"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

September 9, 2003

Abstract

System Verilog has **struct** and **union** types. We propose related extensions: **taggedunion** types, together with pattern matching. These yield the following benefits: (1) complete type-safety (which unions do not have), (2) greater brevity, (3) a more "visual" (*i.e.,* readable) way of programming with structures, (4) tagged union expressions, 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 have been used in some languages for many years.

Contents

The actual extension proposed herein (part of the LRM) is syntactically quite small and 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 ($A + B$) 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 fields are only meaningful in certain contexts. For example, the destination register field is only meaningful in an Add instruction. A condition-code register field is only meaningful in a conditional Jump instruction.

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

Note that the integer field is only meaningful 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 fields can only be examined/assigned when they are meaningful, but 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.

Fields 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 in only 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.
- 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 only be used 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` field acts as a "tag" which indicates how to intepret the remaining bits, *i.e.,* as add operands or as jump operands. Similarly, the `jop` field acts as another tag which indicates whether to interpret the remaining fields 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" fields:

```
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" fields:

```
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 fields are only used meaningfully.

Note: very occasionally, this kind of type loophole, or "type laundering", is exactly what the programmer 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 "**taggedunion**" 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. A tagged union type contains one or more *tags*, each associated with a type (the associated summand type).

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

```
typedef taggedunion {
    struct {
        bit [4:0] reg1;
        bit [4:0] reg2;
        bit [4:0] regd;
    } A;
    taggedunion {
        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 declared as *tags* for the union. The tags can be viewed as an implicit enumeration. The representation of a tagged union value implicitly contains enough additional bits to contain the tags (in this example, 1 bit). Also, in this example, the tag can be seen as identical to the opcode (A or J). Similarly, `JU` and `JC` are declared as tags for the nested union. The `JU` address field is not named (it could be named if desired).

Note that we have to invent far fewer names for intermediate unions and structs.

4.2 Tagged union expressions

We extend the syntax of expressions to include tagged union expressions. (Whereas structs distinguish between "structure literals" and "structure expressions", we make no such distinction.)

Tagged union expressions are expressions that evaluate to tagged union values, and can be used in any expression context. A tagged union expression consists of a tag followed by the corresponding component(s).

taggedunion_expression ::=
    identifier { expression ; ... ; expression }
  | identifier { field = expression ; ... ; field = expression }

field ::= identifier

The latter form is used if the tagged union member is a struct and the fields are defined by name (instead of by position).

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

```
A { e1; e2; ed }                          // by position
A { reg2 = e2; regd = ed; reg1 = e1 }     // by name
J {JU {eOffset}}
J {JC { eC; eAddr }}                      // inner expr by position
J {JC { cc = eC; addr = eAddr }}          // inner expr by name
```

4.3 Pattern Matching

We extend the syntax of case statements to include tagged union patterns before the "`:`" in each case item. Here is our example, again:

```
Instr instr;

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

Here, the pattern "`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 fields `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.

Similarly, the pattern "`J(j)`" only matches 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 between `JU` and `JC`.

Patterns can be nested, so the above example can also be written very succinctly as follows:

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

(Note: although the tag J now appears in two patterns, this does not imply any ineffiency in testing. It is well understood how to implement this so that the test for A *vs.* J is performed no more than once, *i.e.,* how to convert this into a deterministic decision tree).

Observe the substantial increase in brevity, and the more "visual" access to the fields of the structures due to pattern matching. In particular, tagged union expressions and tagged union patterns look the same, *i.e.,* a tag followed by the component. 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 fields is not relevant):

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

Further, when done by name, it is ok to omit fields 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 do not have to be exhaustive. The usual **default** mechanism can be used as a final catch-all if all patterns fail.

Note: it is well understood in the literature how to compile this kind of pattern matching to avoid repeated tests, to exploit 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 J{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 variables `c` and `a` can be used in the then-arm, and will be bound to the values of the corresponding fields.

If this example were written using ordinary struct-and-union notation, and without the implicit declaration of pattern variables, it would look like this:

```
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 only meaningful if the left operand is true). Compare this to the brevity of the pattern-matching version.

4.4 Type-safety and verification

All these examples are also completely type-safe because the compiler ensures that the `reg1`, `reg2` and `regd` fields can only be accessed in the A case, and the `cc` and `addr` fields can only be accessed in the J/JC case. It is syntactically impossible, for example, to access the `reg1` field 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 fields are only accessed/updated meaningfully w.r.t. 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 now known and controllable.

4.5 Dot notation to select and assign fields type-safely

"Dot notation" can still be used to access fields of tagged unions:

```
r2 = instr.reg2
```

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

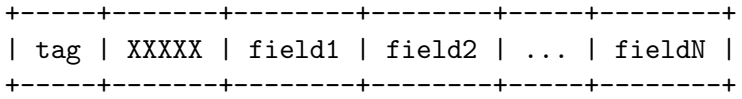
```
case (instr)
    A{reg2=x} : r2 = x;
    default   : $error (....);
endcase
```

Similarly, it is possible to use dot notation to assign a field, again with complete type-safety.

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 sometimes useful in real hardware, are discussed in § 4.8). The canonical representation of a tagged union value is:

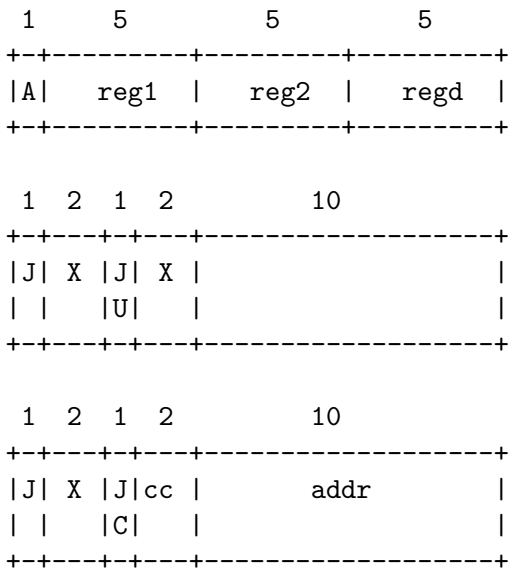


where the `fieldJ`'s are appropriate for the `tag`.

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 tag uses as many bits as necessary to discriminate amongst all the tags, and is always left-justified (*i.e.,* towards the most-significant bits).
- No matter what the current tag value, the size of a tagged union value is always the size of the tag plus the size of the largest summand.
- For smaller summands, the summand bits are right-justified (i.e., towards the least-significant bits), and the bits between the tag and the summand 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 (note: tags are always at known bit positions).

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. Both the struct-and-union version of the `Instr` structure and the tagged union version are represented in 16 bits: 15 bits for the largest summand (A) plus 1 bit for the tag.

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 fields 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 programmer 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 programmer can customize the representations by defining two methods **pack** and **unpack** associated with the tagged union type. For example,

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

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

With this mechanism, the programmer 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 trick in implementing garbage-collected languages.) The tagged union definition for this might be:

```
typedef taggedunion {
    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, using **pack** and **unpack** explicitly to define a non-canonical representation, we can squeeze the representation into 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, one simply provides definitions for **pack** and **unpack**. Except for this, the tagged unions are used as usual in expressions and pattern matching. 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 taggedunion {
    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:

```
Valid {e}
```

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

```
Invalid {}
```

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

```
case (v)
    Invalid {} : $display ("Is invalid");
    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 More on 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 taggedUnion {
    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 taggedunion {
    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). Using a tagged union instead of the corresponding struct may sometimes be preferable because it allows the use of pattern matching and tagged union expressions instead of dot notation.

Thus, in principle, tagged unions could subsume enumerations, structs and unions. However, since enumerations, structs and unions have a long history in C/C++ and are widely familiar amongst 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

The following is a new section to be added after Section 3.11:

Section 3.11+ Tagged unions

```
data_type ::=                                                    from Annex A.2.2.1
    ...
  | taggedunion packed [ signing ] { { struct_union_member } } { packed_dimension }
  | taggedunion [ signing ] { { struct_union_member } }
```

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 the case of tagged unions, each struct_union_member is also known as a *summand*. The identifiers declared by the summands are called *tags*. Conceptually, a tagged union is a like a union, but is one in which we use the tag to remember which summand the value came from. This, in turn, allows us to type-safely extract a value from a tagged union into the correct summand domain.

Example: an integer together with a valid bit:

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

Section 3.11+.1 Canonical representations

As with structs and unions, the **packed**, **signed** and **unsigned** qualifiers may also be used with tagged unions.

When the **packed** qualifier is used, all the summand types must also be packed types, and the canonical representation for a tagged union is the given by following rules.

- The tag uses as many bits as necessary to discriminate amongst all the tags, and is always left-justified (*i.e.,* towards the most-significant bits).
- No matter what the current tag value, the size of a tagged union value is always the size of the tag plus the size of the largest summand.
- For smaller summands, the summand bits are right-justified (i.e., towards the least-significant bits), and the bits between the tag and the summand are undefined. In the extreme case of a summand of `void` type, only the tag is significant and all the remaining bits are undefined.

Section 3.11+.2 Non-canonical (custom) representations

For a tagged union type `T`, a non-canonical representation can be given by defining the following methods in the same scope as `T`'s declaration:

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

These functions can perform arbitrary packing and unpacking to and from bits.

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*
  ...
  | taggedunion_expression

taggedunion_expression ::=
  identifier { expression ; ... ; expression }
  | identifier { field = expression ; ... ; field = expression }

field ::= identifier

A tagged union expression (packed or unpacked) can be built from member expressions using a tag followed by values for the components of the summand indicated by the tag.

Examples:

```
Valid {e}
Invalid {}

A { e1; e2; ed }                          // by position
```

```
A { reg2 = e2; regd = ed; reg1 = e1 }      // by name
J {JU {eOffset}}
J {JC { eC; eAddr }}                       // inner expr by position
J {JC { cc = eC; addr = eAddr }}           // inner expr by name
```

If the summand is a struct, the normal rules apply for using **default**, type keys, *etc.* (described in Section 7.13).

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, the expression being tested can evaluate to a tagged union value. In such a case statement, in each case item, the left-hand side of the colon ":" can contain a *pattern*, which 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
  | tag { pattern ; ... ; pattern }
  | tag { field = pattern ; ... ; field = pattern }

field ::= identifier

The value being tested by the case statement is *matched* against the patterns in the case items, one at a time, in top-to-bottom (textual) order. If the value is a tagged union, its tag must match the pattern's tag, and then the fields of the value must match with the fields in the pattern. An identifier pattern matches any value. a value-pattern (number, string) only matches that value.

If a case-item's pattern matches successfully, that case-item is selected, and the identifiers in the pattern are bound to the corresponding fields in the value. These identifiers can then be used in the right-hand side of the selected case-item. The identifiers in the pattern are implicitly declared to have the type of the corresponding fields (this is statically determinable from the pattern).

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

Example:

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

Example:

```
Instr instr;
```

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

Example:

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

Example:

```
case (instr)
  A{reg2=r2;regd=rd;reg1=r1} : rf[rd] = rf[r1] + rf[r2];
  J{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.

Example:

```
if (e matches J{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 ::=
  ...
| taggedunion_expression

taggedunion_expression ::=
  identifier { expression ; ... ; expression }
| identifier { field = expression ; ... ; field = expression }

field ::= 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 tagged union 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.