Valid   e
tagged Invalid ()
```

```
tagged Invalid

tagged A { e1, e2, ed }                          // struct members by position
tagged A { reg2:e2, regd:ed, reg1:e1 }           // by name
tagged J (tagged JU eOffset)
tagged J (tagged JC { eC, eAddr })               // inner struct by position
tagged J (tagged JC { cc:eC, addr:eAddr })       // by name
```

Section 7.13+ Tagged union member access

An entire tagged union summand, or a sub-field within a tag union summand, can be read or assigned with the usual dot-notation, but the operation is valid only if the tagged union value has the correct tag. Examples:

```
x            = instr.A.reg1    // legal if instr has tag A

instr.A.reg2 = eNew;           // legal if instr has tag A
```

In general this is a runtime check but it can often be removed by optimization.

The following is a new sub-section to be added at the end of Section 8.4:

Section 8.4.1 Pattern matching

In a case statement, if the expression being tested is a structure or a tagged union, then *patterns* may be used on the left-hand side of each case item. A pattern is simply an expression built from the following syntax (this syntax is not in the formal grammar because it is simply a subset of the syntax of expressions):

pattern ::=
  identifier
| number
| string_literal
| **tagged** identifier
| **tagged** identifier ()
| **tagged** identifier pattern
| **{** pattern, ... ,pattern **}**
| **{** identifier : pattern, ... , identifier : pattern **}**

i.e., nested tagged union and struct expressions with identifiers, integral constants or string literals at the leaves. The identifiers in a pattern must be unique. In structure patterns with named members, the order of members does not matter, and some members may be omitted.

The value V being tested by the case statement is *matched* against the patterns in the case items, one case item at a time, in top-to-bottom (textual) order. The pattern matching rules are simple:

- An identifier pattern always succeeds (matches any value), and the identifier is bound to that value.
- A number or string literal pattern succeeds if V is equal to that value.

- a tagged union pattern succeeds if the value has the same tag, and if the nested pattern matches the summand contents of the tagged union.
- a structure pattern succeeds if each of the members patterns matches the corresponding members in the structure value.

Note that type-checking ensures that V is of the correct type for the pattern (number, string, structure of the correct type, tagged union of the correct type).

If a case-item's pattern matches successfully, that case-item is selected, and the identifiers in the pattern are bound to the corresponding members in the value, so that they can be used in the right-hand side of the case-item.

The identifiers in a pattern are implicitly declared to have the type of the corresponding members (this is statically determinable from the pattern and the tagged union declaration), and their scope is the right-hand side of the same case-item.

For struct components, pattern matching can be done either positionally or by name.

Example:

```
case (v)
  tagged Invalid    : $display ("Is invalid");
  tagged Valid  x   : $display ("Valid with value %d", x);
endcase
```

Example:

```
Instr instr;

...
case (instr)
  tagged A {r1,r2,rd} : rf[rd] = rf[r1] + rf[r2];
  tagged J j          : case (j)
                          tagged JU  a   : pc = pc + a;
                          tagged JC {c,a}: if (cf[c]) pc = a;
                        endcase
endcase
```

Example (nested patterns):

```
case (instr)
  tagged A {r1,r2,rd}       : rf[rd] = rf[r1] + rf[r2];
  tagged J (tagged JU a)    : pc = pc + a;
  tagged J (tagged JC {c,a}): if (cf[c]) pc = a;
endcase
```

Example (nested patterns and struct components by name):

```
case (instr)
  tagged A {reg2:r2,regd:rd,reg1:r1} : rf[rd] = rf[r1] + rf[r2];
  tagged J (tagged JC {cc:c,addr:a}) : if (cf[c]) pc = a;
endcase
```

The following is an optional new subsection to be added at the end of Section 8.4, if pattern-matching in if-statements is adopted:

Section 8.4.2 Pattern matching in if statements

match_expression ::= expression **matches** expression	*from Annex A.6.6*

The predicate of an if statement can use pattern matching using the form e_1 **matches** *pattern*. The matching rules are exactly as described in Section 8.4.1. If the pattern matches, the "then" arm of the if statement is executed, and the identifiers bound during the pattern-match may be used in the "then" arm. If the pattern fails, the "else" arm is executed.

The identifiers in the pattern are implicitly declared to have the type of the corresponding members (this is statically determinable from the pattern and the tagged union declaration), and their scope is the "then arm" of the conditional statement.

Example:

```
if (e matches (tagged J (tagged JC {cc:c,addr:a})))
     ...     // c and a can be used here
else
     ...
```

B Additions to LRM BNF

A.2.2.1 Net and variable types

data_type ::=
  ...
| **taggedunion packed** [signing] { { struct_union_member } } { packed_dimension }
| **taggedunion** [signing] { { struct_union_member } }

A.8.3 Expressions

expression ::=
  ...
| **tagged** identifier expression
| **tagged** identifier ()
| **tagged** identifier

No BNF extension is necessary for tagged union patterns in case statements, since the LHS of `case_item` is already an `expression`, and patterns are just a subset of expressions.

B.1 Optional extension: tagged union patterns in if statements

If we add the option of pattern matching in **if**-statements:

A.6.6 Conditional statements

conditional_statement ::=
  [unique_priority] **if** (match_expression) statement_or_null
  [**else** statement_or_null]
| ...

match_expression ::= expression **matches** expression

where the expression after the new **matches** keyword is restricted to be a tagged union pattern.

C Additions to LRM Annex B Keywords

Add the keyword "`tagged`".

For pattern-matching in if-statements, add the keyword "`matches`".