# CHAPTER 1 CSL Language common features



**TABLE 1.1** Chapter Outline

## 1.1 CSL Language Commands Summary

**TABLE 1.2** CSL features presented in this chapter

| |
|---|
| 1.1.1 Directives |
| 1.1.2 CSL Enumerated type |
| 1.1.3 CSL Bitrange object |
| 1.1.4 CSL Field |
| 1.1.5 CSL Parameter |
| 1.1.6 Prefixes and suffixes |
| 1.1.7 Operators |
| 1.1.8 Numbers |

### *1.1.1 Directives*

Directives are used to communicate the CSL compiler various tasks, for example file inclusion.

#### *1.1.1.1 CSL include directive*

The CSL include directive allows the user to include code written in the C++ or RTL language in the generated CSL files.
Currently two languages are supported:

- Verilog HDL - used mostly for describing the logic of a design/unit or reusing code from previous designs written in Verilog.
- C++ programming language - used to generate the C++ user code for the C++ Simulator (Csim).
- Future languages include: System Verilog and System C.

The syntax rules for the include directive is straightforward and can be seen in the example below:

```
csl_include(language_type, "file_name");
language_type ::= file_cplusplus | file_verilog
```

**Bold** text represents language reserved syntax, *italics* are user defined variables .
In the include directive above, language type specifies the language the included file is written in and it's possible values are given below:

#### *1.1.1.1.1 CSL Include usage and rules*

Include directive can be called anywhere in the global scope or inside certain CSL classes (Table 1.3). In the generated code, the directive will be replaced with the code found in the specified file name. Note that so far, CSLC does not check the code in included files.

**TABLE 1.3** CSL include directive used in other CSL classes

| CSL class | accepts calling include directive |
|---|---|
| CSL Unit | YES |
| CSL Testbench | YES |
| CSL Vector | YES |
| CSL State Data | YES |
| CSL Register | YES |
| CSL Register File | YES |
| CSL Fifo | YES |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field | - |

Below is an example of the CSL include directive and the generated code. The included language type is Verilog in this case:

**TABLE 1.4** CSL Include directive

**CSL code**

```
//Input CSL code
csl_unit dff {
  csl_port d  (input)      ,
           clk(input)      ,
           q  (output, reg);
  csl_include(file_verilog,"vlogic");
  dff(){}
};
```

```
//File to be included: vlogic
always @ (posedge clk)
  q <= d;
```

**CSLC Generated Verilog code**

```
module dff(d,
           q);
  input d;
  input clk;
  output reg q;

  //Begin included file: vlogic
  always @ (posedge clk)
    q <= d;
  //End included file: vlogic
endmodule
```

### *1.1.2 CSL Enumerated type*

Another language construct similar to C++ is the CSL enumerated type. Just like in C++, CSL enum is a collection of named integer constants. The syntax for declaring a CSL enum is given below:

```
csl_enum enum_name {
  (enum item)+
};
```

**Bold** text represents language reserved syntax, *italics* are user defined variables.

Enum items are the named integers that form the enumerated type.

```
enum item ::= enum_item_name ( = enum_item_value ) ;
```

Enum items can be optionally assigned a numeric value. If this value is not explicitly set, it will default to the value of the previous enum item incremented by 1. If the first enum item has not a value explicitly set, then the default value for it is zero.

#### *1.1.2.1 CSL Enumerated type usage and rules*

Enumerated types can only be declared the global scope: Table 1.5

**TABLE 1.5** CSL const int declaration in other CSL classes

| CSL class | Accepts enum declaration |
|---|---|
| global scope | YES |
| CSL Unit | - |
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |
| CSL Register | - |
| CSL Register File | - |
| CSL Fifo | - |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field | - |

The width of the signal can be set by assigning a field associated with an enum. The width is inferred from the enum width. The following example code shows this case:

```
csl_enum e {
a = 0
};

csl_unit top {
csl_field f(2,e);
csl_signal s;
top() {
  s.set_bitrange(f);
 }
};
```

### *1.1.2.2 CSL Enumerated commands*

```
   csl_enum enum_name {enum_item
[=item_index](,enum_item[=item_index])*};
```

### *1.1.3 CSL Bitrange object*

A bitrange object contains a range with an upper to lower index or lower to upper index. Bitranges can be set or retrieved with appropriate methods detailed later. There are two types of bitrange objects: simple and multi dimensional. The declaration of a simple bitrange is given below:

```
csl_bitrange bitrange_name(constructor_parameters);
```

**Bold** text represents language reserved syntax, *italics* are user defined variables.

The constructor parameters can be a single numeric expression for width, two numeric expressions for specifically setting the lower and upper index, or an identifier that can be a numeric value (e.g. const int) or the name of another bitrange object (copy constructor - the properties of the copied object are replicated under a new object with the specified name):

```
constructor_parameters ::= numeric_expression
                          |upper_index , lower_index
                          |bitrange_object_name
```

A short example is provided below:

```
csl_bitrange br1(4);      //width=4, bit range = [3:0]
csl_bitrange br2(1,0);    //width=2, bit range =[1:0]
csl_bitrange br3(br1);    //copy constructor
csl_unit u{
  csl_port p1(input,  br1),
           p2(output, br2),
           p3(output, br3);
```

```
    u(){}
};
```

The resulting Verilog code is:

```
module u(p1,p2,p3);
  input  [3:0] p1;
  output [1:0] p2;
  output [3:0] p3;
endmodule
```

### *1.1.3.1 Bitrange usage and rules*

Bitranges can be declared on the global scope as well as inside certain CSL classes as shown in Table 1.6:

**TABLE 1.6** CSL bitrange declaration in other CSL classes

| CSL class | Can be declared in |
|---|---|
| global scope | YES |
| CSL Unit | YES |
| CSL Signal Group | YES |
| CSL Interface | YES |
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |
| CSL Register | - |
| CSL Register File | - |
| CSL Fifo | - |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | YES |
| CSL Field | YES |

### *1.1.3.2 CSL Bitrange commands*

The following commands are specific to csl_bitranges

| CSL bitrange commands |
|---|
| csl_bitrange obj_name(num_expr); |
| csl_bitrange bitrange_object_name(upper_limit, lower_limit); |
| csl_bitrange bitrange_object_name1(bitrange_obj_name0); |
| bitrange_name.set_offset(numeric_expression); |

### *1.1.4 CSL Field*

Fields are named bitranges. Fields can be the full bit range of a signal or part of that bitrange. A field has the same characteristics as a bitrange with following differences:

- fields can have a hierarchical structure
- an enum can be associated with a field
- a field in variable can be set an enum item value from the associated enum

Fields are declared with different constructors. If the field is non-hierarchical, the declaration is similar to the bitrange object, the only difference being the optional enum item name that can be optionally passed to the constructor parameter list as shown below:

```
csl_field field_name(constructor_parameters);
```

**Bold** text represents language reserved syntax, *italics* are user defined variables.

The constructor parameters above can be a single numerical expression (the width of the field), two numerical expressions (the upper and lower index of the width of the field) or the name of a bitrange object (this is the copy constructor that creates a field with the same properties of the copied bitrange). All these parameters can be optionally followed by an enum name or enum item name:

```
constructor parameters ::= num_expr [, enum]
                         |num_expr, num_expr [, enum]
                         |bitrange_name [, enum]
```

### *1.1.4.1 CSL Field usage and rules*

Non hierarchical fields can be declared in the CSL source file according to the rules in Table 1.7

**TABLE 1.7** CSL non-hierarchical field declaration rules

| CSL class | Can be declared in |
|---|---|
| global scope | YES |
| CSL Unit | YES |
| CSL Signal Group | YES |
| CSL Interface | YES |

**TABLE 1.7** CSL non-hierarchical field declaration rules

| CSL class | Can be declared in |
|---|---|
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |
| CSL Register | - |
| CSL Register File | - |
| CSL Fifo | - |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field* | YES |

*only if field is hierarchical

If the field is hierarchical, it will be defined, using a CSL class syntax, in the global scope (see Table 1.8) as shown below:

```
csl_field field_name {
  (objects declarations/instantiations)+
  field_name(){
    (field methods calls)+
  }
};
```

**Bold** text represents language reserved syntax, *italics* are user defined variables.

**TABLE 1.8** CSL hierarchical field definition rules

| CSL class | Can be defined in |
|---|---|
| global scope | YES |
| CSL Unit | - |
| CSL Signal Group | - |
| CSL Interface | - |
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |
| CSL Register | - |
| CSL Register File | - |

**TABLE 1.8** CSL hierarchical field definition rules

| CSL class | Can be defined in |
|---|---|
| CSL Fifo | - |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field | - |

Other hierarchical fields can be instantiated inside, or non hierarchical fields can be declared (see and Table 1.7)

**TABLE 1.9** CSL hierarchical field instantiation rules

| CSL class | Can be instantiated in |
|---|---|
| global scope | - |
| CSL Unit | YES |
| CSL Signal Group | YES |
| CSL Interface | YES |
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |
| CSL Register | - |
| CSL Register File | - |
| CSL Fifo | - |
| CSL Memory Map | - |
| CSL Memory Map Page | - |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field | YES |

If a field does not have a width the width should be inferred from the set_enum command.
Note this only applies to base fields.

```
csl_enum e {
a = 0
```

```
};
csl_field f {
f () {
   set_enum(e); // this should set the width automatically
                // set_width should be optional
}
};
```

A field with an associated enumerated type is used to declare other objects. The other object can be set to an enum item value.

### *1.1.4.2 CSL Field common commands*

Most of the commands available for bitrange are also available for field. These can be called on a field object (non-hierarchical) or inside the field constructor (hierarchical). The following commands apply both to hierarchical and non-hierarchical fields .

**CSL Field common commands**

```
csl_field field_name;
csl_field field_name([int width][, csl_enum enum]);
csl_field field_name(int upper, int lower[, csl_enum enum]);
csl_field obj_name(num_expr[,enum_or_enum_item]);
csl_field obj_name(num_expr, num_expr[, enum_or_enum_item]);
csl_field field_name(bitrange_name[, enum_name | enum_item]);
csl_field::set_width(num_expr);
csl_field::set_bitrange(num_expr);
[field_name.]set_offset(num_expr);
```

The following commands apply to non-hierarchical fields only:

**CSL Field non-hierarchical only commands**

```
csl_field::set_value(num_expr);
csl_field::set_enum(enum_name);
[field_name.]set_enum_item(enum_item_name);
```

To control the positioning of field objects or instances in hierarchical fields, the following hierarchical specific commands are used

**CSL Field hierarchical only commands**

```
csl_field::set_field_position(field_name, numeric_expression);
csl_field::set_next(field_name_left, field_name_right);
```

**CSL Field hierarchical only commands**

```
csl_field::set_previous(field_name_right, field_name_left);
field_obj_name.add_allowed_range(num_expr,num_expr);
```

### *1.1.5 CSL Parameter*

#### *1.1.5.1 Syntax for declaration:*

```
csl_parameter parameter_definition_list;
parameter_definition_list = identifier( numeric_expression
[,width_expression ]);
```

**Legal Examples:**

```
csl_parameter x(1);
csl_parameter y(2'b1);
csl_parameter z(2,5);  //the value is 2, width is set to 5 bits
                       // (5'b00010)
csl_parameter x1(x);  // parameter x1 will be 1
csl_parameter x2(4'd10); // the value of parameter is 10, the width is
                         // 4 bits
```

**Illegal Examples:**

```
csl_parameter x2(2,x1); //width is set by parameter x1
csl_parameter x3(x2,4); //value is set by parameter x2
csl_parameter x4(x1,x2);
```

**Parameters can be declared in:**

**TABLE 1.10** CSL_parameter declaration rules

| CSL class | Can be declared in |
|---|---|
| global scope | - |
| CSL Unit | YES |
| CSL Signal Group | YES |
| CSL Interface | YES |
| CSL Testbench | - |
| CSL Vector | - |
| CSL State Data | - |

| CSL class | Can be declared in |
|---|---|
| CSL Register | YES |
| CSL Register File | YES |
| CSL Fifo | YES |
| CSL Memory | YES |
| CSL Memory Map | ? |
| CSL Memory Map Page | ? |
| CSL Isa Element | - |
| CSL Isa Field | - |
| CSL Field* | - |

**NOTE:Add to future: to be included in other classes too**

Parameters can have a width (specified by the width_expression). The radix and signed type is specified with the assigned value.
- A parameter declaration with no width shall default to the width of the final value assigned to the parameter, after all value overrides have been applied.
- A parameter with a width specification shall be the width of the parameter declaration and shall not be affected by value overrides.
- A parameter with no width specification and for which the final values assigned to it is unsized, shall have an implied width of 32bit.

```
e.g. csl_parameter x(3);  //x is 32 bits
```

If the values is sized, then the parameter width is given by the specified width:

```
e.g. csl_parameter x(4'd3); //x is 4 bits
```

**NOTE: If parameter width is lesser than size of actual parameter value, then it is an error.**

### *1.1.5.2 Syntax for overriding parameter values:*

```
type_instantiation := type #(list_of_parameters_override_values)
instance_name instantiation_options;
list_of_parameters_override_values := formal_to_actual_override |
ordered_override
formal_to_actual_override := .parameter_identifier(
numeric_expression)
ordered_override := numeric_expression (, numeric_expression)+
```

The difference between formal to actual override and ordered override is that in the formal to actual

override the override values can be given in any order since they are always grouped with their targeted parameter identifier, whereas in order override the override values have to be specified in the same order the parameters have been declared.

#### *1.1.5.2.1 Examples for ordered override:*

Assuming we have ONE parameter defined in class definition for type a:

```
a #(4) a0(.x(y),.z(t));
//overrides the first parameter in instance a0 of type a with value 4
```

Assuming we have THREE parameters defined in class definition for type b:

```
b #(4,3,2) b0;
//overrides all 3 parameters in instance b0 of type b with values 4,3
and 2 respectively
```

Assuming we have FIVE parameters defined in class definition for type b:

```
b #(4,3,2) b0;
//overrides the first 3 parameters in instance b0 of type b with values
4,3 and 2 respectively and leaves the remain 2 with their assigned or
default values.
```

**EXAMPLE :**

Order override

CSL CODE

```
csl_unit a{
  csl_parameter pr_a1(6);
  csl_parameter pr_a2(3,5);
  csl_parameter pr_a3(2'b11);
  a(){}
};
csl_unit b{
  csl_parameter pr_b1(20);
  csl_parameter pr_b2(3,8);
  csl_parameter pr_b3(3'b011);
  b(){}
};
csl_unit top{
   a #(4, ,2) a0;
   b #(3,1,2) b0;
  top(){}
};
```

VERILOG CODE

```
module a;
  parameter pr_a1=32'd6;
  parameter pr_a2=5'd3;
  parameter pr_a3=2'b11;
endmodule
module b;
  parameter pr_b1=32'd20;
  parameter pr_b2=8'd3;
  parameter pr_b3=3'b011;
endmodule
module top;
  a #(4, ,2) a0;
  b #(3,1,2) b0;
endmodule
```

#### *1.1.5.2.2 Example of formal to actual override:*

If there are 3 parameters in a unit: m,n,p (declared in this order) and we want to override p and m with values 8 and 7 this is how it would look like for each type of override:

formal to actual override:

```
a #(.p(8),.m(7)) a0(.x(y),.z(t));
//notice how the order is not important
```

as opposed to ordered override:

```
a #(7,,8) a0(.x(y),.z(t));
//order is the same as the order of declaration of parameters + the
skipped parameter is specified with an extra comma
```

**EXAMPLE :**

If an interface ifc1 has 2 parameters (param1, param2) and 2 input ports (p1, p2) and it is instantiated in the unit u, then we can use the parameters in different ways:

CSL CODE:

```
csl_interface ifc1{
    csl_parameter param1(6,16);
    csl_parameter param2(8'b00000000);
    csl_port p1(input,param1);
    csl_port p2(input,param2);
    ifc1(){}
};
csl_unit u{
    ifc1 #(.p1(12),.p2(10)) ifc_in;
```

```
    ifc1 #(10,12) ifc_out;
    u(){
    ifc_out.reverse();
    }
};
```

VERILOG CODE:

```
module unit_u(ifc_in_p1,ifc_in_p2, ifc_out_p1, ifc_out_p2);
parameter param1 = 12;
parameter param2 = 10;
input [param1-1:0] ifc_in_p1;
input [param2-1:0] ifc_in_p2;
output [param1-1:0] ifc_out_p1;
output [param2-1:0] ifc_out_p2;
endmodule
```

#### *1.1.5.2.3 Override function*

Also a method is available for overriding a parameter and is invoked according to the syntax:

*instance_name*`.override_parameter(`*parameter_name, numeric_expression*`);`

parameter_name is the identifier of the csl parameter defined in the instantiated type and numeric_expression is the new value assigned to that parameter. The instance can be any of the types where is allowed to have a parameter declared:

- units
- interfaces
- signal groups
- registers
- register files
- fifos
- memoris

**EXAMPLE :**

In this example the unit *a* has three parameters and the first one is overriden in *top* unit.

CSL CODE

```
csl_unit a{
    csl_parameter pr_a1(6);
    csl_parameter pr_a2(3,5);
    csl_parameter pr_a3(2'b11);
  a(){}
};
csl_unit top{
```

```
  a a0;
top(){
  a0.override_parameter(pr_a1,4);
}
};
```

VERILOG CODE

```
module a;
  parameter pr_a1=32'd6;
  parameter pr_a2=5'd3;
  parameter pr_a3=2'b11;
endmodule
module top;
a #(.pr_a1(4)) a0();
endmodule
```

### *1.1.6 Prefixes and suffixes*

**NOTE:**
**Objects contained in CSL classes may be prefixed/suffixed with a user defined string. This is set by calling the set_prefix/set_suffix methods inside the class constructor.**

At this point there are 2 CSL classes that support prefixing and suffixing in the generated code: interfaces and signal groups. For both types there is also default automatic prefixing activated.
The default prefix is the instantiated container object's name prepended to the contained object's name.

**EXAMPLE :**
In the following example an interface and a signal group are instantiated in a top unit. In this case the default prefix for the ports and signals will be the name of interface instances and the name of signal_group instance.
If an interface ifc contains a port x and the interface is instantiated by the name ifc0 and the set_prefix() method is not called, the generated port name will be: ifc0_x.

CSL Code

```
csl_interface ifc{
  csl_port p_in(input,8);
  csl_port p_out(output,8);
  ifc() {}
};
csl_signal_group sg{
  csl_signal s1(8);
```

```
  csl_signal s2(8);
  sg() {}
};
csl_unit top{
  ifc ifc0;      //generated ports will be : ifc0_p_in , ifc0_p_out
  sg sg0;        //generated ports will be : sg0_s1 , sg0_s2
  top(){}
};
```

VERILOG Code

```
module top(ifc0_p_in,
           ifc0_p_out);
  input [7:0] ifc0_p_in;
  output [7:0] ifc0_p_out;
  wire [7:0] sg0_s1;
  wire [7:0] sg0_s2;
endmodule
```

### *1.1.6.1 Syntax for set_prefix() method:*

```
[ifc/sg_instance_name.]set_prefix("string");
```

The set_prefix() method can be called on both inside the class constructor and on the instance name due to the special nature of the interface and signal group instances. Note that calling set_prefix() inside the class constructor of an interface, can result in duplicate ports if that interface is instantiated multiple times in the same class.
The prefix specified with the set_prefix() method overrides the default prefix set by the compiler.

**EXAMPLE :**

The *set_prefix()* method is called on the instance name of an interface.

CSL CODE

```
csl_interface ifc{
  csl_port p_in1(input,8);
  csl_port p_in2(input,4);
  csl_port p_in3(input);
  csl_port p_out1(output,8);
  csl_port p_out2(output,4);
  ifc(){}
};
csl_unit a{
  ifc ifca;
```

```
  a(){
  ifca.set_prefix("addr");
  }
};
csl_unit b{
  ifc ifcb;
  b(){}
};
csl_unit top{
  a a0;
  b b0;
  top(){}
};
```

VERILOG CODE

```
module a(addr_p_in1,
         addr_p_in2,
         addr_p_in3,
         addr_p_out1,
         addr_p_out2);
  input [7:0] addr_p_in1;
  input [3:0] addr_p_in2;
  input  addr_p_in3;
  output [7:0] addr_p_out1;
  output [3:0] addr_p_out2;
endmodule
module b(ifcb_p_in1,
         ifcb_p_in2,
         ifcb_p_in3,
         ifcb_p_out1,
         ifcb_p_out2);
  input [7:0] ifcb_p_in1;
  input [3:0] ifcb_p_in2;
  input  ifcb_p_in3;
  output [7:0] ifcb_p_out1;
  output [3:0] ifcb_p_out2;
endmodule
module top();
  a a0();
  b b0();
```

```
endmodule
```

**EXAMPLE :**

The *set_prefix()* method is called inside the interface class constructor.

CSL CODE

```
csl_interface ifc{
  csl_port p_in1(input,8);
  csl_port p_in2(input,16);
  csl_port p_out(output,8);
ifc(){
set_prefix("addr");
}
};
csl_unit a{
  ifc ifca;
  a(){}
};
csl_unit b{
 ifc ifcb;
 b(){}
};
csl_unit top{
 a a0;
 b b0;
top(){}
};
```

VERILOG CODE

```
module a(addr_p_in1,
         addr_p_in2,
         addr_p_out);
  input [7:0] addr_p_in1;
  input [15:0] addr_p_in2;
  output [7:0] addr_p_out;
endmodule
module b(addr_p_in1,
         addr_p_in2,
         addr_p_out);
  input [7:0] addr_p_in1;
  input [15:0] addr_p_in2;
  output [7:0] addr_p_out;
```

```
endmodule
module top();
 a a0;
 b b0;
endmodule
```

**Hierarchical interfaces:**
Interfaces can be hierarchical. For example an interface ifc1 can contain interface ifc2 which on its turn contains interface ifc3. The examples bellow show some possible situations and how names are generated in verilog code based on where the set_prefix () method are called.

CSL CODE

```
csl_interface ifca{
  csl_port p_in(input,4);
  csl_port p_out(output,4);
ifca(){
// set_prefix("q");
}
};
csl_interface ifcb{
  ifca ifca0;
csl_port x(input,8);
ifcb(){
//set_prefix("r");
}
};
csl_interface ifcc{
  ifcb ifcb0;
csl_port y(output,8);
ifcc(){
//set_prefix("s");
}
};
csl_unit top{
ifcc ifcc0;
ifcc ifcc1;
top(){
//ifcc0.set_prefix("p");
//ifcc1.set_prefix("t");
}
};
```

The names generated in verilog code from the csl codes obtained by different uncomenting

*set_prefix()* call combinations are given in following tables:

**TABLE 1.11** Case 1: No set_prefix() calls in top unit body

| | | | | ifcc0 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| **prefix** | **q** | **r** | **s** | **x** | **y** | **z** |
| **inter-face** | **ifca** | **ifcb** | **ifcc** | **VERILOG port_names** | | |
| | 0 | 0 | 0 | ifcc0_ifcb0_ifca0 | ifcc0_ifcb0 | ifcc0 |
| | 0 | 0 | 1 | s_ifcb0_ifca0 | s_ifcb0 | s |
| | 0 | 1 | 0 | ifcc0_r_ifca0 | ifcc0_r | ifcc0 |
| | 0 | 1 | 1 | s_r_ifca0 | s_r | s |
| | 1 | 0 | 0 | ifcc0_ifcb0_q | ifcc0_ifcb0 | ifcc0 |
| | 1 | 0 | 1 | s_ifcb0_q | s_ifcb0 | s |
| | 1 | 1 | 0 | ifcc0_r_q | ifcc0_r | ifcc0 |
| | 1 | 1 | 1 | s_r_q | s_r | s |

| | | | | ifcc1 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| **prefix** | **q** | **r** | **s** | **x** | **y** | **z** |
| **inter-face** | **ifca** | **ifcb** | **ifcc** | **VERILOG port_names** | | |
| | 0 | 0 | 0 | ifcc1_ifcb0_ifca0 | ifcc1_ifcb0 | ifcc1 |
| | 0 | 0 | 1 | s_ifcb0_ifca0 | s_ifcb0 | s |
| | 0 | 1 | 0 | ifcc1_r_ifca0 | ifcc1_r | ifcc1 |
| | 0 | 1 | 1 | s_r_ifca0 | s_r | s |
| | 1 | 0 | 0 | ifcc1_ifcb0_q | ifcc1_ifcb0 | ifcc1 |
| | 1 | 0 | 1 | s_ifcb0_q | s_ifcb0 | s |
| | 1 | 1 | 0 | ifcc1_r_q | ifcc1_r | ifcc1 |
| | 1 | 1 | 1 | s_r_q | s_r | s |

0=no set_prefix in interface ifci body
1= set_prefix in interface ifcx body
ifci=ifca, ifcb or ifcc

**TABLE 1.12** Case 2: ifcc0.set_prefix("p") in top unit body

| | | | | ifcc0 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| **prefix** | **q** | **r** | **s** | **x** | **y** | **z** |
| **interface** | **ifca** | **ifcb** | **ifcc** | **VERILOG port names** | | |
| | 0 | 0 | 0 | p_ifcb0_ifca0 | p_ifcb0 | p |
| | 0 | 0 | 1 | p_ifcb0_ifca0 | p_ifcb0 | p |
| | 0 | 1 | 0 | p_r_ifca0 | p_r | p |
| | 0 | 1 | 1 | p_r_ifca0 | p_r | p |
| | 1 | 0 | 0 | p_ifcb0_q | p_ifcb0 | p |
| | 1 | 0 | 1 | p_ifcb0_q | p_ifcb0 | p |
| | 1 | 1 | 0 | p_r_q | p_r | p |
| | 1 | 1 | 1 | p_r_q | p_r | p |

| | | | | ifcc1 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| **prefix** | **q** | **r** | **s** | **x** | **y** | **z** |
| **inter-face** | **ifca** | **ifcb** | **ifcc** | **VERILOG port names** | | |
| | 0 | 0 | 0 | ifcc1_ifcb0_ifca0 | ifcc1_ifcb0 | ifcc1 |
| | 0 | 0 | 1 | s_ifcb0_ifca0 | s_ifcb0 | s |
| | 0 | 1 | 0 | ifcc1_r_ifca0 | ifcc1_r | ifcc1 |
| | 0 | 1 | 1 | s_r_ifca0 | s_r | s |
| | 1 | 0 | 0 | ifcc1_ifcb0_q | ifcc1_ifcb0 | ifcc1 |
| | 1 | 0 | 1 | s_ifcb0_q | s_ifcb0 | s |
| | 1 | 1 | 0 | ifcc1_r_q | ifcc1_r | ifcc1 |
| | 1 | 1 | 1 | s_r_q | s_r | s |

0=no set_prefix in interface ifci body
1= set_prefix in interface ifcx body
ifci=ifca, ifcb or ifcc

**TABLE 1.13** Case 3: ifcc0.set_prefix("p") and ifc1.set_prefix("t") in top unit body

| | | | | ifcc0 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| prefix | q | r | s | x | y | z |
| interface | ifca | ifcb | ifcc | VERILOG port names | | |
| | 0 | 0 | 0 | p_ifcb0_ifca0 | p_ifcb0 | p |
| | 0 | 0 | 1 | p_ifcb0_ifca0 | p_ifcb0 | p |
| | 0 | 1 | 0 | p_r_ifca0 | p_r | p |
| | 0 | 1 | 1 | p_r_ifca0 | p_r | p |
| | 1 | 0 | 0 | p_ifcb0_q | p_ifcb0 | p |
| | 1 | 0 | 1 | p_ifcb0_q | p_ifcb0 | p |
| | 1 | 1 | 0 | p_r_q | p_r | p |
| | 1 | 1 | 1 | p_r_q | p_r | p |

| | | | | ifcc1 | | |
|---|---|---|---|---|---|---|
| | | | | CSL port names | | |
| prefix | q | r | s | x | y | z |
| interface | ifca | ifcb | ifcc | VERILOG port names | | |
| | 0 | 0 | 0 | t_ifcb0_ifca0 | t_ifcb0 | t |
| | 0 | 0 | 1 | t_ifcb0_ifca0 | t_ifcb0 | t |
| | 0 | 1 | 0 | t_r_ifca0 | t_r | t |
| | 0 | 1 | 1 | t_r_ifca0 | t_r | t |
| | 1 | 0 | 0 | t_ifcb0_q | t_ifcb0 | t |
| | 1 | 0 | 1 | t_ifcb0_q | t_ifcb0 | t |
| | 1 | 1 | 0 | t_r_q | t_r | t |
| | 1 | 1 | 1 | t_r_q | t_r | t |

0=no set_prefix in interface ifci body
1= set_prefix in interface ifcx body
ifci=ifca, ifcb or ifcc

### *1.1.6.2 No prefix:*

Sometimes it may be required that port or signal names should not be prefixed by the compiler. There are two ways to accomplish this using the following syntax options:

- calling set_prefix() method with an empty string:

```
[ifc/sg_instance_name.]set_prefix("");
```

- calling the specialized no_prefix() method

```
[ifc/sg_instance_name.]no_prefix();
```

Note that just like set_prefix() method, no_prefix() can also be called both inside the constructor for a signal group or interface or on a signal group/interface instance.

If a port (x) belongs to an interface (ifc) in a unit the output name for the port will be ifc_x
Using no_prefix()/set_prefix(“”), the resulting name for port x will be x.

**EXAMPLE :**

Using set_prefix() method.

CSL CODE

```
csl_interface ifc{
   csl_port p_in(input,8);
   csl_port p_out(output,8);
   ifc(){}
};
csl_unit top{
   ifc ifc0;
   top(){
   ifc0.set_prefix(“”);
   }
};
```

VERILOG CODE

```
module top(p_in,
           p_out);
  input [7:0] p_in;
  output [7:0] p_out;
endmodule
```

**EXAMPLE :**

This example use the no_prefix() method.

CSL CODE

```
csl_interface ifc{
   csl_port p_in(input,8);
   csl_port p_out(output,8);
   ifc(){}
};
```

```
csl_unit top{
  ifc ifc0;
  top(){
  ifc0.no_prefix();
  }
};
```

VERILOG CODE

```
module top(p_in,
           p_out);
  input [7:0] p_in;
  output [7:0] p_out;
endmodule
```

**EXAMPLE :**

Mixing the set_prefix() and the no_prefix() methods.

CSL CODE

```
csl_interface ifca{
   csl_port in1(input,4);
   csl_port out1(output,4);
   ifca(){
   set_prefix("q");
   }
};
csl_interface ifcb{
   ifca ifca0;
   csl_port x(input,8);
   ifcb(){
   no_prefix();
   }
};
csl_interface ifcc{
   ifcb ifcb0;
   csl_port y(output,8);
   ifcc(){}
};
csl_unit top{
  ifcc ifcc0;
  ifcc ifcc1;
  top(){
```

```
  ifc0.set_prefix("p");
  }
};
```

VERILOG CODE

```
module top(p_q_in1,
           p_q_out1,
           p_x,
           p_y,
           ifcc1_q_in1,
           ifcc1_q_out1,
           ifcc1_x,
           ifcc1_y);
input [7:0] p_q_in1;
output [7:0] p_q_out1;
input [7:0] p_x;
output [7:0] p_y;
input [7:0] ifcc1_q_in1;
output [7:0] ifcc1_q_out1;
input [7:0] ifcc1_x;
output [7:0] ifcc1_y;
endmodule
```

### *1.1.6.3 Syntax for set_suffix() method*

```
[if/sg_instance_name.]set_suffix("string");
```

Just like prefixing, objects can be suffixed with a user defined string. Unlike prefixes however there is no default suffixing added by the compiler.

Also, if the suffixing method is called as follows:

```
set_suffix("");
```

A warning will be printed because the prefix supplied is empty and nothing will happen since this is a redundant command.

**EXAMPLE :**

//

CSL CODE

```
csl_interface ifc1{
  csl_port p1(input);
  csl_port p2(output);
  ifc1(){
```

```
    set_suffix("xxx");
    }
};
csl_interface ifc2{
    csl_port p21(input),p22(output);
    ifc2(){}
};
csl_unit u1{
    ifc1 ifc11;
    ifc2 ifc21;
    u1(){
    ifc21.set_suffix("21");
    }
};
```

VERILOG CODE

```
module u1(ifc11_xxx_p1,
          ifc11_xxx_p2,
          ifc21_21_p21,
          ifc21_21_p22);
input ifc11_xxx_p1;
output ifc11_xxx_p2;
input ifc21_21_p21;
output ifc21_21_p22;
endmodule
```

### *1.1.7 Operators*

The following operators are supported in CSL

**TABLE 1.14**

| |
|---|
| 1.1.7.1 Part Select |
| 1.1.7.2 Concatenation operator |
| 1.1.7.3 Replication operator |

#### *1.1.7.1 Part Select*

Part select is used to address contiguous bits in a multibit port or signal, field and ISA field.
We can have constant part select and indexed part select. The constant part select is specified using the following syntax:

```
object[upper_index:lower_index]
```

Where *upper_index* indicates the more significant bit, while the *lower_index* indicates the less significant bit of the part select. Both *lower_index* and *upper_index* must be an integer or constant expressions that evaluate to an integer.

**EXAMPLE :**
Legal example

CSL CODE

```
csl_unit u{
   csl_port pin(input,8);
   csl_port pout(output,8);
   u(){
     pout[5:9]=pin[4:0];
}
};
```

Upper_index and lower_index must be within the bitrange of the multibit object. Values outside the object bitrange issue a compiler error.

**EXAMPLE :**
Illegal example

CSL CODE

```
csl_bitrange br(4,7);
csl_unit u{
   csl_port pin(br);
   csl_port pout(br);
   u(){
   pout[3:0]=pin[7:4]; // error because pout[3:0] is
                       // out of the pout bitrange
}
};
```

The indexed part select is specified using the following syntax:

```
object[base_index +: range_width] or

object[base_index -: range_width]
```

In the first expression we have an ascending bitrange that starts at the *base_index* and has the width *range_width*. The second expression specifies a descending bitrange starting at *base_index* and that has width *range_width*. The *range_width* must be an integer or a constant expression that evaluates to an integer. The *base_index* can be an integer or an expression that evaluate to an integer that can vary at run time.

**EXAMPLE :**

```
csl_signal data(8);

data[0 +:3] is equivalent to data[0:2]

data[5 -:4] is equivalent to data[5:2]
```

**Bit select:**

Bit select is used to address a particular bit from a multibit port or signal, a field or an ISA field.

The syntax of the bit select is:

```
object[index]
```

**EXAMPLE :**

```
csl_unit u{
   csl_port pin(input,8);
   csl_port pout(output);
   u(){
   pout=pin[6];
}
};
```

A bit can also be addressed using a part select whose *upper_index* and *lower_index* are equal.

**EXAMPLE :**

```
csl_unit u{
   csl_port pin(input,8);
   csl_port pout(output);
   u(){
   pout=pin[6:6];
}
};
```

Legal cases where part select can be used:

**TABLE 1.15** Part Select can be used with

| csl objects |
|---|
| Non-hierarchical fields |
| Ports |
| Signals |
| Non hierarchical ISA fields |
| HIDs when these identify one of the objects above |

**TABLE 1.16** Part Select can be used in

| operators and methods |
|---|
| LHS and RHS of assign statements |
| Expressions |
| Concatenation expressions |
| Replication expressions |
| F2a as actuals |
| Methods that accept ports and signals as arguments |

### EXAMPLE :

Part select used with ports.

CSL CODE

```
csl_interface ifcd{
  csl_port din(input,0,15);
  csl_port dout(output,0,15);
  ifcd(){
  }
};
csl_unit u_a{
  csl_port dt(input,4);
  csl_port pc(input);
  ifcd ifcd0;
u_a(){
  ifcd0.dout[15]=pc;
  ifcd0.dout[7:0]=ifcd0.din[15:8];
  ifcd0.dout[14:8]={ifcd0.din,ifcd0.din[3:0]};
  }
};
```

**EXAMPLE :**
Part Select used with signals.

CSL CODE

```
csl_interface ifcd{
  csl_port din(input,0,15);
  csl_port dout(output,0,15);
ifcd(){
  }
};
csl_unit u_a{
  ifcd ifcd0;
  csl_signal s(16);
u_a(){
    ifcd0.dout=s;
    s[7:0]=ifcd0.din[15:8]
    s[15:8]=ifcd0.din[7:0];
  }
};
```

### *1.1.7.2 Concatenation operator*

**Syntax:**

```
concatenation := { expression (, expression)* }
expression := port_hid | signal_hid |
              signal_group_hid | interface_hid |
             sized_constant | concatenation | replication_expression |
              operator_expression | part_select_expression
```

Concatenation is used to group a set of the same element type together.
There can be any number of expression arguments (at least 1). Arguments should be processed and concatenated in the given order.

If container structures (like signal group or interfaces) are used, the container elements are processed in the order they were declared in the container.

**Example:**
ifcB contains:

```
port x
```

ifcA contains:

```
port y, ifcB, port z
```

In a concatenation using ifcA (eg. **{** ifcA, t, 2'b01 **}** ) the output will be equivalent to:

```
{ ifcA_y, ifcA_ifcB_x, ifcA_z, t, 2'b01 }
```

Signal groups are used in concatenation in the same way.

**Where can concatenation be used:**

- assignment:

```
x = {a,b,c} or {a,b,c} = x; (LHS and RHS)
```

- connect statement:

```
x.connect({a,b,c});
This is NOT legal: {a,b,c}.connect(x);
```

- formal to actual:

```
(.x({a,b,c})); (actual only)
```

**EXAMPLE :**

These examples show the concatenation cases:

**1. Concatenation in assign**

CSL CODE

```
csl_unit u1{
  csl_port p11(input,4);
  csl_port p12(input,8);
  csl_port p13(input,32);
  csl_port p14(output,32);
  csl_port p15(output,4);
  csl_signal s1(44),s2(32);
  csl_signal s3(4);
   u1(){
    s1 ={p12,p11,p13};
    {p14,p15}={s2,s3};
}
};
```

VERILOG CODE

```
module u1(p11,
          p12,
          p13,
          p14,
          p15);
input [3:0] p11;
input [7:0] p12;
input [31:0] p13;
```

```
output [31:0] p14;
output [3:0] p15;
wire [43:0] s1;
wire [31:0] s2;
wire [3:0] s3;
assign s1={p12,p11,p13};
assign {p14,p15}={s2,s3};
endmodule
```

**2.Concatenation in connect() method**

CSL CODE

```
csl_unit u1{
  csl_port p1(input,32),p2(output,16);
  u1(){}
};
csl_unit u2{
  csl_port p1(input,16);
  csl_port p2(input,8);
  csl_port p3(input,8);
  csl_port p4(output,8);
  csl_port p5(output,8);
  u1 u1i;
  u2(){
    u1i.p1.connect_by_name({p1,p2,p3});
    u1i.p2.connect_by_name({p4,p5});
  }
};
```

VERILOG CODE

```
module u1(p1,
          p2);
input [31:0] p1;
output [15:0] p2;
endmodule
module u2(p1,
          p2,
          p3,
          p4,
          p5);
input [15:0] p1;
input [7:0] p2;
```

```
input [7:0] p3;
output [7:0] p4;
output [7:0] p5;
u1 u1i(.p1({p1,p2,p3}),.p2({p4,p5}));
endmodule
```

**3.Concatenation in formal to actual**

CSL CODE

```
csl_unit u1{
  csl_port p1(input,8);
  csl_port p2(output,64);
  u1(){}
};

csl_unit u2{
  csl_signal  s3(4),s4(4);
  csl_port p5(output,16),p1(output,32),p2(output,16);
   u1 u1i(.p1({s3,s4}),.p2({p1,p2,p5}));
  u2(){}
};
```

VERILOG CODE

```
module u1(p1,
          p2);
input [7:0] p1;
output [63:0] p2;
endmodule
module u2(p5,
          p1,
          p2);
output [15:0] p5;
output [31:0] p1;
output [15:0] p2;
wire [3:0] s3;
wire [3:0] s4;
u1 u1i(.p1({s3,s4}),.p2({p1,p2,p5}));
endmodule
```

### *1.1.7.3 Replication operator*

(also referred to as multi concatenation)

**Syntax:**

```
replication := { constant_expression concatenation }
```

The constant_expression is a constant numeric_expression (e.g: number, parameter) and has to have a positive, non-zero, non-z, non-x value. Replication joins together as many concatenations as given by the constant_expression value.

**Replications can be used in:**

- assignment (RHS only):

```
x = {2{a}};
This is NOT legal {2{a}} = x;
```

- connect statement (actual only):

```
x.connect({2{a,b,c}});
This is NOT legal: {2{a,b,c}}.connect(x)
```

- formal to actual (actual only):

```
(.x({4{a,b,c}}));
```

As a generale rule: Replications shall NOT be in expressions on the LHS of the assignment nor connected to output or inout ports.

**EXAMPLE :**

These examples show the replicate cases:

**1. Replicate in assign**

CSL CODE

```
csl_unit u_1{
 csl_port p21(input,8);
 csl_port p22(input,4);
 csl_signal s1(2),s2,s3(32);
 u_1(){
   s3={4{p21}};
   p22={2{s1}};
 }
};
```

VERILOG CODE

```
module u_1(p21,
           p22);
 input [7:0] p21;
```

```
  input [3:0] p22;
  wire [1:0] s1;
  wire s2;
  wire [31:0] s3;
  assign s3={4{p21}};
  assign p22={2{s1}};
endmodule
```

**2. Replicate in connect() method**

CSL CODE

```
csl_unit u_1{
  csl_port p11(input,16);
  csl_port p12(output,64);
  u_1(){}
};
csl_unit u_2{
  csl_port p21(input,4);
  csl_port p22(output,32);
  u_1 u1_i;
  u_2(){
    u1_i.p11.connect_by_name({4{p21}});
    u1_i.p12.connect_by_name({2{p22}});
}
};
```

VERILOG CODE:

```
module u_1(p11,
           p12);
  input [15:0] p11;
  output [63:0] p12;
endmodule
module u_2(p21,
           p22);
  input [3:0] p21;
  output [31:0] p22;
  u_1 u1_i(.p11({4{p21}}, .p12({2{p22}}));
endmodule
```

**3. Replicate in formal to actual**

CSL CODE

```
csl_unit u_1{
```

```
  csl_port p11(output,16);
  csl_port p12(input,32);
  u_1(){}
};
csl_unit u_2{
  csl_signal s21(4);
  csl_signal s22(4);
  u_1 u1_i(.p11({4{s21}}),.p12({8{s22}}));
  u_2(){}
};
```

VERILOG CODE

```
module u_1(p11,
           p12);
output [15:0] p11;
input [31:0] p12;
endmodule

module u_2;
wire [3:0] s21;
wire [3:0] s22;
u_1 u1_i(.p11({4{s21}}),.p12({8{s22}}));
endmodule
```

### *1.1.8 Numbers*

#### *1.1.8.1 Z and X numbers*

"Z "and " X" numbers don't exist in decimal number format. These numbers exist only for binary, hexadecimal and octal formats. In hardware language, the numerical digits "x" is equivalent with "unknown value " and "z" is equivalent with "high impedance". "z" number is the default value for a port when it doesn't receive any value.
The "x" and "z" apear for each bit from the number:

| | | | |
|---|---|---|---|
| In binary: | x => x; | z =>z; | 1 digit |
| In hexa: | x => xxxx ; | z => zzzz; | 4 digits |
| In octal: | x => xxx; | z => zzz; | 3 digits |

**EXAMPLE :**

a=4'bx; => a=xxxx;
b=3'bz; => b=zzz;

**EXAMPLE :**

" z" and "x" numbers used in concatenation expression:

CSL CODE

```
csl_unit u1{
  csl_port p1(input,3);
  csl_port p2(output,3);
  csl_signal s1(5);
  u1(){
    s1={p1,2'bz};
    p2={2'bx,1'bz};
  }
};
```

VERILOG CODE

```
module u1(p1,
          p2);
  input [2:0] p1;
  output [2:0] p2;
  wire [4:0] s1;
  assign   s1 = {p1,2'bz};
  assign   p2 = {2'bx,1'bz};
endmodule
```

**NOTE:Uncertain & Unknown & to be sorted out**

**Common commands:**

```
generate_decoder(csl_unit);
```

## 1.2 CSL Language Commands

### *1.2.1 CSL Enum*

```
csl_enum enum_name {enum_item
[=item_index](,enum_item[=item_index])*};
```

**DESCRIPTION :**

Declares a new csl_enum.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

The following example declares an enum named *e_opcodes*.

CSL CODE

```
csl_enum e_opcodes {
  ADD = 1,
  SUB = 5,
  SUBC = 9
};
```

### *1.2.2 Bitrange*

A bit range is used to declare the upper index (MSB) and the lower index (LSB) of a signal. Moreover, bit ranges are also used in expressions to selects parts of a signal bit ranges using either a upper index and a lower index (i.e. [upper:lower]) or a single index (i.e. [index]).

**csl_bitrange** *obj_name*(num_expr);

**DESCRIPTION :**

Creates a bit range object called *obj_name*: *num_expr* - width of the bitrange, it will transform into a range[n-1:0];

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

Creates a bitrange named *br* using a numeric expression to specifies the bit range's width, which is used to set the range for a signal *s1* and a port *p1*.

CSL CODE

```
csl_bitrange br(4+2);
csl_unit u{
  csl_signal s1(br);
  csl_port p1(input, br);
  u(){}
};
```

VERILOG CODE

```
module u(p1);
  input [5:0] p1;
  wire [5:0] s1;
endmodule
```

**csl_bitrange** *bitrange_object_name(upper_limit, lower_limit);*

## DESCRIPTION :

Creates a bit range object which can be added to a signal in the current scope. The upper and lower limits of the bit range object must be specified with constant numeric expressions (otherwise compile time error will occur).

[ ***CSL Language Commands Summary*** ]

## EXAMPLE :

A bitrange *br2* is declared with an upper limit and lower limit.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(7,0);
csl_unit u{
  csl_port p1(input, br1), p2(output, br2);
  u(){}
};
```

VERILOG CODE

```
module u(p1,
         p2);
  input [3:0] p1;
  output [7:0] p2;
endmodule
```

**csl_bitrange** *bitrange_object_name1*(*bitrange_obj_name0*);

**DESCRIPTION :**

Creates a bitrange with the param bitrange object - this is the copy constructor;

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

A bitrange *br2* is declared, using another bitrange *br1*.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(br1);
csl_unit u{
  csl_port p1(input, br1), p2(output, br2);
  u(){}
};
```

VERILOG CODE

```
module u(p1,
         p2);
  input [3:0] p1;
  output [3:0] p2;
endmodule
```

*bitrange_name*.**set_offset(***numeric_expression***);**

**DESCRIPTION :**

Set the offset, value to be added to both lower and upper index of the bitrange.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

The *set_offset()* method left shifts changes the index values of a bitrange *br1* by the shift amount.

CSL CODE

```
csl_bitrange br1(4);
br1.set_offset(8);
csl_unit u{
  csl_port p1(input, br1), p2(output, br1);
  u(){}
};
```

VERILOG CODE

```
module u(p1,
         p2);
  input [11:8] p1;
  output [11:8] p2;
endmodule
```

### *1.2.3 Field*

**csl_field** *field_name*;

**DESCRIPTION :**

Creates a field named *field_name.*

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

In this example a field is declared, named *fd1*.

CSL CODE

```
csl_field fd1;
```

VERILOG CODE

```
//
```

```
csl_field field_name([int width][, csl_enum enum]);
```

**DESCRIPTION :**

Creates a field *field_name* , by adding the width and an enum.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example creates a field named *fd1* with width 4.

CSL CODE

```
csl_field fd1(4);
```

VERILOG CODE

```
//
```

**csl_field** *field_name*(int upper, int lower**[**, csl_enum enum**]);**

**DESCRIPTION :**

Creates a field *field_name* using the upper and lower range.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example creates a 4 bit field named *fd1.*

CSL CODE

```
csl_field fd1(3,0);
```

VERILOG CODE

```
//
```

### *1.2.4 CSL Address Range*

Non-hierarchical:

**csl_field** *obj_name*(num_expr[,*enum_or_enum_item*]);

**DESCRIPTION :**

Instantiate a field:

num_expr - width of the field, it will transform into a range [n-1:0];

enum_or_enum_item - the enum or the enum item will be associated to the field;

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example creates a field named *fd1* with width 4.

CSL CODE

```
csl_field fd1(4);
```

**csl_field** *obj_name(*num_expr, num_expr*[, enum_or_enum_item]);*

**DESCRIPTION :**

Instantiate a field with lower bit and upper bit :

param num_expr - lower bound of the field;
param num_expr- upper bound of the field;
param enum_or_enum_item - the enum or the enum item will be associated to the field;

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example creates a field named *fd2* with the width 8, using the lower bound and upper bound constructor.

CSL CODE

```
csl_field fd2(0,7);
```

**csl_field** *field_name*(*bitrange_name*[, *enum_name* | *enum_item*]);

**DESCRIPTION :**

Instantiate a field with the width given by a bitrange object name;

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example creates a field named *fd* with the width set by a bitrange *br,* using the bit range constructor.

CSL CODE

```
csl_bitrange br(4);
csl_field fd(br);
```

**csl_field::set_width(**num_expr);

**DESCRIPTION :**

Sets the width of field to num_expr.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example declares a field named *fld,* with a field class declaration.

CSL CODE

```
csl_field fld{
fld(){
  set_width(4);
}
};
```

```
csl_field::set_bitrange(num_expr);
```

**DESCRIPTION :**

Set the bitrange for the field.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example declares a bitrange *br1* and creates a field *fld,* using the field class.

CSL CODE

```
csl_bitrange br1(4);
csl_field fld{
fld(){
  set_bitrange(br1);
}
};
```

```
[field_name.]set_offset(num_expr);
```

**DESCRIPTION :**

Set the value to be added to both lower and upper index of the field. This method cannot be called if the field has not set a width befor. In this case will be an error. The lower and upper indices are both incremented by the offset amount.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

Set the offset for a field *fd1*.

CSL CODE

```
csl_field fld{
  fld(){
    set_width(4);
  }};
csl_unit u{
  fld fld1;
  u(){
    fld1.set_offset(2);
  }};
```

```
csl_field::set_value(num_expr);
```

**DESCRIPTION :**

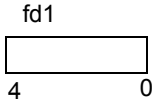
Associates an numeric value to the field object.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

This example declares a field *fd1* and assigns it the value 16.

**FIGURE 1.1**




CSL CODE

```
csl_field fd1{
  fd1(){
     set_value(16);
  }
};
```

```
csl_field::set_enum(enum_name);
```

**DESCRIPTION :**

Associates an enum to the field object.

[ *CSL Language Commands Summary* ]

**EXAMPLE :**

This example associates an enum with the field *fd.*

CSL CODE

```
csl_enum enum1{fd1, fd2, fd3};
csl_field fd{
  fd(){
    set_enum(enum1);
  }};
```

```
[field_name.]set_enum_item(enum_item_name);
```

**DESCRIPTION :**

Associates an enum_item to the field object. This method can be called if that field has an enum associated.

[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

In this example an enum itemit is associated with the field *fd.*The field *fd_i* is declared and assign the enum item value s1.

CSL CODE

```
csl_enum enum1{s1, s2, s3};
csl_field fd{
  fd(){
    set_enum(enum1);
  }};
csl_unit u{
  fd fd_i;
  u(){
    fd_i.set_enum_item(s1);
  }};
```

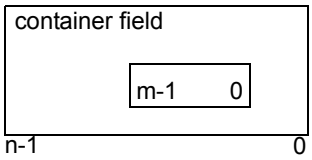
Hierarchical:

**csl_field::set_field_position**(field_name, numeric_expression);

**DESCRIPTION :**

Set the absolute position of a field in another field. If set_field_position is not set then the first instance instantiated is the left most field. The position of the field is the number of bits from 0-bit position of the container field.

**FIGURE 1.2**

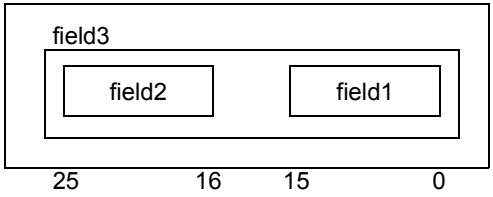



[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

1. Declare two field types: *field1* and *field2*.
2. Create a *field1* and *field2* object in *field3*.
3. Set the width of *field3* to 26 bits.
4. Set the position of *f1* and *f2*.
5. Set the position of *f1* to bit position 0 in *field3*.
6. Set the position of f2 to bit position16 in *field3*.

**FIGURE 1.3**




CSL CODE

```
csl_field field1(0,15),field2(16,25);
csl_field field3 {
   field1 f1;
   field2 f2;
field3() {
   set_width(26);
   set_field_position(f1,0);
   set_field_position(f2,16);
}
};
```

```
csl_field::set_next(field_name_left, field_name_right);
```

**DESCRIPTION :**

Set “the next” field position of a field inside a format in a “linked-list“ way. In this way if the size of the fields change they will remain adjacent to each other and there are no offset changes;

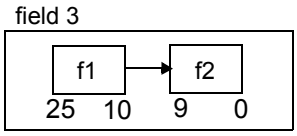
[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

1. Declare two field types: *field1* and *field2*.
2. Create a *field1* and *field2* object in *field3*.
3. Set the width of *field3* to 26 bits.
4. Set the position of *f1* and *f2*.
5. Set the position of *f1* to bit position 10 in *field3*.
6. Set_next position of f2 to the right of *f1.*

*f1* is located in *field3* bitrange [25:10] ;
*f2* is located in *field3* bitrange [9:0];

**FIGURE 1.4**




CSL CODE

```
csl_field field1(16),field2(10);
csl_field field3 {
   field1 f1;
   field2 f2;
field3() {
   set_width(26);
   set_field_position(f1,10);
   set_next(f1,f2);
}
};
```

```
csl_field::set_previous(field_name_right, field_name_left);
```

**DESCRIPTION :**

Set “the previous” field position of a field inside a format in a “linked-list“ way. In this way if the size of the fields change they will remain adjacent to each other and there are no offset changes;

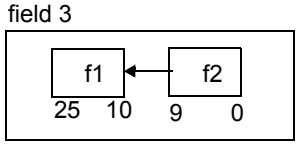
[ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

1. Declare two field types: *field1* and *field2*.
2. Create a *field1* and *field2* object in *field3*.
3. Set the width of *field3* to 26 bits.
4. Set the position of *f1* and *f2*.
5. Set the position of *f1* to bit position 10 in *field3*.
6. Set_previous position of f2 to the right of *f1*.

*f1* is located in *field3* bitrange [25:10] ;
*f2* is located in *field3* bitrange [9:0];

**FIGURE 1.5**




CSL CODE

```
csl_field field1(16),field2(10);
csl_field field3 {
   field1 f1;
   field2 f2;
field3() {
   set_width(26);
   set_field_position(f2,0);
   set_previous(f2,f1);
}
};
```

```
field_obj_name.add_allowed_range(num_expr,num_expr);
```

**DESCRIPTION :**

Sets the lower and upper bounds of the values that can be associated with a field.
to the field. [ ***CSL Language Commands Summary*** ]

**EXAMPLE :**

Adds an allowed range to a field *fld.Fld* can hold values in the range 0-15, but it is restricted to the value range 4-7. An assertion is generated which will trigger during simulation if *fld* is greater then 7 or less then 4.

CSL CODE

```
csl_field fld(0,3);
fld.add_allowed_range(4,7);
```

VERILOG CODE