1.1 CSL Interconnect Command Summary

Signals

```
    signal_object_name.set_offset(numeric_expression);
   constant_numeric_expression signal_object_name.get_offset();
   ir.instr.field.create_signal(signal_name);
   ir.inst_fmt_name.field_name.connect(signal_name);
```

Multidimensional Signals

```
    signal_object_name.set_number_of_dimensions(numeric_expression);
    int signal_object_name.get_number_of_dimensions();
    signal_object_name.set_dim_width(dimension_number,
    numeric_expression);
    int signal_object_name.get_dim_width(dimension_number);
    signal_object_name.set_dim_bitrange(dimension_number,bitrange_object_n
    ame);
    bitrange_object signal_object_name.get_dim_bitrange(dimension_number);
    signal_object_name.set_dim_range(dimension_number,upper_limit,
    lower_limit);
    int signal_object_name.get_dim_lower_index(dimension_number);
    int signal_object_name.get_dim_upper_index(dimension_number);
    signal_object_name.set_dim_offset(dimension_number,
    constant_numeric_expression);
   constant_numeric_expression
signal_object_name.get_dim_offset(dimension_number);
csl_list_name.set_attr(csl_signal_attribute);
```

Units: prefix

```
set_unit_prefix(prefix_string[,prefix_specifier]);
set_signal_prefix(prefix_string);
set_signal_prefix_local(prefix_string);
set_clk_all();
```

Units: instance control bit

```
set_instance_alteration_bit(status);
```

Units: input/output file type

```
    unit_object_name.input_verilog_type(verilog_type);
unit_object_name.output_verilog_type(verilog_type);
```

New commands

```
add_logic(external_unit_enable);
(unit_name | instance_name).add_logic(unit_address_decoder,
address_signal_name);
signal_name.(field_name.)*add_logic(gen_decoder);
[(interface_name.)+]register_ios(input|output [,
.reset[_](optional_reset), reset_value][,.en(optional_enable)]);
generate_individual_rtl_signals(on|off);
```

Gets methods

```
int signal_object.get_width();
bitrange_object signal_object.get_bitrange();
int signal_object.get_lower_index();
int signal_object.get_upper_index();
csl_signal_type signal_object.get_type();
csl_list_name.set_attr(csl_signal_attribute);
csl_list_name.set_attr(csl_signal_attribute);
int signal_group_object.get_width();
unit_name.add_port_list([port_direction,]interface_object);
int port_object.get_width();
bitrange_object port_object.get_bitrange();
int port_object_name.get_lower_index();
int port_object_name.get_upper_index();
csl_port_type port_object_name.get_type();
csl_port_attr port_object_name.get_attr();
int interface_object.get_width();
string signal_object_name.get_signal_prefix_local();
set_signal_prefix(prefix_string);
set_signal_prefix_local(prefix_string);
```

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

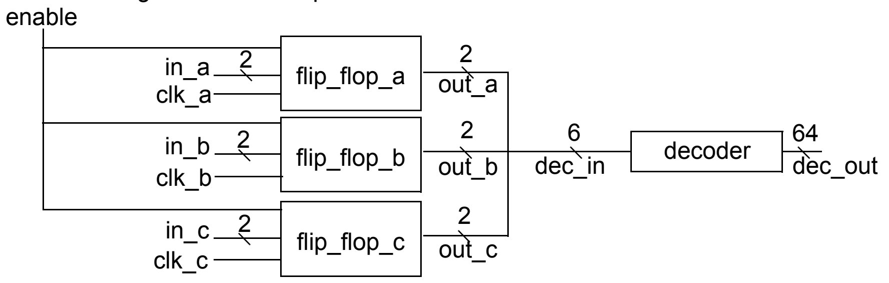
DESCRIPTION :

Set the offset value for signal *signal_object_name*.It adds the value *numeric_expression* both to lower index and upper index of the signal's range.This command is usefull in the context of the autorouter usage.

EXAMPLE :

In this example we use an decoder that decodes a word of 6 bits composed of the concatenation of the outputs of 3 2-bit D flipflops

FIGURE 1.1 signal offset example



CSL CODE

```
csl_unit _2bit_flip_flop,decoder,top;
scope _2bit_flip_flop {
add_port(input,2,data);
add_port(input,clock);
add_port(input,enable);
add_port(output,reg,2,q);
}
scope decoder {
add_port(input,6,dec_in);
add_port(output,reg,64,dec_out)
}
scope top {
//signal definitions
csl_signal
clk_a,clk_b,clk_c,in_a(2),in_b(2),in_c(2),enable,dec_in(6),dec_out(64)
;
//instantiation and interconnection of modules
add_instance(_2bit_flip_flop,flip_flop_a(.data(in_a),.enable(enable),.
clock(clk_a),.q(out_a)));
add_instance(_2bit_flip_flop,flip_flop_b(.data(in_b),.enable(enable),.
clock(clk_b),.q(out_b)));
add_instance(_2bit_flip_flop,flip_flop_c(.data(in_c),.enable(enable),.
clock(clk_c),.q(out_c)));
add_instance(decoder,decoder(.dec_in(dec_in),.dec_out(dec_out)));

//set_offset illustration
out_b.set_offset(2);out_c.set_offset(4);
set_autorouter(ON);

}
```

//

NEW CSL CODE

```
csl_unit _2bit_flip_flop, decoder, top;

csl_unit  _2bit_flip_flop {
csl_port data(input,2), clock (input), enable(input), q(output,reg,2);
_2bit_flip_flop(){}
};
csl_unit decoder {
csl_port dec_in(input,6), dec_out(output,reg,64);
decoder(){}
};
csl_unit  top {
csl_signal clk_a, clk_b, clk_c, in_a(2), in_b(2), in_c(2), enable,
dec_in(6), dec_out(64), out_a(2), out_b(2), out_c(2);

_2bit_flip_flop flip_flop_a( .data(in_a), .enable(enable),
.clock(clk_a), .q(out_a));
_2bit_flip_flop flip_flop_b( .data(in_b), .enable(enable),
.clock(clk_b), .q(out_b));
_2bit_flip_flop flip_flop_c( .data(in_c), .enable(enable),
.clock(clk_c), .q(out_c));
decoder decoder( .dec_in(dec_in), .dec_out(dec_out));

top(){



//set_autorouter(ON);
 }
};
```

NEW VERILOG CODE

```
module _2bit_flip_flop(data,
                       clock,
                       enable,
                       q);
  input [1:0] data;
  input [1:0] clock;
  input [1:0] enable;
  output reg [1:0] q;
endmodule

module decoder(dec_in,
               dec_out);
  input [5:0] dec_in;
  output reg [63:0] dec_out;
endmodule

module top();
  wire clk_a;
  wire clk_b;
  wire clk_c;
  wire [1:0] in_a;
  wire [1:0] in_b;
  wire [1:0] in_c;
  wire enable;
  wire [5:0] dec_in;
  wire [63:0] dec_out;
  wire [1:0] out_a;
  wire [3:2] out_b;
  wire [5:4] out_c;
  _2bit_flip_flop flip_flop_a(.clock(clk_a),
                              .data(in_a),
                              .enable(enable),
                              .q(out_a));
  _2bit_flip_flop flip_flop_b(.clock(clk_b),
                              .data(in_b),
                              .enable(enable),
                              .q(out_b));
  _2bit_flip_flop flip_flop_c(.clock(clk_c),
                              .data(in_c),
                              .enable(enable),
                              .q(out_c));
  decoder decoder(.dec_in(dec_in),
                  .dec_out(dec_out));
endmodule
```

constant_numeric_expression signal_object_name.**get_offset**();

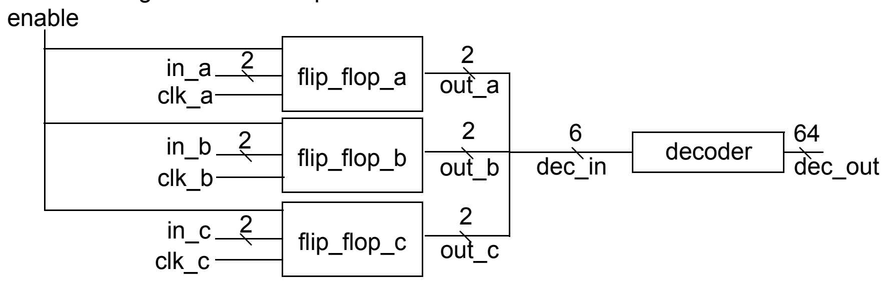
DESCRIPTION :

Return the offset value for a signal.The offset for a signal is,if it is not already set, 0.

EXAMPLE :

In this example we use an decoder that decodes a word of 6 bits composed of the concatenation of the outputs of 3 2-bit D flipflops

FIGURE 1.2 signal offset example



CSL CODE

```
csl_unit _2bit_flip_flop,decoder,top;
scope _2bit_flip_flop {
add_port(input,2,data);
add_port(input,clock);
add_port(input,enable);
add_port(output,reg,2,q);
}
scope decoder {
add_port(input,6,dec_in);
add_port(output,reg,64,dec_out)
}
scope top {
//signal definitions
csl_signal
clk_a,clk_b,clk_c,in_a(2),in_b(2),in_c(2),enable,dec_in(6),dec_out(64)
;
//instantiation and interconnection of modules
add_instance(_2bit_flip_flop,flip_flop_a(.data(in_a),.enable(enable),.
clock(clk_a),.q(out_a)));
```

```
    add_instance(_2bit_flip_flop,flip_flop_b(.data(in_b),.enable(enable),.
    clock(clk_b),.q(out_b)));
    add_instance(_2bit_flip_flop,flip_flop_c(.data(in_c),.enable(enable),.
    clock(clk_c),.q(out_c)));
    add_instance(decoder,decoder(.dec_in(dec_in),.dec_out(dec_out)));

    //set_offset illustration
    out_b.set_offset(2);
    out_c.set_offset(4);
    set_autorouter(ON);

}
```

//

NEW CSL CODE

```
    csl_unit _2bit_flip_flop, decoder, top;

    csl_unit  _2bit_flip_flop {
    csl_port data(input,2), clock (input), enable(input), q(output,reg,2);
    _2bit_flip_flop(){}
    };
    csl_unit decoder {
    csl_port dec_in(input,6), dec_out(output,reg,64);
    decoder(){}
    };
    csl_unit  top {
    csl_signal clk_a, clk_b, clk_c, in_a(2), in_b(2), in_c(2), enable,
    dec_in(6), dec_out(64), out_a(2), out_b(2), out_c(2);

    _2bit_flip_flop flip_flop_a( .data(in_a), .enable(enable),
    .clock(clk_a), .q(out_a));
    _2bit_flip_flop flip_flop_b( .data(in_b), .enable(enable),
    .clock(clk_b), .q(out_b));
    _2bit_flip_flop flip_flop_c( .data(in_c), .enable(enable),
    .clock(clk_c), .q(out_c));
    decoder decoder( .dec_in(dec_in), .dec_out(dec_out));

    top(){



    //set_autorouter(ON);
    }
};
```

//

NEW VERILOG CODE

```
   module _2bit_flip_flop(data,
                          clock,
                          enable,
                          q);
     input [1:0] data;
     input [1:0] clock;
     input [1:0] enable;
     output reg [1:0] q;
   endmodule

   module decoder(dec_in,
                  dec_out);
     input [5:0] dec_in;
     output reg [63:0] dec_out;
   endmodule

   module top();
     wire clk_a;
     wire clk_b;
     wire clk_c;
     wire [1:0] in_a;
     wire [1:0] in_b;
     wire [1:0] in_c;
     wire enable;
     wire [5:0] dec_in;
     wire [63:0] dec_out;
     wire [1:0] out_a;
     wire [3:2] out_b;
     wire [5:4] out_c;
     _2bit_flip_flop flip_flop_a(.clock(clk_a),
                                 .data(in_a),
                                 .enable(enable),
                                 .q(out_a));
     _2bit_flip_flop flip_flop_b(.clock(clk_b),
                                 .data(in_b),
                                 .enable(enable),
                                 .q(out_b));
     _2bit_flip_flop flip_flop_c(.clock(clk_c),
                                 .data(in_c),
                                 .enable(enable),
                                 .q(out_c));
     decoder decoder(.dec_in(dec_in),
                     .dec_out(dec_out));
   endmodule
```

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

DESCRIPTION :

EXAMPLE :

```
csl_field ir(3);
csl_signal s0; // the width is set automatically
csl_signal s1; // the width is set automatically

ir.instr.field.create_signal(s0);
ir.inst_fmt_name.field_name.connect(s1);
ir.inst_fmt_name.field_name.connect(signal_name);
```

DESCRIPTION :

EXAMPLE :

```
   csl_field ir(3);
   csl_signal s0; // the width is set automatically
   csl_signal s1; // the width is set automatically

   ir.instr.field.create_signal(s0);
   ir.inst_fmt_name.field_name.connect(s1);
```

1.1.0.1 Multidimesional Signals

signal_object_name.**set_number_of_dimensions(***numeric_expression***);**

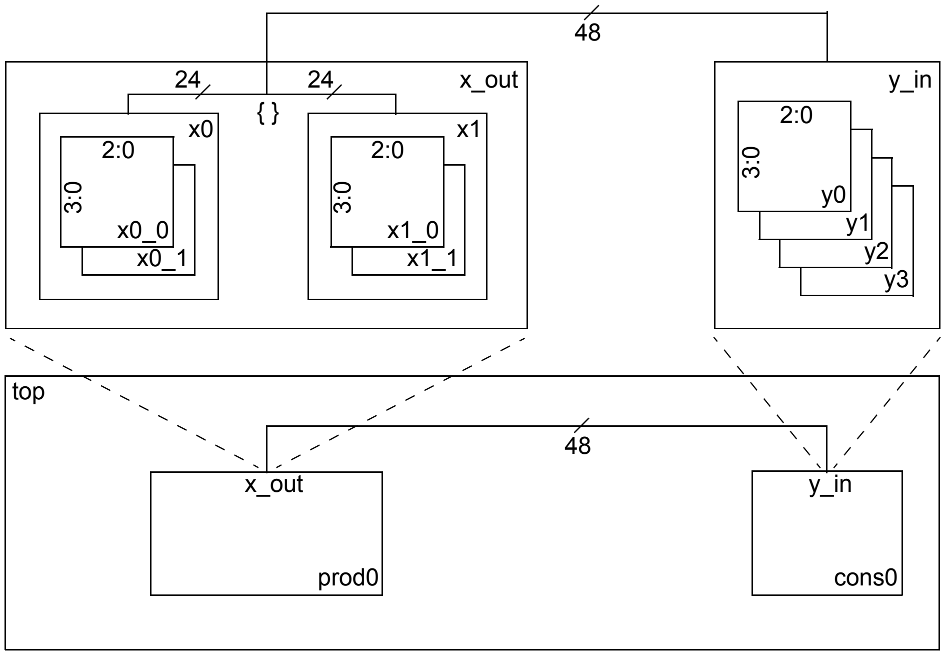
DESCRIPTION :

Sets the dimension number of a multi dimensional signal to *numeric_expression*. If the user does not declare *numeric_expression* dimensions in the signal, a compile time error occurs.
!!add to warn/error doc

EXAMPLE :

small description of the example.

FIGURE 1.3 Multidimensional signals setup



CSL CODE:

```
csl_unit u{
  csl_signal sig1;
  u(){
    sig1.set_number_of_dimension(3);
  }
};
```

CSL CODE

```
//AB
csl_unit top, prod, cons;
```

```
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions

//each dimension width is set
x_out.set_dim_width(0,3);
x_out.set_dim_width(1,4);
x_out.set_dim_width(2,2);
x_out.set_dim_width(3,get_dim_width(2));
}
cons.add_signal(wire,y_in);
scope cons{
//signal y_in of unit cons is set to 3 dimensions; (4-1)


y_in.set_dim_width(0,prod.x_out.get_dim_width(0));
y_in.set_dim_width(1,prod.x_out.get_dim_width(1));
y_in.set_dim_width(2,prod.x_out.get_dim_width(2)+prod.x_out.get_dim_wi
dth(3)));
}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0 3
`define SIG1_DIM1 4
`define SIG1_DIM2 2
`define SIG1_DIM3 2
`define SIG1_DIM_MAX `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2*`SIG1_DIM3
`define SIG1_DIM_MED `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2
`define SIG1_DIM_MIN `SIG1_DIM0*`SIG1_DIM1

`define SIG2_DIM0 3
`define SIG2_DIM1 4
`define SIG2_DIM2 4
`define SIG2_DIM_MAX `SIG2_DIM0*`SIG2_DIM1*`SIG2_DIM2
`define SIG2_DIM_MIN `SIG2_DIM0*`SIG2_DIM1
```

```
module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
  prod prod0(.x_out(trans));
  cons cons0(.y_in(trans));
endmodule

module prod(x_out);
   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0 ;
   wire [`SIG1_DIM_MIN-1:0] x0_1 ;
   wire [`SIG1_DIM_MIN-1:0] x1_0 ;
   wire [`SIG1_DIM_MIN-1:0] x1_1 ;

   wire [`SIG1_DIM0 - 1 : 0] x000;
   wire [`SIG1_DIM0 - 1 : 0] x001;
   wire [`SIG1_DIM0 - 1 : 0] x002;
   wire [`SIG1_DIM0 - 1 : 0] x003;

   wire [`SIG1_DIM0 - 1 : 0] x010;
   wire [`SIG1_DIM0 - 1 : 0] x011;
   wire [`SIG1_DIM0 - 1 : 0] x012;
   wire [`SIG1_DIM0 - 1 : 0] x013;

   wire [`SIG1_DIM0 - 1 : 0] x100;
   wire [`SIG1_DIM0 - 1 : 0] x101;
   wire [`SIG1_DIM0 - 1 : 0] x102;
   wire [`SIG1_DIM0 - 1 : 0] x103;

   wire [`SIG1_DIM0 - 1 : 0] x110;
   wire [`SIG1_DIM0 - 1 : 0] x111;
   wire [`SIG1_DIM0 - 1 : 0] x112;
   wire [`SIG1_DIM0 - 1 : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x0 = {x0_0, x0_1};
   assign x1 = {x1_0, x1_1};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
   input [`SIG2_DIM_MAX-1:0] y_in;

   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;

   wire [`SIG1_DIM0 - 1 : 0] y000;
   wire [`SIG1_DIM0 - 1 : 0] y001;
   wire [`SIG1_DIM0 - 1 : 0] y002;
   wire [`SIG1_DIM0 - 1 : 0] y003;

   wire [`SIG1_DIM0 - 1 : 0] y010;
   wire [`SIG1_DIM0 - 1 : 0] y011;
   wire [`SIG1_DIM0 - 1 : 0] y012;
   wire [`SIG1_DIM0 - 1 : 0] y013;

   wire [`SIG1_DIM0 - 1 : 0] y100;
   wire [`SIG1_DIM0 - 1 : 0] y101;
   wire [`SIG1_DIM0 - 1 : 0] y102;
   wire [`SIG1_DIM0 - 1 : 0] y103;

   wire [`SIG1_DIM0 - 1 : 0] y110;
   wire [`SIG1_DIM0 - 1 : 0] y111;
   wire [`SIG1_DIM0 - 1 : 0] y112;
   wire [`SIG1_DIM0 - 1 : 0] y113;

   assign y0 = {y000, y001, y002, y003};
   assign y1 = {y010, y011, y012, y013};
   assign y2 = {y100, y101, y102, y103};
   assign y3 = {y110, y111, y112, y113};
   assign y_in = {y0, y1, y2, y3};

endmodule
```

`int signal_object_name.get_number_of_dimensions();`

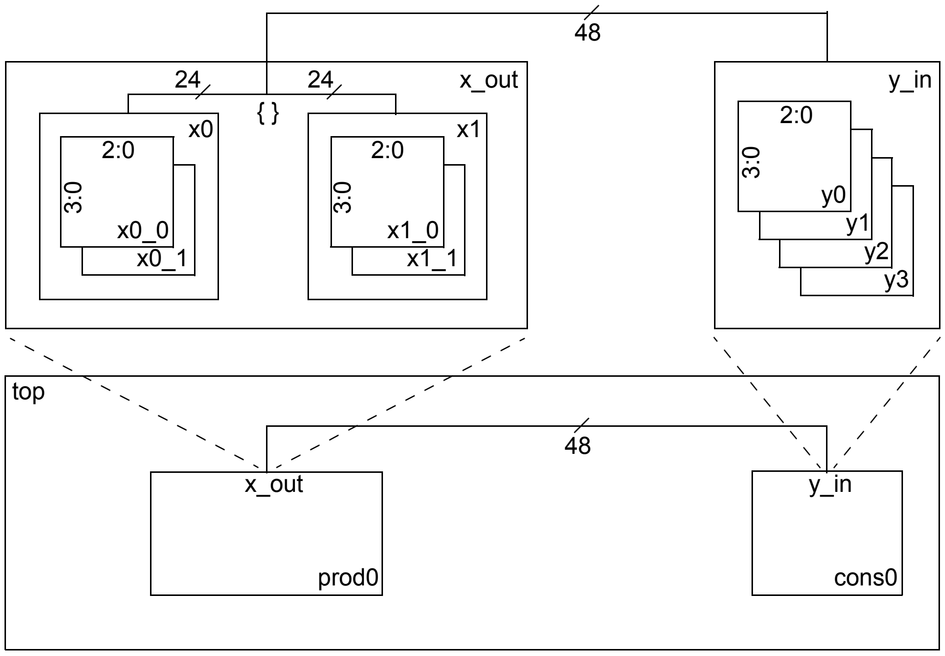
DESCRIPTION :

Sets the dimension number of a multi dimensional signal to *numeric_expression*. If the user does not declare *numeric_expression* dimensions in the signal, a compile time error occurs.
!!add to warn/error doc

EXAMPLE :

small description of the example.

FIGURE 1.4 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//each dimension width is set
x_out.set_dim_width(0,3);
x_out.set_dim_width(1,4);
x_out.set_dim_width(2,2);
x_out.set_dim_width(3,get_dim_width(2));
}
```

```
cons.add_signal(wire,y_in);
scope cons{
/* signal y_in of unit cons is set to 4-1=3 dimensions according to
 signal x_out of unit prod; this way a dependency is created */


y_in.set_dim_width(0,prod.x_out.get_dim_width(0));
y_in.set_dim_width(1,prod.x_out.get_dim_width(1));
y_in.set_dim_width(2,prod.x_out.get_dim_width(2)+prod.x_out.get_dim_wi
dth(3)));
}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0 3
`define SIG1_DIM1 4
`define SIG1_DIM2 2
`define SIG1_DIM3 2
`define SIG1_DIM_MAX `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2*`SIG1_DIM3
`define SIG1_DIM_MED `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2
`define SIG1_DIM_MIN `SIG1_DIM0*`SIG1_DIM1

`define SIG2_DIM0 3
`define SIG2_DIM1 4
`define SIG2_DIM2 4
`define SIG2_DIM_MAX `SIG2_DIM0*`SIG2_DIM1*`SIG2_DIM2
`define SIG2_DIM_MIN `SIG2_DIM0*`SIG2_DIM1

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
  prod prod0(.x_out(trans));
  cons cons0(.y_in(trans));
endmodule

module prod(x_out);
   output [`SIG1_DIM_MAX-1:0] x_out;
   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0 ;
   wire [`SIG1_DIM_MIN-1:0] x0_1 ;
   wire [`SIG1_DIM_MIN-1:0] x1_0 ;
   wire [`SIG1_DIM_MIN-1:0] x1_1 ;

   wire [`SIG1_DIM0 - 1 : 0] x000;
   wire [`SIG1_DIM0 - 1 : 0] x001;
   wire [`SIG1_DIM0 - 1 : 0] x002;
   wire [`SIG1_DIM0 - 1 : 0] x003;

   wire [`SIG1_DIM0 - 1 : 0] x010;
   wire [`SIG1_DIM0 - 1 : 0] x011;
   wire [`SIG1_DIM0 - 1 : 0] x012;
   wire [`SIG1_DIM0 - 1 : 0] x013;

   wire [`SIG1_DIM0 - 1 : 0] x100;
   wire [`SIG1_DIM0 - 1 : 0] x101;
   wire [`SIG1_DIM0 - 1 : 0] x102;
   wire [`SIG1_DIM0 - 1 : 0] x103;

   wire [`SIG1_DIM0 - 1 : 0] x110;
   wire [`SIG1_DIM0 - 1 : 0] x111;
   wire [`SIG1_DIM0 - 1 : 0] x112;
   wire [`SIG1_DIM0 - 1 : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x0 = {x0_0, x0_1};
   assign x1 = {x1_0, x1_1};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
```

```
   input [`SIG2_DIM_MAX-1:0] y_in;

   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;

   wire [`SIG1_DIM0 - 1 : 0] y000;
   wire [`SIG1_DIM0 - 1 : 0] y001;
   wire [`SIG1_DIM0 - 1 : 0] y002;
   wire [`SIG1_DIM0 - 1 : 0] y003;

   wire [`SIG1_DIM0 - 1 : 0] y010;
   wire [`SIG1_DIM0 - 1 : 0] y011;
   wire [`SIG1_DIM0 - 1 : 0] y012;
   wire [`SIG1_DIM0 - 1 : 0] y013;

   wire [`SIG1_DIM0 - 1 : 0] y100;
   wire [`SIG1_DIM0 - 1 : 0] y101;
   wire [`SIG1_DIM0 - 1 : 0] y102;
   wire [`SIG1_DIM0 - 1 : 0] y103;

   wire [`SIG1_DIM0 - 1 : 0] y110;
   wire [`SIG1_DIM0 - 1 : 0] y111;
   wire [`SIG1_DIM0 - 1 : 0] y112;
   wire [`SIG1_DIM0 - 1 : 0] y113;

   assign y0 = {y000, y001, y002, y003};
   assign y1 = {y010, y011, y012, y013};
   assign y2 = {y100, y101, y102, y103};
   assign y3 = {y110, y111, y112, y113};

   assign y_in = {y0, y1, y2, y3};

endmodule
```

signal_object_name.**set_dim_width(***dimension_number*, *numeric_expression***);**

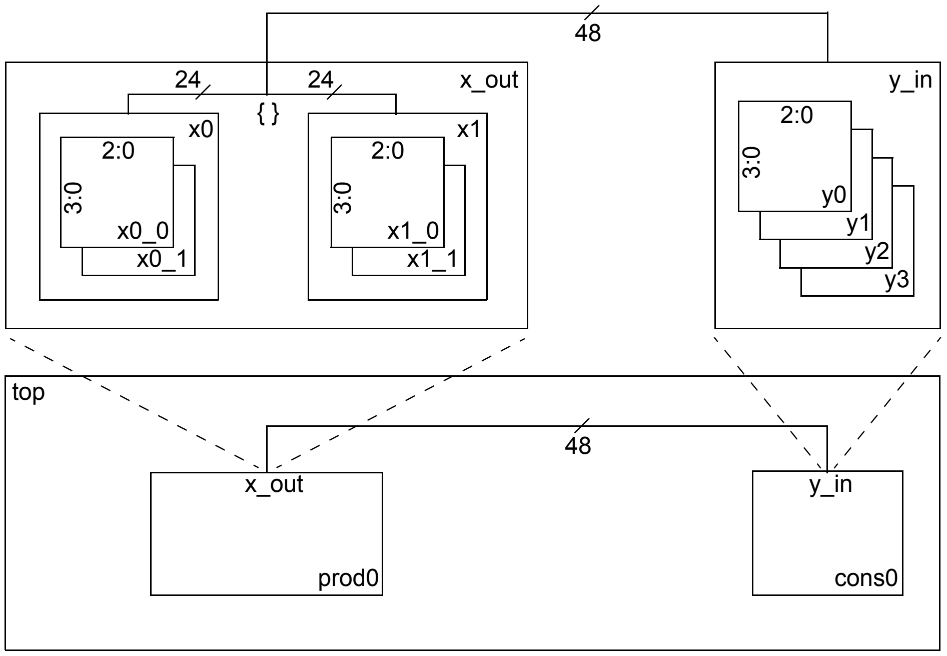
DESCRIPTION :

Sets the dimension *dimension_number* width of a multi dimensional signal to *numeric_expression*. If dimension *dimension_number* does not exist, the compiler will flag an error.

EXAMPLE :

A short example on connecting a 4 dimensional signal from a unit with a 3 dimensional signal from another unit is provided. Figure 1.5 shows the general layout of the mulidimensional signals (top) and the position in their corresponding units respectively (bottom).

FIGURE 1.5 Multidimensional signals setup



CSL CODE

```
//AB
//I can also define dimensions as constants like in the verilog
//example below but we need to agree on the declaration of constants
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//each dimension width is set
}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//set_dim_width() method with get_dim_width() method




}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0 3
`define SIG1_DIM1 4
`define SIG1_DIM2 2
`define SIG1_DIM3 2
`define SIG1_DIM_MAX `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2*`SIG1_DIM3
`define SIG1_DIM_MED `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2
`define SIG1_DIM_MIN `SIG1_DIM0*`SIG1_DIM1

`define SIG2_DIM0 3
`define SIG2_DIM1 4
`define SIG2_DIM2 4
`define SIG2_DIM_MAX `SIG2_DIM0*`SIG2_DIM1*`SIG2_DIM2
`define SIG2_DIM_MIN `SIG2_DIM0*`SIG2_DIM1

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
  prod prod0(.x_out(trans));
  cons cons0(.y_in(trans));
endmodule
```

```
module prod(x_out);
   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0 ;
   wire [`SIG1_DIM_MIN-1:0] x0_1 ;
   wire [`SIG1_DIM_MIN-1:0] x1_0 ;
   wire [`SIG1_DIM_MIN-1:0] x1_1 ;

   wire [`SIG1_DIM0 - 1 : 0] x000;
   wire [`SIG1_DIM0 - 1 : 0] x001;
   wire [`SIG1_DIM0 - 1 : 0] x002;
   wire [`SIG1_DIM0 - 1 : 0] x003;

   wire [`SIG1_DIM0 - 1 : 0] x010;
   wire [`SIG1_DIM0 - 1 : 0] x011;
   wire [`SIG1_DIM0 - 1 : 0] x012;
   wire [`SIG1_DIM0 - 1 : 0] x013;

   wire [`SIG1_DIM0 - 1 : 0] x100;
   wire [`SIG1_DIM0 - 1 : 0] x101;
   wire [`SIG1_DIM0 - 1 : 0] x102;
   wire [`SIG1_DIM0 - 1 : 0] x103;

   wire [`SIG1_DIM0 - 1 : 0] x110;
   wire [`SIG1_DIM0 - 1 : 0] x111;
   wire [`SIG1_DIM0 - 1 : 0] x112;
   wire [`SIG1_DIM0 - 1 : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x0 = {x0_0, x0_1};
   assign x1 = {x1_0, x1_1};

   assign x_out = {x0,x1};
```

```
endmodule

module cons(y_in);
   input [`SIG2_DIM_MAX-1:0] y_in;

   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;

   wire [`SIG1_DIM0 - 1 : 0] y000;
   wire [`SIG1_DIM0 - 1 : 0] y001;
   wire [`SIG1_DIM0 - 1 : 0] y002;
   wire [`SIG1_DIM0 - 1 : 0] y003;

   wire [`SIG1_DIM0 - 1 : 0] y010;
   wire [`SIG1_DIM0 - 1 : 0] y011;
   wire [`SIG1_DIM0 - 1 : 0] y012;
   wire [`SIG1_DIM0 - 1 : 0] y013;

   wire [`SIG1_DIM0 - 1 : 0] y100;
   wire [`SIG1_DIM0 - 1 : 0] y101;
   wire [`SIG1_DIM0 - 1 : 0] y102;
   wire [`SIG1_DIM0 - 1 : 0] y103;

   wire [`SIG1_DIM0 - 1 : 0] y110;
   wire [`SIG1_DIM0 - 1 : 0] y111;
   wire [`SIG1_DIM0 - 1 : 0] y112;
   wire [`SIG1_DIM0 - 1 : 0] y113;

   assign y0 = {y000, y001, y002, y003};
   assign y1 = {y010, y011, y012, y013};
   assign y2 = {y100, y101, y102, y103};
   assign y3 = {y110, y111, y112, y113};

   assign y_in = {y0, y1, y2, y3};

endmodule
```

int *signal_object_name*.**get_dim_width**(*dimension_number*);

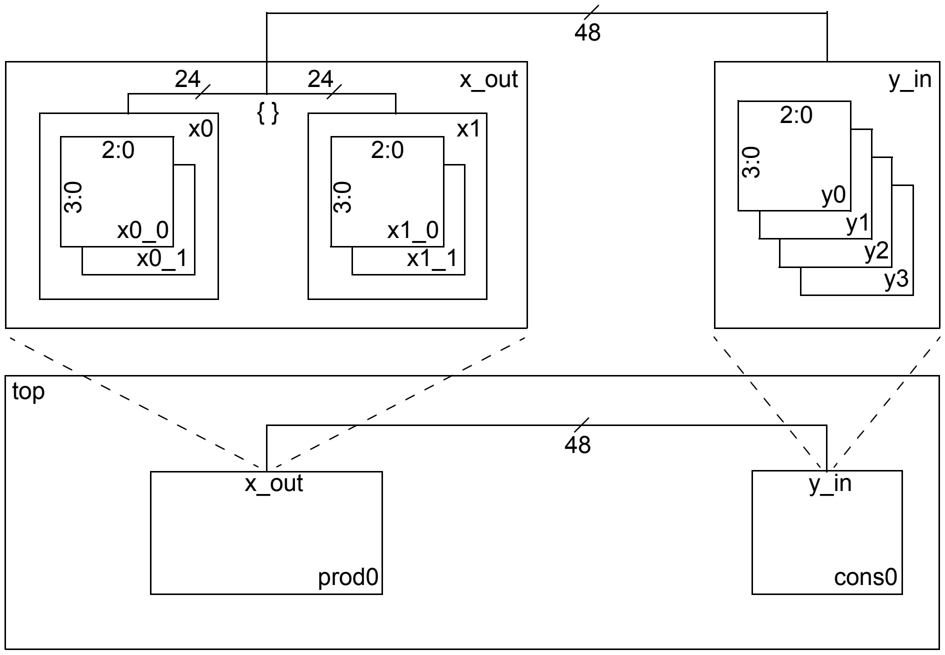
DESCRIPTION :

Returns the *dimension_number* dimension width of a multi dimensional signal.

EXAMPLE :

Following the example from the set_dim_width() method, the usage of get_dim_width() becomes intuitive.

FIGURE 1.6 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//each dimension width is set
x_out.set_dim_width(0,3);
x_out.set_dim_width(1,4);
x_out.set_dim_width(2,2);
//the dimension is set using the values from another dimension
}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//set_dim_width() method with get_dim_width() method




}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0 3
`define SIG1_DIM1 4
`define SIG1_DIM2 2
`define SIG1_DIM3 2
`define SIG1_DIM_MAX `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2*`SIG1_DIM3
`define SIG1_DIM_MED `SIG1_DIM0*`SIG1_DIM1*`SIG1_DIM2
`define SIG1_DIM_MIN `SIG1_DIM0*`SIG1_DIM1

`define SIG2_DIM0 3
`define SIG2_DIM1 4
`define SIG2_DIM2 4
`define SIG2_DIM_MAX `SIG2_DIM0*`SIG2_DIM1*`SIG2_DIM2
`define SIG2_DIM_MIN `SIG2_DIM0*`SIG2_DIM1

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
  prod prod0(.x_out(trans));
  cons cons0(.y_in(trans));
endmodule

module prod(x_out);
   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0 ;
   wire [`SIG1_DIM_MIN-1:0] x0_1 ;
   wire [`SIG1_DIM_MIN-1:0] x1_0 ;
   wire [`SIG1_DIM_MIN-1:0] x1_1 ;

   wire [`SIG1_DIM0 - 1 : 0] x000;
   wire [`SIG1_DIM0 - 1 : 0] x001;
   wire [`SIG1_DIM0 - 1 : 0] x002;
   wire [`SIG1_DIM0 - 1 : 0] x003;

   wire [`SIG1_DIM0 - 1 : 0] x010;
   wire [`SIG1_DIM0 - 1 : 0] x011;
   wire [`SIG1_DIM0 - 1 : 0] x012;
   wire [`SIG1_DIM0 - 1 : 0] x013;

   wire [`SIG1_DIM0 - 1 : 0] x100;
   wire [`SIG1_DIM0 - 1 : 0] x101;
   wire [`SIG1_DIM0 - 1 : 0] x102;
   wire [`SIG1_DIM0 - 1 : 0] x103;

   wire [`SIG1_DIM0 - 1 : 0] x110;
   wire [`SIG1_DIM0 - 1 : 0] x111;
   wire [`SIG1_DIM0 - 1 : 0] x112;
   wire [`SIG1_DIM0 - 1 : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x0 = {x0_0, x0_1};
   assign x1 = {x1_0, x1_1};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
   input [`SIG2_DIM_MAX-1:0] y_in;
   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;

   wire [`SIG1_DIM0 - 1 : 0] y000;
   wire [`SIG1_DIM0 - 1 : 0] y001;
   wire [`SIG1_DIM0 - 1 : 0] y002;
   wire [`SIG1_DIM0 - 1 : 0] y003;

   wire [`SIG1_DIM0 - 1 : 0] y010;
   wire [`SIG1_DIM0 - 1 : 0] y011;
   wire [`SIG1_DIM0 - 1 : 0] y012;
   wire [`SIG1_DIM0 - 1 : 0] y013;

   wire [`SIG1_DIM0 - 1 : 0] y100;
   wire [`SIG1_DIM0 - 1 : 0] y101;
   wire [`SIG1_DIM0 - 1 : 0] y102;
   wire [`SIG1_DIM0 - 1 : 0] y103;

   wire [`SIG1_DIM0 - 1 : 0] y110;
   wire [`SIG1_DIM0 - 1 : 0] y111;
   wire [`SIG1_DIM0 - 1 : 0] y112;
   wire [`SIG1_DIM0 - 1 : 0] y113;

   assign y0 = {y000, y001, y002, y003};
   assign y1 = {y010, y011, y012, y013};
   assign y2 = {y100, y101, y102, y103};
   assign y3 = {y110, y111, y112, y113};

   assign y_in = {y0, y1, y2, y3};

endmodule
```

signal_object_name.**set_dim_bitrange**(*dimension_number*,*bitrange_object_name*);

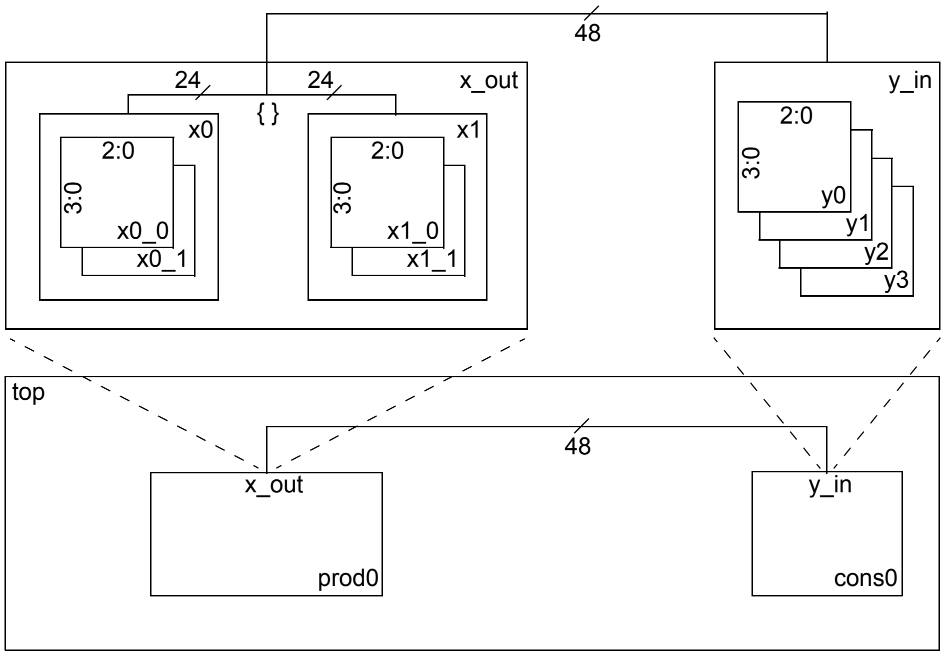
DESCRIPTION :

Set the bitrange for the single dimension *dimension_number* of a multi dimensional signal. If the signal is not multidimensional then a compile time error is generated.

EXAMPLE :

An example of a 4 dimensional signal is shown below. The signal is 4 x 3 x 2 x 2 or [3:0] x [2:0] x [1:0] x [1:0]

FIGURE 1.7 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//declaring some bitranges to use with each dimension
csl_bitrange br1(3); //this is equivalent to [2:0]
csl_bitrange br2(3,0); //this is equivalent to [3:0]
csl_bitrange br3(2);
//each dimension bitrange is set
}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//set_dim_bitrange() used along with get_dim_bitrange()




}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
//set the bitrange to 2:0 (or width to 3)
`define SIG1_DIM0_BRUP 2
`define SIG1_DIM0_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG1_DIM1_BRUP 3
`define SIG1_DIM1_BRLOW 0

//set the bitrange to 1:0 (or width to 2)
`define SIG1_DIM2_BRUP 1
`define SIG1_DIM2_BRLOW 0

//set the bitrange to 1:0 (or width to 2)
`define SIG1_DIM3_BRUP 1
 `define SIG1_DIM3_BRLOW 0
`define SIG1_DIM_MAX (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-
`SIG1_DIM2_BRLOW+1)*(`SIG1_DIM3_BRUP-`SIG1_DIM3_BRLOW+1)
```

```
`define SIG1_DIM_MED (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-`SIG1_DIM2_BRLOW+1)
`define SIG1_DIM_MIN (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-`SIG1_DIM1_BRLOW+1)
`define SIG1_DIM (`SIG1_DIM0_BRUP-`SIG1_DIM0_BRLOW)


//set the bitrange to 2:0 (or width to 3)
`define SIG2_DIM0_BRUP 2
`define SIG2_DIM0_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG2_DIM1_BRUP 3
`define SIG2_DIM1_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG2_DIM2_BRUP 3
`define SIG2_DIM2_BRLOW 0
`define SIG2_DIM_MAX (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-
`SIG2_DIM1_BRLOW+1)*(`SIG2_DIM2_BRUP-`SIG2_DIM2_BRLOW+1)
`define SIG2_DIM_MIN (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-`SIG2_DIM1_BRLOW+1)
`define SIG2_DIM (`SIG2_DIM0_BRUP-`SIG2_DIM0_BRLOW)

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
 prod prod0(.x_out(trans));
 cons cons0(.y_in(trans));
endmodule

module prod(x_out);

   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0;
   wire [`SIG1_DIM_MIN-1:0] x0_1;
   wire [`SIG1_DIM_MIN-1:0] x1_0;
   wire [`SIG1_DIM_MIN-1:0] x1_1;

   wire [`SIG1_DIM : 0] x000;
   wire [`SIG1_DIM : 0] x001;
   wire [`SIG1_DIM : 0] x002;
   wire [`SIG1_DIM : 0] x003;

   wire [`SIG1_DIM : 0] x010;
   wire [`SIG1_DIM : 0] x011;
   wire [`SIG1_DIM : 0] x012;
   wire [`SIG1_DIM : 0] x013;

   wire [`SIG1_DIM : 0] x100;
   wire [`SIG1_DIM : 0] x101;
   wire [`SIG1_DIM : 0] x102;
   wire [`SIG1_DIM : 0] x103;

   wire [`SIG1_DIM : 0] x110;
   wire [`SIG1_DIM : 0] x111;
   wire [`SIG1_DIM : 0] x112;
   wire [`SIG1_DIM : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
input [`SIG2_DIM_MAX-1:0] y_in;

wire [`SIG2_DIM_MIN-1:0] y0;
wire [`SIG2_DIM_MIN-1:0] y1;
wire [`SIG2_DIM_MIN-1:0] y2;
wire [`SIG2_DIM_MIN-1:0] y3;

wire [`SIG2_DIM : 0] y000;
wire [`SIG2_DIM : 0] y001;
wire [`SIG2_DIM : 0] y002;
wire [`SIG2_DIM : 0] y003;

wire [`SIG2_DIM : 0] y010;
wire [`SIG2_DIM : 0] y011;
wire [`SIG2_DIM : 0] y012;
wire [`SIG2_DIM : 0] y013;

wire [`SIG2_DIM : 0] y100;
wire [`SIG2_DIM : 0] y101;
wire [`SIG2_DIM : 0] y102;
wire [`SIG2_DIM : 0] y103;

wire [`SIG2_DIM : 0] y110;
wire [`SIG2_DIM : 0] y111;
wire [`SIG2_DIM : 0] y112;
wire [`SIG2_DIM : 0] y113;

assign y0 = {y000, y001, y002, y003};
assign y1 = {y010, y011, y012, y013};
assign y2 = {y100, y101, y102, y103};
assign y3 = {y110, y111, y112, y113};

assign y_in = {y0, y1, y2, y3};

endmodule
```

```
bitrange_object
signal_object_name.get_dim_bitrange(dimension_number);
```

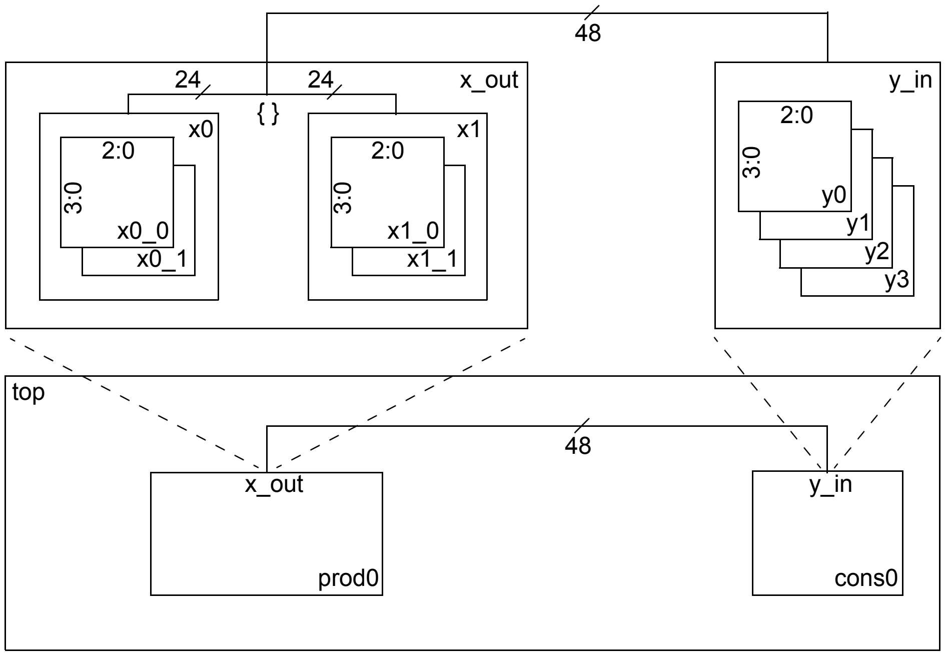
DESCRIPTION :

Returns the bitrange for the single dimension *dimension_number* of a multi dimensional signal. If the signal is not multidimensional then a compile time error is generated.

EXAMPLE :

small description of the example.

FIGURE 1.8 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//declaring some bitranges to use with each dimension
csl_bitrange br1(3); //this is equivalent to [2:0]
csl_bitrange br2(3,0); //this is equivalent to [3:0]
csl_bitrange br3(2);
//each dimension bitrange is set
x_out.set_dim_bitrange(0,br1);
x_out.set_dim_bitrange(1,br2);
x_out.set_dim_bitrange(2,br3);
//get_dim_bitrange() used to set a bitrange

}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//set_dim_bitrange() used along with get_dim_bitrange()




}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
//for signals in unit prod
//set the bitrange to 2:0 (or width to 3)
`define SIG1_DIM0_BRUP 2
`define SIG1_DIM0_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG1_DIM1_BRUP 3
`define SIG1_DIM1_BRLOW 0

//set the bitrange to 1:0 (or width to 2)
`define SIG1_DIM2_BRUP 1
`define SIG1_DIM2_BRLOW 0

//set the bitrange to 1:0 (or width to 2)
`define SIG1_DIM3_BRUP 1
`define SIG1_DIM3_BRLOW 0

`define SIG1_DIM_MAX (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-
`SIG1_DIM2_BRLOW+1)*(`SIG1_DIM3_BRUP-`SIG1_DIM3_BRLOW+1)
`define SIG1_DIM_MED (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-`SIG1_DIM2_BRLOW+1)
`define SIG1_DIM_MIN (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-`SIG1_DIM1_BRLOW+1)
`define SIG1_DIM (`SIG1_DIM0_BRUP-`SIG1_DIM0_BRLOW)


//for signals in unit cons
//set the bitrange to 2:0 (or width to 3)
`define SIG2_DIM0_BRUP 2
`define SIG2_DIM0_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG2_DIM1_BRUP 3
`define SIG2_DIM1_BRLOW 0

//set the bitrange to 3:0 (or width to 4)
`define SIG2_DIM2_BRUP 3
`define SIG2_DIM2_BRLOW 0
`define SIG2_DIM_MAX (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-
`SIG2_DIM1_BRLOW+1)*(`SIG2_DIM2_BRUP-`SIG2_DIM2_BRLOW+1)
`define SIG2_DIM_MIN (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-`SIG2_DIM1_BRLOW+1)
`define SIG2_DIM (`SIG2_DIM0_BRUP-`SIG2_DIM0_BRLOW)

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
 prod prod0(.x_out(trans));
 cons cons0(.y_in(trans));
endmodule

module prod(x_out);

   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;
   wire [`SIG1_DIM_MIN-1:0] x0_0;
   wire [`SIG1_DIM_MIN-1:0] x0_1;
   wire [`SIG1_DIM_MIN-1:0] x1_0;
   wire [`SIG1_DIM_MIN-1:0] x1_1;

   wire [`SIG1_DIM : 0] x000;
   wire [`SIG1_DIM : 0] x001;
   wire [`SIG1_DIM : 0] x002;
   wire [`SIG1_DIM : 0] x003;

   wire [`SIG1_DIM : 0] x010;
   wire [`SIG1_DIM : 0] x011;
   wire [`SIG1_DIM : 0] x012;
   wire [`SIG1_DIM : 0] x013;

   wire [`SIG1_DIM : 0] x100;
   wire [`SIG1_DIM : 0] x101;
   wire [`SIG1_DIM : 0] x102;
   wire [`SIG1_DIM : 0] x103;

   wire [`SIG1_DIM : 0] x110;
   wire [`SIG1_DIM : 0] x111;
   wire [`SIG1_DIM : 0] x112;
   wire [`SIG1_DIM : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
input [`SIG2_DIM_MAX-1:0] y_in;

wire [`SIG2_DIM_MIN-1:0] y0;
wire [`SIG2_DIM_MIN-1:0] y1;
wire [`SIG2_DIM_MIN-1:0] y2;
wire [`SIG2_DIM_MIN-1:0] y3;

wire [`SIG2_DIM : 0] y000;
wire [`SIG2_DIM : 0] y001;
wire [`SIG2_DIM : 0] y002;
wire [`SIG2_DIM : 0] y003;

wire [`SIG2_DIM : 0] y010;
wire [`SIG2_DIM : 0] y011;
wire [`SIG2_DIM : 0] y012;
wire [`SIG2_DIM : 0] y013;

wire [`SIG2_DIM : 0] y100;
wire [`SIG2_DIM : 0] y101;
wire [`SIG2_DIM : 0] y102;
wire [`SIG2_DIM : 0] y103;

wire [`SIG2_DIM : 0] y110;
wire [`SIG2_DIM : 0] y111;
wire [`SIG2_DIM : 0] y112;
wire [`SIG2_DIM : 0] y113;

assign y0 = {y000, y001, y002, y003};
assign y1 = {y010, y011, y012, y013};
assign y2 = {y100, y101, y102, y103};
assign y3 = {y110, y111, y112, y113};

assign y_in = {y0, y1, y2, y3};

endmodule
```

signal_object_name.**set_dim_range**(*dimension_number*,*upper_limit*, *lower_limit*);

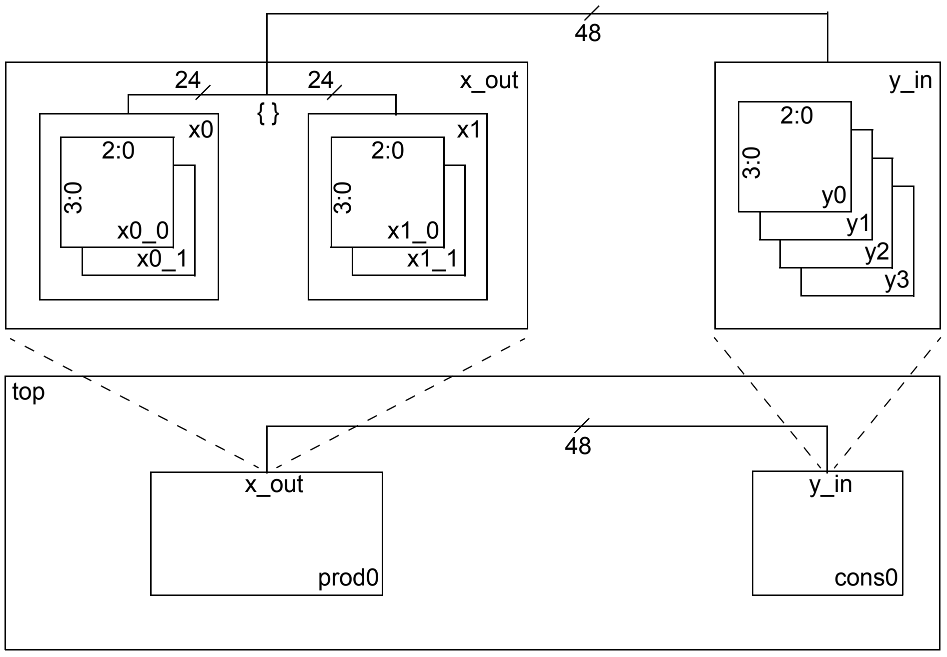
DESCRIPTION :

Set the range for the dimension *dimension_number* of *signal_object_name* using the *upper_limit* and *lower_limit* delimiters.This method does not have a corresponding get() method

EXAMPLE :

Similar to the bitrange command, except that in CSL the bitrange methods operate on objects with objects, while the range methods operate on objects with parameters

FIGURE 1.9 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//each dimension range is set
}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//using a set method with a get method







}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0_BRUP 2
`define SIG1_DIM0_BRLOW 0 //set the bitrange to 2:0 (or width to 3)

`define SIG1_DIM1_BRUP 3
`define SIG1_DIM1_BRLOW 0 //set the bitrange to 3:0 (or width to 4)

`define SIG1_DIM2_BRUP 1
`define SIG1_DIM2_BRLOW 0 //set the bitrange to 1:0 (or width to 2)

`define SIG1_DIM3_BRUP 1
`define SIG1_DIM3_BRLOW 0 //set the bitrange to 1:0 (or width to 2)
`define SIG1_DIM_MAX (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-
`SIG1_DIM2_BRLOW+1)*(`SIG1_DIM3_BRUP-`SIG1_DIM3_BRLOW+1)
`define SIG1_DIM_MED (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-
`SIG1_DIM1_BRLOW+1)*(`SIG1_DIM2_BRUP-`SIG1_DIM2_BRLOW+1)
`define SIG1_DIM_MIN (`SIG1_DIM0_BRUP-
`SIG1_DIM0_BRLOW+1)*(`SIG1_DIM1_BRUP-`SIG1_DIM1_BRLOW+1)
`define SIG1_DIM (`SIG1_DIM0_BRUP-`SIG1_DIM0_BRLOW)
```

```
`define SIG2_DIM0_BRUP 2
`define SIG2_DIM0_BRLOW 0 //set the bitrange to 2:0 (or width to 3)

`define SIG2_DIM1_BRUP 3
`define SIG2_DIM1_BRLOW 0 //set the bitrange to 3:0 (or width to 4)

`define SIG2_DIM2_BRUP 3
`define SIG2_DIM2_BRLOW 0 //set the bitrange to 3:0 (or width to 4)
`define SIG2_DIM_MAX (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-
`SIG2_DIM1_BRLOW+1)*(`SIG2_DIM2_BRUP-`SIG2_DIM2_BRLOW+1)
`define SIG2_DIM_MIN (`SIG2_DIM0_BRUP-
`SIG2_DIM0_BRLOW+1)*(`SIG2_DIM1_BRUP-`SIG2_DIM1_BRLOW+1)
`define SIG2_DIM (`SIG2_DIM0_BRUP-`SIG2_DIM0_BRLOW)

module top();
  wire [`SIG1_DIM_MAX-1:0] trans;
 prod prod0(.x_out(trans));
 cons cons0(.y_in(trans));
endmodule

module prod(x_out);

   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [`SIG1_DIM_MIN-1:0] x0_0;
   wire [`SIG1_DIM_MIN-1:0] x0_1;
   wire [`SIG1_DIM_MIN-1:0] x1_0;
   wire [`SIG1_DIM_MIN-1:0] x1_1;

   wire [`SIG1_DIM : 0] x000;
   wire [`SIG1_DIM : 0] x001;
   wire [`SIG1_DIM : 0] x002;
   wire [`SIG1_DIM : 0] x003;

   wire [`SIG1_DIM : 0] x010;
   wire [`SIG1_DIM : 0] x011;
   wire [`SIG1_DIM : 0] x012;
   wire [`SIG1_DIM : 0] x013;

   wire [`SIG1_DIM : 0] x100;
   wire [`SIG1_DIM : 0] x101;
   wire [`SIG1_DIM : 0] x102;
   wire [`SIG1_DIM : 0] x103;

   wire [`SIG1_DIM : 0] x110;
   wire [`SIG1_DIM : 0] x111;
   wire [`SIG1_DIM : 0] x112;
   wire [`SIG1_DIM : 0] x113;

   assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
input [`SIG2_DIM_MAX-1:0] y_in;

wire [`SIG2_DIM_MIN-1:0] y0;
wire [`SIG2_DIM_MIN-1:0] y1;
wire [`SIG2_DIM_MIN-1:0] y2;
wire [`SIG2_DIM_MIN-1:0] y3;

wire [`SIG2_DIM : 0] y000;
wire [`SIG2_DIM : 0] y001;
wire [`SIG2_DIM : 0] y002;
wire [`SIG2_DIM : 0] y003;

wire [`SIG2_DIM : 0] y010;
wire [`SIG2_DIM : 0] y011;
wire [`SIG2_DIM : 0] y012;
wire [`SIG2_DIM : 0] y013;

wire [`SIG2_DIM : 0] y100;
wire [`SIG2_DIM : 0] y101;
wire [`SIG2_DIM : 0] y102;
wire [`SIG2_DIM : 0] y103;

wire [`SIG2_DIM : 0] y110;
wire [`SIG2_DIM : 0] y111;
wire [`SIG2_DIM : 0] y112;
wire [`SIG2_DIM : 0] y113;

assign y0 = {y000, y001, y002, y003};
assign y1 = {y010, y011, y012, y013};
assign y2 = {y100, y101, y102, y103};
assign y3 = {y110, y111, y112, y113};

assign y_in = {y0, y1, y2, y3};

endmodule
```

int *signal_object_name*.**get_dim_lower_index**(*dimension_number*);

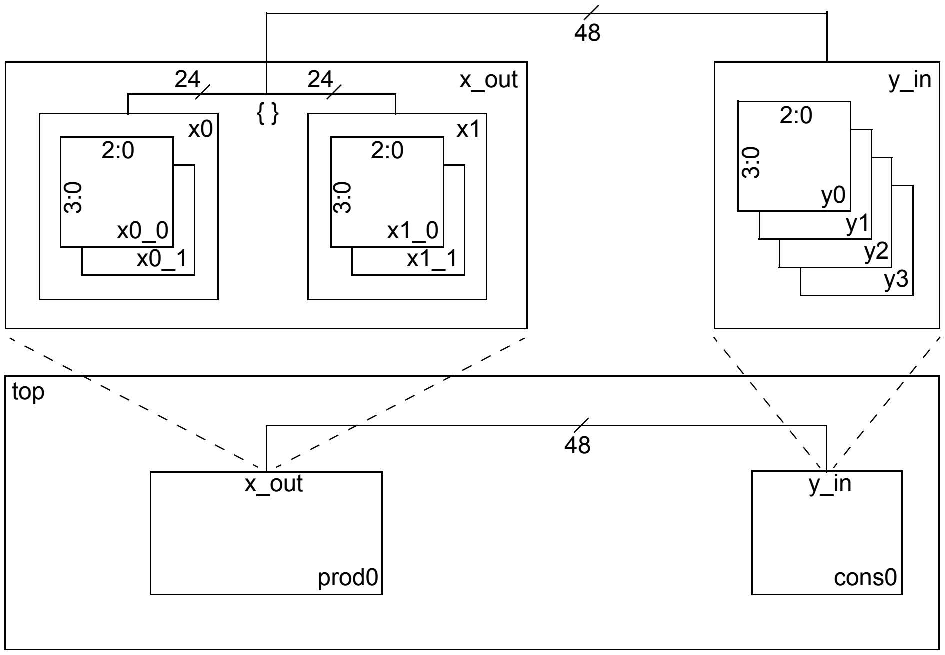
DESCRIPTION :

Returns the lower index value of the dimension *dimension_number* for a multidimensional signal.

EXAMPLE :

small description of the example.

FIGURE 1.10 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//for each dimension the lower index is set
x_out.set_dim_lower_index(0,0);
x_out.set_dim_upper_index(0,2);
x_out.set_dim_lower_index(1,0);
x_out.set_dim_upper_index(1,3);
x_out.set_dim_lower_index(2,0);
x_out.set_dim_upper_index(2,1);
//using the get() method for objects in the same scope

x_out.set_dim_upper_index(3,get_dim_upper_index(2));
}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//using the get() method for objects in different scopes

y_in.set_dim_upper_index(0,prod.x_out.get_dim_upper_index(0));

y_in.set_dim_upper_index(1,prod.x_out.get_dim_upper_index(1));




}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0_UPIDX 2
`define SIG1_DIM0_LOWIDX 0
//set the lower index to 0

`define SIG1_DIM1_UPIDX 3
`define SIG1_DIM1_LOWIDX 0
//set the lower index to 0

`define SIG1_DIM2_UPIDX 1
`define SIG1_DIM2_LOWIDX 0
//set the lower index to 0

`define SIG1_DIM3_UPIDX 1
`define SIG1_DIM3_LOWIDX 0
//set the lower index to 0
`define SIG1_DIM_MAX (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-
`SIG1_DIM1_LOWIDX+1)*(`SIG1_DIM2_UPIDX-
`SIG1_DIM2_LOWIDX+1)*(`SIG1_DIM3_UPIDX-`SIG1_DIM3_LOWIDX+1)
`define SIG1_DIM_MED (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-
`SIG1_DIM1_LOWIDX+1)*(`SIG1_DIM2_UPIDX-`SIG1_DIM2_LOWIDX+1)
`define SIG1_DIM_MIN (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-`SIG1_DIM1_LOWIDX+1)
`define SIG1_DIM `SIG1_DIM0_UPIDX-`SIG1_DIM0_LOWIDX



`define SIG2_DIM0_UPIDX 2
`define SIG2_DIM0_LOWIDX 0
//set the lower index to 0

`define SIG2_DIM1_UPIDX 3
`define SIG2_DIM1_LOWIDX 0
//set the lower index to 0

`define SIG2_DIM2_UPIDX 3
`define SIG2_DIM2_LOWIDX 0
//set the lower index to 0
`define SIG2_DIM_MAX (`SIG2_DIM0_UPIDX-
`SIG2_DIM0_LOWIDX+1)*(`SIG2_DIM1_UPIDX-
`SIG2_DIM1_LOWIDX+1)*(`SIG2_DIM2_UPIDX-`SIG2_DIM2_LOWIDX+1)
`define SIG2_DIM_MIN (`SIG2_DIM0_UPIDX-
`SIG2_DIM0_LOWIDX+1)*(`SIG2_DIM1_UPIDX-`SIG2_DIM1_LOWIDX+1)
`define SIG2_DIM `SIG1_DIM0_UPIDX-`SIG1_DIM0_LOWIDX

module top();
 wire [`SIG1_DIM_MAX-1:0] trans;
prod prod0(.x_out(trans));
cons cons0(.y_in(trans));
endmodule

module prod(x_out);

   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;
   wire [` SIG1_DIM_MIN-1:0] x0_0;
   wire [` SIG1_DIM_MIN-1:0] x0_1;
   wire [` SIG1_DIM_MIN-1:0] x1_0;
   wire [` SIG1_DIM_MIN-1:0] x1_1;

   wire [`SIG1_DIM : 0] x000;
   wire [`SIG1_DIM : 0] x001;
   wire [`SIG1_DIM : 0] x002;
   wire [`SIG1_DIM : 0] x003;

   wire [`SIG1_DIM : 0] x010;
   wire [`SIG1_DIM : 0] x011;
   wire [`SIG1_DIM : 0] x012;
   wire [`SIG1_DIM : 0] x013;

   wire [`SIG1_DIM : 0] x100;
   wire [`SIG1_DIM : 0] x101;
   wire [`SIG1_DIM : 0] x102;
   wire [`SIG1_DIM : 0] x103;

   wire [`SIG1_DIM : 0] x110;
   wire [`SIG1_DIM : 0] x111;
   wire [`SIG1_DIM : 0] x112;
   wire [`SIG1_DIM : 0] x113;

assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
   input [`SIG2_DIM_MAX-1:0] y_in;

   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;
```

```
    wire [`SIG2_DIM : 0] y000;
    wire [`SIG2_DIM : 0] y001;
    wire [`SIG2_DIM : 0] y002;
    wire [`SIG2_DIM : 0] y003;

    wire [`SIG2_DIM : 0] y010;
    wire [`SIG2_DIM : 0] y011;
    wire [`SIG2_DIM : 0] y012;
    wire [`SIG2_DIM : 0] y013;

    wire [`SIG2_DIM : 0] y100;
    wire [`SIG2_DIM : 0] y101;
    wire [`SIG2_DIM : 0] y102;
    wire [`SIG2_DIM : 0] y103;

    wire [`SIG2_DIM : 0] y110;
    wire [`SIG2_DIM : 0] y111;
    wire [`SIG2_DIM : 0] y112;
    wire [`SIG2_DIM : 0] y113;

    assign y0 = {y000, y001, y002, y003};
    assign y1 = {y010, y011, y012, y013};
    assign y2 = {y100, y101, y102, y103};
    assign y3 = {y110, y111, y112, y113};

    assign y_in = {y0, y1, y2, y3};

endmodule
```

int *signal_object_name*.**get_dim_upper_index**(*dimension_number*);

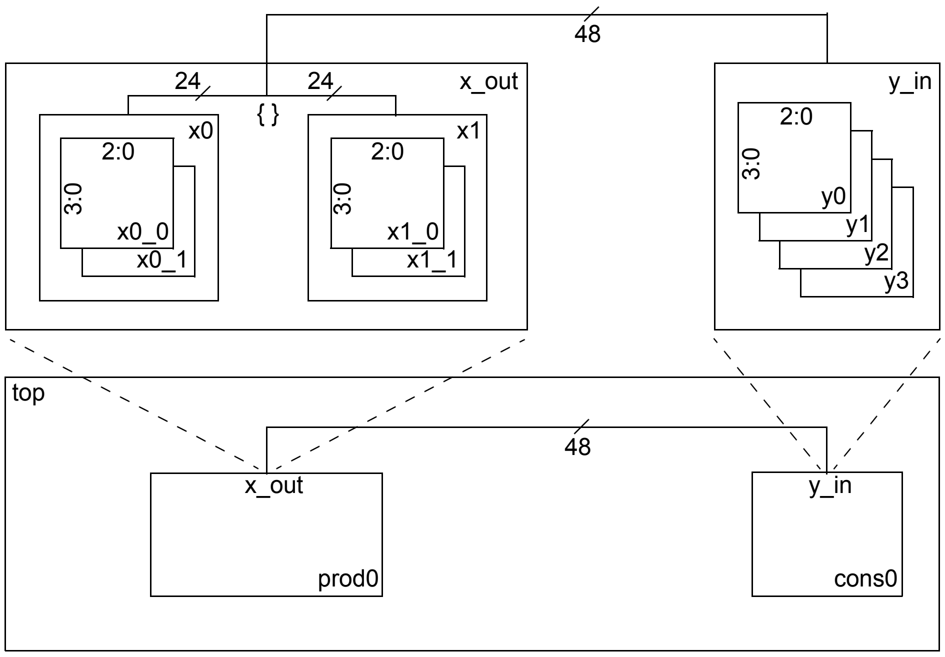
DESCRIPTION :

Return the upper index value for the multi dimensional signal *signal_object_name*. If this method is called on a single dimensional signal object an error is generated

EXAMPLE :

small description of the example.

FIGURE 1.11 Multidimensional signals setup



CSL CODE

```
//AB
csl_unit top, prod, cons;
prod.add_signal(wire,x_out);
scope prod{
//signal x_out of unit prod is set to 4 dimensions
x_out.set_number_of_dimensions(4);
//for each dimension the lower index is set
x_out.set_dim_lower_index(0,0);
x_out.set_dim_upper_index(0,2);
x_out.set_dim_lower_index(1,0);
x_out.set_dim_upper_index(1,3);
x_out.set_dim_lower_index(2,0);
x_out.set_dim_upper_index(2,1);
//using the set() method with a get() method in the same scope
x_out.set_dim_lower_index(3,get_dim_lower_index(2));

}
cons.add_signal(wire,y_in);
scope cons{
y_in.set_number_of_dimensions(3);
//using the set() method with a get() method in different scopes
y_in.set_dim_lower_index(0,prod.x_out.get_dim_lower_index(0));

y_in.set_dim_lower_index(1,prod.x_out.get_dim_lower_index(1));

y_in.set_dim_lower_index(2,prod.x_out.get_dim_lower_index(2)+prod.x_ou
t.get_dim_lower_index(3));


}
top.add_instance(prod, prod0);
top.add_instance(cons, cons0);
prod0.x_out.connect(cons0.y_in);
```

VERILOG CODE

```
//AB + AV
`define SIG1_DIM0_UPIDX 2
`define SIG1_DIM0_LOWIDX 0

`define SIG1_DIM1_UPIDX 3
`define SIG1_DIM1_LOWIDX 0

`define SIG1_DIM2_UPIDX 1
`define SIG1_DIM2_LOWIDX 0

`define SIG1_DIM3_UPIDX 1
`define SIG1_DIM3_LOWIDX 0

`define SIG1_DIM_MAX (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-
`SIG1_DIM1_LOWIDX+1)*(`SIG1_DIM2_UPIDX-
`SIG1_DIM2_LOWIDX+1)*(`SIG1_DIM3_UPIDX-`SIG1_DIM3_LOWIDX+1)
```

```
`define SIG1_DIM_MED (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-
`SIG1_DIM1_LOWIDX+1)*(`SIG1_DIM2_UPIDX-`SIG1_DIM2_LOWIDX+1)
`define SIG1_DIM_MIN (`SIG1_DIM0_UPIDX-
`SIG1_DIM0_LOWIDX+1)*(`SIG1_DIM1_UPIDX-`SIG1_DIM1_LOWIDX+1)
`define SIG1_DIM `SIG1_DIM0_UPIDX-`SIG1_DIM0_LOWIDX



`define SIG2_DIM0_UPIDX 2
`define SIG2_DIM0_LOWIDX 0

`define SIG2_DIM1_UPIDX 3
`define SIG2_DIM1_LOWIDX 0

`define SIG2_DIM2_UPIDX 3
`define SIG2_DIM2_LOWIDX 0

`define SIG2_DIM_MAX (`SIG2_DIM0_UPIDX-
`SIG2_DIM0_LOWIDX+1)*(`SIG2_DIM1_UPIDX-
`SIG2_DIM1_LOWIDX+1)*(`SIG2_DIM2_UPIDX-`SIG2_DIM2_LOWIDX+1)
`define SIG2_DIM_MIN (`SIG2_DIM0_UPIDX-
`SIG2_DIM0_LOWIDX+1)*(`SIG2_DIM1_UPIDX-`SIG2_DIM1_LOWIDX+1)
`define SIG2_DIM `SIG1_DIM0_UPIDX-`SIG1_DIM0_LOWIDX

module top();
 wire [`SIG1_DIM_MAX-1:0] trans;
prod prod0(.x_out(trans));
cons cons0(.y_in(trans));
endmodule

module prod(x_out);

   output [`SIG1_DIM_MAX-1:0] x_out;

   wire [`SIG1_DIM_MED-1:0] x0;
   wire [`SIG1_DIM_MED-1:0] x1;

   wire [` SIG1_DIM_MIN-1:0] x0_0;
   wire [` SIG1_DIM_MIN-1:0] x0_1;
   wire [` SIG1_DIM_MIN-1:0] x1_0;
   wire [` SIG1_DIM_MIN-1:0] x1_1;
   wire [`SIG1_DIM : 0] x000;
   wire [`SIG1_DIM : 0] x001;
   wire [`SIG1_DIM : 0] x002;
   wire [`SIG1_DIM : 0] x003;

   wire [`SIG1_DIM : 0] x010;
   wire [`SIG1_DIM : 0] x011;
   wire [`SIG1_DIM : 0] x012;
   wire [`SIG1_DIM : 0] x013;

   wire [`SIG1_DIM : 0] x100;
   wire [`SIG1_DIM : 0] x101;
   wire [`SIG1_DIM : 0] x102;
   wire [`SIG1_DIM : 0] x103;

   wire [`SIG1_DIM : 0] x110;
   wire [`SIG1_DIM : 0] x111;
   wire [`SIG1_DIM : 0] x112;
   wire [`SIG1_DIM : 0] x113;

assign x0_0 = {x000, x001, x002, x003};
   assign x0_1 = {x010, x011, x012, x013};
   assign x1_0 = {x100, x101, x102, x103};
   assign x1_1 = {x110, x111, x112, x113};

   assign x_out = {x0,x1};
endmodule

module cons(y_in);
   input [`SIG2_DIM_MAX-1:0] y_in;

   wire [`SIG2_DIM_MIN-1:0] y0;
   wire [`SIG2_DIM_MIN-1:0] y1;
   wire [`SIG2_DIM_MIN-1:0] y2;
   wire [`SIG2_DIM_MIN-1:0] y3;

   wire [`SIG2_DIM : 0] y000;
   wire [`SIG2_DIM : 0] y001;
   wire [`SIG2_DIM : 0] y002;
   wire [`SIG2_DIM : 0] y003;

   wire [`SIG2_DIM : 0] y010;
   wire [`SIG2_DIM : 0] y011;
   wire [`SIG2_DIM : 0] y012;
   wire [`SIG2_DIM : 0] y013;

   wire [`SIG2_DIM : 0] y100;
   wire [`SIG2_DIM : 0] y101;
   wire [`SIG2_DIM : 0] y102;
   wire [`SIG2_DIM : 0] y103;

   wire [`SIG2_DIM : 0] y110;
   wire [`SIG2_DIM : 0] y111;
   wire [`SIG2_DIM : 0] y112;
   wire [`SIG2_DIM : 0] y113;

   assign y0 = {y000, y001, y002, y003};
   assign y1 = {y010, y011, y012, y013};
   assign y2 = {y100, y101, y102, y103};
   assign y3 = {y110, y111, y112, y113};

   assign y_in = {y0, y1, y2, y3};

endmodule
```

signal_object_name.**set_dim_offset**(*dimension_number*, *constant_numeric_expression*);

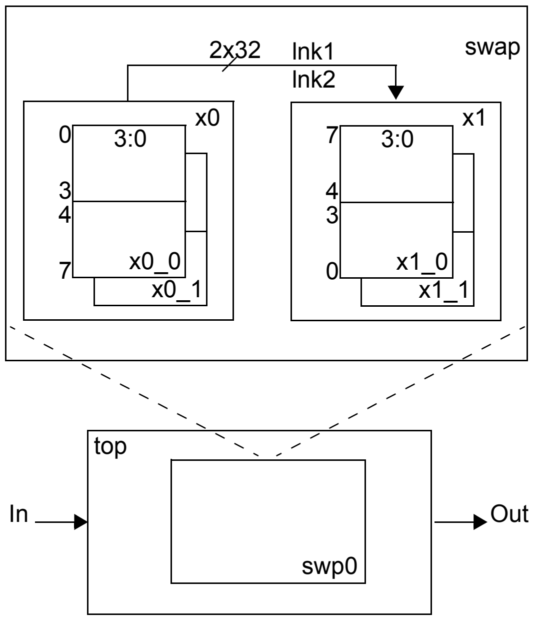
DESCRIPTION :

Set the offset value for the dimension *dimension_number* of a multidimensional signal.

EXAMPLE :

small description of the example.

FIGURE 1.12



CSL CODE

```
//AV
csl_unit top, swap,x0,x1;
scope x0 {
  csl_signal lnk1_0,lnk1_1,lnk2_0,lnk2_1;
  csl_bitrange width1(3,0);
  lnk1_0.set_number_of_dimensions(2);
  lnk1_1.set_number_of_dimensions(lnk1_0.get_number_of_dimensions());
  lnk2_0.set_number_of_dimensions(2);
  lnk2_1.set_number_of_dimensions(lnk2_0.get_number_of_dimensions());
  lnk1_0.set_dim_range(1,width1);
  lnk1_0.set_dim_range(2,width1);
  lnk1_1.set_dim_range(1,width1);
  lnk1_1.set_dim_range(2,lnk1_0.get_dim_range(2));
  // set a positive offset from the lower index
  lnk2_0.set_dim_range(1,width1);
  lnk2_0.set_dim_range(2,width1);
  lnk2_1.set_dim_range(1,width1);
  lnk2_1.set_dim_range(2,lnk2_0.get_dim_range(2));

  csl_signal lnk1,lnk2;
  lnk1.set_number_of_dimensions(2);
  lnk2.set_number_of_dimensions(2);
  lnk1.connect(lnk1_0,lnk1_1);
  lnk2.connect(lnk2_0,lnk2_1);
  add_port(output,lnk1,lnk2);
}
scope x1 {
  csl_signal lnk1_0,lnk1_1,lnk2_0,lnk2_1;
  csl_bitrange width2(7,4);
  lnk1_0.set_number_of_dimensions(2);
  lnk1_1.set_number_of_dimensions(lnk1_0.get_number_of_dimensions());
  lnk2_0.set_number_of_dimensions(2);
  lnk2_1.set_number_of_dimensions(lnk2_0.get_number_of_dimensions());
  lnk1_0.set_dim_range(1,width2);
  lnk1_0.set_dim_range(2,width2);
  lnk1_1.set_dim_range(1,width2);
  lnk1_1.set_dim_range(2,lnk1_0.get_dim_range(2));
  // set a negative offset from the upper index



  lnk2_0.set_dim_range(1,width2);
  lnk2_0.set_dim_range(2,width2);
  lnk2_1.set_dim_range(1,width2);
  lnk2_1.set_dim_range(2,lnk2_0.get_dim_range(2));



  csl_signal lnk1,lnk2;
  lnk1.set_number_of_dimensions(2);
  lnk2.set_number_of_dimensions(2);
  lnk1.connect(lnk1_1,lnk1_0);
  lnk2.connect(lnk2_0,lnk2_1);
```

```
    add_port(output,lnk1,lnk2);

  }
    csl_signal lnk1,lnk2;
  scope swap {
    add_instance(x0,x0_0);
    add_instance(x1,x1_1);
    x0.connect(.lnk1(x1.lnk1),.lnk2(x2.lnk2));
  }
  scope top {
    add_instance(swap,swp);
    add_port(input,4,in);
    add_port(output,4,out);
  }
```

VERILOG CODE

```
//AV
`define IO_DIM 4
`define LINK_DIM 32
`define HALF_DIM `LINK_DIM/2

module top(in,out);
    input [`IO_DIM-1:0] in;
    output [`IO_DIM-1:0] out;
    swap swp0();
endmodule
module swap;
    wire [`LINK_DIM-1:0] lnk1, lnk2;
    first x0(lnk1,lnk2);
    last x1(lnk1,lnk2);
endmodule
module first(lnk1,lnk2);
    output [`LINK_DIM-1:0] lnk1, lnk2;
    wire [`HALF_DIM-1:0] lnk1_0,lnk1_1,lnk2_0,lnk2_1;
    wire [`IO_DIM-1:0]
x0_000,x0_001,x0_010,x0_011,x0_100,x0_101,x0_110,x0_111;
    wire [`IO_DIM-1:0]
x1_000,x1_001,x1_010,x1_011,x1_100,x1_101,x1_110,x1_111;
    assign lnk1_0 = {x0_000,x0_001,x0_010,x0_011};
    assign lnk1_1 = {x0_100,x0_101,x0_110,x0_111};
    assign lnk2_0 = {x1_000,x1_001,x1_010,x1_011};
    assign lnk2_1 = {x1_100,x1_101,x1_110,x1_111};
    assign lnk1 = {lnk1_0,lnk1_1};
    assign lnk2 = {lnk2_0,lnk2_1};
endmodule
module last (lnk1,lnk2);
    input [31:0] lnk1,lnk2;
    wire [`HALF_DIM-1:0] lnk1_0,lnk1_1,lnk2_0,lnk2_1;
    wire [3:0]
y0_000,y0_001,y0_010,y0_011,y0_100,y0_101,y0_110,y0_111;
    wire [3:0]
y1_000,y1_001,y1_010,y1_011,y1_100,y1_101,y1_110,y1_111;
    assign lnk1_0 = {y0_011,y0_010,y0_001,y0_000};
    assign lnk1_1 = {y0_111,y0_110,y0_101,y0_100};
    assign lnk2_0 = {y1_011,y1_010,y1_001,y1_000};
    assign lnk2_1 = {y1_111,y1_110,y1_101,y1_100};
    assign lnk1 = {lnk1_1,lnk1_0};
    assign lnk1 = {lnk2_1,lnk2_0};
endmodule
```

constant_numeric_expression
signal_object_name.**get_dim_offset**(*dimension_number*);

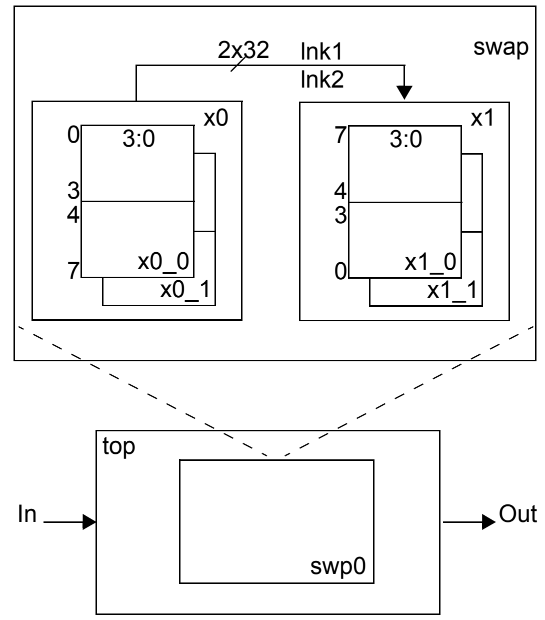
DESCRIPTION :

Return the offset value for dimension *dimension_number* of a multidimensional signal.

EXAMPLE :

small description of the example.

FIGURE 1.13



CSL CODE

```
csl_unit top, swap,x0,x1;
scope x0 {
  csl_signal lnk1_0,lnk1_1,lnk2_0,lnk2_1;
  csl_bitrange width1(3,0);
  lnk1_0.set_number_of_dimensions(2);
  lnk1_1.set_number_of_dimensions(lnk1_0.get_number_of_dimensions());
  lnk2_0.set_number_of_dimensions(2);
  lnk2_1.set_number_of_dimensions(lnk2_0.get_number_of_dimensions());
  lnk1_0.set_dim_range(1,width1);
  lnk1_0.set_dim_range(2,width1);
  lnk1_1.set_dim_range(1,width1);
  lnk1_1.set_dim_range(2,lnk1_0.get_dim_range(2));
  lnk1_1.set_dim_offset(2,4);
  lnk2_0.set_dim_range(1,width1);
  lnk2_0.set_dim_range(2,width1);
  lnk2_1.set_dim_range(1,width1);
  lnk2_1.set_dim_range(2,lnk2_0.get_dim_range(2));

  csl_signal lnk1,lnk2;
  lnk1.set_number_of_dimensions(2);
  lnk2.set_number_of_dimensions(2);
  lnk1.connect(lnk1_0,lnk1_1);
  lnk2.connect(lnk2_0,lnk2_1);
  add_port(output,lnk1,lnk2);
}
scope x1 {
  csl_signal lnk1_0,lnk1_1,lnk2_0,lnk2_1;
  csl_bitrange width2(7,4);
  lnk1_0.set_number_of_dimensions(2);
  lnk1_1.set_number_of_dimensions(lnk1_0.get_number_of_dimensions());
  lnk2_0.set_number_of_dimensions(2);
  lnk2_1.set_number_of_dimensions(lnk2_0.get_number_of_dimensions());
  lnk1_0.set_dim_range(1,width2);
  lnk1_0.set_dim_range(2,width2);
  lnk1_1.set_dim_range(1,width2);
  lnk1_1.set_dim_range(2,lnk1_0.get_dim_range(2));
  lnk1_0.set_dim_offset(1,-4);

  lnk1_1.set_dim_offset(2,-4);
  lnk2_0.set_dim_range(1,width2);
  lnk2_0.set_dim_range(2,width2);
  lnk2_1.set_dim_range(1,width2);
  lnk2_1.set_dim_range(2,lnk2_0.get_dim_range(2));
  lnk2_0.set_dim_offset(1,-4);


  csl_signal lnk1,lnk2;
  lnk1.set_number_of_dimensions(2);
  lnk2.set_number_of_dimensions(2);
  lnk1.connect(lnk1_1,lnk1_0);
  lnk2.connect(lnk2_1,lnk2_0);
  add_port(output,lnk1,lnk2);

}
```

```
  csl_signal lnk1,lnk2;
scope swap {
  add_instance(x0,x0_0);
  add_instance(x1,x1_1);
  x0.connect(.lnk1(x1.lnk1),.lnk2(x2.lnk2));
}
scope top {
  add_instance(swap,swp);
  add_port(input,4,in);
  add_port(output,4,out);
}
```

VERILOG CODE

```
`define IO_DIM 4
`define LINK_DIM 32
`define HALF_DIM `LINK_DIM/2

module top(in,out);
    input [`IO_DIM-1:0] in;
    output [`IO_DIM-1:0] out;
    swap swp0();
endmodule
module swap;
    wire [`LINK_DIM-1:0] lnk1, lnk2;
    first x0(lnk1,lnk2);
    last x1(lnk1,lnk2);
endmodule
module first(lnk1,lnk2);
    output [`LINK_DIM-1:0] lnk1, lnk2;
    wire [`HALF_DIM-1:0] lnk1_0,lnk1_1,lnk2_0,lnk2_1;
    wire [`IO_DIM-1:0]
x0_000,x0_001,x0_010,x0_011,x0_100,x0_101,x0_110,x0_111;
    wire [`IO_DIM-1:0]
x1_000,x1_001,x1_010,x1_011,x1_100,x1_101,x1_110,x1_111;
    assign lnk1_0 = {x0_000,x0_001,x0_010,x0_011};
    assign lnk1_1 = {x0_100,x0_101,x0_110,x0_111};
    assign lnk2_0 = {x1_000,x1_001,x1_010,x1_011};
    assign lnk2_1 = {x1_100,x1_101,x1_110,x1_111};
    assign lnk1 = {lnk1_0,lnk1_1};
    assign lnk2 = {lnk2_0,lnk2_1};
```

```
endmodule
module last (lnk1,lnk2);
    input [31:0] lnk1,lnk2;
    wire [`HALF_DIM-1:0] lnk1_0,lnk1_1,lnk2_0,lnk2_1;
    wire [3:0]
y0_000,y0_001,y0_010,y0_011,y0_100,y0_101,y0_110,y0_111;
    wire [3:0]
y1_000,y1_001,y1_010,y1_011,y1_100,y1_101,y1_110,y1_111;
    assign lnk1_0 = {y0_011,y0_010,y0_001,y0_000};
    assign lnk1_1 = {y0_111,y0_110,y0_101,y0_100};
    assign lnk2_0 = {y1_011,y1_010,y1_001,y1_000};
    assign lnk2_1 = {y1_111,y1_110,y1_101,y1_100};
    assign lnk1 = {lnk1_1,lnk1_0};
    assign lnk1 = {lnk2_1,lnk2_0};
endmodule
```

```
csl_list_name.set_attr(csl_signal_attribute);
```

Description :

Assign an attribute to *signal_object_name*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 1.1

TABLE 1.1 signal attributes

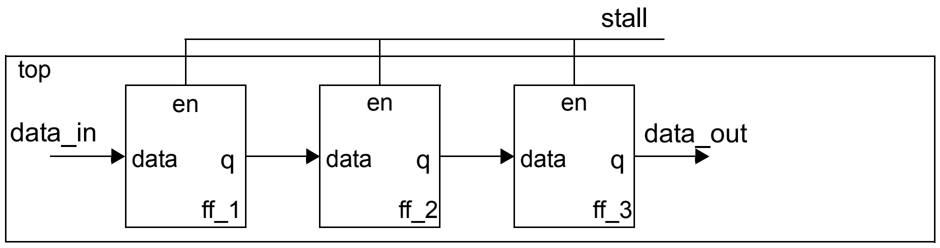
mnemonic	signal attribute
en	enable
stall	stall
pe	pipe enable
ps	pipe stall
ms	mux select
decode	decoded minterm/maxterm
clk	clock signal
rst	reset signal
wr_en	write enable

[CSL Interconnect Command Summary]

Example :

We want to interconnect two instances of a flip flop ,*ff_1* and ff2.The module *top* contains both *ff1* and *ff2* and additionally clock and enable signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 1.14



CSL CODE

```
csl_unit ff{
csl_port en1(input), clk1(input), data (input);
csl_port q(output);
ff(){}
};
csl_unit top{
ff ff1;
ff ff2;
csl_signal data_path, clock, enable, data_in, data_out;
top(){


     }};
```

VERILOG CODE

```
module ff(en1,
          clk1,
          data,
          q);
  input en1;
  input clk1;
  input data;
  output q;
endmodule
module top();
  wire data_path;
  wire clock;
  wire enable;
  wire data_in;
  wire data_out;
  ff ff1();
  ff ff2();
endmodule
```

1.1.0.2 Units: prefix

Units must have an input port with a clock attribute.The exception is if the unit has a set_type(combinational) then the unit does not have to have a clock. But the combinational must be instantiated in a design hierarchy which does have one or more clock inputs originating at the root of the design hierarchy.

set_unit_prefix(*prefix_string[,prefix_specifier]*);

DESCRIPTION :

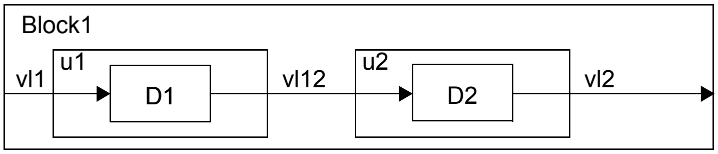
Sets the signals within the *unit_object_name* with the prefix specified by *prefix_string*. Because some signals may be bound to ports, the same prefix is applied to these ports. Optionally the user can choose to apply *prefix_object_name* only to the unit interface or it's local elements by adding the *IFC_ONLY* or *LOCAL_ONLY* prefix pecifiers . Default both specifiers are active.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the prefix " box1" for the unit *Block.*

FIGURE 1.15 An unit named *block1* with 3 instances. Then set a prefix for an unit.



CSL CODE

```
    csl_unit d{
    csl_port d_i(input), d_o(output);
    d(){ }
    };
    csl_unit u1{
    csl_port v1_i(input), v1_o(output);
    csl_signal v1,v12;
    d1 d1(.d_i(v1),.d_o(v12));
    u1(){ }
    };
    csl_unit u2{
    csl_port v2_i(input), v2_o(output);
    csl_signal v12,v2;
    d d2(.d_i(v12),.d_o(v2));
    u2(){ }
    };
    csl_unit block{
    csl_signal vl1,vl12,vl2;
    u1 u1(.v1_i(vl1),.v1_o(vl12));
    u2 u2(.v2_i(vl12),.v2_o(vl2));
    block(){
                                 ;
      }
};
```

VERILOG CODE

set_signal_prefix(*prefix_string*);

DESCRIPTION :

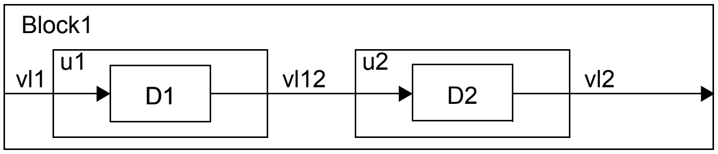
Sets the signals within the *unit_object_name* with the prefix specified by *prefix_string*. Because some signals may be bound to ports, the same prefix is applied to these ports.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a prefix to one or more signals.

FIGURE 1.16 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
                       ;
    }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
                       ;
    }
};
csl_unit block{
    csl_signal vl1,vl12,vl2;
    u1 u1(.v1_i(vl1),.v1_o(vl12));
    u2 u2(.v2_i(vl12),.v2_o(vl2));
    block(){

};
```

VERILOG CODE

set_signal_prefix_local(*prefix_string*);

DESCRIPTION :

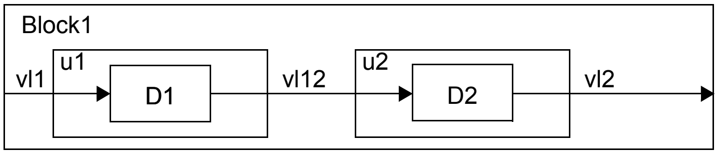
All the local signals previously declared within the same specific unit are prefixed with the *prefix_string* passed as a command argument. The difference between set_signal_prefix() and set_signal_prefix_local is that that the first command prefixes both local signals and ports (ports are signals with direction: input, output or inout, therefore **not** local), and the second command only prefixes local signals.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a local prefix "uv" to v1 and v2 signals .

FIGURE 1.17 An unit named *block1* with 3 instances. Then set a prefix for a signal.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){}
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");

  }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix("uv2");
```

```
                                                ;
          }
      };
      csl_unit block{
      csl_signal vl1,vl12,vl2;
      u1 u1(.v1_i(vl1),.v1_o(vl12));
      u2 u2(.v2_i(vl12),.v2_o(vl2));
      block(){
      set_unit_prefix("box1");
        }
  };
```

VERILOG CODE

1.1.0.3 Units: instance control bit

`set_instance_alteration_bit(``status`**`);`**

DESCRIPTION :

Set the instance alteration bit to asserted (on) or disserted (off) with the *status* enum parameter. When instance alteration is allowed (on) other objects can be added to instances. Note that this triggers a hierarchical modification down to the unit prototype the instance was derived from. When instance alteration is disallowed (off) instances cannot be modified, except by parameter override methods. Default setting for unit alteration is off (off).

[CSL Interconnect Command Summary]

EXAMPLE :

In this example the instance alteration bit is allowed (on) for the instances of unit *u.*

CSL CODE

```
csl_unit u{
csl_port x(output), y(input);
u(){
                                        ;
  }};
```

VERILOG CODE

```
module u(x,
         y);
  input y;
  output x;
endmodule
```

`set_clk_all();`

DESCRIPTION :

`Only can be used if there is one clock signal in the unit.Only one set_clk_all() function can be used per unit.`
`This function can be used only if there is one clock connected to the unit.`

[CSL Interconnect Command Summary]

EXAMPLE :

///

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

DESCRIPTION :

f

TABLE 1.2 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

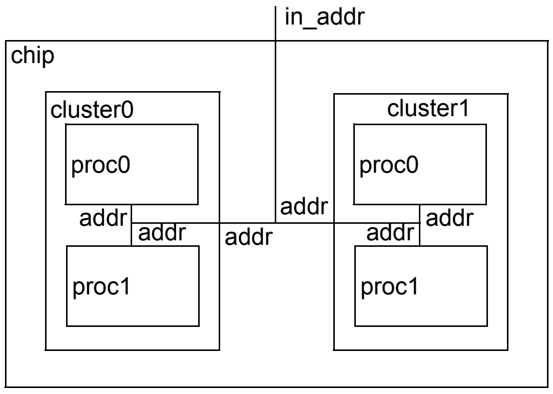
Where verilog_type is V1995, V2001, V2005, system_verilog. **output_verilog_type** will control the type of the generated verilog code . **input_verilog_type** specifies the verilog type which the design can recognize as input.(Add ex. which show the diferences between the 3 verilog types!!!!!)

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 1.18 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
```

```
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

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 :

f

TABLE 1.3 Input and output types

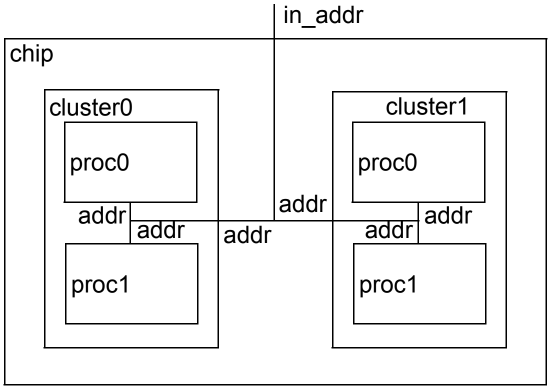
output verilog type	description
v1995	Verilog IEEE Std 1364-1995
v2001	Verilog IEEE Std 1364-2001
v2005	Verilog IEEE Std 1364-2005
sysv	IEEE 1800 SystemVerilog

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 1.19 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

//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
```

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
```

1.1.0.4 New commands

```
add_logic(external_unit_enable);
```

DESCRIPTION :

This generates an input port called *unit_name*_en. This port is an enable signal for the internal unit address decoder.

[CSL Interconnect Command Summary]

EXAMPLE :

Adds a port named *u_en.*

CSL CODE

```
    csl_unit u{
    csl_port p(input);
    u(){

    }
    };
```

VERILOG CODE

```
    module u(p,u_en);
      input p;
      input u_en;
    endmodule
```

```
(unit_name | instance_name).add_logic(unit_address_decoder,
address_signal_name);
```

DESCRIPTION :

This generates a unit address decoder which is optionally enabled by *unit_name*_en. The input to the address decoder is an input port or signal named address_signal_name. The outputs of the decoder are named *unit_name*_addr_dec_[0-2^n-1] where n is the width of the address_signal_name.

TABLE 1.4

	Unit enable required	Source
flat	no	none
virtual	no	upper address bits
hierarchical	yes	input port name

[CSL Interconnect Command Summary]

EXAMPLE :

//small description of the example

CSL CODE

```
    csl_unit u{
     u(){}
    };
    csl_unit top{
    csl_signal s(4);
    u u0;
    top(){

    }
    };
```

VERILOG CODE

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

```
signal_name.(field_name.)*add_logic(gen_decoder);
```

DESCRIPTION :

TABLE 1.5 generated Verilog decoder types

associated enum	Verilog output type
no	wire [2^n-1:0] dec = 1'b1 << sn
yes	case statement

[CSL Interconnect Command Summary]

EXAMPLE :

//small description of the example

CSL CODE

```
csl_unit u{
csl_signal sig;
u(){

}
};
```

VERILOG CODE

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

int *signal_object*.**get_width**();

DESCRIPTION :

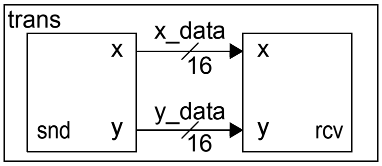
Returns the width of a single dimensional signal, otherwise it generates a cslc compile time error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 1.20 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the signals is set using *set_width* and *get_width* commands.

FIGURE 1.20 A sender and a receiver connected by two signals



CSL CODE

```
csl_unit snd{
csl_port x(output,16), y(output,16);
snd(){}
};
csl_unit rcv{
csl_port x(input,16), y(input,16);
rcv(){}
};
csl_unit trans{
csl_signal x_data, y_data;
snd snd( .x(x_data), .y(y_data));
rcv rcv( .x(x_data), .y(y_data));
trans(){
x_data.set_width(16);
y_data.set_width(                    );
  }
};
```

VERILOG CODE

```
module snd(x,y);
  output [15:0] x;
  output [15:0] y;
endmodule
module rcv(x,y);
  input [15:0] x;
  input [15:0] y;
endmodule
```

```
module trans();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x(x_data), .y(y_data));
  rcv rcv(.x(x_data), .y(y_data));
endmodule
```

```
bitrange_object signal_object.get_bitrange();
```

DESCRIPTION :

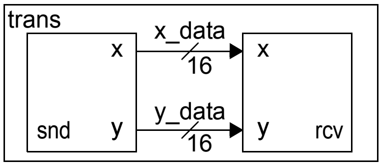
Returns a bitrange object. The return type of this function is an object of type bitrange.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 1.21 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the signals is set using *set_bitrange* and *get_bitrange* commands.

FIGURE 1.21



CSL CODE

```
csl_bitrange br1(16);
csl_unit snd{
csl_port x1(output,16), y1(output,16);
snd(){}
};
csl_unit rcv{
csl_port x2(input,16), y2(input,16);
rcv(){}
};
csl_unit trans{
csl_signal x_data, y_data;
snd snd( .x1(x_data), .y1(y_data));
rcv rcv( .x2(x_data), .y2(y_data));
top(){
x_data.set_bitrange(br1);
y_data.set_bitrange(                          );
}};
```

VERILOG CODE

```
module snd(x1,y1);
  output [15:0] x1;
  output [15:0] y1;
endmodule
module rcv(x2,y2);
  input [15:0] x2;
  input [15:0] y2;
endmodule
```

```
module top();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x1(x_data),
      .y1(y_data));
  rcv rcv(.x2(x_data),
      .y2(y_data));
endmodule
```

int *signal_object*.**get_lower_index**();

DESCRIPTION :

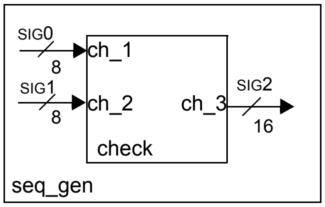
Returns the lower index value for the signal_object_name.

[CSL Interconnect Command Summary]

EXAMPLE :

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

FIGURE 1.22 Connecting three signals to the check unit



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output, 16);
check(){}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2;
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){
sig2.set_range(                     , sig0.get_upper_index() +
sig1.get_upper_index() +1);}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

int *signal_object*.**get_upper_index**();

DESCRIPTION :

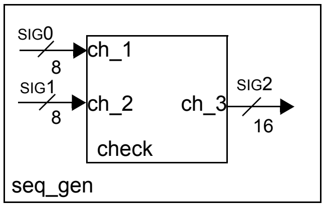
Return the upper index value for the signal.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the return value of the get_upper_index() method is used to set another upper index. Since the return value is an int, it can also be used in other contexts where an int parameter would be allowed.

FIGURE 1.23 Connecting three signals to the check unit



CSL CODE

```
  csl_unit check{
  csl_port ch_1(input,8), ch_2(input,8), ch_3(output, 16);
  check(){}
  };
  csl_unit seq_gen{
  csl_signal sig0(wire,8), sig1(wire,8), sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
  seq_gen(){
  sig2.set_range(sig0.get_lower_index(),                        +
                            +1);
  }};
```

VERILOG CODE

```
  module check(ch_1,ch_2,ch_3);
    input [7:0] ch_1;
    input [7:0] ch_2;
    output [15:0] ch_3;
  endmodule
  module seq_gen();
    wire [7:0] sig0;
    wire [7:0] sig1;
    wire [15:0] sig2;
    check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
   endmodule
```

csl_signal_type signal_object.**get_type**();

DESCRIPTION :

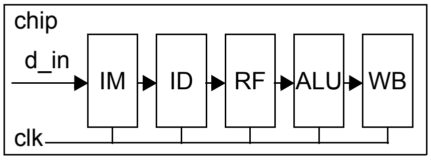
Returns the type of the signal object or port object for which is called.

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the use of get_type() method combined with set_type(). This way, signal/ports dependencies are created.

FIGURE 1.24 Connecting clocks and a input signal to units



CSL CODE

```
csl_unit im{
csl_port im_c(input), im_s(input);
im(){}
};
csl_unit id{
csl_port id_c(input), id_s(input);
id(){}
};
csl_unit rf{
csl_port rf_c(input), rf_s(input);
rf(){}
};
csl_unit alu{
csl_port alu_c(input), alu_s(input);
alu(){}
};
csl_unit wb{
csl_port wb_c(input), wb_s(input);
wb(){}
};
csl_unit chip{
csl_signal  clk1, d_in;
im im(.im_c(clk1), .im_s(d_in));
id id(.id_c(clk1));
rf rf(.rf_c(clk1));
alu alu(.alu_c(clk1));
wb wb(.wb_c(clk1));
chip(){
clk1.set_type(wire);
d_in.set_type(                );
}
};
```

VERILOG CODE

```
 module im(im_p);
  input im_p;
endmodule
module id(id_p);
  input id_p;
endmodule
module rf(rf_p);
  input rf_p;
endmodule
module alu(alu_p);
  input alu_p;
endmodule
module wb(wb_p);
  input wb_p;
endmodule
module chip();
  wire [7:0] clk1;
  wire [7:0] d_in;
  im im(.im_p(clk1));
  id id(.id_p(clk1));
  rf rf(.rf_p(clk1));
  alu alu(.alu_p(clk1));
  wb wb(.wb_p(clk1));
endmodule
```

```
csl_signal_attr signal_object.get_attr();
```

DESCRIPTION :

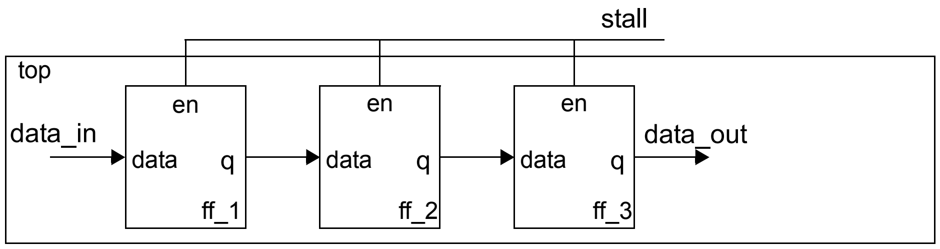
Returns the attribute of *signal_object_name*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 2.12.
It can only be called on a signal object and returns an object of type *csl_signal_attr* .

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1, ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 1.25



CSL CODE

```
    csl_unit ff{
    csl_port q(output), data(input), e(input);
    ff(){}
    };
    csl_unit top{
    csl_signal d_data1,d_data2,data_in,data_out,st;
    ff ff_1(.data(data_in),.e(st),.q(d_data1));
    ff ff_2(.data(d_data1),.e(st),.q(d_data2));
    ff ff_3(.data(d_data2),.e(st),.q(data_out));
    top(){
    st.set_attr(en);
    data_out.set_attr(              );
      }
    };
```

VERILOG CODE

```
    module ff(q,
              data,
              e);
      input data;
```

```
  input e;
  output q;
endmodule
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

```
int signal_group_object.get_width();
```

DESCRIPTION :

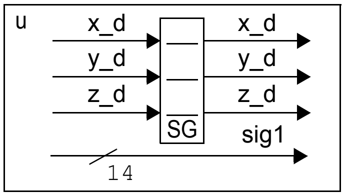
Will returns the sum of all widths of the signals in the signal_group named `signal_group_object.`

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows one signal group named SG which contains three signals with different widths, and a signal named *sig1*. The width of signal *sig1* is set to be equal with width of signal_group using the commands *set_width* and *get_width().*

FIGURE 1.26



CSL CODE

```
    csl_signal_group sg{
      csl_signal  x_d(2);
      csl_signal  y_d(4);
      csl_signal  z_d(8);
    sg(){ }
     };
    csl_unit u{
    sg sginst;
    csl_signal sig1;
    u(){
    sig1.set_width(sginst.           );}
     };
```

VERILOG CODE

```
    module u();
      wire [1:0]  sg1inst_x_d;
      wire [3:0] sg1inst_y_d;
      wire [7:0] sg1inst_z_d;
      wire [13:0] sig1;
    endmodule
```

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

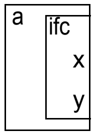
DESCRIPTION :

This command adds all the ports from an interface object (or only ports specified by the optional parameter *port_direction*) the default interface of the unit. This can be useful when there is a need to have the port names in the generated verilog code without interface names appended.

[CSL Interconnect Command Summary]

EXAMPLE :

In the following example the ports *x* and *y* are introduced in a *csl_list* which was added to the interface *ifc.*



CSL CODE

```
csl_interface ifc{
ifc(){
                                                   ;

}
};
csl_unit a{
ifc ifc;
a(){}
};
```

VERILOG CODE

```
 module a(ifc_x,
         ifc_y);
  input [1:0] ifc_x;
  input [1:0] ifc_y;
endmodule
```

```
int port_object.get_width();
```

DESCRIPTION :

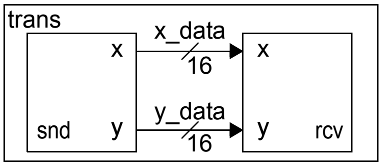
Returns the width of a port, otherwise it generates a cslc compile time error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 1.20 is illustrated the use of signals and ports to connect *snd* and *rcv* units inside a *trans* unit. The width of the ports is set using *set_width* and *get_width* commands.

FIGURE 1.27 A sender and a receiver connected by two signals



CSL CODE

```
    csl_unit snd{
    csl_port x(output), y(output);
    snd(){
    x.set_width(16);
    y.set_width(              );
    }
    };
    csl_unit rcv{
    csl_port x(input), y(input);
    rcv(){
    x.set_width(16);
    y.set_width(              );
    }
    };
    csl_unit trans{
    csl_signal x_data(16), y_data(16);
    snd snd( .x(x_data), .y(y_data));
    rcv rcv( .x(x_data), .y(y_data));
    trans(){ }
    };
```

VERILOG CODE

```
    module snd(x,y);
      output [15:0] x;
      output [15:0] y;
    endmodule
    module rcv(x,y);
```

```
  input [15:0] x;
  input [15:0] y;
endmodule
module trans();
  wire  [15:0] x_data;
  wire  [15:0] y_data;
  snd snd(.x(x_data), .y(y_data));
  rcv rcv(.x(x_data), .y(y_data));
endmodule
```

bitrange_object port_object.**get_bitrange()**;

DESCRIPTION :

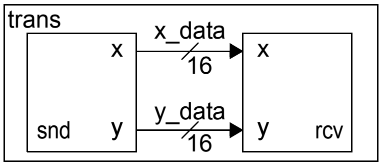
Returns a bitrange object. The return type of this function is an object of type bitrange.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 1.21 is illustrated the use of signals to connect *snd* and *rcv* units inside a *trans* unit. The width of the ports is set using *set_bitrange* and *get_bitrange* commands.

FIGURE 1.28



CSL CODE

```
csl_bitrange br1(16);
csl_unit snd{
csl_port x1(output), y1(output);
snd(){
x1.set_bitrange(br1);
y1.set_bitrange(                  );
}
};
csl_unit rcv{
csl_port x2(input), y2(input);
rcv(){
x2.set_bitrange(br1);
y2.set_bitrange(                  );
}
};
csl_unit top{
csl_signal x_data(br1) , y_data(br1);
snd snd( .x1(x_data), .y1(y_data));
rcv rcv( .x2(x_data), .y2(y_data));
top(){
}};
```

VERILOG CODE

```
module snd(x1,y1);
  output [15:0] x1;
  output [15:0] y1;
endmodule
```

```
module rcv(x2,y2);
  input [15:0] x2;
  input [15:0] y2;
endmodule
module top();
  wire [15:0] x_data;
  wire [15:0] y_data;
  snd snd(.x1(x_data),
      .y1(y_data);
  rcv rcv(.x2(x_data),
      .y2(y_data));
endmodule
```

int *port_object_name*.**get_lower_index**();

DESCRIPTION :

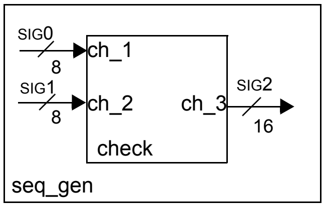
Returns the lower index value for the port_object_name.

[CSL Interconnect Command Summary]

EXAMPLE :

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

FIGURE 1.29 Connecting three to the check unit



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output);
check(){
ch_3.set_range(                    , ch_1.get_upper_index() +
ch_2.get_upper_index() +1);
}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2(wire,16);
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

int *port_object_name*.**get_upper_index**();

DESCRIPTION :

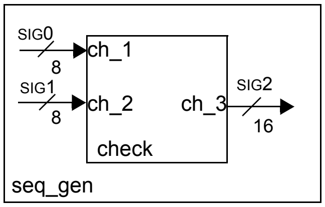
Return the upper index value for the port.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example, the return value of the get_upper_index() method is used to set another upper index. Since the return value is an int, it can also be used in other contexts where an int parameter would be allowed.

FIGURE 1.30



CSL CODE

```
csl_unit check{
csl_port ch_1(input,8), ch_2(input,8), ch_3(output);
check(){
ch_3.set_range(ch_1.get_lower_index(),                        +
                           +1);
}
};
csl_unit seq_gen{
csl_signal sig0(wire,8), sig1(wire,8), sig2(wire,16);
check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
seq_gen(){}
};
```

VERILOG CODE

```
module check(ch_1,ch_2,ch_3);
  input [7:0] ch_1;
  input [7:0] ch_2;
  output [15:0] ch_3;
endmodule
module seq_gen();
  wire [7:0] sig0;
  wire [7:0] sig1;
  wire [15:0] sig2;
  check check( .ch_1(sig0),.ch_2(sig1), .ch_3(sig2));
 endmodule
```

csl_port_type port_object_name.**get_type**();

DESCRIPTION :

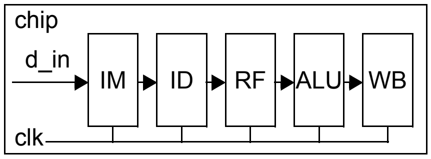
Returns the type of the port object or signal object for which is called *port_object_name*.

[CSL Interconnect Command Summary]

EXAMPLE :

This example illustrates the use of get_type() method combined with set_type(). This way, signal/ports dependencies are created.

FIGURE 1.31 Connecting clocks and a input signal to units



CSL CODE

```
csl_unit im{
csl_port im_p(input), im_c(input);
im(){
im_p.set_type(wire);
im_c.set_type(                );}
};
csl_unit id{
csl_port id_p(input), id_c(input);
id(){
id_p.set_type(wire);
id_c.set_type(                );}
};
csl_unit rf{
csl_port rf_p(input), rf_c(input);
rf(){
rf_p.set_type(wire);
rf_c.set_type(                );}
}
};
csl_unit alu{
csl_port alu_p(input), alu_c(input);
alu(){
alu_p.set_type(wire);
alu_c.set_type(                );}
};
csl_unit wb{
```

```
csl_port wb_p(input), wb_c(input);
wb(){
wb_p.set_type(wire);
wb_c.set_type(               );}
};
csl_unit chip{
csl_signal clk1(8), d_in(8);
im im( .im_p(clk1), .im_c(d_in));
id id( .id_p(clk1),.id_c(d_in));
rf rf( .rf_p(clk1),.rf_c(d_in));
alu alu( .alu_p(clk1),.alu_c(d_in));
wb wb( .wb_p(clk1),.wb_c(d_in));
chip(){
clk1.set_type(wire);}
};
```

VERILOG CODE

```
module im(im_p, im_c);
  input im_p;
  input im_c;
endmodule
module id(id_p, id_c);
  input  id_p;
  input  id_c;
endmodule
module rf(rf_p, rf_c);
  input rf_p;
  input  rf_c;
endmodule
module alu(alu_p, alu_c);
  input  alu_p;
  input  alu_c;
endmodule
module wb(wb_p, wb_c);
  input wb_p;
  input wb_c;
endmodule
module chip();
  wire [7:0] clk1;
  wire [7:0] d_in;
  im im(.im_p(clk1),.im_c(d_in));
  id id(.id_p(clk1), .id_c(d_in));
  rf rf(.rf_p(clk1), .rf_c(d_in));
  alu alu(.alu_p(clk1), .alu_c(d_in));
  wb wb(.wb_p(clk1), .wb_c(d_in));
endmodule
```

```
csl_port_attr port_object_name.get_attr();
```

DESCRIPTION :

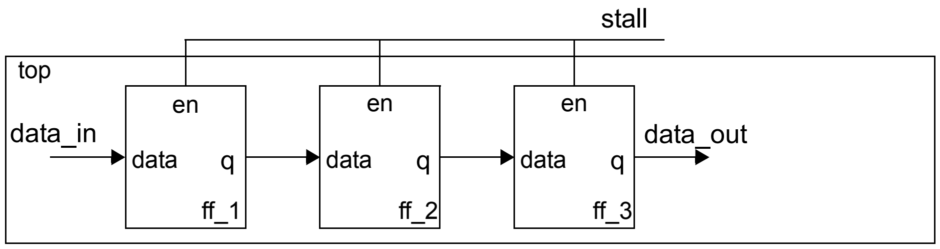
Returns the attribute of *port_object_name*. Attributes describe what the port is used for in the design. An attribute listing is given in Table 2.12.
It can only be called on a port object and returns an object of type *csl_port_attr* .

[CSL Interconnect Command Summary]

EXAMPLE :

We want to interconnect three instances of a flip flop ,*ff_1, ff_2* and *ff_3*.The module *top* contains both *ff_1, ff_2* and *ff_3* and additionally *stall* and *data_in, data_out* signals.When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 1.32



CSL CODE

```
csl_unit ff{
csl_port q(output), data(input), e(input);
ff(){
q.set_attr(en);
e.set_attr(              );
}
};
csl_unit top{
csl_signal d_data1,d_data2,data_in,data_out,st;
ff ff_1(.data(data_in),.e(st),.q(d_data1));
ff ff_2(.data(d_data1),.e(st),.q(d_data2));
ff ff_3(.data(d_data2),.e(st),.q(data_out));
top(){ }
};
```

VERILOG CODE

```
module ff(q,
          data,
          e);
  input data;
  input e;
  output q;
endmodule
module top();
  wire d_data1;
  wire d_data2;
  wire data_in;
  wire data_out;
  wire st;
  ff ff_1(.data(data_in),
          .e(st),
          .q(d_data1));
  ff ff_2(.data(d_data1),
          .e(st),
          .q(d_data2));
  ff ff_3(.data(d_data2),
          .e(st),
          .q(data_out));
endmodule
```

```
int interface_object.get_width();
```

DESCRIPTION :

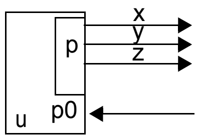
Will return the sum of all widths of the ports in the interface.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows an interface named *p* which contains three ports with different widths, and a port named *p0*. The width of port *p0* is set to be equal with width of interface using the commands *set_width* and *get_width().*

FIGURE 1.33



CSL CODE

```
    csl_interface p{
    csl_port x(output, 2), y(output,4), z(output, 8);
    p(){}
    };
    csl_unit u{
    p p;
    csl_port p0(input);
    u(){
    p0.set_width(p.get_width());
    }
    };
```

VERILOG CODE

```
    //
```

```
string unit_object_name.get_unit_prefix();
```

DESCRIPTION :

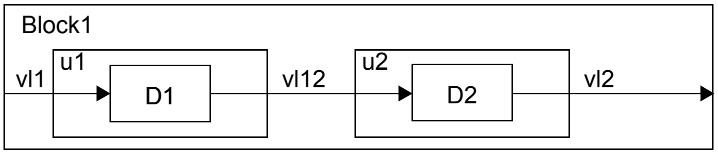
Returns the *unit_object_name*'s *string_prefix*.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the prefix “box1” for the unit *u2* using *get_unit_prefix method().*

FIGURE 1.34 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){ }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){ }
};
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
u1.set_unit_prefix("box1");
u2.set_unit_prefix(                     LOCAL_ONLY); }
};
```

VERILOG CODE

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

DESCRIPTION :

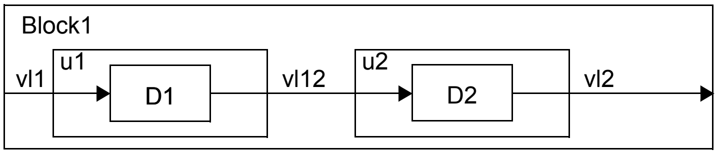
Returns the signal_prefix and the port_prefix previously set by the set_signal_prefix() command.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows who to set a prefix to a signal named *v12* using *get_signal_prefix()* method.

FIGURE 1.35 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");
v1.set_signal_prefix(v12.                    );
 }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix("uv2");
v2.set_signal_prefix(                        );
    }
};
```

```
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
set_unit_prefix("box1");
 }
};
```

VERILOG CODE

string *signal_object_name*.**get_signal_prefix_local();**

DESCRIPTION :

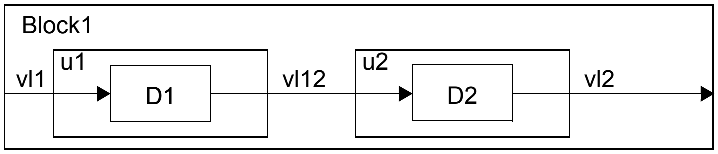
Returns the local signal prefixes that have been set by a previous set_signal_prefix_local() command. This command does not affect ports (ports are not local signals).

[CSL Interconnect Command Summary]

EXAMPLE :

Sets a local prefix “uv2” for signal v2 using the *get_signal_prefix_local()* method.

FIGURE 1.36 An unit named *block1* with 3 instances.



CSL CODE

```
csl_unit d1{
csl_port d1_i(input), d1_o(output);
d1(){ }
};
csl_unit d2{
csl_port d2_i(input), d2_o(output);
d2(){ }
};
csl_unit u1{
csl_port v1_i(input), v1_o(output);
csl_signal v1,v12;
d1 d1(.d1_i(v1),.d1_o(v12));
u1(){
v12.set_signal_prefix("uv2");
v1.set_signal_prefix_local("uv");
 }
};
csl_unit u2{
csl_port v2_i(input), v2_o(output);
csl_signal v12,v2;
d2 d2(.d2_i(v12),.d2_o(v2));
u2(){
v12.set_signal_prefix_local("uv2");
v2.set_signal_prefix_local(                              );
  }
};
csl_unit block{
csl_signal vl1,vl12,vl2;
u1 u1(.v1_i(vl1),.v1_o(vl12));
u2 u2(.v2_i(vl12),.v2_o(vl2));
block(){
set_unit_prefix("box1");
  }
};
```

VERILOG CODE

[(*interface_name*.)+]**register_ios(input**|**output** [, **.reset**[_]**(***optional_reset***)**, reset_value][**,.en(***optional_enable***)**]**);**

DESCRIPTION :

The register IOS (Inputs/Outputs) command applies to units and allows for inputs/outputs to be registered with flip-flops.
The command can be called on all ports/interfaces of a unit or it can specify a certain interface and it will apply only to that particular interface and contained interfaces (if any).
Only input and output ports will be affected by this command: **inout** type will not be registered.

Note: In order for this to work all connectivity elements involved need to have a clock associated to them with the *set_clock()* method (because each port in the unit's interface can be clocked by a different clock line) otherwise there will be an error.
For the ports with special attributes (clock, en, reset), the r*egister_ios()* method doesn't create a flip flop.

Generated verilog code for (always blocks) for the unit with the selected ports according to the specified direction (input/output) is given according to different specified options below:

- standard

```
always @(posedge clk) begin
 local_signal <= port_name;
end
```

- optional reset (low true):

```
always @(posedge clk or negedge optional_reset) begin
 if(~optional_reset) begin
   local_signal <= reset_value;
  end
 else begin
   local_signal <= <port_name>;
 end
end
```

- optional enable and reset (high true reset):

```
always @(posedge clk or posedge optional_reset) begin
  if(optional_reset) begin
   local_signal <= reset_value;
  end
 else if(optional_enable) begin
   local_signal <= port_name;
  end
end
```

If the *register_ios()* method is applied to more than one port , in the generated verilog code, for each port there is an *always* block created.
The *local_signal* (that registers the port) has it's name inferred from the *port_name* like this:

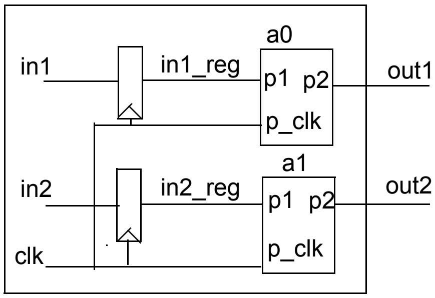
```
port_name_reg
```

NOTE: if the name exists in the scope append another _reg ?

EXAMPLE :

In this example the method is applied only for input ports.

FIGURE 1.37



CSL CODE

```
csl_unit a{
  csl_port p1(input,4);
  csl_port p2(output,4);
  csl_port p_clk(input);
  a(){
    p_clk.set_attr(clock);
}
};
csl_unit top{
  csl_port in1(input,4);
  csl_port in2(input,4);
  csl_port out1(output,4);
  csl_port out2(output,4);
  csl_port clk(input);
  csl_signal op_res;
  csl_signal op_en;
  a a0(.p1(in1_reg),.p2(out1),.p_clk(clk));
  a a1(.p1(in2_reg),.p2(out2),.p_clk(clk));
  top(){
    clk.set_attr(clock);
```

```
      set_clock(clk);

    }
  };
```

VERILOG CODE

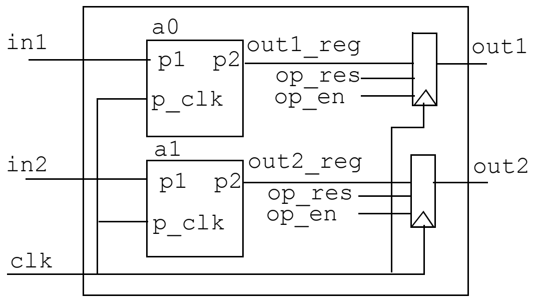
```
module a(p1,
         p2,
         p_clk);
input [3:0] p1;
output [3:0] p2;
input p_clk;
endmodule
module top(in1,
           in2,
           out1,
           out2,
           clk);
input [3:0] in1;
input [3:0] in2;
output [3:0] out1;
output [3:0] out2;
input clk;
wire op_res;
wire op_en;
reg [3:0] in1_reg;
reg [3:0] in2_reg;
always @(posedge clk or posedge op_res) begin
  if(op_res) begin
   in1_reg <= 4'b0000;
  end
 else if(op_en) begin
   in1_reg <= in1;
  end
end
always @(posedge clk or posedge op_res) begin
  if(op_res) begin
   in2_reg <= 4'b0000;
  end
 else if(op_en) begin
   in2_reg <= in2;
```

```
     end
   end
    a a0(.p1(in1_reg),.p2(out1),.p_clk(clk));
    a a1(.p1(in2_reg),.p2(out2),.p_clk(clk));
   endmodule
```

EXAMPLE :

In this example the method is applied only for output ports.

FIGURE 1.38



CSL CODE

```
   csl_unit a{
      csl_port p1(input,4);
      csl_port p2(output,4);
      csl_port p_clk(input);
   a(){
     p_clk.set_attr(clock);
   }
   };
   csl_unit top{
      csl_port in1(input,4);
      csl_port in2(input,4);
      csl_port out1(output,4);
      csl_port out2(output,4);
      csl_port clk(input);
      csl_signal op_res;
      csl_signal op_en;
   a a0(.p1(in1),.p2(out1_reg),.p_clk(clk));
   a a1(.p1(in2),.p2(out2_reg),.p_clk(clk));
   top(){
```

```
clk.set_attr(clock);
set_clock(clk);
register_ios(output,.reset(op_res),0,.en(op_en));
}
};
```

VERILOG CODE

```
module a(p1,p2,p_clk);
  input [3:0] p1;
  input [3:0] p2;
  input p_clk;
endmodule
module top(in1,in2,out1,out2,clk);
  input [3:0] in1;
  input [3:0] in2;
  output [3:0] out1;
  output [3:0] out2;
  input clk;
  wire op_res;
  wire op_en;
  reg [3:0] out1_reg;
  reg [3:0] out2_reg;
  reg [3:0] out1;
  reg [3:0] out2;

always @(posedge clk or posedge op_res) begin
   if(op_res) begin
      out1_reg =4’b0000;
   end
   else if(op_en) begin
      out1<=out1_reg;
   end
end
always @(posedge clk or posedge op_res) begin
   if(op_res) begin
      out2_reg =4’b0000;
   end
   else if(op_en) begin
      out2<=out1_reg;
   end
end
```

```
a a0(.p1(in1),.p2(out1_reg),.p_clk(clk));
a a1(.p1(in2),.p2(out2_reg,.p_clk(clk));
endmodule
```

`generate_individual_rtl_signals(on|off);`

DESCRIPTION :

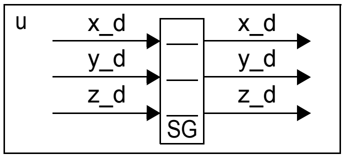
When this is set to **on**, as the group "traverses" scopes the autorouter component will generate a port for each signal inside the group. If it is set to **off** (default behaviour), the grouped signals will be "merged" into a single port as they come across scope boundaries, else if the *status* is set to true, the grouped signals will be used as individual ports.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example we have one unit *u* which contains a signal-group *SG* with signals *x_d, y_d, z_d.* The command *generate_individual_rtl_signals(on)* is used for generate a port for each signal inside the SG group.

FIGURE 1.39



CSL CODE

```
csl_signal_group sg{
csl_signal x_d;
csl_signal y_d;
csl_signal z_d;
sg(){

 }
 };
csl_unit u{
sg sg1inst;
u(){  }
 };
```

VERILOG CODE

```
module u();
  wire sg1inst_x_d;
  wire sg1inst_y_d;
  wire sg1inst_z_d;
endmodule
```