csl_signal *signal_object_name*;

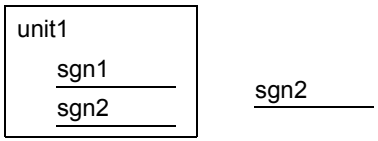
DESCRIPTION :

A signal named *signal_object_name* is produced. The signal can be created inside or outside a CSL unit's scope. If a signal object is created inside a scope, then it is added to the scope where the declaration occurs.Signals defined in a global scope can be reused in other scopes. Signals cause wires or other nets to be generated. The default type for a signal is wire and the default width is 1.

EXAMPLE :

In this example a simple CSL unit with two signals (default to wire) inside and outside its scope is shown.

FIGURE 1.1 Signals inside and outside unit



CSL CODE

```
//create a unit
csl_unit unit1;
//create signal outside the unit box, called sgn2
csl_signal sgn2;
scope unit1 {
  //create a signal inside the unit unit1, called sgn1
  //the signal type is wire
  csl_signal sgn1;
  //create a signal inside the unit unit1, called sgn2
  //the signal type is wire
  csl_signal sgn2;
}
```

VERILOG CODE

```
//create signals only inside unit unit1
module unit1();
  wire sgn1;
  wire sgn2;
endmodule
```

csl_signal *signal_object_name*(*[signal_data_type][,width]*);

DESCRIPTION :

It creates a signal named *signal_object_name* with width *width*. Optionally, the *signal_data_type* parameter can be specified, setting the type of the signal to reg or wire, tri etc. The parameters are optional and they specify a signal dimension (bitrange) or a signal with a multidimensional range. If no parameters are passed, the instruction will default *signal_object_name* to a 1 bit wire within the current scope (eg. *unit_object_name*).

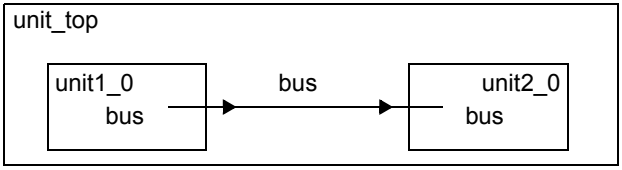
TABLE 1.1 Signal Types

signal_data_type	Description
reg	data type that can store a value
wire	data type that cannot store a value, but connects two points
tri	models a net that has multiple drivers
tri1	models resistive pullup device
tri0	models resistive pulldown device
triand	same functionality as wand
trior	same functionality as wor
wand	performs an and operation on multiple drivers logic
wor	performs an or operation on multiple drivers logic
supply0	models ground
supply1	models a power supply
trireg	stores a value and is used to model charge storage nodes

EXAMPLE :

In this example a signal object and two instances of other units are declared inside the scope of a unit.Then we connect the signal to the instances

FIGURE 1.2 Verilog connections for the example



CSL CODE

```
csl_unit unit1,unit2,unit_top;
scope unit_top {
//create a single dimension 32 bit wide wire
  csl_signal bus(32);
}
//add the signal bus1 to the unit box1 as output port
unit1.add_port(output, unit_top.bus);
//note: it is not the same with the signal from unit box1
unit2.add_port(input, unit_top.bus);
scope unit_top {
  add_instance(unit1, unit1_0(.bus(bus)));
  add_instance(unit2, unit2_0(.bus(bus)));
}
```

VERILOG CODE

```
module unit_top();
  wire [31:0] bus;
  unit1 unit1_0(.bus(bus));
  unit2 unit2_0(.bus(bus));
endmodule

module unit1(bus);
  //default type for a signal is wire
  output [31:0] bus;
endmodule

module unit2(bus);
  //default type for a signal is wire
  input [31:0] bus;
endmodule
```

csl_signal
signal_object_name(*[signal_data_type,]upper_limit,lower_limit*);

DESCRIPTION :

A signal named *signal_object_name is produced.* 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 bit range associated with the signal. Optionally, the *signal_data_type* parameter can be specified, setting the type of the signal to *signal_data_type*(see Signal Types) .

EXAMPLE :

In this example a signal object and two instances of other units are declared inside the scope of a unit.Then we connect the signal to the instances

CSL CODE

```
csl_unit unit1,unit2,unit_top;
scope unit_top {
//create a single dimension 32 bit wide wire
  csl_signal bus(31,0);
}
//add the signal bus to the unit unit1 as output port
unit1.add_port(output, unit_top.bus);
//note: it is not the same with the signal from unit1
unit2.add_port(input, unit_top.bus);
scope unit_top {
  add_instance(unit1, unit1_0(.bus(bus)));
  add_instance(unit2, unit2_0(.bus(bus)));
}
```

VERILOG CODE

```
module unit_top();
  wire [31:0] bus;
  unit1 unit1_0(.bus(bus));
  unit2 unit2_0(.bus(bus));
endmodule

module unit1(bus);
  //default type for a signal is wire
  output [31:0] bus;
endmodule

module unit2(bus);
  //default type for a signal is wire
  input [31:0] bus;
endmodule
```

csl_signal
signal_object_name(*[signal_data_type,]bitrange_object_name*);

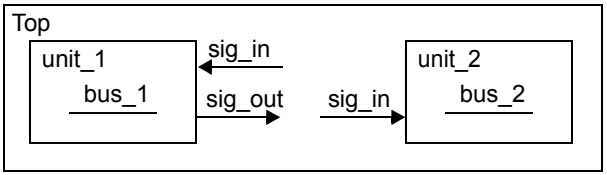
DESCRIPTION :

It creates a signal named *signal_object_name* with bitrange *bitrange_object_name*. Every signal object has a bitrange object attached by default. The default bitrange width for a new created signal is one. 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).The format of an anonymuos bitrange is: *[upper_index : lower_index] or [lower_index : upper_index]* where the words in italic represent integers or integer part of a numeric expression. Optionally, the *signal_data_type* parameter can be specified by setting the type of the signal to *signal_data_type*(see Signal Types).

EXAMPLE :

In this example a bitrange object is passed as a parameter to three signal objects' constructors that will then, be used selectively in three different units either as signals or as ports.

FIGURE 1.3



CSL CODE

VERILOG CODE

```
//AB
`define BR 7:0

module top();
  wire [`BR] bus1;
  a a0(.sig_in(),.sig_out(bus1));
  b b0(.sig_in(bus1));
endmodule

module a(sig_in,sig_out);
  input [`BR] sig_in;
  output [`BR] sig_out;
  reg [`BR] sig_out;
  wire [`BR] bus1;
endmodule

module b(sig_in);
```

```
  input [`BR] sig_in;
  wire [`BR] bus_1;
endmodule
```

signal_object_name.**set_range**(*upper_limit, lower_limit*);

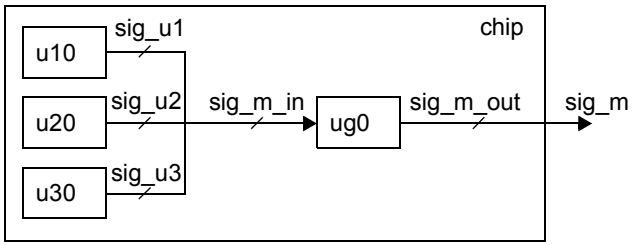
DESCRIPTION :

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

EXAMPLE :

In this example signals are assigned custom bit index values so that a wider signal could be connected to a number of concatenated signals.

FIGURE 1.4 Signals concatenation



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
scope chip {
  //creating 4 uninitialized signal objects
  csl_signal sig_u1,sig_u2,sig_u3,sig_m;
  //setting the range for each signal
  sig_u1.set_range(11,6);
  sig_u2.set_range(5,2);
  sig_u3.set_range(1,0);
  sig_m.set_range(11,0);
  /* creating an additional signal and "linking"
  its properties to sig_m */
  sig_m_in.set_bitrange(sig_m.get_bitrange());
  add_port(output,sig_m);
}
ug.add_port(input,chip.sig_m_in);
ug.add_port(output,chip.sig_m);
u1.add_port(output,chip.sig_u1);
u2.add_port(output,chip.sig_u2);
u3.add_port(output,chip.sig_u3);
scope chip {
  add_instance(ug, ug0);
  add_instance(u1, u10);
  add_instance(u2, u20);
```

```
    add_instance(u3, u30);
    //making the connections
    ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
    ug0.sig_m.connect(sig_m);
  }
```

VERILOG CODE

```
  //AB
  module chip(sig_m);
    output [11:0] sig_m;
    wire [11:6] sig_u1;
    wire [5:2] sig_u2;
    wire [1:0] sig_u3;
    wire [11:0] sig_m_in;
      u1 u10(.sig_u1(sig_u1));
      u2 u20(.sig_u2(sig_u2));
      u3 u30(.sig_u3(sig_u3));
      ug ug0(.sig_m_in(sig_m_in),.sig_m(sig_m));
    assign sig_m_in = {sig_u1,sig_u2,sig_u3};
  endmodule

  module u1(sig_u1);
    output [11:6] sig_u1;
  endmodule

  module u2(sig_u2);
    output [5:2] sig_u2;
  endmodule

  module u3(sig_u3);
    output [1:0] sig_u3;
  endmodule

  module ug(sig_m_in,sig_m);
    input [11:0] sig_m_in;
    output [11:0] sig_m;
  endmodule
```

signal_object_name.**set_lower_index**(*numeric_expression*);

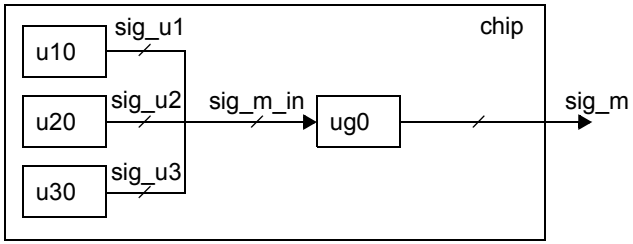
DESCRIPTION :

Set the lower index value for the single dimension signal *signal_object_name*.

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 1.5 Signals concatenation



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
scope chip {
  //creating 4 uninitialized signal objects
  csl_signal sig_u1,sig_u2,sig_u3,sig_m;
  sig_u2.set_range(5,2);
  //setting the range for each signal by specifying each index
  sig_u1.set_lower_index(6);
  sig_u1.set_upper_index(11);
  sig_u3.set_lower_index(sig_m.get_lower_index());
  sig_u3.set_upper_index(sig_u2.get_lower_index()-1);
  sig_m.set_range(11,0);
  /* creating an additional signal and "linking"
  its properties to sig_m */
  sig_m_in.set_lower_index(sig_m.get_lower_index());
  sig_m_in.set_upper_index(sig_m.get_upper_index());
  add_port(output,sig_m);
}
ug.add_port(input,chip.sig_m_in);
ug.add_port(output,chip.sig_m);
u1.add_port(output,chip.sig_u1);
u2.add_port(output,chip.sig_u2);
u3.add_port(output,chip.sig_u3);
scope chip {
  add_instance(ug, ug0);
```

```
    add_instance(u1, u10);
    add_instance(u2, u20);
    add_instance(u3, u30);
    //making the connections
    ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
    ug0.sig_m.connect(sig_m);
  }
```

VERILOG CODE

```
  //AB
  module chip(sig_m);
    output [11:0] sig_m;
    wire [11:6] sig_u1;
    wire [5:2] sig_u2;
    wire [1:0] sig_u3;
    wire [11:0] sig_m_in;
      u1 u10(.sig_u1(sig_u1));
      u2 u20(.sig_u2(sig_u2));
      u3 u30(.sig_u3(sig_u3));
      ug ug0(.sig_m_in(sig_m_in),.sig_m(sig_m));
    assign sig_m_in = {sig_u1,sig_u2,sig_u3};
  endmodule

  module u1(sig_u1);
    output [11:6] sig_u1;
  endmodule

  module u2(sig_u2);
    output [5:2] sig_u2;
  endmodule

  module u3(sig_u3);
    output [1:0] sig_u3;
  endmodule

  module ug(sig_m_in,sig_m);
    input [11:0] sig_m_in;
    output [11:0] sig_m;
  endmodule
```

int *signal_object_name*.**get_lower_index**();

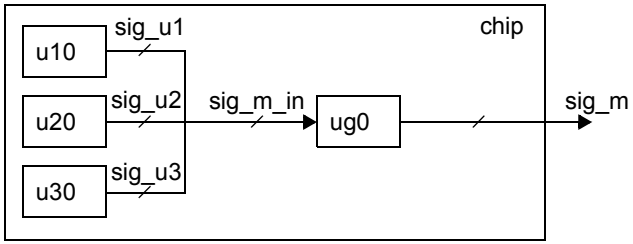
DESCRIPTION :

Returns the lower index value for the signal

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 1.6 Signals concatenation



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
scope chip {
  //create 4 uninitialized signal objects
  csl_signal sig_u1,sig_u2,sig_u3,sig_m;
  sig_u2.set_range(5,2);
  //set the range for each signal by specifying each index
  sig_u1.set_lower_index(6);
  sig_u1.set_upper_index(11);
  //using get return values to set indexes
  sig_u3.set_lower_index(sig_m.get_lower_index());
  sig_u3.set_upper_index(sig_u2.get_lower_index()-1);
  sig_m.set_range(11,0);
  /* create an additional signal and "linking"
  its properties to sig_m */
  sig_m_in.set_lower_index(sig_m.get_lower_index());
  sig_m_in.set_upper_index(sig_m.get_upper_index());
  add_port(output,sig_m);
}
ug.add_port(input,chip.sig_m_in);
ug.add_port(output,chip.sig_m);
u1.add_port(output,chip.sig_u1);
u2.add_port(output,chip.sig_u2);
u3.add_port(output,chip.sig_u3);
scope chip {
```

```
    add_instance(ug, ug0);
    add_instance(u1, u10);
    add_instance(u2, u20);
    add_instance(u3, u30);
    //making the connections
    ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
    ug0.sig_m.connect(sig_m);
  }
```

VERILOG CODE

```
  //AB
  module chip(sig_m);
    output [11:0] sig_m;
    wire [11:6] sig_u1;
    wire [5:2] sig_u2;
    wire [1:0] sig_u3;
    wire [11:0] sig_m_in;
      u1 u10(.sig_u1(sig_u1));
      u2 u20(.sig_u2(sig_u2));
      u3 u30(.sig_u3(sig_u3));
      ug ug0(.sig_m_in(sig_m_in),.sig_m(sig_m));
    assign sig_m_in = {sig_u1,sig_u2,sig_u3};
  endmodule

  module u1(sig_u1);
    output [11:6] sig_u1;
  endmodule

  module u2(sig_u2);
    output [5:2] sig_u2;
  endmodule

  module u3(sig_u3);
    output [1:0] sig_u3;
  endmodule

  module ug(sig_m_in,sig_m);
    input [11:0] sig_m_in;
    output [11:0] sig_m;
  endmodule
```

signal_object_name.**set_upper_index**(*numeric_expression*);

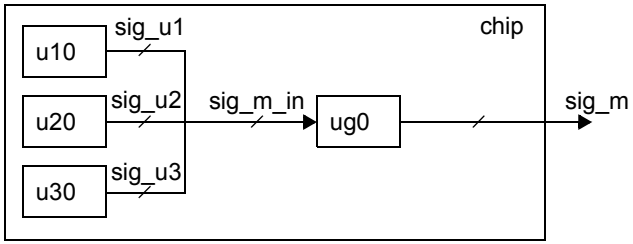
DESCRIPTION :

Set the upper index value for the signal.

EXAMPLE :

In this example, the upper index of a signal can be set explicitly or it can be the result of a get method applied on another object.

FIGURE 1.7 Signals concatenation



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
scope chip {
  //create 4 uninitialized signal objects
  csl_signal sig_u1,sig_u2,sig_u3,sig_m;
  sig_u2.set_range(5,2);
  //set the range for each signal by specifying each index
  sig_u1.set_lower_index(6);
  //set the upper index value
  sig_u1.set_upper_index(11);
  //using get return values to set indexes
  sig_u3.set_lower_index(sig_m.get_lower_index());
  sig_u3.set_upper_index(sig_u2.get_lower_index()-1);
  sig_m.set_range(11,0);
  /* create an additional signal and "linking"
  its properties to sig_m */
  sig_m_in.set_lower_index(sig_m.get_lower_index());
  sig_m_in.set_upper_index(sig_m.get_upper_index());
  add_port(output,sig_m);
}
ug.add_port(input,chip.sig_m_in);
ug.add_port(output,chip.sig_m);
u1.add_port(output,chip.sig_u1);
u2.add_port(output,chip.sig_u2);
u3.add_port(output,chip.sig_u3);
```

```
scope chip {
  add_instance(ug, ug0);
  add_instance(u1, u10);
  add_instance(u2, u20);
  add_instance(u3, u30);
  //making the connections
  ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
  ug0.sig_m.connect(sig_m);
}
```

VERILOG CODE

```
//AB
module chip(sig_m);
  output [11:0] sig_m;
  wire [11:6] sig_u1;
  wire [5:2] sig_u2;
  wire [1:0] sig_u3;
  wire [11:0] sig_m_in;
    u1 u10(.sig_u1(sig_u1));
    u2 u20(.sig_u2(sig_u2));
    u3 u30(.sig_u3(sig_u3));
    ug ug0(.sig_m_in(sig_m_in),.sig_m(sig_m));
  assign sig_m_in = {sig_u1,sig_u2,sig_u3};
endmodule

module u1(sig_u1);
  output [11:6] sig_u1;
endmodule

module u2(sig_u2);
  output [5:2] sig_u2;
endmodule

module u3(sig_u3);
  output [1:0] sig_u3;
endmodule

module ug(sig_m_in,sig_m);
  input [11:0] sig_m_in;
  output [11:0] sig_m;
endmodule
```

```
int signal_object_name.get_upper_index();
```

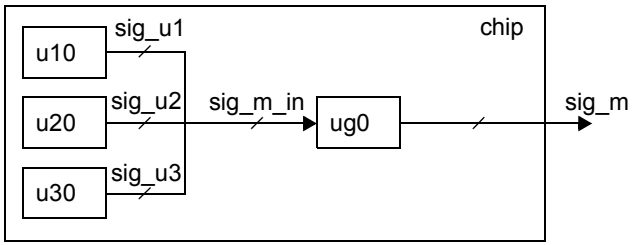
DESCRIPTION :

Return the upper index value for the signal

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 1.8 Signals concatenation



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
scope chip {
  //create 4 uninitialized signal objects
  csl_signal sig_u1,sig_u2,sig_u3,sig_m;
  sig_u2.set_range(5,2);
  sig_u1.set_lower_index(6);
  sig_u1.set_upper_index(11);
  sig_u3.set_lower_index(sig_m.get_lower_index());
  sig_u3.set_upper_index(sig_u2.get_lower_index()-1);
  sig_m.set_range(11,0);
  //using get return values to set indexes
  sig_m_in.set_lower_index(sig_m.get_lower_index());
  sig_m_in.set_upper_index(sig_m.get_upper_index());
  add_port(output,sig_m);
}
ug.add_port(input,chip.sig_m_in);
ug.add_port(output,chip.sig_m);
u1.add_port(output,chip.sig_u1);
u2.add_port(output,chip.sig_u2);
u3.add_port(output,chip.sig_u3);
scope chip {
  add_instance(ug, ug0);
  add_instance(u1, u10);
```

```
    add_instance(u2, u20);
    add_instance(u3, u30);
    //making the connections
    ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
    ug0.sig_m.connect(sig_m);
  }
```

VERILOG CODE

```
  //AB
  module chip(sig_m);
    output [11:0] sig_m;
    wire [11:6] sig_u1;
    wire [5:2] sig_u2;
    wire [1:0] sig_u3;
    wire [11:0] sig_m_in;
      u1 u10(.sig_u1(sig_u1));
      u2 u20(.sig_u2(sig_u2));
      u3 u30(.sig_u3(sig_u3));
      ug ug0(.sig_m_in(sig_m_in),.sig_m(sig_m));
    assign sig_m_in = {sig_u1,sig_u2,sig_u3};
  endmodule

  module u1(sig_u1);
    output [11:6] sig_u1;
  endmodule

  module u2(sig_u2);
    output [5:2] sig_u2;
  endmodule

  module u3(sig_u3);
    output [1:0] sig_u3;
  endmodule

  module ug(sig_m_in,sig_m);
    input [11:0] sig_m_in;
    output [11:0] sig_m;
  endmodule
```

signal_object_name.**set_offset**(*numeric_expression*);

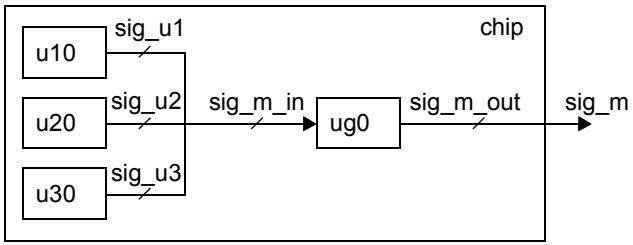
DESCRIPTION :

Set the offset value for signal *signal_object_name*.

EXAMPLE :

In this example

FIGURE 1.9 offset



CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
//creating 4 uninitialized signal objects
csl_signal sig_u1,sig_u2,sig_u3,sig_m;
sig_u1.set_range(11,6);
sig_u2.set_range(5,2);
sig_u3.set_range(1,0);
sig_m.set_range(11,0);
/* creating 2 additional signals and "linking"
their properties to sig_m */
sig_m_in.set_bitrange(sig_m.get_bitrange());
sig_m_out.set_bitrange(sig_m.get_bitrange());
chip.add_port(output,sig_m);
//setting the offset for sig_m; bitrange becomes [15:4]
chip.sig_m.set_offset(4);

ug.add_port(input,sig_m_in);
//setting the offsets for sig_m_in and sig_m_out of ug unit
ug.sig_m_in.set_offset(chip.sig_m.get_offset());
ug.add_port(output,sig_m_out);
ug.sig_m_out.set_offset(chip.sig_m.get_offset());
u1.add_port(output,sig_u1);
//setting a specific offset for signals sig_u1, sig_u2 and sig_u3
sig_u1.set_offset(chip.sig_m.get_offset()+4);
u2.add_port(output,sig_u2);
```

```
sig_u2.set_offset(sig_u1.get_offset());
u3.add_port(output,sig_u3);
sig_u3.set_offset(sig_u1.get_offset());

chip.add_instance(ug, ug0);
chip.add_instance(u1, u10);
chip.add_instance(u2, u20);
chip.add_instance(u3, u30);
//making the connections
scope chip{
ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
/* the following line may be redundant if
the autorouter width inference is on */
ug0.sig_m_out.connect(sig_m);
}
```

VERILOG CODE

```
//AB
//the offset values are set
`define SIG_M_UPPER 11
`define SIG_M_LOWER 0
`define OFFSET_M 4
`define OFFSET_U 8
`define SIG_M_UPOF `SIG_M_UPPER+`OFFSET_M
`define SIG_M_LOWOF `SIG_M_LOWER+`OFFSET_M
`define SIG_U1_UPPER 11
`define SIG_U1_LOWER 6
`define SIG_U1_UPOF `SIG_U1_UPPER+`OFFSET_M
`define SIG_U1_LOWOF `SIG_U1_LOWER+`OFFSET_M
`define SIG_U2_UPPER 5
`define SIG_U2_LOWER 2
`define SIG_U1_UPOF `SIG_U1_UPPER+`OFFSET_M
`define SIG_U1_LOWOF `SIG_U1_LOWER+`OFFSET_M

module chip(sig_m);
  output [11+`OFFSET_M:0+`OFFSET_M] sig_m;
  wire [11+`OFFSET_M:0+`OFFSET_M] sig_connect;
    u1 u10(.sig_u1(sig_connect[11+`OFFSET_M:6+`OFFSET_M]));
    u2 u20(.sig_u2(sig_connect[5+`OFFSET_M:2+`OFFSET_M]));
    u3 u30(.sig_u3(sig_connect[1+`OFFSET_M:0+`OFFSET_M]));
    ug ug0(.sig_m_in(sig_connect),.sig_m_out(sig_m));
```

```
endmodule

module u1(sig_u1);
  output [11+`OFFSET_U:6+`OFFSET_U] sig_u1;
endmodule

module u2(sig_u2);
  output [5+`OFFSET_U:2+`OFFSET_U] sig_u2;
endmodule

module u3(sig_u3);
  output [1+`OFFSET_U:0+`OFFSET_U] sig_u3;
endmodule

module ug(sig_m_in,sig_m_out);
  input [11+`OFFSET_M:0+`OFFSET_M] sig_m_in;
  output [11+`OFFSET_M:0+`OFFSET_M] sig_m_out;
endmodule
```

constant_numeric_expression signal_object_name.**get_offset**();

DESCRIPTION :

Return the offset value for a signal

EXAMPLE :

small description of the example.

CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
//creating 4 uninitialized signal objects
csl_signal sig_u1,sig_u2,sig_u3,sig_m;
sig_u1.set_range(11,6);
sig_u2.set_range(5,2);
sig_u3.set_range(1,0);
sig_m.set_range(11,0);
/* creating 2 additional signals and "linking"
their properties to sig_m */
sig_m_in.set_bitrange(sig_m.get_bitrange());
sig_m_out.set_bitrange(sig_m.get_bitrange());
chip.add_port(output,sig_m);
//setting the offset for sig_m; bitrange becomes [15:4]
chip.sig_m.set_offset(4);

ug.add_port(input,sig_m_in);
//using get() method along with a set method
ug.sig_m_in.set_offset(chip.sig_m.get_offset());
ug.add_port(output,sig_m_out);
ug.sig_m_out.set_offset(chip.sig_m.get_offset());
u1.add_port(output,sig_u1);
//setting a specific offset using get() method
sig_u1.set_offset(chip.sig_m.get_offset()+4);
u2.add_port(output,sig_u2);
sig_u2.set_offset(sig_u1.get_offset());
u3.add_port(output,sig_u3);
sig_u3.set_offset(sig_u1.get_offset());

chip.add_instance(ug, ug0);
chip.add_instance(u1, u10);
chip.add_instance(u2, u20);
chip.add_instance(u3, u30);
//making the connections
```

```
scope chip{
ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
/* the following line may be redundant if
the autorouter width inference is on */
ug0.sig_m_out.connect(sig_m);
}
```

VERILOG CODE

```
//AB
`define OFFSET_M 4
`define OFFSET_U 8

module chip(sig_m);
  output [11+`OFFSET_M:0+`OFFSET_M] sig_m;
  wire [11+`OFFSET_M:0+`OFFSET_M] sig_connect;
    u1 u10(.sig_u1(sig_connect[11+`OFFSET_M:6+`OFFSET_M]));
    u2 u20(.sig_u2(sig_connect[5+`OFFSET_M:2+`OFFSET_M]));
    u3 u30(.sig_u3(sig_connect[1+`OFFSET_M:0+`OFFSET_M]));
    ug ug0(.sig_m_in(sig_connect),.sig_m_out(sig_m));
endmodule

module u1(sig_u1);
  output [11+`OFFSET_U:6+`OFFSET_U] sig_u1;
endmodule

module u2(sig_u2);
  output [5+`OFFSET_U:2+`OFFSET_U] sig_u2;
endmodule

module u3(sig_u3);
  output [1+`OFFSET_U:0+`OFFSET_U] sig_u3;
endmodule

module ug(sig_m_in,sig_m_out);
  input [11+`OFFSET_M:0+`OFFSET_M] sig_m_in;
  output [11+`OFFSET_M:0++`OFFSET_M] sig_m_out;
endmodule
```

csl_signal_group *signal_group_object_name(list_object_name);*

DESCRIPTION :

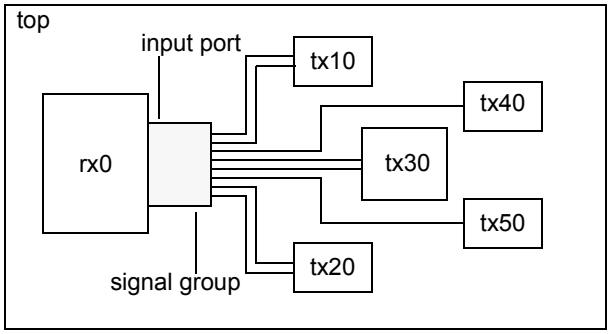
create csl signal group

associates an ordered list of signals with the *signal_group_object_name*. The list object can contain signals, signals with bit ranges, constants, other groups, *instance_object_names* or interfaces. If an *instance_object_name* is used as an argument to a *csl_signal_group* then the port signals in the instance are added to the list of signals which belong to the *signal_group_object_name*. Since groups are used to conect objects and can traverse multiple levels of hierarchies there may be cases where signals may need to be added or deleted from the group.

EXAMPLE :

Using a signal group to connect multiple signals from transmitting units to a single port in a receiving unit. The top_design unit where the component units will get instantiated is generated automatically.

FIGURE 1.10 Grouping signals



CSL CODE

```
csl_unit rx, tx1, tx2, tx3, tx4, tx5, top;
scope top {
  csl_signal s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0;
  csl_list sig_list1(s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0);
  //create the signal group with the previously declared signal list
  csl_signal_group sgn(sig_list1);
}
tx1.add_port_list(output,csl_list(top.s1_0,top.s1_1));
tx2.add_port_list(output,csl_list(top.s2_0,top.s2_1));
tx3.add_port_list(output,csl_list(top.s3_0,top.s3_1));
tx4.add_port(output,top.s4_0);
tx5.add_port(output,top.s5_0);
//create an input port sgn width wide
rx.add_port(input,sgn.get_width(),top.sgn);
scope top {
  add_instance(rx,rx0(.sgn(sgn));
  add_instance(tx1,tx10(.s1_0(s1_0),.s1_1(s1_1));
```

```
  add_instance(tx2,tx20(.s2_0(s2_0),.s2_1(s2_1));
  add_instance(tx3,tx30(.s3_0(s3_0),.s3_1(s3_1));
  add_instance(tx4,tx40(.s4_0(s4_0));
  add_instance(tx5,tx50(.s5_0(s5_0));
}
csl_unit rx, tx1, tx2, tx3, tx4, tx5;
csl_signal s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0;
//will rely on autorouter name inference for connection
tx1.add_port_list(output,csl_list(s1_0,s1_1));
tx2.add_port_list(output,csl_list(s2_0,s2_1));
tx3.add_port_list(output,csl_list(s3_0,s3_1));
tx4.add_port(output,s4_0);
tx5.add_port(output,s5_0);
csl_list sig_list1(s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0);
//creating the signal group with the previously declared signal list
csl_signal_group sgn(sig_list1);
rx.add_port(input,sgn.get_width(),sgn);
/*creates an input port sgn width wide and by using the same name
 will rely on autorouter to make the connection */
```

VERILOG CODE

```
`define WIDTH_s1_1 1
`define WIDTH_s2_0 1
`define WIDTH_s2_1 1
`define WIDTH_s3_0 1
`define WIDTH_s3_1 1
`define WIDTH_s4_0 1
`define WIDTH_s5_0 1

module top();
  wire [`WIDTH_s1_0-1:0] s1_0;
  wire [`WIDTH_s1_1-1:0] s1_1;
  wire [`WIDTH_s2_0-1:0] s2_0;
  wire [`WIDTH_s2_1-1:0] s2_1;
  wire [`WIDTH_s3_0-1:0] s3_0;
  wire [`WIDTH_s3_1-1:0] s3_1;
  wire [`WIDTH_s4_0-1:0] s4_0;
  wire [`WIDTH_s5_0-1:0] s5_0;
wire
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
```

```
  rx rx0(.sgn(sgn));
  tx1 tx10(.s1_0(s1_0),.s1_1(s1_1));
  tx2 tx20(.s2_0(s2_0),.s2_1(s2_1));
  tx3 tx30(.s3_0(s3_0),.s3_1(s3_1));
  tx4 tx40(.s4_0(s4_0));
  tx5 tx50(.s5_0(s5_0));

  assign sgn = {s1_0,s1_1,s2_0,s2_1,s3_0,s3_1,s4_0,s5_0};

endmodule

module rx(sgn);
  input
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
endmodule

module tx1(s1_0,s1_1);
  output [`WIDTH_s1_0-1:0] s1_0;
  output [`WIDTH_s1_1-1:0] s1_1;
endmodule

module tx2(s2_0,s2_1);
  output [`WIDTH_s2_0-1:0] s2_0;
  output [`WIDTH_s2_1-1:0] s2_1;
endmodule

module tx3(s3_0,s3_1);
  output [`WIDTH_s3_0-1:0] s3_0;
  output [`WIDTH_s3_1-1:0] s3_1;
endmodule

module tx4(s4_0);
  output [`WIDTH_s4_0-1:0] s4_0;
endmodule

module tx5(s5_0);
  output [`WIDTH_s5_0-1:0] s5_0;
endmodule
```

csl_signal_group
signal_group_object_name(signal_group_object_name);

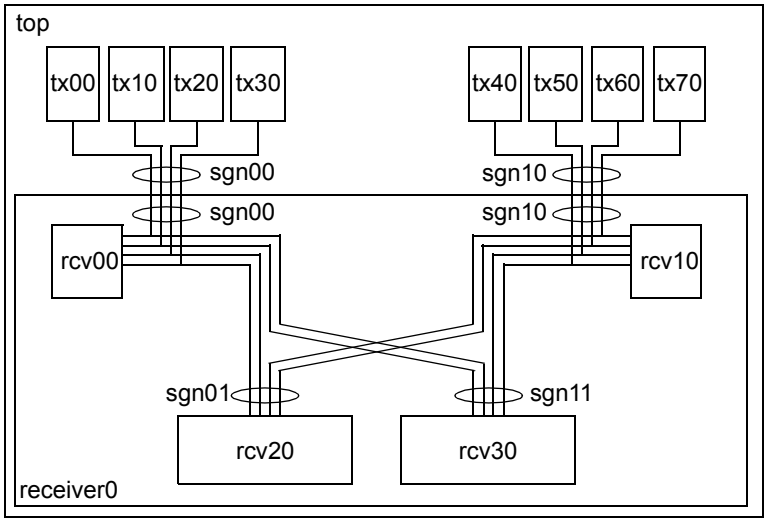
DESCRIPTION :

create csl signal group by using another signal group object as parameter. This behaves like a copy constructor. The groups can then be connected.

EXAMPLE :

In this example the copy constructor is used to create new signal groups from other signal groups and then to operate changes on these new groups.

FIGURE 1.11 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
```

```
}
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
  add_instance(tx1,tx10);
  add_instance(tx2,tx20);
```

```
  add_instance(tx3,tx30);
    add_instance(tx4,tx40);
    add_instance(tx5,tx50);
    add_instance(tx6,tx60);
    add_instance(tx7,tx70);
  }
```

VERILOG CODE

```
  //AB
  `define WIDTH_S0 1
  `define WIDTH_S1 1
  `define WIDTH_S2 1
  `define WIDTH_S3 1
  `define WIDTH_S4 1
  `define WIDTH_S5 1
  `define WIDTH_S6 1
  `define WIDTH_S7 1

  module top();
    wire [`WIDTH_S0-1:0] s0;
    wire [`WIDTH_S1-1:0] s1;
    wire [`WIDTH_S2-1:0] s2;
    wire [`WIDTH_S3-1:0] s3;
    wire [`WIDTH_S4-1:0] s4;
    wire [`WIDTH_S5-1:0] s5;
    wire [`WIDTH_S6-1:0] s6;
    wire [`WIDTH_S7-1:0] s7;
    wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
    wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

    assign sgn00 = {s0,s1,s2,s3};
    assign sgn10 = {s4,s5,s6,s7};

    tx0 tx00(.s0(s0));
    tx1 tx10(.s1(s1));
    tx2 tx20(.s2(s2));
    tx3 tx30(.s3(s3));
    tx4 tx40(.s4(s4));
    tx5 tx50(.s5(s5));
    tx6 tx60(.s6(s6));
    tx7 tx70(.s7(s7));
```

```
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(sgn01);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
```

```
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

csl_signal_group *signal_group_object_name*(list_of_objects | signal_list_object);

DESCRIPTION :

This command createa a signal group,associating an ordered list of signals or a signal list object with *signal_group_object_name. list_of_signals* can contain signals, signals with bit ranges, constants, other groups, *instance_object_names* or interfaces.

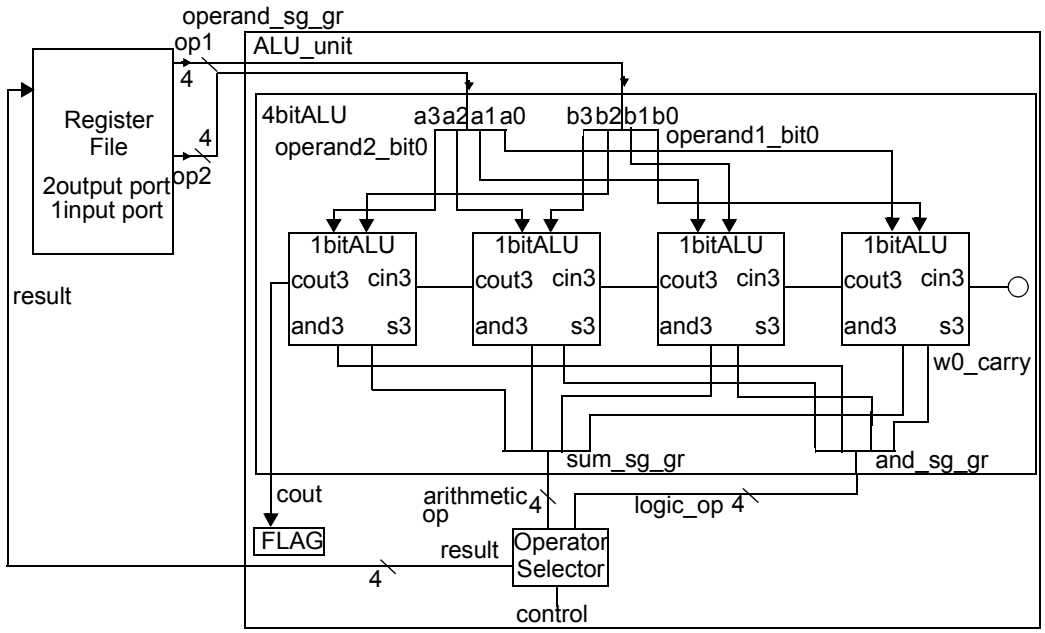
If an instance of an object is used as an argument to a *csl_signal_group* constructor then all the port signals in the instance are added to the list of signals which belongs to the that signal group.

Since groups are used to connect objects and can traverse multiple levels of hierarchies there may be cases where signals may to be added or deleted from the group.

EXAMPLE :

In this example we present a 4 bit ALU .This ALU performes arithmetic and logical operations.The operands and the result are located in the register file.There are also a module for selecting th operation and a register used to store the value of the last carry signal.

FIGURE 1.12



csl_signal_group *signal_group_object_name(list_of_signals);*

DESCRIPTION :

FIX DESCRIPTION

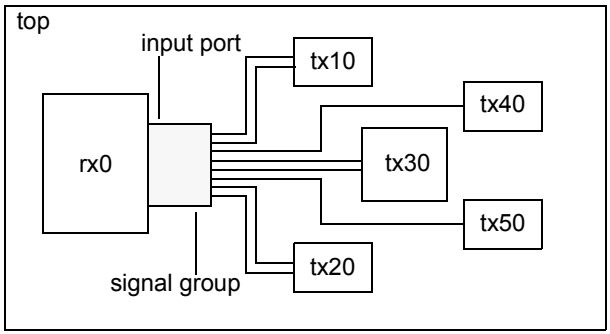
create csl signal group

associates an ordered list of signals with the *signal_group_object_name. list_of_signals* can contain signals, signals with bit ranges, constants, other groups, *instance_object_names* or interfaces. If an *instance_object_name* is used as an argument to a *csl_signal_group* then the port signals in the instance are added to the list of signals which belong to the *signal_group_object_name*. Since groups are used to conect objects and can traverse multiple levels of hierarchies there may be cases where signals may to be added or deleted from the group.

EXAMPLE :

Using a signal group to connect multiple signals from transmitting units to a single port in a receiving unit. The top_design unit where the component units will get instantiated is generated automatically.

FIGURE 1.13 Grouping signals



CSL CODE

```
csl_unit rx, tx1, tx2, tx3, tx4, tx5, top;
scope top {
  csl_signal s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0;
  //create the signal group with the previously declared signals
  csl_signal_group sgn(s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0);
}
tx1.add_port_list(output,csl_list(top.s1_0,top.s1_1));
tx2.add_port_list(output,csl_list(top.s2_0,top.s2_1));
tx3.add_port_list(output,csl_list(top.s3_0,top.s3_1));
tx4.add_port(output,top.s4_0);
tx5.add_port(output,top.s5_0);
//create an input port sgn width wide
rx.add_port(input,sgn.get_width(),top.sgn);
scope top {
  add_instance(rx,rx0(.sgn(sgn));
  add_instance(tx1,tx10(.s1_0(s1_0),.s1_1(s1_1));
```

```
  add_instance(tx2,tx20(.s2_0(s2_0),.s2_1(s2_1));
  add_instance(tx3,tx30(.s3_0(s3_0),.s3_1(s3_1));
  add_instance(tx4,tx40(.s4_0(s4_0));
  add_instance(tx5,tx50(.s5_0(s5_0));
}
```

VERILOG CODE

```
`define WIDTH_s1_1 1
`define WIDTH_s2_0 1
`define WIDTH_s2_1 1
`define WIDTH_s3_0 1
`define WIDTH_s3_1 1
`define WIDTH_s4_0 1
`define WIDTH_s5_0 1

module top();
  wire [`WIDTH_s1_0-1:0] s1_0;
  wire [`WIDTH_s1_1-1:0] s1_1;
  wire [`WIDTH_s2_0-1:0] s2_0;
  wire [`WIDTH_s2_1-1:0] s2_1;
  wire [`WIDTH_s3_0-1:0] s3_0;
  wire [`WIDTH_s3_1-1:0] s3_1;
  wire [`WIDTH_s4_0-1:0] s4_0;
  wire [`WIDTH_s5_0-1:0] s5_0;
  wire
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
  rx rx0(.sgn(sgn));
  tx1 tx10(.s1_0(s1_0),.s1_1(s1_1));
  tx2 tx20(.s2_0(s2_0),.s2_1(s2_1));
  tx3 tx30(.s3_0(s3_0),.s3_1(s3_1));
  tx4 tx40(.s4_0(s4_0));
  tx5 tx50(.s5_0(s5_0));

  assign sgn = {s1_0,s1_1,s2_0,s2_1,s3_0,s3_1,s4_0,s5_0};

endmodule

module rx(sgn);
  input
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
```

```
endmodule

module tx1(s1_0,s1_1);
  output [`WIDTH_s1_0-1:0] s1_0;
  output [`WIDTH_s1_1-1:0] s1_1;
endmodule

module tx2(s2_0,s2_1);
  output [`WIDTH_s2_0-1:0] s2_0;
  output [`WIDTH_s2_1-1:0] s2_1;
endmodule

module tx3(s3_0,s3_1);
  output [`WIDTH_s3_0-1:0] s3_0;
  output [`WIDTH_s3_1-1:0] s3_1;
endmodule

module tx4(s4_0);
  output [`WIDTH_s4_0-1:0] s4_0;
endmodule

module tx5(s5_0);
  output [`WIDTH_s5_0-1:0] s5_0;
endmodule
```

signal_group_object_name.**generate_individual_rtl_signals(***status***);**

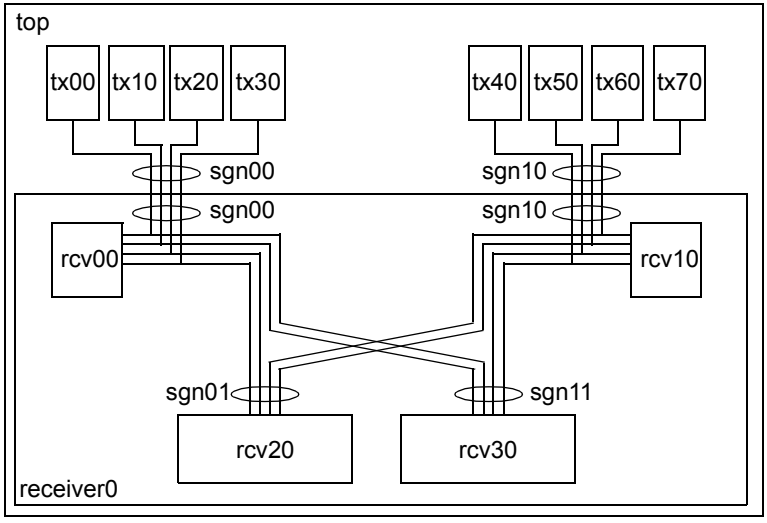
DESCRIPTION :

When this is set as the group “traverses” scopes the autorouter will generate a port for each signal inside the group. If it is set to false (default), 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. *status* **off** stands for false, while **on** stands for true. When a signal group is constructed by copying another signal grup, the generate_individual_rtl_signals property is inherited, and should be overriden if the designer whishes a different behaviour for that signal group.

EXAMPLE :

By setting the generate_individual_rtl_signals directive *status* on, the signal groups will generate individual ports for each signal as they “traverse” unit’s scopes. This is visible in the generated Verilog code, as the CSL code remains mostly the same.

FIGURE 1.14 Intersected signal groups example



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
```

```
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
//the following will generate a port for each signal in the group
sgn00.generate_individual_rtl_signals(on);
csl_signal_group sgn10(s4,s5,s6,s7);
//the following will generate a port for each signal in the group
sgn10.generate_individual_rtl_signals(on);
}
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /* since sgn10 is a copy of top.sgn10 it also preserved the
  attribute regarding individual rtl signals generation and if
  this is not desired it can be turned off like below */
  sgn10.generate_individual_rtl_signals(off);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  sgn11.generate_individual_rtl_signals(off);
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_interface(input,sgn00);
  add_interface(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
  /*the connect method will automatically create ports and
  prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
  /*no need to connect sgn00 with sgn01 and sgn10 with sgn11
  because these groups share the same scope and only specify
  different signal organization */
}
```

```
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
  add_instance(tx1,tx10);
  add_instance(tx2,tx20);
  add_instance(tx3,tx30);
  add_instance(tx4,tx40);
  add_instance(tx5,tx50);
  add_instance(tx6,tx60);
  add_instance(tx7,tx70);
}
```

VERILOG CODE

```
//AB
`define WIDTH_S0 1
`define WIDTH_S1 1
`define WIDTH_S2 1
`define WIDTH_S3 1
`define WIDTH_S4 1
`define WIDTH_S5 1
`define WIDTH_S6 1
`define WIDTH_S7 1

module top();
wire [`WIDTH_S0-1:0] s0;
wire [`WIDTH_S1-1:0] s1;
wire [`WIDTH_S2-1:0] s2;
wire [`WIDTH_S3-1:0] s3;
wire [`WIDTH_S4-1:0] s4;
wire [`WIDTH_S5-1:0] s5;
wire [`WIDTH_S6-1:0] s6;
wire [`WIDTH_S7-1:0] s7;
```

```
tx0 tx00(.s0(s0));
tx1 tx10(.s1(s1));
tx2 tx20(.s2(s2));
tx3 tx30(.s3(s3));
tx4 tx40(.s4(s4));
tx5 tx50(.s5(s5));
tx6 tx60(.s6(s6));
tx7 tx70(.s7(s7));
receiver
receiver0(.s0(s0),.s1(s1),.s2(s2),.s3(s3),.s4(s4),.s5(s5),.s6(s6),.s7(
s7));

endmodule

module receiver(s0,s1,s2,s3,s4,s5,s6,s7);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S2-1:0] s2;
input [`WIDTH_S3-1:0] s3;
input [`WIDTH_S0-1:0] s4;
input [`WIDTH_S1-1:0] s5;
input [`WIDTH_S2-1:0] s6;
input [`WIDTH_S3-1:0] s7;
wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

assign {s4,s5,s6,s7} = sgn10;
assign sgn11 = {s2,s3,s6,s7};

rcv0 rcv00(.s0(s0),.s1(s1),.s2(s2),.s3(s3));
rcv1 rcv10(.sgn10(sgn10));
rcv2 rcv20(.s0(s0),.s1(s1),.s4(s4),.s5(s5));
rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(s0,s1,s2,s3);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S2-1:0] s2;
```

```
input [`WIDTH_S3-1:0] s3;
endmodule

module rcv1(sgn10);
input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(s0,s1,s4,s5);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S4-1:0] s4;
input [`WIDTH_S5-1:0] s5;
endmodule

module rcv3(sgn11);
input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
output [`WIDTH_S5-1:0] s5;
```

```
endmodule

module tx6(s6);
output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
output [`WIDTH_S7-1:0] s7;
endmodule
```

signal_group_object_name.**add_signal**(*signal_object_name*);

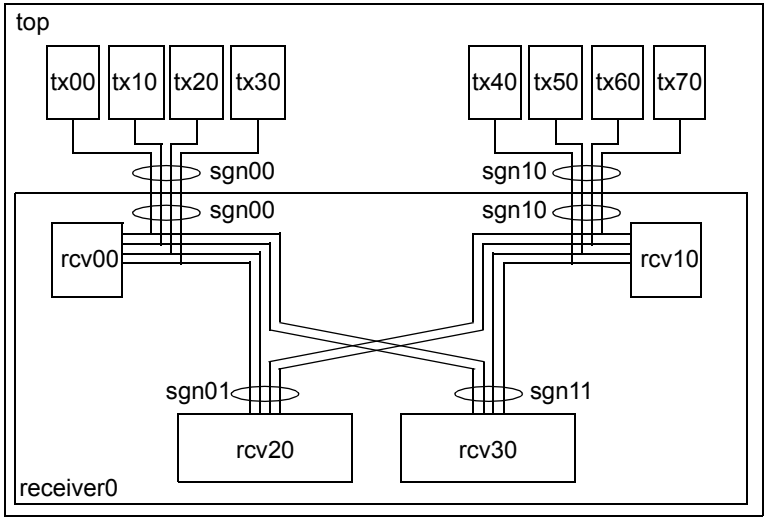
DESCRIPTION :

Add *signal_object_name* to the *signal_group_object_name*.

EXAMPLE :

The examples shows how a signal object can be added to an previously created signal group.

FIGURE 1.15 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
}
rcv0.add_interface(input,top.sgn00);
```

```
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  //signals are being added to signal groups one at a time
  sgn11.add_signal(sgn01.s2);
  sgn11.add_signal(sgn01.s3);
  sgn01.add_signal(sgn11.s4);
  sgn01.add_signal(sgn11.s5);
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
  add_instance(tx1,tx10);
```

```
    add_instance(tx2,tx20);
    add_instance(tx3,tx30);
    add_instance(tx4,tx40);
    add_instance(tx5,tx50);
    add_instance(tx6,tx60);
    add_instance(tx7,tx70);
  }
```

VERILOG CODE

```
  //AB
  `define WIDTH_S0 1
  `define WIDTH_S1 1
  `define WIDTH_S2 1
  `define WIDTH_S3 1
  `define WIDTH_S4 1
  `define WIDTH_S5 1
  `define WIDTH_S6 1
  `define WIDTH_S7 1

  module top();
    wire [`WIDTH_S0-1:0] s0;
    wire [`WIDTH_S1-1:0] s1;
    wire [`WIDTH_S2-1:0] s2;
    wire [`WIDTH_S3-1:0] s3;
    wire [`WIDTH_S4-1:0] s4;
    wire [`WIDTH_S5-1:0] s5;
    wire [`WIDTH_S6-1:0] s6;
    wire [`WIDTH_S7-1:0] s7;
    wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
    wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

    assign sgn00 = {s0,s1,s2,s3};
    assign sgn10 = {s4,s5,s6,s7};

    tx0 tx00(.s0(s0));
    tx1 tx10(.s1(s1));
    tx2 tx20(.s2(s2));
    tx3 tx30(.s3(s3));
    tx4 tx40(.s4(s4));
    tx5 tx50(.s5(s5));
    tx6 tx60(.s6(s6));
```

```
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(sgn01);
```

```
    input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  endmodule

  module rcv3(sgn11);
    input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
  endmodule

  module tx0(s0);
    output [`WIDTH_S0-1:0] s0;
  endmodule

  module tx1(s1);
    output [`WIDTH_S1-1:0] s1;
  endmodule

  module tx2(s2);
    output [`WIDTH_S2-1:0] s2;
  endmodule

  module tx3(s3);
    output [`WIDTH_S3-1:0] s3;
  endmodule

  module tx4(s4);
    output [`WIDTH_S4-1:0] s4;
  endmodule

  module tx5(s5);
    output [`WIDTH_S5-1:0] s5;
  endmodule

  module tx6(s6);
    output [`WIDTH_S6-1:0] s6;
  endmodule

  module tx7(s7);
    output [`WIDTH_S7-1:0] s7;
endmodule
```

signal_group_object_name.**add_signal_list**(*list_object*);

DESCRIPTION :

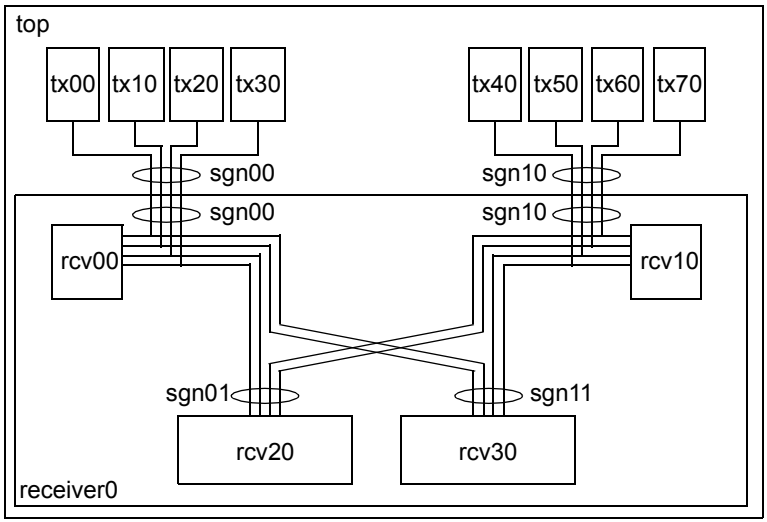
Add the list of signals *list_object* to the signal group *signal_group_object_name*. The list may be previously created or it may be an anonymous list. It must contain objects of type signal, otherwise a compiler error is generated.

!!add to warn/error

EXAMPLE :

This example shows how a list of signals can be added to a signal group.

FIGURE 1.16 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
```

```
}
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  //signals are being added to a signal group by using a list
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
  add_instance(tx1,tx10);
```

```
    add_instance(tx2,tx20);
    add_instance(tx3,tx30);
    add_instance(tx4,tx40);
    add_instance(tx5,tx50);
    add_instance(tx6,tx60);
    add_instance(tx7,tx70);
  }
```

VERILOG CODE

```
  //AB
  `define WIDTH_S0 1
  `define WIDTH_S1 1
  `define WIDTH_S2 1
  `define WIDTH_S3 1
  `define WIDTH_S4 1
  `define WIDTH_S5 1
  `define WIDTH_S6 1
  `define WIDTH_S7 1

  module top();
    wire [`WIDTH_S0-1:0] s0;
    wire [`WIDTH_S1-1:0] s1;
    wire [`WIDTH_S2-1:0] s2;
    wire [`WIDTH_S3-1:0] s3;
    wire [`WIDTH_S4-1:0] s4;
    wire [`WIDTH_S5-1:0] s5;
    wire [`WIDTH_S6-1:0] s6;
    wire [`WIDTH_S7-1:0] s7;
    wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
    wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

    assign sgn00 = {s0,s1,s2,s3};
    assign sgn10 = {s4,s5,s6,s7};

    tx0 tx00(.s0(s0));
    tx1 tx10(.s1(s1));
    tx2 tx20(.s2(s2));
    tx3 tx30(.s3(s3));
    tx4 tx40(.s4(s4));
    tx5 tx50(.s5(s5));
    tx6 tx60(.s6(s6));
```

```
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(sgn01);
```

```
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

signal_group_object_name.**remove_signal**(*signal_object_name*);

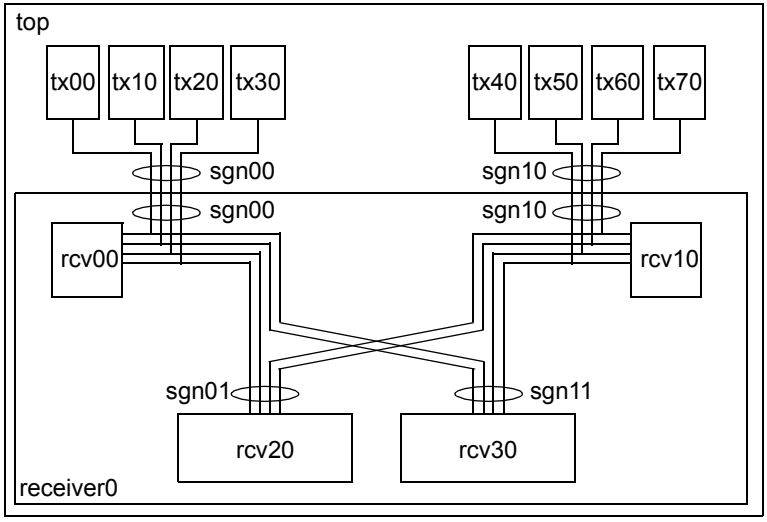
DESCRIPTION :

Removes *signal_object_name* from the group *signal_group_name.*

EXAMPLE :

In this example, signals are being removed from signal groups in order to restructure the group.

FIGURE 1.17 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
}
rcv0.add_interface(input,top.sgn00);
```

```
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  //signals are being added to a signal group by using a list
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  //remove signals from groups one at a time
  sgn11.remove_signal(s4);
  sgn11.remove_signal(s5);
  sgn01.remove_signal(s2);
  sgn01.remove_signal(s3);
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
```

```
  add_instance(tx1,tx10);
  add_instance(tx2,tx20);
  add_instance(tx3,tx30);
  add_instance(tx4,tx40);
  add_instance(tx5,tx50);
  add_instance(tx6,tx60);
  add_instance(tx7,tx70);
}
```

VERILOG CODE

```
//AB
`define WIDTH_S0 1
`define WIDTH_S1 1
`define WIDTH_S2 1
`define WIDTH_S3 1
`define WIDTH_S4 1
`define WIDTH_S5 1
`define WIDTH_S6 1
`define WIDTH_S7 1

module top();
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

  assign sgn00 = {s0,s1,s2,s3};
  assign sgn10 = {s4,s5,s6,s7};

  tx0 tx00(.s0(s0));
  tx1 tx10(.s1(s1));
  tx2 tx20(.s2(s2));
  tx3 tx30(.s3(s3));
  tx4 tx40(.s4(s4));
  tx5 tx50(.s5(s5));
```

```
  tx6 tx60(.s6(s6));
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule
```

```
module rcv2(sgn01);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

signal_group_object_name.**remove_signal_list**(*list_object*);

DESCRIPTION :

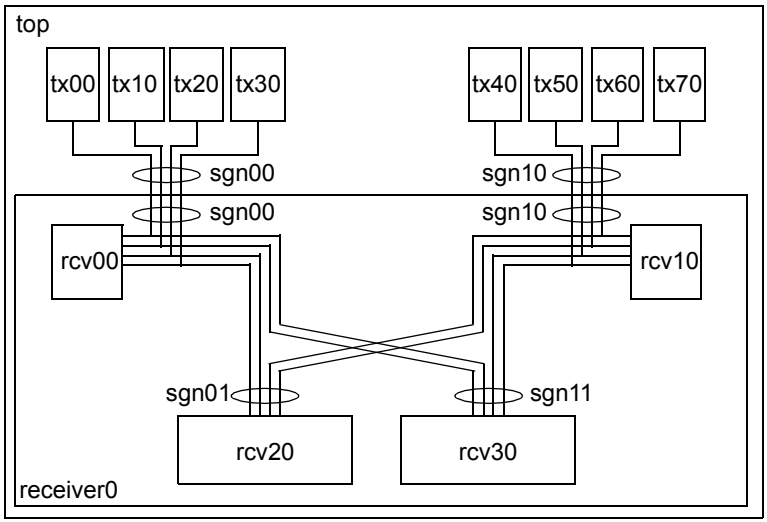
Remove list of signals *list_object* from signal group *signal_group_object*. The list may be previously created or it may be an anonymous list. It must contain objects of type signal, otherwise a compiler error is generated.

!!add to warn/error

EXAMPLE :

This example shows how multiple signals can be removed from a group by using lists of signals.

FIGURE 1.18 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
```

```
}
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
  //signals are being added to a signal group by using a list
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  //remove a list of signals from a signal group
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
```

```
  add_instance(tx1,tx10);
  add_instance(tx2,tx20);
  add_instance(tx3,tx30);
  add_instance(tx4,tx40);
  add_instance(tx5,tx50);
  add_instance(tx6,tx60);
  add_instance(tx7,tx70);
}
```

VERILOG CODE

```
//AB
`define WIDTH_S0 1
`define WIDTH_S1 1
`define WIDTH_S2 1
`define WIDTH_S3 1
`define WIDTH_S4 1
`define WIDTH_S5 1
`define WIDTH_S6 1
`define WIDTH_S7 1

module top();
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

  assign sgn00 = {s0,s1,s2,s3};
  assign sgn10 = {s4,s5,s6,s7};

  tx0 tx00(.s0(s0));
  tx1 tx10(.s1(s1));
  tx2 tx20(.s2(s2));
  tx3 tx30(.s3(s3));
  tx4 tx40(.s4(s4));
  tx5 tx50(.s5(s5));
```

```
  tx6 tx60(.s6(s6));
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule
```

```
module rcv2(sgn01);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

csl_unit *unit_object_name*;

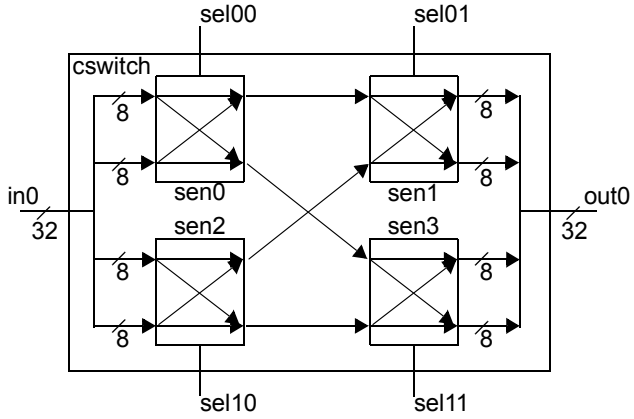
DESCRIPTION :

Creates a new unit with the declared name *unit_object_name*. The **csl_unit** is the equivalent of a module in Verilog: it can have a module interface, local signals, instances of other units and logic. It can also contain other modules (we will call them instances) that are connected one to another to implement the unit's behavior.

EXAMPLE :

In the following example

FIGURE 1.19 Clos Switch



CSL CODE

```
//AB
//create the two building blocks: cswitch and sen
csl_unit cswitch,sen;
scope cswitch {
  add_port(input,32,in0);
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  add_port(output,32,out0);
}
scope sen {
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
```

```
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
}
```

SEE ALSO :

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

scope *unit_object_name* { *csl statement+* }

DESCRIPTION :

Description needs to be fixed

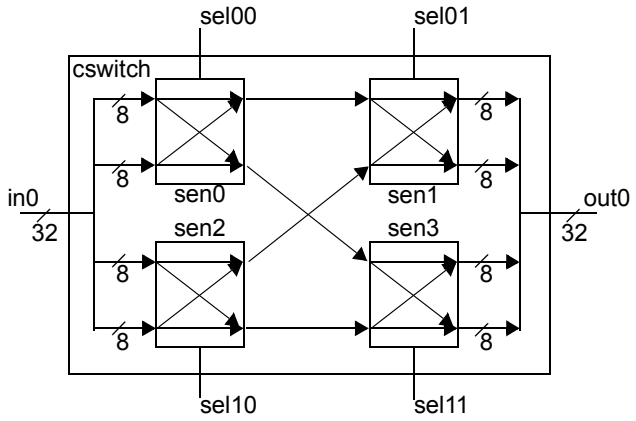
The objects inside the scope *unit_object_name* are prepended with the *unit_object_name*.

EXAMPLE :

//

CSL CODE

FIGURE 1.20 Clos Switch



CSL CODE

```
//AB
csl_unit cswitch,sen;
scope cswitch {
  //prepended with cswitch
  add_port(input,32,in0);
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  add_port(output,32,out0);
}
scope sen {
  //prepended with sen.
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  add_instance(sen,sen0);
  add_instance(sen,sen1);
  add_instance(sen,sen2);
  add_instance(sen,sen3);
```

```
    in0[31:24].connect(sen0.x0);
    in0[23:16].connect(sen0.x1);
    in0[15:8].connect(sen2.x0);
    in0[7:0].connect(sen2.x1);
    scope sen0 {
      y0.connect(sen1.x0);
      y1.connect(sen3.x0);
      sel.connect(sel00);
    }
    scope sen1 {
      y0.connect(out0(31,24));
      y1.connect(out0(23,16));
      sel.connect(sel01);
    }
    scope sen2 {
      y0.connect(sen1.x1);
      y1.connect(sen3.x1);
      sel.connect(sel10);
    }
    scope sen3 {
      y0.connect(out0(15,8));
      y1.connect(out0(7,0));
      sel.connect(sel11);
    }
  }
```

SEE ALSO :

VERILOG CODE

```
  //AV
  module cswitch(in0,out0, sel00,sel01,sel10,sel11);
      input [31:0] in0;
      input sel00,sel01,sel10,sel11;
      output [31:0] out0;
      wire sel00,sel01,sel10,sel11;
      wire [7:0] x00,x01,y00,y01;
      wire [7:0] x10,x11,y10,y11;
      wire [7:0] x20,x21,y20,y21;
      wire [7:0] x30,x31,y30,y31;
      assign x00 = in0[31:24];
      assign x01 = in0[23:16];
```

```
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name0.**add_instance**(*unit_object_name1*, *instance_object_name*[.*formal_connection_object_name*(*actual_connection_object_name*)]);

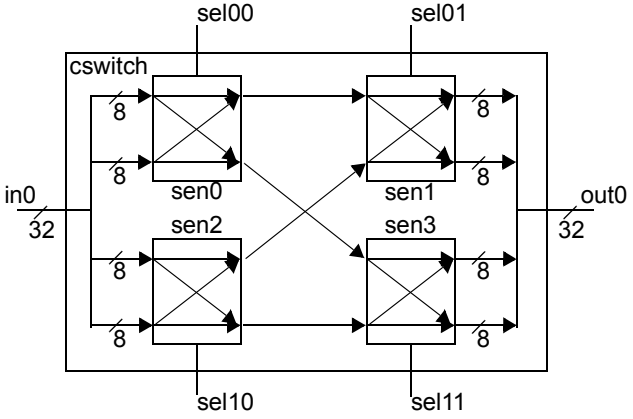
DESCRIPTION :

Create a new instance of *unit_object_name1* inside *unit_object_name* called *instance_object_name*. This command requires at least 2 csl units declarations in the current design where one is the *unit_object_name0* and the other is *unit_object_name1*. The CSL **instance** is the equivalent of a Verilog module instantiated in a module. Optionally the user can specify connections for the instance: *formal_connection_object_name* and *actual_connection_object_name* can be ports or interfaces.

EXAMPLE :

//description of the switch

FIGURE 1.21 Clos Switch



CSL CODE

```
//ab
csl_unit cswitch,sen;
scope cswitch {
  add_port(input,32,in0);
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  add_port(output,32,out0);
}
scope sen {
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
//sen unit behaviour goes here
}
scope cswitch {
```

```
//adds an instance of sen called sen0
cswitch.add_instance(sen,sen0);
//adds an instance of sen called sen1
cswitch.add_instance(sen,sen1);
/* adds an instance of sen called sen2 and make the connections at
instantiation */
cswitch.add_instance(sen,sen2(.x0(cswitch.in0[15:8]),.x1(cswitch.in0[7
:0]),.y0(sen1.x1),.y1(sen3.x1),.sel(sel10)));
connect
/* adds an instance of sen called sen3 and make the connections at
instantiation */
cswitch.add_instance(sen,sen3(.x0(sen0.y1),.x1(sen2.y1),.y0(cswitch.ou
t0[15:8]),.y1(cswitch.out0[7:0]),.sel(sel11)));
sen0.connect(.x0(in0[31:24]),.x1(in0[23:16],.sel(sel00)));
sen1.connect(.y0(out0[31:24]),.y1(out0[23:16]),.x0(sen0.y0),
.sel(sel01));
```

SEE ALSO :

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
```

```
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name.**add_instance_list**(*unit_object_name, list_object*);

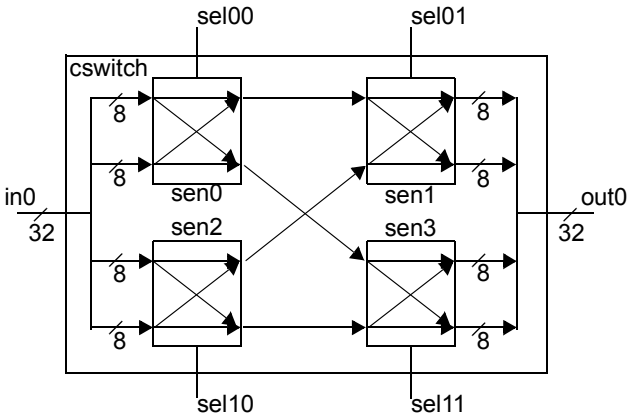
DESCRIPTION :

Create a new instance of *csl_unit_object_object_name* inside *unit_object_name* called *instance_object_name*. This command requires at least 2 csl units declarations in the current design where one is the *unit_object_name* and the other is *csl_unit_object_object_name*. The CSL **instance** is the equivalent of a Verilog module instantiated in a module.

EXAMPLE :

//description of the switch

FIGURE 1.22 Clos Switch



CSL CODE

```
//AB
csl_unit cswitch,sen;
scope cswitch {
  add_port(input,32,in0);
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  add_port(output,32,out0);
}
scope sen {
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  /* adds multiple instances of sen (4 instances) by declaring
  an anonymous list. Note that this list cannot be reused */
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
```

```
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
```

SEE ALSO :

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name.**add_port**(*port_direction*[,*port_type*][,*bitrange*],*port_name*);

DESCRIPTION :

Adds the port *port_name* or to the unit *unit_object_name*. The user specifies the port direction of ports for a **csl_unit** using **INPUT**, **OUTPUT** or **INOUT** in the *port_direction* signal attribute.

wrong, use below in add_port_list method

Instead of *port_object_name* there can be the name of a list of ports previously defined with **csl_list** command or an actual list of port names.

<this needs to be moved>

If the bit range is not specfied then the width of the port is the width of the signal. If the bit range is specified then the width of the signal is overridden and the width of the port is the width specified in the bit range. Note that the width of the bit range must be equal to or less than the width of the signal. For example, if the signal bit range is [31:0] and the bit range in the add_port function is [24:3] which is 22 bit wide then a port with a bit range [21:0] is declared. The signal in the upper scope is connected to the signal in the unit as follows: .x[21:0] (x[24:3]). Note that by setting the signal range we can select the bits of the upper signal to connect to the port that was added to the unit.

<end of this needs to be moved>

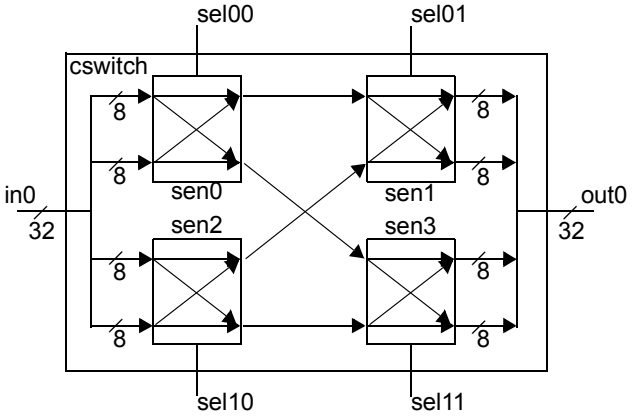
TABLE 1.2 Port types

Type	Description
INPUT	Input port
OUTPUT	Output port
INOUT	Inout port

EXAMPLE :

//description of the switch

FIGURE 1.23 Clos Switch



CSL CODE

```
//AB
```

```
csl_unit cswitch,sen;

// add_port() method prepended with the scope name
cswitch.add_port(input,32,in0);
cswitch.add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
cswitch.add_port(output,32,out0);
scope sen {
/* add_port() method within a scope. It applies to the current scope
object */
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
}
```

SEE ALSO :

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
```

```
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name.**add_port**(*port_direction*[,*port_type*][,*bitrange*],*port_name*);

DESCRIPTION :

Adds the port *port_name* or to the unit *unit_object_name*. The user specifies the port direction of ports for a **csl_unit** using **INPUT**, **OUTPUT** or **INOUT** in the *port_direction* signal attribute.

wrong, use below in add_port_list method

Instead of *port_object_name* there can be the name of a list of ports previously defined with **csl_list** command or an actual list of port names.

<this needs to be moved>

If the bit range is not specfied then the width of the port is the width of the signal. If the bit range is specified then the width of the signal is overridden and the width of the port is the width specified in the bit range. Note that the width of the bit range must be equal to or less than the width of the signal. For example, if the signal bit range is [31:0] and the bit range in the add_port function is [24:3] which is 22 bit wide then a port with a bit range [21:0] is declared. The signal in the upper scope is connected to the signal in the unit as follows: .x[21:0] (x[24:3]). Note that by setting the signal range we can select the bits of the upper signal to connect to the port that was added to the unit.

<end of this needs to be moved>

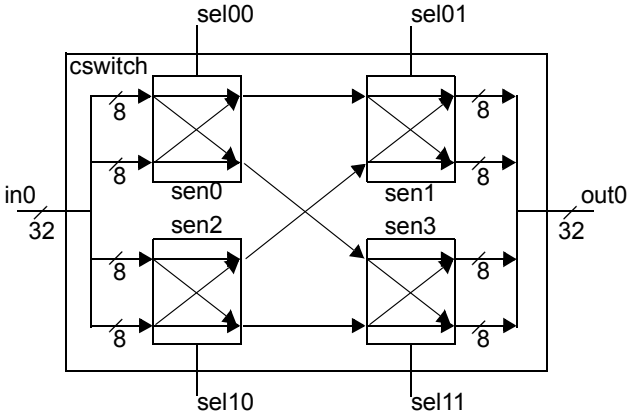
TABLE 1.3 Port types

Type	Description
INPUT	Input port
OUTPUT	Output port
INOUT	Inout port

EXAMPLE :

//description of the switch

FIGURE 1.24 Clos Switch



CSL CODE

```
//AB
```

```
csl_unit cswitch,sen;

// add_port() method prepended with the scope name
cswitch.add_port(input,32,in0);
cswitch.add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
cswitch.add_port(output,32,out0);
scope sen {
/* add_port() method within a scope. It applies to the current scope
object */
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
}
```

SEE ALSO :

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
```

```
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name.**add_port**(*port_direction*[,*port_type*][,*bitrange*],*signal_object_name*);

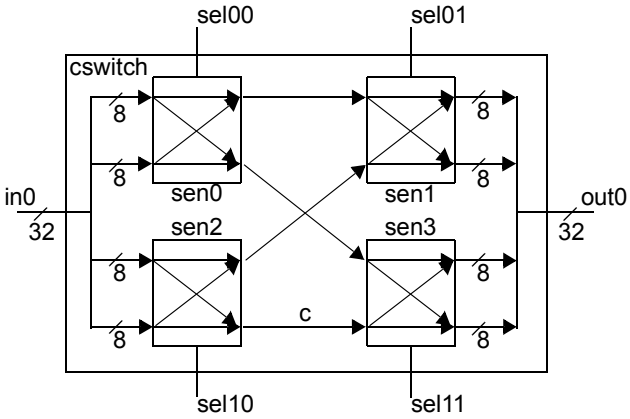
DESCRIPTION :

Adds a previously declared signal object to the units interface. The signal becomes a port. Also, *signal_object_name* can be another port (in another unit/interface). By specifying a different *port_direction* for an existing port object in the add_port method, the direction gets overriden, and the new port will use the overriden direction.

EXAMPLE :

//description of the switch

FIGURE 1.25 Clos Switch



CSL CODE

```
//AB
csl_unit cswitch,sen;
scope cswitch {
  //declare a signal in this scope and then set as port
  csl_signal in0;
  add_port(input,32,in0);
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  //declare a 32 bit signal (this defaults to wire)
  csl_signal out0(32);
  //set the signal data type to reg
  out0.set_type(reg);
  //use the reg as an output for the cswitch unit
  add_port(output,out0);
}
Question: by not specifying the port bitrange, is this attribute
infered from that of the csl_signal declared above ?
scope sen {
```

```
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  //add a signal to the cswitch unit
  add_signal(wire,8,c);
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  /* use the previously declared signal c as an
  intermmediate to make a connection between two siblings
  inside the unit cswitch */
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(c), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(c), .sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
```

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign c = y21;
    assign x31 = c;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
```

```
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

port_object_name.**reverse();**

DESCRIPTION :

Reverse the direction of a port. This method is called on a port object and it reverses its direction

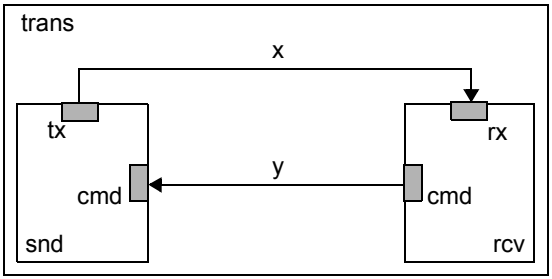
!!add to warn error doc

warn/erro should state that this method can only be called for input/output ports, otherwise the function has no effect and a warning should be produced

EXAMPLE :

Created a port for an unit and then reversed the direction for the copy of this port from another unit

FIGURE 1.26 A block named *trans* with two instances named *snd* and *rcv* interconnected



CSL CODE

```
//AV
csl_unit trans,sender,receiver;
scope sender {
  add_port(output,tx);
}
scope receiver {
  add_port(input,rx);
  add_port(sender.rx);
}
scope trans {
  add_instance(sender,snd);
  add_instance(receiver,recv);
  snd.add_port(input,cmd);
  rcv.add_port(snd.cmd);
  scope rcv {
    cmd.reverse();
  }
}
```

VERILOG CODE

```
//AV
module trans;
```

```
    wire x,y;
    sender snd(x,y);
    receiver rcv(x,y);
endmodule
module sender(tx,cmd);
    output tx;
    input cmd;
endmodule
module receiver(rx,cmd);
    input rx;
    output cmd;
endmodule
```

unit_object_name.**add_port_list**(*port_direction[,port_type][,bitrange],list_object*);

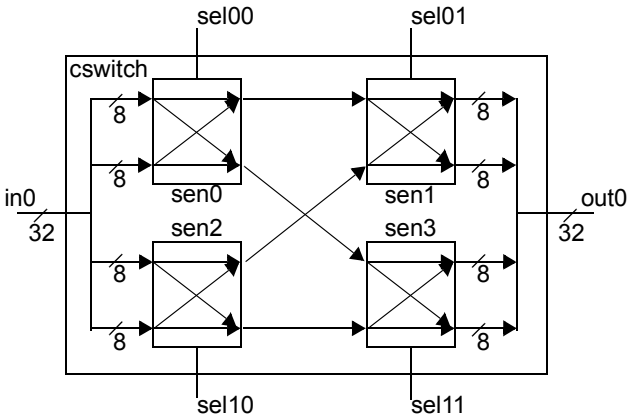
DESCRIPTION :

Adds a port list to the unit's interface. The portlist is declared within an CSL anoymous list (eg: **csl_list (a,b,c,d)**)

EXAMPLE :

//description of the switch

FIGURE 1.27 Clos Switch



CSL CODE

```
//AB
csl_unit cswitch,sen;
scope cswitch {
  add_port(input,32,in0);
  //using an anonymous list to add ports to cswitch unit's interface
  add_port_list(input,csl_list(sel00,sel01,sel10,sel11));
  add_port(output,32,out0);
scope sen {
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
scope cswitch {
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
```

```
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
}
```

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_name.**add_port_list(**[*port_direction*,]*interface_object*);

DESCRIPTION :

See if this doesn't conflict with the other add_port_list

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.

EXAMPLE :

In the following example the ports from a named interface within a unit are used to populate the default interface in another unit

CSL CODE

```
code needs to be re written and figure needs to be added
//AB - untested code
csl_unit a,b,c;
csl_interface ifc;
ifc.add_port_list(input, 2, csl_list(x,y));
ifc.add_port_list(output, 1, csl_list(z,t));
a.add_interface(ifc, ifc0);
b.add_port_list(a.ifc0);
c.add_port_list(input,a.ifc0);
```

VERILOG CODE

```
//AB - untested code
module a(ifc0_z,ifc0_t,ifc0_x,ifc0_y);
  output [1:0] ifc0_z,ifc0_t;
  input ifc0_x,ifc0_y;
endmodule

module b(z,t,x,y);
  output [1:0] z,t;
  input x,y;
endmodule

module c(x,y);
 input x,y;
endmodule
```

unit_object_name.**add_interface**(*port_direction,signal_group*);

DESCRIPTION :

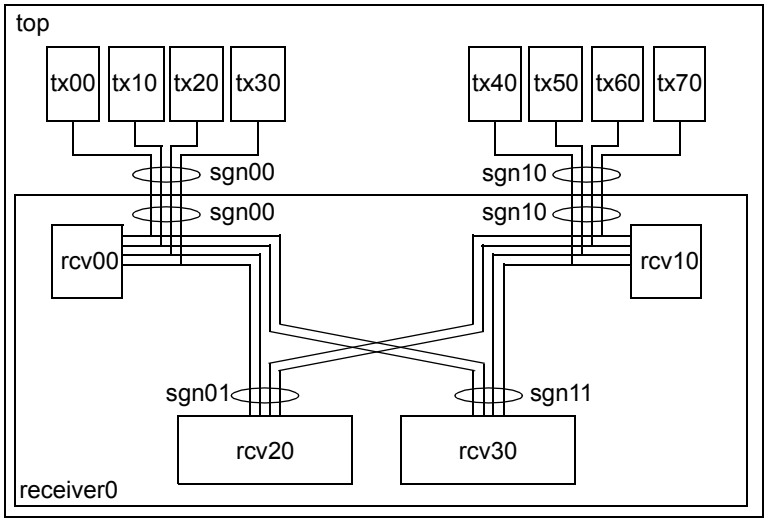
!!update this example to reflect the command syntax above and then move to interface - changed add_interface to add_interface

Adds a signal group as a port to the unit's interface, or as a port list if the signal_group_object_name.generate_individual_rtl_signals(status); is set

EXAMPLE :

This signals in a group are added as ports to a unit's interface. By setting the generate_individual_rtl_signals directive *status* on, the signal groups will generate individual ports for each signal as they "traverse" unit's scopes.

FIGURE 1.28 Intersected signal groups example



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
```

```
csl_signal_group sgn00(s0,s1,s2,s3);
//the following will generate a port for each signal in the group
sgn00.generate_individual_rtl_signals(on);
csl_signal_group sgn10(s4,s5,s6,s7);
//the following will generate a port for each signal in the group
sgn10.generate_individual_rtl_signals(on);
}
/* because of the individual rtl signals setting above the
following commands will generate a port for for each signal */
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  csl_signal_group sgn00(top.sgn00);
  csl_signal_group sgn10(top.sgn10);
  /* since sgn10 is a copy of top.sgn10 it also preserved the
  attribute regarding individual rtl signals generation and if
  this is not desired it can be turned off like below */
  sgn10.generate_individual_rtl_signals(off);
  /*the following groups are the same as above
  but will be modified */
  csl_signal_group sgn01(top.sgn00);
  csl_signal_group sgn11(top.sgn10);
   sgn11.generate_individual_rtl_signals(off);
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  /* this will behave differently than the group in
  the top scope because the individual rtl setting
  has been changed */
  add_interface(input,sgn00);
  //will generate only one port of concatenated signals from the group
  add_interface(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
  /*the connect method will automatically create ports and
  prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
```

```
  /*no need to connect sgn00 with sgn01 and sgn10 with sgn11
  because these groups share the same scope and only specify
  different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
/* this will create a port for each signal in the group
because it inherits the individual rtl signals attribute
of the signal group from which it was copied */
rcv2.add_interface(input,receiver.sgn01);
/* this concatenates the signals as they pass through the
interface and expandes them back on the other side */
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
}
scope top {
  add_instance(receiver,receiver0);
  add_instance(tx0,tx00);
  add_instance(tx1,tx10);
  add_instance(tx2,tx20);
  add_instance(tx3,tx30);
  add_instance(tx4,tx40);
  add_instance(tx5,tx50);
  add_instance(tx6,tx60);
  add_instance(tx7,tx70);
}
```

VERILOG CODE

```
//AB
`define WIDTH_S0 1
`define WIDTH_S1 1
`define WIDTH_S2 1
`define WIDTH_S3 1
`define WIDTH_S4 1
`define WIDTH_S5 1
`define WIDTH_S6 1
`define WIDTH_S7 1

module top();
```

```
wire [`WIDTH_S0-1:0] s0;
wire [`WIDTH_S1-1:0] s1;
wire [`WIDTH_S2-1:0] s2;
wire [`WIDTH_S3-1:0] s3;
wire [`WIDTH_S4-1:0] s4;
wire [`WIDTH_S5-1:0] s5;
wire [`WIDTH_S6-1:0] s6;
wire [`WIDTH_S7-1:0] s7;

tx0 tx00(.s0(s0));
tx1 tx10(.s1(s1));
tx2 tx20(.s2(s2));
tx3 tx30(.s3(s3));
tx4 tx40(.s4(s4));
tx5 tx50(.s5(s5));
tx6 tx60(.s6(s6));
tx7 tx70(.s7(s7));
receiver
receiver0(.s0(s0),.s1(s1),.s2(s2),.s3(s3),.s4(s4),.s5(s5),.s6(s6),.s7(
s7));

endmodule

module receiver(s0,s1,s2,s3,s4,s5,s6,s7);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S2-1:0] s2;
input [`WIDTH_S3-1:0] s3;
input [`WIDTH_S0-1:0] s4;
input [`WIDTH_S1-1:0] s5;
input [`WIDTH_S2-1:0] s6;
input [`WIDTH_S3-1:0] s7;
wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

assign {s4,s5,s6,s7} = sgn10;
assign sgn11 = {s2,s3,s6,s7};

rcv0 rcv00(.s0(s0),.s1(s1),.s2(s2),.s3(s3));
rcv1 rcv10(.sgn10(sgn10));
```

```
rcv2 rcv20(.s0(s0),.s1(s1),.s4(s4),.s5(s5));
rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(s0,s1,s2,s3);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S2-1:0] s2;
input [`WIDTH_S3-1:0] s3;
endmodule

module rcv1(sgn10);
input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(s0,s1,s4,s5);
input [`WIDTH_S0-1:0] s0;
input [`WIDTH_S1-1:0] s1;
input [`WIDTH_S4-1:0] s4;
input [`WIDTH_S5-1:0] s5;
endmodule

module rcv3(sgn11);
input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
```

```
output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
output [`WIDTH_S7-1:0] s7;
endmodule
```

signal_group unit_object_name.**get_inputs();**

DESCRIPTION :

fix description

Returns a group of the input ports from the unit named *unit_object_name*. This is very useful for interconnect two units.

EXAMPLE :

//.

add example with code and figure

CSL CODE

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

signal_group unit_object_name.**get_outputs();**

DESCRIPTION :

fix description

Returns the width of a single dimensional signal, otherwise it generates a cslc compile time error.

!!add this to warn error document - where does this belong?

Returns a group of the output ports from the unit named *unit_object_name.* This is very useful for interconnect two units.

EXAMPLE :

//.

add example with code and figure

CSL CODE

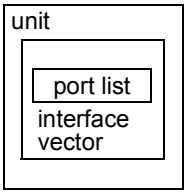
```
//csl code goes here
```

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.

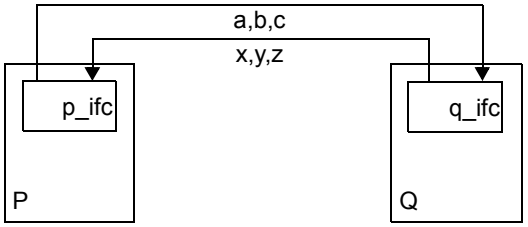
FIGURE 1.29 Interface organization



EXAMPLE :

In this example two interface objects are created, modified and then assigned to different units

FIGURE 1.30



CSL CODE

```
//AB
csl_unit p,q,top;
//create two interface objects
csl_interface p_ifc,q_ifc;
csl_list abc_list(a, b, c);
csl_list xyz_list(x, y, z);
p_ifc.add_port_list(input,xyz_list);
p_ifc.add_port_list(output,abc_list);
q_ifc.add_port_list(output,xyz_list);
q_ifc.add_port_list(input,abc_list);
p.add_interface(p_ifc);
q.add_interface(q_ifc);
top.add_instance(p, p0);
top.add_instance(q, q0);
```

VERILOG CODE

```
//AB
module top();
```

```
p p0 (.a(q0.x),.b(q0.y),.c(q0.z),.x(q0.a),.y(q0.y),.z(q0.c));
q q0 (.x(p0.a),.y(p0.b),.z(p0.c),.a(p0.x),.b(p0.y),.c(p0.z));

endmodule

module p(a,b,c,x,y,z);
  output a,b,c;
  input x,y,z;
endmodule

module q(x,y,z,a,b,c);
  output x,y,z;
  input a,b,c;
endmodule
```

csl_interface *interface_object_name1*(*interface_object_name0*);

DESCRIPTION :

Creates a new interface by copying the interface object passed as constructor argument.

EXAMPLE :

Create an interface for a half-adder and after that use if to create an interface for an full-adder by copying the frist one and adding another input port to it.

FIGURE 1.31 Two units with the similar interface

CSL CODE

```
csl_interface ha_ifc;
ha_ifc.add_port_list(input, csl_list(x, y));
ha_ifc.add_port_list(output, csl_list(sum, co));
csl_interface fa_ifc(ha_ifc);
fa_ifc.add_port(input, ci);
csl_unit HA, FA;
HA.add_interface(ha_ifc, ha_ifc);
FA.add_interface(fa_ifc, fa_ifc);
```

VERILOG CODE

```
module HA(x, y, sum, co);
   input x,y;
   output sum, co;
endmodule
module FA(ci, x, y, sum, co)
    output ci, x, y;
    input sum, co;
endmodule
```

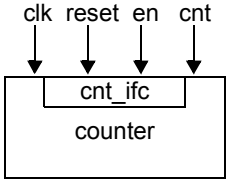
csl_interface *interface_object_name(port_declaration, {port_declaration});*

DESCRIPTION :

port_width can be a width, a range with *lower_limit* and *upper_limit* or a *bitrange_object_name.*
port_declaration = port_name (port_direction [,port_width] [, port_type]);

EXAMPLE :

FIGURE 1.32



CSL CODE

```
csl_interface ifc(csl_list(clk, reset, en)(input), cnt(output, 4,
reg));
csl_unit counter;
couter.add_interface(ifc, cnt_ifc);
```

VERILOG CODE

```
module counter(clk, reset, en, cnt);
    input clk, reset, en;
    output [3:0] cnt;
    reg [3:0] cnt;
endmodule
```

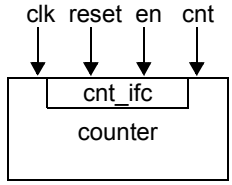
csl_interface *interface_object_name(csl_list (port_direction[, port_width][, signal_type]));*

DESCRIPTION :

port_width can be a width, a range with *lower_limit* and *upper_limit* or a *bitrange_object_name.*

EXAMPLE :

FIGURE 1.33



CSL CODE

```
csl_interface ifc(clk (input), reset(input), en(input), cnt(output, 4, reg));
csl_unit counter;
```

VERILOG CODE

```
module counter(clk, reset, en, cnt);
    input clk, reset, en;
    output [3:0] cnt;
    reg [3:0] cnt;
endmodule
```

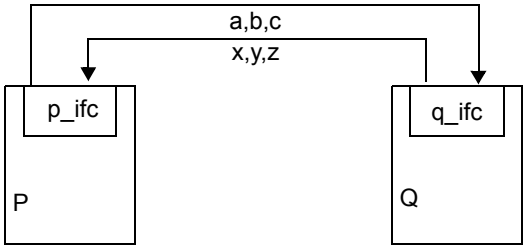
interface_object_name.**reverse();**

DESCRIPTION :

Reverse will invert the input ports and make them output ports, while the output ports will become input ports for *interface_object_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.

EXAMPLE :

FIGURE 1.34



CSL CODE

```
//AB
csl_unit p,q,top;
//create the first interface
csl_interface p_ifc;
csl_list abc_list(a, b, c);
csl_list xyz_list(x, y, z);
p_ifc.add_port_list(input,xyz_list);
p_ifc.add_port_list(output,abc_list);
//create the second interface by copying the first
csl_interface q_ifc(p_ifc);
//reverse the second interface to connect the first one
q_ifc.reverse();
p.add_interface(p_ifc);
q.add_interface(q_ifc);
top.add_instance(p, p0);
top.add_instance(q, q0);
```

VERILOG CODE

```
//AB
module top();

  p p0 (.a(q0.x),.b(q0.y),.c(q0.z),.x(q0.a),.y(q0.y),.z(q0.c));
  q q0 (.x(p0.a),.y(p0.b),.z(p0.c),.a(p0.x),.b(p0.y),.c(p0.z));
endmodule
```

```
module p(a,b,c,x,y,z);
  output a,b,c;
  input x,y,z;
endmodule

module q(x,y,z,a,b,c);
  output x,y,z;
  input a,b,c;
endmodule
```

interface_object_name.**reverse();**

CSL CODE

```
//AB
csl_unit p,q,top;
//create the first interface
csl_interface p_ifc;
csl_list abc_list(a, b, c);
csl_list xyz_list(x, y, z);
p_ifc.add_port_list(input,xyz_list);
p_ifc.add_port_list(output,abc_list);
//create the second interface by copying the first
csl_interface q_ifc(p_ifc);
//reverse the second interface to connect the first one
q_ifc.reverse();
p.add_interface(p_ifc);
q.add_interface(q_ifc);
top.add_instance(p, p0);
top.add_instance(q, q0);
```

unit_object_name.**add_interface(***interface_name*[,*interface_instance_name*]**);**

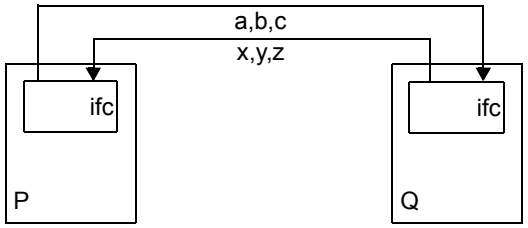
DESCRIPTION :

add_interface() will add to the default interface of *unit_object_name* an instance of interface_name. The second parameter is optional and if it's not used the ports names don't get prepended with *interface_instance_name* (this would be useful to keep port names in generated Verilog modules).

!!CSLOm writer needs to agree on this; last time it was ok

EXAMPLE :

FIGURE 1.35



CSL CODE

```
//AB
csl_unit p,q;
//create the first interface
csl_interface p_ifc;
csl_list abc_list(a, b, c);
csl_list xyz_list(x, y, z);
p_ifc.add_port_list(input,xyz_list);
p_ifc.add_port_list(output,abc_list);
//set the interface of the p module
p.add_interface(p_ifc);
//set the interface of the q module
q.add_interface(p.get_interface());
q.get_interface().reverse();
top.add_instance(p, p0);
top.add_instance(q, q0);
```

VERILOG CODE

```
fix code - use wires in top to connect instances
//AB
module top();

  p p0 (.a(q0.x),.b(q0.y),.c(q0.z),.x(q0.a),.y(q0.y),.z(q0.c));
  q q0 (.x(p0.a),.y(p0.b),.z(p0.c),.a(p0.x),.b(p0.y),.c(p0.z));
endmodule
```

```
module p(a,b,c,x,y,z);
  output a,b,c;
  input x,y,z;
endmodule

module q(x,y,z,a,b,c);
  output x,y,z;
  input a,b,c;
endmodule
```

interface_object_name.**add_interface**(*interface_name*[,*interface_instance_name*]);

DESCRIPTION :

Adds an instance of the interface named *interface_name* to the *interface_object_name* object.The second parameter is optional and when used ports added by the command above to the *interface_object_name* get prepended with the *interface_instance_name*.If the second parameter is not used,the ports of the interface *interface_name* are aded directly to *interface_object_name*,without any prepending.

EXAMPLE :

..

CSL CODE

```
//
```

VERILOG CODE

```
//
```

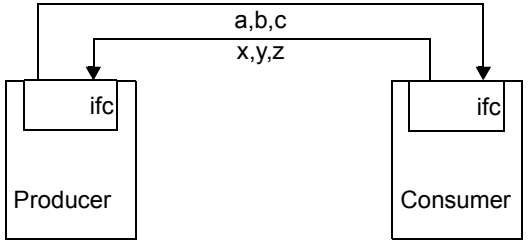
interface_object unit_object_name.**get_interface();**

DESCRIPTION :

Returns an object of type interface. It can only be called on a unit/instance object.

EXAMPLE :

FIGURE 1.36



CSL CODE

```
//AB
csl_unit producer,consumer;
//create the first interface
csl_interface producer_ifc;
csl_list abc_list(a, b, c);
csl_list xyz_list(x, y, z);
producer_ifc.add_port_list(input,xyz_list);
producer_ifc.add_port_list(output,abc_list);
producer.add_interface(producer_ifc);
//use get_interface with set_interface
consumer.add_interface(producer.get_interface());
//reverse module q's interface using get_interface
consumer.get_interface().reverse();
top.add_instance(producer, producer0);
top.add_instance(consumer, consumer0);
```

VERILOG CODE

```
//AB
module top();

  poducer poducer0
(.a(consumer0.x),.b(consumer0.y),.c(consumer0.z),.x(consumer0.a),.y(co
nsumer0.y),.z(consumer0.c));
  consumer consumer0
(.x(poducer0.a),.y(poducer0.b),.z(poducer0.c),.a(poducer0.x),.b(produce
r0.y),.c(poducer0.z));
endmodule
```

```
module producer(a,b,c,x,y,z);
  output a,b,c;
  input x,y,z;
endmodule

module consumer(x,y,z,a,b,c);
  output x,y,z;
  input a,b,c;
endmodule
```

unit_object_name.**add_interface**(port_direction,*interface_object_name*);

DESCRIPTION :

This function will change every port direction in the port objects in the *interface_object_name* to the port direction specified by **port_direction**. This function overrides the port directions fo the objects in the *interface_object_name*. This function will cause a compiler error if all port directions are not the same in *interface_object_name* . This function will cause a compiler error if the port direction specified by **port_direction** do not match the inferred port directions in *interface_object_name* .

!!add to warn error

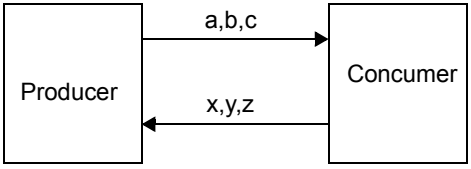
port_direction = *input*|*output*|*inout*

Note if the user wants to override only the inputs in a *interface_object_name* and change the inputs to outputs then the following method should be used

unit_object_name.**add_interface(output,** *interface_object_name* .get_interface**(input));**

EXAMPLE :

FIGURE 1.37



CSL CODE

```
//AB
csl_unit producer,consumer,r,top;
//create a interface object
csl_interface producer_ifc;
producer_ifc.add_port_list(output,csl_list(a,b));
producer_ifc.add_port_list(input,csl_list(x,y));
producer.add_interface(producer_ifc);
//use only the output ports from the interface
consumer.add_interface(output,p_ifc);
consumer.get_interface().reverse();
//use only the input ports from the interface
r.add_interface(input,producer_ifc);
r.get_interface().reverse();
top.add_instance(producer,producer0);
top.add_instance(consumer,consumer0);
```

```
top.add_instance(r,r0);
```

VERILOG CODE

```
//AB
module top();
  wire a0;
  wire b0;
  wire x0;
  wire y0;
producer producer0(.a(a0),.b(b0),.x(x0),.y(y0));
consumer consumer0(.a(a0),.b(b0));
r r0(.x(x0),.y(y0));

endmodule

module producer(a,b,x,y);
  output a,b;
  input x,y;
endmodule

module consumer(a,b);
  input a,b;
endmodule

module r(x,y);
  output x,y;
endmodule
```

interface_object unit_object_name.**get_interface**(*port_direction*);

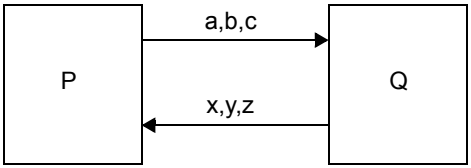
DESCRIPTION :

Return a *interface_object_name* which is a list of all ports in *unit_object_name* with **port_direction.**

port_direction = *INPUT|OUTPUT|INOUT*

EXAMPLE :

FIGURE 1.38



CSL CODE

```
//AB
csl_unit p,q,r,top;
//create a interface object
csl_interface p_ifc;
p_ifc.add_port_list(output,csl_list(a,b));
p_ifc.add_port_list(input,csl_list(x,y));
p.add_interface(p_ifc);
//use only the output ports from the p interface
q.add_interface(p.get_interface(output));
q.get_interface().reverse();
//use only the input ports from the p interface
r.add_interface(p.get_interface(input));
r.get_interface().reverse();
top.add_instance(p,p0);
top.add_instance(q,q0);
top.add_instance(r,r0);
```

VERILOG CODE

```
//AB
module top();
  wire a0;
  wire b0;
  wire x0;
  wire y0;
p p0(.a(a0),.b(b0),.x(x0),.y(y0));
q q0(.a(a0),.b(b0));
r r0(.x(x0),.y(y0));
```

```
endmodule

module p(a,b,x,y);
  output a,b;
  input x,y;
endmodule

module q(a,b);
  input a,b;
endmodule

module r(x,y);
  output x,y;
endmodule
```

unit_object_name.**add_interface_group**(*interface_group_name*);

DESCRIPTION :

need to clear this up

//

EXAMPLE :

//

CSL CODE

```
csl_interface ifc(csl_list(ci, x, y)(input), csl_list(co, s)(output));
csl_unit FA0, FA1, FA2, 3_bit_FA;
FA0.add_interface(ifc);
FA1.add_interface(ifc);
FA2.add_interface(ifc);
csl_interface ifco(ifc), ifc1(ifc), ifc2(ifc);
ifc0.remove_port(co);
ifc1.remove_port_list(csl_list(co, ci));
ifc2.remove_port(ci);
3_bit_FA.add_interface_list(csl_list(ifc0, ifc1, ifc2));
```

VERILOG CODE

```
//verilog code goes here
```

interface_object_name.**add_port**(*port_direction[,port_type][,bitrange],port_name*);

DESCRIPTION :

Adds the port *port_name* to the interface *interface_object_name*. The user specifies the port direction of ports for a **csl_unit** using **INPUT**, **OUTPUT** or **INOUT** in the *port_type* signal attribute.

TABLE 1.4 Port types

Type	Description
PT_INPUT	Input port
PT_OUTPUT	Output port
PT_INOUT	Inout port

EXAMPLE :

//

CSL CODE

```
//AB
csl_unit p,q,r,top;
//create a interface object
csl_interface p_ifc;
csl_bitrange br1;
br1.set_range(3,0);
p_ifc.add_port(output,a);
p_ifc.add_port(output,2,b);
p_ifc.add_port(input,[3:0],x);
p_ifc.add_port(input,br1,y);
p.add_interface(p_ifc);
//use only the output ports from the interface
q.add_interface(output,p_ifc);
q.get_interface().reverse();
//use only the input ports from the interface
r.add_interface(input,p_ifc);
r.get_interface().reverse();
top.add_instance(p,p0);
top.add_instance(q,q0);
top.add_instance(r,r0);
```

VERILOG CODE

```
//AB
`define BR1 3:0

module top();
  wire a0;
```

```
  wire [1:0] b0;
  wire [3:0] x0;
  wire [BR1] y0;
p p0(.a(a0),.b(b0),.x(x0),.y(y0));
q q0(.a(a0),.b(b0));
r r0(.x(x0),.y(y0));

endmodule

module p(a,b,x,y);
  output a;
  output [1:0] b;
  input  [3:0] x;
  input  [BR1] y;
endmodule

module q(a,b);
  input a;
  input [1:0] b;
endmodule

module r(x,y);
  output [3:0] x;
  output [BR1] y;
endmodule
```

interface_object_name.**remove_port(***port_name***);**

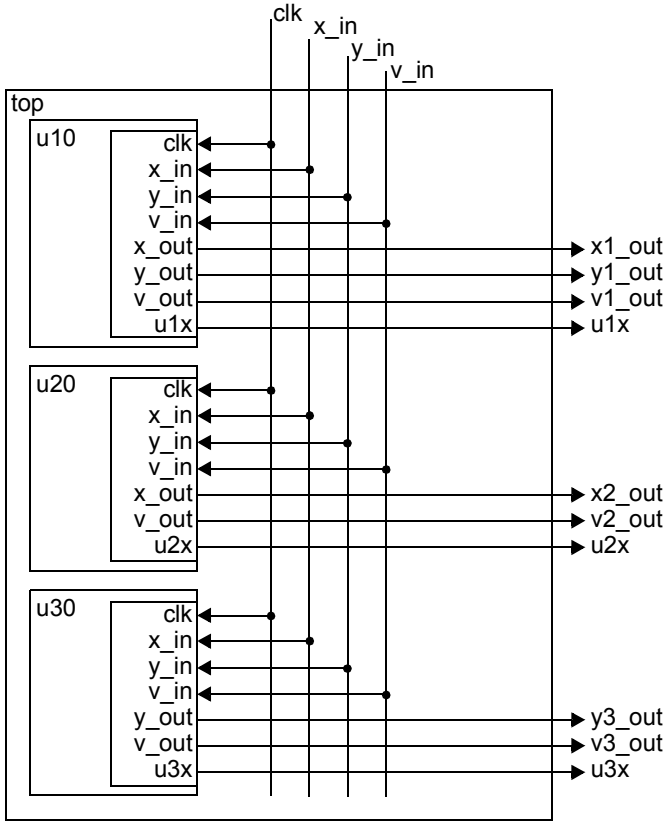
DESCRIPTION :

Removes a port from a unit's interface. This is useful when the interface of a unit is reused and some ports are not needed or need to be changed.

EXAMPLE :

In the following example, three units share similar interfaces (receive the same data, but output different data). The remove_port() method is used along with copy constructors to customize these interfaces.

FIGURE 1.39 Units sharing similar interfaces



CSL CODE

```
//AB
csl_unit top, u1, u2, u3;
scope top {
  add_port_list(input, csl_list(clk,x_in,y_in,v_in));
  add_port_list(output, csl_list(
x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x));
```

```
}

csl_interface if1;
if1.add_port_list(input,csl_list(clk,x_in,y_in,v_in));
if1.add_port_list(output,csl_list(x_out,y_out,v_out,u1x));
csl_interface if2(if1);
//ports are removed to customize the interface
if2.remove_port(u1x);
if2.remove_port(y_out);
if2.add_port(output,u2x);
csl_interface if3(if1);
if3.remove_port_list(csl_list(u1x,x_out));
if3.add_port(output,u3x);

u1.add_interface(if1,if1);
u2.add_interface(if2,if2);
u3.add_interface(if3,if3);

top.add_instance(u1,u10(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.x_out(x1_out),.y_out(y1_out),.v_out(v1_out),.u1x(u1x)));
top.add_instance(u2,u20(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.x_out(x2_out),.v_out(v2_out),.u2x(u2x)));
top.add_instance(u3,u30(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.y_out(y3_out),.v_out(v3_out),.u3x(u3x)));
```

VERILOG TYPE

```
//AB
module top(
x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x,clk,x_in,
y_in,v_in);
  input clk,x_in,y_in,v_in;
  output x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x;
  u1 u10(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.x_out(x1_out),.y_out(y1
_out),.v_out(v1_out),.u1x(u1x));
  u2 u20(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.x_out(x2_out),.v_out(v2
_out),.u2x(u2x));
  u3 u30(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.y_out(y3_out),.v_out(v3
_out),.u3x(u3x));
endmodule
```

```
module u1(x_out,y_out,v_out,u1x,clk,x_in,y_in,v_in);
  input clk,x_in,y_in,v_in;
  output x_out,y_out,v_out,u1x;
endmodule

module u2(x_out,v_out,u2x,clk,x_in,y_in,v_in);
  input clk,x_in,y_in,v_in;
  output x_out,v_out,u2x;
endmodule

module u3(y_out,v_out,u3x,clk,x_in,y_in,v_in);
  input clk,x_in,y_in,v_in;
  output y_out,v_out,u3x;
endmodule
```

interface_object_name.**add_port_list(***port_direction[,port_type][,bitrange],list_object***);**

DESCRIPTION :

Adds a port list to the unit's interface. The portlist can be declared within a CSL anoymous list (eg: **csl_list (a,b,c,d)**)

EXAMPLE :

//

CSL CODE

```
///AB
csl_unit p,q,r,top;
//create a interface object
csl_interface p_ifc;
p_ifc.add_port_list(output,csl_list(a,b));
p_ifc.add_port_list(input,csl_list(x,y));
p.add_interface(p_ifc);
//use only the output ports from the interface
q.add_interface(output,p_ifc);
q.get_interface().reverse();
//use only the input ports from the interface
r.add_interface(input,p_ifc);
r.get_interface().reverse();
top.add_instance(p,p0);
top.add_instance(q,q0);
top.add_instance(r,r0);
```

VERILOG CODE

```
//AB
module top();
  wire a0;
  wire b0;
  wire x0;
  wire y0;
p p0(.a(a0),.b(b0),.x(x0),.y(y0));
q q0(.a(a0),.b(b0));
r r0(.x(x0),.y(y0));

endmodule

module p(a,b,x,y);
  output a,b;
  input x,y;
```

```
endmodule

module q(a,b);
  input a,b;
endmodule

module r(x,y);
  output x,y;
endmodule
```

nterface_object_name.**remove_port_list(***list_object***);**

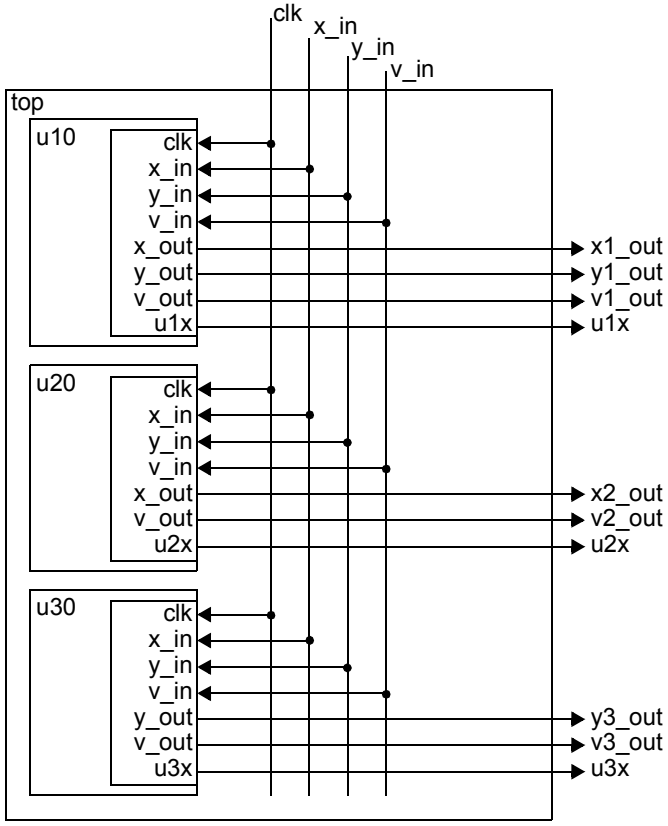
DESCRIPTION :

Removes a list of ports from a unit's interface. This is useful when the interface of a unit is reused and some ports are not needed or need to be changed. The list object can be previously declared or an anonymous list could be used; remove_port_list() can remove ports of different widths at once.

EXAMPLE :

In the following example, three units share similar interfaces (receive the same data, but output different data). The remove_port_list() method is used along with copy constructors to customize these interfaces.

FIGURE 1.40 Units sharing similar interfaces



CSL CODE

```
//AB
csl_unit top, u1, u2, u3;
scope top {
  add_port_list(input, csl_list(clk,x_in,y_in,v_in));
  add_port_list(output, csl_list(
```

```
x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x));
}

csl_interface if1;
if1.add_port_list(input,csl_list(clk,x_in,y_in,v_in));
if1.add_port_list(output,csl_list(x_out,y_out,v_out,u1x));
csl_interface if2(if1);
//ports are removed to customize the interface
if2.remove_port_list(csl_list(y_out,u1x));
if2.add_port(output,u2x);
csl_interface if3(if1);
if3.remove_port_list(csl_list(u1x,x_out));
if3.add_port(output,u3x);

u1.add_interface(if1,if1);
u2.add_interface(if2,if2);
u3.add_interface(if3,if3);

top.add_instance(u1,u10(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.x_out(x1_out),.y_out(y1_out),.v_out(v1_out),.u1x(u1x)));
top.add_instance(u2,u20(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.x_out(x2_out),.v_out(v2_out),.u2x(u2x)));
top.add_instance(u3,u30(.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),
.y_out(y3_out),.v_out(v3_out),.u3x(u3x)));
```

VERILOG TYPE

```
//AB
module top(
x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x,clk,x_in,
y_in,v_in);
  input clk,x_in,y_in,v_in;
  output x1_out,y1_out,v1_out,u1x,x2_out,v2_out,u2x,y3_out,v3_out,u3x;
  u1 u10(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.x_out(x1_out),.y_out(y1
_out),.v_out(v1_out),.u1x(u1x));
  u2 u20(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.x_out(x2_out),.v_out(v2
_out),.u2x(u2x));
  u3 u30(
.clk(clk),.x_in(x_in),.y_in(y_in),.v_in(v_in),.y_out(y3_out),.v_out(v3
_out),.u3x(u3x));
endmodule
```

```
    module u1(x_out,y_out,v_out,u1x,clk,x_in,y_in,v_in);
      input clk,x_in,y_in,v_in;
      output x_out,y_out,v_out,u1x;
    endmodule

    module u2(x_out,v_out,u2x,clk,x_in,y_in,v_in);
      input clk,x_in,y_in,v_in;
      output x_out,v_out,u2x;
    endmodule

    module u3(y_out,v_out,u3x,clk,x_in,y_in,v_in);
      input clk,x_in,y_in,v_in;
      output y_out,v_out,u3x;
endmodule
```

interface_object_name.**add_interface**(*port_direction,signal_group*);

DESCRIPTION :

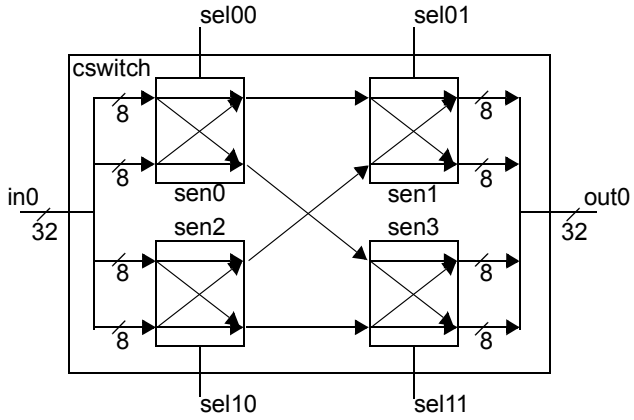
pending removal

Adds a signal group to the unit's interface.

EXAMPLE :

An example will be provided here

FIGURE 1.41 Clos Switch



CSL CODE

```
//AV
csl_unit cswitch,sen;
//create the interfaces
csl_interface selection;
csl_interface input_signs,output_signs;
scope cswitch {
  add_port(input,32,in0);
  //using an anonymous list to add ports to cswitch unit's interface
  csl_signal_group sel (csl_list(sel00,sel01,sel10,sel11));
  add_interface(selection);
  selection.add_interface(input,sel);
  add_port(output,32,out0);
scope sen {
  csl_signal_group in (csl_list(x0,x1));
  csl_signal_group out (csl_list(y0,y1));
  add_interface();
  add_port_list(input,8,csl_list(x0,x1));
  add_port_list(output,8,csl_list(y0,y1));
  add_port(input,sel);
}
```

```
scope cswitch {
  add_instance_list(sen,csl_list(sen0,sen1,sen2,sen3));
  sen0.connect(.x0(in0[31:24]), .x1(in0[23:16]), .y0(sen1.x0),
.sel(sel00));
  sen1.connect(.x1(sen2.y0), .y0(out0[31:24]), .y1(out0[23:16]),
.sel(sel01));
  sen2.connect(.x0(in0[15:8]), .x1(in0[7:0]), .y0(sen1.x1),
.sel(sel10));
  sen3.connect(.x0(sen0.y1), .y0(out0[15:8]), .y1(out0[7:0]),
.sel(sel11));
}
```

VERILOG CODE

```
//AV
module cswitch(in0,out0, sel00,sel01,sel10,sel11);
    input [31:0] in0;
    input sel00,sel01,sel10,sel11;
    output [31:0] out0;
    wire sel00,sel01,sel10,sel11;
    wire [7:0] x00,x01,y00,y01;
    wire [7:0] x10,x11,y10,y11;
    wire [7:0] x20,x21,y20,y21;
    wire [7:0] x30,x31,y30,y31;
    assign x00 = in0[31:24];
    assign x01 = in0[23:16];
    assign x20 = in0[15:8];
    assign x21 = in0[7:0];
    assign y10 = out0[31:24];
    assign y11 = out0[23:16];
    assign y30 = out0[15:8];
    assign y31 = out0[7:0];
    sen sen0(x00,x01,y00,y01,sel00);
    sen sen1(x10,x11,y10,y11,sel01);
    sen sen2(x20,x21,y20,y21,sel10);
    sen sen3(x30,x31,y30,y31,sel11);
endmodule

module sen(x0,x1,y0,y1,sel);
    input [7:0] x0,x1;
    input sel;
    output [7:0] y0,y1;
endmodule
```

unit_object_name.**add_signal**([*signal_data_type,*][*bitrange,*]*signal_name*);

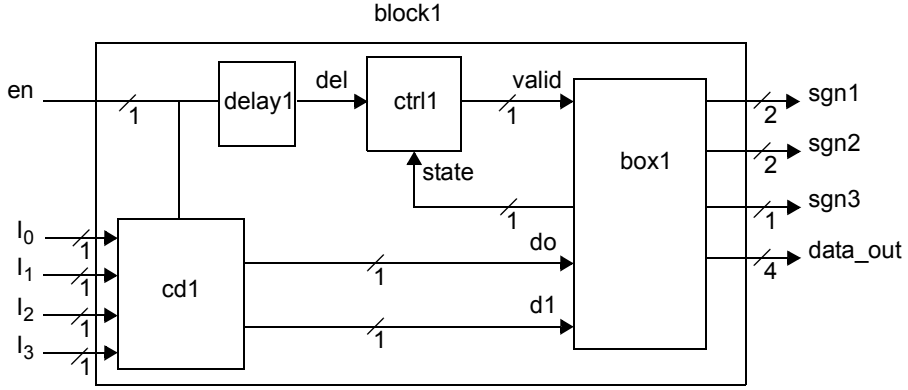
DESCRIPTION :

Add the signal *signal_object_name* to the unit *unit_object_name.* Multiple signals that share the same characteristics may be added by simply writing them in the argument list of the add_signal() method or by first creating a list of signals, and using the list name as a parameter instead of *signal_object_name*. The following parameters for this method are optional: *bitrange* specifies the bit range of the declared signal. If there is a list of signals, the bit range is applied to all the signals in list; *signal_data_type* refers to the data type of the signal: this can be **wire** or **reg**;

EXAMPLE :

Create an unit with 4 instances, and then add input and output ports and internal signals.

FIGURE 1.42 An unit with input signals, output signals and internal wires



CSL CODE

```
// AV
csl_unit block1, delay, cd, ctrl, box;
scope block1 {
  csl_signal i0,i1,i2,i3,en;
  csl_signal sgn1(1,0), sgn2(1,0), sgn3, data_out (3,0);
  csl_list input_signals(i0,i1,i2,i3);
  add_port_list(input,1,input_signals);
  add_port(input,1,en);
  csl_list output_signals(sgn1,sgn2,sgn3,data_out);
  add_port_list(output,output_signals);
  add_signal(wire,1,del);
  add_signal(wire,1,valid);
  add_signal(reg,1,state);
  add_signal_list(wire,1,csl_list(d0,d1));
  add_instance(delay,delay1);
  add_instance(cd,cd1);
```

```
  add_instance(ctrl,ctrl1);
  add_instance(box,box1);
}
delay.add_port(input,1,en);
delay.add_port(output,1,del);
cd.add_port_list(input,1,input_signals);
cd.add_port(output,1,do);
cd.add_port(output,1,d1);
ctrl.add_port_list(input,1,csl_list(del,state));
ctrl.add_port(input,1,valid);
box.add_port_list(input,1,csl_list(valid,d0,d1));
box.add_port_list(input,output_signals);
box.add_port(output,1,state);
//if auto route is set then the signals will be automatically connected
by name and the next lines can be ignored
scope block1 {
  delay1.connect(.en(en),.del(ctrl1.del));
  cd1.connect(.i0(i0),.i1(i1),.i2(i2),.i3(i3),.en(en),
.do(box1.do),.d1(box1.d1));
  ctrl1.connect(.valid(box1.valid),.state(box1.state));
  box1.connect(.sgn1(sgn1),.sgn2(sgn2),.sgn3(sgn3),
.data_out(data_out));
}
```

VERILOG CODE

```
// AV
module block1(en,i0,i1,i2,i3,sgn1,sgn2,sgn3,data_out);
    input en, i0,i1,i2,i3;
    output [1:0] sgn1, sgn2;
    output sgn3;
    output [3:0] data_out;
    reg [3:0] data_out;
    wire i0,i1,i2,i3,en,del,valid,d0,d1,sgn3;
    wire [1:0] sgn1, sgn2;
    wire [3:0] d_out;
    wire state;
    delay delay1(.en(en),.del(del));
    cd cd1(.en(en),.i0(i0),.i1(i1),.i2(i2),.i3(i3),.d0(d0),.d1(d1));
    ctrl ctrl1(.del(del),.state(state),.valid(valid));
```

```
    box
box1(.valid(valid),.d0(d0),.d1(d1),.state(state),.sgn1(sgn1),.sgn2(sgn
2),.sgn3(sgn3),.data_out(d_out));
endmodule
module delay(en,del);
    input en;
    output del;
endmodule
module cd(en,i0,i1,i2,i3,d0,d1);
    input en,i0,i1,i2,i3;
    output d0,d1;
endmodule
module ctrl(del,state,valid);
    input del,state;
    output valid;
endmodule
module box(valid,d0,d1,state,sgn1,sgn2,sgn3,data_out);
    input valid,d0,d1;
    output [1:0] sgn1,sgn2;
    output sgn3,state;
    output[3:0] data_out;
endmodule
```

unit_object_name.**add_signal_list**([*signal_data_type,*][*bitrange,*] *list_object*);

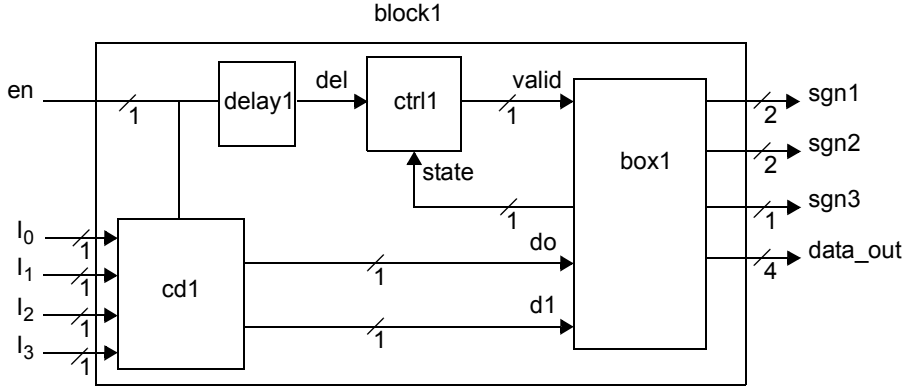
DESCRIPTION :

Add the signal *signal_object_name* to the unit *unit_object_name.* Multiple signals that share the same characteristics may be added by simply writing them in the argument list of the add_signal() method or by first creating a list of signals, and using the list name as a parameter instead of *signal_object_name*. The following parameters for this method are optional: *bitrange* specifies the bit range of the declared signal. If there is a list of signals, the bit range is applied to all the signals in list; *signal_data_type* refers to the data type of the signal: this can be **wire** or **reg**;

EXAMPLE :

Create an unit with 4 instances, and then add input and output ports and internal signals.

FIGURE 1.43 An unit with input signals, output signals and internal wires



CSL CODE

```
// AV
csl_unit block1, delay, cd, ctrl, box;
scope block1 {
  csl_signal i0,i1,i2,i3,en;
  csl_signal sgn1(1,0), sgn2(1,0), sgn3, data_out (3,0);
  csl_list input_signals(i0,i1,i2,i3);
  add_port_list(input,1,input_signals);
  add_port(input,1,en);
  csl_list output_signals(sgn1,sgn2,sgn3,data_out);
  add_port_list(output,output_signals);
  add_signal(wire,1,del);
  add_signal(wire,1,valid);
  add_signal(reg,1,state);
  add_signal_list(wire,1,csl_list(d0,d1));
  add_instance(delay,delay1);
  add_instance(cd,cd1);
```

```
  add_instance(ctrl,ctrl1);
  add_instance(box,box1);
}
delay.add_port(input,1,en);
delay.add_port(output,1,del);
cd.add_port_list(input,1,input_signals);
cd.add_port(output,1,do);
cd.add_port(output,1,d1);
ctrl.add_port_list(input,1,csl_list(del,state));
ctrl.add_port(input,1,valid);
box.add_port_list(input,1,csl_list(valid,d0,d1));
box.add_port_list(input,output_signals);
box.add_port(output,1,state);
/* if auto route is set then the signals will be automatically con-
nected by name and the next lines can be ignored */
scope block1 {
  delay1.connect(.en(en),.del(ctrl1.del));
  cd1.connect(.i0(i0),.i1(i1),.i2(i2),.i3(i3),.en(en),
.do(box1.do),.d1(box1.d1));
  ctrl1.connect(.valid(box1.valid),.state(box1.state));
  box1.connect(.sgn1(sgn1),.sgn2(sgn2),.sgn3(sgn3),
.data_out(data_out));
}
```

VERILOG CODE

```
// AV
module block1(en,i0,i1,i2,i3,sgn1,sgn2,sgn3,data_out);
    input en, i0,i1,i2,i3;
    output [1:0] sgn1, sgn2;
    output sgn3;
    output [3:0] data_out;
    reg [3:0] data_out;
    wire i0,i1,i2,i3,en,del,valid,d0,d1,sgn3;
    wire [1:0] sgn1, sgn2;
    wire [3:0] d_out;
    wire state;
    delay delay1(.en(en),.del(del));
    cd cd1(.en(en),.i0(i0),.i1(i1),.i2(i2),.i3(i3),.d0(d0),.d1(d1));
    ctrl ctrl1(.del(del),.state(state),.valid(valid));
```

```
    box
box1(.valid(valid),.d0(d0),.d1(d1),.state(state),.sgn1(sgn1),.sgn2(sgn
2),.sgn3(sgn3),.data_out(d_out));
endmodule
module delay(en,del);
    input en;
    output del;
endmodule
module cd(en,i0,i1,i2,i3,d0,d1);
    input en,i0,i1,i2,i3;
    output d0,d1;
endmodule
module ctrl(del,state,valid);
    input del,state;
    output valid;
endmodule
module box(valid,d0,d1,state,sgn1,sgn2,sgn3,data_out);
    input valid,d0,d1;
    output [1:0] sgn1,sgn2;
    output sgn3,state;
    output[3:0] data_out;
endmodule
```

unit_object_name.**add_signal_group(***signal_group_name***);**

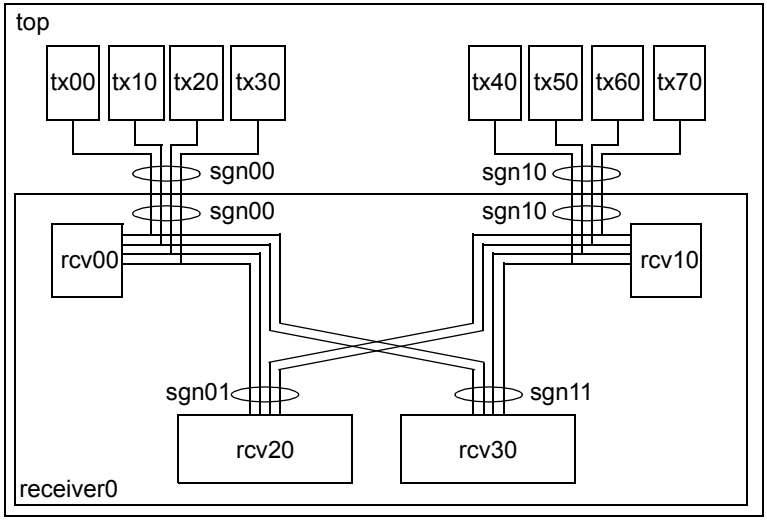
DESCRIPTION :

Add the signal group *signal_group_name* to the unit *unit_object_name.* If the parameter is an existing signal group from another scope, a copy of the respective signal group will be placed in the current scope. If the parameter is a non existing signal group, a new object of type signal group will be created in the current scope.

EXAMPLE :

This example shows how add_signal_group command can copy a signal group from another scope or create a new signal group with its default constructor if the name used as a parameter has not been previously declared as a signal group.

FIGURE 1.44 Intersected signal groups



CSL CODE

```
//AB
csl_unit
tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver,top;
top.add_signal_list(csl_list(s0,s1,s2,s3,s4,s5,s6,s7));
tx0.add_port(output,top.s0);
tx1.add_port(output,top.s1);
tx2.add_port(output,top.s2);
tx3.add_port(output,top.s3);
tx4.add_port(output,top.s4);
tx5.add_port(output,top.s5);
tx6.add_port(output,top.s6);
tx7.add_port(output,top.s7);
scope top {
```

```
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
}
rcv0.add_interface(input,top.sgn00);
rcv1.add_interface(input,top.sgn10);
scope receiver {
  //copy a signal group from another scope
  add_signal_group(top.sgn00);
  add_signal_group(top.sgn10);
  //the following signal groups are being created
  add_signal_group(sgn01);
  add_signal_group(sgn11);
  //for illustrative purposes
  sgn01.add_signal_list(csl_list(s0,s1,s2,s3));
  sgn11.add_signal_list(csl_list(s4,s5,s6,s7));
  //Question: once a group is being added to a scope
  //is it possible to reference the signals in the group
  //only by their name ?
  sgn11.add_signal_list(csl_list(sgn01.s2,sgn01.s3));
  sgn01.add_signal_list(csl_list(sgn11.s4,sgn11.s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top_design.sgn00);
  sgn10.connect(top_design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
}
/* we do the following to preserve naming and have autorouter
connect the signals/ports without explicitly connecting them */
rcv2.add_interface(input,receiver.sgn01);
rcv3.add_interface(input,receiver.sgn11);
scope receiver {
  add_instance(rcv2,rcv20);
```

```
    add_instance(rcv3,rcv30);
  }
  scope top {
    add_instance(receiver,receiver0);
    add_instance(tx0,tx00);
    add_instance(tx1,tx10);
    add_instance(tx2,tx20);
    add_instance(tx3,tx30);
    add_instance(tx4,tx40);
    add_instance(tx5,tx50);
    add_instance(tx6,tx60);
    add_instance(tx7,tx70);
  }
```

VERILOG CODE

```
  //AB
  `define WIDTH_S0 1
  `define WIDTH_S1 1
  `define WIDTH_S2 1
  `define WIDTH_S3 1
  `define WIDTH_S4 1
  `define WIDTH_S5 1
  `define WIDTH_S6 1
  `define WIDTH_S7 1

  module top();
    wire [`WIDTH_S0-1:0] s0;
    wire [`WIDTH_S1-1:0] s1;
    wire [`WIDTH_S2-1:0] s2;
    wire [`WIDTH_S3-1:0] s3;
    wire [`WIDTH_S4-1:0] s4;
    wire [`WIDTH_S5-1:0] s5;
    wire [`WIDTH_S6-1:0] s6;
    wire [`WIDTH_S7-1:0] s7;
    wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
    wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

    assign sgn00 = {s0,s1,s2,s3};
    assign sgn10 = {s4,s5,s6,s7};

    tx0 tx00(.s0(s0));
```

```
  tx1 tx10(.s1(s1));
  tx2 tx20(.s2(s2));
  tx3 tx30(.s3(s3));
  tx4 tx40(.s4(s4));
  tx5 tx50(.s5(s5));
  tx6 tx60(.s6(s6));
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule
```

```
module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(sgn01);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule
```

```
module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

nit_object_name.**add_unit_parameter**(*parameter_object_name*, *default_value*);

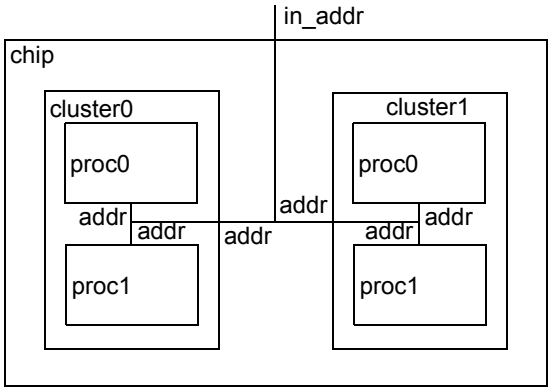
DESCRIPTION :

Create a module parameter in *unit_object_name* with the name *parameter_object_name* and the default value *default_value*.

EXAMPLE :

The unit hierarchy example shows how to add parameters to unit declarations.

FIGURE 1.45 Unit hierarchy



CSL CODE

```
//AB
csl_unit proc, cluster, chip;
proc.add_port(input,8,addr);
//add unit parameter with default value
proc.add_unit_parameter(PN,-1);
proc.add_unit_parameter(CLN,-1);
scope cluster {
  add_port(input,8,addr);
  //default value will be overriden in instances
  add_unit_parameter(CLN,-1);
  add_instance(proc,proc0);
  proc0.override_unit_parameter(PN,0);
  proc0.override_unit_parameter(CLN,CLN);
  addr.connect(proc0);
  add_instance(proc,proc1);
  proc0.override_unit_parameter(PN,1);
  proc0.override_unit_parameter(CLN,CLN);
  addr.connect(proc1);
}
```

```
scope chip {
  add_port(input,8,in_addr);
  add_instance(cluster, cluster0);
  cluster0.override_unit_parameter(CLN,0);
  in_addr.connect(cluster0);
  add_instance(cluster, cluster1);
  cluster1.override_unit_parameter(CLN,1);
  in_addr.connect(cluster1);
}
```

VERILOG CODE

```
//AB
module chip(in_addr);
  input [7:0] in_addr;
  cluster #(0) cluster0 (.addr(in_addr));
  cluster #(1) cluster1 (.addr(in_addr));
endmodule

module cluster(addr);
  parameter CLN=-1;
  input [7:0] addr;
  proc #(0,CLN) proc0 (.addr(addr));
  proc #(1,CLN) proc1 (.addr(addr));
endmodule

module proc(addr);
  parameter PN=-1, CLN=-1;
  input [7:0] addr;
endmodule
```

instance_object_name.**override_unit_parameter**(*parameter_object_name*, *parameter_value*);

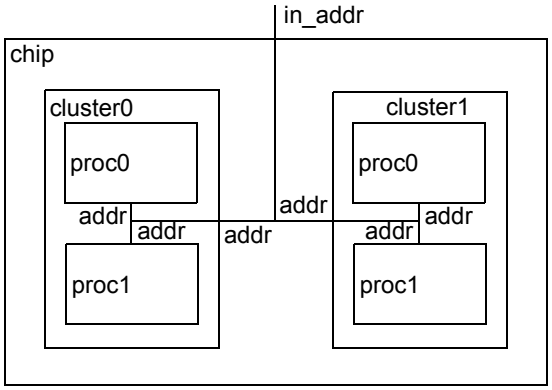
DESCRIPTION :

Set the unit parameter *parameter_object_name* equal to *parameter_value*

EXAMPLE :

The unit hierarchy example shows how to override the parameters of each instance.

FIGURE 1.46 Unit hierarchy



CSL CODE

```
//AB
csl_unit proc, cluster, chip;
proc.add_port(input,8,addr);
proc.add_unit_parameter(PN,-1);
proc.add_unit_parameter(CLN,-1);
scope cluster {
  add_port(input,8,addr);
  add_unit_parameter(CLN,-1);
  add_instance(proc,proc0);
  //parameter in instance overridden with a value
  proc0.override_unit_parameter(PN,0);
  //parameter overridden with a local parameter
  proc0.override_unit_parameter(CLN,CLN);
  addr.connect(proc0);
  add_instance(proc,proc1);
  proc0.override_unit_parameter(PN,1);
  proc0.override_unit_parameter(CLN,CLN);
  addr.connect(proc1);
}
scope chip {
```

```
    add_port(input,8,in_addr);
    add_instance(cluster, cluster0);
    cluster0.override_unit_parameter(CLN,0);
    in_addr.connect(cluster0);
    add_instance(cluster, cluster1);
    cluster1.override_unit_parameter(CLN,1);
    in_addr.connect(cluster1);
  }
```

VERILOG CODE

```
  module chip(in_addr);
    input [7:0] in_addr;
    cluster #(0) cluster0 (.addr(in_addr));
    cluster #(1) cluster1 (.addr(in_addr));
  endmodule

  module cluster(addr);
    parameter CLN=-1;
    input [7:0] addr;
    proc #(0,CLN) proc0 (.addr(addr));
    proc #(1,CLN) proc1 (.addr(addr));
  endmodule

  module proc(addr);
    parameter PN=-1, CLN=-1;
    input [7:0] addr;
  endmodule
```

unit_object_name.**set_unit_prefix**(*prefix_string[,prefix_specifier]*)
;

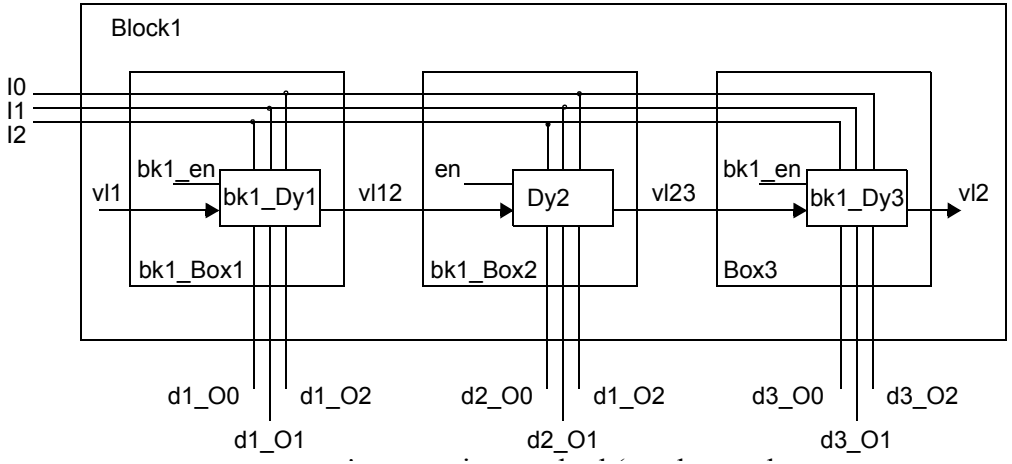
DESCRIPTION :

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 (add table with enum here). Default both specifiers are active.

EXAMPLE :

small description of the example.

FIGURE 1.47 An unit named *block1* with 3 instances. Then set a prefix for an unit.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
```

```
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
  add_instance(box,box3);
  scope box1 {
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
    add_signal(en);
    add_instance(dy,dy1);
    scope dy1 {
      add_port(input,en);
      add_port(input,vl1);
      add_port(output,vl12);
      add_port_list(output,block1.out1_signs);
    }
  }
  scope box2 {
    add_port(input,vl2);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
    add_signal(en);
    add_instance(dy,dy2);
    scope dy2 {
      add_port(input,en);
      add_port(input,vl12);
      add_port(output,vl23);
      add_port_list(output,block1.out2_signs);
    }
  }
  scope box3 {
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
    add_signal(en);
    add_instance(dy,dy3);
    scope dy3 {
      add_port(input,en);
      add_port(input,vl23);
      add_port(output,vl3);
```

```
      add_port_list(output,block1.out3_signs);
    }
  }
}
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
    input i0,i1,i2;
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
```

```
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

string *unit*_object_name.**get_unit_prefix**();

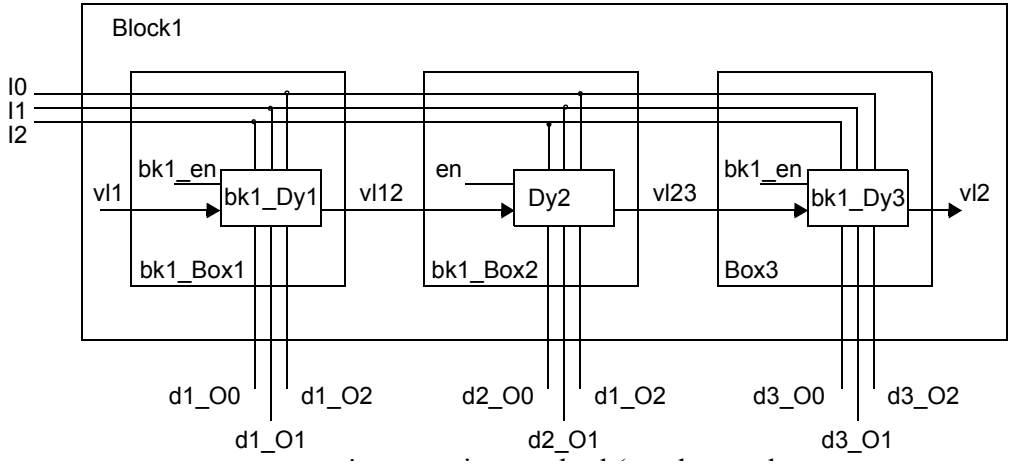
DESCRIPTION :

Returns the *unit_object_name*'s *string_prefix*.

EXAMPLE :

small description of the example.

FIGURE 1.48 An unit named *block1* with 3 instances.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
  add_instance(box,box3);
```

```
  scope box1 {
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
    add_signal(en);
    add_instance(dy,dy1);
    scope dy1 {
      add_port(input,en);
      add_port(input,vl1);
      add_port(output,vl12);
      add_port_list(output,block1.out1_signs);
    }
  }
  scope box2 {
    add_port(input,vl2);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
    add_signal(en);
    add_instance(dy,dy2);
    scope dy2 {
      add_port(input,en);
      add_port(input,vl12);
      add_port(output,vl23);
      add_port_list(output,block1.out2_signs);
    }
  }
  scope box3 {
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
    add_signal(en);
    add_instance(dy,dy3);
    scope dy3 {
      add_port(input,en);
      add_port(input,vl23);
      add_port(output,vl3);
      add_port_list(output,block1.out3_signs);
    }
  }
}
```

```
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
    input i0,i1,i2;
```

```
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
```

```
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

unit_object_name.**set_signal_prefix**(*prefix_string*);

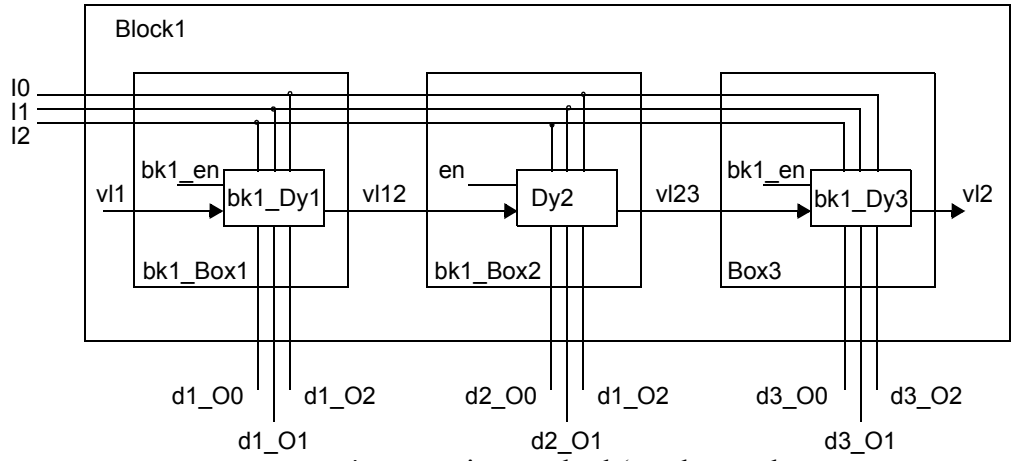
DESCRIPTION :

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.

EXAMPLE :

small description of the example.

FIGURE 1.49 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
```

```
add_instance(box,box3);
scope box1 {
  add_port(input,vl1);
  add_port(output,vl12);
  add_port_list(output,block1.out1_signs);
  add_signal(en);
  add_instance(dy,dy1);
  scope dy1 {
    add_port(input,en);
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
  }
}
scope box2 {
  add_port(input,vl2);
  add_port(output,vl23);
  add_port_list(output,block1.out2_signs);
  add_signal(en);
  add_instance(dy,dy2);
  scope dy2 {
    add_port(input,en);
    add_port(input,vl12);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
  }
}
scope box3 {
  add_port(input,vl23);
  add_port(output,vl3);
  add_port_list(output,block1.out3_signs);
  add_signal(en);
  add_instance(dy,dy3);
  scope dy3 {
    add_port(input,en);
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
  }
}
```

```
}
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
  block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
```

```
    input i0,i1,i2;
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
```

```
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

string *unit*_object_name.**get_signal_prefix**();

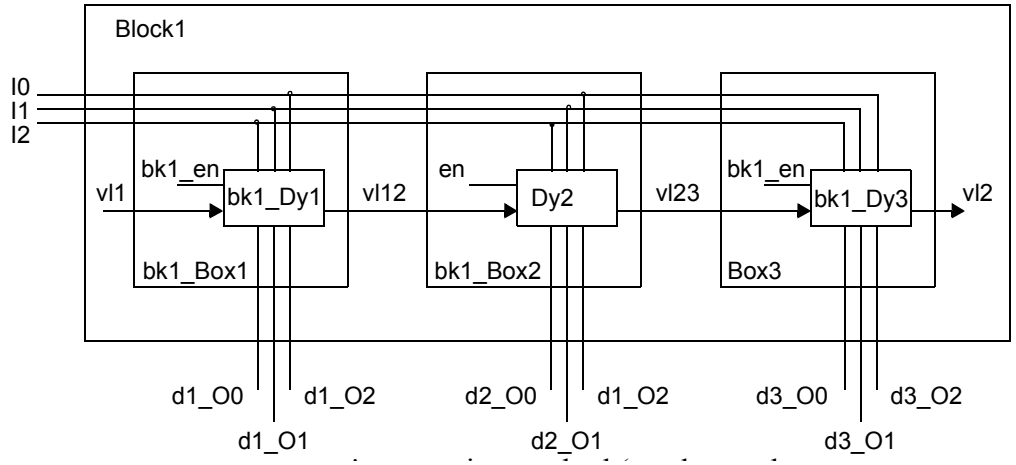
DESCRIPTION :

Returns the signal_prefix and the port_prefix previously set by the set_signal_prefix() command.

EXAMPLE :

small description of the example.

FIGURE 1.50 An unit named *block1* with 3 instances.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
```

```
add_instance(box,box3);
scope box1 {
  add_port(input,vl1);
  add_port(output,vl12);
  add_port_list(output,block1.out1_signs);
  add_signal(en);
  add_instance(dy,dy1);
  scope dy1 {
    add_port(input,en);
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
  }
}
scope box2 {
  add_port(input,vl2);
  add_port(output,vl23);
  add_port_list(output,block1.out2_signs);
  add_signal(en);
  add_instance(dy,dy2);
  scope dy2 {
    add_port(input,en);
    add_port(input,vl12);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
  }
}
scope box3 {
  add_port(input,vl23);
  add_port(output,vl3);
  add_port_list(output,block1.out3_signs);
  add_signal(en);
  add_instance(dy,dy3);
  scope dy3 {
    add_port(input,en);
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
  }
}
```

```
}
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
  block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
```

```
    input i0,i1,i2;
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
```

```
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

CSL CODE

```
csl_unit unit1, unit2;
unit1.add_signal_list(reg, 16, csl_list(sig1, sig2, sig3));
```

```
unit2.add_signal_list(wire, 32, csl_list(sig4, sig5, sig6));
unit1.set_signal_prefix(super_);
//sig1, sig2, sig3 are now super_sig1, super_sig2, super_sig3
unit2.set_signal_prefix(unit1.get_signal_prefix());
//sig4, sig5, sig6 are now super_sig4, super_sig5, super_sig6
```

VERILOG CODE

```
//no code exists. maybe the same as the one at set_unit_prefix()
```

unit_object_name.**set_signal_prefix_local(***prefix_string***);**

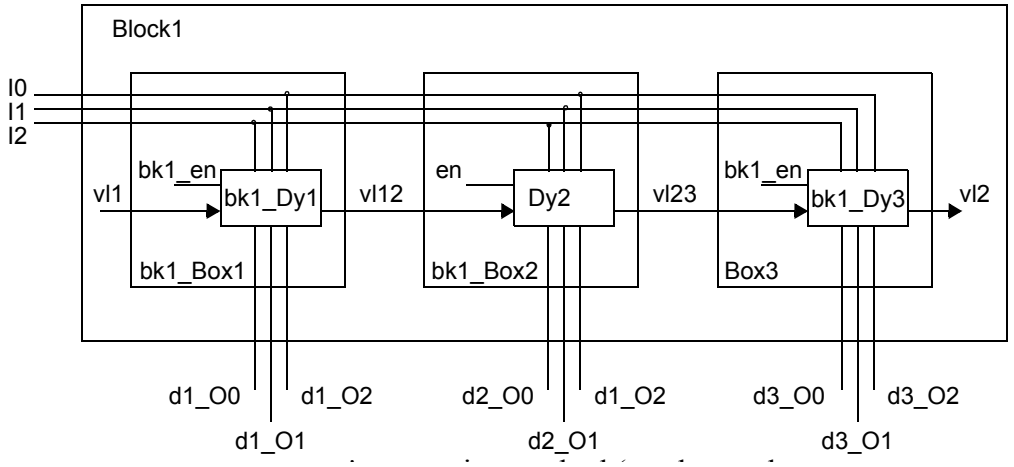
DESCRIPTION :

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.

EXAMPLE :

small description of the example.

FIGURE 1.51 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
```

```
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
  add_instance(box,box3);
  scope box1 {
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
    add_signal(en);
    add_instance(dy,dy1);
    scope dy1 {
      add_port(input,en);
      add_port(input,vl1);
      add_port(output,vl12);
      add_port_list(output,block1.out1_signs);
    }
  }
  scope box2 {
    add_port(input,vl2);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
    add_signal(en);
    add_instance(dy,dy2);
    scope dy2 {
      add_port(input,en);
      add_port(input,vl12);
      add_port(output,vl23);
      add_port_list(output,block1.out2_signs);
    }
  }
  scope box3 {
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
    add_signal(en);
    add_instance(dy,dy3);
    scope dy3 {
      add_port(input,en);
      add_port(input,vl23);
      add_port(output,vl3);
```

```
      add_port_list(output,block1.out3_signs);
    }
  }
}
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
  block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
    input i0,i1,i2;
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
```

```
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

CSL CODE

```
csl_unit newUnit;
newUnit.add_signal_list(wire, 32, csl_list(sig1, sig2, sig3));
newUnit.set_signal_prefix_local(new_unit);
/*now sig1, sig2 and sig3 becaome new_unit_sig1, new_unit_sig2 and
new_unit_sig3 */
```

VERILOG CODE

```
//verilog code goes here
```

string *unit_object_name*.**get_signal_prefix_local();**

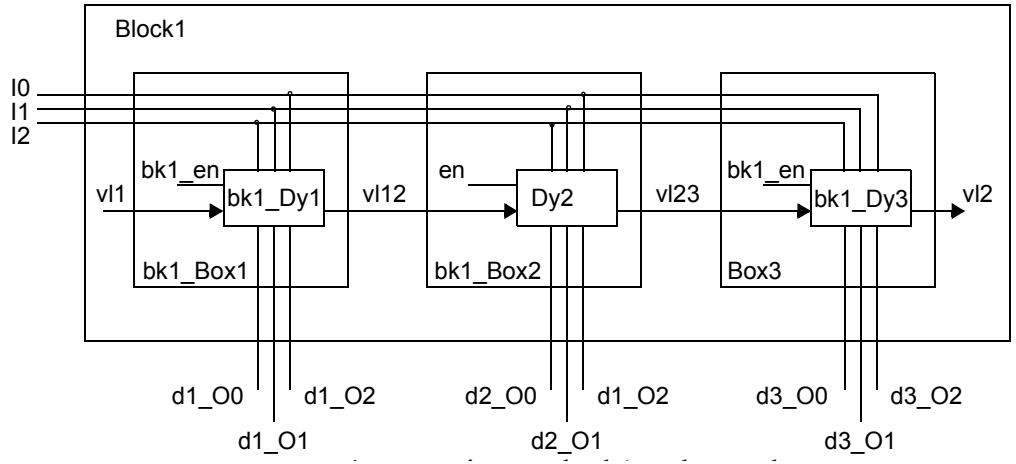
DESCRIPTION :

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).

EXAMPLE :

small description of the example.

FIGURE 1.52 An unit named *block1* with 3 instances.



CSL CODE

```
//AV
csl_unit block1,box,dy;
scope block1 {
  csl_list in_signs(i0,i1,i2);
  csl_list out1_signs(d1_o0,d1_o1,d1_o2);
  csl_list out2_signs(d2_o0,d2_o1,d2_o2);
  csl_list out3_signs(d3_o0,d3_o1,d3_o2);
  csl_list out_signs(out1_signs,out2_signs,out3_signs);
  add_port_list(input,1,in_signs);
  add_port_list(output,1,out_signs);
  csl_list valid (vl1,vl12,vl23,vl3);
  add_signal_list(valid);
}
box.add_port_list(input,1,block1.in_signs);
dy.add_port_list(input,1,block1.in_signs);
scope block1 {
  add_instance(box,box1);
  add_instance(box,box2);
```

```
  add_instance(box,box3);
  scope box1 {
    add_port(input,vl1);
    add_port(output,vl12);
    add_port_list(output,block1.out1_signs);
    add_signal(en);
    add_instance(dy,dy1);
    scope dy1 {
      add_port(input,en);
      add_port(input,vl1);
      add_port(output,vl12);
      add_port_list(output,block1.out1_signs);
    }
  }
  scope box2 {
    add_port(input,vl2);
    add_port(output,vl23);
    add_port_list(output,block1.out2_signs);
    add_signal(en);
    add_instance(dy,dy2);
    scope dy2 {
      add_port(input,en);
      add_port(input,vl12);
      add_port(output,vl23);
      add_port_list(output,block1.out2_signs);
    }
  }
  scope box3 {
    add_port(input,vl23);
    add_port(output,vl3);
    add_port_list(output,block1.out3_signs);
    add_signal(en);
    add_instance(dy,dy3);
    scope dy3 {
      add_port(input,en);
      add_port(input,vl23);
      add_port(output,vl3);
      add_port_list(output,block1.out3_signs);
    }
  }
```

```
}
block1.box1.set_unit_prefix(bx1);
/* all the instances, signals, ports and interface are
   now prefixed with the bk1 prefix only in verilog code
   en -> bk1_en, dy -> bk1_dy, vl1 -> bk1_vl1, i0 -> bk1_i0,
   i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0, d1_o1 -> bk1_d1_o1,
   d1_o2 -> bk1_d1_o2, box1 -> bk1_box1 */
  block1.box2.set_unit_prefix(block1.box1.get_unit_prefix(),IFC_ONLY);
/* we have only in verilog: box2 -> bk1_box2, vl12 -> bk1_vl12,
   vl23 -> bk1_vl23
   i0 -> bk1_i0, i1 -> bk1_i1, i2 -> bk1_i2, d1_o0 -> bk1_d1_o0,
   d1_o1 -> bk1_d1_o1, d1_o2 -> bk1_d1_o2 */

block1.box3.set_unit_prefix(block1.box1.get_unit_prefix(),LOCAL_ONLY);
// en -> bk1_en, dy3 -> bk1_dy3
  box1.vl12.set_signal_prefix(b1_2);
// from now the name for this signal and the will be b1_2_bk1_vl12
  box2.d2_o0.set_signal_prefix(bx2);
// the signal and the port will be: bx2_bk1_d2_o0
box2.d2_o2.set_signal_prefix(block1.box1.vl12.get_signal_prefix());
// bx2_bk1_d2_o2
  box3.vl23.set_signal_prefix_local(box2_3);
// only the signal will have the new name: box2_3_bk1_vl23
block1.box3.i0.set_signal_prefix_local(bx3);
block1.box3.i2.set_signal_prefix_local(block1.box3.i0.get_signal_prefi
x_local());
/* the name for ports, signals and interface from the unit is unchanged
in csl
if the auto route is not set the connection must to be declared */
scope block1 {
   box1.dy1.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl1(vl1),
.vl12(vl12),.d1_o0(d1_o0),.d1_o1(d1_o1),.d1_o2(d1_o2),.en(box1.en));
   box2.dy2.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl12(vl12),
.vl23(vl23),.d2_o0(d2_o0),.d2_o1(d2_o1),.d2_o2(d2_o2),.en(box2.en));
   box3.dy3.connect(.i0(i0),.i1(i1),.i2(i2),.,i3(i3),.,vl23(vl23),
.vl3(vl3),.d3_o0(d3_o0),.d3_o1(d3_o1),.d3_o2(d3_o2),.en(box1.en));
}
```

VERILOG CODE

```
//AV
module block1(i0,i1,i2,d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,
d3_o0,d3_o1,d3_o2);
```

```
    input i0,i1,i2;
    output d1_o0,d1_o1,d1_o2,d2_o0,d2_o1,d2_o2,d3_o0,d3_o1,d3_o2;
    wire [2:0] in_signs;
    wire [2:0] out1_signs,out2_signs,out3_signs;
    wire [8:0] out_signs;
    assign in_signs = {i0,i1,i2};
    assign out1_signs = {d1_o0,d1_o1,d1_o2};
    assign out2_signs = {d2_o0,d2_o1,d2_o2};
    assign out3_signs = {d3_o0,d3_o1,d3_o2};
    assign out_signs = {out1_signs,out2_signs,out3_signs};
    wire vl1,vl12,vl23,vl3;
    box01 box1(i0,i1,i2,d1_o0,d1_o1,d1_o2,vl1,vl12);
    box02 box2(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23);
    box03 box3(i0,i1,i2,d3_o0,d3_o1,d3_o2,vl23,vl3);
endmodule
//change the name of the interfaces signals and instances
module box01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
    wire bk1_en;
    dy01 bk1_dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
endmodule
//change only the name of the interfaces
module
box02(bk1_i0,bk1_i1,bk1_i2,bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,
bk1_vl12,bk1_vl23);
    input bk1_i0,bk1_i1,bk1_i2,bk1_vl12;
    output bx2_bk1_d2_o0,bk1_d2_o1,bx2_bk1_d2_o2,bk1_vl23;
    wire en;
    wire i0,i1,i2,d2_o0,d2_o1,d2_o2,vl12,vl23;
    assign i0 = bk1_i0;
    assign i1 = bk1_i1;
    assign i2 = bk1_i2;
    assign bx2_d2_o0 = bx2_bk1_d2_o0;
    assign d2_o1 = bk1_d2_o1;
    assign bx2_d2_o2 = bx2_bk1_d2_o2;
    assign vl12 = bk1_vl12;
    assign vl23 = bk1_vl23;
```

```
    dy02 dy2(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
endmodule
//change only the name of the signals and instances
module box03(i0,i1,i2,d2_o0,d2_o1,d2_o2,vl23,vl3);
    input i0,i1,i2,vl23;
    output d2_o0,d2_o1,d2_o2,vl3;
    wire bk1_en,box2_3_bk1_vl23,bk1_vl3;
    wire bx3_bk1_i0,bk1_i1,bx3_bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2;
    assign bx3_bk1_i0 = i0;
    assign bk1_i1 = i1;
    assign bx3_bk1_i2 = i2;
    assign bk1_d2_o0 = d2_o0;
    assign bk1_d2_o1 = d2_o1;
    assign bk1_d2_o2 = d2_o2;
    assign box2_3_bk1_vl23 = vl23;
    assign bk1_vl3 = vl3;
    dy03 bk1_dy3(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
endmodule
//the prefix affect this
module dy01(bk1_i0,bk1_i1,bk1_i2,bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,
bk1_en,bk1_vl1,b1_2_bk1_vl12);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl1;
    output bk1_d1_o0,bk1_d1_o1,bk1_d1_o2,b1_2_bk1_vl12;
endmodule
//the prefix affect this
module dy02(i0,i1,i2,bx2_d2_o0,d2_o1,bx2_d2_o2,en,vl12,vl23);
    input i0,i1,i2,en,vl12;
    output bx2_d2_o0,d2_o1,bx2_d2_o2,vl23;
endmodule
//the prefix affect this
module dy03(bk1_i0,bk1_i1,bk1_i2,bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,
bk1_en,bk1_vl23,bk1_vl3);
    input bk1_i0,bk1_i1,bk1_i2,bk1_en,bk1_vl23;
    output bk1_d2_o0,bk1_d2_o1,bk1_d2_o2,bk1_vl3;
endmodule
```

CSL CODE

```
csl_unit myUnit, yourUnit;
csl_signal_list sl(sig1, sig2, sig3), sl1(sig4, sig5, sig6);
myUnit.add_signal_list(wire, 32, sl);
```

```
myUnit.add_port(input, 16, myPort);
myUnit.set_signal_prefix_local(super_);
/* sig1, sig2 and sig3 are now super_sig1, super_sig2, super_sig3 but
myPort stays the same */
yourUnit.add_signal_list(wire, 32, sl1);
yourUnit.add_port(output, 16, yourPort);
yourUnit.set_signal_prefix(myUnit.get_signal_prefix_local());
/* sig4, sig5 and sig6 become super_sig4, super_sig5 and super_sig6 and
so does yourPort which becomes super_yourPort */
```

VERILOG CODE

```
//verilog code goes here
```

unit_object_name.**input_verilog_type**(*verilog_type*);

DESCRIPTION :

fix description

TABLE 1.5 Input and output types

output verilog type	description
v1995	Verilog IEEE Std 1364-1995
v2001	Verilog IEEE Std 1364-2001
v2005	Verilog IEEE Std 1364-2005
sysv	IEEE 1800 SystemVerilog

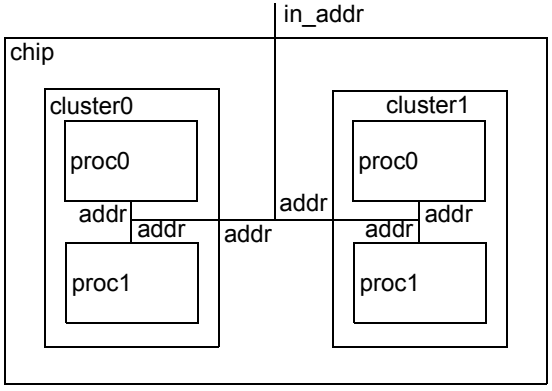
Where verilog_type is V1995, V2001, V2005, system_verilog. **output_verilog_type** will control the type of the generated verilog code . **input_verilog_type** specifies the verilog type which the design can recognize as input.(Add ex. which show the diferences between the 3 verilog types!!!!!)

Note: need to be able to have module declaration inside unit scope in order to fully support standard specific syntaxes

EXAMPLE :

In the following unit hierarchy example, the user can set different input verilog code types.

FIGURE 1.53 Unit hierarchy



CSL CODE

```
//AB
csl_unit proc, cluster, chip;
scope proc {
  add_port(input,8,addr);
  add_unit_parameter(PN,-1);
  add_unit_parameter(CLN,-1);
}
//set the input type for cluster unit as Verilog2001
cluster.input_verilog_type(v2001);
```

```
scope cluster {
  //Verilog2001 code
  parameter CLN=-1;
  input wire [7:0] addr;
  proc #(.CLN(CLN),.PN(0)) proc0 (.addr(addr));
  proc #(.PN(1),.CLN(CLN)) proc1 (.addr(addr));
}
//set the output type for cluster unit as Verilog2001
//this generates Verilog1995 compliant code
cluster.output_verilog_type(v1995);
//the input type for chip is set to Verilog1995
chip.input_verilog_type(v1995);
scope chip {
  //Verilog1995 code
  input [7:0] in_addr;
  wire [7:0] in_addr;
  cluster #(0) cluster0 (.addr(in_addr));
  cluster #(1) cluster1 (.addr(in_addr));
}
//the output type for chip will be Verilog2001
chip.output_verilog_type(v2001);
```

VERILOG CODE

```
//AB
//Verilog2001 output
module chip(in_addr);
  input wire [7:0] in_addr;

  cluster #(.CLN(0)) cluster0 (.addr(in_addr));
  cluster #(.CLN(1)) cluster1 (.addr(in_addr));
endmodule
//Verilog1995 output
module cluster(addr);
  parameter CLN=-1;
  input [7:0] addr;
  wire [7:0] addr;
  proc proc0 (.addr(addr));
  defparam proc0.CLN=CLN;
  defparam proc0.PN=0;
  proc proc1 (.addr(addr));
  defparam proc0.PN=1;
```

```
  defparam proc0.CLN=CLN;
endmodule

module proc(addr);
  parameter PN=-1, CLN=-1;
  input [7:0] addr;
endmodule
```

unit_object_name.**output_verilog_type**(*verilog_type*);

DESCRIPTION :

fix description

TABLE 1.6 Input and output types

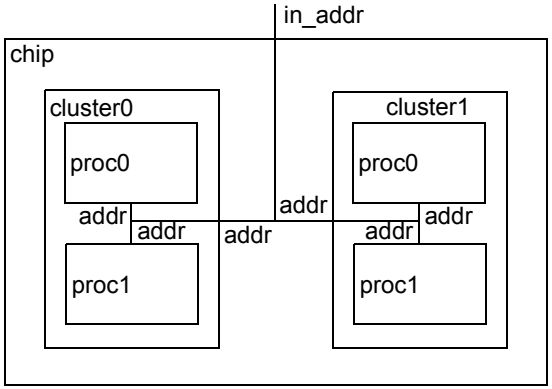
output verilog type	description
v1995	Verilog IEEE Std 1364-1995
v2001	Verilog IEEE Std 1364-2001
v2005	Verilog IEEE Std 1364-2005
sysv	IEEE 1800 SystemVerilog

where verilog_type is V1995, V2001, V2005, system_verilog. **output_verilog_type** will control the type of the generated verilog code . **input_verilog_type** specifies the verilog type which the design can recognize as input.(Add ex. which show the diferences between the 3 verilog types!!!!!)

EXAMPLE :

In the following unit hierarchy example the user can set different verilog output types.

FIGURE 1.54 Unit hierarchy



CSL CODE

```
//AB
csl_unit proc, cluster, chip;
scope proc {
  add_port(input,8,addr);
  add_unit_parameter(PN,-1);
  add_unit_parameter(CLN,-1);
}
//set the input type for cluster unit as Verilog2001
cluster.input_verilog_type(v2001);
scope cluster {
```

```
  //Verilog2001 code
  parameter CLN=-1;
  input wire [7:0] addr;
  proc #(.CLN(CLN),.PN(0)) proc0 (.addr(addr));
  proc #(.PN(1),.CLN(CLN)) proc1 (.addr(addr));
}
//set the output type for cluster unit as Verilog2001
//this generates Verilog1995 compliant code
cluster.output_verilog_type(v1995);
//the input type for chip is set to Verilog1995
chip.input_verilog_type(v1995);
scope chip {
  //Verilog1995 code
  input [7:0] in_addr;
  wire [7:0] in_addr;
  cluster #(0) cluster0 (.addr(in_addr));
  cluster #(1) cluster1 (.addr(in_addr));
}
//the output type for chip will be Verilog2001
chip.output_verilog_type(v2001);
```

VERILOG CODE

```
//AB
//Verilog2001 output
module chip(in_addr);
  input wire [7:0] in_addr;

  cluster #(.CLN(0)) cluster0 (.addr(in_addr));
  cluster #(.CLN(1)) cluster1 (.addr(in_addr));
endmodule
//Verilog1995 output
module cluster(addr);
  parameter CLN=-1;
  input [7:0] addr;
  wire [7:0] addr;
  proc proc0 (.addr(addr));
  defparam proc0.CLN=CLN;
  defparam proc0.PN=0;
  proc proc1 (.addr(addr));
  defparam proc0.PN=1;
  defparam proc0.CLN=CLN;
```

```
endmodule

module proc(addr);
  parameter PN=-1, CLN=-1;
  input [7:0] addr;
endmodule
```

```
unit_object_name.set_instance_alteration_bit(status);
```

DESCRIPTION :

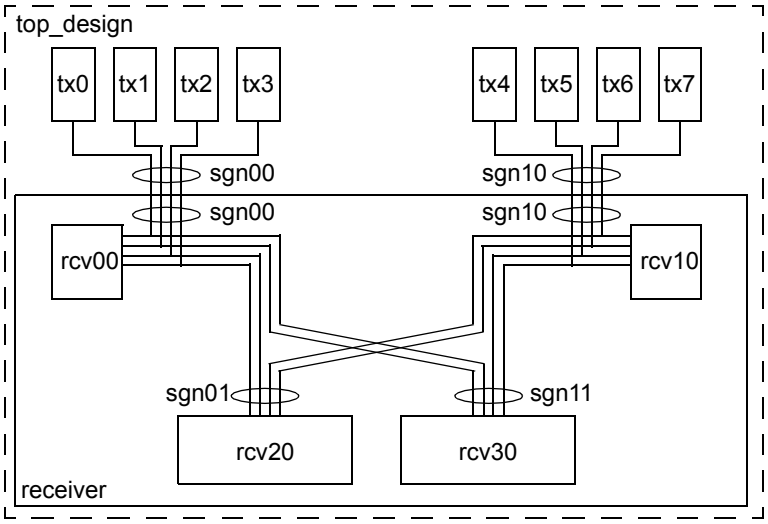
!!add enum table

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).

EXAMPLE :

In this example the copy constructor is used to create new signal groups from other signal groups and then to operate changes on these new groups.

FIGURE 1.55 Intersected signal groups



CSL CODE

```
update example to reflect the remove of csl_design
//AB
csl_design top_design;
//setting the instance alteration bit
top_design.set_instance_alteration_bit(on);
csl_unit tx0,tx1,tx2,tx3,tx4,tx5,tx6,tx7,rcv0,rcv1,rcv2,rcv3,receiver;
csl_signal s0,s1,s2,s3,s4,s5,s6,s7;
tx0.add_port(output,s0);//waiting for regex here
tx1.add_port(output,s1);
tx2.add_port(output,s2);
tx3.add_port(output,s3);
tx4.add_port(output,s4);
```

```
tx5.add_port(output,s5);
tx6.add_port(output,s6);
tx7.add_port(output,s7);
csl_signal_group sgn00(s0,s1,s2,s3);
csl_signal_group sgn10(s4,s5,s6,s7);
scope receiver {
  csl_signal_group sgn00(top_design.sgn00);
  csl_signal_group sgn10(top_design.sgn10);
  csl_signal_group sgn01(top_design.sgn00);
  csl_signal_group sgn11(top_design.sgn10);
  sgn11.add_signal_list(csl_list(s2,s3));
  sgn01.add_signal_list(csl_list(s4,s5));
  sgn11.remove_signal_list(csl_list(s4,s5));
  sgn01.remove_signal_list(csl_list(s2,s3));
  add_port_list(input,sgn00);
  add_port_list(input,sgn10);
  add_instance(rcv0,rcv00);
  add_instance(rcv1,rcv10);
  add_instance(rcv2,rcv20);
  add_instance(rcv3,rcv30);
/*the connect method will automatically create ports and
prefix them so that no name clashes would occur */
  sgn00.connect(top.design.sgn00);
  sgn10.connect(top.design.sgn10);
/*no need to connect sgn00 with sgn01 and sgn10 with sgn11
because these groups share the same scope and only specify
different signal organization */
  rcv00.add_port_list(input,sgn00);
/*this alters the instance and the original unit, and is
allowed because the instance alteration bit is set */
  rcv01.add_port_list(input,sgn10);
  rcv20.add_port_list(input,sgn01);
  rcv30.add_port_list(input,sgn11);
//leave autorouter do the connections
}
```

VERILOG CODE

```
//AB
`define WIDTH_S0 1
`define WIDTH_S1 1
`define WIDTH_S2 1
```

```
`define WIDTH_S3 1
`define WIDTH_S4 1
`define WIDTH_S5 1
`define WIDTH_S6 1
`define WIDTH_S7 1

module top_design();
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  wire [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;

  assign sgn00 = {s0,s1,s2,s3};
  assign sgn10 = {s4,s5,s6,s7};

  tx0 tx00(.s0(s0));
  tx1 tx10(.s1(s1));
  tx2 tx20(.s2(s2));
  tx3 tx30(.s3(s3));
  tx4 tx40(.s4(s4));
  tx5 tx50(.s5(s5));
  tx6 tx60(.s6(s6));
  tx7 tx70(.s7(s7));
  receiver receiver0(.sgn00(sgn00),.sgn10(sgn10));

endmodule

module receiver(sgn00,sgn10);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
  wire [`WIDTH_S0-1:0] s0;
  wire [`WIDTH_S1-1:0] s1;
  wire [`WIDTH_S2-1:0] s2;
  wire [`WIDTH_S3-1:0] s3;
```

```
  wire [`WIDTH_S4-1:0] s4;
  wire [`WIDTH_S5-1:0] s5;
  wire [`WIDTH_S6-1:0] s6;
  wire [`WIDTH_S7-1:0] s7;
  wire [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
  wire [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;

  assign {s0,s1,s2,s3} = sgn00;
  assign {s4,s5,s6,s7} = sgn10;
  assign sgn01 = {s0,s1,s4,s5};
  assign sgn11 = {s2,s3,s6,s7};

  rcv0 rcv00(.sgn00(sgn00));
  rcv1 rcv10(.sgn10(sgn10));
  rcv2 rcv20(.sgn01(sgn01));
  rcv3 rcv30(.sgn11(sgn11));

endmodule

module rcv0(sgn00);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S2+`WIDTH_S3)-1:0] sgn00;
endmodule

module rcv1(sgn10);
  input [(`WIDTH_S4+`WIDTH_S5+`WIDTH_S6+`WIDTH_S7)-1:0] sgn10;
endmodule

module rcv2(sgn01);
  input [(`WIDTH_S0+`WIDTH_S1+`WIDTH_S4+`WIDTH_S5)-1:0] sgn01;
endmodule

module rcv3(sgn11);
  input [(`WIDTH_S2+`WIDTH_S3+`WIDTH_S6+`WIDTH_S7)-1:0] sgn11;
endmodule

module tx0(s0);
  output [`WIDTH_S0-1:0] s0;
endmodule

module tx1(s1);
```

```
  output [`WIDTH_S1-1:0] s1;
endmodule

module tx2(s2);
  output [`WIDTH_S2-1:0] s2;
endmodule

module tx3(s3);
  output [`WIDTH_S3-1:0] s3;
endmodule

module tx4(s4);
  output [`WIDTH_S4-1:0] s4;
endmodule

module tx5(s5);
  output [`WIDTH_S5-1:0] s5;
endmodule

module tx6(s6);
  output [`WIDTH_S6-1:0] s6;
endmodule

module tx7(s7);
  output [`WIDTH_S7-1:0] s7;
endmodule
```

```
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 1.7 contains the list of connection objects and the allowed connections between connection object types.

TABLE 1.7 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	-	-	-	-	-	-	-	-

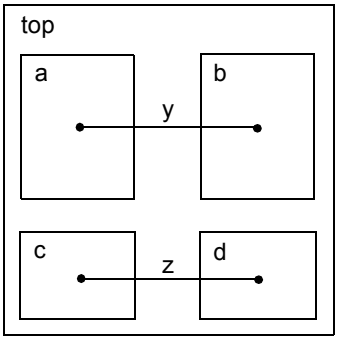
x? = to be discussed
Connection needs support for signal subranges and expressions
The following may not be needed anymore
Coonect can be used to ‘link’ signals, ports, groups of signals, or unit interfaces (when connect is apllied to unit names it’s still the interfaces of those units that get connected)

```
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);

EXAMPLE :

FIGURE 1.56



//small description of the example

CSL CODE

```
csl_unit top;
csl_unit a, b, c, d;
top.add_instance(a, a1);
top.add_instance(b, b1);
b1.add_signal(y);
scope a1 {
  add_signal(y);
  y.connect(b1.y);
}
csl_unit c, d;
top.add_instance(c, c1);
top.add_instance(d, d1);
c1.add_signal(z);
d1.add_signal(t);
c1.z.connect(d1.t);
```

VERILOG CODE

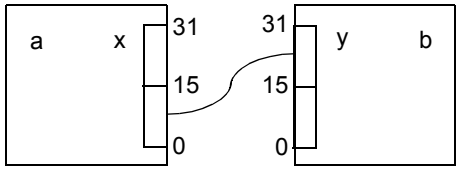
```
//AV
module top;
    wire y,z;
    ab a1(y);
    ab b1(y);
    c c1(z);
    d d1(z);
endmodule
module ab(y);
    inout y;
endmodule
```

```
module c(z);
    inout z;
endmodule
module d(z);
    inout z;
    wire t;
    assign t = z;
endmodule
```

.

EXAMPLE :

FIGURE 1.57



would the above figure be useful for shrinked examples ? if not, remove it

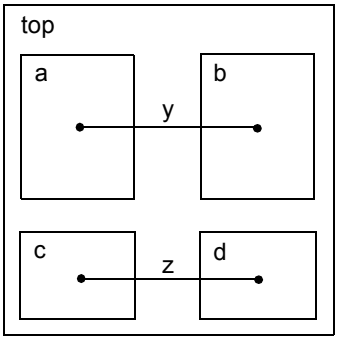
connect(*connection_object_name0*,*connection_object_name1*);

DESCRIPTION :

Global connect function.

EXAMPLE :

FIGURE 1.58



//small description of the example

CSL CODE

```
csl_unit top;
csl_unit a, b, c, d;
top.add_instance(a, a1);
top.add_instance(b, b1);
a1.add_signal(y);
scope b1 {
  add_signal(y);
  connect(y,a1.y);
}
csl_unit c, d;
top.add_instance(c, c1);
top.add_instance(d, d1);
c1.add_signal(z);
d1.add_signal(t);
connect(c1.z,d1.t);
```

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
```

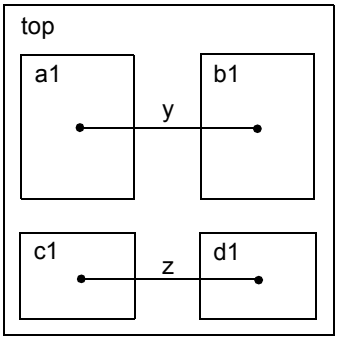
connection_object_name.**connect**({.*formal_connection_object_name*(*actual_connection_object_name*)}+);

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.

EXAMPLE :

FIGURE 1.59



//small description of the example

fix code - update according to latest commands

CSL CODE

```
csl_unit top;
csl_unit a, b, c, d;
top.add_instance(a, a1);
top.add_instance(b, b1);
b1.add_signal(y);
scope a1 {
  add_signal(y);
  connect(.y(b1.y));
}
csl_unit c, d;
top.add_instance(c, c1);
top.add_instance(d, d1);
c1.add_signal(z);
d1.add_signal(t);
c1.connect(.z(d1.t));
```

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
```

signal_object_name.**merge(***merg_op*,*list_of_signals***);**

DESCRIPTION :

Performs the operation *op* on the signals in *list_of_signals* and assigns the output of the merge operation to *signal_object_name.*

EXAMPLE :

//small description of the example

CSL CODE

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

VERILOG CODE

```
//Verilog code goes here
```

signal_object_name.**merge**(*merg_op*, *signal_list_object_name*);

DESCRIPTION :

Performs the operation *merg_op* on the signals in *list_of_signals* and assigns the output of the merge operation to *signal_object_name.*

fix table and update example code

TABLE 1.8

	effect

EXAMPLE :

//small description of the example

CSL CODE

```
csl_signal sig;
sig.merge(merg_op, csl_list(s1, s2, s3));

csl_signal_list sl(a1, a2);
sig.merge(merg_op, sl);

csl_signal_list sl(s1, s2);
sl.add_signal_item(s3);
sig.merge(merg_op, sl);
```

VERILOG CODE

//Verilog code goes here

1.1 CSL Examples

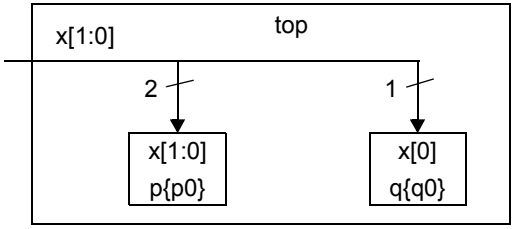
1.1.0.1 Input bus example

The following code declares 3 units and creates the design hierarchy. A signal X is added as an input to each of the three modules. p and q are modules. p0 and q0 are instances inside of top.

1.1.0.1.1 CSL code

```
csl_unit top, p, q;
csl_list l = {p, q};
top.add_instance(l);
top.add_port(input,2,x);
p.add_port(input,2,x);
q.add_port(input,1,x);
```

FIGURE 1.60 Assigning a bus to 2 or more instances or using bit ranges to extract fields from a vector



1.1.0.1.2 Verilog code

```
module top (x);
  input [1:0] x;
  p p0(.x(x[1:0]));
  q q0(.x(x[0]));
endmodule

module p(x);
  input [1:0] x;
endmodule

module q(x);
  input x;
```

```
endmodule
```

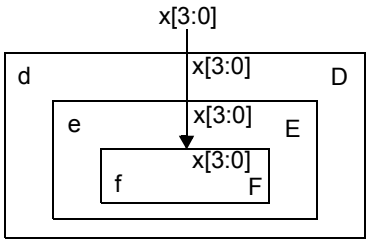
1.1.0.2 Input block diagram

1.1.0.2.1 CSL code

in this The three units d, e, and f are declared. Then the instance hierarchy is created and the input signals are added to the instances.

```
csl_unit d,e,f;
d.add_instance(e);
e.add_instance(f);
d.add_port(input,4,x);
e.add_port(input,4,x);
f.add_port(input,4,x);
d.x.connect(e.x);
e.x.connect(f.x);
```

FIGURE 1.61 Input block diagram



1.1.0.2.2 Verilog code

```
module d(x);
 input [3:0] x;
 e E(.x(x));
endmodule

module  e(x);
 input [3:0] x;
 f F(.x(x));
endmodule

module f(x);
 input [3:0] x;
```

```
endmodule
```

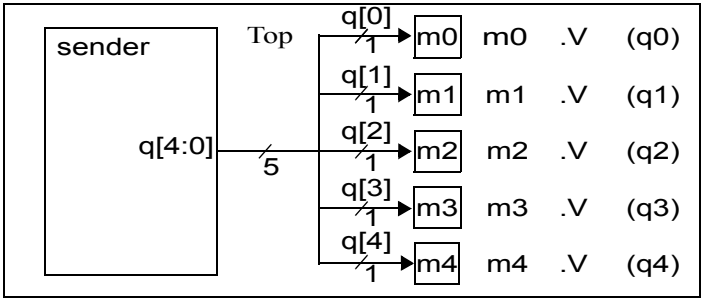
1.1.0.3 Concatenation example

In this example we map a vector to a concatenation of 5 1-bit signals.

1.1.0.3.1 CSL code

```
//this example needs to be debated - skip this
csl_unit top, sender, m0, m1, m2, m3, m4;
top.add_instance(sender, m0, m1, m2, m3, m4);
assign  {q4, q3, q2, q1, q0} = q;
sender.output(q[4:0]);
m0.input(q[0]);
m1.input(q[1]);
m2.input(q[2]);
m3.input(q[3]);
m4 input(q[4]);

or

csl_unit top, sender, m0, m1, m2, m3, m4;
top.add_instance(sender, m0, m1, m2, m3, m4);
csl_signal q;
assign  {q[4], q[3], q[2], q[1], q[0]} = q;
sender.output(q[4:0]);
m0.input(q[0]);
m1.input(q[1]);
m2.input(q[2]);
m3.input(q[3]);
m4 input(q[4]);
```

FIGURE 1.62 Split the input



1.1.0.3.2 Verilog code

```
module top;
  sender sender0(.q(q));
  wire [4:0] q;
   m m0(.q(q[0]));
   m m1(.q(q[1]));
   m m2(.q(q[2]));
   m m3(.q(q[3]));
   m m4(.q(q[4]));
endmodule

module sender(q);
  output [4:0] q;
endmodule

module m(q);
  input q;
endmodule
```

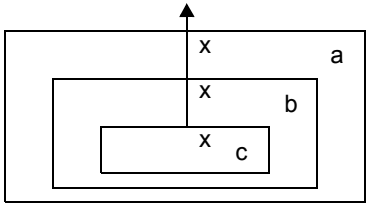
1.1.0.4 Output examples

1.1.0.4.1 CSL code

```
csl_unit a,b,c;
a.add_instance(b,b0);
b.add_instance(c,c0);
a.add_port(output,x);
b.add_port(output,x);
```

```
   c.add_port(output,x);  //or use signal router to determine path -
name inference
```

FIGURE 1.63 Output block diagram



1.1.0.4.2 Verilog code

```
module a(x);
  output x;
  b b0 (.x(x));
endmodule

module b(x);
  output x;
  c c0(.x(x));
endmodule

module c(x);
  output x;
endmodule
```

1.1.0.5 Output to input

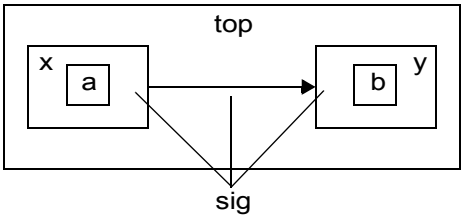
The signal *sig* is not declared in the *top*, *x* or *y* units. The auto router adds *sig* to the

1.1.0.5.1 CSL code

```
csl_unit top, int, a, b, x ,y;
a.add_port(output,sig);
b.add_port(input,sig);
top.add_instance(x,x0);
x.add_instance(a,a0);
top.add_instance(y);
y.add_instance(b);
```

```
   x.a.sig.connect(y.b.sig);
   //should name inference work over scopes
   //eg. x.sig.connect(y.sig); <=> auto connect by name inference
   //x.a.sig.connect(y.b.sig); <=> does auto connect work over scope
   //names? so that the connect() method may not be used and would this
add confusion if there are many scopes ?
```

FIGURE 1.64 Connect output to input



1.1.0.5.2 Verilog code

```
module Top;
 wire x;
 x x0(.sig(sig));
 y y0(.sig(sig));
endmodule

module x(sig);
output sig;
 a a0(.sig(sig));
endmodule

module y(sig);
input sig;
 b b0(.sig(sig));
endmodule

module a(sig);
 output sig;
endmodule

module b(sig);
 input sig;
endmodule
```

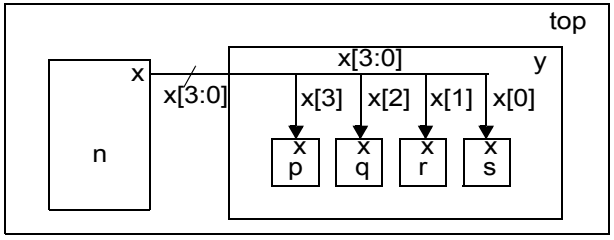
1.1.0.6 Hierarchy

The example shows how to split a bit vector into single bits and connect each bit to a different unit.

1.1.0.6.1 CSL code

```
csl_unit top, n, y, p, q, r, s;
top.add_instance(n, n0);
top.add_instance(y, y0);
y.add_instance(p, p0);
y.add_instance(q, q0);
y.add_instance(r, r0);
y.add_instance(s, s0);
n.add_port(output,4,x);
y.add_port(input,4,x);

//what is being done here ?
//p.input(x[3]);
//q.input(x[2]);
//connect_signal r.input(x[1]);
//connect_signal s.input(x[0]);

p.add_port(input,x);
q.add_port(input,x);
r.add_port(input,x);
s.add_port(input,x);
p.x.connect(top.x[3]);
q.x.connect(top.x[2]);
r.x.connect(top.x[1]);
s.x.connect(top.x[0]);
```

FIGURE 1.65 Hierarchy.



1.1.0.6.2 Verilog code

```
module top;
wire [3:0] x;
  n n0 (.x(x));
  y y0 (.x(x));
endmodule

module y(x);
  input [3:0] x;
  p p0 (.x(x[3]));
  q q0 (.x(x[2]));
  r r0 (.x(x[1]));
  s s0 (.x(x[0]));
endmodule

module n(x);
  output [3:0] x;
endmodule

module p(x);
  input x;
endmodule

module q(x);
  input x;
endmodule

module r(x);
  input x;
endmodule

module s(x);
  input x;
endmodule
```

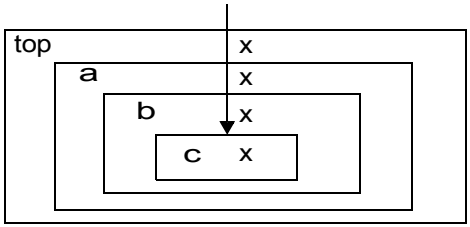
1.1.0.7 Create module inputs

Add an input to the top module and bottom module. The auto router will add the signals to the intermediate units' port lists.

1.1.0.7.1 CSL code

```
csl_unit top,a,b,c;
top.add_instance(a,a0);
a.add_instance(b,b0);
b.add_instance(c,c0);
top.add_port(input,x);
c0.add_port(input,x);   //use signal router to connect signals
```

FIGURE 1.66 Module hierarchy



This will create three instances: modules a, b and c and each will have an input called x.

1.1.0.7.2 Verilog code

```
module top (x);
  input x;
  a a0(.x(x));
endmodule

module a (x);
  input x;
  b b0(.x(x));
endmodule

module b(x);
  input x;
  c c0(.x(x));
endmodule

module c(x);
  input x;
endmodule
```

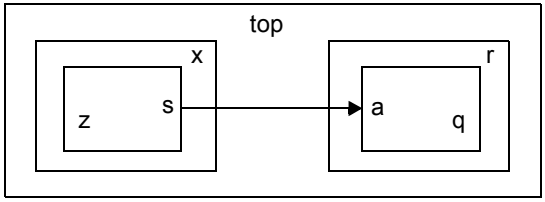
1.1.1 Interconnect

In the following example we create an output called s in module z and we create an input in module q called a. We then use HID's to connect the input and the output. The signal router will connect the two HID's by adding the ports to the upper level modules and mapping the two different signal names at the highest level.

1.1.1.1 CSL code

```
csl_unit x, r, z, q, top;
top.add_instance(x,x0);
top.add_instance(r,r0);
x.add_instance(z,z0);
r.add_instance(q,q0);
z.add_port(output,s);
q.add_port(input,a);
x.z.s.connect(r.q.a);
```

FIGURE 1.67 Signal route



1.1.1.2 Verilog code

```
module top;
 wire a ,s ;
 assign a = s;
 x x0(.a(a));
 r r0(.s(s));
endmodule

module r(a);
 input a;
 q q0(.a(a));
endmodule

module x(s);
```

```
 output s;
 z z0(.s(s));
endmodule

module z(s);
 output s;
endmodule

module q(a);
 input a;
endmodule
```

1.1.2 Formal to actual mapping

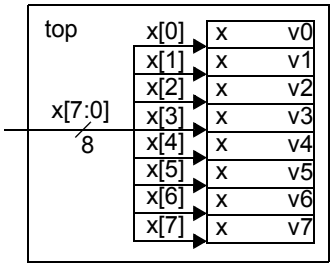
1.1.2.1 Multiple instances mapping formal to actual using bit vectors and a for loop

If the actual is one expression then the expression is entered in the actual name box. The module x is instantiated 8 times in the following example.

1.1.2.1.1 CSL code

```
csl_unit top,v;
top.add_port(input,8,x);
csl_for(int i = 0; i <= 7; i++) {
  //concatenate "v" with the current value of index
  //is this implemented ?
  string w = "v" + i;
  top.add_instance(v,w);
  top.x[i].connect(w.x);
  //now csl just like C++ top.x[i].connect(w.x);
}
```

FIGURE 1.68 Implicit naming of formal to actuals using bit vectors



1.1.2.1.2 Verilog code

```
module top(x);
  input [7:0] x;
  v v0(.x(x[0]));
  v v1(.x(x[1]));
  v v2(.x(x[2]));
  v v3(.x(x[3]));
  v v4(.x(x[4]));
  v v5(.x(x[5]));
  v v6(.x(x[6]));
  v v7(.x(x[7]));
endmodule

module v(x);
  input x;
endmodule
```

1.1.2.2 Formal and Actual Names and Formal to Actual Mapping Examples

The formal name will always be a name of a port in the instance.

```
port_signal = namespace_direction_type_signalname_range
port_signal = namespace_direction_signalname_range
 defaults to wire

signal = namespace_type_signalname_range
formal_to_actual_mapping(formal, actual);
//adds the formal to the current namespaces port list and adds the
formal to actual name to all instantiations of the module.

namespace.formal_to_actual_mapping(formal, actual);
```

```
   //adds the formal to the namespace port list and adds the formal to
actual name to the instantiation of the module. connection serdes
```

Formal port is multi-bit and there are individual CSL connections to each port bit. Note that the CSL supports splitting the formal name Into individual bits. The CDOM will "gather" the different connections to the formal bits and create a concatenation in the actual name. Note that the port can be an input or an output.
If the target module's verilog code looks like the following:

```
module foo (output [3:0] x);
.....
endmodule
```

And the module foo is instantiated In the module bar:

```
module bar:
wire a,b,c,d;
 //overides the signal naming convention and prefix all signals in the
connection
 //declaration with the prefix specified in the signal_prefix_override
statement


signal_prefix_override:
"signal_prefix_override" "("prefix_override_object_name")"";"

   csl_signal bus_object_name = split(Iist_of_signals_ranges);
   // creates a new bus with the width of list_of_signals_ranges and
assigns he new bus to the list_of_signals_ranges
   // the width of the bus can be discovered with the int i = width
(list_of_signals_ranges); operation
```

Example: create new signals with a width of 1-bit and assign each new signal to a single bit from a bus

```
   for (í = 0 ; í < width (list_of_signals_ranges); í++) {
     csl_signal signal_object_name_${í} = split( bus_object_name );
     // creates a set of new signals with the name csl_signal
signal_object_name_${i} and assigns each new name to
signal_object_name[${i}]
}
```

Example: create new signals with a width of width::*number_expression* and assign each new signal to multiple bits from a bus

```
   for (i = 0 ; í < width (list_of_signals_ranges); i++) {
     csl_signal signal_object_name_${i} = split(bus_object_name,
width::number_expression);
     // creates a set of new signals with the name csl_signal
signal_object_name_${í} and assigns each new name to
signal_object_name[${i}+number_expression-1 : ${í} ]
   }
```

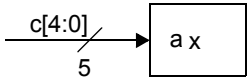
here begins the assignment part - not modified

1.1.2.3 Assigning constants to signals

1.1.2.3.1 CSL code

```
csl_unit x;
x.input(a = 5'H0A);
```

FIGURE 1.69 5'H0A



1.1.2.3.2 Verilog code

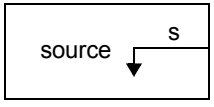
```
module x(c);
  input [4:0] c;
  wire [4:0] a;
  assign a = c;
endmodule
```

1.1.2.4 Source example

1.1.2.4.1 CSL code

```
csl_unit x;
x.output(s = 0);
```

FIGURE 1.70 Source



1.1.2.4.2 Verilog code

```
module x(s);
  parameter WIDTH =1 ;
  output [WIDTH-1:0] s;
  wire [WIDTH-1:0] s;
  assign s = {WIDTH {1'b0}};
endmodule
```

1.1.2.5 Output tied to GND and output tied to VDD example

1.1.2.5.1 CSL code

```
csl_unit a, b;
a.input(x = 0);
b.input(y = 1);
```

FIGURE 1.71

input tied to ground output tied to Vdd

1.1.2.5.2 Verilog code

```
module a(in);
  input in;
  wire x;
  assign x = in;
endmodule

module b(in);
  input in;
  wire y;
  assign y = in;
endmodule
```

here ends the assignment part - resume

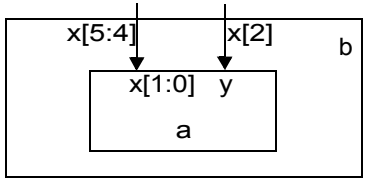
1.1.2.6 Bitrange Change

x[3] is not assigned. The default driven value is 0.

1.1.2.6.1 CSL code

```
csl_unit a, b;
csl_signal x(5,2);
b.add_instance(a,a0);

// maybe we will need this some time later
//b.add_port(input,[5:4],x); //or maybe b.add_port(input,(5,4),x);
//b.add_port(inputx[2]);

b.add_port(input,x);
a.add_port(input,2,x);
a.add_port(input,1,y)
a.x.connect(x[5:4]);
a.y.connect(x[2]);
```

FIGURE 1.72 Missing bits in range



1.1.2.6.2 Verilog code

```
module b(x);
  input [5:2] x;  //bit 3 is not used
  a a0(.x(x[5:4]), .y(x[2]));
endmodule

module a(x,y);
  input [1:0] x;
  input y;
endmodule
```

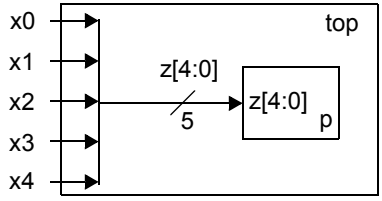
1.1.3 Regular expressions

The inputs x0, ... , x4 are concatenated together and assigned to port z in module p.

1.1.3.1 CSL code

```
csl_unit top, p;
top.add_instance(p);
top.add_port(input,x["0-4]"); //creates x1, x2, x3, x4
p.add_port(input,5,z);
top.{x[0-4]}.connect(p.z);

//old version - preserved for reference
//csl_connect p.zmodule = top.{x[0-5]};   // creates a 5-bit signal
//called z in unit p
//csl_connect y[0-4].x = top.x[0-4];
//the concat will be expanded first {x[0-4} = {x0, x1, x2, x3, x4}
//and then the LHS will be
```

FIGURE 1.73 Formal to actual mapping input concatenation



NOTE: Regular expressions are expanded into this: p.z.input(x[4-0])

1.1.3.2 Verilog code

```
module top(x4, x3, x2, x1, x0);
  input x4, x3, x2, x1, x0;
  wire [4:0] z;
  assign z = {x4, x3, x2, x1, x0};
  p p0(.z(z[4:0]));
endmodule

module p(z);
  input [4:0] z;
endmodule
```

The following formal to actual mappings are supported:

- constant
- concatenation
- renaming

in either formal or actual name position (a[3:0],b[2:0],c[3:0] <= foo) (bar <= {x,y,z})
signals may ne declared and then used later

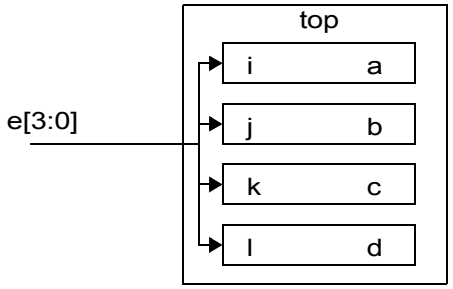
1.1.4 Split a vector into bits and assigning different bit units example

The input vector e[3:0] is split into 4 different signals.

1.1.4.1 CSL code

```
csl_unit top, a, b, c, d;
top.add_instance (a, a0);
top.add_instance (b, b0);
top.add_instance (c, c0);
top.add_instance (d, d0);
top.add_port(input,4,e);
a.i.connect(top.e[3]);
b.j.connect(top.e[2]);
c.k.connect(top.e[1]);
d.l.connect(top.e[0]);
//is this going to work ? Will the i,j,k,l ports be build automati-
cally by the cslc?

// or
top.e.connect({ a.i, b.j, c.k, d.l });
```

FIGURE 1.74 Bit blast



1.1.4.2 Verilog code

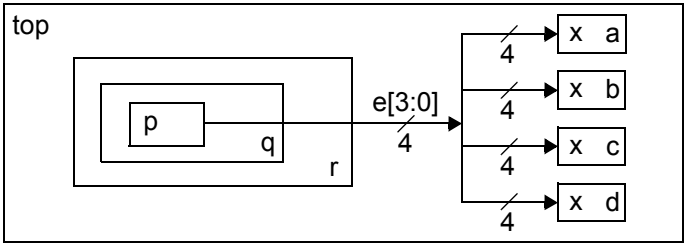
```
module top(e);
  input [3:0] e;
  a a0(.i(e[0]));
  b b0(.j(e[1]));
  c c0(.k(e[2]));
  d d0(.l(e[3]));
endmodule

module a(i);
  input i;
endmodule

module b(j);
  input j;
endmodule

module c(k);
  input k;
endmodule

module d(l);
  input ;
endmodule
```

1.1.5 Assigning an entire vector to a 4 bus instance

1.1.5.1 CSL code

```
csl_unit top, p, q, r, a, b, c, d;
top.add_instance (a, a0);
top.add_instance (b, b0);
top.add_instance (c, c0);
top.add_instance (d, d0);
top.add_instance (r, r0);
r.add_instance(q, q0);
q.add_instance(p, p0);
q.add_port(output,4,e);
p.add_port(output,4,e);
x.a.connect(r.e[3:0]);  infer that e is an output of r because r's
                  child q has an output e
x.b.connect(r.e[3:0]);
x.c.connect(r.e[3:0]);
x.d.connect(r.e[3:0]);
```

FIGURE 1.75 Assign an entire vector to a bus instance



NOTE: instances {a, b, c, d} do not have to be instantiated in module top but they can be instantiated in a lower level . The signal router will connect signals to the instances.

1.1.5.2 Verilog code

```
module top();
  wire [3:0] e;
  a a0(.x(e));
```

```
  b b0(.x(e));
  c c0(.x(e));
  d d0(.x(e));
  r r0(.e(e));
endmodule

module r(e);
  output [3:0] e;
  q q0(.e(e));
endmodule

module p(e);
  output [3:0] e;
endmodule

module a(x);
  input [3:0] x;
endmodule

module b (x);
  input [3:0] x;
endmodule

module c(x);
  input x;
endmodule

module d(x);
  input [3:0] x;
endmodule
```

1.1.6 Name change

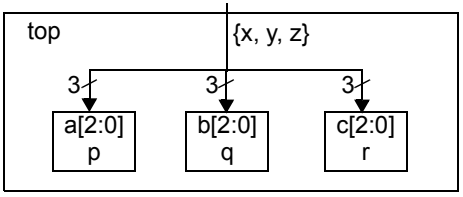
1.1.7 Concatenation

1.1.7.1 Concatenation driving inputs example

1.1.7.1.1 CSL code

```
csl_unit top, p, q, r;
top.add_instance(p, p0);
top.add_instance(q, q0);
top.add_instance(r, r0);
//top, x, y, z;
p.add_port(input,3,a);
q.add_port(input,3,b);
r.add_port(input,3,c);
a = top.input({x, y, z});  fix per the diagram
b = top.input({x, y, z});
c = top.input({x, y, z});
```

FIGURE 1.76 Concatenation



1.1.7.1.2 Verilog code

```
module top (x, y, z);
  input x, y, z;
  p p0(a({x,y,z}));
  q q0(b({x,y,z}));
  r  r0(c({x,y,z}));
endmodule
```

```
module p (a);
  input [2:0] a;
endmodule

module q(b);
  input [2:0] b;
endmodule

module r(c);
  input [2:0] c;
endmodule
```

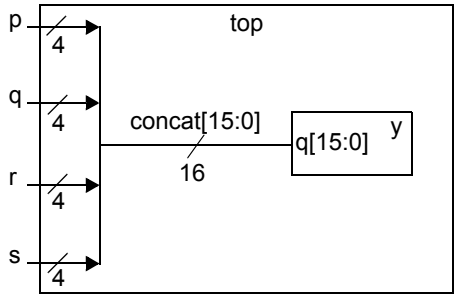
1.1.7.2 Concatenation of inputs example

The figure shows the concatenation of signals. The concat is assigned to one input. The input is a 16 bit bus.

1.1.7.2.1 CSL code

```
csl_unit top, y;
top.add_instance(y, y0);
top.add_port(input,4,p);
top.add_port(input,4,q);
top.add_port(input,4,r);
top.add_port(input,4,s);
y.add_port(input,16,q);
y.q.connect(top.{p, q, r, s});
 Note: module y has an input q which is 16 bits wide.
```

FIGURE 1.77 Concatenation signals



1.1.7.2.2 Verilog code

```
module top(p, q, r, s);
  input [3:0] p, q, r, s;
  wire [15:0] concat = {p, q, r, s};
  y y0(.q(concat));
endmodule

module y(q);
  input [15:0] q;
endmodule
```

1.1.7.3 Concatenation of outputs from different modules

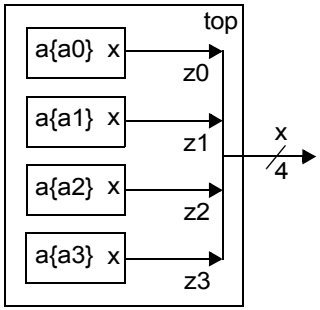
1.1.7.3.1 CSL code

```
csl_unit top, a;
top.add_instance(a{a"[0-3]"});
// adds 4 instances to top: a0, a1, a2, a3
top.add_port(output,4,x);
```

1.1.7.3.2 CSL code

```
csl_unit a, b, c, d, e, top;
top.add_instance(e, e0);
// builds the hierarchy
e.add_instance(c, c0);
e.add_instance(d, d0);
c.add_instance(a, a0);
d.add_instance(b, b0);
b.add_port(output,p);
a.add_port(input,q);
top.output(m);
```

FIGURE 1.78 concat output. assign x= {z3, z2, z1, z0}



1.1.7.3.3 Verilog code

```
module foo (x);
  output [3:0] x;
  wire z0, z1, z2, z3;
  assign x = {z3, z2, z1, z0};
  a a0(.z(z0));
  a a1(.z(z1));
  a a2(.z(z2));
  a a3(.z(z3));
endmodule

module a(x);
  output x;
endmodule
```

1.1.7.3.4 Verilog code

```
module top(m);
  output m;
  e e0(m);
endmodule

module a(q);
  input q;
endmodule
```

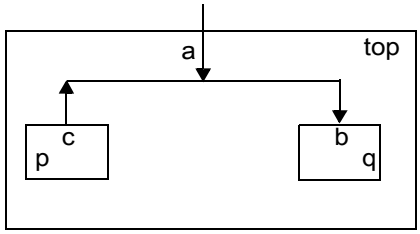
```
module b(p);
  output p;
endmodule

module c(q);
  input q;
  a a0(.q(q));
endmodule

module d(p);
  output p;
  b b0(.p(p));
endmodule

module e(m);
  output m;
  c c0(.q(q));
  d d0(.p(m));
endmodule
```

1.1.7.4 Concatenation error

1.1.7.4.1 CSL code

```
csl_unit p, q, top;
top.input(a);
q.input(b);
p.output(c);
top.add_instance(p, q);
```

FIGURE 1.79 Concatenation error



1.1.7.4.2 Verilog code

```
module top(a);
  input a;
  p p0(.p(c));
  q q0(.q(b));
endmodule

module p(c);
  output c;
endmodule

module q(b);
  input b;
endmodule
```

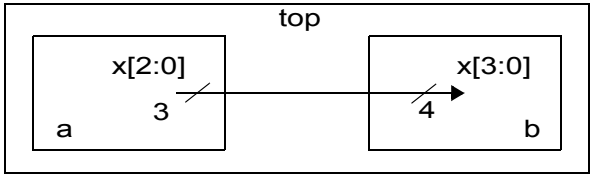
1.1.7.5 Unused bits error example

x is 3 bits on left side, and x is 4 bits on the right side.

1.1.7.5.1 CSL code

```
csl_unit top, a, b;
a.output(x[2:0]);
b.input(x[3:0]);
top.add_instance(a, b);
```

FIGURE 1.80 Unused bits



1.1.7.5.2 Verilog code

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

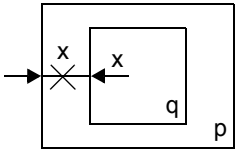
```
module a(x);
  output [2:0] x;
endmodule

module b(x);
  input [3:0] x;
endmodule
```

1.1.7.6 Concatenation error example (output and input connected)

1.1.7.6.1 CSL code

```
csl_unit p, q;
p.input(x);
q.output(x);
p.add_instance(q);
```

FIGURE 1.81 Output and input connected



1.1.7.6.2 Verilog code

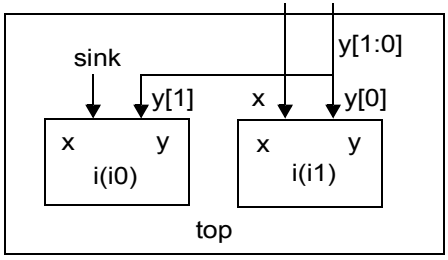
```
module p(x);
  output x;
  q q0(.x(x));
endmodule

module q(x);
  output x;
endmodule
```

1.1.8 Multiple instance

1.1.8.1 CSL code

```
    csl_unit top, i;
    top.add_instance (i0, i1);
    // alternative syntax: do not create the instances; pass the
instance names in the instance list
     top.add_instance (i{i0}, i{i1});
    csl_instance i, i0, i1;
    top.input(x);
    top.input(y[1:0]);
    i0.input(x = 0);
    i0.input(y = y[1]);
    i1.input(x = top.x);
    i1.input(y = y[0]);
```

FIGURE 1.82 instantating multiple instances



1.1.8.2 Verilog code

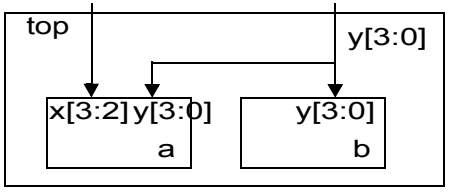
```
module top(x, y, sink);
  input [1:0] y;
  input sink, x;
  i i0(.x(sink), .y(y[1]));
  i i1(.x(X), .y(y[0]));
endmodule

module i(x, y);
  input x, y;
endmodule
```

1.1.9 Implicit mapping

1.1.9.1 CSL code

```
csl_unit top, a, b;
top.add_instance(a, b);
top.input(x[3:2]);
top.input(y[3:0]);
a.input(x[3:2]);
a.input(y[3:0]);
b.input(y[3:0]);
```

FIGURE 1.83 Implicit mappings



1.1.9.2 Verilog code

```
module top;
  a a0(.x(x[3:2]), .y(y[3:0]));
  b b0(.y(y[3:0]));
endmodule

module a(x, y);
  input [1:0] x;
  input [3:0] y;
endmodule

module b(y);
  input [3:0] y;
endmodule
```

1.1.9.3 Example Verilog to CSI signal connection

FIGURE 1.84

1.1.9.3.1 Verilog + CSI coded specfication

```
csl_unit top, x,y;
top.add_instance(x,y);

module x (a);
  output a;
endmodule

module y;
  csl_signal input(q =x.a);
endmodule
```

1.1.10 Example: implicit mapping

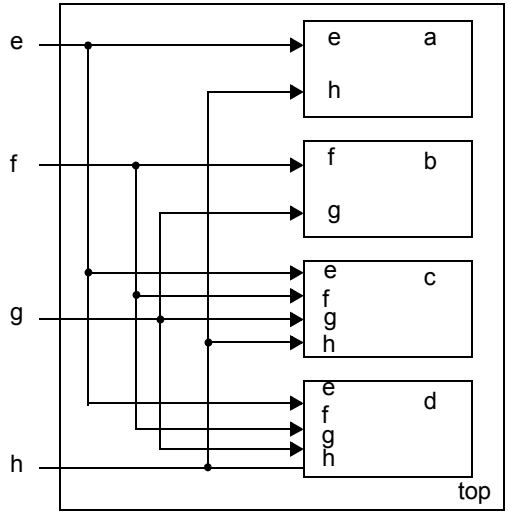
An entire upper level port list can be assigned to one or more lower level units. Ports within the portlist can be excluded from specified units. All signals are connected to all instances unless excluded.

1.1.10.1 CSL code

```
csl_unit top, a, b, c, d;
add_instance (top, a, b, c, d);
top.input(e, f, g);
csl_connect d = top;
csl_connect c = top;
```

```
   csl_connect b = top.exclude (e, h);
   // alternative: csl_connect [c, d] =  top;
   // csl_signal unitname signal_object_name - creates a new signal
with either the same signal name or a new signal name
   // csl_reverse_port_direction (unit_object_name  signal_name) -
signal name is optional;
   // if signal_name is not present entire unit portlist is copied to
the new module and the port directions are reversed
```

FIGURE 1.85 Assigning entire port lists to sub-units

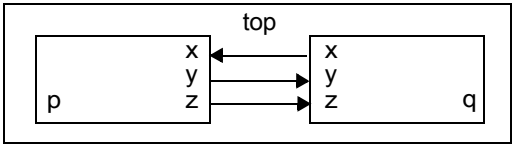


1.1.11 Reversing port lists

1.1.11.1 CSL code

```
csl_unit top, p, q;
top.add_instance(p, q);
p.output(x, y, z);
q.input(reverse(x, y, z));
// reverse the signal direction of each signal in p's interface
or
q.input(reverse(p.port()));

csl_connect p = q; // module to module connection
```

FIGURE 1.86 Port reverse



1.1.11.2 Verilog code

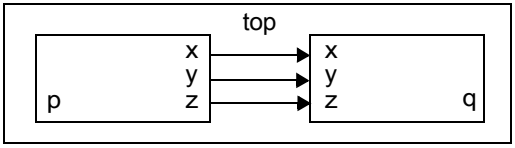
```
module top(e, f, g, h);
  input e, f, g, h;
  a a0(e, h);
  b b0 (f, g);
  c c0 (e, f, g, h);
  d d0 (e, f, g, h);
endmodule
```

1.1.12 Create a list of ports

1.1.12.1 CSL code

```
csl_unit top, p, q;
top.add_instance (p, q);
csl_list portlist = {x, y, z};
p.output(portlist);
q.input(portlist);
```

FIGURE 1.87 Port reverse



1.1.12.1.1 Verilog code

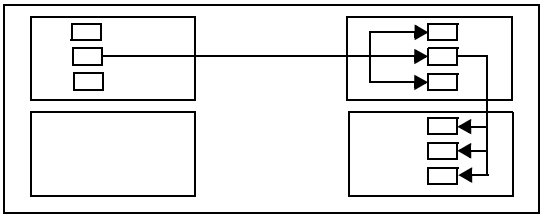
```
module top (e, f, g, h);
```

```
  input e, f, g, h;
  a a0(e, h);
  b b0(f, g);
  c c0(e, f, g, h);
  d d0(e, f, g, h);
endmodule
```

1.1.13 Connect to all units in a sub-hierarchy

1.1.13.1 CSL code

FIGURE 1.88 Connect to all units in sub-hierarchy



1.1.13.2 Verilog code

1.1.14 Examples of regular expression pattern matching

The following CSL code will create 16 different signals with the above names

```
csl_signal foo"[0-3]"_"[a-d]" = bar_"[0-3]"_"[p-s]";
```

will create 16 signals with a default type of wire and assign the wires to 16 different signals.

```
wire foo_a_0 = bar_p_0;
wire foo_a_1 = bar_p_1;
wire foo_a_2 = bar_p_2;
wire foo_a_3 = bar_p_3;
wire foo_b_0 = bar_q_0;
wire foo_?_1 = bar_q_1;
wire foo_b_2 = bar_q_2;
wire foo_b_3 = bar_q_3;
wire foo_b_0 = bar_r_0;
wire foo_c_1 = bar_r_1;
```

```
wire foo_c_2 = bar_r_2;
wire foo_c_3 = bar_r_3;
wire foo_d_0 = bar_s_0;
wire foo_d_1 = bar_s_1;
wire foo_d_2 = bar_s_2;
wire foo_d_3 = bar_s_3;
```

A pattern match can be on either the LHS or the RHS of a statement . The following csl code will create 1'b wires

```
connect bar_$1_$2= foo_(*)_(*)
wire_bar_a_0= foo_a_0;
```

Regular expressions can be used to search for names in the current name space or a specified name space.

Another way to code the above example using Perl like regular expressions is as follows:

```
csl_signal foo_[a-d]_[0-3]:
connect foo_(*)_(*) = bar_$1_$2;
```

Pattern match, just like perl.

The names are searched for all instances of **foo_(*)_(*)** and return a list of all pattern matches. Then the tool will iterate through the list and will extract the $1 and $2 pattern matches. Where $1°-6' matches the first pattern In foo_(*)_(*) and $2 matches the second pattern in **foo_(*)_(*)**. Then the pattern matches wí11 be used to create the **bar_*_*** variables and the 16 assignments will be created.

The above example uses regular expression expansion to create the signals, find the signals that were created. Extract the patterns from the list that was returned and then uses the pattern matches to create the signals on the RHS of the connection statements.

The reverse operation copies the modules port list and then reverses the port directions.

Examples of regular expression expansion:

```
foo[0-3] expands to:
foo_0
foo_1
foo_2
foo 3

bar_[a-d] expands to:
bar_a
bar_b
bar_c
bar_ d

car_[xx, yy, zz] expands to:
car_xx
car_yy
car zz

foo_[0-3]_[a-d] expands to:
foo_a_0
foo_a_1
foo_a_2
foo_a_3
foo_b_0
foo_b_1
foo_b_2
foo_b_3
foo_c_0
foo_c_1
foo_c_2
foo_c_3
foo_d_0
foo_d_1
foo_d_2
foo_d_3
```

Corresponding CSL statement:

```
csl_signal foo [0-3] _[a-d];
```

Example:

```
foo[31:12] msb(foo);  returns 31 or bar[31:0] [12:4]
```

```
msb(get_width_x(bar));  returns 12
log(numeric_expression);    takes the log base 2 of the expression
```

Example:

Address buses typically are the log bus 2 of the number of memory words. If memory has 256 words then the log base 2 of the memory is 8. 8-bits will address all possible words In the memory. Assume that the function **number_of_memory_words()** when applied to a memory returns the maximum number of words in the memory.

```
csl_signal addr csl_bitrange(log(memory.number_of_memory_words());
// creates an 8-bit wide bus If memory has 256 words

exp(base, constant_numeric_expression);
// returns base ^ constant_numeric_expression where the ^ character
Is the exponent operator
```

1.1.15 Namespace example

```
csl_instance (bar, bar2);
csl_instance (bar, bar3);

//1 to 1 connection
csl_instance (top);
csl_instance (rs);
csl_instance (fpadd);

// create the hierarchy
list top_inst_list = {rs, fpadd}
top.add_instance(top_inst_list);
current_namespace = fpadd;
csl_signal (y);
connect rs.x = fpadd.y;

// 1 to many connections
csl_unit (top, pc, id, rf, intadd, fpadd)
connect top_inst_list = debug_clk;
foreach m (pc, id, rf){
  connect m = special_clk;
}
current_namepace = (top);
```

```
   // create the hierarchy
   list top_inst_list = {pc, id, rd, fpadd, intadd};
   top.add_instance (top_inst_list);

   current_namespace = top;

   csl_signal (clk);
   clk.dir (input);
   connect top_inst_list = clk;
   // connect the to all of the instances in the top level

   csl_signal (reset_);
   reset_.dir (input);
   connect  top_inst_list = reset_;
   // connect the reset_ to all of the instance_list in the top level
to  create the signal clk in
   // all of the module in the instance_list inside each module
   // create a clk and connect each clk port to the upper level clk

   csl_signal (reset_);
```

1.1.16 Extracting data

1.1.16.1 Using functions to extract lists

Wild cards are useful in an interconnect tool. We can refer to the elements under a unit using the unit function.

- all units in this unit

```
list u = unit.get_units();
```

- all ports in this unit

```
list p = unit.get_ports();
```

- all locals in this unit

```
list l = unit.get_local_ports();
```

- all inputs:

```
list i = unit.get_inputs();
```

- all outputs

```
list o = unit.get_outputs();
```

- all inouts

```
list o = unit.get_inouts();
```

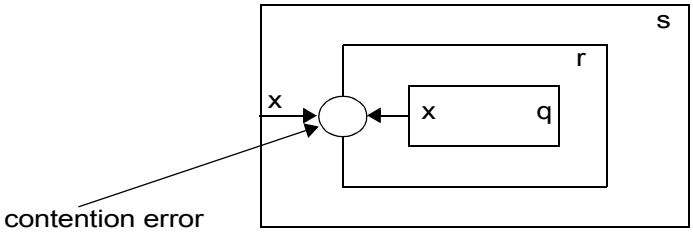
- use exclude operator to exclude a item

```
n.exclude (item_list)
```

1.1.16.1.1 CSL code

```
csl_unit q, r, s;
csl_signal s.input(x);
csl_signal q.output(x);
```

FIGURE 1.89



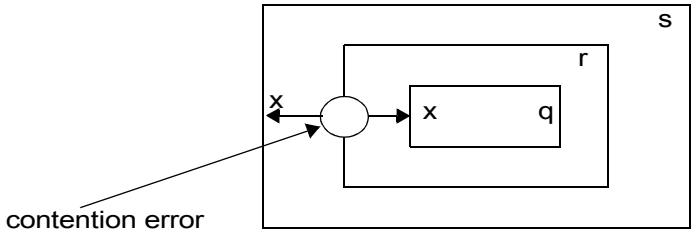
1.1.16.1.2 Verilog code

```
module s(x);
  input x;
endmodule

module q(x);
  output x;
endmodule
```

1.1.16.1.3 CSL code

```
csl_unit q, r, s;
csl_signal s.output(x);
csl_signal q.input(x);
```

FIGURE 1.90



1.1.16.1.4 Verilog code

```
module s(x);
  output x;
endmodule

module q(x);
  input x;
endmodule
```

1.1.17 CSL Interconnect Miscellaneous

1.1.17.1 Low Assertion Level suffixes

The user can use the default low assertion level suffix character which is the underscore character '_' or the user can specify the low assertion level suffix character to use to indicate that a signal has a low assertion level. To set the low assertion level suffix character to a different character set the following variable, **CSL_LCW_ASSERTION_LEVEL_CHAR**

Example:
To use the character 'n' instead of " _"
CSL LOW ASSERTION LEVEL CHAR ='n':

```
foo foo1 (
.x ({a, b, c, d})
);
endmodule
```

Then the CSL language can be used to assign x in one statement.

```
csl_unit (foo);
```

```
csl_instance(foo0, foo1);
csl_signal foo x [3:0];
x.type(input);
foo1.x = {a, b, c, d};
// automatically add the signal x to foo1's port list
```

The CSL can assign each bit of x in a separate statement

```
foo1.x [0]=a;
foo1.x [1]=b;
foo1.x [2]=c;
foo1.x [3]=d;
```

This adds x to the module foo

1.1.17.2 Searching Name Spaces Using Regular Expressions

Different search algorithms will be used depending on the symbol type that Is being searched for. The name spaces are searched for one or more symbols from the current name space up to the top of the tree . The name spaces are searched for one or more symbols from the current name space depth first to the bottom of the sub-tree. The name spaces are searched for one or more symbols from the current name space up one level at a time and then In each sub-tree in that level.

Examples:

Note that multidimensional signals can be used to write and read blocks of data to and from memories. Multidimensional signals can be indexed with one or more ranges.

Example:

```
csl_signal mem_data_out;
csl_bitrange br; br([31:0]);
mem_data_out.bitrange(8, br);
```

If a memory Is declared with the size In each dimension of 32 x 128 x 8-bits then there are 8x64 blocks of size 4x2x8-bits.

1.1.18 Verilog CSI commands

csl_connect signal_name = signal_name;
csl_reverse_ports_(unit_name);
csl_signal [type] signal_name [bitrange];

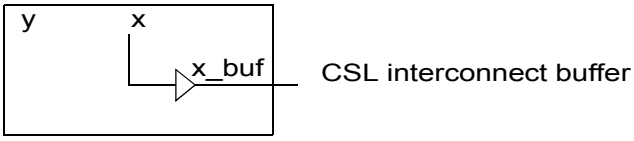
```
csl_interface interface_name(v2001_signal_declaration);


   module module_name(
     (bundle bundle_name , |
      v2001_signal_declaration, |
      insert_interface optional_reverse interface_name
optional_formal_to_actual_name_mapping_list, )*
   );
```

1.1.19 Insert output buffer

1.1.19.1 CSL code

```
csl_unit y;
y.add_signal(x);
y.output(x_inv = csl_inv (x));
```

FIGURE 1.91 insert an output buffer



1.1.19.2 Verilog code

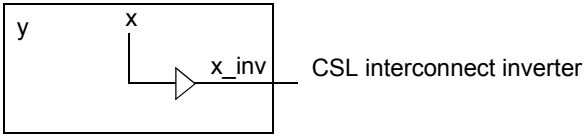
```
module y(x_buf);
 output x_buf;
 wire x;
 assign x_buf = x;
endmodule
```

1.1.20 Insert output inverter

1.1.20.1 CSL code

```
csl_unit y;
y.signal(x);
y.output(x_inv) = csl_inv(x);
```

FIGURE 1.92 insert an output inverter



1.1.20.2 Verilog code

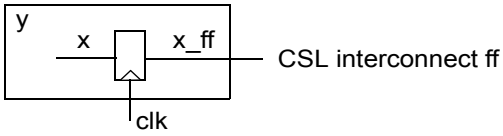
```
module y(x_inv);
  output x_inv;
  wire x;
  assign x_inv = ~ x;
endmodule
```

1.1.21 Insert output latch

1.1.21.1 CSL code

```
csl_unit y;
y.signal(x);  add a signal to y
y.output(x_ff) = csl_ff(x);
```

FIGURE 1.93 insert output latch



1.1.21.2 Verilog code

```
module y (clk, x_ff);
  output x_ff;
  input clk;
  wire x;
  assign x_ff = clk ? x : x_ff;
endmodule
```

<new>
Gen_interconnect prepending prefixes or appending suffixes automatically across modules/ pipe-stages
Transmission block takes an input signal and replicates the signal and routes the replicated signals to their destinations.
</new>

<new>
Naming convention hierarchies
suffix
signal naming convention
<signaltype> <pipestage> <valid clkedge | level> <assertion level>
re rising edge high
fe falling edge low
c combinational
L Latch
f FF
p precharge
d domino

c clk
gc gated clk
signal lint

</new>

All objects have:
- get methods which are used in expressions
- set methods which are used in commands

These methods can be used together like in the example:

```
csl_signal a,b;
csl_bitrange br(4);
a.set_bitrange(br);
b.set_bitrange(a.get_bitrange());
```

Usage of csl_list:

```
csl_list l(a,b,c,d);
csl_signal l; // a,b,c,d become signals
```

Scopes

```
csl_unit a,b,c;
a.add_instance(b,b1);
b.add_instance(c,c1);
```

In the above code:
a
a.b
a.b.c
b
b.c
c
are scopes

Connect

A port specifies a signal with a direction, the name of the signal and the type.

1.1.22 example 1

```
csl_unit un1, un0;
csl_unit un1, un0;
un1.set_interface(un0.get_interface());
```

```
//this copies the interface from un0 and sets it to un1
```

1.1.23 example 2

```
csl_unit un0, un1;
un1.connect.un0;
//by connecting the two units the auto router will set the inputs in
//un1 interface to outputs and the outpus to inputs so that the
//connection with un0 would be possible. Note that the auto router will
//reverse the port directions only if un1 and un0 are at the same level
//in the design hierarchy
```

<ADD>
Configuration
Naming convention hierarchies
suffix
signal naming convention
<signaltype> <pipestage> <valid clkedge | level> <assertion level>
re rising edge high
fe falling edge low
c combinational
L Latch
f FF
p precharge
d domino
c clk
gc gated clk
signal lint

</ADD>
old set_offset()

CSL CODE

```
//AB
csl_unit chip, u1, u2, u3, ug;
//creating 4 uninitialized signal objects
csl_signal sig_u1,sig_u2,sig_u3,sig_m;
sig_u1.set_range(11,6);
sig_u2.set_range(5,2);
sig_u3.set_range(1,0);
sig_m.set_range(11,0);
```

```
/* creating 2 additional signals and "linking"
their properties to sig_m */
sig_m_in.set_bitrange(sig_m.get_bitrange());
sig_m_out.set_bitrange(sig_m.get_bitrange());
chip.add_port(output,sig_m);
//setting the offset for sig_m; bitrange becomes [15:4]
chip.sig_m.set_offset(4);

ug.add_port(input,sig_m_in);
//using get() method along with a set method
ug.sig_m_in.set_offset(chip.sig_m.get_offset());
ug.add_port(output,sig_m_out);
ug.sig_m_out.set_offset(chip.sig_m.get_offset());
u1.add_port(output,sig_u1);
//setting a specific offset using get() method
sig_u1.set_offset(chip.sig_m.get_offset()+4);
u2.add_port(output,sig_u2);
sig_u2.set_offset(sig_u1.get_offset());
u3.add_port(output,sig_u3);
sig_u3.set_offset(sig_u1.get_offset());

chip.add_instance(ug, ug0);
chip.add_instance(u1, u10);
chip.add_instance(u2, u20);
chip.add_instance(u3, u30);
//making the connections
scope chip{
ug0.sig_m_in.connect({u10.sig_u1,u20.sig_u2,u30.sig_u3});
/* the following line may be redundant if
the autorouter width inference is on */
ug0.sig_m_out.connect(sig_m);
}
```

VERILOG CODE

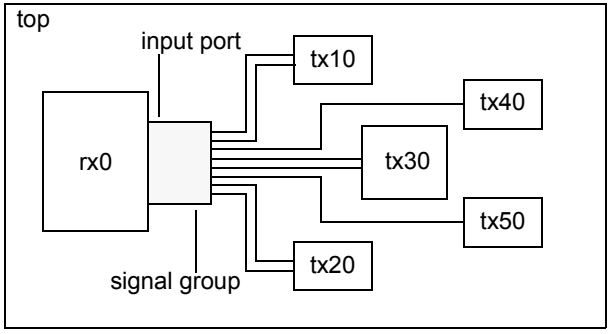
```
//AB
`define OFFSET_M 4
`define OFFSET_U 8

module chip(sig_m);
  output [11+`OFFSET_M:0+`OFFSET_M] sig_m;
  wire [11+`OFFSET_M:0+`OFFSET_M] sig_connect;
```

```
    u1 u10(.sig_u1(sig_connect[11+`OFFSET_M:6+`OFFSET_M]));
    u2 u20(.sig_u2(sig_connect[5+`OFFSET_M:2+`OFFSET_M]));
    u3 u30(.sig_u3(sig_connect[1+`OFFSET_M:0+`OFFSET_M]));
    ug ug0(.sig_m_in(sig_connect),.sig_m_out(sig_m));
endmodule

module u1(sig_u1);
  output [11+`OFFSET_U:6+`OFFSET_U] sig_u1;
endmodule

module u2(sig_u2);
  output [5+`OFFSET_U:2+`OFFSET_U] sig_u2;
endmodule

module u3(sig_u3);
  output [1+`OFFSET_U:0+`OFFSET_U] sig_u3;
endmodule

module ug(sig_m_in,sig_m_out);
  input [11+`OFFSET_M:0+`OFFSET_M] sig_m_in;
  output [11+`OFFSET_M:0++`OFFSET_M] sig_m_out;
endmodule
```

EXAMPLE :

Using a signal group to connect multiple signals from transmitting units to a single port in a receiving unit. The top_design unit where the component units will get instantiated is generated automatically.

FIGURE 1.94 Grouping signals



CSL CODE

```
FIX all examples with add_port group -> convert it to add_interface
csl_unit rx, tx1, tx2, tx3, tx4, tx5, top;
```

```
scope top {
  csl_signal s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,s5_0;
  //create the signal group with no parameters for the constructor
  csl_signal_group sgn;
  sgn.add_signal_list(csl_list(s1_0,s2_0,s2_1,s1_1,s3_0,s3_1,s4_0,
s5_0);
}
tx1.add_port_list(output,csl_list(top.s1_0,top.s1_1));
tx2.add_port_list(output,csl_list(top.s2_0,top.s2_1));
tx3.add_port_list(output,csl_list(top.s3_0,top.s3_1));
tx4.add_port(output,top.s4_0);
tx5.add_port(output,top.s5_0);
//creates an input port sgn width wide
rx.add_port(input,sgn.get_width(),top.sgn);
scope top {
  add_instance(rx,rx0(.sgn(sgn));
  add_instance(tx1,tx10(.s1_0(s1_0),.s1_1(s1_1));
  add_instance(tx2,tx20(.s2_0(s2_0),.s2_1(s2_1));
  add_instance(tx3,tx30(.s3_0(s3_0),.s3_1(s3_1));
  add_instance(tx4,tx40(.s4_0(s4_0));
  add_instance(tx5,tx50(.s5_0(s5_0));
}
```

VERILOG CODE

```
`define WIDTH_s1_1 1
`define WIDTH_s2_0 1
`define WIDTH_s2_1 1
`define WIDTH_s3_0 1
`define WIDTH_s3_1 1
`define WIDTH_s4_0 1
`define WIDTH_s5_0 1

module top();
  wire [`WIDTH_s1_0-1:0] s1_0;
  wire [`WIDTH_s1_1-1:0] s1_1;
  wire [`WIDTH_s2_0-1:0] s2_0;
  wire [`WIDTH_s2_1-1:0] s2_1;
  wire [`WIDTH_s3_0-1:0] s3_0;
  wire [`WIDTH_s3_1-1:0] s3_1;
  wire [`WIDTH_s4_0-1:0] s4_0;
  wire [`WIDTH_s5_0-1:0] s5_0;
```

```
  wire
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
  rx rx0(.sgn(sgn));
  tx1 tx10(.s1_0(s1_0),.s1_1(s1_1));
  tx2 tx20(.s2_0(s2_0),.s2_1(s2_1));
  tx3 tx30(.s3_0(s3_0),.s3_1(s3_1));
  tx4 tx40(.s4_0(s4_0));
  tx5 tx50(.s5_0(s5_0));

  assign sgn = {s1_0,s1_1,s2_0,s2_1,s3_0,s3_1,s4_0,s5_0};

endmodule

module rx(sgn);
  input
[`WIDTH_s1_0+`WIDTH_s1_1+`WIDTH_s2_0+`WIDTH_s2_1+`WIDTH_s3_0+`WIDTH_s3
_1+`WIDTH_s4_0+`WIDTH_s5_0-1:0] sgn;
endmodule

module tx1(s1_0,s1_1);
  output [`WIDTH_s1_0-1:0] s1_0;
  output [`WIDTH_s1_1-1:0] s1_1;
endmodule

module tx2(s2_0,s2_1);
  output [`WIDTH_s2_0-1:0] s2_0;
  output [`WIDTH_s2_1-1:0] s2_1;
endmodule

module tx3(s3_0,s3_1);
  output [`WIDTH_s3_0-1:0] s3_0;
  output [`WIDTH_s3_1-1:0] s3_1;
endmodule

module tx4(s4_0);
  output [`WIDTH_s4_0-1:0] s4_0;
endmodule

module tx5(s5_0);
  output [`WIDTH_s5_0-1:0] s5_0;
```

```
endmodule
```