5. Data types

The ***e*** language has a number of predefined data types, including the integer and Boolean scalar types common to most programming languages. In addition, new scalar data types (*enumerated types*) that are appropriate for programming, modeling hardware, and interfacing with hardware simulators can be created. The ***e*** language also provides a powerful mechanism for defining OO hierarchical data structures (*structs*) and ordered collections of elements of the same type (*lists*). The following subclauses provide a basic explanation of ***e*** data types.

5.1 *e* data types

Most ***e*** expressions have an explicit data type, as follows:

- Scalar types
- Scalar subtypes
- Enumerated scalar types
- Casting of enumerated types in comparisons
- Struct types
- Struct subtypes
- Referencing fields in when constructs
- List types
- The set type
- The string type
- The real type
- The external_pointer type
- The “untyped” pseudo type

Certain expressions, such as HDL objects, have no explicit data type. See 5.2 for information on how these expressions are handled.

5.1.1 Scalar types

Scalar types in ***e*** are one of the following: numeric, Boolean, or enumerated. Table 17 shows the predefined numeric and Boolean types.

Both signed and unsigned integers can be of any size and, thus, of any range. See 5.1.2 for information on how to specify the size and range of a scalar field or variable explicitly. See also Clause 4.

5.1.2 Scalar subtypes

A *scalar subtype* can be named and created by using a scalar modifier to specify the range or bit width of a scalar type. Unbounded integers are a predefined scalar subtype. The following subclauses describe scalar modifiers, named scalar subtypes, and unbounded integers in more detail.

5.1.2.1 Scalar modifiers

There are two types of scalar modifiers that can be used to modify predefined scalar types:

- Range modifiers
- Width modifiers

Table 17—Predefined scalar types

Type name	Function	Default size for packing	Default value
int	Represents numeric data, both negative and non-negative integers.	32 bits	`0`
uint	Represents unsigned numeric data, non-negative integers only.	32 bits	`0`
bit	An unsigned integer in the range 0–1.	1 bit	`0`
byte	An unsigned integer in the range 0–255.	8 bits	`0`
time	An integer in the range 0–(2^{63}–1).	64 bits	`0`
bool	Represents truth (logical) values, `TRUE` (`1`), and `FALSE` (`0`).	1 bit	`FALSE (0)`
real	Represents double-precision floating-point numbers, identical to the precision of a C type **double**.	64 bits	`0`

Range modifiers define the range of values that are valid. For example, the range modifier in the following expression restricts valid values to those between `0` and `100`, inclusive.

```
int [0..100]
```

Width modifiers define the width in bits or bytes. For example, the width modifiers in the following expressions restrict the bit width to `8`.

```
int (bits:8);
int (bytes:1)
```

Width and range modifiers can also be used in combination, e.g.,

```
int [0..100] (bits: 7)
```

5.1.2.2 Named scalar subtypes

Named scalar subtypes are useful in a context where it is desirable to declare a counter variable, such as the variable `count`, in several places in the program, e.g.,

```
var count : int [0..100] (bits:7);
```

The **type** name can then be used to introduce new variables with this type, e.g.,

```
type int_count : int [0..99] (bits:7);
var count      : int_count
```

See also 5.7.1.

5.1.2.3 Unbounded integers

Unbounded integers represent arbitrarily large positive or negative numbers. Unbounded integers are specified as:

```
int (bits:*)
```

Use an unbounded integer variable when the exact size of the data is unknown. Unbounded integers can be used in expressions just as signed or unsigned integers are, with the following exceptions:

— Fields or variables declared as unbounded integers shall not be generated, packed, or unpacked.
— Unbounded unsigned integers are not allowed, so a declaration of a type such as `uint (bits:*)` shall generate a compile-time error.

5.1.3 Enumerated scalar types

The valid values for a variable or field can be defined as a list of symbolic constants, e.g., the following declaration defines the variable `kind` as having two legal values:

```
var kind : [immediate, register]
```

These symbolic constants have associated unsigned integer values. By default, the first name in the list is assigned the value zero (`0`). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items `+1`. Explicit unsigned integer values can also be assigned to the symbolic constants.

```
var kind : [immediate = 1, register = 2]
```

The associated unsigned integer value of a symbolic constant in an enumerated type can be obtained by using the **as_a()** type casting operator (see 5.8.1). Similarly, an unsigned integer value that is within the range of the values of the symbolic constants can be cast as the corresponding symbolic constant.

Value assignments can also be mixed; some can explicitly be assigned to symbolic constants and others can be automatically assigned. The following declaration assigns the value `3` to `immediate`; the value `4` is automatically assigned to `register`.

```
var kind : [immediate = 3, register]
```

NOTE—Explicitly assigning values to all enumerators aids in avoiding unexpected values.

An enumerated type can be named to facilitate its reuse throughout a program. In the following example, the first statement defines a new enumerated type named `instr_kind`. The variable `i_kind` has the two legal values defined by the `instr_kind` type.

```
type instr_kind : [immediate, register];
var i_kind : instr_kind
```

Enumerated types can also be sized.

```
type instr_kind : [immediate, register] (bits: 2)
```

Variables or fields with an enumerated type can also be restricted to a range. The following variable declaration excludes `foreign` from its legal values:

```
type packet_protocol : [Ethernet, IEEE, foreign];
var p : packet_protocol [Ethernet..IEEE]
```

The default value for an enumerated type is zero (`0`), even if zero (`0`) is not a legal value for that type. For example, the variable `i_kind` has the value zero (`0`) until it is explicitly initialized or generated.

```
type instr_kind : [immediate = 1, register = 2];
var i_kind : instr_kind
```

5.1.4 Casting of enumerated types in comparisons

Enumerated scalar types, like Boolean types, are not automatically converted to or from integers or unsigned integers in comparison operations (i.e., comparisons using the <, <=, >, >=, ==, or **!=** operators). This is consistent with the strong typing in ***e*** and helps avoid the introduction of bugs if the order of symbolic names in an enumerated type declaration is changed. To perform such comparisons, explicit casting or tick notation (') needs to be used to specify the type.

5.1.5 Struct types

Structs are the basis for constructing compound data structures (see also Clause 6). The default value for a struct is `NULL`. A struct type can also be used to define a variable (**var**). For more information on **vars**, see 19.2.

The following statement creates a struct type called `packet` with a field `protocol` of type `packet_protocol`.

```
struct packet {
    protocol : packet_protocol
}
```

The struct type `packet` can then be used in any context where a type is required. For example, in this statement, `packet` defines the type of a field in another struct.

```
struct port {
    data_in : packet
}
```

5.1.6 Struct subtypes

When a struct field has a Boolean type or an enumerated type, a struct subtype can be defined for one or more of the possible values for that field.

Example

The struct `packet` defined as follows has three possible subtypes based on its `protocol` field. The `gen_eth_packet` method generates an instance of the `legal Ethernet packet` subtype, where `legal == TRUE` and `protocol == Ethernet`.

```
type packet_protocol : [Ethernet, IEEE, foreign];

struct packet {
    protocol   : packet_protocol;
    size       : int [0..1k];
    data[size] : list of byte;
    legal      : bool
};

extend sys {
    gen_eth_packet () is {
        var packet : legal Ethernet packet;
        gen packet keeping {it.size < 10};
        print packet
```



```
    }
}
```

To refer to a Boolean struct subtype, in this case, `legal packet`, use this syntax:

```
field_name struct_type
```

To refer to an enumerated struct subtype in a struct where no values are shared between the enumerated types, use this syntax:

```
value_name struct_type
```

In structs where more than one enumerated field can have the same value, use the following syntax instead to refer to the struct subtype:

```
value'field_name struct_type
```

The **extend**, **when**, or **like** constructs can also be used to add fields, methods, or method extensions that are required for a particular subtype. Use the **when** or **extend** construct (see Clause 6) to define struct subtypes with very similar results. These constructs are appropriate for most modeling purposes (see also Annex C).

5.1.7 Referencing fields in when constructs

To refer to a field of a struct subtype outside of a **when**, **like**, or **extend** construct, assign a temporary name to the struct subtype and then use that name. To reference a field in a **when** construct, first specify the appropriate value for the **when** determinant (see Annex C).

5.1.8 List types

List types hold ordered collections of data elements, where each data element conforms to the same type. Items in a list can be indexed with the subscript operator `[]`, by placing a non-negative integer expression in the brackets. List indexes start at zero (`0`). To select an item from a list, specify its index, e.g., `my_list[0]` refers to the first item in the list named `my_list`.

Lists are defined by using the **list of** keyword in a variable or a field definition. The following example defines a list of bytes named lob and explicitly assigns five literal values to it. The print statement displays the first three elements of lob: `15`, `31`, and `63`.

```
var lob : list of byte = {15; 31; 63; 127; 255};
print lob[0..2]
```

The following considerations also apply:

— The default value of a list is an empty list.

— To set a size for lists that have variable sizes, use a **keep** constraint or the **resize()** list pseudo-method.

5.1.9 Keyed lists

A *keyed list data type* is similar to hash tables or association lists found in other programming languages. If the element type of the list is a scalar type or a string type, then the hash key shall be the predefined implicit variable **it**. The only restriction on the type of the list elements is they shall not be lists or sets. However, they can be struct types containing fields that are lists or sets.

See also 20.4.2 and Clause 27.

Syntax example:

```
struct location {
    address : uint;
    data    : uint
};

struct holder {
   !locations : list(key:address) of location
}
```

5.1.10 The set type

The predefined type **set** is used to represent unordered sets of unbounded integer values.

Values of type **set** can be expressed using a *set literal*, which is specified by a *range* construct with numeric value ranges (see 4.4). The actual values in the set shall be evaluated using the unbounded integer semantics, regardless of the actual types of the expressions used inside the set type literal, and regardless of the context. For example, this expression:

```
MAX_UINT in [-5..-1]
```

shall return FALSE, even though -1 would be the result of casting **MAX_UINT** to **int**.

An empty set can be expressed using an empty set type literal: []

The inclusion relation between a numeric value and a set, and the containment relation between two sets, shall be determined by using the **in** operator (see 4.10.5). Operations between sets, such as union, intersect and diff, and queries on sets, such as size, min and max, shall be performed by set pseudo-methods (see 28.4).

Fields or variables of type **set** shall not be generated, packed, or unpacked.

The *canonical form* representation of a value of type **set** shows it as a set literal, as follows:

- The minimum number of intervals is shown, meaning that there are no overlapping or neighboring intervals. For example, the canonical form of both `[1..5, 3..15]` and `[1..5, 6..15]` is `[1..15]`.
- Single value intervals are shown with the single value. For example, the canonical form of `[1..5, 10..10]` is `[1..5,10]`.
- The intervals are shown in ascending order. For example, the canonical form of `[10..15, 1..5]` is `[1..5,10..15]`.

5.1.11 The string type

The predefined type **string** is the same as the `C NULL` terminated (zero-terminated) string type. A series of ASCII characters enclosed by quotes (`" "`) can be assigned to a variable or field of type string, for example:

```
var message : string;
message = "Beginning initialization sequence..."
```


Bits or bit ranges of a string cannot be accessed, but the string can be converted to a list of bytes and that list can be used to access a portion of the string, e.g., the following print statement displays `/test1`:

```
var dir :  string = "/tmp/test1";
var tmp := dir.as_a(list of byte);

tmp = tmp[4..9];
print tmp.as_a(string)
```

The default value of a variable of type **string** is `NULL`.

See also 20.4.4 and Clause 29.

5.1.12 The real type

The **real** type in ***e*** is used to handle and manipulate non-integer numeric values. Real values are physically represented as double-precision floating-point numbers, equivalent to the representation of **double** values in C.

See 5.4.

5.1.13 The external_pointer type

The **external_pointer** type is used to hold a pointer into an external (non-***e***) entity, such as a C struct. Unlike pointers to structs in ***e***, external pointers are not changed during garbage collection.

Syntax example:

```
var c_handle : external_pointer     // holds a foreign pointer
```

5.1.14 The “untyped” pseudo type

This is a type placeholder for untyped values that can be used when runtime values of different types need to be manipulated in a generic way. For example, when objects are manipulated with the reflection API, their types are typically unknown at compile-time; thus, untyped expressions need to be used (see 31.4). Values of any type may be assigned to variables of the “untyped” pseudo type using the **unsafe()** operator (see 5.8.2). Similarly, “untyped” expressions may be used in typed contexts by using **unsafe()**.

NOTE—Untyped variables are left unchanged during garbage collection, which allows struct references to be corrupted.

5.2 Untyped expressions

All ***e*** expressions have an explicit type, except for the following types:

- HDL objects, such as `top.w_en`
- **pack()** expressions, such as `pack(packing.low, 5)`
- bit concatenations, such as `%{slb1, slb2}`

The default type of HDL objects is a 32-bit uint, while **pack()** expressions and bit concatenations have a default type of list of bit. However, due to implicit packing and unpacking, these expressions can be converted to the required data type and bit-size in certain contexts, as follows:

a) When an untyped expression is assigned to a scalar or list of scalars, it is implicitly unpacked and converted to the same type and bit-size as the expression on the LHS. Implicit unpacking is not supported for strings, structs, or lists of non-scalar types.

b) When a scalar or list of scalars is assigned to an untyped expression, it is implicitly packed before it is assigned. Implicit packing is not supported for strings, structs, or lists of non-scalar types.

c) When the untyped expression is the operand of any binary operator (`+`, `-`, `*`, `/`, or `%`), the expression is assumed to be a numeric type. The precision of the operation is determined by the expected type and the type of the operands (see 5.5).

d) When a **pack()** expression includes the parameter or the return value of a method call, the expression takes the type and size as specified in the method declaration. The method parameter or return value in the pack expression shall be a scalar type or a list of scalar type.

e) When an untyped expression appears in one of the following contexts, it is treated as a Boolean expression:

```
if (untyped_exp) then {..}
while (untyped_exp) do {..}
check that (untyped_exp)
not untyped_exp
rise(untyped_exp), fall(untyped_exp), true(untyped_exp)
```

When the type and bit-size cannot be determined from the context, the expression is automatically cast according to the following rules:

— The default type of an HDL signal is an unsigned integer; the default bit-size is 32.
— The default type of a pack expression and a bit concatenation expression is a list of bit.

When expressions are untyped, an implicit pack/unpack is performed according to the expected type. See also 20.5.

5.3 Assignment rules

Assignment rules define what is a legal assignment and how values are assigned to entities. The following subclauses describe various aspects of assignments.

5.3.1 What is an assignment?

There are several legal ways to assign values, as follows:

— Assignment actions
— Return actions
— Parameter passing
— Variable declaration

Here is an example of an assignment action, where a value is explicitly assigned to a variable `x` and to a field `sys.x`.

```
extend sys {
    x : int;
    m() is {
        var x: int;
        sys.x = '~/top/address';
        x = sys.x + 1
```



```
    }
}
```

Here is an example of a **return** action, which implicitly assigns a value to the **result** variable:

```
extend sys {
    n(): int (bits:64) is {
        return 1
    }
}
```

Here is an example of assigning a value (`6`) to a method parameter (`i`):

```
extend sys {
    k(i: int) @sys.any is {
        wait [i] * cycle
    };

    run() is also {
        start k(6)
    }
}
```

Here is an example of how variables are assigned during declaration:

```
extend sys {
    b() is {
        var x : int = 5;
        var y := "ABC"
    }
}
```

Values shall not be assigned to fields during declaration, however.

5.3.2 Assignments create identical references

Assigning one struct, list, or value to another object of the same type results in two references pointing to the same memory location, so that changes to one of the objects also occur in the other object immediately.

Example

```
data1 : list of byte;
data2 : list of byte;

run() is also {
    data2 = data1;
    data1[0] = 0
}
```

After generation, the two lists `data1` and `data2` are different lists. However, after the `data2 = data1` assignment, both lists refer to the same memory location; therefore, changing the `data1[0]` value also changes the `data2[0]` value immediately.

5.3.3 Assignment to different (but compatible) types

This subclause describes the assignment between different, yet compatible, types.

5.3.3.1 Assignment of numeric types

Any numeric type (e.g., **uint**, **int**, or one of their subtypes) can be assigned with any other numeric type. Untyped expressions, such as HDL objects, can also appear in assignments of numeric types (see 5.2).

Automatic casting is performed when a numeric type is assigned to a different numeric type, and automatic extension or truncation is performed if the types have a different bit-size (see 5.6; see also 5.5.)

5.3.3.2 Assignment of Boolean types

A Boolean type can only be assigned to another Boolean type.

```
var x : bool;
x = 'top.a' >= 16
```

5.3.3.3 Assignment of enumerated types

An enumerated type can be assigned with that same type or its scalar subtype. (The scalar subtype differs only in range or bit-size from the base type.) The following example shows:

— An assignment of the same type

```
var x : color = blue
```

— An assignment of a scalar subtype

```
var y : color2 = x
```

To assign any scalar type (numeric, enumerated, or Boolean type) to any different scalar type, use the **as_a()** operator (see 5.8.1).

5.3.3.4 Assignment of structs

An entity of type struct can be assigned with a struct of that same type or with one of its subtypes. The following example shows:

— A same type assignment

```
p2 = p1
```

— An assignment of a subtype (`Ether_8023 packet`)

```
var p : Ether_8023 packet;
set_cell(p)
```

— An assignment of a derived struct (`cell_8023`)

```
set_cell(p:packet) is {
    p.cell = new cell_8023;
    ...
```

Although a subtype can be assigned to its parent struct without any explicit casting, to perform the reverse assignment (assign a parent struct to one of its subtypes), the **as_a()** method needs to be used (see 5.8.1).

5.3.3.5 Assignment of strings

A string can be assigned only with strings, as follows:


```
extend sys {
    m(): string is {
        return "aaa"    // assignment of a string
    }
}
```

5.3.3.6 Assignment of lists

An entity of a type list can be assigned only with a list of the same type. In the following example, the assignment of `list1` to `x` is legal because both lists are lists of integers:

```
extend sys {
    list1 : list of int;
    m() is {
        var x : list of int = list1;
    }
}
```

However, an assignment such as `var y: list of int (bits: 16) = list1;` is not legal, because `list1` is not the same list type as `y`. `y` has a size modifier, so it is a subtype of `list1`.

Use the **as_a()** method to cast between lists and their subtypes (see 5.8.1).

5.3.3.7 Assignment of sets

A set can be assigned only with sets, ase follows:

```
extend sys {
    m(): set is {
        return [1..10]    // assignment of a set
    }
}
```

5.4 Real data type

Objects of type **real** are double-precision floating-point numbers, the same as C type **double**. The representation of real values and the semantics of arithmetic and cast operators uses the double-precision floating-point implementation on the underlying machine, which should be compliant with IEEE Std 754™.

5.4.1 Real data type usage

A **real** object may be used (or is legal) in any context except in the following cases:

- Both operands of the shift operators (<<, >>)
- Bitwise operators (|, &, ^)
- Bitwise routines
- Modulo (%)
- **odd()**
- **even()**

5.4.2 Real literals

Real literals are numbers that have a decimal point or an exponential part or both. If a decimal point exists, there must be digits on both sides of the decimal point. Underscores can be added for readability and are ignored. See Table 18.

Table 18—Examples of real literals

Real constant	Value
5.3876e4	53 876
4e–11	0.00000000004
1e+5	100 000
7.321E–3	0.007321
3.2E+4	32 000
0.5e–6	0.0000005
0.45	0.45
6.e10	60 000 000 000

5.4.3 Real constants

The **real** constants in Table 19 and Table 20 are defined in both ***e*** code and in C code that includes a suitable header file:

Table 19—Mathematical constants

Constant	Value
IEEE_1647_M_E	e
IEEE_1647_M_LOG2E	Logarithm base 2 of e
IEEE_1647_M_LOG10E	Logarithm base 10 of e
IEEE_1647_M_LN2	Natural logarithm of 2

Table 19—Mathematical constants *(continued)*

Constant	Value
IEEE_1647_M_LN10	Natural logarithm of 10
IEEE_1647_M_PI	PI
IEEE_1647_M_TWO_PI	2*PI
IEEE_1647_M_PI_2	PI/2
IEEE_1647_M_PI_4	PI/4
IEEE_1647_M_1_PI	1/PI
IEEE_1647_M_2_PI	2/PI
IEEE_1647_M_2_SQRTPI	2/sqrt(PI)
IEEE_1647_M_SQRT2	sqrt(2)
IEEE_1647_M_SQRT1_2	sqrt(1/2)

Table 20—Physical constants

Constant	Value
IEEE_1647_P_Q	Charge of electron in coulombs
IEEE_1647_P_C	Speed of light in vacuum in meters/second
IEEE_1647_P_K	Boltzmann’s constant in joules/kelvin
IEEE_1647_P_H	Planck’s constant in joules*second
IEEE_1647_P_EPS0	Permittivity of vacuum in farads/meter
IEEE_1647_P_U0	Permeability of vacuum in henrys/meter
IEEE_1647_P_CELSIUS0	Zero Celsius in kelvin

NOTE 1—All mathematical constants are prefixed by IEEE_1647_M_.

NOTE 2—All physical constants are prefixed by IEEE_1647_P_.

5.4.4 Real type limitations

— The key of a keyed list shall not be of type **real**.

5.5 Precision rules for numeric operations

For precision rules, there are two types of numeric expressions in ***e***, as follows:

— *context-independent* expressions, where the precision of the operation (bit width) and numeric type (signed or unsigned) depend only on the types of the operands

— *context-dependent* expressions, where the precision of the operation and the numeric type depend on the precision and numeric type of other expressions involved in the operation (the *context*), as well as the types of the operands

A numeric operation in ***e*** is performed in one of three possible combinations of precision and numeric type:

a) Unsigned 32-bit integer (**uint**)

b) Signed 32-bit integer (**int**)

c) Infinite signed integer (**int (bits: *)**

The ***e*** language has rules for determining the context of an expression or deciding the precision, and performing data conversion and sign extension.

5.5.1 Determining the context of an expression

The rules for defining the context of an expression are applied in the following order:

a) In an assignment (*lhs* = *rhs*), the right-hand side (*rhs*) expression inherits the context of the left-hand side (*lhs*) expression.

b) A sub-expression inherits the context of its enclosing expression.

c) In a binary-operator expression (*lho OP rho*), the right-hand operand (*rho*) inherits context from the left-hand operand (*lho*), as well as from the enclosing expression.

Table 21 summarizes context inheritance for each type of operator that can be used in numeric expressions.

Table 21—Summary of context inheritance in numeric operations

Operator	Function	Context
*** / % + –** **< <= > >=** **== !=** **=== !==** **& | ^**	Arithmetic, comparison, equality, and bit-wise Boolean	The right-hand operand *(rho)* inherits context from the left-hand operand (*lho*), as well as from the enclosing expression. *lho* inherits only from the enclosing expression.
~ ! unary **+ –**	Bit-wise not, Boolean not, unary plus, minus	The operand inherits context from the enclosing expression.
[ ]	List indexing	The list index is context-independent.
[..]	List slicing	The indices of the slice are context-independent.
[:]	Bit slicing	The indices of the slice are context-independent.
f**(...)**	Method or routine call	The context of a parameter to a method is the type and bit width of the formal parameter.
{...; ...}	List concatenation	Context is passed from the *lhs* of the assignment, but not from left-to-right between the list members.

Table 21—Summary of context inheritance in numeric operations *(continued)*

Operator	Function	Context
%{..., ...}	Bit concatenation	The elements of the concatenation are context-independent.
>>, <<	Shift	Context is passed from the enclosing expression to the left operand. The context of the right operand is always a 32-bit **uint**.
in	Inclusion and containment operator	Both operands are context-independent.
[*i*..*j*, ...]	Set literal	The elements are context-independent.
&&, ||	Boolean	All operands are context-independent.
a ? *b* : *c*	Conditional operator	*a* is context-independent, *b* inherits the context from the enclosing expression, *c* inherits context from *b*, as well as from the enclosing expression.
as_a()	Casting	The operand is context-independent.
abs(), **odd()**, **even()**	Arithmetic routine	The parameter is context-independent.
min(), **max()**	Arithmetic routine	The right parameter inherits context from the left parameter (*lp*), as well as from the enclosing expression. *lp* inherits only from the enclosing expression.
ilog2(), **ilog10()**, **isqrt()**	Arithmetic routine	The context of the parameter is always a 32-bit **uint**.
ipow()	Arithmetic routine	Both parameters inherit the context of the enclosing expression, but the right parameter does not inherit context from the left.

5.5.2 Deciding precision and performing data conversion and sign extension

The rules for deciding precision, and performing data conversion and sign extension are as follows:

Determine the context of the expression, which can be comprised of a maximum of two types:

a) If all types involved in an expression and its context are 32 bits in width or less:

   1) The operation is performed in 32 bits.

   2) If any of the types is unsigned, the operation is performed with unsigned integers.

      Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers, unless preceded by a hyphen (-).

   3) Each operand is automatically cast, if necessary, to the required type.

      Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.

b) If any of the types is greater than 32 bits:

   1) The operation is performed in infinite precision [**int (bits:*)**].

   2) Each operand is zero-extended (if it is unsigned) or sign-extended (if it is signed) to infinite precision.

5.6 Automatic type casting

During assignment of a type to a different but compatible type, automatic type casting is performed in the following contexts:

— Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types. For example:

```
var x : uint;
var y : int;
x = y
```

— Untyped expressions are automatically cast on assignment. See 5.2 for more information.

— Sized scalars are automatically type cast to differently sized scalars of the same type.

— Struct subtypes are automatically cast to their base struct type.

There are three important ramifications to automatic type casting.

a) If the two types differ in bit-size, the assigned value is extended or truncated to the required bit-size.

b) Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.

c) There is no automatic casting to a reference parameter (see 18.3).

5.6.1 Conversion between real and integer data types

Automatic casting is performed between the **real** type and the other numeric types.

Converting a **real** type object to an integer type object uses the following process:

a) The object is first converted to type **int** (bits:*) with the value of the largest integer whose absolute value is less than or equal to the absolute value of the **real** object.

b) The object is then converted to the expected integer type.

Additional rules apply to converting **real** objects to integer objects:

— If the object's floating-point value is infinity (inf), negative infinity (-inf), or Not-a-Number (NaN), an error will be emitted when trying to convert to an integer value.

— When converting an integer object to the **real** type, the object is converted to the value closest to the integer value that can be represented in the double-precision format.

When converting from an integer data type to a **real**, the integer value is simply converted to its identical value represented as a real.

Automatic casting of reals to integers or integers to reals is not performed in the context of constraints. Explicit casting is required within constraints that involve both integer and real expressions so that all resulting terms are of the same kind.

5.6.2 Real data type precision, data conversion, and sign extension

The rules for deciding precision, performing data conversion, and sign extension are as follows:

a) Determine the context of the expression. The context may be comprised of up to three types.

b) If all types involved in an expression, and its context is integer values of 32 bits in width or less:

1) The operation is performed in 32 bits.

2) If any of the types are unsigned, the operation is performed with unsigned integers.

NOTE—Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers unless preceded by a hyphen.

3) Each operand is automatically cast, if necessary, to the required type.

NOTE—Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.

c) If all types are integer types, and any of the types is greater than 32 bits:

1) The operation is performed in infinite precision [int(bits:*)].

2) Each operand is zero-extended if it is unsigned, or sign-extended if it is signed, to infinite precision.

d) If any of the types is a **real** type, then the operation is done in double precision, and all objects should first be converted according to the rules described above.

5.7 Defining and extending scalar types

The following constructs can be used to define and extend scalar types.

5.7.1 type enumerated scalar

Purpose	Define an enumerated scalar type	
Category	Statement	
Syntax	**type** *enum-type-name*: **[**[*name*[**=***exp*], ...]**]** [**(bits** | **bytes:** *width-exp***)**]	
Parameters	*enum-type-name*	A legal ***e*** name. The name shall be unique in the global type-name space.
	name	A legal ***e*** name. Each name shall be unique within the same type.
	exp	A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (`0`). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items `+1`.
	width-exp	A positive constant expression. The valid range of values for sized enumerated scalar types is limited to the range `1` to 2^{n-1}, where *n* is the number of bits.

This defines an enumerated scalar type consisting of a set of names or name-value pairs. If no values are specified, the names get corresponding numerical values starting with `0` for the first name, and casting can be done between the names and the numerical values.

Syntax example:

```
type PacketType : [rx=1, tx, ctrl]
```

5.7.2 type scalar subtype

Purpose	Define a scalar subtype	
Category	Statement	
Syntax	**type** *scalar-subtype-name*: *scalar-type* [*range*, ...]	
Parameters	*scalar-subtype-name*	A unique ***e*** name. The name shall be unique in the global type-name space.
	scalar-type	Any previously defined enumerated scalar type, any of the predefined scalar types, including **int**, **uint**, **bool**, **bit**, **byte**, or **time**, or any previously defined scalar subtype.
	range	A constant expression or two constant expressions separated by two dots (`..`). All constant expressions shall resolve to legal values of the named type.

This defines a subtype of a scalar type by restricting the legal values that can be generated for this subtype to the specified range. The default value for variables or fields of this type "size" is zero (`0`), which is the default for all integers. The *range* only affects any generated values.

Syntax example:

```
type size : int [8, 16]
```

5.7.3 type sized scalar

Purpose	Define a sized scalar	
Category	Statement	
Syntax	**type** *sized-scalar-name*: *type* **(bits** | **bytes:** *exp***)**	
Parameters	*sized-scalar-name*	A unique ***e*** name. The name shall be unique in the global type-name space.
	type	Any previously defined enumerated type or any of the predefined scalar types, including **int**, **uint**, **bool**, or **time**.
	exp	A positive constant expression. The valid range of values for sized scalars is limited to the range `1` to 2^{n-1}, where *n* is the number of bits.

This defines a scalar type with a specified bit width. The actual bit width is *exp* * 1 for bits and *exp* * 8 for bytes.

When assigning any expression into a sized scalar variable or field, the expression's value is truncated or extended automatically to fit into the variable. An expression with more bits than the variable is chopped down to the size of the variable. An expression with fewer bits is extended to the length of the variable. The added upper bits are filled with zeros (`0`) if the expression is unsigned or with the appropriate sign bit (`0` or `1`) if the expression is signed.

Syntax example:

```
type word    : uint( bits:16);
type address : uint(bytes: 2)
```

5.7.4 extend type

Purpose	Extend an enumerated scalar type	
Category	Statement	
Syntax	**extend** *enum-type*: [*name*[= *exp*], ...]	
Parameters	*enum-type*	Any previously defined enumerated type.
	name	A legal ***e*** name. Each name shall be unique within the type.
	exp	A unique 32-bit constant expression. Names or name-value pairs can appear in any order. By default, the first name in the list is assigned the integer value zero (`0`). Subsequent names are assigned values based upon the maximum value of the previously defined enumerated items `+1`.

This extends the specified enumerated scalar type to include the specified names or name-value pairs.

Syntax example:

```
type   PacketType : [rx, tx, ctrl];
extend PacketType : [status]
```

5.8 Type-related constructs

The **as_a()** expression is used to convert an expression from one data type to another. The **unsafe()** expression casts the expression to the type that is required by the context. The **all_values()** pseudo-routine returns a list of all of the legal values of a specified scalar type. Information about how different types are converted, such as strings to scalars or lists of scalars, is contained in Table 22 and Table 23.

5.8.1 as_a()

Purpose	Casting operator	
Category	Expression	
Syntax	*exp*.**as_a(***type*: type name**)**: type	
Parameters	*exp*	Any ***e*** expression.
	type	Any legal ***e*** type.

This returns the expression converted into the specified type. Although some casting is done automatically (see 5.6), explicit casting is required to make assignments between different but compatible types.

Following are assignment compatible types requiring explicit casting:

— Scalars and lists of scalars
— Strings and scalars or lists of scalars
— Structs and list of structs
— Simple lists and keyed lists

Syntax example:

```
print (b).as_a(uint)
```

5.8.2 unsafe()

Purpose	Force casting
Category	Expression
Syntax	*exp*.**unsafe()**
Parameters	*exp* Any ***e*** expression.

This casts the expression to the type that is required by the context, regardless of any static or dynamic type rules. This operator may be used only in contexts where the required type is explicit, such as assignment and parameter passing to methods.

Syntax example:

```
var value : int = param.unsafe()
```

5.8.2.1 Type conversion between scalars and lists of scalars

Numeric expressions (unsigned and signed integers) of any size are automatically type cast upon assignment to different numeric types.

For other scalars and lists of scalars, there are a number of ways to perform type conversion, including the **as_a()** method, the **pack()** method, the **%{}** bit concatenation operator, and various string routines. Table 22 shows how to convert between scalars and lists of scalars.

In Table 22, **int** represents **int**/**uint** of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. **int(bits:***x***)** means *x* as any constant; variables shall not be used as the integer width.

The solutions presume variables are declared as follows:

```
var int          : int;
var bool         : bool;
var enum         : enum;
var list_of_bit  : list of bit;
var list_of_byte : list of byte;
var list_of_int  : list of int
```

Any conversions not explicitly shown might have to be accomplished in two stages.

Table 22—Type conversion between scalars and lists of scalars

From	To	Solutions
int	*list of bit*	list_of_bit = int**[..]**
int	*list of int(bits:x)*	list_of_int = **%{**int**}** list_of_int = **pack(packing.low**, int**)** (LSB of int goes to list[0] for either choice)
list of bit *list of byte*	*int*	int = list_of_bit**[:]**
list of int(bits:x)	*int*	int = **pack(packing.low**, list_of_int**)** (use **packing.high** for list in the other order)
int(bits:x)	*int(bits:y)*	intx = inty (truncation or extension is automatic) intx.**as_a(**int(bits:y)**)**
bool	*int*	int = bool.**as_a(**int**)** (`TRUE` becomes `1`, `FALSE` becomes `0`)
int	*bool*	bool = int.**as_a(**bool**)** (`0` becomes `FALSE`, non-`0` becomes `TRUE`)
int	*enum*	*enum* = int.**as_a(***enum***)** (no checking is performed to make sure the int value is valid for the range of the enum)
enum	*int*	int = *enum*.**as_a(**int**)** (truncation is automatic)
enum	*bool*	*enum*.**as_a(**bool**)** [enumerated types with an associated unsigned integer value of `0` become `FALSE`; those with an associated non-`0` values become `TRUE` (see 5.1.3)]
bool	*enum*	bool.**as_a(***enum***)** (Boolean types with a value of `FALSE` are converted to the enumerated type value that is associated with the unsigned integer value of `0`; those with a value of `TRUE` are converted to the enumerated type value that is associated with the unsigned integer value of `1`; no checking is performed to make sure the Boolean value is valid for the range of the enum)
enum	*enum*	*enum1* = *enum2*.**as_a(***enum1***)** (no checking is performed to make sure the int value is valid for the range of the enum)
list of int(bits:x)	*list of int(bits:y)*	listx.**as_a(**list of int(bits:y)**)** (the same number of items, each padded or truncated) listy = **pack(packing.low**, listx**)** (concatenated data, different number of items)

5.8.2.2 Type conversion between strings and scalars or lists of scalars

There are a number of ways to perform type conversion between strings and scalars or lists of scalars, including the **as_a()** method, the **pack()** method, the **%{}** bit concatenation operator, and various string routines. Table 23 shows how to convert between strings and scalars or lists of scalars.

Table 23—Type conversion between strings and scalars or lists of scalars

From	To	ASCII convert?	Solutions
list of int *list of byte*	*string*	Yes	list_of_int.**as_a(**string**)** (each list item is converted to its ASCII character and the characters are concatenated into a single string; `int[0]` represents left-most character; if a list item is not a printable ASCII character, the string returned is empty)
string	*list of int* *list of byte*	Yes	string.**as_a(**list of int**)** (each character in the string is converted to its numeric value and assigned to a separate element in the list; the left-most character becomes `int[0]`)
string	*list of int*	Yes	list_of_int = **pack(packing.low**, string**)** list_of_int = **%{**string**}** (the numeric values of the characters are concatenated before assigning them to the list; any pack option gives same result; the null byte, `00`, is the last item in the list)
string	*int*	Yes	int = **%{**string**}** int = **pack(packing.low**, string**)** (any pack option gives the same result)
int	*string*	Yes	**unpack(packing.low**, %{8'b0, int}, string**)** (any pack option with `scalar_reorder={}` gives the same result)
string	*int*	No	string.**as_a(**int**)** (converts to decimal) append("0b", string).**as_a(**int**)** (converts to binary) append("0x", string).**as_a(**int**)** (converts to hexadecimal)
int	*string*	No	int**.as_a(**string**)** (uses the current print radix) **append(**int**)** (converts int according to the current print radix) **dec(**int**)**, **hex(**int**)**, **bin(**int**)** (converts int according to a specific radix)
string	*bool*	No	bool = string**.as_a(**bool**)** (only `TRUE` and `FALSE` can be converted to Boolean; all other strings return an error)
bool	*string*	No	string = bool**.as_a(**string**)**
string	*enum*	No	*enum* = string.**as_a(***enum***)**
enum	*string*	No	string = *enum*.**as_a(**string**)**

In Table 23, **int** represents **int**/**uint** of any size, including bit, byte, and any user-created size. If a solution is specific to bit or byte, then bit or byte is explicitly stated. **int(bits:***x***)** means *x* as any constant; variables shall not be used as the integer width.

The solutions presume variables are declared as follows:

```
var int          : int;
var list_of_byte : list of byte;
```

```
var list_of_int  : list of int;
var bool         : bool;
var enum         : enum;
var string       : string
```

Any conversions not explicitly shown might have to be accomplished in two stages.

5.8.2.3 Type conversion between structs, struct subtypes, and lists of structs

Struct subtypes are automatically cast to their base struct type, so for example, a variable of type Ethernet packet can be assigned to a variable of type packet without using **as_a()**. **as_a()** can be used to cast a base struct type to one of its subtypes; if a mismatch occurs, then NULL is assigned. For example, the **print** pkt.**as_a(**foreign packet**)** action results in `pkt.as_a(foreign packet) = NULL` if `pkt` is not a foreign packet.

When the expression to be converted is a list of structs, **as_a()** returns a new list of items whose type matches the specified type parameter. If no items match the type parameter, an empty list is returned. The list can contain items of various subtypes, but all items shall have a common parent type, i.e., the specified type parameter shall be a subtype of the type of the list.

Assigning a struct subtype to a base struct type does not change the declared type. Thus, **as_a()** needs to be used to cast the base struct type as the subtype and access any of the subtype-specific struct members.

Subtypes created through **like** inheritance exhibit the same behavior as subtypes created through **when** inheritance.

5.8.2.4 Type conversion between simple lists and keyed lists

Simple lists can be converted to keyed lists and vice versa. The hash key is dropped in converting a keyed list to a simple list. However, a key needs to be specified first to convert a simple list to a keyed list.

Example

To convert a simple list of packets `sys.packets` to a keyed list, where the `len` field of the packet struct is the key:

```
var pkts : list (key:len) of packet;
pkts = sys.packets.as_a(list (key:len) of packet)
```

Using the **as_a()** method returns a copy of `sys.packets`, so the original `sys.packets` is still a simple list, not a keyed list. Thus, `print pkts.key_index(130)` returns the index of the item that has a `len` field of `130`, while `print sys.packets.key_index(130)` shall return an error.

If a conversion between a simple list and a keyed list also involves a conversion of the type of each item, that conversion of each item follows the standard rules, e.g., when **as_a()** is used to convert an integer to a string, no ASCII conversion is performed. Similarly, if **as_a()** is used to convert a simple list of integers to a keyed list of strings, no ASCII conversion is performed.

No checking is performed to ensure the value is valid when casting from a numeric or Boolean type to an enumerated type, or when casting between enumerated types.

— The **as_a()** pseudo-method, when applied to a scalar list, creates a new list whose size is the same as the original size and then casts each element separately.

— When the **as_a()** operator is applied to a list of structs, the list items for which the casting failed are omitted from the list.

— **as_a()** can be used to convert a string to an enumerated type. The string has to exactly match one of the possible values of that type, using a case-sensitive string comparison, or a runtime error shall be issued.

See also 4.16.1.

5.8.2.5 Type conversion between reals and non-numeric scalars

Converting a non-numeric scalar type object to a **real** type object using the **as_a()** operator uses the following process:

a) The scalar type object is first converted to an integer value.

b) The object is then converted to a **real** value according to process and rules listed in 5.5.

Additional rules apply to converting non-numeric scalar objects to **real** objects using the **as_a()** operator:

— When converting a string value to **real** using the **as_a()** operator, the string is parsed as if it was a **real** literal, and the value of the **real** literal is returned.

— If the string does not conform to the definition of a **real** literal, an error is emitted.

5.8.2.6 Type conversion between numeric lists and sets

Numeric lists (including keyed lists) can be converted to sets and vice versa using the **as_a()** operator.

When a numeric list is converted to a set, **as_a()** returns a set that contains the numeric values of all and only items of the list. The order of items in the list and the number of their appearances are disregarded. An empty list is converted to an empty set.

For example, `{1;5;3;2;1}.as_a(set)` returns `[1..3,5]`.

Converting a set to a numeric list uses the following process:

a) All the numeric values are retrieved from the set from the lower bound to upper bound, i.e., in the increasing order.

a) Each numeric value is then automatically cast to the type of list elements, according to rules listed in 5.6

An empty set is converted to an empty list.

For example; `[-1..1].as_a(list of int)` returns `{-1;0;1}`, and `[-1..1].as_a(list of uint)` returns `{MAX_UINT;0;1}`.

5.8.3 all_values()

Purpose	Access all values of a scalar type
Category	Pseudo-routine
Syntax	**all_values(***scalar-type*: type name**)**: list of scalar type
Parameters	*scalar-type* Any legal ***e*** scalar type.

This returns a list that contains all the legal values of the specified scalar type. The order of the items is from the smallest to the largest. When the type is an enumerated type, this order is determined by the numeric values of the items.

Syntax example:

```
print all_values(reg_address)
```

5.8.4 set_of_values()

Purpose	Access the set of all legal values of a numeric type
Category	Pseudo-routine
Syntax	**set_of_values(***numeric-type*: type name**)**: set
Parameters	*numeric-type* Any legal ***e*** numeric type, except unbounded integer types that have no range restriction.

This returns a set of all the legal values of the specified numeric type. For example, `set_of_values(uint(bits:4))` returns the set `[0..15]`, and `set_of_values(int[1..10])` returns the set `[1..10]`.

Syntax example:

```
print set_of_values(reg_address)
```

5.8.5 full_set_of_values()

Purpose	Access the set of all possible values of a numeric type
Category	Pseudo-routine
Syntax	**full_set_of_values(***numeric-type*: type name**)**: set
Parameters	*numeric-type* Any legal ***e*** numeric type, except unbounded integer types that have no range restriction.

This returns a set of all possible values of the specified numeric type. It is determined by the bit width and signedness of the type only, and any range restriction specified by a range modifier is disregarded. The resulting set may include values that are not legal values of the type, provided that they are possible values.

For example, `set_of_values(uint(bits:4)[1..10])` returns `[1..10]`, but `full_set_of_values(uint(bits:4)[1..10])` returns `[0..15]`.

For types that have no range restrictions, the result of **full_set_of_values()** is equivalent to the result of **set_of_values()**.

Syntax example:

```
print full_set_of_values(reg_address)
```