CHAPTER 1 CSL Language common features

TABLE 1.1 Chapter Outline

1.1 CSL Language Commands Summary

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 1.2). 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 1.2 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	-

TABLE 1.2 CSL include directive used in other CSL classes

CSL class	accepts calling include directive
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 1.3 CSL Include directive

Input CSL code	Generated Verilog code

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

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

1.1.2 Constants

CSL allows for definition of constant integers just like in the C++ programming language; The syntax for declaring a const int is also borrowed from the aforementioned language and is shown in the example below:

```
const int const_name = numeric_expression;
```

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 const int definition, `numeric_expression` can be anything from a plain number to an expression (unary, binary, ternary or a function call) that results in a number.

1.1.2.1 Const int usage and rules

Constants can be declared in the global scope (outside the scope of any class) or inside certain CSL classes as can be seen in Table 1.4:

TABLE 1.4 CSL const int declaration in other CSL classes

CSL class	Accepts const int declaration
CSL Unit	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	-

1.1.3 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.

1.1.3.1 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
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 an enum to it. The width is inferred from the enum width.

```
csl_enum e {
a = 0
};

csl_unit top {
csl_signal s(wire,32);

top() {
  s.set_bitrange(f); // this shoulsd set the width automatically
  // set_width should be optional-is it?
}
};
```

1.1.4 CSL Bitrange object

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(){}
};
```

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 multidimnesional 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), s2(mbr2);
  u(){}
};
```

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

1.1.4.1 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	-
CSL Field	-

1.1.4.1.1 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_upper_index	x											
get_upper_index	x	x	x	x		x	x	x	x	x	x	x
set_lower_index	x											
get_lower_index	x	x	x	x		x	x	x	x	x	x	x
get_width	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;

1.1.4.1.2 Single dimension bitrange commands

The following commands are specific to single dimension bitranges

Single dimension bitrange commands

```
bitrange_name.set_upper_index(numeric_expression);
bitrange_name.get_upper_index(numeric_expression);
bitrange_name.set_lower_index(numeric_expression);
bitrange_name.get_lower_index(numeric_expression);
int bitrange_name.get_width();
bitrange_name.set_offset(numeric_expression);
bitrange_name.get_offset();
```

1.1.4.1.3 Multidimension bitrange commands

The following commands are specific to multidimension bitranges

Multidimension bitrange commands
`bitrange_name.set_dim_width(numeric_expr,numeric_expr);`
`bitrange_name.get_dim_width(numeric_expr);`
`object_multi_dim.set_dim_bitrange(num_expr,br_obj_name);`
`object_multi_dim.get_dim_bitrange(num_expr);`
`object_multi_dim.set_dim_range(num_expr,num_expr,num_expr);`
`object_multi_dim.get_dim_lower(num_expr);`
`object_multi_dim.get_dim_upper(num_expr);`
`object_multi_dim.set_dim_offset(num_expr,num_expr);`
`object_multi_dim.get_dim_offset(num_expr);`

1.1.5 CSL Field

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 two main differences:

- fileds also hold a name
- fields can have a hierarchical structure

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_or_enum_item]
                          |num_expr, num_expr [, enum_or_enum_item]
                          |bitrange_name [, enum_or_enum_item]
```

1.1.5.1 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	-
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	-

TABLE 1.8 CSL hierarchical field definition rules

CSL class	Can be defined in
CSL Signal Group	-
CSL Interface	-
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	-

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	-

TABLE 1.9 CSL hierarchical field instantiation rules

CSL class	Can be instantiated in
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-is it?
}
};
```

1.1.5.2 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

```
[field_name.]set_upper_index(num_expr);
[field_name.]set_lower_index(num_expr);
[field_name.]set_offset(num_expr);
[field_name.]set_enum(enum_name);
field_or_field_instance_name.set_enum_item(enum_item_name);
int field_name.get_width();
int field_name.get_upper_index();
int field_name.get_lower_index();
[field_name.]get_offset();
field_name.get_enum();
```

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_name.]set_value(num_expr);
field_name.get_value();

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);

NOTE:Uncertain & Unknown & to be sorted out

Common commands:

```
signal_object_name.generate_decoder(decoder_unit_name);
ir.instr.field.create_signal(signal_name);
ir.inst_fmt_name.field_name.connect(signal_name);
field_obj_name.add_allowed_range(num_expr,num_expr);
```

List

```
csl_list list_object_name;
csl_list list_object_name({identifier|object_of_the same_type});
list_object_name.set_method(parameters);
```

CSL Address Range:

```
set_width(numeric expression);
set_range(numeric expression, numeric expression);
set_offset(numeric expression);set_type(undefined | reserved | valid |
illegal);
```

1.2 CSL Language Commands

```
csl_list list_object_name;
```

DESCRIPTION :

Creates a list named `list_object_name;`

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

EXAMPLE :

In this example are created two lists. The first list in named *list1* and the second list is a list of signals named *list2* .

CSL CODE

```
csl_list  list1;
csl_unit u{
csl_signal s1,s2,s3;
csl_list list2( s2,s2,s3);
 u(){}
};
```

VERILOG CODE

```
module u();
  wire s1;
  wire s2;
  wire s3;
endmodule
```

```
csl_list list_object_name({identifier|object_of_the same_type});
```

DESCRIPTION :

Creates a list named `list_object_name` by specify the the objects of the same type.

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

EXAMPLE :

It was created a list named *list2* which contains three signals.

CSL CODE

```
csl_unit u{
csl_signal s1,s2,s3;
csl_list list2({ s2,s2,s3});
u(){}
};
```

VERILOG CODE

```
module u();
  wire s1;
  wire s2;
  wire s3;
endmodule
```

```
list_object_name.set_method(parameters);
```

DESCRIPTION :

Apply parameter to all members of the list `list_object_name.`

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

EXAMPLE :

In this example is applied a parameter to the list *list1.*

CSL CODE

```
csl_unit u{
csl_signal s1,s2,s3;
csl_list list1 (s1,s2,s3);
u(){
list1.set_method(4);
}
};
```

VERILOG CODE

```
//
```

1.2.1 Memory elements

For multiple reads and writes in same time

1.2.2 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

```
//
```

set_enum(enum_name**);**

DESCRIPTION :

//

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

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

set_enum_item(identifier**);**

DESCRIPTION :

//

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

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 :

//

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

EXAMPLE :

//

CSL CODE

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

Verilog code

```
generate_decoder(csl_unit some_unit);
```

DESCRIPTION :

//

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

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

1.2.3 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]).

bitrange_name.**set_upper_index(***numeric_expression***);**

DESCRIPTION :

//

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

EXAMPLE :

small description of the example.

CSL CODE

```
csl_bitrange br(4);
br.set_upper_index(3);
```

VERILOG CODE

bitrange_name.**get_upper_index(***numeric_expression***);**

DESCRIPTION :

//

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

EXAMPLE :

small description of the example.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(4);
br1.set_upper_index(3);
br1.set_lower_index(0);
br2.set_upper_index(br1.get_upper_index());
br2.set_lower_index(br1.get_lower_index());
```

VERILOG CODE

bitrange_name.**set_lower_index(***numeric_expression***);**

DESCRIPTION :

//

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

EXAMPLE :

small description of the example.

CSL CODE

```
csl_bitrange br(4);
br.set_upper_index(3);
br.set_lower_index(0);
```

VERILOG CODE

bitrange_name.**get_lower_index(***numeric_expression***);**

DESCRIPTION :

//

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

EXAMPLE :

small description of the example.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(4);
br1.set_upper_index(3);
br1.set_lower_index(0);
br2.set_upper_index(br1.get_upper_index());
br2.set_lower_index(br1.get_lower_index());
```

VERILOG CODE

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

int *bitrange_name*.**get_width();**

DESCRIPTION :

Returns the width of a bitrange object.

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

EXAMPLE :

Sets the width of the bitrange *br2* using the width of bitrange *br1*.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(br1.get_width());
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 :

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
```

bitrange_name.**get_offset();**

DESCRIPTION :

Returns the offset of the bitrange.

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

EXAMPLE :

Sets the offset for *br2* using the offset of *br1*.

CSL CODE

```
csl_bitrange br1(4);
csl_bitrange br2(4);
br1.set_offset(8)
br2.set_offset(br1.get_offset());
csl_unit u{
  csl_port p1(input, br1), p2(output, br2);
  u(){}
};
```

VERILOG CODE

bitrange_name.**get_lower_index**();

DESCRIPTION :

Returns the lower index of the bitrange.

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

EXAMPLE :

Sets the range for bitrange *br2* using lower index and upper index from bitrange *br1*.

CSL CODE

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

VERILOG CODE

bitrange_name.**get_upper_index**();

DESCRIPTION :

Returns the upper index of the bitrange.

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

EXAMPLE :

Sets the range for bitrange *br2* using lower index and upper index from bitrange *br1*.

CSL CODE

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

VERILOG CODE

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 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

bitrange_name.**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

bitrange_name.**get_dim_width**(*numeric_expr*);

DESCRIPTION :

Returns the numeric expresion that represents the width of the specified dimension of *object_multi_dim* :

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

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *get_dim_width()* method to set the width for the *mbr2* 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_width(0,mbr1.get_dim_width());
mbr2.set_dim_width(1,mbr1.get_dim_width());
mbr2.set_dim_width(2,mbr1.get_dim_width());
csl_unit u{
  csl_signal s1(mbr1), s2(mbr2);
  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 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.**get_dim_bitrange**(num_expr);

DESCRIPTION :

Returns : the simple dimension bitrange associated with the specified dimension of *object_multi_dim* :

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

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *get_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.get_dim_bitrange(0));
mbr2.set_dim_bitrange(1, mbr1.get_dim_bitrange(1));
mbr2.set_dim_bitrange(2, mbr1.get_dim_bitrange(2));
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.**get_dim_lower**(num_expr);

DESCRIPTION :

Returns : numeric expresion that represents the lower index of the specified dimension of *object_multi_dim* :

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

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *get_dim_lower() and get_dim_upper()* methods to set the range for the *mbr2* dimensions like *mbr1* dimensions.

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_multi_dim_bitrange mbr2(3);
mbr2.set_dim_range(0, mbr1.get_dim_lower(0), mbr1.get_dim_upper(0));
mbr2.set_dim_range(1, mbr1.get_dim_lower(1), mbr1.get_dim_upper(1));
mbr2.set_dim_range(2, mbr1.get_dim_lower(2), mbr1.get_dim_upper(2));
csl_unit u{
  csl_signal s1(mbr1), s2(mbr2);
  u(){}
};
```

VERILOG CODE

object_multi_dim.**get_dim_upper**(num_expr);

DESCRIPTION :

Returns : numeric expresion that represents the upper index of the specified dimension of *object_multi_dim* :

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

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *get_dim_lower() and get_dim_upper()* methods to set the range for the *mbr2* dimensions like *mbr1* dimensions.

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_multi_dim_bitrange mbr2(3);
mbr2.set_dim_range(0, mbr1.get_dim_lower(0), mbr1.get_dim_upper(0));
mbr2.set_dim_range(1, mbr1.get_dim_lower(1), mbr1.get_dim_upper(1));
mbr2.set_dim_range(2, mbr1.get_dim_lower(2), mbr1.get_dim_upper(2));
csl_unit u{
  csl_signal s1(mbr1), s2(mbr2);
  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

object_multi_dim.**get_dim_offset**(num_expr);

DESCRIPTION :

Returns : numeric expresion that represents the offset of thespecified dimension of *object_multi_dim* :

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

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

EXAMPLE :

In this example it is created two multidimensional bitrange *mbr1* and *mbr2* with 3 dimensions. We used the *get_dim_offset()* methods to make the same changes like *mbr1* for *mbr2* 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);

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

csl_multi_dim_bitrange mbr2(3);
mbr2.set_dim_width(0,2);
mbr2.set_dim_width(1,4);
mbr2.set_dim_width(2,8);

mbr2.set_dim_offset(0,mbr1.get_dim_offset(0));
mbr2.set_dim_offset(1,mbr1.get_dim_offset(1));
mbr2.set_dim_offset(2,mbr1.get_dim_offset(2));

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

VERILOG CODE

1.2.4 CSL Address Range

set_width(numeric expression);

DESCRIPTION :

Set the width to a field, fifo,isa_element, isa_field, register_file, register, memory .

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

EXAMPLE :

//

CSL CODE

```
csl_register reg1{
  reg1(){
    set_width(32);
        }};
```

```
set_range(numeric expression, numeric expression);
```

DESCRIPTION :

Sets the range to a field, fifo,isa_element, isa_field, register_file, register, memory .

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

EXAMPLE :

//

CSL CODE

```
csl_register reg1{
  reg1(){
    set_range(0, 128);
      }};
```

```
set_offset(numeric expression);
```

DESCRIPTION :

Sets the offset to a csl_field or csl_isa_field .

[*CSL Language Commands Summary*]

EXAMPLE :

//

CSL CODE

```
csl_field fd{
  fd(){
     set_offset(32);
  }
};
```

```
set_type(undefined | reserved | valid | illegal);
```

DESCRIPTION :

//

[*CSL Language Commands Summary*]

EXAMPLE :

//

CSL CODE

```
csl_memory_map_page mmap{
  mmap(){
    add_address_range(0,128);
    set_data_word_width(16);
  }
};
csl_memory_map mp{
  mmap mmap;
  mp(){
    set_data_word_width(16);
    set_type(reserved);
  }
};
```

1.2.5 CSL field

All :

```
signal_object_name.generate_decoder(decoder_unit_name);
```

DESCRIPTION :

Will generate a decoder in verilog. If the field is hierarchical it will generate a decoder for every child in the hierarchy.

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

EXAMPLE :

//

CSL CODE

```
csl_unit u{
  csl_signal sig1;
  u(){
    sig1.generate_decoder(dec1);
  }};
```

VERILOG CODE

*ir.instr.field.***create_signal(***signal_name***);**

DESCRIPTION :

Creates a signal a field width connected to the ir[field.get_up(): field.get_low()]
Creates a signal with the field that is connected to ir[field.lower:field.upper];

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

EXAMPLE :

//

CSL CODE

Verilog code

ir.inst_fmt_name.field_name.**connect(***signal_name***);**

DESCRIPTION :

Creates a signal that is connected to the field in ir.inst_fmt_name.field_name
Will search for *signal_name* signal and then will connect with ir[field.upper:field.lowers];

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

EXAMPLE :

//

CSL CODE

```
csl_register ir;
ir.add_isa(isa_name);
```

ir.inst_fmt_name.field_name**.connect(***signal_name***);**

```
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

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);
```

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);
```

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

DESCRIPTION :

//

[*CSL Language Commands Summary*]

EXAMPLE :

//

CSL CODE

```
csl_field fd1{
  fd1(){
    set_upper_index(15);
  }
};
```

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

DESCRIPTION :

//

[*CSL Language Commands Summary*]

EXAMPLE :

//

CSL CODE

```
csl_field fd1{
  fd1(){
    set_lower_index(0);
  }
};
```

```
[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);
  }};
```

```
[field_name.]get_offset();
```

DESCRIPTION :

Returns the offset of the field.

[*CSL Language Commands Summary*]

EXAMPLE :

set the offset for the field *fd2* using the offset from *fd1.*

CSL CODE

```
csl_field fd1{
  fd1(){
fd1.set_offset(8);
  }};
csl_field fd2{
  fd1 fd1;
  fd2(){
set_offset(fd1.get_offset());
  }};
```

```
int field_name.get_lower_index();
```

DESCRIPTION :

Returns the lower index of the field.

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

EXAMPLE :

Creates a field *fd2* with the *fd1* range , using *get_lower()* and *get_upper_index()* methods .

CSL CODE

```
csl_field fd1(4);
csl_field fd2(fd1.get_lower_index(), fd1.get_upper_index());
```

```
int field_name.get_upper_index();
```

DESCRIPTION :

Returns the upper index of the field.

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

EXAMPLE :

Creates a field *fd2* with the *fd1* range , using *get_lower()* and *get_upper_index()* methods .

CSL CODE

```
csl_field fd1(4);
csl_field fd2(fd1.get_lower_index(), fd1.get_upper_index());
```

```
int field_name.get_width();
```

DESCRIPTION :

Returns the width of the field.

[*CSL Language Commands Summary*]

EXAMPLE :

Sets the width for the field *fd2* , using the *fd1* width.

CSL CODE

```
csl_field fd1(4);
csl_field fd2(fd1.get_width());
```

```
[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_or_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);
  }};
```

```
field_name.get_enum();
```

DESCRIPTION :

Returns the associated enum item of the field object;

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

EXAMPLE :

In this example it is associated an enum item to the field *fd0* using *get_enum()* method.

CSL CODE

```
csl_enum enum1{fd1, fd2, fd3};
csl_field fd{
  fd(){
    set_enum(enum1);
  }};
csl_field fd0{
fd fd;
  fd0(){
set_enum(fd.get_enum());
  }};
```

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

DESCRIPTION :

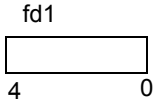
Associates an numeric value to the field object.

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

EXAMPLE :

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

FIGURE 1.1



CSL CODE

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

```
field_name.get_value();
```

DESCRIPTION :

Returns the associated numeric value of the field object;

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

EXAMPLE :

In this example it is associated a value to the field fd2 using the *get_value()* method.

CSL CODE

```
csl_field fd1{
  fd1(){
set_value(16);
  }};
csl_field fd2{
fd1 fd1;
fd2(){
set_value(fd1.get_value());
}};
```

Hierarchical:

```
set_field_position(field_name, numeric_expression);
```

DESCRIPTION :

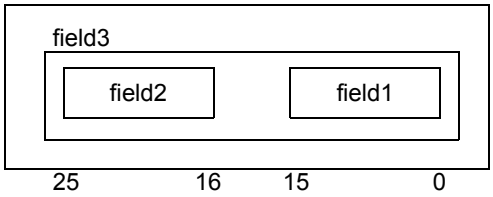
Set the absolute position of a field;

[***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);
}
};
```

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 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_next(f1,f2);
}
};
```

```
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 :

//

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

EXAMPLE :

//

CSL CODE

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

CHAPTER 1 CSL Testbench

TABLE 1.1 Chapter Outline

1.1 CSL Testbench Syntax and Command Summary

1.1.1 CSL Testbench class

A testbench is roughly a testing unit that is used to test other units (designs) which are instantiated in it. Such an instance of a design in a testbench is commonly refered to as a DUT (design under test). Testbenches are declared as classes and so behave as scope holders.

1.1.2 CSL Testbench class declaration

A CSL Testbench can only be declared in the global scope just like any other CSL class. The CSL Testbench class is declared as in the below example:

```
csl_testbench testbench_name {
  (objects declarations/instantiations)+
  testbench_name(){
    (testbench 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.

In the testbench class' scope DUTs are instantiated. A design under test can be a CSL unit or other predefined CSL classes as shown in Table 1.2

TABLE 1.2 Rules for instantiating objects in a testbench scope

CSL class	Is instantiated in CSL Testbench scope
CSL Unit	YES
CSL Testbench	-
CSL Vector	-
CSL State Data	-
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	-

1.1.3 CSL Testbench mandatory commands

The following command is mandatory to the CSL testbench class as it sets the clock signal that will be used throughout the testbench.

CSL Testbench mandatory commands

```
add_logic(clock,clock_name,period,time_base);
```

1.1.4 CSL Testbench usage and rules

Testbenches can only be declared in the global scope and cannot be instantiated anywhere as can also be seen from Table 1.3

TABLE 1.3 Vector usage rules

CSL class	Instantiates CSL Testbench
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	-

Testbenches are thus a particular case of top level unit but should not be confused with the design's top level unit (a CSL design can have a top level unit and a testbench at the same time).

1.1.5 CSL Testbench commands

The following commands apply to CSL tesbench classes.

CSL Testbench commands

```
set_testbench_verilog_filename(name);
add_logic(generate_waves, filename, wave_type, [list_of_scopes]);
add_logic(generate_report [,columns|rows]);
```

1.2 Testbench Commands

add_logic(clock,clock_name,period,time_base);

DESCRIPTION :

Sets the clock signal for the testbench. This will be used to create a clock generator that will output a clock signal on the *clock_name* signal, with the period given by the *period* parameter and the time base specified by *time_base*.

[***CSL Testbench Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port in_v(input,1), in_d(input, 8);
  csl_port out_v(output, 1), out_d(output, 8);
  csl_port clk_p(input);
   dut(){
clk_p.set_attr(clock);}
};
csl_vector stim{
  stim(){
    set_vc_header_comment("stimulus_vector");
    set_vc_output_filename("stim_data_out");
    set_unit_name ( dut );
    set_direction   ( input );
   }
};
csl_vector exp{
  exp(){
    set_vc_header_comment("expected_vector");
    set_vc_output_filename("exp_data_out");
    set_unit_name ( dut );
    set_direction   ( output );
  }
};
csl_testbench tb{
  csl_signal clk;
  dut dut_i;
    tb(){
  clk.set_attr(clock);
  add_logic(clock,clk,10,ps);
  }};
```

VERILOG CODE

```
//
```

set_testbench_verilog_filename(name);

DESCRIPTION :

The default name of the Verilog module and the filename for the testbench is the name of the CSL Testbench class. This command can be used to override the default testbench name with *name*.
Use csl_include to add an include file containing Verilog code to the Verilog testbench.

[***CSL Testbench Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port in_v(input,1), in_d(input, 8);
  csl_port out_v(output, 1), out_d(output, 8);
  csl_port clk_p (input);
dut(){
clk_p.set_attr(clock);}
};
csl_vector stim{
  stim(){
    set_vc_header_comment("stimulus_vector");
    set_vc_output_filename("stim_data_out");
    set_unit_name ( dut );
    set_direction ( input );
  }
};
csl_vector exp{
  exp(){
    set_vc_header_comment("expected_vector");
    set_vc_output_filename("exp_data_out");
    set_unit_name ( dut );
    set_direction   ( output );
}
};
csl_testbench tb{
  csl_signal clk;
  dut dut_i;
  tb(){
    clk.set_attr(clock);
    add_logic(clock,clk,10,ps);
    set_testbench_verilog_filename("testbench");
  }};
```

VERILOG CODE
//

```
add_logic(generate_waves, filename, wave_type, [list_of_scopes]);
```

DESCRIPTION :

Sets the wave generator to dump waves in an output file specified by the *filename* parameter. The wave format is is specified by the wave_type parameter as shown in Table 1.4. Default is to dump waves for every signal and port from the top design below. Optionally, a list of scopes can be passed in to limit the amount of waves to be dumped.

[***CSL Testbench Syntax and Command Summary***]

TABLE 1.4 Wave types

wave_type	generated wave format
fsdb	generates fsdb wave format
vcd	generates vcd wave format

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port in_v(input,1), in_d(input, 8);
  csl_port out_v(output, 1), out_d(output, 8);
  csl_port clk_p(input);
dut(){
clk_p.set_attr(clock);}
};
csl_vector stim{
  stim(){
    set_vc_header_comment("stimulus_vector");
    set_vc_output_filename("stim_data_out");
    set_unit_name( dut );
    set_direction( input );
  }
};
csl_vector exp{
  exp(){
    set_vc_header_comment("expected_vector");
    set_vc_output_filename("exp_data_out");
    set_unit_name( dut );
    set_direction( output );
  }
};
csl_testbench tb{
```

```
    csl_signal clk;
    dut dut_i;
      tb(){
      clk.set_attr(clock);
      add_logic(clock,clk,10,ps);
      add_logic(generate_waves,"wave.dump", vcd);
    }
  };
```

VERILOG CODE

```
  //
```

add_logic(generate_report [,columns|rows]**);**

DESCRIPTION :

This will create a report generator that will output simulation details related to testing elements on the testbench (vectors and state data) and DUT's responses. The report can be fine tuned to generate the output in colums or rows

[***CSL Testbench Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port in_v(input,1), in_d(input, 8);
  csl_port out_v(output, 1), out_d(output, 8);
  csl_port clk_p(input);
  dut(){
clk_p.set_attr(clock);}
};
csl_vector stim{
  stim(){
    set_vc_header_comment("stimulus_vector");
    set_vc_output_filename("stim_data_out");
    set_unit_name ( dut );
    set_direction   ( input );
  }
};
csl_vector exp{
  exp(){
    set_vc_header_comment("expected_vector");
    set_vc_output_filename("exp_data_out");
    set_unit_name ( dut );
    set_direction   ( input );
     }
};
csl_testbench tb{
  csl_signal clk;
  dut dut_i;
  tb(){
    clk.set_attr(clock);
    add_logic(clock,clk,10,ps);
    add_logic(generate_report);
  }
};
```

VERILOG CODE
//

CHAPTER 2 CSL Register File

TABLE 2.1 Chapter Overview

2.1 CSL Register File Command Summary
2.2 CSL Register File Commands

2.1 CSL Register File Command Summary

2.1.1 Usage tables

CSL Register file
can be defined and instantiated
- can be defined in

TABLE 2.2 CSL unit definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be instantiated in

TABLE 2.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
global scope	-
CSL Unit	YES
CSL Signal Group	-
CSL Interface	-
CSL Testbench	YES
CSL Vector	-

TABLE 2.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
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	-

REGISTER FILE: Signal

```
csl_register_file register_file_name;
int get_width();
int get_depth();
add_logic(read_valid);
```

REGISTER FILE:

```
create_rtl_module();
```

REGISTER FILE: Custom port naming

```
set_reset_name(string);
string register_file_hid.get_reset_name();
set_clock_name(string);
string register_file_hid.get_clock_name();
set_wr_data_name(string);
string register_file_hid.get_wr_data_name();
set_rd_data_name(string);
string register_file_hid.get_rd_data_name();
set_wr_addr_name(string);
string register_file_hid.get_wr_addr_name();
set_rd_addr_name(string);
string register_file_hid.get_rd_addr_name();
set_wr_en_name(string);
string register_file_hid.get_wr_en_name();
set_rd_en_name(string);
string register_file_hid.get_rd_en_name();
set_valid_name(string);
string register_file_hid.get_valid_name();
```

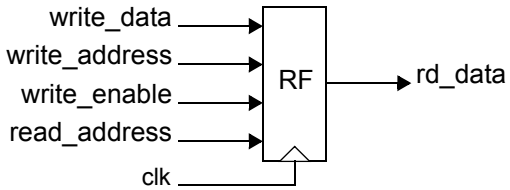
2.2 CSL Register File Commands

csl_register_file *register_file_name;*

DESCRIPTION :

Declares a new register file with the name *register_file_name.*

FIGURE 2.1 Register file with no special options



[CSL Register File Command Summary]

EXAMPLE :

Creates a register file named *RF* and is set the width and depth.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
}
};
```

VERILOG CODE

```
module register_file(clk, reset, rf_wr_data_in, rf_wr_addr, rf_rd_en,
rf_wr_en, rf_rd_addr, rf_data_out);
parameter D_WIDTH=32;
parameter A_WIDTH=16;
input [D_WIDTH - 1: 0] rf_wr_data_in;
input [A_WIDTH - 1: 0] rf_wr_addr;
input rf_rd_en, rf_wr_en, clk, reset ;
output [D_WIDTH - 1: 0] rf_data_out;
input [A_WIDTH - 1: 0] rf_rd_addr;
reg [D_WIDTH - 1: 0] rf [A_WIDTH - 1: 0], rf_data_out;
integer i;
integer j;
   always @ (posedge clk) begin
      if(!reset) begin
         for(i = 0; i < A_WIDTH; i=i+1) begin
              rf[i] = {{D_WIDTH}, 1'b0};
          end
      end
   end
```

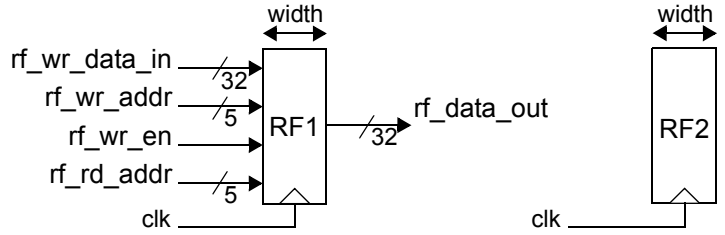
```
   always @ (posedge clk) begin
      if(rf_wr_en) begin
         rf[rf_wr_addr] <= rf_wr_data_in;
      end
      if(rf_rd_en) begin
       rf_data_out <= rf[rf_rd_addr];
      end
   end
endmodule
```

```
int get_width();
```

DESCRIPTION :

It gets the width of the register file words.

FIGURE 2.2



[CSL Register File Command Summary]

EXAMPLE :

In this example we simply create two instances of a register file called *RF1* and *RF2*.

CSL CODE

```
csl_register_file RF1{
RF1(){
set_width(32);
set_depth(4);
}
};
csl_register_file RF2{
RF2(){
set_width(RF1.get_width());
set_depth(4);
}
};
```

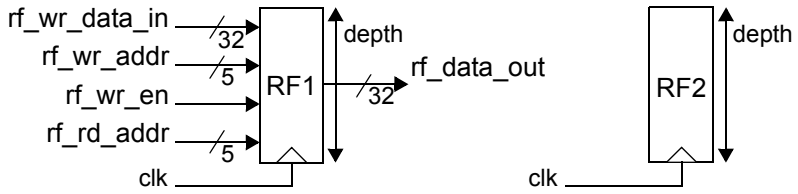
VERILOG CODE

```
int get_depth();
```

DESCRIPTION :

It gets the depth of the register file.

FIGURE 2.3 Register file with no special options



[CSL Register File Command Summary]

EXAMPLE :

In this example we simply create two instances of a register file called *RF1* and *RF2*.

CSL CODE

```
csl_register_file RF1{
RF1(){
set_width(32);
set_depth(4);
}
};
csl_register_file RF2{
RF2(){
set_width(RF1.get_width());
set_depth(RF1.get_depth());
}
};
```

VERILOG CODE

```
add_logic(read_valid);
   port: output - valid_<register_file>
```

DESCRIPTION :

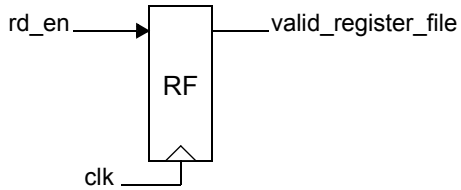
Create an output port for the register file's valid bit. The valid bit is the read enable delayed with 1 clock cycle and is used in pipelines.

[CSL Register File Command Summary]

EXAMPLE :

In this example it is created an output port for valid bit of a register file named RF.

FIGURE 2.4 Register file with valid bit



CSL CODE:

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
add_logic(read_valid);
}
};
```

VERILOG CODE:

```
//to be changed
module
RF(wr_data,wr_address,rd_data,rd_address,clk,wr_en,validation_bits);
input wr_en;
input [7:0] wr_data;
input clk;
input [1:0] wr_address;
input [1:0] rd_address;
output [7:0] rd_data;
output [3:0]validation_bits;
reg [7:0] rd_data;
reg [3:0] validation_bits;
reg [7:0]reg_file[3:0];
integer i;
integer j;
```

```
//initilize register file
initial begin
for(j=0;j<5;j=j+1)
begin
    reg_file[j]=8'b0;
end
end
always@(posedge clk)
begin
if(wr_en) begin
   reg_file[wr_address]=wr_data;
end
end
always@(posedge clk)
begin
    rd_data=reg_file[rd_address];
end
always@(posedge clk)
begin

    if(wr_en)
     begin
         validation_bits[wr_address]=1'b0;
        for(i=0;i<5;i=i+1)
        begin
        if(i!=wr_address)
        begin
        validation_bits[i]=1'b1;
    end
    end
end
    else
    begin
        for(i=0;i<5;i=i+1)
        begin
        validation_bits[i]=1'b1;
    end
    end
end
endmodule
```

create_rtl_module();

DESCRIPTION :

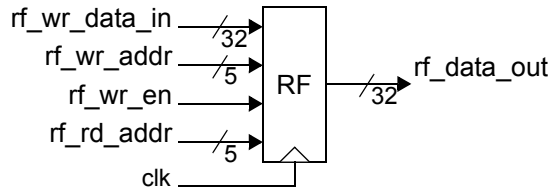
Creates a RTL module from a previous declared register file with the name *module_name* and put it in a file named *rtl_language_extension*.

[CSL Register File Command Summary]

EXAMPLE :

Creates a RTL module for the register file RF.

FIGURE 2.5



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
create_rtl_module();
}
};
```

VERILOG CODE

set_reset_name(string);

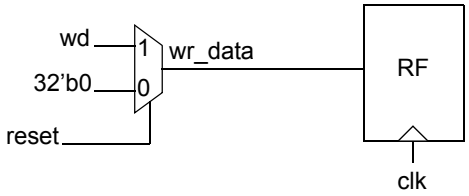
DESCRIPTION :

It sets the *name* for the reset port of register_file.

EXAMPLE :

Sets the name “rst” for reset.

FIGURE 2.6



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_reset_name("rst");
}
};
```

VERILOG CODE

```
string register_file_hid.get_reset_name();
```

DESCRIPTION :

It gets the *name* for the reset port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of reset.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_reset_name("rst");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_reset_name();
```

VERILOG CODE

```
set_clock_name(string);
```

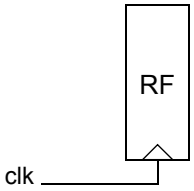
DESCRIPTION :

It sets the *name* for the clock port of register_file.

EXAMPLE :

Sets the name “clk” for the clock.

FIGURE 2.7



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_clock_name("clk");
}
};
```

VERILOG CODE

```
string register_file_hid.get_clock_name();
```

DESCRIPTION :

It gets the *name* for the clock port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of the clock port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_clock_name("clk");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_clock_name();
```

VERILOG CODE

set_wr_data_name(string);

DESCRIPTION :

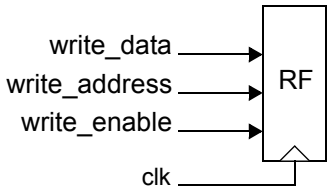
It sets the *name* for the wr_data port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “write_data” for wr_data port of register file RF.

FIGURE 2.8



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_data_name("write_data");
}
};
```

VERILOG CODE

```
string register_file_hid.get_wr_data_name();
```

DESCRIPTION :

It gets the *name* for the wr_data port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of wr_data port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_data_name("write_data");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_wr_data_name();
```

VERILOG CODE

```
set_rd_data_name(string);
```

DESCRIPTION :

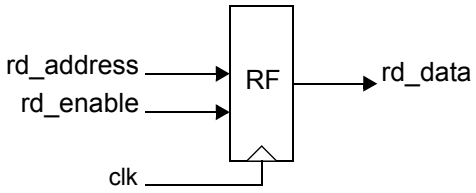
It sets the *name* for the rd_data port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “read_data” for rd_data port of register file RF.

FIGURE 2.9



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_data_name("read_data");
}
};
```

VERILOG CODE

```
string register_file_hid.get_rd_data_name();
```

DESCRIPTION :

It get the *name* for the rd_data port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of rd_data port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_data_name("read_data");
 }
};
csl_unit u{
  RF RF;
  u(){}
};
u.RF.get_rd_data_name();
```

VERILOG CODE

set_wr_addr_name(string);

DESCRIPTION :

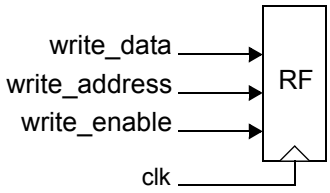
It sets the *name* for the wr_addr port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “write_address” for wr_addr port of register file RF.

FIGURE 2.10



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_addr_name("write_address");
}
};
```

VERILOG CODE

```
string register_file_hid.get_wr_addr_name();
```

DESCRIPTION :

It get the *name* for the wr_addr port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of wr_addr port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_addr_name("write_address");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_wr_addr_name();
```

VERILOG CODE

set_rd_addr_name(string);

DESCRIPTION :

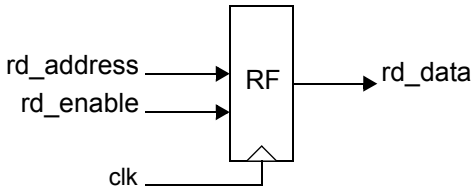
It sets the *name* for the rd_addr port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “read_address” for rd_addr port of register file RF.

FIGURE 2.11



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_addr_name("read_address");
}
};
```

VERILOG COD

```
string register_file_hid.get_rd_addr_name();
```

DESCRIPTION :

It gets the *name* for the rd_addr port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of rd_addr port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_addr_name("read_address");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_rd_addr_name();
```

VERILOG CODE

```
set_wr_en_name(string);
```

DESCRIPTION :

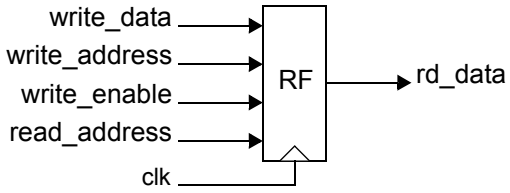
It sets the *name* for the wr_en port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “write_enable” for wr_en port of register file RF.

FIGURE 2.12



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_en_name("write_enable");
}
};
```

VERILOG COD

```
string register_file_hid.get_wr_en_name();
```

DESCRIPTION :

It gets the *name* for the wr_en port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name wr_en port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_wr_en_name("write_enable");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_wr_en_name();
```

VERILOG CODE

```
set_rd_en_name(string);
```

DESCRIPTION :

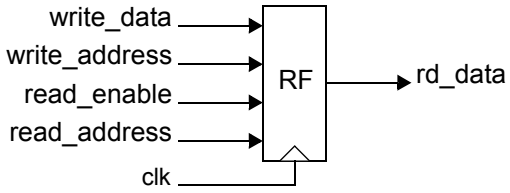
It sets the *name* for the rd_en port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Sets the name “read_enable” for rd_en port of register file RF.

FIGURE 2.13



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_en_name("read_enable");
 }
};
```

VERILOG CODE

```
string register_file_hid.get_rd_en_name();
```

DESCRIPTION :

It gets the *name* for the rd_en port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of rd_en port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_rd_en_name("read_enable");
}
};
csl_unit u{
RF RF;
u(){}
};
u.RF.get_rd_en_name();
```

VERILOG CODE

set_valid_name(string);

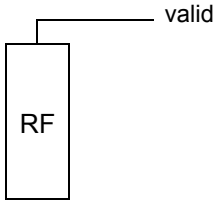
DESCRIPTION :

It sets the *name* for the valid port of register_file.

EXAMPLE :

Sets the name “vld” for valid port of register file RF.

FIGURE 2.14



CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_valid_name("vld");
}
};
```

VERILOG CODE

```
string register_file_hid.get_valid_name();
```

DESCRIPTION :

It gets the *name* for the valid port of register_file.

[CSL Register File Command Summary]

EXAMPLE :

Gets the name of valid port of register file RF.

CSL CODE

```
csl_register_file RF{
RF(){
set_width(32);
set_depth(4);
set_valid_name("valid");
}
};
csl_unit u{
RF RF;
u(){ }
};
u.RF.get_valid_name();
```

VERILOG CODE

CHAPTER 1 CSL Register

TABLE 1.1 Chapter Outline

1.1 CSL Register Syntax and Command Summary

1.1.1 CSL Register class

Registers are state elements that can perform various tasks (store data, increment values, shifting, etc.). CSL includes a set of built-in register classes that ease designs which use this type of state element. Support is included for counter and D type registers. This class allows full configuration of the RTL register code.

1.1.2 CSL Register class declaration

A CSL Register can only be declared in the global scope just like any other CSL class. The CSL Register class is declared as in the below example:

```
csl_register register_name {
  //no objects may be instantiated in a CSL register
  register_name(){
    (register methods calls)+
  }
};
```

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

In the register class' scope there aren't any objects to be specifically declared or instantiated as shown in Table 1.2

TABLE 1.2 Rules for instantiating objects in the register's scope

CSL class	Is instantiated in CSL Vector scope
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	-

TABLE 1.2 Rules for instantiating objects in the register's scope

CSL class	Is instantiated in CSL Vector scope
CSL Isa Field	-
CSL Field	-

1.1.3 CSL Register mandatory commands

Registers are differentiated according to their type. Setting the type and the width for a register is mandatory as shown below:

```
set_type(type);
set_width(numeric_expression);
```

1.1.4 CSL Register common commands

The following set of commands can be called on registers regardless of their type.

CSL Register common commands

```
set_attributes(attribute_list);
csl_list object_name.get_attributes();
add_logic(neg_output);
add_logic(serial_input);
add_logic(serial_output);
add_logic(init[,init_value]);
add_logic(set[,set_value]);
add_logic(reset[,reset_value]);
add_logic(clear[,clear_value]);
```

1.1.5 CSL Register specific commands

Once a type is set for a register certain commands apply only to that specific register type. These commands are used to customize the final register according to the user's needs. Note that D type register does not have any specific commands.

1.1.5.1 Counter Register specific commands

The following commands apply only to the counter register. These optional commands prove to be

useful when customizing the features of the generated RTL counter register.

CSL Counter Register specific commands

```
add_logic(gray_output);
add_logic(count_amount,numeric_expression);
add_logic(stop,stop_value);
add_logic(start_value,numeric_expression);
add_logic(end_value, numeric_expression);
add_logic(direction_control,cnt_dir_signal);
signal_object_name reg_name.get_cnt_dir_signal();
```

1.1.6 CSL Register usage and rules

CSL Registers are declared in the global scope and can be used in designs by being instantiated and only in CSL Units as can be seen in Table 1.3:

TABLE 1.3 CSL Register usage rules

CSL class	Instantiates CSL Register
CSL Unit	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	-

```
//The following commands are not needed: reason: redundant - see above
add_logic(reset);
add_logic(set);
```

1.2 CSL Register Commands

1.2.1 Registers

```
set_type(type);
```

DESCRIPTION :

Sets the type of the register. Supported types include counter or register. Counter infers a user configurable counter register while register type infers a D type register. Register types are mutually exclusive. Only one type may be set for a register. Register counter needs a count signal and a direction control signal.

TABLE 1.4 Register types

type	Results in
counter	counter register
register	D type register

EXAMPLE :

The counter has a count signal. The counter defaults to a start value of 1, and up direction counter. An optional down or up/down direction(s)s may be set.
NOTE: it is illegal to use counter add logic options without setting the register type to counter.

CSL CODE

```
csl_register r4{
  r4(){
    set_width(6);
    set_type(counter);
    add_logic(direction_control);
  }
};
```

VERILOG CODE

```
//
```

```
set_width(numeric_expression);
```

DESCRIPTION :

Sets the width (in bits) of the register.

EXAMPLE :

//

CSL CODE

```
//
```

VERILOG CODE

```
//
```

1.2.1.1 Register attribute

set_attributes(attribute_list);

DESCRIPTION :

Apply access attributes to the memory map page using predefined **attribute** bits *object_name* is any state element type which can be part of the memory map page (eg cells, register, table, register file, SRAM). Sets the attribute for the *reg_object_name* register. If the attribute is not set, the default value is read(rd).The memory attributes can be only one at a time like parameter (*rd* or *wr* or *sh* or *rd_wr*). Memory attributes control the access to a memory mapped structure and can be specified according to Table 1.5.

TABLE 1.5 CSL memory map element atrribute bits

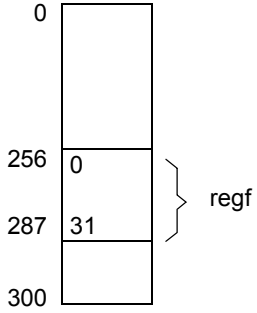
Attribute	Description
read (rd) only	read only register
write (wr) only	write only register
shadow (sh)	shadow register
read_write (rd_wr)	read/write register

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a memory map with a register file inside.

FIGURE 1.1 A memory map page with a register file element



CSL CODE

```
csl_register regA{
  regA(){
    set_type(dff);
    set_width(32);
    set_attributes(rd);
  }
};
csl_memory_map_page mpage{
   regA regA;
```

```
    mpage(){
      add_address_range(0,300);
    }
  };
```

VERILOG CODE

```
`define <MMN>_OBJ_NAME_ATTRIBUTES attribute_list;
```

csl_list *object_name*.**get_attributes();**

DESCRIPTION :

Returns a list of attributes that are applied to *object_name* eithin the memory map.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create two memory maps that have register file elements.

FIGURE 1.2 Two memory map page with a register file elements

CSL CODE

```
csl_register regf{
  regf(){
    set_type(dff);
    set_width(8);
    set_depth(32);
      }
};
csl_memory_map_page mpage1{
regf regf1;
   mpage1(){
   add_address_range(0,300);
   regf1.set_attributes(read);
   }
};
csl_memory_map_page mpage2{
regf regf2;
   mpage2(){
     add_address_range(0,255);
     regf2.set_attributes(regf1.get_attributes())
   }
};
```

VERILOG CODE

```
`define <MMN>_OBJ_NAME_ATTRIBUTES attribute_list;
```

1.2.1.2 Register pins

add_logic(*neg_output***);**
port: output - neg_output

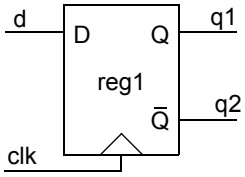
DESCRIPTION :

Automatical create the parallel negative output port with the name *neg_output* for the *reg_object_name* register. The output port is n-bit width (user defined).

[CSL Register Syntax and Command Summary]

EXAMPLE :

FIGURE 1.3



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(dff);
    set_width(4);
    add_logic(neg_output);
  }
};
```

VERILOG CODE

add_logic(*serial_input);*
port: input - serial_input

DESCRIPTION :

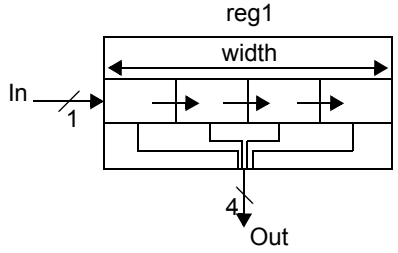
The input signal is connected in serial mode with the register.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a shift register that has serial input and paralel output.

FIGURE 1.4 A register with serial input and paralel output



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(cnt);
    set_width(4);
    add_logic(serial_input);
  }
};
```

VERILOG CODE

```
//AV
module shift_serial_input(rst,clk,in,out);
    input in, clk, rst;
    output [3:0] out;
    reg [3:0] out;
    always @(posedge clk or negedge rst) begin
        if(! rst) begin out = in; end
        else begin out = {in,out} >> 1; end
    end
endmodule
```

add_logic(*serial_output);*
port: output - serial_output

DESCRIPTION :

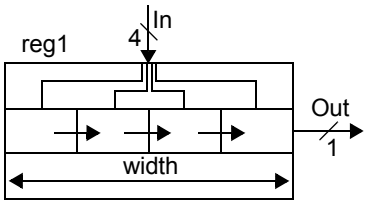
The output signal is connected in serial mode with the register.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a shift register that has serial output and paralel input.

FIGURE 1.5 A register with paralel input and serial output



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(sft);
    set_width(4);
    add_logic(serial_output);
  }
};
```

VERILOG CODE

1.2.1.3 Values

add_logic(init[,*init_value*]**);**

DESCRIPTION :

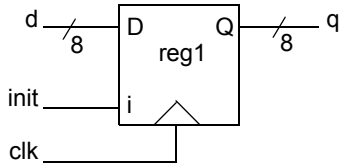
When the init signal is active the memory element is initalized with this value.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Creates a register that have an init signal.

FIGURE 1.6 A register with init signal



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(dff);
    set_width(4);
    add_logic(init,0);
  }
};
```

VERILOG CODE

```
//AV
module reg1(clk,d,q,init);
    parameter init_val = 8'b1;
    input clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    always @(posedge clk) begin
       if(init_sgn) begin q = init_val; end
       else begin q = d; end
    end
endmodule
```

add_logic(set[,*set_value*]**);**

DESCRIPTION :

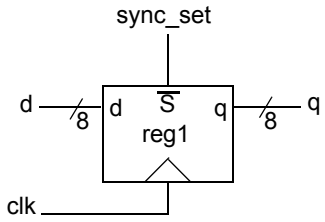
After a set operation the memory element is set to this value. Default is one.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a register named *reg1* that will have an synchronous set pin.

FIGURE 1.7 A register with synchronous set signal



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(dff);
    set_width(4);
    add_logic(set,1);
  }
};
```

VERILOG CODE

```
//AV
module reg1(sync_set,clk,d,q);
    input sync_set,clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    always @(posedge clk ) begin
        if (sync_set) begin q = 8'b1; end
        else begin q = d; end
    end
endmodule
```

add_logic(reset[,*reset_value*]**);**

DESCRIPTION :

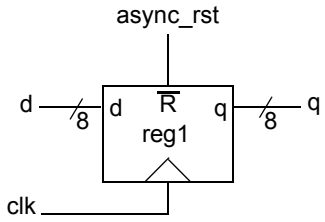
After a reset operation the memory element is set to this value.Default is zero.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a register named *reg1* that will have an asynchronous reset pin.

FIGURE 1.8 A register with asynchronous reset signal



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(dff);
    set_width(4);
    add_logic(reset,0);
  }
};
```

VERILOG CODE

```
//AV
module reg1(async_rst,clk,d,q);
    input async_rst,clk;
    input [7:0] d;
    output [7:0] q;
    reg [7:0] q;
    always @(posedge clk or negedge async_rst) begin
        if (! async_rst) begin q = 8'b0; end
        else begin q = d; end
    end
endmodule
```

add_logic(clear[,*clear_value*]**);**

DESCRIPTION :

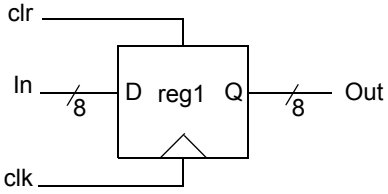
After a clear operation the memory element is set to this value.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a registers. The registers will be cleared when the clear signal will be active.

FIGURE 1.9 A register with clear signal.



CSL CODE

```
csl_register reg1{
  reg1(){
    set_type(dff);
    set_width(4);
    add_logic(clear,0);
  }
};
```

VERILOG CODE

```
//AV
module reg1(clr,clk,in,out);
    parameter clr_val = 8'b0;
    input clr,clk;
    input [7:0] in;
    output [7:0] out;
    reg [7:0] out;
    always @(posedge clk) begin
        if(clr) begin out = clr_val; end
        else begin out = in; end
    end
endmodule
```

1.2.1.4 Register types

1.2.1.4.1 Counter register

add_logic(*gray_output*);
port: output - gray_output

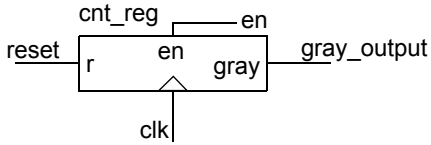
DESCRIPTION :

Automatical create the counter output port signal with the name *gray_output* for the *counter_reg_name.* The *gray_output* signal is in Gray code and the width is equal with the width of counter register.

[CSL Register Syntax and Command Summary]

EXAMPLE :

FIGURE 1.10



CSL CODE

```
csl_register cnt_reg{
  cnt_reg(){
    set_type(cnt);
    set_width(8);
    add_logic(gray_output);
  }
};
```

VERILOG CODE

add_logic(count_amount,*numeric_expression***);**

DESCRIPTION :

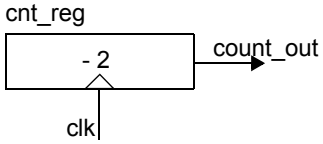
Set the value to increment/decrement the counters.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that will count down to zero from an initial value. The decrementation value is 2

FIGURE 1.11



CSL CODE

```
csl_register cnt_reg{
  cnt_reg(){
    set_type(cnt);
    set_width(8);
    add_logic(count_direction, down);
    add_logic(start_value,8);
    add_logic(count_amount,2);
    add_logic(end_value,2);
  }
};
```

VERILOG CODE

```
//AV
module cnt_down_to_zero(reset, en, clk, count_out);
    parameter count_dir = 0;
    parameter init_val = 8'b1;
    parameter final_val = 8'b0;
    parameter step = 2;
    input reset, en, clk;
    output [7:0] count_out;
    reg [7:0] count_out;
    always @(posedge clk or negedge reset) begin
        if(!reset) begin count_out = init_val; end
        else if(en) begin
            case (count_dir)
            0:count_out = count_out - step;
            1:count_out = count_out + step;
```

```
            default:$display("Wrong step number");
            endcase
         end
    end
endmodule
```

add_logic(count _direction , up|down);

DESCRIPTION :

This command set the count direction by modifing a flag bit for direction named *count_direction* . Default direction is up.

TABLE 1.6 Count direction and flag value

Count direction	Description
up	count up
down	count down

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that will count down to zero from an initial value. The count direction is down

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(cnt);
    set_width(8);
    add_logic(count_direction, down);
    add_logic(start_value,128);
    add_logic(count_amount,1);
    add_logic(end_value,0);
  }
};
```

VERILOG CODE

```
//AV
module cnt_down_to_zero(reset, en, clk, count_out);
    parameter count_dir = 0;
    parameter init_val = 8'b1;
    parameter final_val = 8'b0;
    parameter step = 2;
    input reset, en, clk;
    output [7:0] count_out;
    reg [7:0] count_out;
    always @(posedge clk or negedge reset) begin
        if(!reset) begin count_out = init_val; end
        else if(en) begin
            case (count_dir)
            0:count_out = count_out - step;
            1:count_out = count_out + step;
            default:$display("Wrong value for direction");
```

```
            endcase
          end
    end
endmodule
```

add_logic(start_value,*numeric_expression***);**

DESCRIPTION :

Set the start value. When the counter reaches the end value then the counter resets to the start value. Default is zero.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that count from strat_value 0 up to 128.

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(cnt);
    set_width(8);
    add_logic(count_direction,up);
    add_logic(start_value,0);
    add_logic(count_amount,1);
    add_logic(end_value,128);
  }
};
```

VERILOG CODE

```
//AV
module cnt_up(rst,en1,clk,count_out1);
    parameter start_val = 8'b0;
    parameter end_val = 8'b10000000;
    input rst, en1, clk;
    output [7:0] count_out1;
    reg [7:0] count_out1;
    always @(posedge clk or negedge rst) begin
        if(!rst) begin count_out2 = start_val; end
        else if(en1) begin
            if (count_out1 < end_val)
            count_out2 = count_out2 + 1;
            else if (count_out2 == end_val)
            count_out2 = start_val;
            else
            $display("Wrong value!");
         end
    end
endmodule
```

add_logic(end_value, numeric_expression);

DESCRIPTION :

[CSL Register Syntax and Command Summary]

EXAMPLE :

//

add_logic(stop,*stop_value*);

DESCRIPTION :

When the counter reaches this value then the counter stop to counting and does not reset to the start value until the init signal is asserted.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that count up to 128, and stop_value is 50.

CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(cnt);
    set_width(8);
    add_logic(count_direction,up);
    add_logic(start_value,1);
    add_logic(end_value,128);
    add_logic(count_amount,1);
    add_logic(stop,50);
  }
};
```

VERILOG CODE

```
//AV
module cnt_up(rst,en1,clk,count_out1);
    parameter start_val = 8'b0;
    parameter end_val = 8'b10000000;
    input rst, en1, clk;
    output [7:0] count_out1;
    reg [7:0] count_out1;
    always @(posedge clk or negedge rst) begin
        if(!rst) begin count_out1 = start_val; end
        else if(en1) begin
            if (count_out1 < end_val)
            count_out1 = count_out1 + 1;
         end
    end
endmodule
```

```
add_logic(direction_control,cnt_dir_signal);
   port: input - cnt_dir_signal
```

DESCRIPTION :

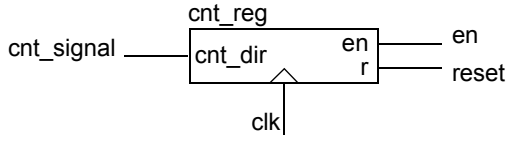
Connect the signal *signal_object_name* to the register *reg_object_name* count pin.
Create the count direction port with the name *cnt_signal* for the *counter_reg_name* counter register.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create a counter that will count up or down from an initial value. The count direction is driven by a signal.

FIGURE 1.12 A counter with a signal that will control the count direction



CSL CODE

```
csl_register reg_cnt{
  reg_cnt(){
    set_type(cnt);
    set_width(8);
    add_logic(direction_control,cnt_signal);
  }
};
```

VERILOG CODE

```
//AV
module cnt1(reset, en, clk, cnt_signal, count_out);
    parameter init_val = 8'b0;
    parameter step = 2;
    input reset, en, clk, cnt_signal;
    output [7:0] count_out;
    reg [7:0] count_out;
    always @(posedge clk or negedge reset) begin
        if(!reset) begin count_out = init_val; end
        else if(en) begin
            case (cnt_signal)
            0:count_out = count_out - step;
            1:count_out = count_out + step;
            default:$display("Wrong value for direction");
            endcase
         end
```

```
    end
endmodule
```

signal_object_name reg_name.**get_cnt_dir_signal();**

DESCRIPTION :

Return the name for the signal that drive the count direction.

[CSL Register Syntax and Command Summary]

EXAMPLE :

Create two counter that will count up or down from an initial value. The count direction is driven by a signal.

CSL CODE

```
csl_register reg_cnt1{
  reg_cnt1(){
    set_type(cnt);
    set_width(8);
    add_logic(direction_control,cnt_signal);
  }
};
csl_register reg_cnt2{
  reg_cnt2(){
    set_type(cnt);
    set_width(8);
    add_logic(direction_control, reg_cnt1.get_cnt_dir_signal());
  }
};
```

VERILOG CODE

```
//AV
module cnt1(reset, en, clk1, cnt_signal, count_out);
    parameter init_val = 8'b0;
    parameter step = 2;
    input reset, en, clk1, cnt_signal;
    output [7:0] count_out;
    reg [7:0] count_out;
    always @(posedge clk1 or negedge reset) begin
        if(!reset) begin count_out = init_val; end
        else if(en) begin
            case (cnt_signal)
            0:count_out = count_out - step;
            1:count_out = count_out + step;
            default:$display("Wrong value for direction");
            endcase
         end
    end
```

```
endmodule
```

1.2.1.4.2 Register register

```
add_logic(reset);
```

DESCRIPTION :

[CSL Register Syntax and Command Summary]

EXAMPLE :

//

CSL CODE

```
csl_register reg1{
reg1(){
set_type(register);
add_logic(reset);
}
};
```

VERILOG CODE

```
//
```

add_logic(set);

DESCRIPTION :

[CSL Register Syntax and Command Summary]

EXAMPLE :

//

CSL CODE

```
csl_register reg1{
reg1(){
set_type(register);
add_logic(set);
}
};
```

VERILOG CODE

```
//
```

CHAPTER 1 CSL Pipeline

TABLE 1.1 Chapter Overview

1.1 CSL Pipeline Command Summary
1.2 CSL Pipeline Commands

1.1 CSL Pipeline Command Summary

CSL Pipeline

```
csl_pipeline pipeline_name(number_of_stages);
```

CSL Pipestage

```
pipestage_naming_convention(NAMING_CONV);
set_prefix(PREFIX_TYPE);
set_prefix(prefix_name);
set_suffix(suffix_TYPE);
set_suffix(suffix_name);
pipeline_name.set_number_of_pipestages(numeric_expression);
pipeline_name.set_attribute(PIPELINE_ATTR);
pipeline_name.set_type(PIPELINE_TYPE);
pipeline_name.associate_pipeline(pipeline_name1);
pipeline_name.replicate(new_pipeline_name);
csl_pipestage pipestage_object_name;
add_pipestage(pipestage_object_name);
set_previous_pipestage(pipestage_object_name,pipestage_object_name);
set_next_pipestage(pipestage_object_name,pipestage_object_name);
set_pipestage_number(n);
set_pipestage_name(name);
connect_stall(stall_signal_name0);
connect_enable(enable_signal_name0);
set_pipestage_valid_input(signal_object_name);
set_pipestage_valid_output(signal_object_name);
branch(list_of_pipestage_names);
merge(list_of_pipestage_names);
inline_file(pipestage_name, file_name);
inline_code(code_statements);
reset_init_value(init_value);
state_element.add_pipeline_delay(direction,expression);
```

1.2 CSL Pipeline Commands

The csl_pipline command creates a new pipeline object. State elements which are part of the processor pipeline are assigned the chip_pipeline object. The first pipestage number is intialized. Each subsequent pipestage is connected to the previous pipestage using the set_previous_pipestage method. Previous pipestage can be connected to subsequent pipestages using the set_next_pipestage method. Pipestages are either automtaically assigned a pipestage number based on the previous pipestage number incremented by one or are explicitly assigned a new pipestage number. Pipestages can be named. The pipestage number and/or the pipestage name may be used in the generated Verilog variable names. The use of the pipestage name and number is controlled by the use_pipestage_name and use_pipe_stage_number methods.
Create a new pipestage. The pipestage is attached to a pipeline. The previous and next piestages are set. The number of the pipestage is automatically based on the value of the previous pipestage. The default number for the first pipestage int he pipeline is 0. The pipestage number of a pipestage can be set explicitly.
All output signals connected to state elementss in each pipestage are optionally prefixed or suffixed with the name _<*pipeline_name*><*stage_name*><*pipestagenumber*>. The sufffix naming convention can be overridden.
If a stall signal or enable signal is used to control the state elements in the pipeline then all of the state elements in the pipeline must have an enable signal.

Pipelines which fork and join can be constructed with the **`set_next_pipestage (branch) and the set_previous_pipestage (merge) methods.`**

CSL Pipeline:

csl_pipeline *pipeline_name(number_of_stages);*

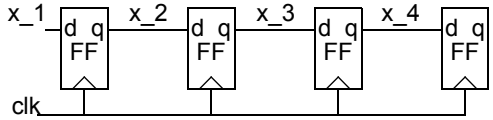
DESCRIPTION :

Create a new pipeline. All signals connected to flip flops in each pipestage are prefixed with the name Each set of pipe stage signals is suffixed with"*_number*" where number is the number of the pipe stage. Pipe stages are added to the pipeline object. The pipe line object can then be checked for various correctness conditions such as correct stall logic and correct signal connections.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.1 A pipeline with four pipestages



CSL CODE

```
csl_pipeline pipeline_name(4);
```

VERILOG CODE

CSL Pipestage:

pipestage_naming_convention(NAMING_CONV);

DESCRIPTION :

Use for naming convention one of the enums in NAMING_CONV.

[***CSL Pipeline Command Summary***]

EXAMPLE :

//

CSL CODE

//

VERILOG CODE

//

```
set_prefix(PREFIX_TYPE);
```

DESCRIPTION :

Override the prefix naming convention using one of the enums in **PREFIX_TYPE**:

```
enum {NO_PREFIX, PIPELINE_NAME, PIPESTAGE_NAME,
PIPELINE_PIPESTAGE_NAME} PREFIX_TYPE;
```

[***CSL Pipeline Command Summary***]

EXAMPLE :

TABLE 1.2

enum	Description	Example
NO_PREFIX		
PIPELINE_NAME		
PIPESTAGE_NAME		
PIPELINE_PIPESTAGE_NAME		

Show the different examples:

- pipeline_name
- pipestage_name
- pipeline n

CSL CODE

```
//csl code goes here
```

set_prefix(*prefix_name***);**

DESCRIPTION :

Override the prefix naming convention using the string *prefix_name*.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
//csl code goes here
```

set_suffix(suffix_TYPE);

DESCRIPTION :

Override the suffix naming convention using one of the enums in **suffix_TYPE**:

```
enum {NO_SUFFIX, PIPELINE_NAME, PIPESTAGE_NAME,
PIPELINE_PIPESTAGE_NAME} SUFFIX_TYPE;
```

[***CSL Pipeline Command Summary***]

EXAMPLE :

TABLE 1.3

enum	Description	Example
NO_SUFFIX		
PIPELINE_NAME		
PIPESTAGE_NAME		
PIPELINE_PIPESTAGE_NAME		

Show the different examples:

- pipeline_name
- pipestage_name
- pipeline n

CSL CODE

```
//csl code goes here
```

set_suffix(*suffix_name***);**

DESCRIPTION :

Override the suffix naming convention using the string *suffix_name*.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
//csl code goes here
```

pipeline_name.**set_number_of_pipestages(***numeric_expression***);**

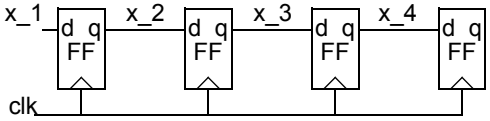
DESCRIPTION :

Set the total number of pipe stages in the pipeline named This method is used by the pipeiine checker to verify that the pipieline is less than or equal to *numeric_expression* pipestages in length.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.2 A pipeline with four pipestages



CSL CODE

```
csl_pipeline pipe;
pipe.set_number_of_pipestages(4);
```

VERILOG CODE

pipeline_name.**set_attribute(PIPELINE_ATTR);**

DESCRIPTION :

Set the pipeline attributes using one of the enums in **PIPELINE_ATTR**:

```
enum {NO_STALL, STALL } PIPELINE_ATTR;
```

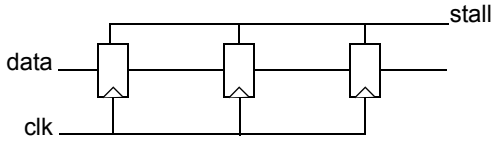
[***CSL Pipeline Command Summary***]

TABLE 1.4

PIPELINE_ATTR	Description
NO_STALL	
STALL	

EXAMPLE :

FIGURE 1.3 A pipeline with stall signal



CSL CODE

```
csl_pipeline pipe(3);
pipe.set_attribute(STALL);
```

VERILOG CODE

pipeline_name.**set_type(PIPELINE_TYPE);**

DESCRIPTION :

Set the pipe stage type using one of the enums in **PIPELINE_TYPE**:

```
enum {VALID, DATA, CONTROL, OTHER} PIPELINE_TYPE;
```

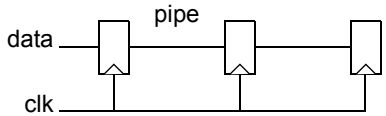
[***CSL Pipeline Command Summary***]

TABLE 1.5

PIPELINE_TYPE	Description
VALID	
DATA	
CONTROL	
OTHER	

EXAMPLE :

FIGURE 1.4 A data pipeline



CSL CODE

```
csl_pipeline pipe(3);
pipe.set_type(DATA);
```

VERILOG CODE

pipeline_name.**associate_pipeline(***pipeline_name1***);**

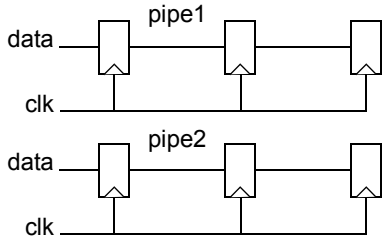
DESCRIPTION :

Associate the pipeline object named *pipeline_name* with the pipeline named *pipeline_name1*. The two pipelines should have the same number of pipe stages, control signals, muliplexers with the same relative pipe stage inputs driving the same relative pipe stages and. Differences between the two pipe stages are flagged by the cslc pipeline checker.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.5



CSL CODE

```
csl_pipeline pipe1(3);
csl_pipeline pipe2(3);
pipe1.associate_pipeline(pipe2);
```

VERILOG CODE

pipeline_name.**replicate(***new_pipeline_name***);**

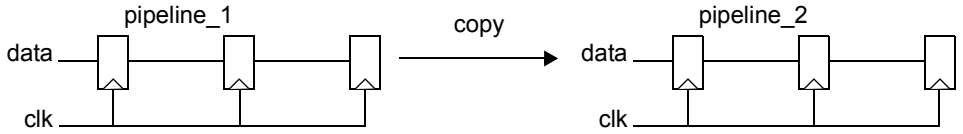
DESCRIPTION :

Create a copy of the pipeline. The new pipeline name *new_pipeline_name* is an object which can have its pipeline type modified. For example, a pipeline for the valid bits can be created and then replicated. The replicated pipeline can be set to a control pipeline. Then the control pipeline can be replicated into a new pipeline and the new pipeline can have its type set to data.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.6



CSL CODE

```
csl_pipeline pipeline_1(3);
pipeline_1.replicate(pipeline_2);
```

VERILOG CODE

csl_pipestage *pipestage_object_name;*

DESCRIPTION :

Create a new pipestage. The pipestage is attached to a pipeline. The previous and next pipestages are set. The number of the pipestage is automatically based on the value of the previous pipestage. The default number for the first pipestage int he pipeline is 0. The pipestage number of a pipestage can be set explicitly.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipestage p3;
```

VERILOG CODE

add_pipestage(*pipestage_object_name*);

DESCRIPTION :

Create a new pipestage.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe;
csl_pipestage p0;
mp_pipe.add_pipestage(p0);
csl_pipestage p1;
mp_pipe.add_pipestage(p1);
csl_pipestage p2;
mp_pipe.add_pipestage(p2);
```

VERILOG CODE

set_previous_pipestage(*pipestage_object_name,pipestage_object_name*);

DESCRIPTION :

Set the previous pipestage of the current pipestage.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe;
csl_pipestage p0;
mp_pipe.add_pipestage(p0);
csl_pipestage p1;
mp_pipe.add_pipestage(p1);
csl_pipestage p2;
mp_pipe.add_pipestage(p2);
p3.set_previous_pipestage(p2);
p2.set_previous_pipestage(p1);
p1.set_previous_pipestage(p0);
```

VERILOG CODE

set_next_pipestage(*pipestage_object_name,pipestage_object_name*);

DESCRIPTION :

Set the next pipestage of the current pipestage.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe;
csl_pipestage p0;
mp_pipe.add_pipestage(p0);
csl_pipestage p1;
mp_pipe.add_pipestage(p1);
csl_pipestage p2;
mp_pipe.add_pipestage(p2);
p3.set_next_pipestage(p2);
p2.set_next_pipestage(p1);
p1.set_next_pipestage(p0);
```

VERILOG CODE

set_pipestage_number(*n*);

DESCRIPTION :

Set the pipestage number belonging to This will be the pipe stage number for the pipe stage. If this method is not used then the default for the pipestage is 0 ("_0") if this is the first piestage and each subsequent pipestage is assigned the auto-incremented pipestage number(n + 1) **. The pipeline checker verifies that all pipestage numbers are unique.**

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe;
mpo_pipe.set_pipestage_number(2);
```

set_pipestage_name(*name***);**

DESCRIPTION :

The name of the pipestage belonging to Each subsequent pipestage uses the autoincremented pipestage name(n + 1) unless the pipestage name is set using the **set_pipestage_name** method. The pipeline checker verifies that all pipestage names are unique.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe();
mpo_pipe.set_pipestage_name("stage1");
```

connect_stall(*stall_signal_name0***);**

DESCRIPTION :

Connect a stall signal to a pipeline. The stall is inverted before being connected to the FF enable. The stall signal controls the enable fo the state elements in the pipeline.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe();
csl_signal stall;
mpo_pipe.connect_stall(stall);
```

connect_enable(*enable_signal_name0***);**

DESCRIPTION :

Connect a enable signal to a pipeline. The enable is inverted before being connected to the FF enable. The enable signal controls the enable fo the state elements in the pipeline.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
csl_pipeline mp_pipe();
csl_signal enable;
mpo_pipe.connect_enable(enable);
```

set_pipestage_valid_input(signal_object_name);

DESCRIPTION :

//

[***CSL Pipeline Command Summary***]

EXAMPLE :

//

CSL CODE:

//

VERILOG CODE:

set_pipestage_valid_output(signal_object_name);

DESCRIPTION :

//

[***CSL Pipeline Command Summary***]

EXAMPLE :

//

CSL CODE:

VERILOG CODE:

branch(*list_of_pipestage_names***);**

DESCRIPTION :

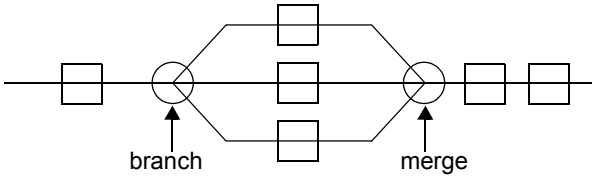
Pipeline branch specifies that the pipe stages named *list_of_pipestage_names* are all part of a pipeline branch. The valid pipe that is assocated with the pipeline has the valid bits qualified by the pipeline mux selects.

[***CSL Pipeline Command Summary***]

USE set_preivous_pipestage and set_next_pipestage instead of branch and merge
Use the example below with set_preivous_pipestage and set_next_pipestage methods

EXAMPLE :

FIGURE 1.7



CSL CODE

```
//csl code goes here
```

merge(*list_of_pipestage_names***);**

DESCRIPTION :

Pipeline merge specifies that the pipe stages named *list_of_pipestage_names* are all part of a pipeline merge. Each branch which is part of the merge has previously been specified to be part of a pipeline branch. The merge mechanism is a mulitplexer which is controlled by a mux select line. The mux select line is used to create the valid bit qualifiers for the entry point into each pipe stage in the branch.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
//csl code goes here
```

inline_file(*pipestage_name, file_name***);**

DESCRIPTION :

Insert the contents of the named file *file_name* after the pipestage named *pipestage_name* in the generated verolog code.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
//csl code goes here
inline_file(fn)
```

VERILOG CODE

```
module
  generated pipestage code
  `include "fn
```

inline_code(*code_statements***);**

DESCRIPTION :

Insert the code after the pipestage.

[***CSL Pipeline Command Summary***]

EXAMPLE :

CSL CODE

```
//csl code goes here
```

VERILOG CODE

```
`include "fn
```

reset_init_value(*init_value***);**

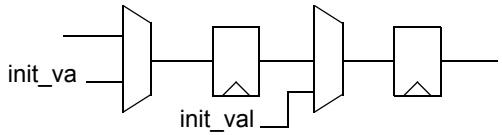
DESCRIPTION :

At reset initialize all flip flops in the pipeline with the value *init_value*.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.8



CSL CODE

```
//csl code goes here
```

state_element.**add_pipeline_delay(**direction,expression**);**

DESCRIPTION :

//

[***CSL Pipeline Command Summary***]

EXAMPLE :

We add a pipeline to the output of the unit with the delay of 1

CSL CODE:

```
csl_register_file rf;
rf.add_pipeline_delay(output,1);
```

VERILOG CODE:

match_pipeline_latency(in0, outn, new_pipeline_name, in10, out1n**);**

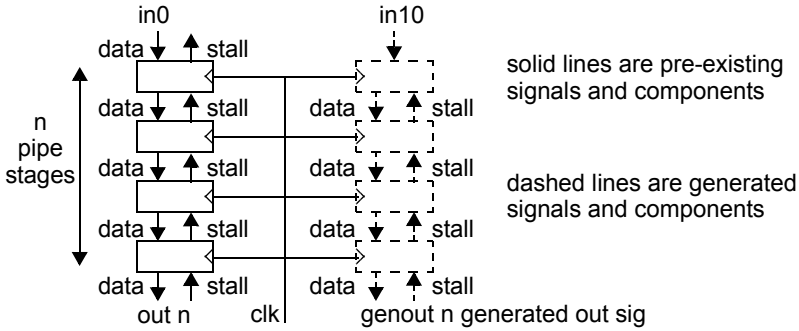
DESCRIPTION :

Pipeline latency match generator
Pipeline latency checker
The pipeline will measure the number (**n**) of pipestages between two specified signals in an existing pipeline and generate **n** pipe stages in a new pipeline. The import signal to the **n** pipestages is specified. The clk signal and enable signals for each stage are extracted from the original set of pipestages.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.9



Checks that the two pipelines are the same length. If the pipelines branch and merge the checker will follow all paths & check all paths

get_pipeline_latency(in0, outn**);**

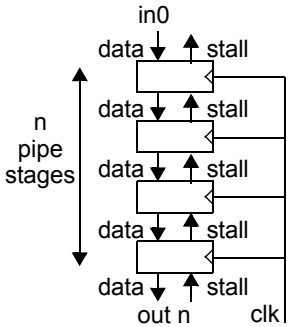
DESCRIPTION :

Return the number of pipestages between the 2 signals.

[***CSL Pipeline Command Summary***]

EXAMPLE :

FIGURE 1.10



If the pipelines branch and merge the checker will follow all paths & check all paths.

CHAPTER 2 CSL Memory Map

TABLE 2.1 Chapter Outline

2.1 CSL Memory Map Command Summary

2.1.1 Memory map rules

1. Every unit has its own local address map in the conceptual model.

2. An object from a unit can only be added to a memory map once.

3. Multiple instances of the same unit all have the same local address map

4. A unit can contian multiple instances of unit's mulitple instance.

5. The parent map is formed by the concatenation of the address maps of the child units

6. memory map ranges are declared as free or reserved.

7. The address ranges which an object is added to is no longer free.

8. It is illegal to add an object to an address range which is reserved.

9. It is illegal to add an object outside of a declared address range

10. It is illegal to add an object to a used/not free address range

11. It is illegal to have a memory map page which is not added to a memory map.

11. It is illegal to have a memory map with out memory map page instances.

12. It is illegal to have flat memory map with multiple page instances..

13. It is illegal for a flat memory map to have hierarchical memory pages.

14. It is illegal for a virtual memory map to have hierarchical memory pages.

15. It is legal to have Hierarchical memory pages for hierarchical memory map.

16. What can be added to a memory map page? Instances of an addressable object
    which include:
    csl_memory
    csl_register
    csl_register_file
    csl_fifo

17. Before adding an object ot a memory map page the following are required:
    add all address ranges (legal, reserved)

18. After the memory map page is built it is then added to the memory map.

19. It is legal to add the memory map page to the memory map and then add objects to the memory page.

20. mandatory cmds for a memory map are set_type

21. memory map must have memory map pages instantiated

22. mandatory cmds for a memory map page are add_address_range

23. a memory map page must have at least one adressable object added to it. A memory map page cannot be empty

24. The design address limits for the memory map is optional
set_address_limits(lower, upper) -this is the valid address range for the chip
set_address_width(numeric_expression)-this is the width of the address bus on the chip
the limits and the width can be set. This is related to the address bus on the chip.

25. If the design address limits for the memory map is set then check all of the pages to see that the memory map page address range is wth in the design address limits and that the width of the memory map page...

26. A memory map page has a set_unit

27. A memory map page has to have a set_unit before adding addressable objects to the memory map page.

27. Every addressable object added to the memory map page has to be instantiated in the unit specified by set_unit in the memory map page.

28. Only one set_unit can be called per memory map page

```
csl_unit a {
 rf rf0;
 a() {}
};
csl_memory_map_page mpa {
  mpa(){
    set_unit(a);
    add(rf0);
  }
};
```

29. A unit can only be added to one memory map page

30. It is illegal to add a unit can only be added to more than one memory map page

31. The unit instances must have different base addresses

32. Multiple instances of the same page need to have different base addresses

33. If no base address is specified for a unit instance then the base address of the unit instance in the design memory map is calculted by the order of the instantiation of the pages in the memory map csl file(s).

34. The base address for the unit instances can be set using the following code example

```
csl_unit a { a(){} };
csl_unit b { a a0; a a1;  b(){} };
csl_unit top  { b b0; b b1; top(){} };

csl_memory_map_page apage{
  apage(){
    set_unit(a);
    add_address_range(0,511);
    set_address_increment(2);
  }
};

csl_memory_map mm{
  mm(){
    set_type(hier);
    top.b0.a0.set_base_address(  500);
    top.b0.a1.set_base_address( 1000);
    top.b1.a0.set_base_address( 1500);
    top.b1.a1.set_base_address( 2000);
  }
};
```

35. Memory map pages are automatically added to the memory map.

36. Memory map pages will be automatically self registering with the design's single memory map class. The memory map page will NOT be instantiated in the memory map. This is consistent with testbench and vc's.

37. the ***add_reserved_address_range*** can NOT be called on a memory_map_page instance.
38. A range may only be set once in a memory page. Setting any part of the range again is illegal
39. Memory map page methods can only be called in memory map page constructors.

2.1.2 Usage tables

CSL Memory map page
can be defined and instantiated
- can be defined in

TABLE 2.2 CSL memory map page definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be instantiated in

TABLE 2.3 CSL memory map page instantiation in other CSL classes

CSL class	Can be instantiated in
global scope	-
CSL Unit	-
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-

TABLE 2.3 CSL memory map page instantiation in other CSL classes

CSL class	Can be instantiated in
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	YES
CSL Memory Map Page	YES
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

CSL Memory map

can be defined and NOT instantiated

- can be defined in

TABLE 2.4 CSL memory map definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

Memory Map Page: Declaration

```
csl_memory_map_page memory_map_page_name;
memory_map_page_name.add_address_range(lower_bound,upper_bound);
memory_map_page_name.set_address_increment(numeric_expression);
//numeric_expression memory_map_page_name.get_address_increment();
memory_map_page_name.set_next_address(numeric_expression);
//numeric_expression memory_map_page_name.get_next_address();
[memory_map_page_name.]address_range.set_access_rights(group,acc_right
);
//enum memory_map_page_name.get_access_rights_enum(group);
memory_map_page_name.add_reserved_address_range(lower_bound,upper_boun
d);
memory_map_page_name.add(addressable_object,symbol [,base_address]);
addr_obj.add_to_memory_map();
memory_map_page_name.add(memory_map_page_object_name);
int get_lower_bound();
int get_upper_bound();
```

Memory Map Page: Word width

```
set_data_word_width(width);
int get_data_word_width();
```

Memory Map Page: Alignment

```
set_alignment(numeric_expression);
int get_alignment();
```

Memory Map Page: Endianess

```
set_endianess(endianess_type);
enum get_endianess();
```

Memory Map Page: Naming

```
set_symbol_max_length(numeric_expression);
int get_symbol_length();
```

Memory Map: Declaration

```
csl_memory_map memory_map_name;
```

Memory Map: Auto generate

```
auto_gen_memory_map();
set_top_unit(unit_object_name);
object_name.add_to_memory_map([address],[group,access_right]);
```

Memory Map: Declaration

```
set_type(memory_map_type);
```

Memory Map: Access rights

```
memory_map_page_name.set_access_rights_enum(enum_name);
//enum memory_map_page_name.get_access_rights();
```

Memory Map: Word width

```
    set_data_word_width(numeric_expression);
    int get_data_word_width();
```

Memory Map: Prefix

```
    set_prefix(string);
    string get_prefix();
```

Memory Map: Suffix

```
    set_sufix(string);
    string get_sufix();
```

TBD:
set_addr_abs(constant_numeric); //new address
set_addr_rel(constant_numeric); //new address = offset from current address
//add set max number of words command

2.2 CSL Memory Map Commands

csl_memory_map_page *memory_map_page_name*;

DESCRIPTION :

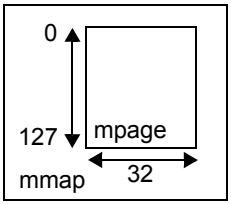
Creates a memory map page named *memory_map_page_name*. It is a scope delimited by curly braces. Each memory map page has an address range.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example we have one memory map *mmap* which contains a memory map page *mpage*.

FIGURE 2.1



CSL CODE

```
csl_memory_map_page mpage{
mpage(){
add_address_range(0,128);
}};
csl_memory_map mmap{
mpage mpage;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

memory_map_page_name.**add_address_range(***lower_bound*,*upper_bound***);**

DESCRIPTION :

This command adds an addres range to a memory map page named *memory_map_page_name* The address range is set using *lower_bound* and *upper_bound.*

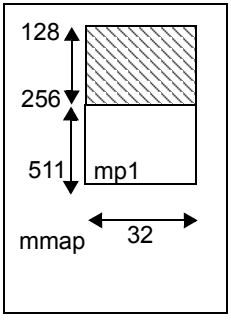
The *add_reserved_address_range* command shows that a part of the initial address range is marked as reserved(and that add_address_range has to be called before add_reserved_address_range).

[CSL Memory Map Command Summary]

EXAMPLE :

In this example it was created a memory map named *mmap* with a memory map page named *mpage_1.*

FIGURE 2.2



CSL CODE

```
csl_memory_map_page mpage_1{
mpage_1(){
add_address_range(128,511);
add_reserved_address_range(128, 256);
}
};
csl_memory_map mmap{
mpage_1 mp1;
mmap(){
set_data_word_width(32);
     }
};
```

VERILOG CODE

memory_map_page_name.**set_address_increment(***numeric_expression***);**

DESCRIPTION :

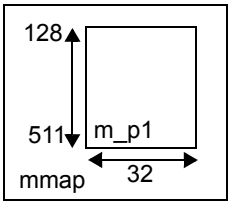
The address increment is the amount that the memory address increments from word to word for a memory map page named *memory_map_page_name* .The *numeric_expression* represent the increment of address.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example it is set the address increment for a memory map page named *m_p1* from a memory map named *mmap*.

FIGURE 2.3



CSL CODE:

```
csl_memory_map_page memp1{
memp1(){
set_address_increment(4);
add_address_range(128, 511);}
};
csl_memory_map mmap{
memp1 m_p1;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE:

memory_map_page_name.**set_next_address(***numeric_expression***);**

DESCRIPTION :

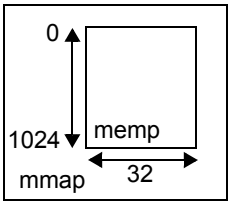
Sets the next address for a memory map page named *memory_map_page_name* . The *numeric_expression* represents the next address.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example it is set the next address in a memory map page named *memp.*

FIGURE 2.4



CSL CODE

```
csl_memory_map_page mpage{
mpage(){
add_address_range(0, 1024);
set_next_address(64);
}
};
csl_memory_map mmap{
mpage memp;
mmap(){
set_data_word_width(32);}
};
```

VERILOG CODE

[*memory_map_page_name.*]*address_range*.**set_access_rights(**group,acc_right**);**

DESCRIPTION :

With this command we can set the access rights for an address range *address_range* from a memory map page *memory_map_page_name.*
The access rights can be the following:

TABLE 2.5

acc_right	Description
access_none	without access rights
access_read	access rights only for read
access_write	access rights only for write
access_read_write	access rights for read-write

[CSL Memory Map Command Summary]

EXAMPLE :

Sets the access rights for the memory map page named *mp.*

CSL CODE

```
csl_memory_map_page mp{
mp(){
set_access_rights(HWR, access_read_write);
set_access_rights(SWR, access_read);
}
};
csl_memory_map mmap{
mp mp;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

memory_map_page_name.**add_reserved_address_range(**lower_bound,upper_bound**);**

DESCRIPTION :

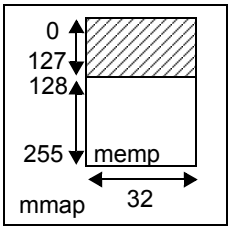
Adds a reserved address range in a memory map page *memory_map_page_name*. The reserved address range is adds by `lower_bound and upper_bound.`
This command shows that a part of the initial address range is marked as reserved(and that add_address_range has to be called before add_reserved_address_range).

[CSL Memory Map Command Summary]

EXAMPLE :

In a memory map page *mpage* is added a reserved address range.

FIGURE 2.5



CSL CODE

```
csl_memory_map_page mpage{
mpage(){
add_address_range(0,255);
add_reserved_address_range(0,127);
set_address_increment(2);
}
};
csl_memory_map mmap{
mpage mpage;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

memory_map_page_name.**add(**addressable_object,symbol [,base_address]**);**

DESCRIPTION :

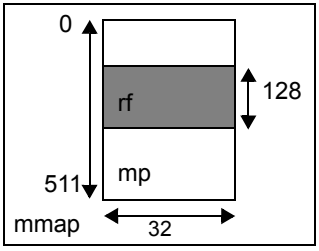
Adds to a memory map page named *memory_map_page_name* an *addressable_object* which can be a fifo , register, register file, memory that are instantiate in a unit.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example it is added a register file named *rf* in a memory map page *mp.* The *symbol* for register file is “rf” and the *base_address* is 64.

FIGURE 2.6



CSL CODE

```
csl_register_file rf{
rf(){
   set_width(32);
   set_depth(128);
   }
};
csl_unit a {
   rf r1;
 };
csl_memory_map_page mpage{
mpage(){
   add_address_range(0,511);
   set_address_increment(2);
   add(a.r1, "rf", 64);
   }
};
```

addr_obj.**add_to_memory_map**();

DESCRIPTION :

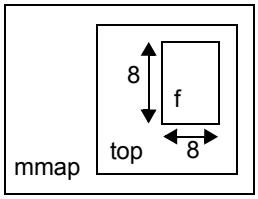
Adds to a memory map an addressable object *addr_obj* which can be a fifo , register, register file, memory that are instantiate in a unit.

[CSL Memory Map Command Summary]

EXAMPLE :

We added in a memory map *mmap* a fifo named *f* which is instantiated in a unit *top.*

FIGURE 2.7



CSL CODE

```
csl_fifo f1{
f1(){
set_width(8);
set_depth(8);
add_to_memory_map();
}
};
csl_unit top{
f1 f;
top(){}
};
csl_memory_map mmap{
mmap(){}
};
```

VERILOG CODE

```
`define DEFAULT_GROUP_hwr 2
`define DEFAULT_GROUP_test 3
`define DEFAULT_GROUP_driver 4
module f1(push,
          pop,
          full,
          empty,
          data_out,
          data_in,
          reset_,
```

```
        clock,
        valid);
parameter ADDR_WIDTH = 3'd4;
parameter DATA_WIDTH = 4'd8;
input push;
input pop;
input [7:0] data_in;
input reset_;
input clock;
output reg full;
output reg empty;
output reg [7:0] data_out;
output reg valid;
reg [ADDR_WIDTH - 1:0] wr_addr;
reg [ADDR_WIDTH - 1:0] rd_addr;
reg wr_en;
reg rd_en;
assign   full = wr_addr + 1 == rd_addr;
assign   empty = wr_addr == rd_addr;
assign   wr_en = !full && push;
assign   rd_en = !empty && pop;
f1_fifo_memory fifo_memory_instance(.clock(clock),
                                    .data_in(data_in),
                                    .data_out(data_out),
                                    .rd_addr(rd_addr),
                                    .rd_en(rd_en),
                                    .reset_(reset_),
                                    .valid(valid),
                                    .wr_addr(wr_addr),
                                    .wr_en(wr_en));

always @( posedge clock or negedge reset_ )  begin
  if ( ~reset_ )  begin
    rd_addr <= 1'd0;
  end
  else       if ( pop )  begin
      rd_addr <= rd_addr + 1'd1;
    end
end
```

```
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      wr_addr <= 1'd0;
    end
    else       if ( push )  begin
        wr_addr <= wr_addr + 1'd1;
      end
  end
endmodule

module f1_fifo_memory(clock,
                      reset_,
                      data_in,
                      data_out,
                      valid,
                      wr_addr,
                      rd_addr,
                      wr_en,
                      rd_en);
  parameter ADDR_WIDTH = 3'd4;
  parameter DATA_WIDTH = 4'd8;
  parameter NUM_WORDS = (1'd1 << DATA_WIDTH);
  input clock;
  input reset_;
  input [DATA_WIDTH - 1:0] data_in;
  input [ADDR_WIDTH - 1:0] wr_addr;
  input [ADDR_WIDTH - 1:0] rd_addr;
  input wr_en;
  input rd_en;
  output reg [DATA_WIDTH - 1:0] data_out;
  output reg valid;
  reg [DATA_WIDTH - 1:0] internal_memory[1'd0:NUM_WORDS - 1'd1] ;

  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      valid <= 1'd1;
    end
    else  begin
      valid <= rd_en;
      data_out <= internal_memory[rd_addr];
```

```
      if ( wr_en )  begin
        internal_memory[wr_addr] <= data_in;
      end
    end
  end
endmodule

module top();
  f1 f();
endmodule
```

memory_map_page_name.**add(***memory_map_page_object_name***);**

DESCRIPTION :

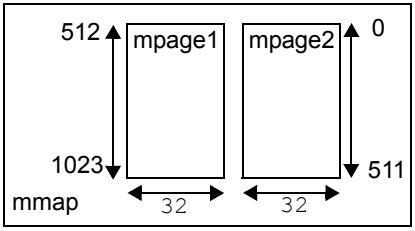
This command adds a memory map page object named *memory_map_page_object_name* to a memory map page *memory_map_page_name*.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example two memory map pages *mpage1* and *mpage2* are added.

FIGURE 2.8



CSL CODE

```
csl_memory_map_page mpage1{
mpage1(){
add_address_range(512,1023);
add(mpage1);
}
};
csl_memory_map_page mpage2{
mpage2(){
add_address_range(0,511);
add(mpage2);
}
};
csl_memory_map mmap{
mpage1 mpage1;
mpage2 mpage2;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

```
//
```

```
int get_lower_bound();
```

DESCRIPTION :

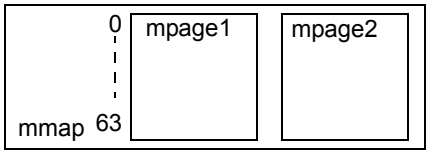
Returns the lower address of a memory range object. Return type can be integer, hex or octal number.

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages with the same address range. The address range for the second memory map page is set using *get_lower_bound()* and *get_upper_bound()* methods.

FIGURE 2.9 Two memory map pages



CSL CODE

```
csl_memory_map_page mpage1{
mpage1(){
add_address_range(0, 63);
}
};
csl_memory_map_page mpage2{
mpage1 mpage1;
mpage2(){
mpage2.add_address_range(mpage1.get_lower_bound(),mpage1.get_upper_bou
nd());
}
};
csl_memory_map mmap{
mpage1 mpage1;
mpage2 mpage2;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

```
//AV
```

```
int get_upper_bound();
```

DESCRIPTION :

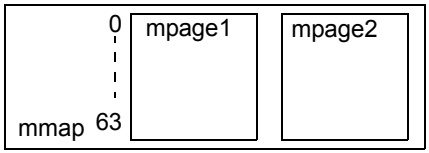
Returns the upper address of a memory range object. Return type can be integer, hex or octal number.

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages with the same address range. The address range for the second memory map page is set using *get_lower_bound()* and *get_upper_bound()* methods.

FIGURE 2.10 Two memory map pages



CSL CODE

```
//AV
csl_memory_map_page mpage1{
mpage1(){
add_address_range(0, 63);
}
};
csl_memory_map_page mpage2{
mpage1 mpage1;
mpage2(){
mpage2.add_address_range(mpage1.get_lower_bound(),mpage1.get_upper_bou
nd());
}
};
csl_memory_map mmap{
mpage1 mpage1;
mpage2 mpage2;
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

```
//AV
```

set_data_word_width(*width*);

DESCRIPTION :

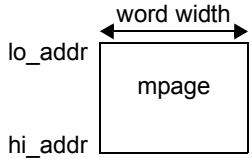
Sets the width of the words in the memory map page. If the memory map page width is specified, it is not necessary to declare the width of each individual word. The elements that will be added to the memory map page must have the width less or equal with the word with of memory map page.

[CSL Memory Map Command Summary]

EXAMPLE :

Create a memory map page named *mpage* with the word width 32.

FIGURE 2.11 A memory map page with word width 32



CSL CODE

```
csl_memory_map_page mpage{
mpage(){
set_data_word_width(32);
}
};
```

VERILOG CODE

C++ CODE

```
const int WORD_WIDTH = 16;
```

```
int get_data_word_width();
```

DESCRIPTION :

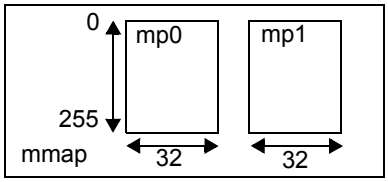
Return the width of the words in memory map page.

[CSL Memory Map Command Summary]

EXAMPLE :

Create a memory map with two memory map pages. The data word width for the second memory map page is set using *get_data_word_method* ().

FIGURE 2.12 A memory map named mem_map with two memory map pages



CSL CODE

```
//AV
csl_memory_map_page mpage0{
mpage0(){
add_address_range(0,255);
set_data_word_width(32);
}
};
csl_memory_map_page mpage1{
  mpage0 mpage0;
mpage1(){
add_address_range(0,255);
set_data_word_width(mpage0.get_data_word_width());
}
};
csl_memory_map mmap{
mpage0 mp0;
mpage1 mp1;
mmap(){}
};
```

VERILOG CODE

```
//AV
```

set_alignment(*numeric_expression***);**

DESCRIPTION :

Address alignment - addresses are byte, half-word (16-bit), word (32-bit), double-word (64-bit), quad-word (128-bit) aligned. The width of the memory elements in a particular memory element range.

TABLE 2.6 Byte aligned

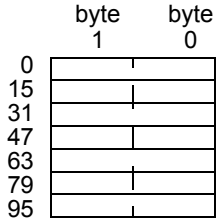
Type	Dimension
byte	8
half word	16
word	32
double word	64
long word	128
quad word	256

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages named *mpage_0* and *mpage_1* set alignment as byte.

FIGURE 2.13 A memory map with byte alignment



CSL CODE

```
csl_memory_map_page mpage_0{
mpage_0(){
add_address_range(0,63);
set_address_increment(2);
set_alignment(16);
}
};
csl_memory_map mmap{
mpage_0 mpage_0;
mmap(){
set_data_word_width(16);
}
};
```

VERILOG CODE

```
`define <MMN>_ALIGN 16
```

```
int get_alignment();
```

DESCRIPTION :

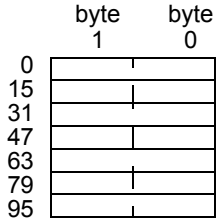
Returns the address alignment setting. The return type is integer.

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages named *mpage_0* and *mpage_1* and set alignment as byte.

FIGURE 2.14 Two memory maps with byte alignment



CSL CODE

```
csl_memory_map_page mpage_0{
mpage_0(){
add_address_range(0,63);
set_address_increment(2);
set_alignment(64);
}
};
csl_memory_map_page mpage_1{
    mpage_0 mpage_0;
mpage_1(){
add_address_range(64,512);
set_address_increment(2);
set_alignment(mpage_0.get_alignment());
}
};
csl_memory_map mmap{
mpage_1 mpage_1;
mmap(){
  set_data_word_width(16);
}
};
```

VERILOG CODE

```
`define <MMN1>_ALIGN 8
`define <MMN2>_ALIGN 8
```

set_endianess(*endianess_type***);**

DESCRIPTION :

The endianess of the memory map can be specified with the **endianess** keword. The *endianess_type* can be either little endian or big endian respectively.

TABLE 2.7 Endianess Type

Endianess	Mnemonic
Little Endian	little_endian
Big Endian	big_endian

[CSL Memory Map Command Summary]

EXAMPLE :

Create a memory map page named *mpage0* and set endianess to big_endian.

FIGURE 2.15 Little Endian

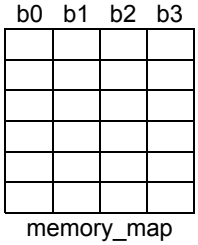
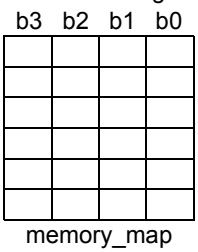


FIGURE 2.16 Big Endian



CSL CODE

```
csl_memory_map_page mpage0{
mpage0(){
add_address_range(0,128);
set_endianess(big_endian);
}
};
csl_memory_map mmap{
mpage0 mpage0;
mmap(){
set_data_word_width(32);
```

```
    }};
```

```
enum get_endianess();
```

DESCRIPTION :

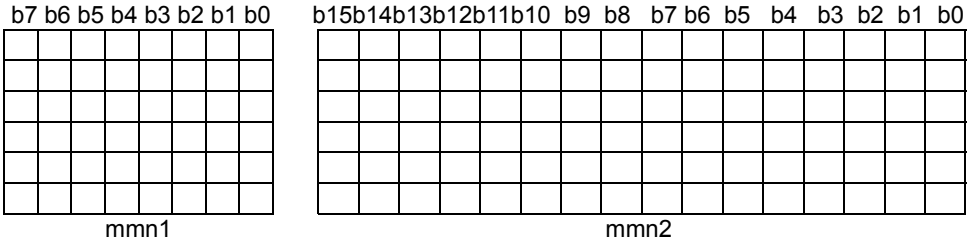
Return the endianess type of a memory map.

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages and set endianess to big_endian.

FIGURE 2.17 Big Endian



CSL CODE

```
csl_memory_map_page pg1{
pg1(){
add_address_range(0, 128);
set_data_word_width(32);
set_endianess(big_endian);
}
};
csl_memory_map_page pg2{
pg1 pg1;
pg2(){
add_address_range(0, 128);
set_data_word_width(32);
set_endianess(pg1.get_endianess());
}
};
csl_memory_map mmn1{
pg2 pg2;
mmn1(){}
};
```

VERILOG CODE

```
//AV
`define MMN1_ENDIANESS ENDIAN_BIG
`define MMN1__ALIGN 8
`define MMN2_ENDIANESS ENDIAN_BIG
`define MMN2__ALIGN 16
```

set_symbol_max_length(*numeric_expression***);**

DESCRIPTION :

Sets the maximum number of characters for each word (name) in the memory map name. The number of characters for each word can be less or equal with the maximum length.

[CSL Memory Map Command Summary]

EXAMPLE :

Create a memory map page with the name *mpage_0* and set the maximum number of characters for the name of elements to 10.

CSL CODE

```
csl_memory_map_page mpage_0 {
mpage_0(){
add_address_range(0, 63);
set_symbol_max_length(10);
}
};
csl_memory_map mmap{
mpage_0 mpage_0;
mmap(){}
};
```

VERILOG CODE

```
`define MEM_MAP_MAX_LENGTH 10;
```

```
int get_symbol_length();
```

DESCRIPTION :

Returns the number which specifies the number of characters in the name.

[CSL Memory Map Command Summary]

EXAMPLE :

Create two memory map pages and set the maximum number of characters for each word to 8

CSL CODE

```
csl_memory_map_page mpage_0 {
mpage_0(){
add_address_range(0, 63);
set_symbol_max_length(8);
}
};
csl_memory_map_page mpage_1 {
mpage_0 mpage_0;
mpage_1(){
add_address_range(64, 512);
set_symbol_max_length(mpage_0.get_symbol_length());
}
};
csl_memory_map mmap{
mpage_1 mpage_1;
mmap(){}
};
```

VERILOG CODE

```
`define MEM_MAP1_MAX_LENGTH 8;
`define MEM_MAP2_MAX_LENGTH 8;
```

csl_memory_map memory_map_name;
This command declares an object of type memory map, named *memory_map_name*.

[CSL Memory Map Command Summary]

EXAMPLE :
Create a memory_map named *mmap*.

FIGURE 2.18 A memory map named mmap

CSL CODE

```
//AV
//creates a memory map with the name mmap;
csl_memory_map mmap{
};
```

VERILOG CODE

```
//AV
```

auto_gen_memory_map();

DESCRIPTION :

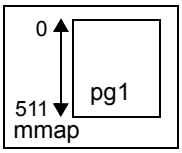
This command is used to generate automatic the memory map for the all memory elements and unit instances.

[CSL Memory Map Command Summary]

EXAMPLE :

Generates automatic a memory map for a memory map page named *pg1*.

FIGURE 2.19



CSL CODE

```
csl_memory_map_page pg1{
pg1(){
add_address_range(0, 511);
}
};
csl_memory_map mmap{
pg1 pg1;
mmap(){
auto_gen_memory_map();
}
};
```

VERILOG CODE

```
set_top_unit(unit_object_name);
```

DESCRIPTION :

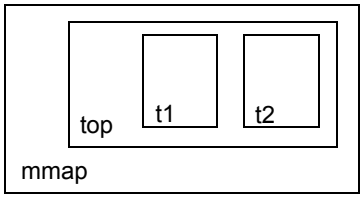
This command is used to set the top unit that has instances of other units.

[CSL Memory Map Command Summary]

EXAMPLE :

In this example we have three units named *top1* , *top2* and *top*. The unit *top* has instances of the *top1* and *top2* units named *t1* and *t2*. In a memory map named mmap is set the top unit using *set_top_unit* method.

FIGURE 2.20



CSL CODE

```
csl_unit top1{
top1(){}
};
csl_unit top2{
top2(){}
};
csl_unit top{
top1 t1;
top2 t2;
top(){}
};
csl_memory_map mmap{
mmap(){
set_top_unit(top);
}
};
```

VERILOG CODE

```
object_name.add_to_memory_map([address],[group,access_right]);
```

DESCRIPTION :

This method will add one object called *object_name* to the memory map. Optionaly can be set the address in the memory map where the object will be mapped and the access righit for this object. At least one parameter should exist.

TABLE 2.8 Acces Rights

ACCES RIGHTS	Description
access_none	None
access_read	Read
access_write	Write
access_read_write	Read/Write

[CSL Memory Map Command Summary]

EXAMPLE :

Adds a *fifo* named *f1* to a memory map named *mmap*.Fifo will be added to the address 32 and with access write.

CSL CODE

```
csl_fifo f1{
f1(){
set_width(32);
set_depth(128);
add_to_memory_map(32, SWR, access_write);}
};
csl_memory_map mmap{
mmap(){ }
};
```

VERILOG CODE

```
`define DEFAULT_GROUP_user 0
`define DEFAULT_GROUP_swr 1
`define DEFAULT_GROUP_hwr 2
`define DEFAULT_GROUP_test 3
`define DEFAULT_GROUP_driver 4
module f1(push,
          pop,
          full,
          empty,
          data_out,
          data_in,
          reset_,
          clock,
```

```
        valid);
  parameter ADDR_WIDTH = 4'd8;
  parameter DATA_WIDTH = 6'd32;
  input push;
  input pop;
  input [31:0] data_in;
  input reset_;
  input clock;
  output reg full;
  output reg empty;
  output reg [31:0] data_out;
  output reg valid;
  reg [ADDR_WIDTH - 1:0] wr_addr;
  reg [ADDR_WIDTH - 1:0] rd_addr;
  reg wr_en;
  reg rd_en;
  assign   full = wr_addr + 1 == rd_addr;
  assign   empty = wr_addr == rd_addr;
  assign   wr_en = !full && push;
  assign   rd_en = !empty && pop;
  f1_fifo_memory fifo_memory_instance(.clock(clock),
                                      .data_in(data_in),
                                      .data_out(data_out),
                                      .rd_addr(rd_addr),
                                      .rd_en(rd_en),
                                      .reset_(reset_),
                                      .valid(valid),
                                      .wr_addr(wr_addr),
                                      .wr_en(wr_en));
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      rd_addr <= 1'd0;
    end
    else       if ( pop )  begin
        rd_addr <= rd_addr + 1'd1;
      end
  end
  always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      wr_addr <= 1'd0;
```

```
    end
    else       if ( push )  begin
        wr_addr <= wr_addr + 1'd1;
      end
  end
endmodule
module f1_fifo_memory(clock,
                      reset_,
                      data_in,
                      data_out,
                      valid,
                      wr_addr,
                      rd_addr,
                      wr_en,
                      rd_en);
  parameter ADDR_WIDTH = 4'd8;
  parameter DATA_WIDTH = 6'd32;
  parameter NUM_WORDS = (1'd1 << DATA_WIDTH);
  input clock;
  input reset_;
  input [DATA_WIDTH - 1:0] data_in;
  input [ADDR_WIDTH - 1:0] wr_addr;
  input [ADDR_WIDTH - 1:0] rd_addr;
  input wr_en;
  input rd_en;
  output reg [DATA_WIDTH - 1:0] data_out;
  output reg valid;
  reg [DATA_WIDTH - 1:0] internal_memory[1'd0:NUM_WORDS - 1'd1] ;
always @( posedge clock or negedge reset_ )  begin
    if ( ~reset_ )  begin
      valid <= 1'd1;
    end
    else  begin
      valid <= rd_en;
      data_out <= internal_memory[rd_addr];
      if ( wr_en )  begin
        internal_memory[wr_addr] <= data_in;
      end
    end
  end
```

```
endmodule
```

```
set_type(memory_map_type);
```

DESCRIPTION :

Sets the type for a memory map. The `memory_map_type` can be one of the following:

TABLE 2.9

Memory map type
flat
hierarchical
virtual_with_page_number_and_address
virtual_with_base_address

[CSL Memory Map Command Summary]

EXAMPLE :

In this example is set the type of a memory map *named* mmap wihch contains the instances of two memory map pages named *map0* and *map1*.

CSL CODE

```
csl_memory_map_page mpage_0{
mpage_0(){
add_address_range(64,511);
set_address_increment(2);
}
};
csl_memory_map_page mpage_1{
mpage_1(){
add_address_range(0,63);
set_address_increment(1);
}
};
csl_memory_map mmap{
mpage_0 map0;
mpage_1 map1;
mmap(){
set_data_word_width(32);
set_type(hierarchical);
}
};
```

VERILOG CODE

memory_map_page_name.**set_access_rights_enum(***enum_name***);**

DESCRIPTION :

Sets the enum *enum_name* for access memory map named *memory_map_page_name.*

[CSL Memory Map Command Summary]

EXAMPLE :

The example shows the way to set a enum named *acc_enum* for access rights to a memory map named *mmap.*

CSL CODE

```
csl_enum acc_enum{
USR0,
USR1,
USR2
};
csl_memory_map mmap{
mmap(){
set_data_word_width(32);
set_access_rights_enum(acc_enum);
}
};
```

set_data_word_width(*numeric_expression***);**

DESCRIPTION :

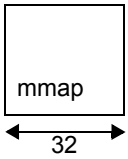
This command sets the data word width for a memory map.

[CSL Memory Map Command Summary]

EXAMPLE :

Sets the data_word_width to 32 in amemory map named *mmap.*

FIGURE 2.21



CSL CODE:

```
csl_memory_map mmap{
mmap(){
set_data_word_width(32);
}
};
```

VERILOG CODE

```
//
```

```
int get_data_word_width();
```

DESCRIPTION :

This command returns the data word width of a memory map or memory map page. Return type can be integer, hex or octal number.

[CSL Memory Map Command Summary]

EXAMPLE :

Sets data word width for a memory map named *mmap*, using the *get_data_word_width()* method.

CSL CODE

```
csl_memory_map_page mpage0{
mpage0(){
add_address_range(0,255);
set_data_word_width(32);
}
};
csl_memory_map mmap{
mpage0 mp0;
mmap(){
set_data_word_width(mp0.get_data_word_width());}
};
```

VERILOG CODE

```
//
```

set_prefix(*string***);**

DESCRIPTION :

Applies *string* as a prefix to all names in the memory map. Acts as a global prefix to the current scope.

[CSL Memory Map Command Summary]

EXAMPLE :

In a memory map *mmap* which contains a memory map page *mpage* is set the prefix "*mem*".

CSL CODE

```
csl_memory_map_page mpage{
mpage(){
add_address_range(0,1023);
set_address_increment(1);
}
};
csl_memory_map mmap{
mpage mpage0;
 mmap(){
set_data_word_width(32);
set_prefix("mem");
}
};
```

VERILOG CODE

```
`define <MMN>_PREFIX name;
```

```
string get_prefix();
```

DESCRIPTION :

Returns the current prefix set to a memory map.

[CSL Memory Map Command Summary]

EXAMPLE :

Sets the prefix “mem” to a memory map *mmap*, using *get_prefix()* method.

CSL CODE

```
csl_memory_map_page mpage_0{
mpage_0{
set_prefix("mem");
}
};
csl_memory_map mmap{
mpage_0 mpage_0;
mmap{
set_prefix(mpage_0.get_prefix());
}
};
```

VERILOG CODE

```
`define <MMN>_PREFIX name;
```

```
set_sufix(string);
```

DESCRIPTION :

Applies *string* as a sufix to all names in the memory map. Acts as a global sufix to the current scope.

[CSL Memory Map Command Summary]

EXAMPLE :

In a memory map *mmap* which contains a memory map page *mpage* is set the sufix "*mem*".

CSL CODE

```
csl_memory_map_page mpage{
mpage(){
add_address_range(0,1023);
set_address_increment(1);
}
};
csl_memory_map mmap{
mpage mpage0;
 mmap(){
set_data_word_width(32);
set_sufix("mem");
}
};
```

VERILOG CODE

```
`define <MMN>_SUFIX name;
```

```
string get_sufix();
```

DESCRIPTION :

Returns the current sufix set to a memory map.

[CSL Memory Map Command Summary]

EXAMPLE :

Sets the sufix “mem” to a memory map *mmap*, using *get_sufix()* method.

CSL CODE

```
csl_memory_map mpage_0{
mmap_0{
set_sufix("mem");
}
};
csl_memory_map mmap{
mpage_0 mpage_0;
mmap{
set_sufix(mpage_0.get_sufix());
}
};
```

VERILOG CODE

```
`define <MMN>_SUFIX name;
```

2.2.1 Generated Code

2.2.1.1 Generated C++ Code !!turn this to H2

```
#ifndef __csl_I_<NAME>_VH_
#define __csl_I_<NAME>_VH_

//
// DO NOT EDIT - automatically generated by <toolname>!
//
// ------------------------------------------------------------------------
-------
//
// Copyright (c) <year>, <company name>
// All Rights Reserved.
//
// This is UNPUBLISHED PROPRIETARY SOURCE CODE of <company name>;
// the contents of this file may not be disclosed to third parties,
copied or
// duplicated in any form, in whole or in part, without the prior writ-
ten
// permission of <company name>.
//
// RESTRICTED RIGHTS LEGEND:
// Use, duplication or disclosure by the Government is subject to
restrictions
// as set forth in subdivision (c)(1)(ii) of the Rights in Technical
Data
// and Computer Software clause at DFARS 252.227-7013, and/or in simi-
lar or
// successor clauses in the FAR, DOD or NASA FAR Supplement. Unpub-
lished -
// rights reserved under the Copyright Laws of the United States.
//

// generated C++ section
// generated from toolname : <toolname>
// path to tool:           : <path>
// tool version:           : <version>
// time stamp for tool:    : <tool time stamp>
```

```
// generated from filename : <filename>
// source filename:        : <filename>
// source file timestamp:  : <source file time stamp>
// generated file timestamp: <current file time stamp>

// Register register_name
#define register_name_REGISTER_ADDRESS        0x<address>

// value to reset the entire register to

// the following two fields can be defined using the field reset and
set values.
#define register_name_REGISTER_RESET_VAL       0x<reset_value>
#define register_name_REGISTER_SET_VAL         0x<set_value>

// the shift value is equal to the LSB bit position of the field
#define register_name_field_name_SHIFT_AMOUNT   <shift_value>
#define register_name_field_name_MASK           <mask>

// use the following define to set the value of the field
#define register_name_field_name_SET_SHIFT_AND_MASK <mask> << <shift>

// use the following define to get the value of the field
#define register_name_field_name_GET_SHIFT_AND_MASK <mask> >> <shift>

#define register_name_field_name_BITRANGE
<msb_bit_position>:<lsb_bit_position>
#define register_name_field_name_INIT_VAL  0x<field_init_value>
#define register_name_field_name_SET_VAL   0x<field_set_value>
#define memory_map_name_END_ADDRESS          <address>
```

2.2.1.2 Generated Verilog Code

```
#ifndef __csl_I_<NAME>_VH_
#define __csl_I_<NAME>_VH_

//
// Generated by <toolname>
// DO NOT MODIFY
```

```
// ------------------------------------------------------------------
-------
//
// Copyright (c) <year>, <company name>
// All Rights Reserved.
//
// This is UNPUBLISHED PROPRIETARY SOURCE CODE of <company name>;
// the contents of this file may not be disclosed to third parties,
copied or
// duplicated in any form, in whole or in part, without the prior writ-
ten
// permission of <company name>.
//
// RESTRICTED RIGHTS LEGEND:
// Use, duplication or disclosure by the Government is subject to
restrictions
// as set forth in subdivision (c)(1)(ii) of the Rights in Technical
Data
// and Computer Software clause at DFARS 252.227-7013, and/or in simi-
lar or
// successor clauses in the FAR, DOD or NASA FAR Supplement. Unpub-
lished -
// rights reserved under the Copyright Laws of the United States.
//

// Generated verilog section
// generated from toolname : <toolname>
// path to tool:             <path>
// tool version:             <version>
// time stamp for tool:      <tool time stamp>
// generated from filename:  <filename>
// source filename:          <filename>
// source file timestamp:    <source file time stamp>
// generated file timestamp: <current file time stamp>

#define <name>_WIDTH16
#define <name>_RANGE15:0
#define <name>_ADDR0

// Register <reg_name>
```

```
#define <reg_name>_WIDTH8
#define <reg_name>_RANGE7:0
#define <reg_name> 32'h0
#define <reg_name>_RESET_NUM8'bxxxxxxxx
#define <reg_name>_INIT_NUM8'h0

//fields belonging to the above register
// there are n fields which in total width can equal but not exceed the
width of the above //register definition
#define <reg_name>_field_name_WIDTH<field_width>
#define <reg_name>_field_name_RANGE<field_range>
#define <reg_name>_field_name_RW<r=2, rw=3> // 10 and 11
#define <reg_name>_field_name_NUM  <field_width>'h<value>  // devulat
for
//<value> is 0

Example:

// Register register_name
#define register_name_ADDRESS 32'h<address>
#define register_name_RESET_VALUE 2'b<value>
#define register_name_SET_VALUE 3'h<value>
#define register_name_BITRANGE [<msb_bit_position>:<lsb_bit_position>]
#define register_name_REGISTER_WIDTH <width>
#define register_name_field_name_BITRANGE
#define register_name_field_name_field_WIDTH 1
#define register_name_field_name_ATTR
#define register_name_field_name_DEFAULT 1'h0
#define BASE_ADDRESS_<module_name>              <address>


class register_name : public register {
  register_name_ADDRESS 32'h<address>
  register_name_RESET_VALUE 2'b<value>
  register_name_SET_VALUE 3'h<value>
  register_name_BITRANGE [<msb_bit_position>:<lsb_bit_position>]
 field_name register_name_REGISTER_WIDTH <width>
  register_name_field_name_BITRANGE
  register_name_field_name_field_WIDTH 1
  register_name_field_name_ATTR
```

```
  register_name_field_name_DEFAULT 1'h0
  BASE_ADDRESS_<module_name>              <address>
}


#endif __csl_I_<NAME>_VH_
```

2.2.2 Generated code

2.2.2.1 Generated C++ Code

```
#ifndef_csl_1_<NAME>_VH_
#define_csl_1_<NAME>_VH_
// DO NOT EDIT =automatically generated by <toolname>!
//
// ----------------------------------------------------------------------
---
//
//Copyright (c) <year><company name>
// All Rights Reserved
//
//This is UNPUBLISHED PROPRIETARY SOURCE CODE of <company name>;
//the contents of this file may not be disclosed to third parties, cop-
ied or duplicated in any form, in whole or in part, without the prior
written permission of <company name>
//
//RESTRICTED RIGHTS LEGEND:
// Use, dulpication or disclosure by the Government is subject to
restrictions as se
//forth in subdivision (c)(1)(ii) of the Rights in Technical Data and
Computer Soft
//ware clause at DFARS 252.227-7013, and/or in similar or succesor
clauses in the
//FAR, DOD or NASA FAR Supplement. Unpublished  rights reserved under
the
//Copyright Laws of the United States
//
#Generated C++ section
#toolname: <toolname>
#path to tool: <path>
```

```
#tool version: <version>
#time stamp for tool: <tool time stamp>
#generated from filename: <filename>
##file timestamp <source file timestamp>
#generated timestamp <current file time stamp>
//Register STATUS_0
#define STATUS_0            .0xc
#define STATUS_0_RESET_NUM 0x0
#define STATUS_0_BSY_SHIFT 7
#define STATUS_0_BSY_FIELD (0x1<<STATUS_0_BSY_SHIFT)
#define STATUS_0_BSY_RANGE 7:7
#define STATUS_0_BSY_DEFAULT 0x0
#define STATUS_0_DRDY_SHIFT 6
#define STATUS_0_DRDY_FIELD (0x1<<STATUS_0_DRDY_SHIFT)
#define STATUS_0_DRDY_RANGE 6:6
#define STATUS_0_DRDY_DEFAULT 0x0
#define STATUS_0_DRQ_SHIFT 3
#define STATUS_0_DRQ_FIELD (0x1<<STATUS_0_DRQ_SHIFT)
#define STATUS_0_DRQ_RANGE 3:3
#define STATUS_0_DRQ_DEFAULT 0x0
#define STATUS_0_ERR_SHIFT 0
#define STATUS_0_ERR_FIELD (0x1<<STATUS_0_ERR_SHIFT)
#define STATUS_0_ERR_RANGE 0:0
#define STATUS_0_ERR_DEFAULT 0x0
#define CEATA0_LAST_REG STATUS_0//0x000d
```

2.2.2.2 Generated Verilog Code

```
#ifndef_csl_|_<NAME>_VH_
#define_csl_|_<NAME>_VH
//DO  NOT EDIT -automatically generated by <toolname>!
// ----------------------------------------------------------------------
--
///
//Copyright (c) <year><company name>
//All Rights Reserved
//
//this is UNPUBLISHED PROPRIETARY SOURCE CODE pf <company name>;
//the contents of this file may not be disclosed to third parties, cop-
ied or duplicated in any form, in whole or in part, without the prior
written permission of <company name>
```

```
//RESTRICTED RIGHTS LEGEND:
// Use, dulpication or disclosure by the Government is subject to
restrictions as se
//forth in subdivision (c)(1)(ii) of the Rights in Technical Data and
Computer Soft
//ware clause at DFARS 252.227-7013, and/or in similar or succesor
clauses in the
//FAR, DOD or NASA FAR Supplement. Unpublished  rights reserved under
the
//Copyright Laws of the United States
#Generated verilog section
#toolname : <toolname>
#path to tool <path>
#tool version : <version>
#time stamp for tool: <tool time stamp>
#generated from filename : <filename>
#file timestamp <source file time stamp>
#generated timestamp <current file time stamp>
#define <name>_WIDTH16
#define <name>_RANGE 15:0
#define <name>_ADDR0
//register <reg_name>_0
#define <reg_name>_0_WIDTH8
#define <reg_name>_0_RANGE 7:0
#define <reg_name>_0 32'h0
#define <reg_name>_0_RESET_NUM8'bxxxxxxxx
#define <reg_name>_0_INIT_NUM8'h0
//fields belonging to the above register
//there are n fields which in total can equal ubt not exceeded the
width of the above register definition
#define <reg_name>_0_field_name_WIDTH<field_width>
#define <reg_name>_0_field_name_RANGE<field_range>
#define <reg_name>_0_field_name_RW<r=2,rw=3>//10 and 11
#define <reg_name>_0_field_name_NUM <field_width>'h<value>//devulat
for <value>
```

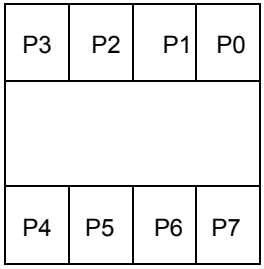
Example:

```
//register register_name_0
#define register_name_032'h5
#define register_name_0_RESET_NUM2'bxx
#define register_name_0_INIT_NUM3'h0
```

```
#define register_name_0_RANGE2:1
#define register_name_0_WIDTH2
#define register_name_0_field_name_RANGE2
#define register_name_0_field_name_WIDTH1
#define register_name_0_field_name_RW3
#define register_name_0_field_name_DEFAULT'h0
#define register_name_0_field_name_RANGE1
#define register_name_0_field_name_WIDTH1
#define register_name_0_field_name_RW3
#define register_name_0_field_name_DEFAULT1'h0
#deine BASE_ADDRESS_MODULE32'h00000000
#endif_csl_I_<NAME>_VH_
```

NOTE:<Move this to commands examples>

FIGURE 2.22

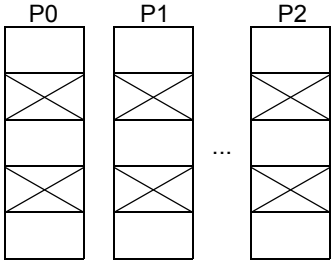


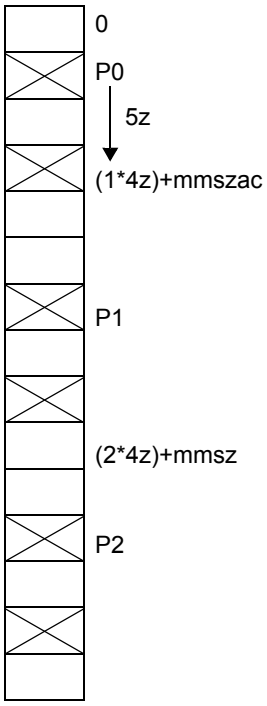

```
csl_memory_map mmn;
csl_unit p[0-7];
p[0-7].set_range(mmn, 0, 29);
set_address(mmn, \1.getadde_size()*\2);
```

- \1=p[0-7]
- \2=[0-7]

```
default address= lastobject.baseaddress()+lastob-
ject.addr_size()+mmn.inc_amounts
p[0-7].add_to_memory_map(mmn);
user next address which is equal to mmn.inc_amounts
```

FIGURE 2.23 Individual processors memory spaces and the combined memory map shrink figure





each processor address space starts at an offset
</Move this to commands examples>
<ADD>
move this ADD to sw components
Consumer Electronic Chips

FIGURE 2.24

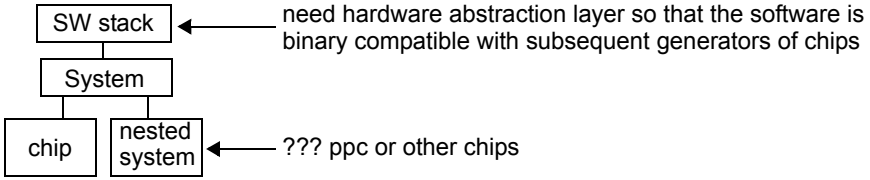
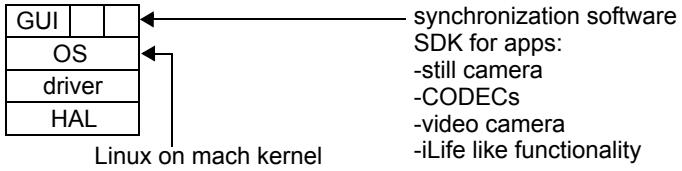


FIGURE 2.25



</ADD>

<ADD>

Code generation - move to code gen doc
!also move the generated code abvoe

<memory_map_class_name>::enum<register_name>_<field_name_in_register>_<enum _name_in_field>

All letters are capitilized except the enum.

Register output abreviations

mme = memory map element
fld = field in a register
enum = enumerated type value for a given field

C/C++ classes will be generated with a prefix letter "C".
C/C++ enumerated types will be generated with a prefix letter "c" (i.e. enum cenum<enumerated_type_name> {...}).

Verilog code will be generated with a prefix letter "v"
Verilog defines which are equivalent to the C/C++ enumerated will be generated with a prefix letter "cv" (i.e. `define venum<enumerated_type_name> <value>).

```
enumerated types should have an illegal field which can be returned
from C/C++ switch default cass and from Verilog cas statement
default cases. The illegal field can be "caught" by the "down-
stream" logic and can flag problems with switch and case statement
selector inputs.
```
</ADD>

Virtual Memory

SW address map is global.
HW address map is local.
Upper bits are the page ID.
Upper bits map to a unit ID.

TABLE 2.10 Virtual memory table

upper bits	global	local
0	m	0
0	n	(n-m)
1	p	0
1	q	(q-p)
2	b	0
2	c	(c-b)
3	d	0
3	e	e-d

FIGURE 2.26

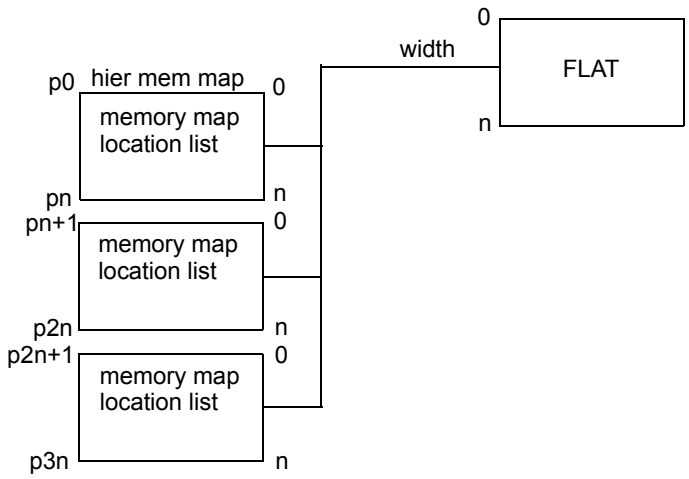


TABLE 2.11

	global	local
flat	x,y	x, y
	psa+m, psa+n address	m, n
hier	x, y	x, y
v m	sa=(pno<<amount) ea=(pno<<amount)	1.x 1.y x,y

VM base	VMPN in Addr
000000	0000000 = (0<<20) | 0
100000	0100000 = (1<<20) | 0000
200000	0200000 = (2<<20) | 0000
20 bits global	28 bit global

<ADDED_2007.05.12>

```
Different methods used for  programming chip registers

Chips contain registers which need to be configured with values
```

The register values also need to be read out to a different unit ont he chip or outside of the chip.

CSL provides a way to write a set of registers on a chip using one or 4 different
physical bus/network topologies. The

All buses contain essentially the same set of commands.
addr (address)
data
v (valid)
cmd (command)

All buses/networks are connected to the controller and all leaf level units.
In the caes of the tree network there may be intermediate nodes which are used to
gather information from a cluster of units and for timing reasons.

1. In band SOC bus
2a. Out of band network tree
2b. Out of band network Ring
3. In band pipeline

==========

1. In band SOC bus
Each bus master waits for a slot on the bus and then sends a bus command to
another unit on the bus. All units "listen" to the bus for bus commands addressed to the unit.

2a. Out of band network tree
The Out of band network tree has both a send and a receive network
The send network contains the following of signals:
addr - data
data - address
v - valid
cmd - command
The send network is used to send data and commands to the leaf level units.

```
The leaf level units execute the commands and if requested send a reply
via the
reply tree to the controller.
The controller broadcasts messages to all units which match the uid in
the message and then
execute the command.

2b. Out of band network Ring
The Out of band network ring connects all units in a ring topology.
The ring contains the following of signals:
addr - data
data - address
v    - valid
cmd  - command

3. In band pipeline
Each pipestage can contain one or more registers which can be read/
written via packets sent down the
command pipeline. The command pipelne packets contain the following
signals.
addr - data
data - address
v    - valid
cmd  - command
When the address in the pipestage address signal matches an address in
the pipestage and the valid is
'1' then the command is exectued and a register is either read or writ-
ten.

================================================================================
==========

// note that in the memory map b elow we do not set the data word width
or the address word
width. The clsc will determine the address word width based on the
address range (start and end
addresses) for the memory map.

csl_memory_map mem_map {
  csl_memory_map_page unit_a;
  mem_map () {
```

```
    set_type(VM_WITH_ADDRESS);
    unit_a.set_range(0, 65767);
  }
}

csl_enum bus_cmd {
  BUS_CMD_RD,
  BUS_CMD_WR,
  BUS_CMD_PING,
  BUS_CMD_NOP
};

// create a bus with signal names that match the pin names on the reg-
isters that the bus
// is logically connected to. There are intermediate units whiuch the
bus is connected to
// for timing and distribution reasons. The units that the bus is con-
nected to have a bus
// interface unit (BIU) that the bus is connected to. The BIU detects
commands that are
// intended for the unit and converts the commands into local control,
address, and data
//
//
//
//

csl_interface reply_bus {
  csl_port data(input,32),
           addr(input, mem_map.get_address_word_width()), // 9 bits
since log2(512) = 9
           v   (input  ) ;
};

csl_interface cmd_bus : reply_bus {
  csl port cmd(input,2);
  ifc(){
    cmd.add_enum(bus_cmd);
  }
};
```

```
csl_unit controller {
  cmd_bus bus_out;
  reply_bus bus_in;
  controller(){
    bus_out.reverse();
  }
};


csl_register_group unit_a_rg {
  csl_register r[[0-31]](32); // create 32 32-bit registers

  unit_a_rg() {

  }
};

csl_unit a{
  cmd_bus   bus_in;
  reply_bus bus_out;
  unit_a_rg unit_a_rg0;
  int unit_a_mem_map_base_addr;

  a() {
    unit_a_mem_map_base_addr = 2048;
    bus_out.reverse();
    mem_map.unit_a.add(unit_a_rg0, "unit_regs",
unit_a_mem_map_base_addr);
    unit_a_rg0.use_biu_to_write();

// unit_a_rg0 has the same interface as bus_in so they can be connected
// each register has a set of pins that match the signal names and
directions
// in the bus.
// however the bus_in and the bus_out are not directly connected to the
registers
// instead intermediate logic is created to write the registers.
//
// The cslc detects that each register in the register group unit_a_rg0
are in the memory
```

```
// map. Since all registers in the memmory map they need to be con-
nected to the the unit_a
// BIU (bus interface unit ) which listens to the bus as described
above and generates the
// write enable (wr_en) signals for each individual register.
//
// The interface bus_out is no directly connected to the register out-
puts. Instead the register
// outputs are connected to a mux and the bus_in_addr selects the reg-
ister to send back to the
// controller which sent the read command to unit_a.
//
// If not all registers are in the memory map generate a compiler
error.

    unit_a_rg0.connect(bus_in);
    bus_in.connect(unit_a_rg0);

    csl_signal a_en =
    reg_0.d = data;
    mem_map.set_unit_address_signal(a,addr);
  }
};


csl_unit top{
  a a0;
  controller cntl0;
  top(){
    a0.set_instance_id(3);

  }
};
```

OLD CSL CODE

```
csl_memory_map mem_map;

csl_enum bus_cmd {
  BUS_CMD_RD,
  BUS_CMD_WR,
```

```
  BUS_CMD_PING,
  BUS_CMD_NOP
};

csl_interface reply_bus {
  csl_port data(input,32),
           addr(input,5) ,
           v(input)      ;
};

csl_interface cmd_bus : reply_bus {
  csl port cmd(input,2);
  ifc(){
    cmd.add_enum(bus_cmd);
  }
};

csl_unit controller {
  cmd_bus bus_out;
  reply_bus bus_in;
  controller(){
    bus_out.reverse();
  }
};

csl_unit a{
  cmd_bus   bus_in;
  reply_bus bus_out;
  csl_register reg_0(32);
  a(){
   bus_out.reverse();
   reg_0.
   csl_signal a_en =
   reg_0.d = data;
  }
};
mem_map.add_logic(object_wr_en, address)
mem_map.add_object(a.reg_0)
mem_map.set_unit_address_signal(a,addr);
```

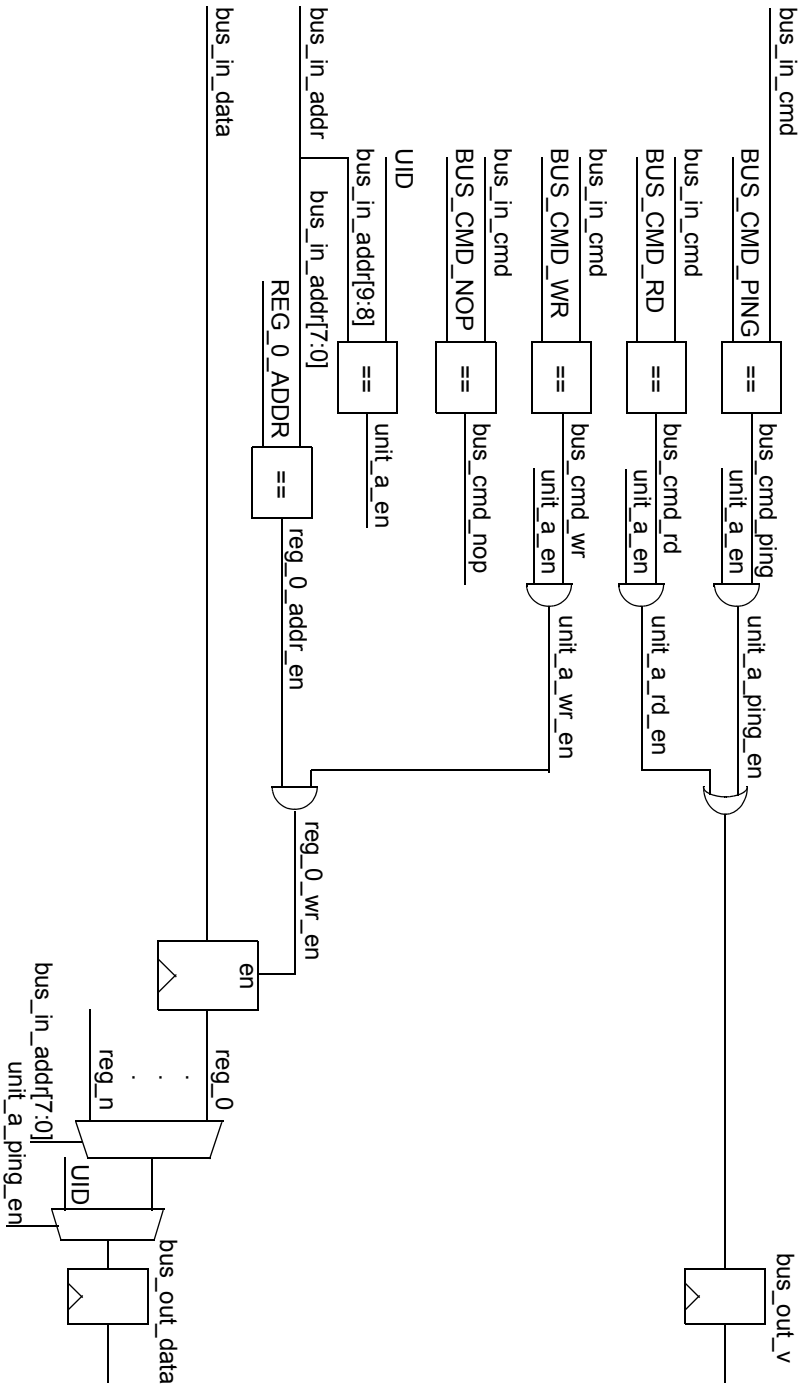
```
csl_unit top{
  a a0;
  top(){
    a0.set_instance_id();
  }
};
```

VERILOG CODE

```
`define BUS_CMD_RD     0
`define BUS_CMD_WR     1
`define BUS_CMD_PING   2
`define BUS_CMD_NOP    3
`define UID            3 //unit id
`define REG_0_ADDR   128 //reg_0 address

module a(bus_in_data,
         bus_in_addr,
         bus_in_cmd ,
         clk,
         bus_out_data,
         bus_out_v
         );

 input [31:0] bus_in_data;
 input [9:0]  bus_in_addr;
 input [1:0]  bus_in_cmd;
 input clk;

 output bus_out_v;
 reg bus_out_v;
 output [31:0] bus_out_data;
 reg [31:0] bus_out_data;

 //local signals
 reg [31:0] reg_0;

 wire bus_cmd_rd   = (`BUS_CMD_RD   == bus_in_cmd)  ;
 wire bus_cmd_wr   = (`BUS_CMD_WR   == bus_in_cmd)  ;
 wire bus_cmd_ping = (`BUS_CMD_PING == bus_in_cmd);
 wire bus_cmd_nop  = (`BUS_CMD_NOP  == bus_in_cmd) ;
```

```
 wire unit_a_en     = bus_in_addr[9:8] == `UID;
 wire unit_a_wr_en  = unit_a_en && bus_cmd_wr;
 wire unit_a_rd_en  = unit_a_en && bus_cmd_rd;
 wire unit_a_ping_en= unit_a_en && bus_cmd_ping;

 wire reg_0_addr_en = bus_in_addr[7:0] == `REG_0_ADDR;
 wire reg_0_wr_en   = unit_a_wr_en && reg_0_addr_en;

 always @(posedge clk) begin
   if(reg_0_wr_en) begin
     reg_0 <= bus_in_data;
   end
 end

always @(posedge clk) begin
    bus_out_v <= unit_a_rd_en & unit_a_ping_en;
end


always @(posedge clk) begin
    if(unit_a_rd_en) begin
  case (bus_in_addr)
  `REG_0_ADDR: bus_out_data <= reg_0;
  endcase
  end
  else if(unit_a_ping_en) begin
      bus_out_data = `UID;
  end
end
endmodule
```

FIGURE 2.27 Bus Interface Unit Command Decoder



</ADDED_2007.05.12>

<ADDED ON 2007.05.16>

NOTE:UPDATE COMMAND SUMMARY ACCORDING TO THIS

Note: There should be 8 examples from :

TABLE 2.12

Type	User defined mem map	automatic mem map
flat	x	x
hierarchical	x	x
virtual with page number and address	x	x
virtual with base address	x	x

0

1

2

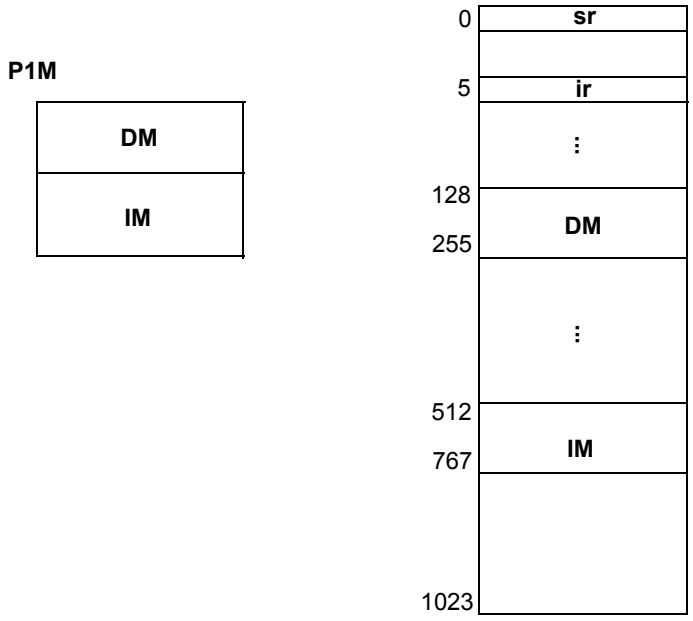
User defined example:

```
csl_unit processor {
   ...
};

csl_unit cluster {
  processor p[[0-7]];
};

csl_unit chip {
  cluster c[[0-7]];
};

csl_memory_map_page mproc {
  mproc(){
    set_unit(processor);
  }
};

csl_memory_map_page mcluster {
  mproc mp[[0-7]](p[[0-7]]); //user specified
  mcluster(){
    set_unit(cluster);
  }
};

csl_memory_map_page mchip {
  mcluster mc[0-7]](c[[0-7]]); //user specified
  mchip(){
    set_unit(chip);
  }
};

csl_memory_map mmap {
  mchip mchip;
  mmap(){
  set_type(hierarchical);
  }
}
```

Automatic example:

```
csl_unit processor {
   ...
};

csl_unit cluster {
  processor p[[0-7]];
};

csl_unit chip {
  cluster c[[0-7]];
};

csl_memory_map_page mproc {
  mproc(){
    set_unit(processor);
  }
};

csl_memory_map mmap {
   mmap(){
  set_top_unit(chip);
  set_type(hierarchical);
  use_instance_decl_order();
  }
}

//This generates the same code as the user defined version above
```

EXAMPLE :




```
csl_register sr;
csl_register ir;

csl_memory im(16,128);
csl_memory dm(16,256);

csl_unit p{
  im im();
  dm dm();

  sr sr();
  ir ir();
  p(){
     sr.add_to_mem_map();
     ir.add_to_mem_map(5);        // insert at address 5
     dm.add_to_mem_map(128);      // insert at address 128
```

```
    im.add_to_mem_map(512);      // insert at address 512
  }

csl_memory_page mp{
  mp(){
    set_unit(p);
  }

csl_memory_map mm{
  mp mp;
  mm(){
    set_top_unit(chip);
    set_type(hierarchical);
    autogen_mem_map;
  }
}

csl_unit cl{
  p p[[0-7]];
}


csl_unit chip{
  cl cl[[0-7]];
  chip(){
  }
}
```

Generated header file:

```
#define sr              0x000
#define ir              0x005
#define mp_start_addr   0x000
#define mp_end_addr     0x3FF
#define dm_start_addr   0x080
#define dm_end_addr     0x0FF
#define im_start_addr   0x200
#define im_end_addr     0x2FF
```

When generating the memory map acces rights and visibility will be specified as parameters to generate only those defines that correspond to that specific options.

The adaptor needs to know how to connect the pins objects in the memory map to the network which will read/write the objects in the memory map.
Flat memory will need - data, address, command (W/R) and valid.
All units will listen to the address bus and they will need an address range checker (optional).

For hierarchical memory maps there will be a tree of enable signals to select the unit .
ex: chip enable + cluster enable + processor enable

Virtual with unit ID and address
The upper bits of the address bus will be used to identify the unit (unit ID), the lower bits will be used to address local memory in the selected unit.

Virtual memory with base address ?

CHAPTER 1 CSL Memory

TABLE 1.1 Chapter Outline

1.1 CSL Memory Command Summary

CSL Memory

```
csl_memory memory_name(width,depth);
initialize_random_values(seed_value);
initialize_random(value);
set_num_rd_ifc(numeric expression);
set_num_wr_ifc(numeric expression);
add_logic(reset);
add_logic(clock);
```

1.2 CSL Memory Commands

CSL Memory

csl_memory memory_name(width,depth);

DESCRIPTION :

Creates a memory named *memory_name* **.** The param width and depth there are numeric expresion and are mandatory for memory.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

//

VERILOG CODE

//

```
initialize(filename);
```

DESCRIPTION :

Use the values in the file to initialize a memory in the C++ simulator or a DUT memory instance in the test bench.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

```
//
```

VERILOG CODE

```
//
```

initialize_random_values(seed_value);

DESCRIPTION :

Initialize each memory location in a memory with random values in the C++ simulator or a DUT memory instance in the test bench. Use the *seed_value* to initialize the random number generator.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

//

VERILOG CODE

//

initialize_random(value);

DESCRIPTION :

Use the same value to initialize each memory location in the memory in the C++ simulator or a DUT memory instance in the test bench.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

//

VERILOG CODE

//

set_num_rd_ifc(numeric expression);

DESCRIPTION :

Sets the number of the read interfaces generated in the memory unit.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

```
//
```

VERILOG CODE

```
//
```

set_num_wr_ifc(numeric expression);

DESCRIPTION :

Sets the number of the write interfaces generated in the memory unit.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

```
//
```

VERILOG CODE

```
//
```

```
add_logic(reset);
```

DESCRIPTION :

This is an reset command. When the reset signal is on the low level (logic “0“) the memory is filled up with 0 values, the clock signal edge doesn’t matter.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

/ /

VERILOG CODE

/ /

```
add_logic(clock);
```

DESCRIPTION :

Adds a clock signal to a memory.

[CSL Memory Command Summary]

EXAMPLE :

//

CSL CODE

//

VERILOG CODE

//

CHAPTER 1 CSL Instruction Set Architecture Generator

TABLE 1.1 Chapter Outline

1.1 CSL ISA Syntax and Command Summary
1.2 CSL ISA Commands

1.1 CSL ISA Syntax and Command Summary

1.1.1 ISA Classes

CSL Isa classes are used to generate Instruction Set Architectures (ISA). In order to achieve this, certain steps need to be followed and understood. In CSL two types of classes are used when defining an ISA: CSL Isa element and CSL Isa field. These classes will form a tree-like structure that will generate the ISA.

1.1.1.1 CSL ISA Field class

An instruction is formed of ranges which hold various parts of the instruction (opcode, source/destination addresses, flags, etc..). In CSL these ranges are represented by a variation of the CSL Field class: CSL Isa Field.

1.1.1.1.1 CSL ISA Field class declaration

CSL Isa Field are an exception from the usual CSL classes in the way they are declared: CSL Isa Fields have a dual object/class declaration mode. The reason for this is because most of the time Isa Fields will be declared just like a bitrange object and a limited set of commands will be called upon (eg. associating an enum). In this case, the declaration appears as the one for a CSL object with optional parameters as explained in each case below:

CSL Isa Field object declaration

```
csl_isa_field isa_field_object(width[,enum_or_enum_item]);
csl_isa_field isa_field_object(field_object_name);
csl_isa_field isa_field_object(lower, upper[,enum_or_enum_item]);
```

NOTE: I see no reason the keep this constructor especially since there is no set_width for isa field + why would anyone create an isa field with noting (not even width)

```
csl_isa_field isa_field_object_name;
```

Sometimes however, the user may require hierarchical Isa Fields. In this case, these are declared in the global scope just like any other CSL class. The Isa Field class declaration is shown in the below example:

```
csl_isa_field isa_field_name {
  (objects declarations/instantiations)+
  isa_field_name(){
    (isa_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.

In the ISA Field class' scope other fields can be instantiated or declared. ISA Fields are declared in another ISA Field using the object declaration syntax. Hierarchical fields can only be instantiated in another hierarchical field.

TABLE 1.2 Rules for instantiating objects in an ISA Field's scope

CSL class	Is instantiated in CSL Isa Fields scope
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	YES
CSL Field	-

1.1.1.1.2 CSL ISA Field usage and rules

Isa Fields are instantiated in Isa elements as discussed later. Isa Fields can also be instantiated in hierarchical Isa Fields. Table 1.3 shows the usage of Isa Fields related to other CSL classes:

TABLE 1.3 Isa Field usage rules

CSL class	Uses CSL Isa Field
CSL Unit	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-

TABLE 1.3 Isa Field usage rules

CSL class	Uses CSL Isa Field
CSL Isa Element	YES
CSL Isa Field	YES
CSL Field	-

1.1.1.1.3 CSL Isa Field class commands

The following commands can be called on both Isa Field class (inside the constructor) and on Isa Fields declared using an object syntax.

CSL Isa Field class commands

```
[isa_field_obj.]set_type(isa_field_type);
[isa_field_obj.]set_name(string);
isa_field_obj_name.set_offset(num_expr);
isa_field_obj_name.set_enum(enum_name);
isa_field_obj_name.set_enum_item(enum_item_name);
```

```
isa_field_obj_name.set_mnemonic(string);

isa_field_obj_name.get_offset();
isa_field_obj_name.get_name();
isa_field_obj_name.get_mnemonic();
isa_field_obj_name.get_type();
isa_field_obj_name.get_lower();
isa_field_obj_name.get_upper();
isa_field_obj_name.get_width();


isa_field_obj_name.get_enum();
isa_field_obj_name.get_enum_item(enum_item_name);
isa_field_obj_name.set_value(num_expr);
isa_field_obj_name.get_value();
isa_field_obj_name.add_allowed_range(num_expr, num_expr);
gen_decoder(csl_unit unit_object_name);
```

1.1.2 Usage tables

CSL Isa element
can be defined and inherited
- can be defined in

TABLE 1.4 CSL Isa element definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be inherited from

TABLE 1.5 CSL Isa element inheritance rules

CSL class	Can be inherited from
global scope	-
CSL Unit	-
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-

TABLE 1.5 CSL Isa element inheritance rules

CSL class	Can be inherited from
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	YES
CSL Isa Field	-
CSL Field	-

CSL_ISA_FIELD METHODS

CSL_ISA_ELEMENT

```
csl_isa_element isa_obj_name [: isa_obj_name0];
```

CSL_ISA_ELEMENT METHODS

```
set_type (instr_format | instr | root_format );
set_width (numeric_expression);
set_position(isa_field, numeric_expression);
set_next(csl_isa_field left_field, csl_isa_field right_field);
set_previous(csl_isa_field right_field, csl_isa_field left_field);
gen_decoder(csl_unit unit_object_name);
```

1.2 CSL ISA Commands

CSL ISA INSTRUCTION FORMAT

csl_isa_field *isa_field_object_name;*

DESCRIPTION :

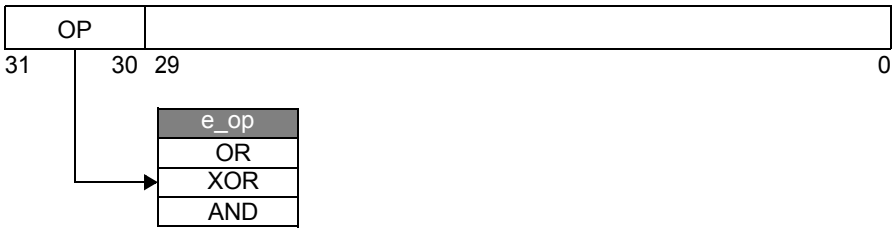
Creates a isa field *isa_field_object.*

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

In this example it was created a field object named *f_op.*

FIGURE 1.1



CSL CODE

```
csl_enum e_op{
 OR,
 XOR,
 AND
};
csl_isa_field f_op{
f_op(){
set_width(2);
set_enum(e_op);
set_type(opcode);
}
};
```

csl_isa_field *isa_field_object*(*width*[,*enum_or_enum_item*]);

DESCRIPTION :

Creates a isa field *isa_field_object.*

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

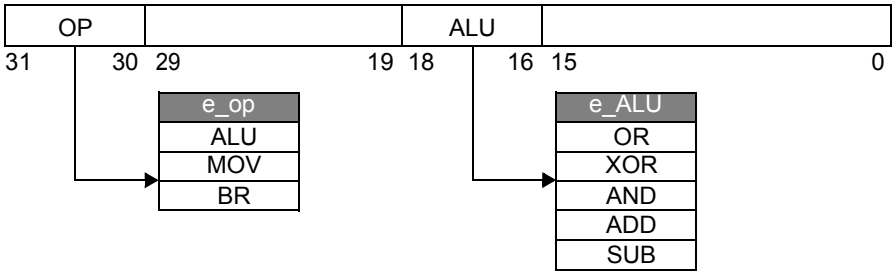
param enum_or_enum_item - the enum or the enum item will be associated to the field;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

In this example are created two isa fields *f_op* and *f_alu.*

FIGURE 1.2



CSL CODE

```
csl_enum e_op{
 ALU,
 MOV,
 BR
};
csl_enum e_ALU{
 OR,
 XOR,
 AND,
 ADD,
 SUB
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);
csl_isa_field f_alu(3,e_ALU);
f_alu.set_type(selector);
```

VERILOG CODE

csl_isa_field *isa_field_object*(*field_object_name*);

DESCRIPTION :

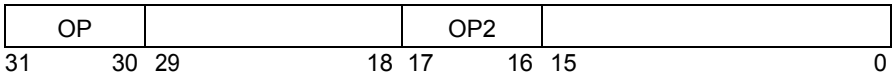
Create a isa field *isa_field_object* derived from another *csl_isa_field.*

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

The isa field *f_op2* is derived from isa field *f_op.*

FIGURE 1.3



CSL CODE

```
csl_enum e_op{
   ADD,
   XOR,
   SUB
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);
csl_isa_field f_op2(f_op);
```

VERILOG CODE

csl_isa_field *isa_field_object*(*lower, upper*[,*enum_or_enum_item*]);

DESCRIPTION :

Create an isa field *isa_field_object*

params lower and upper - range of the field

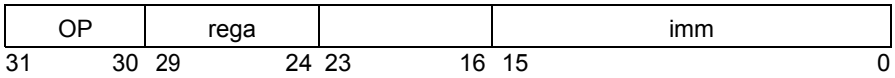
param enum_or_enum_item - the enum or the enum item will beassociated to the field;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

There are created three isa fields *f_op, f_rega, f_imm* using lower and upper range.

FIGURE 1.4



CSL CODE

```
csl_isa_field f_op(0,1);
f_op.set_type(opcode);

csl_isa_field f_rega(24,29);
f_rega.set_type(address);

csl_isa_field f_imm(0,15);
f_imm.set_type(constant);
```

VERILOG CODE

[*isa_field_obj_name*.]**set_type**(isa_field_type);

DESCRIPTION :

Set_type command is mandatory for a csl_isa_field. If set_type command is not set for a csl_isa_field then cslc will throw an error. An hierarchical isa_field can have as children fields only isa_fields. An hierarchical field can have as children fields only fields and not isa_fields;
The types an isa field can have are detailed in Table 1.6

TABLE 1.6 Isa field types

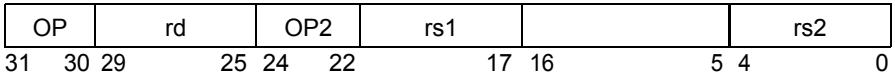
isa_field_type	Details
opcode	
subopcode	
address	
selector	
constant	
unused	
reserved	

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the types of five isa fields.

FIGURE 1.5



CSL CODE

```
csl_isa_field f_op(2);
f_op.set_type(opcode);

csl_isa_field f_op2(3);
f_op2.set_type(subopcode);

csl_isa_field f_rd(5);
f_rd.set_type(address);

csl_isa_field f_rs1(5);
f_rs1.set_type(address);

csl_isa_field f_rs2(5);
f_rs2.set_type(address);
```

VERILOG CODE

```
[isa_field_obj_name.]set_name(string);
```

DESCRIPTION :

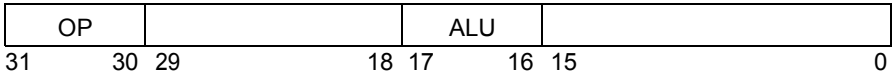
Sets the long name of the isa_field that will be used when generating the pdf docs, if this is not set the default value is the name of the field ;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the name “opcode” for the isa field *f_op*.

FIGURE 1.6



CSL CODE

```
csl_isa_field f_op{
f_op(){
set_name("opcode");
set_type(opcode);
set_width(2);
}
};
csl_isa_field f_alu(2);
f_alu.set_type(selector);
```

VERILOG CODE

isa_field_obj_name.**set_mnemonic**(string);

DESCRIPTION :

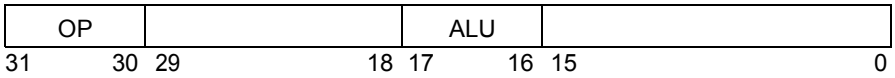
Sets the abrevation of the isa_field that will be used when generating the c++ code, if this is not set the default value is the enum item name in small letters.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the abrevation “alu” of the isa filed named *f_alu.*

FIGURE 1.7



CSL CODE

```
csl_isa_field f_op(2)
f_op.set_type(opcode);
csl_isa_field f_alu{
f_alu(){
set_type(selector);
set_mnemonic("alu");
set_width(2);
}
};
```

VERILOG CODE

```
isa_field_obj_name.set_offset(num_expr);
```

Description :

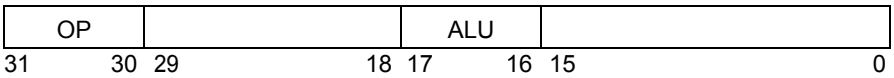
Set the value to be added to both lower and upper index of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the offset for the isa field *f_alu.*

FIGURE 1.8



CSL CODE

```
csl_isa_field f_op(2,e_op);
csl_isa_field f_alu{
f_alu(){
set_width(2);
set_type(opcode);
set_offset(2);
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_offset();
```

DESCRIPTION :

Returns the offset of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the offset for the isa filed f_op2 using the *get_offset()* method.

CSL CODE

```
csl_isa_field f_op{
f_op(){
set_width(2);
set_type(opcode);
set_offset(1);
}
};
csl_isa_field f_op2{
f_op2(){
set_width(3);
set_type(opcode);
set_offset(f_op.get_offset());
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_name();
```

DESCRIPTION :

Returns the name of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the name of isa filed *f_regb* using the *get_name()* method.

CSL CODE

```
csl_isa_field f_rega{
f_rega(){
set_width(6);
set_type(address);
set_name("reg_adr");
}
};
csl_isa_field f_regb{
f_regb(){
set_width(6);
set_type(address);
set_name(f_rega.get_name());
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_mnemonic();
```

DESCRIPTION :

Returns the mnemonic of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the abrevation for isa filed f_op2 using the *get_mnemonic()* method.

CSL CODE

```
csl_isa_field f_op{
f_op(){
set_width(2);
set_type(opcode);
set_mnemonic("alu");
}
};
csl_isa_field f_op2{
f_op2(){
set_width(3);
set_type(opcode);
set_mnemonic(f_op.get_mnemonic());
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_type();
```

DESCRIPTION :

Returns the type of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the type of isa field *f_op2* using the type of *f_op* with *get_type()* method.

CSL CODE

```
csl_isa_field f_op(2);
f_op.set_type(opcode);

csl_isa_field f_op2(3);
f_op2.set_type(f_op.get_type());
```

VERILOG CODE

```
isa_field_obj_name.get_lower();
```

DESCRIPTION :

Returns the lower index of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the lower index for the isa field *f_op2* using *get_lower()* method.

CSL CODE

```
csl_isa_field f_op(16,12);
f_op.set_type(opcode);
csl_isa_field f_op2(14,f_op.get_lower());
f_op2.set_type(subopcode);
```

VERILOG CODE

```
isa_field_obj_name.get_upper();
```

DESCRIPTION :

Returns the upper index of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the upper index for the isa field *f_alu* using *get_upper()* method.

CSL CODE

```
csl_isa_field f_op(16,12);
f_op.set_type(opcode);
csl_isa_field f_alu(f_op.get_upper(),f_op.get_lower());
f_alu.set_type(selector);
```

VERILOG CODE

```
isa_field_obj_name.get_width();
```

DESCRIPTION :

Returns the width of the isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the width of isa filed *f_alu* using the *get_width()* method.

CSL CODE

```
csl_isa_field f_op{
f_op(){
set_width(2);
set_type(opcode);
}
};
csl_isa_field f_alu(f_op.get_width());
f_alu.set_type(selector);
```

VERILOG CODE

```
isa_field_obj_name.set_enum(enum_name);
```

DESCRIPTION :

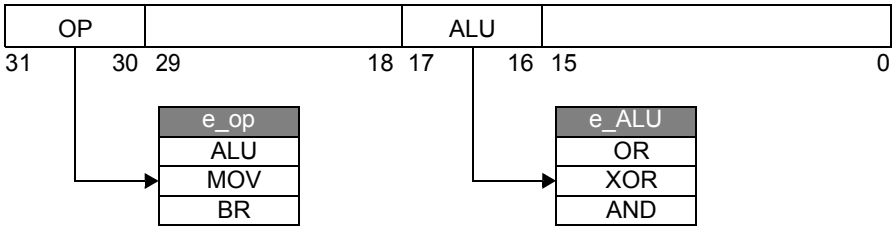
Associates an enum to the isa field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates the enum *op_code* to the isa field *op*.

FIGURE 1.9



CSL CODE

```
csl_enum op_code{
   ALU,
   MOV,
   BR
};
csl_enum eALU{
   OR,
   XOR,
   AND
};
csl_isa_field op{
op(){
set_type(opcode);
set_width(2);
set_enum(op_code);
}
};
csl_isa_field alu{
alu(){
set_type(selector);
set_width(2);
set_enum(eALU);
}
};
```

VERILOG CODE

```
isa_field_obj_name.set_enum_item(enum_item_name);
```

DESCRIPTION :

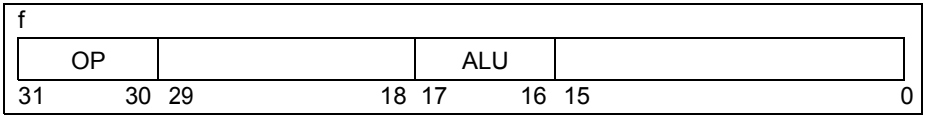
Associates an enum_item to the isa field object, this is legal only for an instruction.

[***CSL ISA Syntax and Command Summary***]

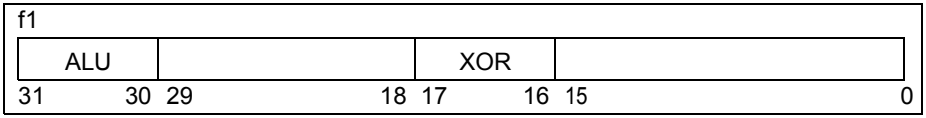
EXAMPLE :

Associates an enum item *ALU* to the isa field f_*op*, and an enum item *XOR* to isa field *f_alu.*

FIGURE 1.10



EXAMPLE:



CSL CODE

```
csl_enum e_op{
   ALU,
   MOV,
   BR
};
csl_enum e_ALU{
   XOR,
   OR,
   AND
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);

csl_isa_field f_alu(2, e_ALU);
f_alu.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f: isa {
```

```
f_op f_op;
f_alu f_alu;
f() {
set_type(instr_format);
set_position(alu, 16);
set_position(op, 30);
}
};
csl_isa_element f1: f{
f1() {
set_type(instr);
f_op.set_enum_item(ALU);
f_alu.set_enum_item(XOR);
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_enum();
```

DESCRIPTION :

Returns the associated enum of the isa field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates the enum *op_code* to the isa field *f_op2,* using *get_enum()* method.

CSL CODE

```
csl_enum e_op{
 ADD,
 MOV,
 SUB
};
 csl_isa_field f_op{
 f_op(){
 set_type(opcode);
 set_width(2);
 set_enum(op_code);
 }
 };
 csl_isa_field f_op2{
 f_op2(){
 set_type(subopcode);
 set_width(2);
 set_enum(f_op.get_enum());
 }
 };
```

VERILOG CODE

```
isa_field_obj_name.get_enum_item(enum_item_name);
```

DESCRIPTION :

Returns the associated enum_item of the isa field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates an enum item *MOV* to the isa field f_*op2*,using *get_enum_item()* method.

CSL CODE

```
csl_enum e_op{
   ALU,
   MOV,
   BR
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);

csl_isa_field f_op2(3);
f_alu.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f1: isa {
f_op f_op;
f1() {
set_type(instr_format);
set_position(f_op, 30);
f_op.set_enum_item(MOV);
}
};
csl_isa_element f2: f1 {
f_op2 f_op2;
f2() {
set_type(instr_format);
set_position(f_op2, 20);
f_op2.set_enum_item(f_op.get_enum_item());
   }
};
```

VERILOG CODE

```
isa_field_obj_name.set_value(num_expr);
```

DESCRIPTION :

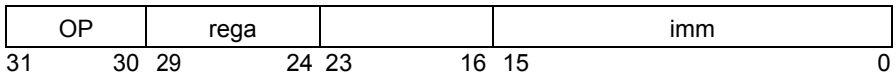
Associates a numeric value to the isa_field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates the numeric value 244 to the isa field *imm.*

FIGURE 1.11



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_field rega(6);
rega.set_type(address);

csl_isa_field imm{
imm(){
set_width(16);
set_type(constant);
set_value(244);
}
};
```

VERILOG CODE

```
isa_field_obj_name.get_value();
```

DESCRIPTION :

Returns the associated numeric value of the isa field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the numeric value for the isa field *rs* , using the *get_value()* method.

CSL CODE

```
csl_isa_field rd{
rd(){
set_width(6);
set_type(constant);
set_value(20);
}
};
csl_isa_field rs
rs(){
set_width(6);
set_type(constant);
set_value(rd.get_value());
}
};
```

VERILOG CODE

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

DESCRIPTION :

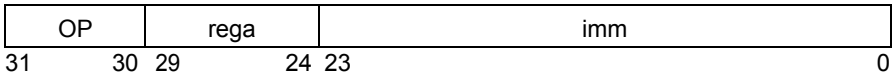
Sets the lower and upper bounds of the values that can be associated to the isa field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Adds the allowed range for the isa field *imm*.

FIGURE 1.12



CSL CODE

```
csl_isa_field op(2)
op.set_type(opcode);

csl_isa_field rega(6)
rega.set_type(address);

csl_isa_field imm{
imm(){
set_width(24);
set_type(constant);
add_allowed_range(1245,36);
}
};
```

VERILOG CODE

gen_decoder(csl_unit *unit_object_name*);

DESCRIPTION :

Generate a decoder for every field in the format.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Generates a decoder for the unit *u*.

FIGURE 1.13

CSL CODE

```
csl_unit u{
u(){}
};
csl_isa_field f_op(2);
f_op.set_type(opcode);
csl_isa_field f_branch{
f_branch(){
set_type(opcode);
set_width(2);
gen_decoder(u);
}
};
```

VERILOG CODE

csl_isa_element *isa_obj_name* [: *isa_obj_name0*];

DESCRIPTION :

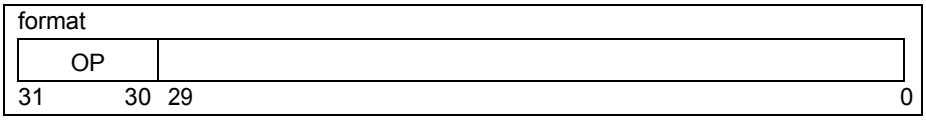
Csl_isa_element is a scope holder object. It can have instances and definitions of csl_isa_field and can be derived from another csl_isa_element .

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

FIGURE 1.14



CSL CODE

```
csl_isa_field f_op(2);
f_op.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format : isa {
f_op f_op;
format() {
set_type(inst_format);
set_position(f_op, 30);
}
};
```

set_type (instr_format | instr | root_format);

DESCRIPTION :

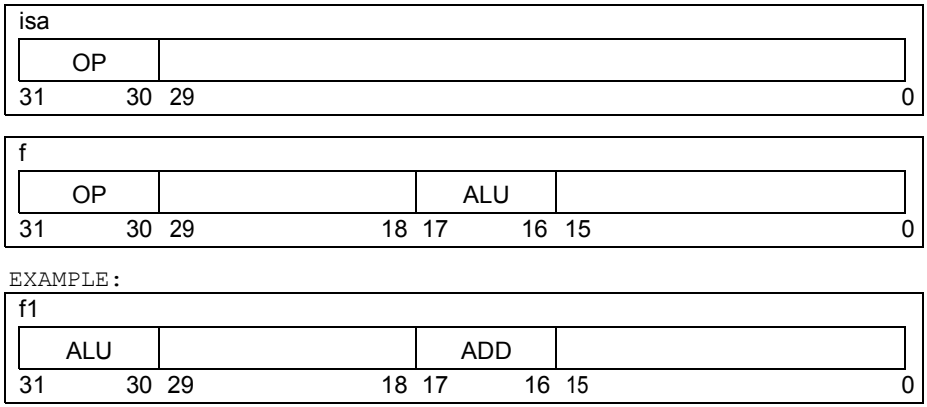
Set_type for csl_isa_element is mandatory. If the isa_element is not derived from another isa_element then the type of the element has to be root_format. A design can have multiple root_formats. A root_format cannot have isa_field instantiations or definitions. The root_format name is the name that is considered as the isa name.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the type "*root_format*" for isa element *isa,* "*instr_format*" for isa element *f* and "*instr*" for isa element *f1*.

FIGURE 1.15



CSL CODE

```
csl_enum e_op{
   ALU,
   MOV
};
csl_enum e_alu{
   ADD,
   SUB,
   XOR
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);

csl_isa_field f_alu(2, e_alu);
f_alu.set_type(opcode);

csl_isa_element isa {
```

```
isa() {
set_type(root_format);
set_width(32);
   }
};
csl_isa_element f: isa {
f_op f_op;
f_alu f_alu;
f() {
set_type(instr_format);
set_position(f_op, 30);
set_position(f_alu, 16);
}
};
csl_isa_element f1: f{
f1() {
set_type(instr);
f_op.set_enum_item(ALU);
f_alu.set_enum_item(ADD);
}
};
```

VERILOG CODE

set_width (numeric_expression);

DESCRIPTION :

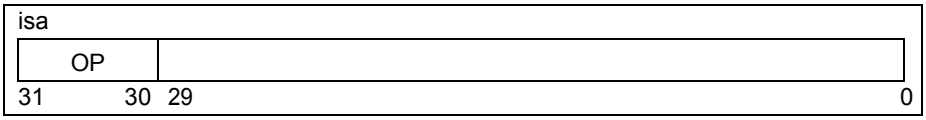
set_width for csl_isa_element is mandatory and can only be called for a root_format(set_type has to be before set_width);

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the width for the isa element *isa.*

FIGURE 1.16



CSL CODE

```
csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
```

VERILOG CODE

set_position(*isa_field*, numeric_expression**);**

DESCRIPTION :

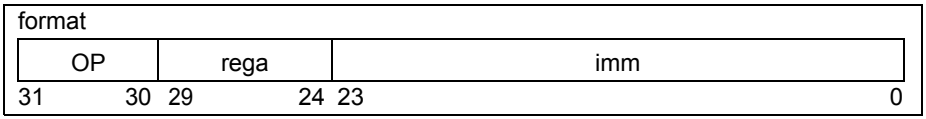
Set the absolute position of the isa field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the position for isa fields *op* and *imm.*

FIGURE 1.17



CSL CODE

```
csl_isa_field op(2);
op.set_type(selector);

csl_isa_field rega(6);
rega.set_type(address);

csl_isa_field imm(24);
imm.set_type(constant);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format {
op op;
rega rega;
imm imm;
format() {
set_type(instr_format);
set_position(op, 30);
set_next(op,rega);
set_position(imm,0);
}
};
```

VERILOG CODE

set_next(csl_isa_field *left_field*, **csl_isa_field** *right_field*);

DESCRIPTION :

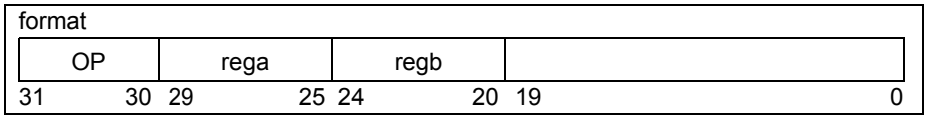
Set “the next “ field position of a field inside a format in a “linked-list“ way. In this way if the size of the fields changes they will remain adjacent to each other;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Arranges the isa fields *op,rega,regb*.

FIGURE 1.18



CSL CODE

```
csl_isa_field op(2);
op.set_type(selector);

csl_isa_field rega(5);
rega.set_type(address);

csl_isa_field regb(5);
regb.set_type(address);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format : isa {
op op;
rega rega;
regb regb;
format() {
set_type(instr_format);
set_position(op, 30);
set_next(op,rega);
set_next(rega,regb);
}
};
```

VERILOG CODE

set_previous(csl_isa_field *right_field*, **csl_isa_field** *left_field*);

DESCRIPTION :

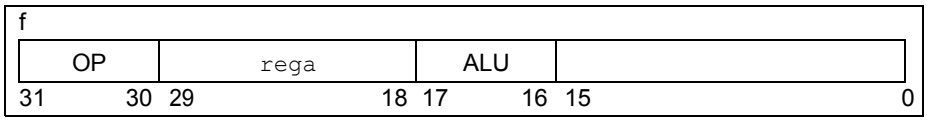
Set “the previous “ field position of a isa field inside a format in a “linked-list“ way. In this way if the size of the isa fields changes they will remain adjacent to each other.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

FIGURE 1.19



CSL CODE

```
csl_isa_field f_op(2);
f_op.set_type(opcode);

csl_isa_file f_rega(12);
f_rega.set_type(address);

csl_isa_field f_alu(2);
f_alu.set_type(selector);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f{
f_op f_op;
f_rega f_rega;
f_alu f_alu;
f() {
set_type(instr_format);
set_position(f_alu, 16);
set_previous(f_alu,f_rega);
set_previous(f_rega,f_op);
}
};
```

VERILOG CODE

gen_decoder(csl_unit *unit_object_name*);

DESCRIPTION :

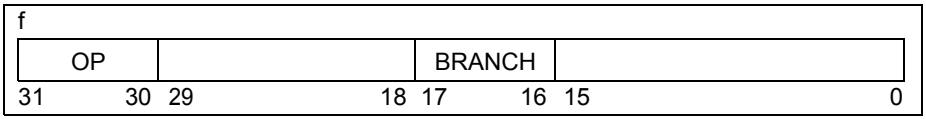
Generate a decoder for every field in the format.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Generates a decoder for the unit *u*.

FIGURE 1.20



CSL CODE

```
csl_unit u{
u(){}
};
csl_isa_field f_op(2);
f_op.set_type(opcode);
csl_isa_field f_branch{
f_branch(){
set_type(opcode);
set_width(2);
}};
csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f : isa {
f_op f_op;
f() {
set_type(inst_format);
set_position(f_op, 30);
gen_decoder(u);
}
};
```

VERILOG CODE

CHAPTER 2 CSL Interconnect

TABLE 2.1 Chapter Outline

2.1 CSL Interconnect Command Summary

2.1.1 Usage tables

CSL Signal

can only de declared

TABLE 2.2 CSL signal declaration in other CSL classes

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

CSL Signal Group

can be defined and instantiated

- can be defined in

TABLE 2.3 CSL signal group definition in other CSL classes

CSL class	Can be defined in
global scope	YES
CSL Unit	-
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-

TABLE 2.3 CSL signal group definition in other CSL classes

CSL class	Can be defined in
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	-

- can be instantiated in

TABLE 2.4 CSL signal group instantiation in other CSL classes

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

CSL Port

can only de declared

TABLE 2.5 CSL port declaration in other CSL classes

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

CSL Interface

can be defined and instantiated

- can be defined in

TABLE 2.6 CSL interface definition in other CSL classes

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 2.6 CSL interface definition in other CSL classes

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

- can be instantiated in

TABLE 2.7 CSL interface instantiation in other CSL classes

CSL class	Can be instantiated in
global scope	-
CSL Unit	YES
CSL Signal Group	-
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	-

CSL Unit

can be defined and instantiated. Units must have an input port with a clock attribute.The exception is if the unit has a set_type(combinational) then the unit does not have to have a clock. But the combinational must be instantiated in a design hierarchy which does have one or more clock inputs originating at the root of the design hierarchy.

- can be defined in

TABLE 2.8 CSL unit definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be instantiated in

TABLE 2.9 CSL unit instantiation in other CSL classes

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

TABLE 2.9 CSL unit instantiation in other CSL classes

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

Signals

```
csl_signal signal_name;
csl_signal signal_name0(signal_name1);
csl_signal signal_name([signal_data_type][,width]);
csl_signal signal_name([signal_data_type,]upper_limit,lower_limit);
csl_signal signal_name([signal_data_type,]bitrange_object_name);
signal_object.set_width(numeric_expression);
int signal_object.get_width();
signal_object.set_bitrange(bitrange_object_name);
bitrange_object signal_object.get_bitrange();
signal_object.set_range(upper_limit, lower_limit);
int signal_object.get_lsb();
int signal_object.get_msb();
```

Signal Types

```
signal_object.set_type(csl_signal_type);
csl_signal_type signal_object.get_type();
signal_object.set_attr(csl_signal_attr);
csl_signal_attr signal_object.get_attr();
???csl_list_name.set_attr(csl_signal_attribute);
```

Signal groups

```
csl_signal_group signal_group_name;
signal_group_instance.generate_individual_rtl_signals(on|off);
int signal_group_object.get_width();
```

Units

```
csl_unit unit_name;
set_type( combinational | sequential );
```

Units: port

```
csl_port port_name(port_direction[,port_type][,range]);
csl_port port_name(port_hierarchical_identifier);
```

```
port_object.reverse();
unit_name.add_port_list([port_direction,]interface_object);
port_object.set_width(numeric_expression);
int port_object.get_width();
port_object.set_bitrange(bitrange_object);
bitrange_object port_object.get_bitrange();
port_object_name.set_range(upper_limit, lower_limit);
int port_object_name.get_lsb();
int port_object_name.get_msb();
```

Port types

```
port_object_name.set_type(csl_port_type);
csl_port_type port_object_name.get_type();
port_object_name.set_attr(csl_port_attr);
csl_port_attr port_object_name.get_attr();
```

Units: interface

```
csl_interface interface_object_name;
interface_name.reverse();
int interface_object.get_width();
```

Units: parameter

```
csl_unit_parameter parameter_name (default_value);
unit_name #(parameter_override_value) unit_instance_name;
```

Units: prefix

```
set_unit_prefix(prefix_string[,prefix_specifier]);
string unit_object_name.get_unit_prefix();
set_signal_prefix(prefix_string);
string signal_object_name.get_signal_prefix();
set_signal_prefix_local(prefix_string);
string signal_object_name.get_signal_prefix_local();
```

Units: instance control bit

```
set_instance_alteration_bit(status);
```

Signal operations

```
signal_object_name.merge(merge_op,list_of_signals);
signal_object_name.merge(merge_op, signal_list_object_name);
```

New commands

```
(unit_name | unit_instance_name).set_unit_id(numeric_expression);
unit_name.add_logic(external_unit_enable);
(unit_name | instance_name).add_logic(unit_address_decoder,
address_signal_name);
signal_name.(field_name.)*add_logic(gen_decoder);
```

Units: signal

2.2 CSL Interconnect Commands

We will now describe the CSL interconnect specification commands. The CSL interconnect specification commands are used to create new unit instances and to connect the instances. The modules have port interfaces or the ports can connect to ports in which are used to connect the modules to signals. The ports can connect to an expression in a parent module or the ports can connect to ports in other instances in the same module or the ports can connect to the ports in the parent module. Connecting ports to logic in a higher level is not recommended.

2.2.1 Signals

The following section describes signals. Signals connect objects. The default type for a signal is a wire. A wire needs to be driven at all times. If it is not driven then the value on the wire is undefined and the wire is treated as a don't care by the synthesis tools. A don't care can be a zero or a one.

csl_signal *signal_name;*

DESCRIPTION :

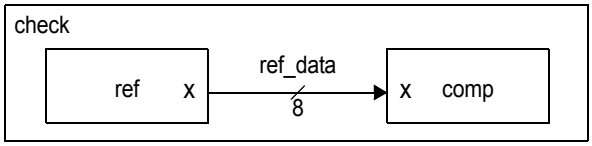
The signal can be created inside or outside a CSL unit's scope.
Signals can be declared inside signal-group and testbench.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is shown a data check unit that uses a reference and a comparator. The connection between the reference and the comparator is made using a 8-bit signal called *ref_data.*

FIGURE 2.1



CSL CODE

```
csl_unit ref{
   csl_port x (output, 8);
   ref(){}
};
csl_unit comp{
  csl_port x (input, 8);
   comp(){}
};
 csl_unit chek{
 csl_signal ref_data(wire, 8);
 ref ref(.x(ref_data));
 comp comp( .x(ref_data));
  chek(){}
};
```

VERILOG CODE

```
 module ref(x);
  output [7:0] x;
endmodule
module comp(x);
  input [7:0] x;
endmodule
module chek();
  wire [7:0] ref_data;
  ref ref(.x(ref_data));
  comp comp(.x(ref_data));
```

```
endmodule
```

csl_signal *signal_name0*(*signal_name1*);

DESCRIPTION :

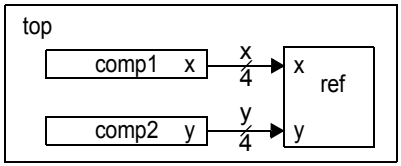
Creates a new signal with the name *signal_object_name0* by copying the signal object *signal_object_name1* passed as constructor argument.Signal_object_name1 can also be a hid (copy from another scope).

[CSL Interconnect Command Summary]

EXAMPLE :

Creates two units named *comp1* and *comp2* , a reference unit named *ref* and a *top* unit. In the *top* unit are defined two signals x , y .

FIGURE 2.2 Three units and a reference unit



CSL CODE

```
csl_unit comp1{
 csl_port x(output,4);
  comp1(){}
};
  csl_unit comp2{
  csl_port y (output,4);
  comp2(){}
};
 csl_unit ref{
 csl_port x(input,4), y(input,4);
 ref(){}
};
csl_unit top {
csl_signal  x(wire,4), y(x);
ref ref (.x(x),.y(y));
comp1 comp1 (.x(x));
comp2 comp2 (.y(y));
top(){}
};
```

VERILOG CODE

```
module comp1(x);
  output [3:0] x;
endmodule
module comp2(y);
```

```
  output [3:0] y;
endmodule
module ref(x,y);
  input [3:0] x;
  input [3:0] y;
endmodule
module top();
  wire [3:0] x;
  wire [3:0] y;
  ref ref(.x(x), .y(y));
  comp1 comp1(.x(x));
  comp2 comp2(.y(y));
endmodule
```

csl_signal *signal_name*(*[signal_data_type][,width]*);

DESCRIPTION :

Creates a signal object named signal_name. Optionally the type and width of the signal can be specified: signal_data_type specifies the type and can take any of the values in table Signal Types ; Width specifies the width of the signal and can be a numeric expression.

[CSL Interconnect Command Summary]

EXAMPLE :

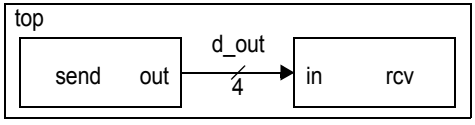
TABLE 2.10 Signal Types

Signal type	Description
wire	Creates a wire type signal
wand	Creates a wand type signal
wor	Creates a wor type signal
tri	Creates a tri type signal
triand	Creates a triand type signal
trior	Creates a trior type signal
tri0	Creates a tri0 type signal
tri1	Creates a tri1 type signal
supply0	Creates a supply0 type signal
supply1	Creates a supply1 type signal
reg	Creates a reg type signal.
integer	Creates a integer type signal.
time	Creates a time type signal.

EXAMPLE :

The example shows two connected units (a sender and a receiver).Each has a 4 bit port and are interconnected by a 4 bit signal in the top unit.

FIGURE 2.3 A sender and a receiver connected by a signal



CSL CODE

```
csl_unit send{
csl_port out(output,4);
```

```
send(){}
};
csl_unit rcv{
csl_port in(input,4);
rcv(){}
};
csl_unit top{
csl_signal d_out(wire,4);
send send( .out(d_out));
rcv rcv( .in(d_out));
top(){}
};
```

VERILOG CODE

```
module send(out);
  output [3:0] out;
endmodule

module rcv(in);
  input [3:0] in;
endmodule

module top();
  wire [3:0] d_out;
  send send(.out(d_out));
  rcv rcv(.in(d_out));
endmodule
```

csl_signal
signal_name(*[signal_data_type,]upper_limit,lower_limit*);

DESCRIPTION :

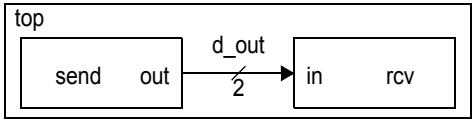
Creates a signal named *signal_object_name.* The constructor takes as parameters two numeric expressions (*upper_limit* and *lower_limit*) that represent the MSB (most significant bit) and LSB (least significant bit) of the bitrange associated with the signal. Optionally, the *signal_data_type* parameter can be specified setting the type of the signal to reg or wire, tri etc.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example are three units: a sender called *send*, a receiver unit called *rcv* and a global unit called *top*. The *send* unit has a 2-bit output port. The *rcv* unit has an input port with 2 bits width. The *top* unit has a 2 -bit signal called *d_out* .

FIGURE 2.4 A sender and a receiver connected by a signal.



CSL CODE

```
csl_unit send{
csl_port out(output,2);
send(){}
};
csl_unit rcv{
csl_port in(input,2);
rcv(){}
};
csl_unit top{
csl_signal d_out(wire,1,0);
send send( .out(d_out));
rcv rcv( .in(d_out));
top(){}
};
```

VERILOG CODE

```
module send(out);
  output [1:0] out;
endmodule
module rcv(in);
  input [1:0] in;
endmodule
module top();
  wire [0:1] d_out;
```

```
  send send(.out(d_out));
  rcv rcv(.in(d_out));
endmodule
```

csl_signal *signal_name*(*[signal_data_type,]bitrange_object_name*);

DESCRIPTION :

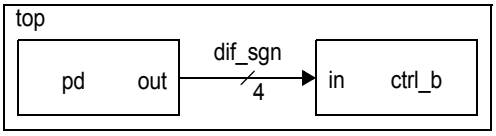
Creates a signal named *signal_name* with bitrange *bitrange_object_name*. Every signal object has a bitrange object attached by default. The default bitrange width for a newly created signal is 1. If a signal is created and the constructor uses a bitrange object as a parameter, a copy of that bitrange object is associated with the particular signal; the bitrange parameter can be a previously defined bitrange object or it can be an anonymous bitrange (defined on the spot). Optionally, the *signal_data_type* parameter can be specified setting the type of the signal to reg or wire, tri etc.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example a bitrange_object named *br1* is passed as a parameter to one signal named *dif_sgn*.

FIGURE 2.5 Two units connected by a signal



CSL CODE

```
csl_bitrange br1(4,1);
csl_unit pd{
csl_port out(output,br1);
pd(){}
};
csl_unit ctrl_b{
csl_port in(input,br1);
ctrl_b(){}
};
csl_unit top{
csl_signal dif_sgn (wire, br1);
pd  pd (.out(dif_sgn));
ctrl_b ctrl_b ( .in(dif_sgn));
top(){}
};
```

VERILOG CODE

```
module pd(out);
  output [4:1] out;
endmodule
module ctrl_b(in);
  input [4:1] in;
endmodule
```

```
module top();
  wire [4:1] dif_sgn;
  pd pd(.out(dif_sgn));
  ctrl_b ctrl_b(.in(dif_sgn));
endmodule
```

signal_object.**set_width**(*numeric_expression*);

DESCRIPTION :

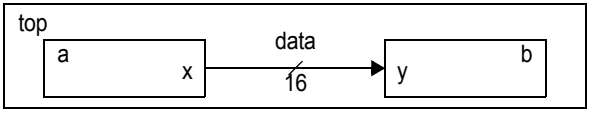
Set the width of a single dimensional signal. The bitrange of the signal is automatically created from the width and can be retrieved using get_bitrange() function. If width has already been set the cslc will flag an error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.6 is illustrated the use of signal *data* to connect *a* and *b* units inside a *top* unit. The width of the signal is set using *set_width* commands.

FIGURE 2.6



CSL CODE

```
csl_unit a{
csl_port x(output,16);
 a(){}
};
csl_unit b{
csl_port  y(input,16);
b(){}
};
 csl_unit top{
 csl_signal  data(wire);
 a a( .x(data));
 b b( .y(data));
 top(){
 data.set_width(16); }
};
```

VERILOG CODE

```
module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));
  b b(.y(data));
```

```
endmodule
```

```
int signal_object.get_width();
```

DESCRIPTION :

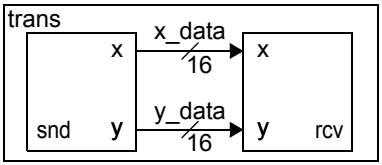
Returns the width of a single dimensional signal, otherwise it generates a cslc compile time error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.7 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the signals is set using *set_width* and *get_width* commands.

FIGURE 2.7 A sender and a receiver connected by two signals



CSL CODE

```
csl_unit snd{
csl_port x(output,16), y(output,16);
snd(){}
};
csl_unit rcv{
csl_port x(input,16), y(input,16);
rcv(){}
};
csl_unit trans{
csl_signal x_data, y_data;
snd snd( .x(x_data), .y(y_data));
rcv rcv( .x(x_data), .y(y_data));
trans(){
x_data.set_width(16);
y_data.set_width(x_data.get_width());
  }
};
```

VERILOG CODE

```
module snd(x,y);
  output [15:0] x;
  output [15:0] y;
endmodule
module rcv(x,y);
  input [15:0] x;
  input [15:0] y;
endmodule
```

```
module trans();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x(x_data), .y(y_data));
  rcv rcv(.x(x_data), .y(y_data));
endmodule
```

signal_object.**set_bitrange(***bitrange_object_name***);**

DESCRIPTION :

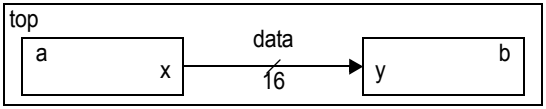
Sets the bitrange object of a signal. The bitrange object passed as a parameter to the set_bitange() method must already be declared. After the assigment of the bitrange the signal contains a reference to the bitrange object *bitrange_object_name*.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.8 is illustrated the use of a signal to connect *a* and *b* units inside a *top* unit. The width of the signal is set using *set_bitrange* commands.

FIGURE 2.8



CSL CODE

```
csl_bitrange br1(16);
csl_unit a{
csl_port x(output,br1);
a(){} };
csl_unit b{
csl_port y(input,br1);
b(){} };
csl_unit top{
csl_signal data;
a a( .x(data));
b b ( .y(data));
top(){
data.set_bitrange(br1); }
};
```

VERILOG CODE

```
module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));
  b b(.y(data));
endmodule
```

bitrange_object signal_object.**get_bitrange();**

DESCRIPTION :

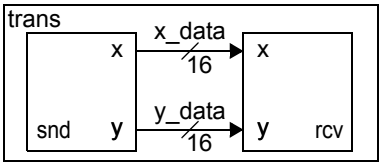
Returns a bitrange object. The return type of this function is an object of type bitrange.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.9 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the signals is set using *set_bitrange* and *get_bitrange* commands.

FIGURE 2.9



CSL CODE

```
csl_bitrange br1(16);
csl_unit snd{
csl_port x1(output,16), y1(output,16);
snd(){}
};
csl_unit rcv{
csl_port x2(input,16), y2(input,16);
rcv(){}
};
csl_unit trans{
csl_signal x_data, y_data;
snd snd( .x1(x_data), .y1(y_data));
rcv rcv( .x2(x_data), .y2(y_data));
top(){
x_data.set_bitrange(br1);
y_data.set_bitrange(x_data.get_bitrange());
}};
```

VERILOG CODE

```
module snd(x1,y1);
  output [15:0] x1;
  output [15:0] y1;
endmodule
module rcv(x2,y2);
  input [15:0] x2;
  input [15:0] y2;
endmodule
```

```
module top();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x1(x_data),
      .y1(y_data));
  rcv rcv(.x2(x_data),
      .y2(y_data));
endmodule
```

signal_object.**set_range**(*upper_limit, lower_limit*);

DESCRIPTION :

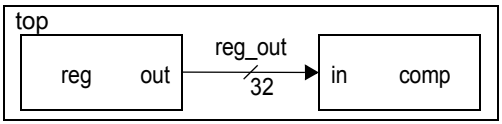
Set the bitrange for the *signal_object_name* using the *upper_limit* and *lower_limit* delimiters.This method does not have a corresponding get method.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example we have one 32-bit signal named *reg_out* which connect a register called *reg* and a comparator called *comp*. The width of signal is set using *set_range* command.

FIGURE 2.10 A comparator and a register



CSL CODE

```
csl_unit reg1{
csl_port  out(output,32);
reg1(){}
};
csl_unit comp{
csl_port in(input, 32);
comp(){}
};
csl_unit top{
csl_signal reg_out;
reg1 reg1( .out(reg_out));
comp comp( .in(reg_out));
top(){
reg_out.set_range(31, 0);}
};
```

VERILOG CODE

```
module reg1(out);
  output [31:0] out;
endmodule
module comp(in);
  input [31:0] in;
endmodule
module top();
  wire [31:0] reg_out;
  reg1 reg1(.out(reg_out));
  comp comp(.in(reg_out));
```

```
endmodule
```

int *signal_object*.**get_lsb**();

DESCRIPTION :

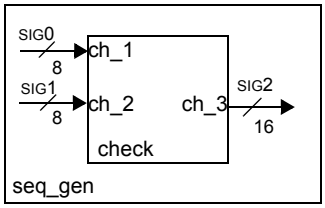
Returns the lower index value for the signal_object_name.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the lower index of a signal can be set explicitly or it can be the result of a get method applied on another object.

FIGURE 2.11 Connecting three signals to the check unit



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output, 16);
check(){}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2;
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){
sig2.set_range(sig0.get_lower(), sig0.get_upper() + sig1.get_upper()
+1);}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

```
int signal_object.get_msb();
```

DESCRIPTION :

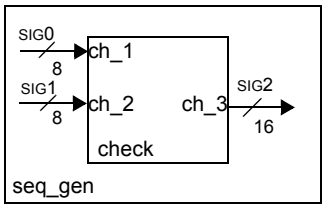
Return the upper index value for the signal.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the return value of the get_upper_index() method is used to set another upper index. Since the return value is an int, it can also be used in other contexts where an int parameter would be allowed.

FIGURE 2.12 Connecting three signals to the check unit



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output, 16);
check(){}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2;
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){
sig2.set_range(sig0.get_lower(), sig0.get_upper() + sig1.get_upper()
+1);
}};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

2.2.1.1 Signal Types

The CSL language will create additional type information for signals. The additional attributes information will be used to check that signals of compatible types are connected to each other. The type information in the CSL is declared. There are two different ways to set the signal type.

signal_object.**set_type**(*csl_signal_type*);

DESCRIPTION :

Assign a net type to *signal_object_name.* Applies to both signals and ports. Csl_signal_type specifies the type of signal and can take any of the values in table Signal Types;

TABLE 2.11 Signal Types

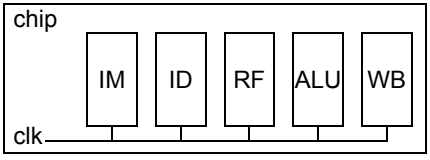
	Description
INTEGER	integer
REG	reg
TRIREG	tristate net
WIRE	wired net
WOR	wired or net
WAND	wired and net
TRI	similar to wire , multiple drivers
TRI0	tri set to 0
TRI1	tri set to 1
TRIOR	tristate or net
TRIAND	tristate and net
SUPPLY0	net set to 0
SUPPLY1	net set to 1
TIME	type of datatypes
REAL	double precision floating point variable
REALTIME	for realtime aplications

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the set_type() method applied on both signal objects and port objects

FIGURE 2.13 Connecting clk to units



CSL CODE

```
csl_unit im{
csl_port im_p(input);
```

```
im(){}
};
csl_unit id{
csl_port id_p(input);
id(){}
};
csl_unit rf{
csl_port rf_p(input);
rf(){
rf_p.set_type(reg);}
};
csl_unit alu{
csl_port alu_p(input);
alu(){}
};
csl_unit wb{
csl_port wb_p(input);
wb(){}
};
csl_unit chip{
csl_signal clk1(8);
im im( .im_p(clk1));
id id( .id_p(clk1));
rf rf( .rf_p(clk1));
alu alu( .alu_p(clk1));
wb wb( .wb_p(clk1));
chip(){
clk1.set_type(wire);}
};
```

VERILOG CODE

```
module im(im_p);
  input im_p;
endmodule
module id(id_p);
  input id_p;
endmodule
module rf(rf_p);
  input reg rf_p;
endmodule
module alu(alu_p);
```

```
  input alu_p;
endmodule
module wb(wb_p);
  input wb_p;
endmodule
module chip();
  wire [7:0] clk1;
  im im(.im_p(clk1));
  id id(.id_p(clk1));
  rf rf(.rf_p(clk1));
  alu alu(.alu_p(clk1));
  wb wb(.wb_p(clk1));
endmodule
```

csl_signal_type signal_object.**get_type**();

DESCRIPTION :

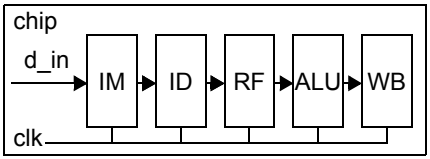
Returns the type of the signal object or port object for which is called.

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the use of get_type() method combined with set_type(). This way, signal/ ports dependencies are created.

FIGURE 2.14 Connecting clocks and a input signal to units



CSL CODE

```
csl_unit im{
csl_port im_c(input), im_s(input);
im(){}
};
csl_unit id{
csl_port id_c(input), id_s(input);
id(){}
};
csl_unit rf{
csl_port rf_c(input), rf_s(input);
rf(){}
};
csl_unit alu{
csl_port alu_c(input), alu_s(input);
alu(){}
};
csl_unit wb{
csl_port wb_c(input), wb_s(input);
wb(){}
};
csl_unit chip{
csl_signal  clk1, d_in;
im im(.im_c(clk1), .im_s(d_in));
id id(.id_c(clk1));
rf rf(.rf_c(clk1));
alu alu(.alu_c(clk1));
```

```
wb wb(.wb_c(clk1));
chip(){
clk1.set_type(wire);
d_in.set_type(clk1.get_type());
}
};
```

VERILOG CODE

```
module im(im_p);
  input im_p;
endmodule
module id(id_p);
  input id_p;
endmodule
module rf(rf_p);
  input rf_p;
endmodule
module alu(alu_p);
  input alu_p;
endmodule
module wb(wb_p);
  input wb_p;
endmodule
module chip();
  wire [7:0] clk1;
  wire [7:0] d_in;
  im im(.im_p(clk1));
  id id(.id_p(clk1));
  rf rf(.rf_p(clk1));
  alu alu(.alu_p(clk1));
  wb wb(.wb_p(clk1));
endmodule
```

signal_object.**set_attr**(*csl_signal_attr*);

DESCRIPTION :

Assign an attribute to *signal_name*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 2.12

TABLE 2.12 signal attributes

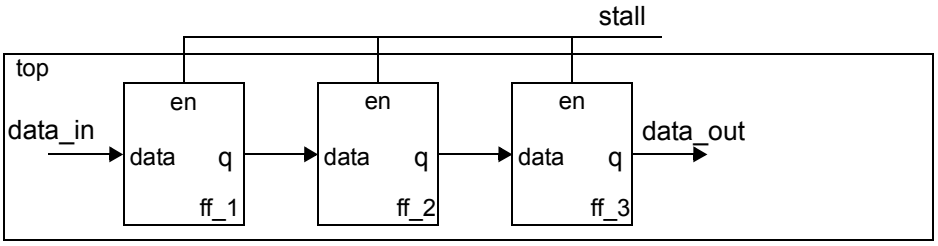
mnemonic	signal attribute
en	enable
stall	stall
mux_sel	mux select
decode	decoded minterm/maxterm
clock	clock signal
reset	reset signal

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1* ,*ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.15



CSL CODE

```
csl_unit ff{
csl_port q(output), data(input), e(input);
ff(){ }
};
csl_unit top{
csl_signal d_data1,d_data2,data_in,data_out,st;
ff ff_1(.data(data_in),.e(st),.q(d_data1));
ff ff_2(.data(d_data1),.e(st),.q(d_data2));
ff ff_3(.data(d_data2),.e(st),.q(data_out));
top(){
st.set_attr(clk);
```

```
    }
  };
```

VERILOG CODE

```
module ff(q,
          data,
          e);
  input data;
  input e;
  output q;
endmodule
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

csl_signal_attr signal_object.**get_attr**();

DESCRIPTION :

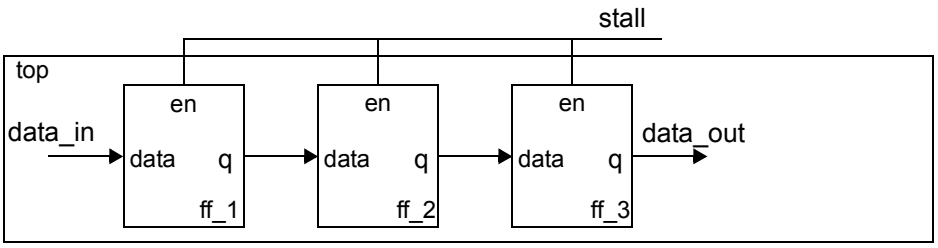
Returns the attribute of *signal_object_name*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 2.12.
It can only be called on a signal object and returns an object of type *csl_signal_attr* .

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1, ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.16



CSL CODE

```
csl_unit ff{
csl_port q(output), data(input), e(input);
ff(){}
};
csl_unit top{
csl_signal d_data1,d_data2,data_in,data_out,st;
ff ff_1(.data(data_in),.e(st),.q(d_data1));
ff ff_2(.data(d_data1),.e(st),.q(d_data2));
ff ff_3(.data(d_data2),.e(st),.q(data_out));
top(){
st.set_attr(en);
data_out.set_attr(st.get_attr());
  }
};
```

VERILOG CODE

```
module ff(q,
          data,
          e);
  input data;
```

```
  input e;
  output q;
endmodule
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

```
csl_list_name.set_attr(csl_signal_attribute);
```

DESCRIPTION :

Assign an attribute to *signal_object_name*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 2.13

TABLE 2.13 signal attributes

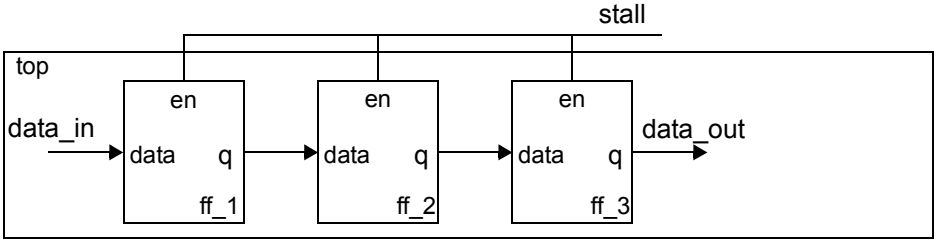
mnemonic	signal attribute
en	enable
stall	stall
pe	pipe enable
ps	pipe stall
ms	mux select
decode	decoded minterm/maxterm
clk	clock signal
rst	reset signal
wr_en	write enable

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect two instances of a flip flop ,*ff_1* and ff2.The module *top* contains both *ff1* and *ff2* and additionally clock and enable signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.17



CSL CODE

```
csl_unit ff{
csl_port en1(input), clk1(input), data (input);
csl_port q(output);
ff(){}
};
csl_unit top{
ff ff1;
```

```
    ff ff2;
    csl_signal data_path, clock, enable, data_in, data_out;
    top(){
    clock.set_attr(clk);
    enable.set_attr(en);
         }};
```

VERILOG CODE

```
    module ff(en1,
              clk1,
              data,
              q);
      input en1;
      input clk1;
      input data;
      output q;
    endmodule
    module top();
      wire data_path;
      wire clock;
      wire enable;
      wire data_in;
      wire data_out;
      ff ff1();
      ff ff2();
    endmodule
```

2.2.1.2 Grouping Signals

Signals can be grouped together and assigned a symbolic name. The great advantage of using signal groups is evident when we intend to connect the objects in our design.Instead of connecting tens of wires to tens of ports we just connect a group of signals(a signal group in CSL) to a group of ports(an interface in CSL).

This connection can be done manually(using the connect command) or automatically by the autorouter.In the first and third case,we must have the signals in the signal group and the ports in the interface put in the order of the connection preference so that the port and the signal we want to connect have the same index in the both lists.

csl_signal_group *signal_group_name;*

DESCRIPTION :

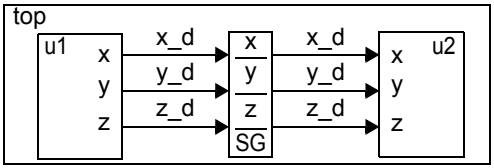
Creates a signal group named *signal_group_name*. Csl_signal_group is a scope delimited by curly braces and it contains a constructor. Inside the scope are declared the signals from group.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example are three units: *u1, u2* and *top*. The *top* unit contains *u1* and *u2* units and a signal group named *SG* with signals *x_d, y_d, z_d*.The u1 and u2 units have the interfaces *ifc1* and *ifc2* which are connected with the signal group *SG*. The interface *ifc1* has 3 output ports called *x, y, z* and ifc2 has 3 input ports called *x, y, z.*

FIGURE 2.18



CSL CODE

```
csl_interface ifc1 {
csl_port x(output), y(output), z(output);
ifc1(){}
};
csl_interface ifc2 {
csl_port x(input), y(input), z(input);
ifc2(){}
};
csl_unit u1{
ifc1 ifc1;
u1(){}
};
csl_unit u2{
ifc2 ifc2;
u2(){}
};
csl_signal_group sg{
csl_signal x_d, y_d, z_d;
sg(){}
};
csl_unit top{
```

```
    sg sg;
    u1 u1( .ifc1(sg));
    u2 u2( .ifc2(sg));
    top(){ }
    };
```

VERILOG CODE

```
    module u1(ifc1_x,
              ifc1_y,
              ifc1_z);
       output ifc1_x;
       output ifc1_y;
       output ifc1_z;
    endmodule
    module u2(ifc2_x,
              ifc2_y,
              ifc2_z);
      input ifc2_x;
      input ifc2_y;
      input ifc2_z;
    endmodule
    module top();
      wire sg_x_d;
      wire sg_y_d;
      wire sg_z_d;
      u1 u1);
      u2 u2);
    endmodule
```

```
signal_group_instance.generate_individual_rtl_signals(on|off);
```

DESCRIPTION :

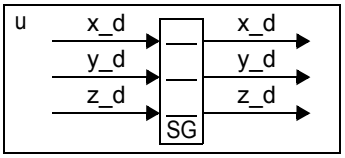
When this is set to **on**, as the group "traverses" scopes the autorouter component will generate a port for each signal inside the group. If it is set to **off** (default behaviour), the grouped signals will be "merged" into a single port as they come across scope boundaries, else if the *status* is set to true, the grouped signals will be used as individual ports.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example we have one unit *u* wh
itch contain a signal-group *SG* with signals *x_d, y_d, z_d.* It was use the command *generate_individual_rtl_signals(on)* for generate a port for each signal inside the SG group.

FIGURE 2.19



CSL CODE

```
csl_signal_group sg{
csl_signal x_d;
csl_signal y_d;
csl_signal z_d;
sg(){
generate_individual_rtl_signals(on); }
 };
csl_unit u{
sg sg1inst;
u(){  }
 };
```

VERILOG CODE

```
module u();
  wire sg1inst_x_d;
  wire sg1inst_y_d;
  wire sg1inst_z_d;
endmodule
```

`int signal_group_object.get_width();`

DESCRIPTION :

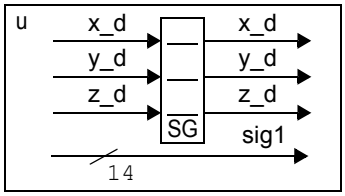
Will returns the sum of all widths of the signals in the signal_group named `signal_group_object.`

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows one signal group named SG which contains three signals with different widths, and a signal named *sig1*. The width of signal *sig1* is set to be equal with width of signal_group using the commands *set_width* and *get_width()*.

FIGURE 2.20



CSL CODE

```
csl_signal_group sg{
  csl_signal  x_d(2);
  csl_signal  y_d(4);
  csl_signal  z_d(8);
sg(){ }
 };
csl_unit u{
sg sginst;
csl_signal sig1;
u(){
sig1.set_width(sginst.get_width());}
 };
```

VERILOG CODE

```
module u();
  wire [1:0]  sg1inst_x_d;
  wire [3:0] sg1inst_y_d;
  wire [7:0] sg1inst_z_d;
  wire [13:0] sig1;
endmodule
```

2.2.1.3 Units

CSL Units act like Verilog modules. Units can be instantiated within other units. A unit represents a scope and every variable, signal, or port defined in that unit will have its name prepended with the unit name.Units must have an input port with a clock attribute.The exception is if the unit has a set_type(combinational) then the unit does not have to have a clock. But the combinational must be instantiated in a design hierarchy which does have one or more clock inputs originating at the root of the design hierarchy.

csl_unit *unit_name;*

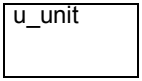
DESCRIPTION :

CSL unit class declaration. For definition see 2.2.1.3 Units above. It represents a scope delimited by curly braces and contains a constuctor.
The CSL unit class declaration alone does not translate into a Verilog module. Only a unit definition results in a Verilog module.

[CSL Interconnect Command Summary]

EXAMPLE :

In the following example is a unit called *u_unit* .



CSL CODE

```
csl_unit u_unit{
u_unit(){ }
};
```

VERILOG CODE

```
module u_unit();
endmodule
```

```
set_type( combinational | sequential );
```

DESCRIPTION :

Units must have an input port with a clock attribute. The exception is if the unit has a set_type(combinational) then the unit does not have to have a clock. But the combinational must be instantiated in a design hierarchy which does have one or more clock inputs originating at the root of the design hierarchy.

EXAMPLE :

2.2.1.4 Units:Ports

A port is a directed signal type. Ports are declared in the interface of a unit or in an unit and are used to make connections (input or output signals, or both).

csl_port *port_name*(*port_direction*[,*port_type*][,*range*]);

DESCRIPTION :

Port object declaration. These types can be declared inside a unit class definition and are thus added to the unit's default interface, or inside an interface definition and become part of the respective interface. Parameters that can be passed to a port declaration are:

- port direction, specifying the direction of the signal passing through the port;

TABLE 2.14 Port direction specifiers

Port direction	Description
input	input port
output	output port
inout	inout port

- port type, details the signal type passing through the port

TABLE 2.15 Port types

Port type	Description
wire	Creates a wire type port
wand	Creates a wand type port
wor	Creates a wor type port
tri	Creates a tri type port
triand	Creates a triand type port
trior	Creates a trior type port
tri0	Creates a tri0 type port
tri1	Creates a tri1 type port
supply0	Creates a supply0 type port
supply1	Creates a supply1 type port
reg	Creates a reg type port. **Note:** only available for output ports
integer	Creates a integer type port. **Note:** only available for output ports
time	Creates a time type port. **Note:** only available for output ports

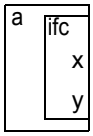
- range, specifies the range of the port and can be a bitrange object, a width numeric expression or a upper index, lower index pair of numeric expressions

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is a unit called *a* which contains an interface called *ifc*. The interface has two ports : x and y;

FIGURE 2.21



CSL CODE

```
csl_interface ifc{
csl_port x(input,wire,4), y(output,reg,4);
ifc(){}
};
csl_unit a{
ifc ifc;
a(){}
};
```

VERILOG CODE

```
module a(ifc_x,
         ifc_y);
  input [3:0] ifc_x;
  output reg [3:0] ifc_y;
endmodule
```

csl_port *port_name*(*port_hierarchical_identifier*);

DESCRIPTION :

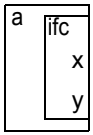
This acts like a copy constructor for a port object. The *port_name* object is created by copying all the properties found in the port object specified by the *port_hierarchical_identifier* (port_direction, port_type and range).

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is a unit called *a* which contains an interface called *ifc.* The interface has two ports : x and y;

FIGURE 2.22



CSL CODE

```
csl_unit a{
csl_port x(output), y(output);
a(){}
};
csl_unit b{
csl_port x1(a.x), y1(a.y);
b(){}
};
```

VERILOG CODE

```
module a(x,
         y);
  output x;
  output y;
endmodule

module b(x1 , y1);
 output x1;
 output y1;
endmodule
```

port_object.**reverse();**

DESCRIPTION :

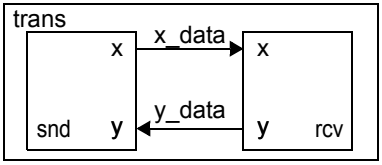
Reverse the direction of a port. This method is called on a port object and it reverses its direction. This method can only be called for input/output ports, otherwise the function has no effect and a warning is produced.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example it use the method *reverse()* to change the direction of the ports *x* and *y.*

FIGURE 2.23 A block named *trans* with two instances named *snd* and *rcv* interconnected



CSL CODE

```
csl_unit snd{
  csl_port x(output), y(input);
  snd(){
}};

csl_unit rcv{
  csl_port x(snd.x), y(snd.y);
  rcv(){
  x.reverse();
  y.reverse();
}};
csl_unit trans{
csl_signal x_data, y_data;
snd snd( .x(x_data), .y(y_data));
rcv rcv( .x(x_data), .y(y_data));
trans(){}
};
```

VERILOG CODE

```
module snd(x,
           y);
  output x;
  output y;
endmodule

module rcv(x,
```

```
          y);
  input x;
  input y;
endmodule

module trans();
  wire x_d;
  wire y_d;
  snd snd(.x(x_d),
          .y(y_d));
  rcv rcv(.x(x_d),
          .y(y_d));
endmodule
```

unit_name.**add_port_list(**[*port_direction*,]*interface_object***);**

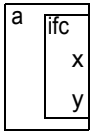
DESCRIPTION :

This command adds all the ports from an interface object (or only ports specified by the optional parameter *port_direction*) the default interface of the unit. This can be useful when there is a need to have the port names in the generated verilog code without interface names appended.

[CSL Interconnect Command Summary]

EXAMPLE :

In the following example the ports *x* and *y* are introduced in a *csl_list* which was added to the interface *ifc.*



CSL CODE

```
csl_interface ifc{
ifc(){
ifc.add_port_list(input, 2, csl_list(x,y));
}
};
csl_unit a{
ifc ifc;
a(){}
};
```

VERILOG CODE

```
 module a(ifc_x,
          ifc_y);
  input [1:0] ifc_x;
  input [1:0] ifc_y;
endmodule
```

```
port_object.set_width(numeric_expression);
```

DESCRIPTION :

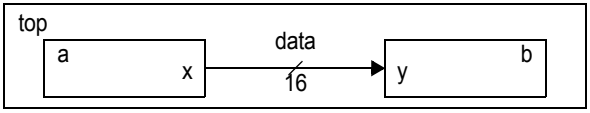
Set the width of a port. The bitrange of the port is automatically created from the width and can be retrieved using get_bitrange() function. If width has already been set the cslc will flag an error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.6 is illustrated the use of signal *data* to connect *a* and *b* units inside a *top* unit.The *a* unit has an output port called *x* and the *b* unit has an input port called *y*. The width of the ports is set using *set_width* command.

FIGURE 2.24



CSL CODE

```
csl_unit a{
csl_port x(output);
a(){
x.set_width(16);}
};
csl_unit b{
csl_port  y(input);
b(){
y.set_width(16);}
};
csl_unit top{
csl_signal  data(wire,16);
a a( .x(data));
b b( .y(data));
top(){}};
```

VERILOG CODE

```
module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));
  b b(.y(data));
```

```
endmodule
```

```
int port_object.get_width();
```

DESCRIPTION :

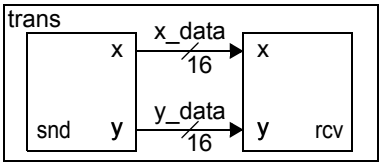
Returns the width of a port, otherwise it generates a cslc compile time error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.7 is illustrated the use of signals and ports to connect *snd* and *rcv* units inside a *trans* unit. The width of the ports is set using *set_width* and *get_width* commands.

FIGURE 2.25 A sender and a receiver connected by two signals



CSL CODE

```
csl_unit snd{
csl_port x(output), y(output);
snd(){
x.set_width(16);
y.set_width(x.get_width());
}
};
csl_unit rcv{
csl_port x(input), y(input);
rcv(){
x.set_width(16);
y.set_width(x.get_width());
}
};
csl_unit trans{
csl_signal x_data(16), y_data(16);
snd snd( .x(x_data), .y(y_data));
rcv rcv( .x(x_data), .y(y_data));
trans(){ }
};
```

VERILOG CODE

```
module snd(x,y);
  output [15:0] x;
  output [15:0] y;
endmodule
module rcv(x,y);
```

```
  input [15:0] x;
  input [15:0] y;
endmodule
module trans();
  wire  [15:0] x_data;
  wire  [15:0] y_data;
  snd snd(.x(x_data), .y(y_data));
  rcv rcv(.x(x_data), .y(y_data));
endmodule
```

port_object.**set_bitrange(***bitrange_object***);**

DESCRIPTION :

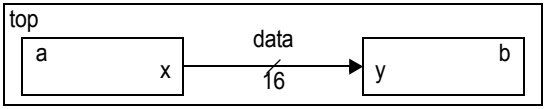
Sets the bitrange object of a port. The bitrange object passed as a parameter to the set_bitange() method must already be declared. After the assigment of the bitrange the port contains a reference to the bitrange object *bitrange_object_name*.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.8 is illustrated the use of a signal and input/output ports to connect *a* and *b* units inside a *top* unit. The width of the ports is set using *set_bitrange* commands.

FIGURE 2.26



CSL CODE

```
csl_bitrange br1(16);
csl_unit a{
csl_port x(output);
a(){
x.set_bitrange(br1);}
};
csl_unit b{
csl_port y(input);
b(){
y.set_bitrange(br1);}
 };
csl_unit top{
csl_signal data(br1);
a a( .x(data));
b b ( .y(data));
top(){}};
```

VERILOG CODE

```
module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));
```

```
  b b(.y(data));
endmodule
```

bitrange_object port_object.**get_bitrange();**

DESCRIPTION :

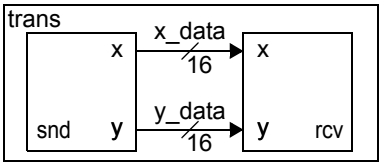
Returns a bitrange object. The return type of this function is an object of type bitrange.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.9 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the ports is set using *set_bitrange* and *get_bitrange* commands.

FIGURE 2.27



CSL CODE

```
csl_bitrange br1(16);
csl_unit snd{
csl_port x1(output), y1(output);
snd(){
x1.set_bitrange(br1);
y1.set_bitrange(x1.get_bitrange());
}
};
csl_unit rcv{
csl_port x2(input), y2(input);
rcv(){
x2.set_bitrange(br1);
y2.set_bitrange(x2.get_bitrange());
}
};
csl_unit top{
csl_signal x_data(br1) , y_data(br1);
snd snd( .x1(x_data), .y1(y_data));
rcv rcv( .x2(x_data), .y2(y_data));
top(){
}};
```

VERILOG CODE

```
module snd(x1,y1);
  output [15:0] x1;
  output [15:0] y1;
endmodule
```

```
module rcv(x2,y2);
  input [15:0] x2;
  input [15:0] y2;
endmodule
module top();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x1(x_data),
      .y1(y_data);
  rcv rcv(.x2(x_data),
      .y2(y_data));
endmodule
```

port_object_name.**set_range**(*upper_limit, lower_limit*);

DESCRIPTION :

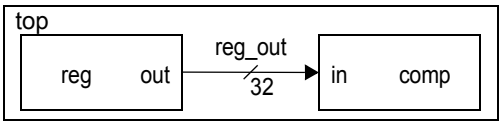
Set the bitrange for the *port_object_name* using the *upper_limit* and *lower_limit* delimiters.This method does not have a corresponding get method.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example we have one 32-bit signal named *reg_out* which connect a register called *reg* and a comparator called *comp*. The width of ports *out* and *in* is set using *set_range* command.

FIGURE 2.28 A comparator and a register



CSL CODE

```
csl_unit reg1{
csl_port  out(output);
reg1(){
out.set_range(31, 0);}
};
csl_unit comp{
csl_port in(input);
comp(){
in.set_range(31, 0);}
};
csl_unit top{
csl_signal reg_out(32);
reg1 reg1( .out(reg_out));
comp comp( .in(reg_out));
top(){}};
```

VERILOG CODE

```
module reg1(out);
  output [31:0] out;
endmodule
module comp(in);
  input [31:0] in;
endmodule
module top();
  wire [31:0] reg_out;
  reg1 reg1(.out(reg_out));
  comp comp(.in(reg_out));
```

```
endmodule
```

```
int port_object_name.get_lsb();
```

DESCRIPTION :

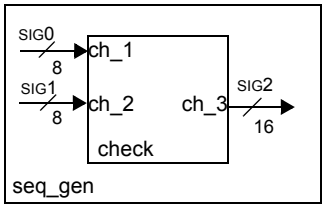
Returns the lower index value for the port_object_name.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the lower index of a port can be set explicitly or it can be the result of a get method applied on another object.

FIGURE 2.29 Connecting three to the check unit



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output);
check(){
ch_3.set_range(ch_1.get_lower(), ch_1.get_upper() + ch_2.get_upper()
+1);
}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2(wire,16);
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

```
int port_object_name.get_msb();
```

DESCRIPTION :

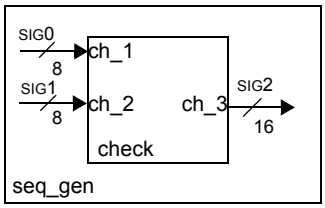
Return the upper index value for the port.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the return value of the get_upper_index() method is used to set another upper index. Since the return value is an int, it can also be used in other contexts where an int parameter would be allowed.

FIGURE 2.30



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output);
check(){
ch_3.set_range(ch_1.get_lower(), ch_1.get_upper() + ch_2.get_upper()
+1);
}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2(wire,16);
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

port_object_name.**set_type**(*csl_port_type*);

DESCRIPTION :

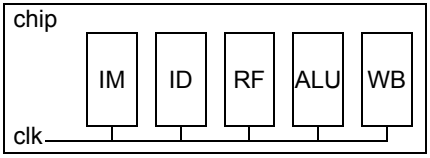
Assign a net type to *port_object_name.* Applies to both signals and ports.The csl_port_type can be one of the port types from Table 4.11Port types .

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the set_type() method applied on both port objects and signal objects

FIGURE 2.31 Connecting clk to units



CSL CODE

```
csl_unit im{
csl_port im_p(input);
im(){
im_p.set_type(wire);}
};
csl_unit id{
csl_port id_p(input);
id(){
id_p.set_type(reg);}
};
csl_unit rf{
csl_port rf_p(input);
rf(){
rf_p.set_type(reg);
}
};
csl_unit alu{
csl_port alu_p(input);
alu(){
alu_p.set_type(wire);}
};
csl_unit wb{
csl_port wb_p(input);
wb(){
wb_p.set_type(reg);}
};
```

```
csl_unit chip{
csl_signal clk1(8);
im im( .im_p(clk1));
id id( .id_p(clk1));
rf rf( .rf_p(clk1));
alu alu( .alu_p(clk1));
wb wb( .wb_p(clk1));
chip(){
clk1.set_type(wire);}
};
```

VERILOG CODE

```
module im(im_p);
  input im_p;
endmodule
module id(id_p);
  input reg id_p;
endmodule
module rf(rf_p);
  input reg rf_p;
endmodule
module alu(alu_p);
  input  alu_p;
endmodule
module wb(wb_p);
  input reg wb_p;
endmodule
module chip();
  wire [7:0] clk1;
  im im(.im_p(clk1));
  id id(.id_p(clk1));
  rf rf(.rf_p(clk1));
  alu alu(.alu_p(clk1));
  wb wb(.wb_p(clk1));
endmodule
```

csl_port_type port_object_name.**get_type**();

DESCRIPTION :

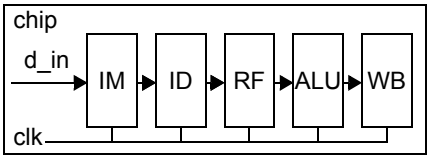
Returns the type of the port object or signal object for which is called *port_object_name*.

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the use of get_type() method combined with set_type(). This way, signal/ports dependencies are created.

FIGURE 2.32 Connecting clocks and a input signal to units



CSL CODE

```
csl_unit im{
csl_port im_p(input), im_c(input);
im(){
im_p.set_type(wire);
im_c.set_type(im_p.get_type());}
};
csl_unit id{
csl_port id_p(input), id_c(input);
id(){
id_p.set_type(wire);
id_c.set_type(id_p.get_type());}
};
csl_unit rf{
csl_port rf_p(input), rf_c(input);
rf(){
rf_p.set_type(wire);
rf_c.set_type(rf_p.get_type());}
}
};
csl_unit alu{
csl_port alu_p(input), alu_c(input);
alu(){
alu_p.set_type(wire);
alu_c.set_type(alu_p.get_type());}
};
```

```
csl_unit wb{
csl_port wb_p(input), wb_c(input);
wb(){
wb_p.set_type(wire);
wb_c.set_type(wb_p.get_type());}
};
csl_unit chip{
csl_signal clk1(8), d_in(8);
im im( .im_p(clk1), .im_c(d_in));
id id( .id_p(clk1),.id_c(d_in));
rf rf( .rf_p(clk1),.rf_c(d_in));
alu alu( .alu_p(clk1),.alu_c(d_in));
wb wb( .wb_p(clk1),.wb_c(d_in));
chip(){
clk1.set_type(wire);}
};
```

VERILOG CODE

```
module im(im_p, im_c);
  input im_p;
  input im_c;
endmodule
module id(id_p, id_c);
  input  id_p;
  input  id_c;
endmodule
module rf(rf_p, rf_c);
  input rf_p;
  input  rf_c;
endmodule
module alu(alu_p, alu_c);
  input  alu_p;
  input  alu_c;
endmodule
module wb(wb_p, wb_c);
  input wb_p;
  input wb_c;
endmodule
module chip();
  wire [7:0] clk1;
  wire [7:0] d_in;
```

```
  im im(.im_p(clk1),.im_c(d_in));
  id id(.id_p(clk1), .id_c(d_in));
  rf rf(.rf_p(clk1), .rf_c(d_in));
  alu alu(.alu_p(clk1), .alu_c(d_in));
  wb wb(.wb_p(clk1), .wb_c(d_in));
endmodule
```

port_object_name.**set_attr**(*csl_port_attr*);

DESCRIPTION :

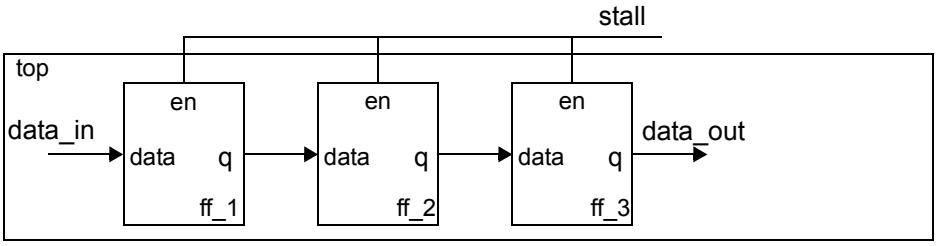
Assign an attribute to *port_object_name*. Attributes describe what the port is used for in the design. . An attribute listing is given in Table 2.12

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1* ,*ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.33



CSL CODE

```
csl_unit ff{
csl_port q(output), data(input), e(input);
ff(){
e.set_attr(clk); }
};
csl_unit top{
csl_signal d_data1,d_data2,data_in,data_out,st;
ff ff_1(.data(data_in),.e(st),.q(d_data1));
ff ff_2(.data(d_data1),.e(st),.q(d_data2));
ff ff_3(.data(d_data2),.e(st),.q(data_out));
top(){ }
};
```

VERILOG CODE

```
module ff(q,
          data,
          e);
  input data;
  input e;
  output q;
endmodule
```

```
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

csl_port_attr port_object_name.**get_attr**();

DESCRIPTION :

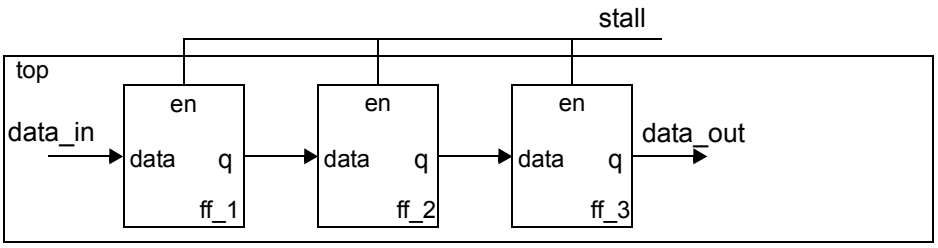
Returns the attribute of *port_object_name*. Attributes describe what the port is used for in the design. An attribute listing is given in Table 2.12.
It can only be called on a port object and returns an object of type *csl_port_attr* .

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1, ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.34



CSL CODE

```
csl_unit ff{
csl_port q(output), data(input), e(input);
ff(){
q.set_attr(en);
e.set_attr(st.get_attr());
}
};
csl_unit top{
csl_signal d_data1,d_data2,data_in,data_out,st;
ff ff_1(.data(data_in),.e(st),.q(d_data1));
ff ff_2(.data(d_data1),.e(st),.q(d_data2));
ff ff_3(.data(d_data2),.e(st),.q(data_out));
top(){ }
};
```

VERILOG CODE

```
module ff(q,
          data,
          e);
  input data;
```

```
  input e;
  output q;
endmodule
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

2.2.1.5 Units: Interface

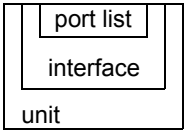
The interface is a container inside of a csl unit which holds all the ports of that unit. The interface sets the connections for a unit.

csl_interface *interface_object_name*;

DESCRIPTION :

Create a new interface object named *interface_object_name*. This object holds the port list for a unit and vector descriptions for the port signals.

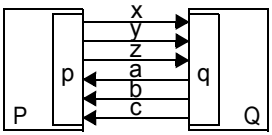
FIGURE 2.35 Interface organization



EXAMPLE :

In this example two interface objects are created, modified and then assigned to different units.

FIGURE 2.36



CSL CODE

```
csl_interface p{
csl_port x(output,1),y(output,1),z(out-
put,1),a(input,1),b(input,1),c(input,1);
p(){ }
};
csl_interface q{
csl_port a(output,1), b(output,1), c(output,1),x(input,1),y(input,1),
z(input,1);
q(){}
};
csl_unit P{
p p0;
P(){}
};
csl_unit Q{
q q0;
Q(){}
};
```

VERILOG CODE

```
module P(p0_x,
         p0_y,
```

```
            p0_z,
            p0_a,
            p0_b,
            p0_c);
    input p0_a;
    input p0_b;
    input p0_c;
    output p0_x;
    output p0_y;
    output p0_z;
  endmodule
  module Q(q0_a,
            q0_b,
            q0_c,
            q0_x,
            q0_y,
            q0_z);
    input q0_x;
    input q0_y;
    input q0_z;
    output q0_a;
    output q0_b;
    output q0_c;
  endmodule
```

`interface_name.reverse()`

DESCRIPTION :

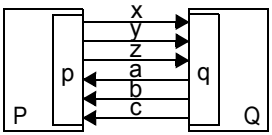
Reverse will invert the input ports and make them output ports, while the output ports will become input ports for *interface_name*.. Reverse will not change inout, tri. Reverse cannot be used with interfaces that have wand or wor types; *interface_object_name* is the name of a list of ports - this will cause a compiler error.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example two interface objects are created, modified and then assigned to different units. Using the method *reverse()* we will change the directions of ports from the instance *q0* of interface *p*. The output ports will become inputs and the input ports will become outputs.

FIGURE 2.37



CSL CODE

```
csl_interface p{
csl_port x(output,1),y(output,1),z(output,1),a(input,1),
b(input,1), c(input,1);
p(){}
};
csl_unit P{
p p0;
P(){}
};
csl_unit Q{
p q0;
Q(){
q0.reverse(); }
};
```

VERILOG CODE

```
module P(p0_x,
         p0_y,
         p0_z,
         p0_a,
         p0_b,
         p0_c);
  input p0_a;
  input p0_b;
  input p0_c;
```

```
  output p0_x;
  output p0_y;
  output p0_z;
endmodule
module Q(q0_a,
         q0_b,
         q0_c,
         q0_x,
         q0_y,
         q0_z);
  input q0_x;
  input q0_y;
  input q0_z;
  output q0_a;
  output q0_b;
  output q0_c;
endmodule
```

```
int interface_object.get_width();
```

DESCRIPTION :

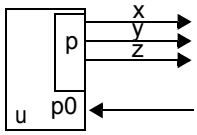
Will return the sum of all widths of the ports in the interface.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows an interface named *p* which contains three ports with different widths, and a port named *p0*. The width of port *p0* is set to be equal with width of interface using the commands *set_width* and *get_width()*.

FIGURE 2.38



CSL CODE

```
csl_interface p{
csl_port x(output, 2), y(output,4), z(output, 8);
p(){}
};
csl_unit u{
p p;
csl_port p0(input);
u(){
p0.set_width(p.get_width());
}
};
```

VERILOG CODE

```
//
```

2.2.1.6 Units: parameter

csl_unit_parameter *parameter_name* **(***default_value***);**

DESCRIPTION :

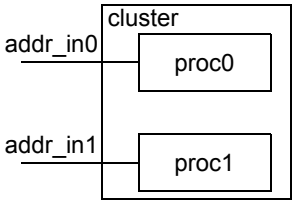
Create a unit parameter in *unit_object_name* with the name *parameter_name* and the default value *default_value*.

[CSL Interconnect Command Summary]

EXAMPLE :

The unit hierarchy example shows how to add parameters to unit declarations.

FIGURE 2.39 Unit hierarchy



CSL CODE

```
csl_unit proc0{
csl_unit_parameter x(5);
csl_port a(input,x);
csl_signal addr_in0(wire,x);
proc0(){ }
};
csl_unit proc1{
csl_unit_parameter x(6);
csl_port a(input,x);
csl_signal addr_in1(wire,x);
proc1(){  }
};
```

VERILOG CODE

```
module proc0(a);
input [4:0] a;
wire [4:0] addr_in0;
endmodule
module proc1(a);
input [5:0] a;
wire [5:0] addr_in1;
endmodule
```

unit_name **#(***parameter_override_value***)** *unit_instance_name*;

DESCRIPTION :

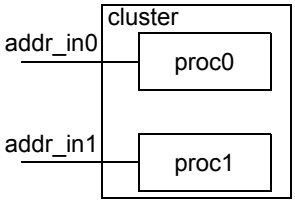
When instantiating a unit, the default value of the units' parameters (if any) can be overridden by using the above syntax. Note that more parameters can be overridden (separated by commas) and the order of the override values, operates on unit parameters in the order the parameters are declared.

[CSL Interconnect Command Summary]

EXAMPLE :

The unit hierarchy example shows how to override the parameters of each instance.

FIGURE 2.40 Unit hierarchy



CSL CODE

```
csl_unit proc0{
csl_unit_parameter x(5);
csl_port a(input,x);
csl_signal addr_in0(wire,x);
proc0(){}
};
csl_unit proc1{
proc0 proc0;
proc1(){
proc1.override_unit_parameter(x,6);
      }
};
```

VERILOG CODE

2.2.1.7 Units: prefix

set_unit_prefix(*prefix_string[,prefix_specifier]*);

DESCRIPTION :

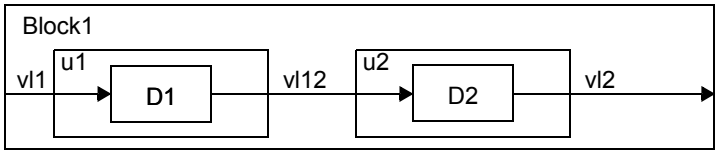
Sets the signals within the *unit_object_name* with the prefix specified by *prefix_string*. Because some signals may be bound to ports, the same prefix is applied to these ports. Optionally the user can choose to apply *prefix_object_name* only to the unit interface or it's local elements by adding the *IFC_ONLY* or *LOCAL_ONLY* prefix pecifiers . Default both specifiers are active.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the prefix " box1" for the unit *Block*.

FIGURE 2.41 An unit named *block1* with 3 instances. Then set a prefix for an unit.



CSL CODE

```
    csl_unit d{
    csl_port d_i(input), d_o(output);
    d(){ }
    };
    csl_unit u1{
    csl_port v1_i(input), v1_o(output);
    csl_signal v1,v12;
    d1 d1(.d_i(v1),.d_o(v12));
    u1(){ }
    };
    csl_unit u2{
    csl_port v2_i(input), v2_o(output);
    csl_signal v12,v2;
    d d2(.d_i(v12),.d_o(v2));
    u2(){ }
    };
    csl_unit block{
    csl_signal vl1,vl12,vl2;
    u1 u1(.v1_i(vl1),.v1_o(vl12));
    u2 u2(.v2_i(vl12),.v2_o(vl2));
    block(){
    set_unit_prefix("box1");
      }
};
```

VERILOG CODE

string *unit*_object_name.**get_unit_prefix**();

DESCRIPTION :

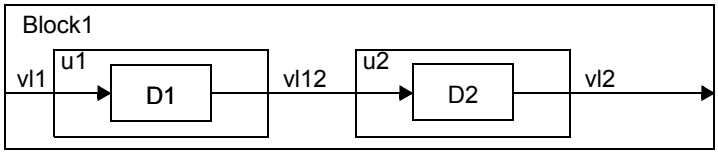
Returns the *unit_object_name*'s *string_prefix*.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the prefix "box1" for the unit *u2* using *get_unit_prefix method()*.

FIGURE 2.42 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){ }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){ }
};
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
u1.set_unit_prefix("box1");
u2.set_unit_prefix(u1.get_unit_prefix(),LOCAL_ONLY); }
```

```
};
```

VERILOG CODE

set_signal_prefix(*prefix_string*);

DESCRIPTION :

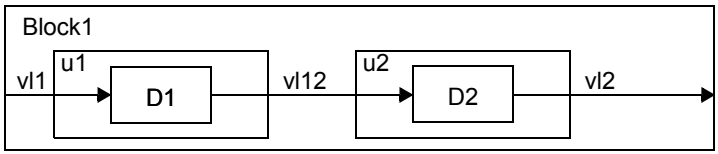
Sets the signals within the *unit_object_name* with the prefix specified by *prefix_string*. Because some signals may be bound to ports, the same prefix is applied to these ports.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a prefix to one or more signals.

FIGURE 2.43 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
set_signal_prefix("uv");
    }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
set_signal_prefix("uv");
    }
};
csl_unit block{
```

```
    csl_signal vl1,vl12,vl2;
    u1 u1(.v1_i(vl1),.v1_o(vl12));
    u2 u2(.v2_i(vl12),.v2_o(vl2));
    block(){
    set_unit_prefix("box1"); }
};
```

VERILOG CODE

```
string signal_object_name.get_signal_prefix();
```

DESCRIPTION :

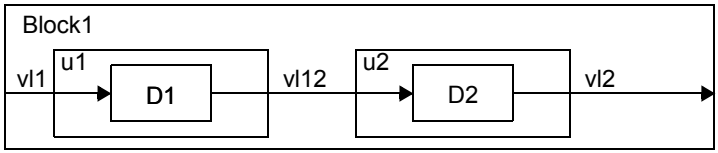
Returns the signal_prefix and the port_prefix previously set by the set_signal_prefix() command.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a prefix to a signal named *v12* using *get_signal_prefix()* method.

FIGURE 2.44 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");
v1.set_signal_prefix(v12.get_signal_prefix());
 }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix("uv2");
v2.set_signal_prefix(v12.get_signal_prefix());
    }
};
```

```
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
set_unit_prefix("box1");
 }
};
```

VERILOG CODE

set_signal_prefix_local(*prefix_string***);**

DESCRIPTION :

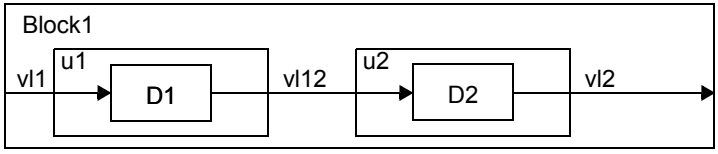
All the local signals previously declared within the same specific unit are prefixed with the *prefix_string* passed as a command argument. The difference between set_signal_prefix() and set_signal_prefix_local is that that the first command prefixes both local signals and ports (ports are signals with direction: input, output or inout, therefore **not** local), and the second command only prefixes local signals.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a local prefix “uv” to v1 and v2 signals .

FIGURE 2.45 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){}
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");
v1.set_signal_prefix_local("uv");
  }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix("uv2");
```

```
    v2.set_signal_prefix_local("uv");
        }
    };
    csl_unit block{
    csl_signal vl1,vl12,vl2;
    u1 u1(.v1_i(vl1),.v1_o(vl12));
    u2 u2(.v2_i(vl12),.v2_o(vl2));
    block(){
    set_unit_prefix("box1");
      }
};
```

VERILOG CODE

string *signal_object_name*.**get_signal_prefix_local();**

DESCRIPTION :

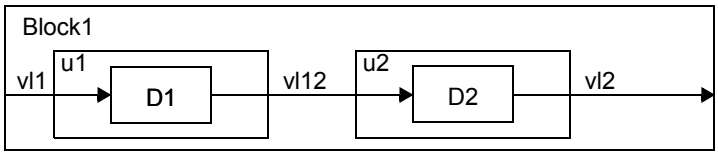
Returns the local signal prefixes that have been set by a previous set_signal_prefix_local() command. This command does not affect ports (ports are not local signals).

[CSL Interconnect Command Summary]

EXAMPLE :

Sets a local prefix “uv2” for signal v2 using the *get_signal_prefix_local()* method.

FIGURE 2.46 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");
v1.set_signal_prefix_local("uv");
 }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix_local("uv2");
v2.set_signal_prefix_local(v12.get_signal_prefix_local());
  }
```

```
};
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
set_unit_prefix("box1");
  }
};
```

VERILOG CODE

2.2.1.8 Units: input/output file type

2.2.1.9 Units: instance control bit

set_instance_alteration_bit(status**);**

DESCRIPTION :

!!add enum table

Set the instance alteration bit to asserted (on) or disserted (off) with the *status* enum parameter. When instance alteration is allowed (on) other objects can be added to instances. Note that this triggers a hierarchical modification down to the unit prototype the instance was derived from. When instance alteration is disallowed (off) instances cannot be modified, except by parameter override methods. Default setting for unit alteration is off (off).

[CSL Interconnect Command Summary]

EXAMPLE :

In this example the instance alteration bit is allowed (on) for the instances of unit *u*.

CSL CODE

```
csl_unit u{
csl_port x(output), y(input);
u(){
set_instance_alteration_bit(on);
  }};
```

VERILOG CODE

```
module u(x,
         y);
  input y;
  output x;
endmodule
```

2.2.1.10 Signal operations

signal_object_name.**merge(***merge_op*,*list_of_signals***);**

DESCRIPTION :

Performs the operation merge_*op* on the signals in *list_of_signals* and assigns the output of the merge operation to *signal_object_name*. The merge operations are in the following table.

TABLE 2.16 The merge operations

merge_op
not
and
or
xor
nand
nor
xnor
plus

[CSL Interconnect Command Summary]

EXAMPLE :

Performs the merge operations "*xor*" between three signals and *"or"* between two signals. The results of operations is *sig1* for *"xor"* and *sig2* for *"or"*.

CSL CODE

```
csl_unit u1{
csl_signal s1,s2,s3;
csl_signal sig1, sig2;
u1(){
sig1.merge(xor,s1,s2,s3);
sig2.merge(or,s1,s2);
}
}:
```

VERILOG CODE

```
//Verilog code goes here
```

signal_object_name.**merge(***merge_op*, *signal_list_object_name***);**

DESCRIPTION :

Performs the operation *merge_op* on the signals in *list_of_signals* and assigns the output of the merge operation to *signal_object_name.*The merge operations are in the following table.

TABLE 2.17 The merge operations

merge_op
not
and
or
xor
nand
nor
xnor
plus

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is performed the merge operations "*or* " and " *and*" between the signals *a1* and *a2* from a signal list named *s1*.The result is signal *sig.*

CSL CODE

```
csl_unit u1{
csl_signal_list sl(a1, a2);
csl_signal sig;
u1(){
sig.merge(or, sl);
sig.merge(and, sl);
}
};
```

VERILOG CODE

```
//Verilog code goes here
```

2.2.1.11 New commands

```
(unit_name | unit_instance_name).set_unit_id(numeric_expression);
```

DESCRIPTION :

Set unit ID to numeric expression.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is set the unit ID “4” for the instance of unit *u*.

CSL CODE

```
csl_unit u{
u(){}
};
csl_unit top1{
u u1;
top1(){
u1.set_unit_id(4);}
};
```

VERILOG CODE

```
module u();
endmodule
module top1();
  u u1();
endmodule
```

```
unit_name.add_logic(external_unit_enable);
```

DESCRIPTION :

This generates a port called *unit_name*_en. This port is an enable signal for the internal unit address decoder.

[CSL Interconnect Command Summary]

EXAMPLE :

Adds a port named *port_en*

CSL CODE

```
csl_unit u{
csl_port port_en(input);
u(){
port_en.add_logic(external_unit_enable);
}
};
```

VERILOG CODE

```
//Verilog code goes here
```

```
(unit_name | instance_name).add_logic(unit_address_decoder,
address_signal_name);
```

DESCRIPTION :

This generates a unit address decoder which is optionally enabled by *unit_name*_en. The input to the address decoder is an input port or signal named address_signal_name. The outputs of the decoder are named *unit_name*_addr_dec_[0-2^n-1] where n is the width of the address_signal_name.

TABLE 2.18

Memory map type	Unit enable required	Source
flat	no	none
virtual	no	upper address bits
hierarchical	yes	input port name

need to put example for each entry in the table

[CSL Interconnect Command Summary]

EXAMPLE :

//small description of the example

CSL CODE

```
csl_unit u{
 u(){}
};
csl_unit top{
csl_signal s(4);
u u0;
top(){
u0.add_logic(unit_address_decoder,s);
}
};
```

VERILOG CODE

```
//Verilog code goes here
```

```
signal_name.(field_name.)*add_logic(gen_decoder);
```

DESCRIPTION :

TABLE 2.19 generated Verilog decoder types

associated enum	Verilog output type
no	wire [2^n-1:0] dec = 1'b1 << sn
yes	case statement

[CSL Interconnect Command Summary]

EXAMPLE :

//small description of the example

CSL CODE

```
csl_unit u{
csl_signal sig;
u(){
sig.add_logic(gen_decoder);
}
};
```

VERILOG CODE

```
//Verilog code goes here
```

2.2.1.12 Merge command

Merge commands are used to merge *n* signals together. The type of merge operation performed is one of the following:

- logical_op *op* use the *logical op* to reduce the n signals to 1 bit
- binary_op *op* var or const to produce 1 signal

The signals are individually *op*-ed with the *var or constant* to produce a new vector of n signals. Note that the var or const is either 1 bit or n bits wide register. Concatenate the n signals together and register the concatenated vector.
And, Or, Mux, Decoder, Xor
Operation modes: bitwise and logical.

CHAPTER 3 CSL FIFO

TABLE 3.1 Chapter Overview

3.1 CSL FIFO Command Summary
3.2 CSL FIFO Commands

3.1 CSL FIFO Command Summary

3.1.1 Usage tables

CSL Fifo
can be defined and instantiated
- can be defined in

TABLE 3.2 CSL unit definition in other CSL classes

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	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

- can be instantiated in

TABLE 3.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
global scope	-
CSL Unit	YES
CSL Signal Group	-
CSL Interface	-
CSL Testbench	YES
CSL Vector	-

TABLE 3.3 CSL unit instantiation in other CSL classes

CSL class	Can be instantiated in
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	-

CSL FIFO Architecture

```
csl_fifo fifo_name (width,depth);
add_logic(programmable_depth, default_depth);
set_physical_implementation(SRAM | FFA);
add_logic(priority_bypass);
add_logic(sync_fifo|async_fifo);
add_logic(depth_extend,numeric_expression);
add_logic(width_extend,numeric_expression);
add_logic(wr_hold, numeric_expession);
set_prefix(prefix_name);
```

CSL FIFO Data

```
add_logic(parallel_output, all | vector_of_addresses);
add_logic(parallel_input,all | vector_of_addresses);
add_logic(rd_words,address_range);
add_logic(wr_words,address_range);
add_logic(sram_rd);
add_logic(sram_wr);
add_logic(async_reset);
```

CSL FIFO Control

```
add_logic(pushback);
add_logic(flow_through, numeric_expression);
add_logic(stall);
add_logic(stall_rd_side);
add_logic(stall_wr_side);
add_logic(wr_release);
```

CSL FIFO Status

```
add_logic(almost_empty,address);
add_logic(almost_full,address);
add_logic(output_wr_addr);
add_logic(output_rd_addr);
add_logic(credit);
add_logic(rd_credit);
add_logic(wr_credit);
add_logic(flow);
```

CSL FIFO Custom Port naming

```
set_reset_name(string);
string fifo_hid.get_reset_name();
set_clock_name(string);
string fifo_hid.get_clock_name();
set_rd_clock_name(string);
string fifo_hid.get_rd_clock_name();
set_wr_clock_name(string);
string fifo_hid.get_wr_clock_name();
set_push_name(string);
string fifo_hid.get_push_name();
set_pop_name(string);
string fifo_hid.get_pop_name();
set_full_name(string);
string fifo_hid.get_full_name();
set_empty_name(string);
string fifo_hid.get_empty_name();
set_wr_data_name(string);
string fifo_hid.get_wr_data_name();
set_rd_data_name(string);
string fifo_hid.get_rd_data_name();
set_valid_name(string);
string fifo_hid.get_valid_name();
```

The following are the ports automatically generated when a csl_fifo is instantiated:
//move signals that don't have examples here

3.2 CSL FIFO Commands

csl_fifo *fifo_name* (width,depth);

DESCRIPTION :

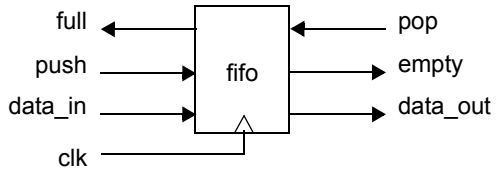
Creates a FIFO named *fifo_name.* Use a read and write pointer and full and empty signal to control the fifo**.** The param width and depth there are numeric expresion and are mandatory for FIFO.

[CSL FIFO Command Summary]

EXAMPLE :

In this example it is created a fifo named *fifo_name.*

FIGURE 3.1



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
}
};
```

VERILOG CODE

```
module fifo_name(data_out, empty, full, data_in, push, pop, clk,
reset);
    parameter stack_width=4;
    parameter stack_depth=16;
    parameter stack_ptr_width=4;

    input [stack_width-1: 0] data_in;
    input push, pop, reset, clk;
    output full, empty;
    output [stack_width-1:0] data_out;

    reg [stack_width-1:0] data_out;
    //gap between wr pointer and rd pointer
    reg [stack_ptr_width:0] gap;

    reg [stack_width-1:0] fifo[0:stack_depth-1];
    reg [stack_width-1:0] stack_widthr, rd;
```

```
assign empty=(gap==0);
assign full=(gap==stack_depth);
    always@(posedge clk or negedge reset)
  begin
      if(reset==0) begin
        data_out=0;
        gap=0;
        stack_widthr=0;
        rd=0;
      end
      else
        begin
            if(!full & push & !pop) begin
                fifo[stack_widthr]=data_in;
                stack_widthr=stack_widthr+1;
                gap=gap+1;
            end
            else if(!empty & pop & !push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
            else if(!full & pop & push) begin
                data_out=fifo[rd];
                fifo[stack_widthr]=data_in;
                rd=rd+1;
                stack_widthr=stack_widthr+1;
            end
            else if(empty & pop & push) begin
                fifo[stack_widthr]=data_in;
                stack_widthr=stack_widthr+1;
                gap=gap+1;
            end
            else if(full & pop & push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
        end
 end
```

```
endmodule
```

add_logic(programmable_depth, *default_depth*);
port: input - programmable_depth

DESCRIPTION :

The FIFO depth is controlled by an input to the fifo. The write and read pointers are reseted when their value equals the value of *signal_name.* The *default_depth* is the maximum depth for the fifo, and the maximum value that *signal_name* can have.

[CSL FIFO Command Summary]

EXAMPLE :

Create a fifo that uses *prg_depth* input signal to controll the depth of the fifo. //desen

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
//size_sgn input signal controlls the depth of the fifo
add_logic(programable_depth,1);
set_width(4);}
};
```

VERILOG CODE

```
module fifo_name(data_out, empty, full, data_in, prg_depth, push, pop,
clk, reset);
    parameter stack_width=4;
    parameter stack_depth=16;
    parameter stack_ptr_width=4;
    input [stack_width-1: 0] data_in;
    input push, pop, reset, clk;
    input [stack_ptr_width:0] prg_depth;
    output full, empty;
    output [stack_width-1:0] data_out;
    reg [stack_width-1:0] data_out;
    reg [stack_ptr_width:0] gap;
    reg [stack_width-1:0] fifo[0:stack_depth-1];
    reg [stack_ptr_width-1:0] wr, rd;
    assign empty=(gap==0);
    assign full=(gap==stack_depth);
    always@(posedge clk or negedge reset)
      begin
        if(reset==0) begin
          data_out=0;
          gap=0;
          wr=0;
```

```
        rd=0;
      end
      else
        begin
            if(!full & push & !pop) begin
                fifo[wr]=data_in;
                wr=wr+1;
                gap=gap+1;
            end
            else if(!empty & pop & !push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
            else if(!full & pop & push) begin
                data_out=fifo[rd];
                fifo[wr]=data_in;
                rd=rd+1;
                wr=wr+1;
            end
            else if(empty & pop & push) begin
                fifo[wr]=data_in;
                wr=wr+1;
                gap=gap+1;
            end
            else if(full & pop & push) begin
                data_out=fifo[rd];
                rd=rd+1;
                gap=gap-1;
            end
            if(rd==(prg_depth-1)) begin
                rd=0;
                $display("rd= %d - 1", prg_depth);
            end
            if(wr==(prg_depth-1)) begin
                wr=0;
                $display("wr= %d - 1", prg_depth);
            end
        end
 end
```

```
endmodule
```

set_physical_implementation(*sram | ffa*);

DESCRIPTION :

FIFO implementation type which is either SRAM or flip flop array.

[CSL FIFO Command Summary]

EXAMPLE :

Sets FIFO implementation which is SRAM.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
set_physical_implementation(sram);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(priority_bypass);
   port: input - priority_select
   port: output - priority_bypass
```

DESCRIPTION :

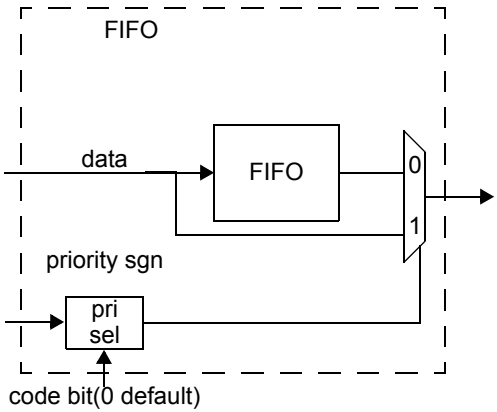
This signal is asserted at the same time as a write to the fifo. When asserted the write to the FIFO is sent to the input of the high prioriy bypass unit.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a priority bypass to the output port of a fifo.

FIGURE 3.2 Bypass



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
add_logic(priority_bypass);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(sync_fifo|async_fifo);

DESCRIPTION :

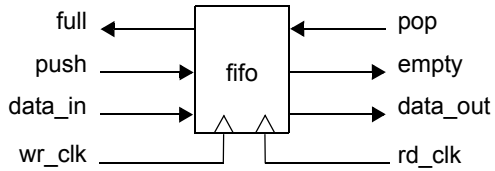
Generate an asynchronous FIFO architecture. Requires the *fifo_name* .wr/rd_clk signals be specified. The read and write counters will be clocked by their respective clocks.
The read and write counters will be grey coded to avoid glitches.
The read and write counters will be synchronized to the other clock domain prior to being compared to the opposite counter.

[CSL FIFO Command Summary]

EXAMPLE :

In this example is generated an asynchronous FIFO architecture.

FIGURE 3.3



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(4);
add_logic(async_fifo);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(depth_extend,*numeric_expression***);**

DESCRIPTION :

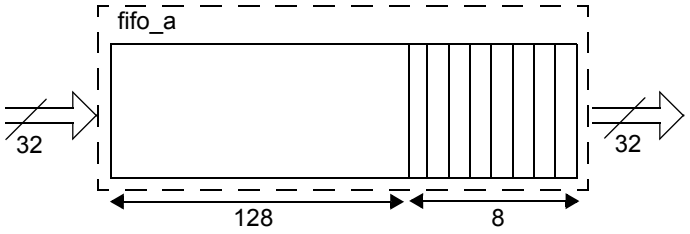
Extend the depth of the FIFO by adding a number of registers to the fifo. The registers or FIFO must be the same width as initial fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Extend *fifo_a* by adding 8 registers. The final depth of *fifo_a* is 136.

FIGURE 3.4 FIFO depth extension



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(128);
set_width(32);
//connect 8 registers to the out of fifo_name
add_logic(depth_extend,8);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(width_extend,*numeric_expression***);**

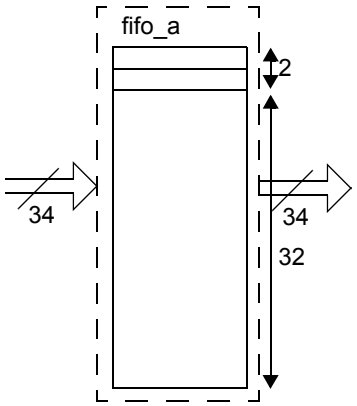
DESCRIPTION :

Extend the width of the FIFO by adding a number of registers FIFO .The registers must be the same depth as initial fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Extend *fifo_a* by adding 2 registers. The final width of *fifo_a* is 34.



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(128);
set_width(32);
//connect 2 registers to the out of fifo_name
add_logic(width_extend,2);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(wr_hold, *numeric_expression*);

DESCRIPTION :

Generate a FIFO with the write hold architecture.

Now we get into the rather esoteric parts of FIFO architeture, design, and implementation. Values may be written by the write side of the FIFO using a push signal. The values are not available to the read of side of the FIFO until the read pointer is allowed to increment into the section of the FIFO memory which contains the write values. In effect we can have a signal which is required after one or more write operations which allows the read pointer to increment into the newly written write region. In other words a write to a FIFO is initiated by a push. The read pointer can access the value written if the < fifo_name > write_hold flag is received. The wr_hold flag will increment a read limit counter which controls the limit count that the read counter can increment to. The wr_addr_hold_limit controls the empty flag. If the FIFO is empty then the pop operation is disabled.

[CSL FIFO Command Summary]

EXAMPLE :

Create a fifo architecture that requires 5 elements in the FIFO between it can be read.

FIGURE 3.5 Write hold architecture

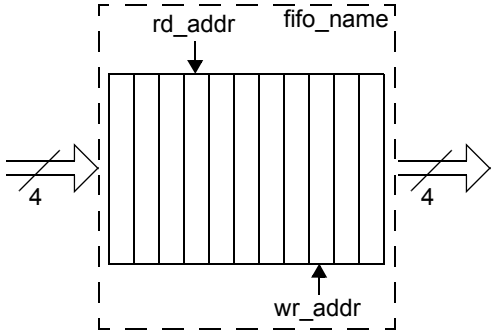


TABLE 3.4 Write hold architecture example

wr_addr	15	16	17	18	19	20	20	21	22	23	23	23	23	23	23	23	23
rd_addr	15	15	15	15	15	15	16	16	16	16	17	18	19	20	21	22	23
rd_wr_gap	0	1	2	3	4	5	4	5	6	7	6	5	4	3	2	1	0
can_pop	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(32);
set_width(4);
add_logic(wr_hold,5);
}
```

```
};
```

VERILOG CODE

```
wire [ADDR_MAX-1:0] rd_addr;
wire [ADDR_MAX-1:0] wr_addr;
wire [ADDR_MAX-1:0] wr_addr_hold_limit     ; // the address of the memory location which was written
wire  [ADDR_MAX-1:0] wr_addr_release_limit; // the address of the current upper limit of released memory locations
wire empty = (rd_addr -wr_addr -1) && (rd_addr !=  wr_addr_hold_limit - 1);
if (wr_release) begin
    wr_addr_hold_limit <=  wr_addr_release_limit;
end
wire rd_addr_inc = pop &&  (rd_addr  <= wr_addr_hold_limit);
```

set_prefix(*prefix_name***);**

DESCRIPTION :

Sets the prefix of all the FIFO signals with *prefix_name.*

EXAMPLE :

Sets the prefix “fifo_a” for the signals of FIFO *fifo_a.*

CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_depth(32);
set_width(4);
set_prefix_name("fifo_a.");
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(parallel_output, all | *vector_of_addresses*);

DESCRIPTION :

Connects the specified FIFO words to individual outputs of the FIFO.

The FIFO read side is an SRAM interface. The FIFO can be read in any order.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are connected all FIFO words to individual outputs.

FIGURE 3.6 FIFO with SRAM read side

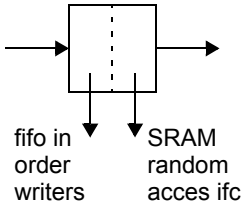
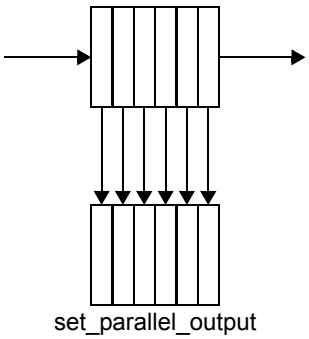


FIGURE 3.7 Paralel output FIFO



CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_width(24);
set_depth(4);
set_parallel_output(all);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(parallel_input,all | *vector_of_addresses*);

DESCRIPTION :

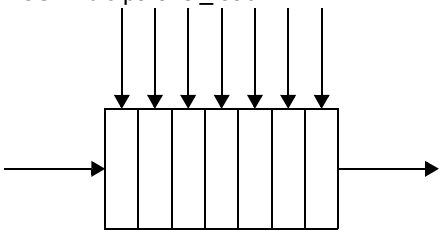
Connects individual inputs of the FIFO to the specified FIFO words.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are connected all FIFO words to individual inputs.

FIGURE 3.8 parallel_load



CSL CODE

```
csl_fifo fifo_a{
fifo_a(){
set_width(24);
set_depth(4);
set_parallel_input(all);
}
};
```

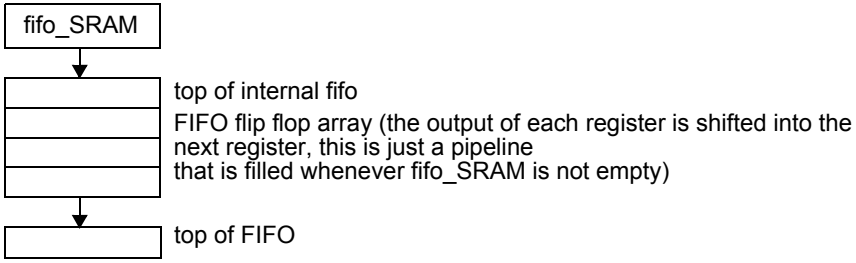
VERILOG CODE

```
//verilog code goes here
```

add_logic(rd_words,*address_range*);

DESCRIPTION :

Connects the FIFO words in the FIFO address range to the interface of the block.
If the architecture type is specified as SRAM then the FIFO words in the address_range are implemented as flip flops. The address range can be up to n (figure out what n can be) words which are implemented as FF's.All words in FIFO are connected to the FIFO block interface.
Connect top words



[CSL FIFO Command Summary]

EXAMPLE :

The words from the fifo address range [0:29] are connected to the interface of the block.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(32);
add_logic(rd_words,0,29);
}
};
```

VERILOG CODE

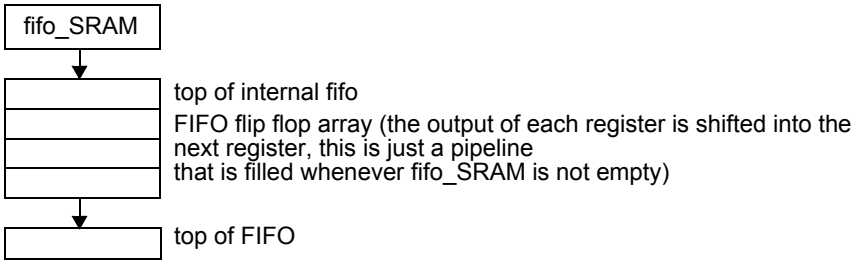
```
//verilog code goes here
```

add_logic(wr_words,*address_range***);**

DESCRIPTION :

Connects the FIFO words in the FIFO address range to the interface of the block.
If the architecture type is specified as SRAM then the FIFO words in the address_range are implemented as flip flops. The address range can be up to n (figure out what n can be) words which are implemented as FF's.

FIGURE 3.9



All words in FIFO are connected to the FIFO block interface.

[CSL FIFO Command Summary]

EXAMPLE :

The words from the fifo address range [0:32] are connected to the interface of the block.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(wr_words,0,32);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(sram_rd);
   port: input - <sram_rd>_en
   port: input - <sram_rd>_addr
   port: output - <sram_rd>_data
```

DESCRIPTION :

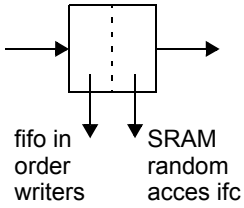
The FIFO read side is an SRAM interface. The FIFO can be read in any order.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an SRAM interface for a FIFO read side.

FIGURE 3.10 FIFO with SRAM read side



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(2);
set_width(4);
add_logic(sram_rd);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

NOTE: add the options for the SRAM read signals which are the same as the CSL SRAM read signals

```
add_logic(sram_wr);
   port: input - <sram_wr>_addr
   port: input - <sram_wr>_data
   port: input - <sram_wr>_en
```

DESCRIPTION :

The FIFO write side is an SRAM interface. The FIFO can be written in any order.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an SRAM interface for a FIFO read side.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(2);
set_width(4);
add_logic(sram_wr);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

NOTE: add the options for the SRAM write signals which are the same as the CSL SRAM write signals

```
add_logic(async_reset);
   port: input - async_reset
```

DESCRIPTION :

This is an asynchronous reset command. When the reset signal is on the low level (logic “0“) the fifo is filled up with 0 values, the clock signal edge doesn’t matter.

[CSL FIFO Command Summary]

EXAMPLE :

Adds an asynchonous reset for the FIFO named *fifo_name*.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(2);
set_width(4);
add_logic(async_reset);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(pushback);
   port: input - pushback
```

DESCRIPTION :

Push back the value popped off of the top of the fifo. Don't really push it back. Instead move the read pointer back one position. This feature requries that the full and empty signals are generated from logic that seperates the write and the read pointers by at least one postion to allow the push_back operation to change the read pointer.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the read pointer is pushed back in a fifo named *fifo_name.*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(pushback);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(flow_through, *numeric_expression*);

DESCRIPTION :

The FIFO is popped automatically whenever there are at least *numeric_expression* valid words in the FIFO. The FIFO can be controlled by the stall signal.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the FIFO is popped automatically when there are at least 9 valid words in the FIFO named *fifo_name*.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(flow_through,9);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall);
   port: input - stall
```

DESCRIPTION :

The entire FIFO is stalled and pop/push operations are not allowed. This signal is used for flow through fifos which pop themselves whenever the FIFO is not empty.

[CSL FIFO Command Summary]

EXAMPLE :

In this example the FIFO named *fifo_name* is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall_rd_side);
   port: input - stall_rd_side
```

DESCRIPTION :

The read side of the FIFO is stalled and FIFO pop operations are not allowed.

[CSL FIFO Command Summary]

EXAMPLE :

For the FIFO named *fifo_name*, the read side is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall_rd_side);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(stall_wr_side);
   port: input - stall_wr_side
```

DESCRIPTION :

The write side of the FIFO is stalled and FIFO push operations are not allowed .

[CSL FIFO Command Summary]

EXAMPLE :

For the FIFO named *fifo_name*, the write side is stalled.

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(stall_wr_side);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(wr_release);
   port: input - wr_release
```

DESCRIPTION :

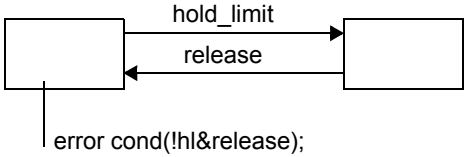
This will set the wr_addr_release_limit register equal to the current wr_addr_hold_limit used in conjuction with the wr_hold_arch switch.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the write address released for the FIFO *fifo_name.*

FIGURE 3.11 Write release
1.hold data
2.release data



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(8);
set_width(10);
add_logic(wr_release);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

add_logic(almost_empty,*address*);
port: output - almost_empty

DESCRIPTION :

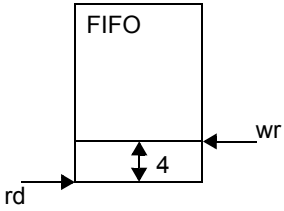
An almost_empty signal is generated when the FIFO almost empty address is reached.
If the FIFO is asynchronous then the signal is generated in the read clock domain.
Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_empty signal is generated.
Optionally use *name* as the name of the almost_empty signal.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an almost empty signal for a FIFO named *fn.*

FIGURE 3.12



CSL CODE

```
csl_fifo fn{
fn(){
set_depth(32);
set_width(4);
//add optional FIFO signals
add_logic(almost_empty, 4);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(almost_full,address);
   port: output - almost_full
```

DESCRIPTION :

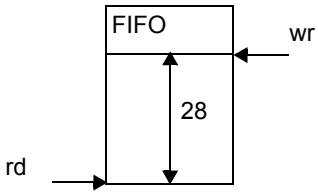
A almost full signal is generated when the FIFO almost full address is reached
If the FIFO is asynchronous then then the signal is generated in the read clock domain.
Note that there is a delay of n cycles, where n is the synchronization delay of the FIFO write address, until the almost_full signal is generated.
Optionally use *name* as the name of the almost_full signal.

[CSL FIFO Command Summary]

EXAMPLE :

Sets an almost full signal for a FIFO named *fn.*

FIGURE 3.13 Almost full



CSL CODE

```
csl_fifo fn{
fn(){
set_depth(32);
set_width(4);
//add optional FIFO signals
add_logic(almost_full,28);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(output_wr_addr);
   port: output - wr_addr
```

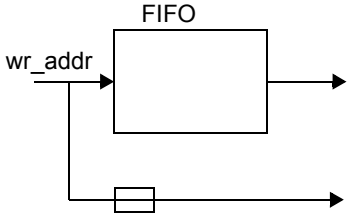
DESCRIPTION :

The FIFO write address is available at the FIFO interface.

EXAMPLE :

Sets the output write address for the FIFO *fifo_name.*

FIGURE 3.14 Status write address



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(output_wr_addr);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(output_rd_addr);
   port: output - rd_addr
```

DESCRIPTION :

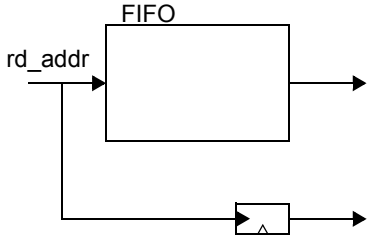
The FIFO read address is available at the FIFO interface.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the output read address for the FIFO *fifo_name*

FIGURE 3.15 Status read address



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(output_rd_addr);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(credit);
   port: output - credit
```

DESCRIPTION :

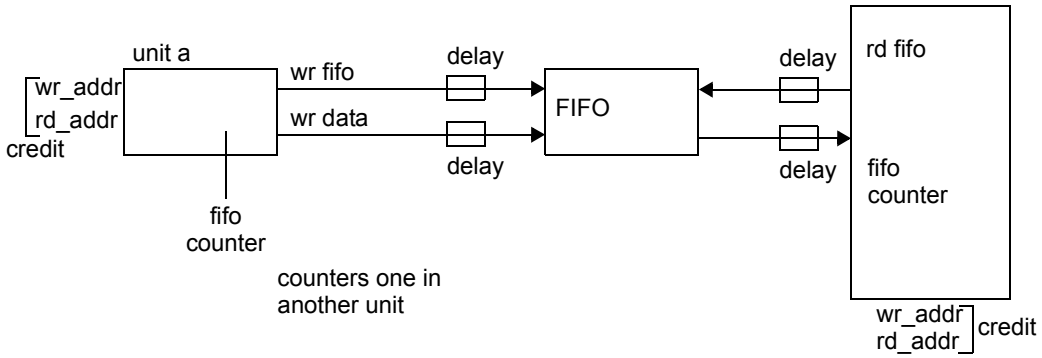
Use a distributed credit debit mechanism to control the fifo. No full or empty signals are generated. Instead FIFO write status/control is handled by the producer and the FIFO read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO named *fifo_name.*

FIGURE 3.16 Credit debit mechanism



CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(rd_credit);
   port: output - rd_credit
```

DESCRIPTION :

Use a distributed credit debit mechanism to cotnrol the FIFO read. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO read named *fifo_name.*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(rd_credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(wr_credit);
   port: output - wr_credit
```

DESCRIPTION :

Use a distributed credit debit mechanism to cotnrol the FIFO write. No full or empty signals are generated. Instead fifo write status/control is handled by the producer and the fifo read status and control is handled by the consumer.

[CSL FIFO Command Summary]

EXAMPLE :

Adds a credit debit mechanism to control a FIFO write named *fifo_name*

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(wr_credit);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

```
add_logic(flow);
   port: output - overflow
   port: output - underflow
```

DESCRIPTION :

When the fifo is full and the upstream device sends a push request, the overflow signal will be validated so that the upstream device will know that it can not push more data into the fifo.
When the fifo is not full the underflow signal is valid.

[CSL FIFO Command Summary]

EXAMPLE :

In this example are activated the flow signals for a FIFO named *fifo_name.*

FIGURE 3.17

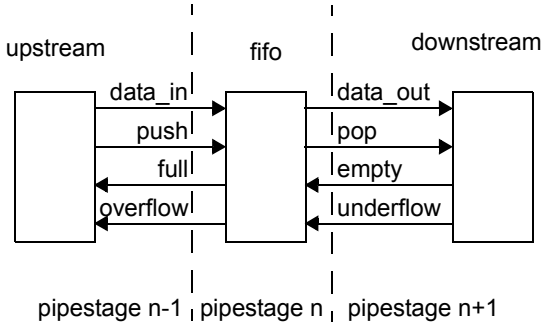


FIGURE 3.18

overflow	output	1	add_logic(flow);
underflow	output	1	add_logic(flow);

CSL CODE

```
csl_fifo fifo_name {
fifo_name(){
set_depth(16);
set_width(32);
add_logic(flow);
}
};
```

VERILOG CODE

```
//verilog code goes here
```

set_reset_name(string);

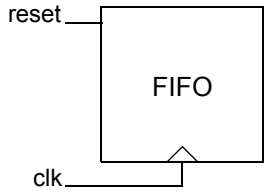
DESCRIPTION :

It sets the *name* for the reset port of fifo.

EXAMPLE :

Sets the name “rst” for reset port of a FIFO named *FO*.

FIGURE 3.19



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_reset_name("rst");
}
};
```

VERILOG CODE

```
string fifo_hid.get_reset_name();
```

DESCRIPTION :

It gets the *name* for the reset port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the reset name for a unit named *u* using get_reset_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_reset_name("rst");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_reset_name());}
};
```

VERILOG CODE

set_clock_name(string);

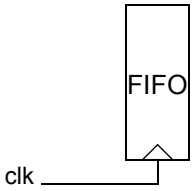
DESCRIPTION :

It sets the *name* for the clock port of fifo.

EXAMPLE :

Sets the name “clk” for clock port of a FIFO named *fifo_name.*

FIGURE 3.20



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_clock_name("clk");
}
};
```

VERILOG CODE

```
string fifo_hid.get_clock_name();
```

DESCRIPTION :

It get the *name* for the clock port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the clock name for a unit named *u* using get_clock_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_clock_name("clk");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_clock_name());  }
};
```

VERILOG CODE

set_rd_clock_name(string);

DESCRIPTION :

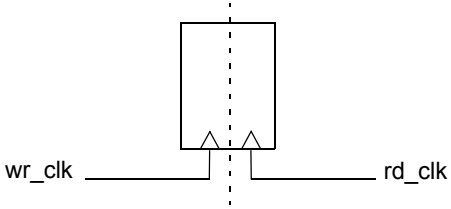
It sets the *name* for the read clock port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the name “rd_clk” for the read clock port of FIFO *fifo_name*.

FIGURE 3.21



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_rd_clock_name("rd_clk");
}
};
```

VERILOG CODE

```
string fifo_hid.get_rd_clock_name();
```

DESCRIPTION :

It gets the *name* for the read clock port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the read clock name for a unit named *u* using get_rd_clock_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_rd_clock_name("rd_clk");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_rd_clock_name());  }
};
```

VERILOG CODE

set_wr_clock_name(string);

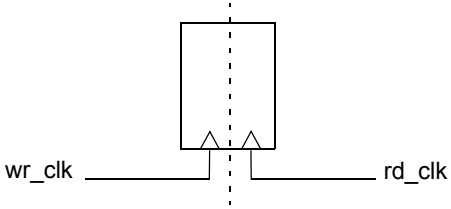
DESCRIPTION :

It sets the *name* for the write clock port of fifo.

EXAMPLE :

Sets the name “wr_clk” for the write clock port of FIFO *fifo_name.*

FIGURE 3.22



CSL CODE

```
csl_fifo fifo_name{
fifo_name(){
set_width(32);
set_depth(4);
set_wr_clock_name("wr_clk");
}
};
```

VERILOG CODE

```
string fifo_hid.get_wr_clock_name();
```

DESCRIPTION :

It gets the *name* for the write clock port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the write clock name for a unit named *u* using get_wr_clock_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_wr_clock_name("wr_clk");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_wr_clock_name());  }
};
```

VERILOG CODE

set_push_name(string);

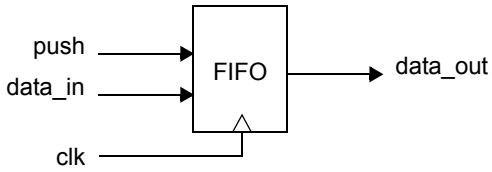
DESCRIPTION :

It sets the *name* for the push port of fifo.

EXAMPLE :

Sets the name “push” for the push port of FIFO *fifo_name*.

FIGURE 3.23



CSL CODE

```
csl_fifo fifo_name{
f(){
set_width(32);
set_depth(4);
set_push_name("push");
}
};
```

VERILOG CODE

```
string fifo_hid.get_push_name();
```

DESCRIPTION :

It gets the *name* for the push port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the push port name for a unit named *u* using get_push_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_push_name("push");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_push_name());}
};
```

VERILOG CODE

set_pop_name(string);

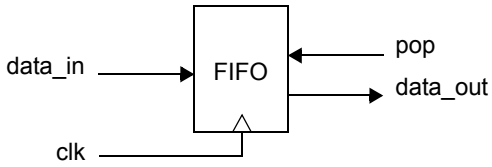
DESCRIPTION :

It sets the *name* for the pop port of fifo.

EXAMPLE :

Sets the name “pop” for the pop port of fifo *FO.*

FIGURE 3.24



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_pop_name("pop");
}
};
```

VERILOG CODE

```
string fifo_hid.get_pop_name();
```

DESCRIPTION :

It gets the *name* for the pop port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the pop port name for a unit named *u* using get_pop_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_pop_name("pop");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_pop_name());  }
};
```

VERILOG CODE

set_full_name(string);

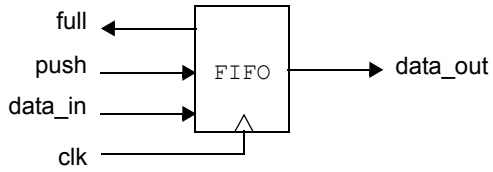
DESCRIPTION :

It sets the *name* for the full port of fifo.

EXAMPLE :

Sets the name “full” for the full port of fifo *FO.*

FIGURE 3.25



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_full_name("full");
}
};
```

VERILOG CODE

```
string fifo_hid.get_full_name();
```

DESCRIPTION :

It gets the *name* for the full port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the full port name for a unit named *u* using get_full_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_full_name("full");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_full_name());  }
};
```

VERILOG CODE

set_empty_name(string);

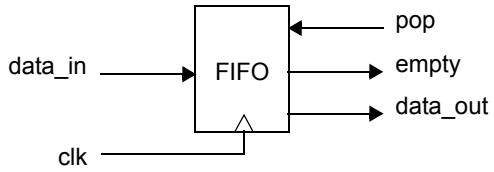
DESCRIPTION :

It sets the *name* for the empty port of fifo.

EXAMPLE :

Sets the name “empty” for the empty port of fifo *FO*.

FIGURE 3.26



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_empty_name("empty");
}
};
```

VERILOG CODE

```
string fifo_hid.get_empty_name();
```

DESCRIPTION :

It gets the *name* for the empty port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the empty port name for a unit named *u* using get_empty_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_empty_name("empty");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_empty_name()); }
};
```

VERILOG CODE

```
set_wr_data_name(string);
```

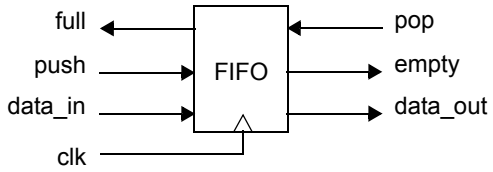
DESCRIPTION :

It sets the *name* for the wr_data port of fifo.

EXAMPLE :

Sets the name “write_data” for the wr_data port of fifo *FO.*

FIGURE 3.27



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_wr_data_name("write_data");
}
};
```

VERILOG CODE

```
string fifo_hid.get_wr_data_name();
```

DESCRIPTION :

It gets the *name* for the wr_data port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the write data port name for a unit named *u* using get_wr_data_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_wr_data_name("write_data");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_wr_data_name());  }
};
```

VERILOG CODE

set_rd_data_name(string);

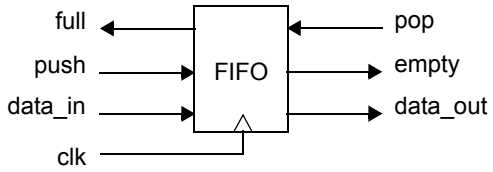
DESCRIPTION :

It sets the *name* for the rd_data port of fifo.

EXAMPLE :

Sets the name “read_data” for the rd_data port of fifo *FO.*

FIGURE 3.28



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_rd_data_name("read_data");
}
};
```

VERILOG CODE

```
string fifo_hid.get_rd_data_name();
```

DESCRIPTION :

It gets the *name* for the rd_data port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the read data port name for a unit named *u* using get_rd_data_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_rd_data_name("read_data");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_rd_data_name());  }
};
```

VERILOG CODE

set_valid_name(string);

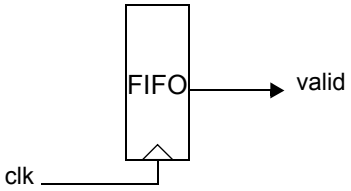
DESCRIPTION :

It sets the *name* for the valid port of fifo.

EXAMPLE :

Sets the name “valid” for the valid port of fifo *FO.*

FIGURE 3.29



CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_valid_name("valid");
}
};
```

VERILOG COD

```
string fifo_hid.get_valid_name();
```

DESCRIPTION :

It gets the *name* for the valid port of fifo.

[CSL FIFO Command Summary]

EXAMPLE :

Sets the valid port name for a unit named *u* using get_valid_name() method.

CSL CODE

```
csl_fifo FO{
FO(){
set_width(32);
set_depth(4);
set_valid_name("valid");
}
};
csl_unit u{
csl_signal sig;
FO FO;
u(){
set_signal_prefix(FO.get_valid_name());  }
};
```

VERILOG CODE

CHAPTER 1 CSL Auto Router

TABLE 1.1 Chapter Overview

1.1 CSL Auto Router Command Summary
1.2 CSL Auto Router Commands

1.1 CSL Auto Router Command Summary

Unit: autoconnect

```
unit_object_name.set_auto_router(ar_flag);
unit_object_name.auto_connect_filter(auto_connect_filter_enum);
connection_object_name0.connect(connection_object_name1);
connect(connection_object_name0,connection_object_name1);
connection_object_name.connect({.formal_connection_object_name(
actual_connection_object_name)}+);
autorouter_connect_bus_to_split_bus(ON|OFF);
```

1.2 CSL Auto Router Commands

unit_object_name.**set_auto_router(***ar_flag***);**

DESCRIPTION :

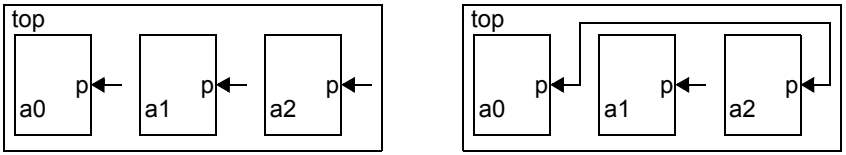
The *ar_flag* can be *on* or *off*.If *ar_flag* is *off*,the auto_router doesn't autoconnect the ports from that unit.If *ar_flag* is *on*,it does.By default the *ar_flag* is *on*.

[***CSL Auto Router Command Summary***]

EXAMPLE :

In the example below the unit named top contains three instances of the unit A.This unit can communicate to the outside through the input port p Because we have set to off the auto_router for a1,the auto_router will connect only the ports from a0 and a2.

FIGURE 1.1 Figure before autorouting and after autorouting



CSL CODE:

```
csl_unit a{
csl_port p(input, 4);
a(){}
};
csl_unit top{
a a1;
a a2;
a a0;
top(){
a1.set_auto_router(off);
auto_connect_filter(CSL_CONNECT_PORTS_BY_NAME );
}
};
```

VERILOG CODE:

//

unit_object_name.**auto_connect_filter(**auto_connect_filter_enum**);**

DESCRIPTION :

Filters can be setup within a unit hierarchy and can be then passed as arguments to **auto_connect_filter()** to achieve more control or impose restrictions over the auto connect process

TABLE 1.2 Auto connect enums

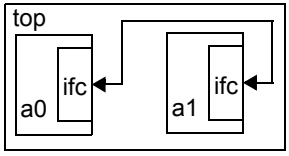
Enum	Description
CSL_CONNECT BY_NAME	Connects interfaces, signal groups, signals and ports by name
CSL_CONNECT_PORTS_BY_NAME	Connects ports by name
CSL_CONNECT_SIGNALS_BY_NAME	Connect signals and signal groups by name (is this feasible?)
CSL_CONNECT_INTERFACES_BY_NAME	Connect interfaces by name
CSL_CONNECT_FUNCTION	Connects units, instances, interfaces, signal groups, signals by function connect()
VERILOG_CONNECT_BY_NAME	Connect only Verilog modules/instances/ports within the design
VERILOG_CONNECT_PORTS_BY_NAME	Connect only Verilog ports within the design
ALL_CONNECT_BY_NAME	Connect both CSL and Verilog connection objects
CREATE_NEW_ENDPOINT_PORTS	Connection objects which do not exist in a scope are infered to be ports. A new port objecft is generated in the scope. isn't this already being done above ?

[***CSL Auto Router Command Summary***]

EXAMPLE :

In this example two units *a0* and *a1* are connected by interfaces named *ifc*.

FIGURE 1.2



CSL CODE

```
csl_interface ifc{
```

```
csl_port a(input, 4),b(input, 4);
ifc(){}
};

csl_unit a{
ifc ifc;
a(){}
};
csl_unit top{
a a0;
a a1;
top(){
auto_connect_filter(CSL_CONNECT_BY_NAME );
}
};
```

VERILOG CODE

```
//
```

```
connection_object_name0.connect(connection_object_name1);
```

DESCRIPTION :

All elements in the table below are objects (except elements that start with “list_of”).
The signal connection is used to create connections between signals. The signals can be ports or local signals. Connections to ports create the formal to actual mapping between the the port name and signal in the upper level unit. Note that actual names are really expressions. The types of expressions supported for the actual expressions are as follows:

- signal
- signal with bit range
- expression (boolean operation on signals)
- concatentation
- signal name change
- constant

The table Table 1.3 contains the list of connection objects and the allowed connections between connection object types.

TABLE 1.3 Valid connections between connection objects

Nr	LHS \ RHS	1 s	2 sg	3 p	4 ifc	5 u	6 ui	7 sc	8 e
1	signal	X	-	X	X	X	X	X	X
2	signal group	-	X	-	X	X	X	X	-
3	port	X	-	X	X	X	X	X	X
4	interface	-	X	-	X	X*	X*	X*	-
5	unit	X	X	X	X	X	X	X	X
6	unit instance	X	X	X	X	X	X	X	X
7	signal concat	X	X	-	X	X	X	X	X
8	expr	-	-	-	-	-	-	-	-

x? = to be discussed
Connection needs support for signal subranges and expressions
The following may not be needed anymore
Coonect can be used to ‘link’ signals, ports, groups of signals, or unit interfaces (when connect is apllied to unit names it’s still the interfaces of those units that get connected)

```
signal_name.connect(signal_name);
port_name.connect(port_name);
group_of_signals.connect(group_of_signals);
interface.connect(interface);
```

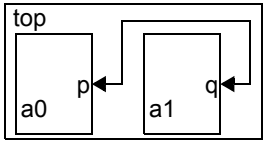
unit_name.connect(unit_name);

[***CSL Auto Router Command Summary***]

EXAMPLE :

In the figure below ports p, q from diferent units are connected.

FIGURE 1.3



CSL CODE

```
csl_unit a0{
csl_port p(input)
a0(){}
};
csl_unit a1{
csl_port q(input)
a1(){}
};

csl_unit top{
a0 a0;
a1 a1;
top(){
ao.p.connect(a1.q);
}
};
```

VERILOG CODE

```
//AV
module top;
    wire y,z;
    ab a1(y);
    ab b1(y);
    c c1(z);
    d d1(z);
endmodule
module ab(y);
    inout y;
endmodule
module c(z);
    inout z;
endmodule
module d(z);
    inout z;
```

```
    wire t;
    assign t = z;
endmodule
```

.

connect(*connection_object_name0*,*connection_object_name1***);**

DESCRIPTION :

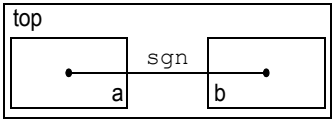
Global connect function.

[***CSL Auto Router Command Summary***]

EXAMPLE :

In this example are connected two signals with the same name *sgn* from different units *a* and *top* or *b* and *top*.

FIGURE 1.4



CSL CODE

```
csl_unit ab{
csl_signal sgn(wire,4)
ab(){}
};
csl_unit top{
a a;
a b;
csl_signal sgn(wire,4)
top(){
connect(sgn,a.sgn);
connect(sgn,b.sgn);
}
};
```

VERILOG CODE

```
module top;
    wire sgn;
    ab a(y);
    ab b(y);
    endmodule
module ab(sgn);
    inout sgn;
endmodule
```

connection_object_name.**connect(**{.*formal_connection_object_name*(*actual_connection_object_name*)}+**);**

DESCRIPTION :

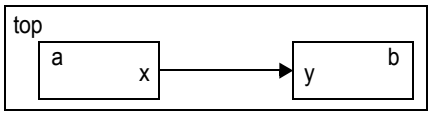
Connections are specified by *formal_connection_object_name* and *actual_connection_object_name* which can be ports or interfaces; *connection_object_name* can be a unit, instance or interface.

[***CSL Auto Router Command Summary***]

EXAMPLE :

In this example are connected two ports . The port *x* from unit *a* and the port *y* from unit *b.*

FIGURE 1.5



CSL CODE

```
csl_unit a{
csl_port x(output)
a(){}
};
csl_unit b{
csl_port y(input)
b(){}
};
csl_unit top{
a a;
b b;
top(){
a.connect(.x(b.y));
}
};
```

VERILOG CODE

```
module top;
    a a(z);
    b b(z);
endmodule
module a(z);
    output x;
endmodule
module b(z);
```

```
    input y;
endmodule
```

autorouter_connect_bus_to_split_bus(ON|OFF);

DESCRIPTION :

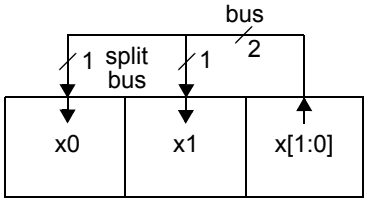
Splits the bus in two or more wires with diferent width.

[***CSL Auto Router Command Summary***]

EXAMPLE :

x[1:0]
x[1] becomes x_1 and x[0] becomes x_0

FIGURE 1.6



//commands
- turn {on/off} (port,signal,interface) inference - specified for or what can be inferred
- connect by (name,function) {on/off}

CSL CODE

```
csl_unit u{
  csl_signal  bus(2);
  csl_signal  x_1(1), X_0(1);
  u(){
    autorouter_connect_bus_to_split_bus(ON);
  }
};
```

CHAPTER 1 CSL Verification Components

TABLE 1.1 Chapter Overview

1.1 CSL Verification Components Syntax and Command Summary
1.2 CSL Verification Components Commands

1.1 CSL Verification Components Syntax and Command Summary

1.1.1 Verification components classes

Verification components are used by testbenches to “feed“ the DUTs (design under test) with test data (stimulus), compare the DUT’s state at a certain time (state data) or check the data output from the DUT(expect). Verification components are declared as classes and so behave as scope holders. There are two main types of classes for verification components: CSL Vector and CSL state Data.

1.1.1.1 CSL Verification Components specific commands

```
set_unit_name(unit_name);
```

1.1.1.2 CSL Vector class

A vector is a collection of signal values sampled at a certain moment in time from a set of selected ports or interfaces (port containers). The actual values contained by verification components vectors are generated automatically by the C++ simulator (Csim). The CSL specification is used to “tune” the vector’s settings and to integrate it inside the testbench (establish connections with the proper ports and signal generators and compare units).

1.1.1.2.1 CSL Vector class declaration

A CSL Vector can only be declared in the global scope just like any other CSL class. The CSL vector class is declared as in the below example:

```
csl_vector vector_class_name {
  //no objects may be instantiated in a CSL vector
  vector_class_name(){
    (vector methods calls)+
  }
};
```

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

In the vector class’ scope there aren’t any objects to be specifically declared or instantiated as

shown in the table below.

TABLE 1.2 Rules for instantiating objects in the vector's scope

CSL class	Is instantiated in CSL Vector scope
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	-

However, objects are added in the vector class scope in a "non-standard" way by calling the set_module_name(), include_only() and exclusion_list() methods. The vector methods calls (commands) are made inside the vector's constructor.

1.1.1.2.2 CSL Vector specific commands

The following commands are specific only to CSL Vector classes and are mandatory:

CSL Vector mandatory commands

```
set_direction(direction);
```

The following methods can be called on a vector class to "adjust" the connection elements a vector controls. That is, if a vector is set to be of type stimulus (input), it will connect to all the input ports of a unit(a DUT in the testbench); if the vector instance is of type expected(output) it will connect to all the output ports of the DUT. If the user does not wish to connect all the ports to the vector (eg. have the clock port be driven by the clock in testbench), or if the user chooses to connect only one sub-interface from the output interface of a DUT to a vector, these methods are used to select the desired connection elements:

CSL Vector optional methods

```
exclusion_list(connection_elements);
include_only(connection_elements);
```

1.1.1.2.3 CSL Vector usage and rules

Vectors are associated with a CSL unit class. It is this unit that holds the ports or interfaces that will be associated with the vector as described later in this command summary. Vectors are used in test-benches, however vector classes are not instantiated.
The rules for vector usage in other CSL classes are contained in the table below:

TABLE 1.3 Vector usage rules

CSL class	Uses CSL Vector
CSL Unit	-
CSL Testbench	YES
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	-

1.1.1.3 CSL State Data class

State data is a collection of the values in a state element. A state element can be a register, register file, fifo or a memory. State data is captured periodically when an associated transaction event occurs. For example, a register file changes state when there is a write enable. The state data file is loaded into the testbench state data memory. Each RTL DUT state data transation is compared against the state data expected results stored in the state data memory. The state data can be regarded as a collection of snapshots of the state element at different moments in time. The actual values contained in the state data snapshot are generated automatically by the C++ simulator (Csim). The CSL specification is used to “tune” state data’s settings and to integrate it inside the test-bench (establish connections with the proper ports and signal generators and state compare units).

1.1.1.3.1 CSL State Data declaration

A CSL State Data can only be declared in the global scope just like any other CSL class. The CSL state data class is declared as in the below example:

```
csl_state_data state_data_class_name {
  //no objects may be instantiated in a CSL vector
  state_data_class_name(){
```

```
        (state data methods calls)+
    }
};
```

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

In the state data class' scope there aren't any objects to be specifically declared or instantiated as shown in the table below.

TABLE 1.4 Rules for instantiating objects in the vector's scope

CSL class	Is instantiated in CSL Vector scope
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	-

1.1.1.3.2 CSL State Data usage and rules

State data is associated with a memory instance. A memory instance can be an instance of a register, register file, fifo or a simple memory. State data is used in testbenches however state data classes are not instantiated.
The rules for state data usage are contained in the table below:

TABLE 1.5 Vector usage rules

CSL class	Uses CSL State Data
CSL Unit	-
CSL Testbench	YES
CSL Vector	-
CSL State Data	-
CSL Register	-

TABLE 1.5 Vector usage rules

CSL class	Uses CSL State Data
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

1.1.1.3.3 CSL State Data specific methods

The following command, specific only to CSL State Data classes, is also mandatory.

CSL State Data mandatory commands

```
set_mem_instance_name(memory_instance_name);
set_vc_unit_name(unit_name);
set_vc_clock(signal);
```

1.1.1.4 CSL Verification components common methods

The following methods apply to both CSL vector and State Data classes.

Common Verification Components commands

```
set_vc_header_comment(string);
set_version(numeric_expression);
set_radix(binary|hex);
set_vc_reset(signal);
set_vc_stall(signal);
set_vc_valid_output_transaction(signal_expression);
set_vc_start_generation_trigger(signal);
set_vc_end_generation_trigger(signal);
set_vc_capture_edge_type(rise|fall);
set_vc_max_number_of_valid_transactions(numeric_expression);
set_vc_max_number_of_mismatches(numeric_expression);
set_vc_max_cycles(numeric_expression);
set_vc_output_filename(filename);
```

Common Verification Components commands

```
int vc_name.get_vc_version();
string vc_name.get_vc_header_comment();
enum vc_name.get_radix();
signal vc_name.get_vc_module();
signal vc_name.get_vc_clock();
signal vc_name.get_vc_reset();
signal vc_name.get_vc_stall();
signal vc_name.get_vc_valid_output_transaction();
signal vc_name.get_vc_start_generation_trigger();
signal vc_name.get_vc_end_generation_trigger();
enum vc_name.get_vc_capture_edge_type();
int vc_name.get_vc_max_number_of_valid_transactions();
int vc_name.get_vc_max_number_of_mismatches();
int vc_name.get_vc_timeout();
string vc_name.get_output_filename();
add_logic(inject_stalls);
```

1.2 CSL Verification Components Commands

set_vc_header_comment(string**);**

DESCRIPTION :

Adds a comment to the top of the vector/state data file. The comment is ignored by the vector reader in the testbench.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.1

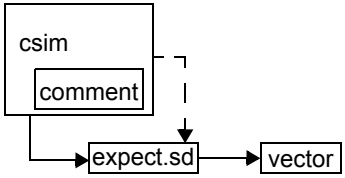
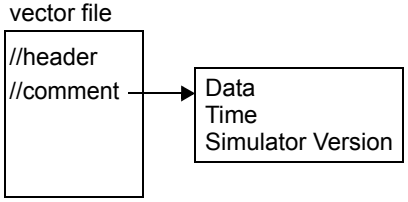


FIGURE 1.2



CSL CODE:

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_header_comment("stimvec");
  }
};
```

VERILOG CODE

```
//
```

```
set_version(numeric_expression);
```

DESCRIPTION :

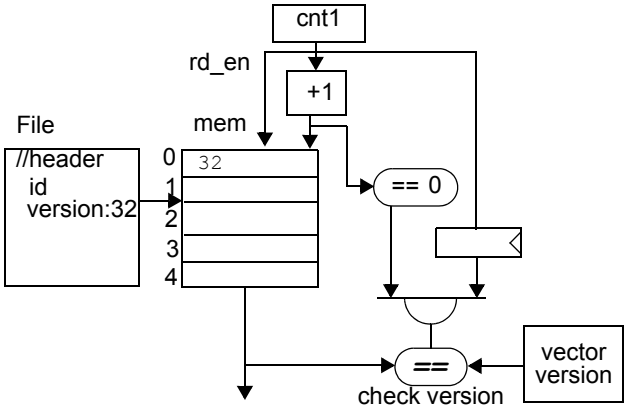
This command sets the version number for the corresponig verification component. This version will be checked against the one from generated vector/state data (from the C++ Simulator) and if a mis-match is detected the generated version checker will halt the simulation.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

FIGURE 1.3 The ID is added to the file header



CSL CODE:

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_version(2);
  }
};
```

VERILOG CODE

```
//
```

set_radix(binary|hex**);**

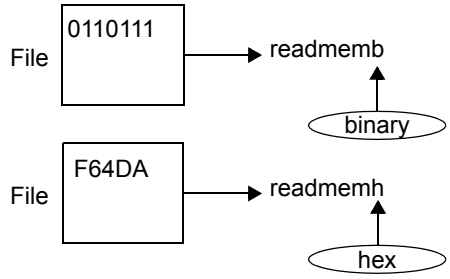
DESCRIPTION :

This option sets the radix for the vector format. The default radix is binary. The vector writer wiil write out the vector in the radix specfied. If the vector radix is binary then the verilog testbench will use the $readmemb function. If the vector radix is hexidecimal then the testbench will use the $readmemh function.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.4



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_radix(hex);
  }
};
```

VERILOG CODE

```
//
```

```
set_vc_unit_name(unit_name);
```

DESCRIPTION :

//

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

set_vc_clock(*signal***);**

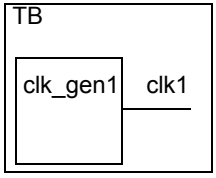
DESCRIPTION :

Sets the clock signal that triggers the capture of the vector or state data. Note that vectors and state data on the generated RTL code operate on different clock signals. Each Testbench unit have one or more clk generators. vc_clock has to be connected to one of the testbench clock signals. It is illegal to connect the vc_clock to any signal declared outside the testbench scope.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.5 One clock generators inside a testbench



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_clock(clk);
      }
};
csl_testbench tb{
csl_signal  clk(wire);
 dut dut;
 stim_vec stim;
 tb(){
   add_logic(clock, clk,10,ns);
}
};
```

VERILOG CODE

```
//timescale 1ns/1ps
module sd;
    reg clk1,clk2;
    initial
```

```
    clk1 = 0;
    always #5 clk1=~clk1;
endmodule
```

set_vc_reset(*signal***);**

DESCRIPTION :

Adds *signal* reset signal to *testbench_object_name*.When the testbench global reset signal is active, the state data and the vectors should be cleared. On top of that, the state dataState data and the vectors have their own reset signals, so they can be reseted even if the glogal reset signal is inactive.

Associate a reset signal with the vector. This command is used to control when to start writing vectors. Start generating vectors after event occurs. This command tells the vector generator to start writing vectors after reset. The typical signal to usee for event is reset which is the default start vector control signal. Many designs generate garbage prior to reset and generating vectors prior to reset is useless for comparison purposes.

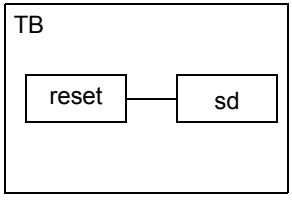
Vc_reset had to be connected to one of the testbench signals.It is ilegal to connect the vc_reset to any signal declared outside the testbench scope.

There is one reset signal in the testbench and that reset drives the testbench, the stimulus and expected vectors and the DUT.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.6 Connect reset to state data



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_reset(r);
     }
};
```

VERILOG CODE

set_vc_stall(signal**);**

DESCRIPTION :

Name of signal that triggers the capture of the vector or state data. Commands the vc_object to stop.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_stall(s);

  }
};
```

VERILOG CODE

```
//
```

set_vc_valid_output_transaction(signal_expression);

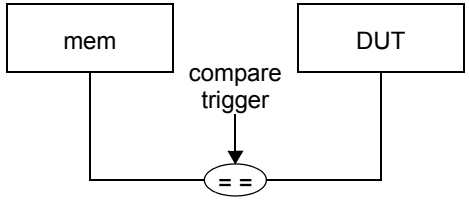
DESCRIPTION :

This command is used to set the method an output transaction is validated by setting signal_expression. Thus, if the expression is a clock, than the transaction is always valid (valid = 1); if the expression is a signal, then the transaction is valid when the signal is true (valid = signal); if the transaction is a signal expression than the transaction is valid when the signal expression is true (eg. valid = x | y | z). This command applies only to output vectors or state data.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.7



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_signal r(wire);
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_valid_output_transactions(r);
  }
};
```

VERILOG CODE

//

set_vc_start_generation_trigger(*signal***);**

DESCRIPTION :

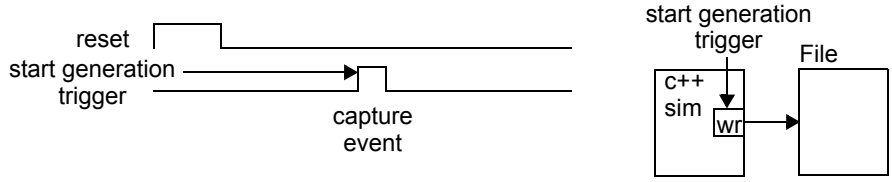
Name of signal that triggers the capture of the vector or state data
Capture or compare the vector or state data on either an event or a clock edge

Overrides the control of start state data generation by reset. Instead start generating state data after event occurs. This command tells the state data generator to start writing state datas. This command is used to control when to start writing state datas. A typical signal to use for event is reset which is the default start state data control signal. Many designs generate garbage prior to reset and generating state datas prior to reset is useless for comparison purposes.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.8



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_signal trigg;
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_start_generation_trigger(trigg);
  }
};
```

VERILOG

```
none
```

```
set_vc_end_generation_trigger(signal);
```

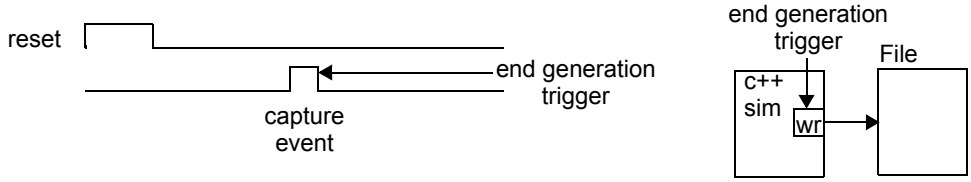
DESCRIPTION :

Name of signal that stops the vector or state data recording (if this is not set by the user then the cslc computes the unique number automatically.-uniquely identifies the state data or vector instance file or socket.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.9



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_signal trigg;
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_end_generation_trigger(trigg);
  }
};
```

VERILOG CODE

```
none
```

set_vc_capture_edge_type(rise|fall);

DESCRIPTION :

Capture the vector or state data on the rising or falling edge of clock.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_capture_edge_type(rise);
  }
};
```

VERILOG CODE

```
none
```

set_vc_max_number_of_valid_transactions(*numeric_expression***);**

DESCRIPTION :

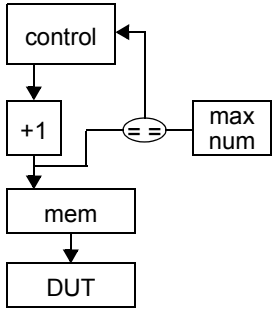
Stop capturing events when maximum number of capture events is reached
The verification transaction are writen to the output file until the maximum number of capture events is reached . After the maximum transact.ion count is reached the C++ vector writer stops writing verfication transactions to the output file or stream.

Stimulus Verification Transaction Counter
The testbench stimulus vector reader counts the number of verification transactions that are read from the stimulus verification transaction input stream or memory and when the transaction count equals max number of vectors no further verification transactions are read and stimulus verification transaction flag is set. The testbench prints a message specifying that the maximum number of stimulus verification transactions has been read.

Expected Verification Transaction Counter
The testbench stimulus vector reader counts the number of verification transactions that are read from the expected verification transaction input stream or memory and when the transaction count equals max number of vectors no further verification transactions are read and expected verification transaction flag is set. The testbench prints a message specifying that the maximum number of expected verification transactions has been read.

[***CSL Verification Components Syntax and Command Summary***]

FIGURE 1.10 Maximum number of vectors



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
```

```
    set_direction(input);
    set_vc_max_number_of_valid_transactions(10);
  }
};
```

VERILOG CODE

set_vc_max_number_of_mismatches(numeric_expression**);**

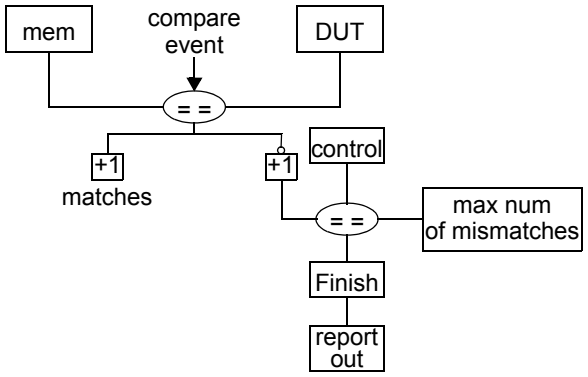
DESCRIPTION :

data-stop state data or vector comparisons after the max number is reached
The comparator in the testbench counts the number of mismatches between the expected vector and the DUT generated vector. When the error count equal the max mismatch count then the simulation stops.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.11



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), stim_v(input);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_max_number_of_mismatches(5);
  }
};
```

VERILOG CODE

```
//
```

set_vc_max_cycles(numeric_expression**);**

DESCRIPTION :

Sets the number of cycles to time out after the last event.

[***CSL Verification Components Syntax and Command Summary***]

FIGURE 1.12

ps	picoseconds
ns	nanosecons
s	seconds

EXAMPLE :

CSL CODE

```
//
```

VERILOG CODE

```
//
```

set_vc_output_filename(filename**);**

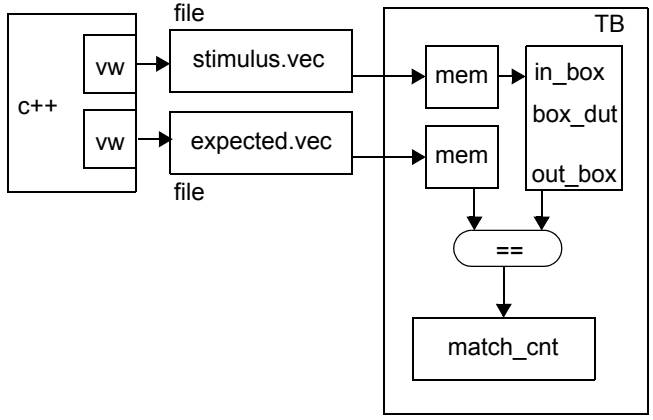
DESCRIPTION :

The name of the file to write the vector or state data to.
* vt in command name comes from verification transaction
Sets the output file. The testbench will write the result of the comparation between the vectors from the DUT output and the expected vector in *filename* .

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.13



CSL CODE:

```
csl_unit dut{
csl_port  exp_out(output),exp_v(output);
 dut() { }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(output);
    set_vc_output_file("expected");;
  }
};
```

VERILOG CODE

int *vc_name*.get_vc_version();

DESCRIPTION :

Return the unique id number for the specified verification component.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

CSL CODE

```
csl_unit dut{
csl_port stim_in(input), stim_v(input), exp_out(output),exp_v(output);
 dut() { }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(output);
    set_version(2);
  }
};
csl_vector stim_vec{
  stim_vec(){
   set_module_name(dut);
    set_direction(input);
    set_version(exp_vec.get_version());
  }
};
```

```
string vc_name.get_vc_header_comment();
```

DESCRIPTION :

Get the comment from the top of the vector/state data file.

[***CSL Verification Components Syntax and Command Summary***]

CSL CODE:

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_header_comment("stimvec");
  }
};
csl_vector exp_vec{
exp_vec(){
   set_module_name(dut);
    set_direction(input);
    set_vc_header_comment(stim_vec.get_vc_header_comment());
```

VERILOG CODE

```
//
```

```
enum vc_name.get_radix();
```

DESCRIPTION :

Get the radix for the vector format.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_radix(hex);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_radix(stim_vec.get_radix());
  }
};
```

VERILOG CODE

```
//
```

```
signal vc_name.get_vc_module();
```

DESCRIPTION :

Get the name of the module that the vector or state data element is associated with.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(stim_vec.get_vc_module());
    set_direction(input);
  }
};
```

VERILOG CODE

```
//
```

```
signal vc_name.get_vc_clock();
```

DESCRIPTION :

Get the name of clock that triggers the capture of the vector or state data.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_signal clk(1);
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_clock(clk);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_clock(stim_vec.get_vc_clock());
  }
};
```

VERILOG CODE

```
//
```

`signal` ***`vc_name`*****`.get_vc_reset();`**

DESCRIPTION :

Get *signal_name* reset signal from *testbench_object_name*.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_signal res(1);
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_reset(res);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_reset(stim_vec.get_vc_reset());
  }
};
```

VERILOG CODE

```
//
```

```
signal vc_name.get_vc_stall();
```

DESCRIPTION :

Get the name of signal that triggers the capture of the vector or state data;

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_signal st(1);
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_stall(st);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_stall(stim_vec.get_vc_stall());
  }
};
```

VERILOG CODE

```
//
```

```
signal vc_name.get_vc_valid_output_transaction();
```

DESCRIPTION :

Get the clock signal or an event object uses to perform the comparisons.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_output_transaction(10);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_output_transaction(stim_vec.get_vc_output_transaction());
  }
};
```

VERILOG CODE

```
//
```

signal vc_name.**get_vc_start_generation_trigger();**

DESCRIPTION :

Get the command used to control when to start writing state datas.

[***CSL Verification Components Syntax and Command Summary***]

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_signal trigg;
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_start_generation_trigger(trigg);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_start_generation_trigger(
              stim_vec.get_vc_start_generation_trigger());
  }
};
```

VERILOG CODE

```
//
```

signal *vc_name*.**get_vc_end_generation_trigger();**

DESCRIPTION :

Get the event that stops the vector or state data recording ;

[***CSL Verification Components Syntax and Command Summary***]

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_signal trigg;
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_end_generation_trigger(trigg);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_end_generation_trigger(
              stim_vec.get_vc_end_generation_trigger());
  }
};
```

VERILOG CODE

```
//
```

enum *vc_name*.**get_vc_capture_edge_type();**

DESCRIPTION :

Get the capture edge type rise or fall of clock of the vector or state data.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_capture_edge_type(rise);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_capture_edge_type(
              stim_vec.get_vc_capture_edge_type());
  }
};
```

VERILOG CODE

```
//
```

```
int vc_name.get_vc_max_number_of_valid_transactions();
```

DESCRIPTION :

Get the maximum number of captured events.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_max_number_of_valid_transactions(10);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_max_number_of_valid_transactions(
              stim_vec.get_vc_max_number_of_valid_transactions());
  }
};
```

VERILOG CODE

```
//
```

```
int vc_name.get_vc_max_number_of_mismatches();
```

DESCRIPTION :

Return the max number of mismatches seted for vector.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_max_number_of_mismatches(10);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_max_number_of_mismatches(
              stim_vec.get_vc_max_number_of_mismatches());
  }
};
```

VERILOG CODE

```
//
```

```
int vc_name.get_vc_timeout();
```

DESCRIPTION :

Get the number of cycles to time out after the last event.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_timeout(10);
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_vc_timeout(stim_vec.get_vc_timeout());
  }
};
```

VERILOG CODE

```
//
```

```
string vc_name.get_output_filename();
```

DESCRIPTION :

Get the name of the output file.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

```
//
```

CSL CODE

```
csl_unit dut{
  csl_port stim_in(input),stim_v(input),exp_out(output),exp_v(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input);
    set_output_filename("vector");
  }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(input);
    set_output_filename(stim_vec.get_output_filename());
  }
};
```

VERILOG CODE

```
//
```

add_logic(inject_stalls);

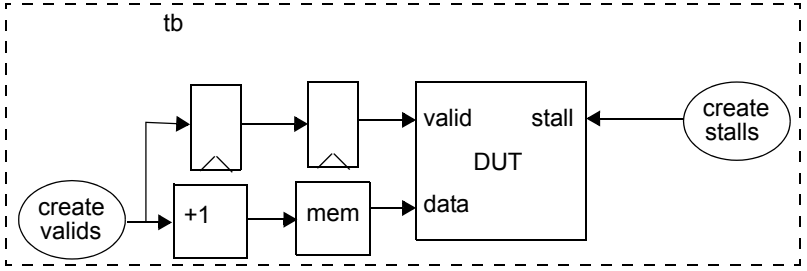
DESCRIPTION :

Inject stalls in the testbench. If the stall signal is set the valid signal is reseted and the data is invalid.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.14 Inject stalls



CSL CODE

```
csl_unit dut{
  csl_port stim_in(input), exp_out(output);
  dut(){
  }
};
csl_vector stim_vec{
  stim_vec(){
    set_module_name(dut);
    set_direction(input); }
};
csl_vector exp_vec{
  exp_vec(){
    set_module_name(dut);
    set_direction(output);
    add_logic(inject_stalls);}
};
csl_testbench tb{
  csl_signal clk;
  dut dut;
  stim_vec stim_vec;
  exp_vec exp_vec;
  tb(){
  add_logic(clock,clk,10,ns);
}
 };
```

VERILOG CODE
//

1.2.1 CSL state data

Architectural state comparisons can be specified at every clock cycle or based on an event output.

csl_state_data *state_data_object_name;*

DESCRIPTION :

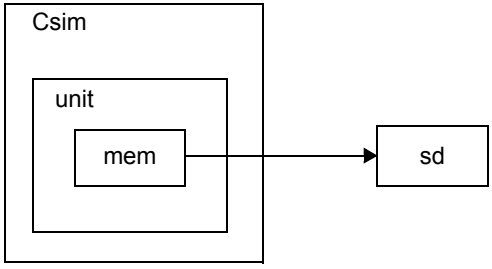
Create a csl_state_data object called *state_data_object_name.*

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

Csim is used to create an state_data.

FIGURE 1.15 state data creation



CSL CODE

```
//
```

VERILOG CODE

```
//
reg [WIDTH-1:0] sd[0:DEPTH-1];
initial begin
//use PLI to write the state data
    sd=$Csim_as_gen;
end
```

```
set_mem_instance_name(memory_instance_name);
```

DESCRIPTION :

Associate to a state data a memory instance. Command is mandatory if the associated unit is of other type than csl_register, csl_register_file or csl_fifo.
csl_fifo

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_register_file reg_sd{
  reg_sd(){
    set_width(8);
    set_depth(256);
}
};
csl_state_data SD{
  SD(){
 set_mem_instance_name(reg_sd);
  }
};
csl_testbench tb{
  csl_signal clk_sgn;
 SD SD;
 tb(){
 add_logic ( clk, clk_sgn, 2 ,ns );
  }
};
```

VERILOG CODE

```
//
```

set_unit_name(unit_name);

DESCRIPTION :

Associates the vector class to a CSL unit. This command is mandatory since it is this associated unit that will determine the vector's structure. This way the vector has "access" to the ports inside the associated unit.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit DUT{
   csl_port stim_in_a0 ( input );
   csl_port stim_in_a1 ( input );
   csl_port exp_out_a0 ( output );
   csl_port exp_out_a1 ( output );
   csl_port clk_p(input);
   DUT(){
  clk_p.set_attr(clock);
   }
};
csl_vector stim_vec{
   stim_vec(){
      set_unit_name ( DUT);
      set_direction   ( input );
   }
};
csl_vector exp_vec{
   exp_vec_a(){
      set_unit_name ( DUT );
      set_direction   ( output );
   }
};
csl_testbench tb{
csl_signal clk_s;
DUT dut;
     tb{
clk_s.set_attr(clock);
add_logic(clock,clk_s,10,ns);
}
};
```

VERILOG CODE

```
//
```

set_direction(direction);

DESCRIPTION :

This mandatory command sets the type of the vector to either stimulus or expected according to the direction parameter; thus, if the direction is set to input, the vector will be of type stimulus and if the direction is set as output, the type will be expected. It is used together with set_unit_name(unit_name); command which associates the vector with a CSL unit. Once the vector is associated with a unit, the type (direction) determines the kind (direction) of ports that the vector will access.

TABLE 1.6 Vector type according to direction parameter

vector direction	vector type
input	stimulus
output	expected

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit DUT{
   csl_port stim_in_a0 ( input );
   csl_port stim_in_a1 ( input );
   csl_port exp_out_a0 ( output );
   csl_port exp_out_a1 ( output );
   csl_port clk_p(input);
   DUT(){
   clk_p.set_attr(clock);
}
};
csl_vector stim_vec{
   stim_vec(){
      set_unit_name ( DUT);
      set_direction   ( input );
   }
};
csl_vector exp_vec{
   exp_vec_a(){
      set_unit_name ( DUT );
      set_direction   ( output );
   }
};
csl_testbench tb{
```

```
csl_signal clk_s;
DUT dut;
     tb{
clk_s.set_attr(clock);
add_logic(clock,clk_s,10,ns);
}
};
```

VERILOG CODE

```
//
```

exclusion_list(connection_elements);

DESCRIPTION :

This optional command is used if there are ports in the unit and that have the direction specified by the vector but are not needed (eg. for a stimulus vector the user doesn't want to include the clock and reset input ports) this command can be used to selects the ports that will not be connected by the vector.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit DUT_a{
   csl_port stim_in_a0 ( input );
   csl_port stim_in_a1 ( input );
   csl_port exp_out_a0 ( output );
   csl_port exp_out_a1 ( output );
   csl_port clk_p(input);
   DUT_a(){
   clk_p.set_attr(clock);
}
};
csl_vector stim_vec_a{
   stim_vec_a(){
      set_unit_name ( DUT_a );
      set_direction   ( input );
      exclusion_list  ( stim_in_a1 );
   }
};
csl_vector exp_vec_a{
   exp_vec_a(){
      set_unit_name ( DUT_a );
      set_direction   ( output );
   }
};
csl_testbench tb{
csl_signal clk_s;
DUT dut;
     tb{
clk_s.set_attr(clock);
add_logic(clock,clk_s,10,ns);
}
};
```

VERILOG CODE

```
//
```

include_only(connection_elements);

DESCRIPTION :

Has the same functionality as exclusion_list(connection_elements); command, except the ports specified with this command will be connected by the vector while those not specified will not. It is usefull when the associated unit containts many ports that have the direction of the vector, but only a few of these are needed.

[***CSL Verification Components Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_unit DUT_a{
   csl_port stim_in_a0 ( input );
   csl_port stim_in_a1 ( input );
   csl_port exp_out_a0 ( output );
   csl_port exp_out_a1 ( output );
   csl_port clk_p(input);
   DUT_a(){
  clk_p.set_attr(clock);
}
};
csl_vector stim_vec_a{
   stim_vec_a(){
      set_unit_name ( DUT_a );
      set_direction   ( input );
      exclusion_list  ( stim_in_a1 );
   }
};
csl_vector exp_vec_a{
   exp_vec_a(){
      set_unit_name ( DUT_a );
      set_direction   ( output );
      include_only    ( exp_out_a0 );
   }
};
csl_testbench tb{
csl_signal clk_s;
DUT dut;
     tb{
clk_s.set_attr(clock);
add_logic(clock,clk_s,10,ns);
}
```

```
};
```

VERILOG CODE

```
//
```

1.1 List of Tables