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 Pattern generator

1.1.1 Directives

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

1.1.1.1 CSL include directive

The CSL include directive allows the user to include code from a 3rd party language to the CSL input. So far 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).

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

```
csl_include(language_type, "file_name");
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

In the include directive above, language type specifies the language the included file is written in and it's possible values are given below:

```
language_type ::= file_cplusplus | file_verilog
```

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 provide checking for 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 CSL include directive and the generated code. The included language type is Verilog in this case:

TABLE 1.4 CSL Include directive

Input CSL code	Generated Verilog code

`//Input CSL code`
`csl_unit dff {`
`  csl_port d (input, ...);`
`  csl_include(file_verilog,"vlogic");`
`};`
	`module dff (d,`
	`  input d;`
	`  input clk;`
	`  output reg q;`
	`  //Begin included file: vlogic`
	`  always @ (posedge clk)`
	`  q <= d;`
	`  //End included file: vlogic`
	`endmodule`

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

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)+
};
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

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.

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	-

A signal can have its width set by assigning a field associated with an enum. The width is inferred from the enum width.

```
csl_enum e {
```

```
a = 0
};

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

// optional width
e() {
setwidth(7);
}

There should be an optional width for enums

1.1.3 CSL Bitrange object

contains a range w/an upper & lower index;

A bitrange object ~~holds the width of a signal or port and gets attached to it (a default bitrange exists but it can be overriden).~~ 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);
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

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 (eg. 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);
csl_bitrange br2(1,0);
csl_bitrange br3(br1);
csl_unit u{
  csl_port p1(input,  br1),
           p2(output, br2),
           p3(output, br3);
  u(){}
};
```

// copy constructor

blue text is difficult to read in printed doc.

The resulting Verilog code is:

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

```
  output [3:0] p3;
endmodule
```

Multidimensional bitranges can only be associated to signal objects as ports don't support multidimensional widths. The declaration syntax for a multidimensional bitrange is shown below:

```
csl_multi_dim_bitrange bitrange_name(constructor_parameters);
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

The constructor parameters can be a single numeric expression for setting the number of dimensions or an identifier that can be a numeric value (eg. const int) or the name of another multidimensional bitrange object (copy constructor - the properties of the copied object are replicated under a new object with the specified name):

```
constructor_parameters ::= numeric_expression
                          |multidimensional_bitrange_object_name
```

A short example is provided below

```
csl_multi_dim_bitrange mbr1(3);
csl_multi_dim_bitrange mbr2(mbr1);
csl_unit u{
  csl_signal s1(mbr1), s3(mbr2);
  u(){}
};
```

Note that in the example above, the widths for each dimension have to be manually set using the appropriate commands.

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	-

TABLE 1.6 CSL bitrange declaration in other CSL classes

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

1.1.3.1.1 Places where bitrange methods are used

method	cnst self br	cnst other br	gbl inst self br	cnst inst other br	field	signal	sig_gr	port	ifc	unit	rf	fifo
set_msb	X											
set_lsb	X											
get_width	X	X	X	X	X	X	X	X	X	X	X	X
set_offset		X		X	X							
get_offset		X		X	X							

cnst = constructor;
inst = instance;
cnst self = the call is made on the object in the object's constructor;
inst self gbl = the call is made on an instance of the object in the global scope;
inst self other = the call is made on an instance of the object in the global scope;

1.1.3.1.2 Multidimension bitrange commands

The following commands are specific to multidimension bitranges

Multidimension bitrange commands
`object_multi_dim.set_dim_bitrange(num_expr,br_obj_name);`
`object_multi_dim.set_dim_range(num_expr,num_expr,num_expr);`

1.1.4 CSL Field

Fields

Fields are named bitranges, because they can be associated with an enum item. Fields can be the full bit range of a signal or part of that bitrange. A field behaves like a bitrange with following differences:

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

The field declaration can be done in two ways depending on the type of the field. 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);
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

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 deeclaration rules

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	-

TABLE 1.7 CSL non-hierarchical field declaration rules

CSL class	Can be declared in
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 filed_name {
  (objects declarations/instantiations)+
  field_name(){
    (field methods calls)+
  }
};
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.:

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	-
CSL Fifo	-
CSL Memory Map	-

TABLE 1.8 CSL hierarchical field definition rules

CSL class	Can be defined in
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.

AB → why only base?

```
csl_enum e {
a = 0
};

csl_field f {
```

```
f () {
   set_enum(e); // this should set the width automatically
                        // set_width should be optional-is it?
}
};
```

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

NOTE:Above commands need a proper description and Xref

The following commands apply to non-hierarchical fields only:

CSL Field non-hierarchical only commands
`[field_or_field_instance.]set_value(num_expr);`
`[field_name.]set_enum(enum_name);`
`field_instance_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
`set_field_position(field_name, numeric_expression);`
`set_next(field_name_left, field_name_right);`
`set_previous(field_name_right, field_name_left);`

1.1.5 CSL Parameter

NOTE:NEEDS TO BE ADAPTED TO DOCS FORMAT & EXPANDED PROPERLY

1.1.5.1 Syntax for declaration:

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

Examples:

```
csl_parameter x(1);
csl_parameter y(2'b1);
csl_parameter z(2,5); //width is set to 5 bit (5'b00010)
```

Parameters can be declared in:

- units
- interfaces
- signal groups
- register class
- register file ~~class~~
- fifo ~~class~~
- memory ~~class~~

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.

```
eg. csl_parameter x(3);  //x is 32 bit
```

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

```
eg. csl_parameter x(4'd3); //x is 4 bit
```

if (param.width < size(param.#)
error

1.1.5.2 Syntax for overriding parameter values:

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

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 remaing 2 with their declared
values.
```

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.

1.1.5.2.2 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 x and z with values 7 and 8 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 oppposed 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
```

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
- register class
- register file class
- fifo class
- memory class

1.1.6 Prefixes and suffixes

> **NOTE: Add to future:**
> **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 supporting 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: If an interface ifc contains a port x and the interface is instantiated by the name ifc0, the generated port name will be:

```
ifc0x
```

1.1.6.1 Syntax for set_prefix() method:

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

Note that 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.

Once a prefix is set, it will override the default prefix set by the compiler. Thus in the example given above:If an interface ifc contains a port x and the interface is instantiated by the name ifc0 and:

- we call `ifc0.set_prefix("ixx_");` the generated code will be ixx_x
- we call `set_prefix("ixx_");` inside the interface constructor the generated code will be ixx_x on all instances of that interface unless a set_prefix() method called on an instance does not override this prefix.

1.1.6.2 No prefix:

Sometimes it may be required that there is no prefix in the generated code. 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 ifcx
With no_prefix()/set_prefix("") on the ouput name for port x should be x.

Hierarchical interfaces:
Interfaces can be hierarchical. Example: ifc1_.ifc2_.ifc3_.x (ifc1_ifc2_ifc3_x)

If ifc3 has no_prefix()/set_prefix("") on,
than the output for port x should look like: ifc1_ifc2_x

If ifc2 has no_prefix()/set_prefix("") on,
than the output for port x should look like: ifc1_ifc3_x

1.1.6.2.1 Syntax for set_suffix() method

```
[ifc/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 sufixing done by the compiler.

Also, if the suffixing method is called as in:
```
set_suffix("");
```
A warning will be printed because the prefix supplied is empty and nothing will happen (redundant command).

1.1.7 Operators

NOTE:Other operators need to be covered in here too

The following operators are supported in CSL

TABLE 1.10

1.1.7.1 Concatenation operator
1.1.7.2 Replication operator

1.1.7.1 Concatenation operator

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

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 these should have all contained elements 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 }
```

Same goes for signal groups.

NOTE:Add note about generate_individual_rtl_signals() for signal groups

Where can concatenation be used:

- assignment:

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

- connect statement: ——— ww about other functions

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

- formal to actual: actual

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

1.1.7.2 Replication operator

(also reffered to as as multi concatenation)

Syntax:

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

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

Replications can be used in:

- assignment:

```
x = {2{a}};  (only RHS)
This is NOT legal {2{a}} = x;
```

- connect statement:

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

- formal to actual: ~~formal~~

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

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

1.1.8 Pattern generator

The pattern generator is a functionality implemented at preprocessor level and it's used to expand user code into multiple lines of code according to a specification syntax.
Example:

```
iob  iob[[0---2]] (.in(x\1\));
```

expands into:

```
iob  iob0 (.in(x0));
iob  iob1 (.in(x1));
iob  iob2 (.in(x2));
```

The pattern generator can modify one line at a time or an entire area of code.

NOTE:this may need an additional syntax to mark the area to be expanded

Example:

```
csl_register reg[[2---4]]{
 reg\1\(){
   set_type(register);
   set_width(\1\);
 }
};
```

expands into:

```
csl_register reg2{
 reg2(){
   set_type(register);
   set_width(2);
 }
};

csl_register reg3{
 reg3(){
   set_type(register);
   set_width(3);
 }
};
```

```
csl_register reg4{
 reg4(){
   set_type(register);
   set_width(4);
 }
};
```

Syntax:
The following are the specifiers for the pattern generator:

```
[[ pattern_specifier ]]
```

DESCRIPTION :
this generates a set of values according to the type of the pattern specifier. The type of the pattern specifier can be:

- range_pattern: This can be a numeric range (eg. 0---9 generates 0 to 9) or a character range (eg. a---z generates a to z)
- list_pattern: this can be a selection (eg. a,m,z will generate a, m and z)

The pattern specifier creates a backreference that can be called later in the code. Thus, the first pattern specifier creates backreference 1, the second creates backreference 2 and so on.. the backreferences can be later called using a specific syntax detailed below:

backreference_number
Syntax:

```
\backreference_number\
```

Example:

```
\1\
```

this will insert generated code according to pattern specifier linked to backreference 1 as in the examples:

- with range pattern

```
iob  iob[[0---2]] (.in(x\1\));
```

which expands into:

```
iob  iob0 (.in(x0));
iob  iob1 (.in(x1));
iob  iob2 (.in(x2));
```

- with list_pattern

```
abc_[[a,c,e,g]] = b_\1\;
```

generates

```
abc_a = b_a;
```

```
abc_c = b_c;
abc_e = b_e;
abc_g = b_g;
```

backreference_number with increment amount
Syntax:

\backreference_numberiincrement_amount

Example:

```
\1i4\
```

this will insert generated code according to pattern specifier linked to backreference 1 incremented at each iteration with a value of 4

```
abc[[0---3]] = xyz\1i4\;
```

generates

```
abc0 = xyz0;
abc1 = xyz4;
abc2 = xyz8;
abc3 = xyz12;
```

Note: this does not work for ASCII character ranges in pattern specifier

reversed backreference_number
Syntax:

\back_reference_numberr

Example:

```
\1r\
```

counts backwards from the end of the range or list to the beginning of the range or list

- with range pattern

```
abc[[0---3]] = xyz\1r\;
```

generates

```
abc0 = xyz3;
abc1 = xyz2;
abc2 = xyz1;
abc3 = xyz0;
```

- with list pattern

```
abc_[[a,c,e,g]] = b_\1r\;
```

generates

```
abc_a = b_g;
abc_c = b_e
abc_e = b_c;
```

```
abc_g = b_a;
```

ignore sequence
Syntax:

```
(?#ignore_sequence)
```

Everything between the (?# and) is the ignore sequence. The pattern generator will not affect the ignore sequence, leaving it as it is.
Example:

```
iob  iob[[(a---d($#_user)) , (e---g($#_driver))]][[0---
9]](.in(x\1\\2\));
```

generates ?

NOTE:Uncertain & Unknown & to be sorted out

Common commands:

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

csl_field fld (0,3);
|| . (4,7);

fld can hold the range 0-15 but is restricted to the range 4-7.

1.2 CSL Language Commands

1.2.1 Field

`csl_field field_name;`

DESCRIPTION :

Creates a field named `field_name`.

[*CSL Language Commands Summary*]

EXAMPLE :

In this example is created a field named *fd1* with the width 5.

CSL CODE

```
csl_field fd1{
fd1(){
set_width(5);
}
};
```

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 :

Creates a field 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 :

Creates a 4 bits field named *fd1*.

CSL CODE

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

VERILOG CODE

```
//
```

csl_field ∴ `set_enum(enum_name);`

DESCRIPTION :

//

EXAMPLE :

Sets the enum named *eee* for a field named *fd*.

CSL CODE

```
csl_enum eee{
  ADD,
  SUB
    };
csl_field fd{
  fd(){
    set_width(4);
    set_enum( eee);
  }
};
```

Verilog code

csl_field::

`set_enum_item(identifier);`

DESCRIPTION :

//

EXAMPLE :

//

CSL CODE

```
csl_enum eee{
  ADD,
  SUB
    };
csl_field fd{
  fd(){
    set_width(4);
    set_enum_item( SUB);
  }
};
```

Verilog code

```
set_value(numeric expression);
```

DESCRIPTION :

//

EXAMPLE :

//

CSL CODE

```
csl_field fd{
  fd(){
    set_value(5);
  }
};
```

Verilog code

```
generate_decoder(csl_unit);
```

DESCRIPTION :

//

EXAMPLE :

//

CSL CODE

```
csl_unit u1{
  csl_signal s1,s2,s3;
  csl_port p1(input), p2(output);
u1(){}
};
csl_field fd{
  fd(){
    generate_decoder(u1);
  }
};
```

VERILOG CODE

1.2.2 Bitrange

A bit range is used to declare the upper index (MSB) and the lower index (LSB) of a signal. Moreover, a bit range is also used in expressions to index into 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*: param width - width of the bitrange, it will transform into a range[n-1:0];

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

EXAMPLE :

Creates a bitrange named *br* which is used to set the range for a signal *s1* and a port *p1*.

CSL CODE

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

VERILOG CODE

```
module u(p1);
  input [3:0] p1;
  wire [3:0] 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 one will get a compile time error).

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

EXAMPLE :

It is created a bitrange *br2* using 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 :

In this example was created a bitrange *br2* , using another bitrange *br1* like a parameter.

CSL CODE

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

VERILOG CODE

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 : Changes the width of a bitrange br1 using the *set_offset* method.

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 [7:0] p1;
  output [7:0] p2;
endmodule
```

csl_multi_dim_bitrange *obj_name*(num_expr);

DESCRIPTION :

Creates a multidimensional bitrange object called *obj_name*: param number of dimensions;

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

EXAMPLE :

~~It was created~~ is created a multidimensional bitrange *mbr1* with 3 dimensions used to set the width of a signal *s1*.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
csl_unit u{
  csl_signal s1(mbr1);
  u(){}
};
```

VERILOG CODE

//

csl_multi_dim_bitrange *obj_name*(*multi_dim_br_obj*);

DESCRIPTION :

Creates a multidimensional bitrange object called *obj_name*: param multi dimensional br object - copy constructor;

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

EXAMPLE :

In this example *mbr2* will has the same number of dimensions like *mbr1*.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
csl_multi_dim_bitrange mbr2(mbr1);
csl_unit u{
  csl_signal s1(mbr1), s2(mbr2);
  u(){}
};
```

VERILOG CODE

object_multi_dim.**set_dim_width**(*numeric_expr*,*numeric_expr*);

DESCRIPTION :

Set the with of *object_multi_dim* :

param1: dimension index (dimension index start from 0 and go up to num_of_dims-1);
param2: width of the dimension

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

EXAMPLE :

In this example ~~it was created a~~ multidimensional bitrange *mbr1* with 3 dimensions. The width of each dimension is set using the *set_dim_width* method.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
mbr1.set_dim_width(0,2);
mbr1.set_dim_width(1,4);
mbr1.set_dim_width(2,8);
csl_unit u{
  csl_signal s1(mbr1);
  u(){}
};
```

VERILOG CODE

object_multi_dim.**set_dim_bitrange**(num_expr,br_obj_name);

DESCRIPTION :

Set the bitrange of *object_multi_dim* :

param1: dimension index (dimension index start from 0 and go up to num_of_dims-1);

param2: imple a csl_bitrange to be set as a dimension to the multi dim bitrange;

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *set_dim_bitrange()* method to set the width for the *mbr2* dimensions like *mbr1* dimensions.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
mbr1.set_dim_width(0,2);
mbr1.set_dim_width(1,4);
mbr1.set_dim_width(2,8);
csl_multi_dim_bitrange mbr2(3);
mbr2.set_dim_bitrange(0, mbr1);
mbr2.set_dim_bitrange(1, mbr1);
mbr2.set_dim_bitrange(2, mbr1);
csl_unit u{
  csl_signal s1(mbr1), s2(mbr2);
  u(){}
};
```

VERILOG CODE

object_multi_dim.**set_dim_range**(num_expr,num_expr,num_expr);

DESCRIPTION :

Set the range of *object_multi_dim* :

param1: dimension index (dimension index start from 0 and go up to num_of_dims-1);
param 2 : lower_index of the dimension
param 3 : upper_index of the dimension

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

EXAMPLE :

In this example ~~it is created~~ a multidimensional bitrange *mbr1* with 3 dimensions. ~~We used the~~ *set_dim_range()* method to set the range for each dimension.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
mbr1.set_dim_range(0,0,1);
mbr1.set_dim_range(1,0,3);
mbr1.set_dim_range(2,0,7);
csl_unit u{
  csl_signal s1(mbr1);
  u(){}
};
```

VERILOG CODE

object_multi_dim.**set_dim_offset**(num_expr,num_expr);

DESCRIPTION :

Set the range of *object_multi_dim* :

param1: dimension index (dimension index start from 0 and go up to num_of_dims-1);

param 2 : offset of the dimension

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

EXAMPLE :

In this example for the multidimensional bitrange *mbr1*, the width of the dimensions are changed, using *set_dim_offset* method.

CSL CODE

```
csl_multi_dim_bitrange mbr1(3);
mbr1.set_dim_width(0,2);
mbr1.set_dim_width(1,4);
mbr1.set_dim_width(2,8);

mbr1.set_dim_offset(0,4);
mbr1.set_dim_offset(1,8);
mbr1.set_dim_offset(2,2);

csl_unit u{
  csl_signal s1(mbr1);
  u(){}
};
```

VERILOG CODE

1.2.3 CSL Address Range

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

DESCRIPTION :

Sets the lower and upper bounds of the values that can be associated to a field.
to the field;

[*CSL Language Commands Summary*]

EXAMPLE :

Adds an allowed range to a field *fd1*.

CSL CODE

```
csl_field fd1(4);
 fd1.add_allowed_range(0,8);
```

VERILOG CODE

which is 4 bits wide to can contain the range 0-15. The " is restricted to 0-8.

Non-hierarchical:

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

DESCRIPTION :

Instantiate a field:

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

param enum_or_enum_item - the enum or the enum item will beassociated to the field;

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

EXAMPLE :

Creates a field named *fd1* with the width 4.

CSL CODE

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

Verilog

Duplicate of page 25

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

DESCRIPTION :

Instantiate a field with lower bit and upper bit :

param lower - lower bound of the field;
param upper - 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 :

Creates a field named *fd2* with the width 8, using lower bound and upper bound.

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 :

Creates a field named *fd* with the width set by a bitrange *br*.

CSL CODE

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

csl_field:: `set_width(num_expr);`

DESCRIPTION :

set the width of field to num_expr.

EXAMPLE :

//

CSL CODE

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

csl_field::set_bitrange(num_expr);

DESCRIPTION :

// set the bitrange for the field

EXAMPLE :

//

CSL CODE

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

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

DESCRIPTION :

Set the value to be added to both lower and upper index of the field.

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

EXAMPLE :

Set the offset for a field *fd1*.

CSL CODE

```
csl_field fd1{
  fd1(){
    set_offset(4);
  }};
```

csl_field::

```
[field_name.]set_enum(enum_name);
```

DESCRIPTION :

Associates an enum to the field object.

[*CSL Language Commands Summary*]

EXAMPLE :

In this example it is associated an enum to the field *fd.*

CSL CODE

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

```
field_instance_name.set_enum_item(enum_item_name);
```

DESCRIPTION :

Associates an enum_item to the field object;

[*CSL Language Commands Summary*]

EXAMPLE :

In this example it is associated an enum item to the field *fd*.

CSL CODE

```
csl_enum enum1{s1, s2, s3};
csl_field fd{
  fd(){
set_enum_item(s2);
  }};
```

csl_field::
[field_or_field_instance.]set_value(num_expr);

DESCRIPTION :

Associates an numeric value to the field object.

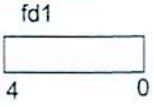
[*CSL Language Commands Summary*]

EXAMPLE :

In this example it is associated a value to the field *fd1*.

'is associated with

FIGURE 1.1



CSL CODE

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

csl_field::

`set_field_position(field_name, numeric_expression);`

DESCRIPTION :

Set the absolute position of a field; If set_filed_position is not set then the ~~top~~ first ~~most~~ instance is the left most field.

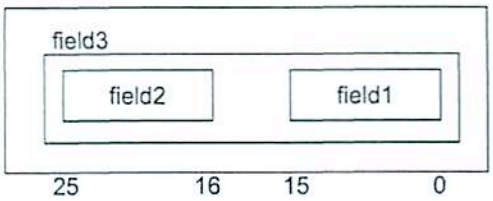
instantiated

[*CSL Language Commands Summary*]

EXAMPLE :

//

FIGURE 1.2



CSL CODE

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

Hierarchical:

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 changes they will remain adjacent to each other and there are no offsets changes.

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

EXAMPLE :

//

CSL CODE

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

csl_field::

`set_previous(field_name_right, field_name_left);`

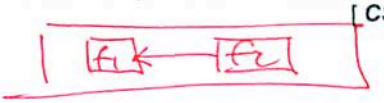
DESCRIPTION :

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 changes they will remain adjacent to each other and there are no offsets change;

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

EXAMPLE :

//



CSL CODE

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

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

DESCRIPTION :

~~Creates~~ Declares a new CSL enum.

[*CSL Language Commands Summary*]

EXAMPLE :

//

CSL CODE

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

opcodes() {
  set-width(4);
}

2.1 CSL Language rules

2.1.1 Design:

Declaration/Definition:

- csl_bitrange
- csl_enum
- csl_event
- csl_field
- csl_fifo
- csl_interface
- csl_isa_field
- csl_isa_element
- csl_list
- csl_memory
- csl_memory_map
- csl_memory_map_page
- csl_multi_dim_bitrange
- csl_port
- csl_register
- csl_register_file
- csl_signal
- csl_signal_group
- csl_signal_pattern_generator
- csl_state_data
- csl_testbench
- csl_unit
- csl_vector

csl_bitrange:
declaration/definition: none
instantiate: none

csl_enum:
declaration/definition: none
instantiate: none

csl_event:
declaration/definition: none
instantiate: none

where class can be declared?
what does this mean add explanation.
where it can be instantiated
allowed
long

csl_field:
declaration/definition: csl_field(not as scope holder)
csl_list
instantiate: csl_field

csl_fifo:
declaration/definition: csl_list
instantiate: none

csl_interface:
declaration/definition: csl_bitrange
csl_field(not as scope holder)
csl_list
csl_multi_dim_bitrange
csl_port
instantiate: csl_interface

csl_isa_field:
declaration/definition: csl_isa_field(not as scope holder)
csl_list
instantiate: csl_isa_field

csl_isa_element:
declaration/definition: csl_isa_field(not as scope holder)
csl_list
instantiate: csl_isa_field

csl_list:
declaration/definition: none
instantiate: none

csl_memory:
declaration/definition: csl_list
instantiate: none

csl_memory_map:
declaration/definition: csl_list
instantiate: csl_memory_map_page

csl_memory_map_page:
declaration/definition: csl_list
instantiate: csl_memory_map_page

csl_multi_dim_bitrange:
declaration/definition: none
instantiate: none

csl_port:
declaration/definition: none
instantiate: none

csl_register:
declaration/definition: csl_field(not as scope holder)
csl_list
instantiate: csl_field

csl_register_file:
declaration/definition: csl_list
instantiate: none

csl_signal:
declaration/definition: none
instantiate: none

csl_signal_group:
declaration/definition: csl_bitrange
csl_field(not as scope holder)
csl_list
csl_multi_dim_bitrange
csl_signal

instantiate: csl_signal_group

csl_signal_pattern_generator:
declaration/definition: none
instantiate: none

csl_state_data:
declaration/definition: none
instantiate: none

csl_testbench:
declaration/definition: csl_bitrange
csl_field(not as scope holder)
csl_list
csl_multi_dim_bitrange
csl_signal
csl_port
csl_vector
instantiate: csl_interface
csl_signal_group
csl_state_data
csl_unit

csl_unit:
declaration/definition: csl_bitrange
csl_field(not as scope holder)
csl_list
csl_multi_dim_bitrange
csl_signal
csl_port
instantiate: csl_field
csl_fifo
csl_interface
csl_memory
csl_register
csl_register_file
csl_signal_group
csl_unit

csl_vector:
declaration/definition: csl_port
csl_signal
instantiate: csl_interface
csl_signal
csl_state_data