
CHAPTER 2 CSL Interconnect

All rights reserved
Copyright ©2008 Fastpath Logic, Inc.
Copying in any form without the expressed written
permission of Fastpath Logic, Inc is prohibited

TABLE 2.1 Chapter Outline

2.1	CSL Interconnect Command Summary
2.2	CSL Interconnect Commands

2.1 CSL Interconnect Command Summary

2.1.1 CSL Signal

2.1.1.1 CSL Signal Rules

Can only be declared

2.1.1.2 Usage tables

TABLE 2.2 CSL signal declaration in other CSL classes

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

2.1.2 CSL Signal Group

2.1.2.1 CSL Signal Group Rules

Csl Signal Group can be defined and instantiated.

Csl Signal Group can be defined in:

TABLE 2.3 CSL signal group definition in other CSL classes

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

Csl signal group can be instantiated in

TABLE 2.4 CSL signal group instantiation in other CSL classes

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

TABLE 2.4 CSL signal group instantiation in other CSL classes

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

CSL Port

Csl port can only be declared

TABLE 2.5 CSL port declaration in other CSL classes

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

CSL Interface

Csl interface can be defined and instantiated.

Csl interface can be defined in:

TABLE 2.6 CSL interface definition in other CSL classes

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

Csl Interface can be instantiated in:

TABLE 2.7 CSL interface instantiation in other CSL classes

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

TABLE 2.7 CSL interface instantiation in other CSL classes

CSL class	Can be instantiated in
CSL Memory Map	-
CSL Memory Map Page	-
CSL Isa Element	-
CSL Isa Field	-
CSL Field	-

NOTE:

The *no_prefix()*, *set_prefix()* and *set_suffix()* commands which can be used for interfaces and signal groups are described in the Language document.

CSL Unit

Csl unit can be defined and instantiated.

Csl unit can be defined in:

TABLE 2.8 CSL unit definition in other CSL classes

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

CSL unit can be instantiated in

TABLE 2.9 CSL unit instantiation in other CSL classes

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

2.1.3 Formal to actual connection

2.1.3.1 Formal to actual rules

TABLE 2.10 Legal objects for formal

formal
port
interface_instance.port
interface_instance

TABLE 2.11 Legal objects for actual

actual
port
port[port_select]
signal
signal[port_select]
expression ¹
interface_instage.port
interface_instage.port[ps]
signal_group_instance.signal
signal_group_instance.signal[ps]
interface_instance
signal_group_instance

NOTE:

1. *expression* can be:

- *concat_expression*, *replicate_expression*, *operation_expression* and *constant_numeric_expression* if formal is an input port or an input interface.
- only *concat_expression* if formal is an output /inout port or output/inout interface.

Rules for the following cases: **.ifc(ifc1) / .ifc(sg)**

- if the formal ifc's ports and ports/signals from actual ifc1/sg have the same name and the same widths the connection will be by name. Else should be an error which specify this.

-if the ifc's ports and ports/signals from ifc/sg don't have the same names then the connection will be in order (first port from formal with first port/signal from actual, second port from formal with second port/signal from actual) and the pair formal port - actual port/signal has to have the same widths.

Signals

```

csl_signal signal_name;
csl_signal signal_name0(signal_name1);
csl_signal signal_name([signal_data_type][,width]);
csl_signal signal_object([signal_data_type,]upper_limit,lower_limit);
csl_signal signal_object([signal_data_type,]bitrange_object_name);
signal_object.set_width(numeric_expression);
signal_object.set_bitrange(bitrange_object_name);

```



```
signal_object.set_range(lower_limit, upper_limit);
signal_object.set_offset(numeric_expression);
```

Signal Types

```
signal_object.set_type(csl_signal_type);
signal_object.set_attr(csl_signal_attr);
```

Signal groups

```
csl_signal_group signal_group_name;
```

Units

```
csl_unit unit_name;
```

Units: port

```
csl_port
port_name(port_direction[,port_type][,range][[,width][[,upper_index,lower_index][[,bitrange_object_name]]];
csl_port port_object(port_hierarchical_identifier);
port_object.reverse();
port_object.set_width(numeric_expression);
port_object.set_bitrange(bitrange_object);
port_object.set_range(lower_limit, upper_limit);
port_object.set_offset(numeric_expression);
```

Port types

```
port_object.set_type(csl_port_type);
port_object.set_attr(csl_port_attr);
```

Units: interface

```
csl_interface interface_object_name;
instance_interface_name.reverse();
```

Units: misc

```
do_not_gen_cpp();
do_not_gen_rtl();
gen_unique_rtl_modules();
```

Clock domains

```
[object_name.]set_clock(clock_name);
```

Units:connection

```
(hid[.ps] | expression).connect_by_name((hid[.ps] | expression)
[,f2a_name]);
scope.connect_units(scope [,"f2a_prefix"]);
hid[.ps].connect_by_pattern(hid_pattern[.ps] [,f2a_name]);
```

2.2 CSL Interconnect Commands

We will now describe the CSL interconnect specification commands. The CSL interconnect specification commands are used to create new unit instances and to connect the instances. The ports can connect to an expression in a parent module or the ports can connect to ports in other instances in the same module or the ports can connect to the ports in the parent module. Connecting ports to logic in a higher level is not recommended.

2.2.1 Signals

The following section describes signals. Signals connect objects. The default type for a signal is a wire. A wire needs to be driven at all times. If it is not driven then the value on the wire is undefined and the wire is treated as a don't care by the synthesis tools. A don't care can be a zero or a one. Signals are internal to the unit. Signals are used to connect different units using formal to actual name mapping. Formals are the ports or interfaces in the unit. Actual names are the signals used to connect to the ports.

cs1_signal *signal_name*;

DESCRIPTION :

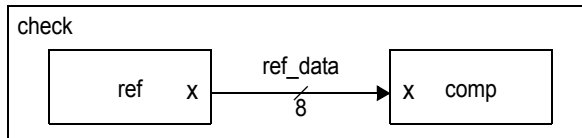
The signal can be declared inside a CSL unit's scope, signal-group or testbench.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example is shown a data check unit that uses a reference and a comparator. The connection between the reference and the comparator is made using a 8-bit signal called *ref_data*.

FIGURE 2.1



CSL CODE

```

cs1_unit ref{
    cs1_port x (output, 8);
    ref(){};
cs1_unit comp{
    cs1_port x (input, 8);
    comp(){}
};
cs1_unit check{
    cs1_signal ref_data(wire, 8);
    ref ref(.x(ref_data));
    comp comp( .x(ref_data));
    check(){}
};
  
```

VERILOG CODE

```

module ref(x);
    output [7:0] x;
endmodule
module comp(x);
    input [7:0] x;
endmodule
module chek();
    wire [7:0] ref_data;
    ref ref(.x(ref_data));
    comp comp(.x(ref_data));
endmodule
  
```

```
csl_signal signal_name0(signal_name1);
```

DESCRIPTION :

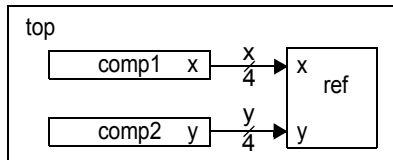
Creates a new signal with the name *signal_name0* by copying the signal object *signal_name1* passed as constructor argument. Signal_name1 can also be a hid (copy from another scope).

[*CSL Interconnect Command Summary*]

EXAMPLE :

Creates two units named *comp1* and *comp2* , a reference unit named *ref* and a *top* unit. In the *top* unit two signals *x* , *y* are declared .

FIGURE 2.2 Three units and a reference unit



CSL CODE

```
csl_unit comp1{
    csl_port x(output,4);
    comp1() {}
};

csl_unit comp2{
    csl_port y (output,4);
    comp2() {}
};

csl_unit ref{
    csl_port x(input,4), y(input,4);
    ref() {}
};

csl_unit top {
    csl_signal x(wire,4), y(x);
    ref ref (.x(x),.y(y));
    comp1 comp1 (.x(x));
    comp2 comp2 (.y(y));
    top() {}
};
```

VERILOG CODE

```
module comp1(x);
    output [3:0] x;
endmodule

module comp2(y);
    output [3:0] y;
```

```
endmodule
module ref(x,y);
    input [3:0] x;
    input [3:0] y;
endmodule
module top();
    wire [3:0] x;
    wire [3:0] y;
    ref ref(.x(x), .y(y));
    comp1 comp1(.x(x));
    comp2 comp2(.y(y));
endmodule
```

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

DESCRIPTION :

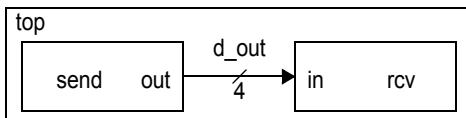
Creates a signal object named *signal_name*. Optionally the type and width of the signal can be specified: *signal_data_type* specifies the type and can take any of the values in table Signal Types. Width specifies the width of the signal and can be a numeric expression.

[CSL Interconnect Command Summary]

EXAMPLE :

The example shows two connected units (a sender and a receiver). Each has a 4 bit port and are interconnected by a 4 bit signal in the top unit.

FIGURE 2.3 A sender and a receiver connected by a signal



CSL CODE

```

csl_unit send{
    csl_port out(output,4);
    send() {}
};

csl_unit rcv{
    csl_port in(input,4);
    rcv() {}
};

csl_unit top{
    csl_signal d_out(wire,4);
    send send( .out(d_out));
    rcv rcv( .in(d_out));
    top() {}
};

```

VERILOG CODE

```

module send(out);
    output [3:0] out;
endmodule

module rcv(in);
    input [3:0] in;
endmodule

module top();
    wire [3:0] d_out;

```

```
send send(.out(d_out));  
rcv rcv(.in(d_out));  
endmodule
```

csl_signal

```
signal_object([signal_data_type,]upper_limit,lower_limit);
```

DESCRIPTION :

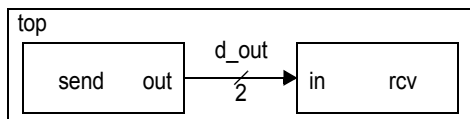
Creates a signal named *signal_object*. The constructor takes as parameters two numeric expressions (*upper_limit* and *lower_limit*) that represent the MSB (most significant bit) and LSB (least significant bit) of the bitrange associated with the signal. Optionally, the *signal_data_type* parameter can be specified setting the type of the signal to reg or wire, tri etc.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example three units are declared: a sender called *send*, a receiver unit called *rcv* and a global unit called *top*. The *send* unit has a 2-bit output port. The *rcv* unit has an input port with 2 bits width. The *top* unit has a 2-bit signal called *d_out*.

FIGURE 2.4 A sender and a receiver connected by a signal.

**CSL CODE**

```

csl_unit send{
    csl_port out(output,2);
    send(){}
};

csl_unit rcv{
    csl_port in(input,2);
    rcv(){}
};

csl_unit top{
    csl_signal d_out(wire,1,0);
    send send( .out(d_out));
    rcv rcv( .in(d_out));
    top(){}
};

```

VERILOG CODE

```

module send(out);
    output [1:0] out;
endmodule

module rcv(in);
    input [1:0] in;
endmodule

module top();
    wire [0:1] d_out;

```



```
send send(.out(d_out));  
rcv rcv(.in(d_out));  
endmodule
```

```
csl_signal signal_object([signal_data_type,]bitrange_object_name);
```

DESCRIPTION :

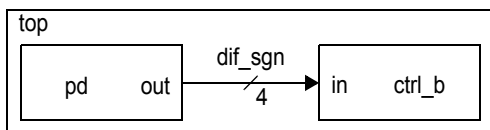
Creates a signal named *signal_object* with bitrange *bitrange_object_name*. Every signal object has a bitrange object attached by default. The default bitrange width for a newly created signal is 1. If a signal is created and the constructor uses a bitrange object as a parameter, a copy of that bitrange object is associated with the particular signal; the bitrange parameter can be a previously defined bitrange object or it can be an anonymous bitrange (defined on the spot). Optionally, the *signal_data_type* parameter can be specified setting the type of the signal to reg or wire, tri etc.

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example a bitrange_object named *br1* is passed as a parameter to one signal named *dif_sgn*.

FIGURE 2.5 Two units connected by a signal

**CSL CODE**

```

csl_bitrange br1(4,1);
csl_unit pd{
    csl_port out(output,br1);
    pd(){}
};
csl_unit ctrl_b{
    csl_port in(input,br1);
    ctrl_b(){}
};
csl_unit top{
    csl_signal dif_sgn (wire, br1);
    pd pd (.out(dif_sgn));
    ctrl_b ctrl_b ( .in(dif_sgn));
    top(){}
};
  
```

VERILOG CODE

```

module pd(out);
    output [4:1] out;
endmodule

module ctrl_b(in);
    input [4:1] in;
endmodule
  
```

```
module top();  
    wire [4:1] dif_sgn;  
    pd pd(.out(dif_sgn));  
    ctrl_b ctrl_b(.in(dif_sgn));  
endmodule
```

```
signal_object.set_width(numeric_expression);
```

DESCRIPTION :

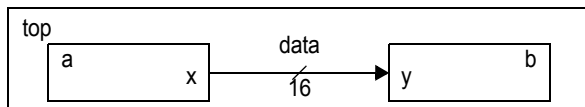
Set the width of a single dimensional signal. The bitrange of the signal is automatically created from the width. If the signal object width has already been set the cslic will flag an error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.6 is illustrated the use of signal *data* to connect *a* and *b* units inside a *top* unit. The width of the signal is set using *set_width* commands.

FIGURE 2.6



CSL CODE

```

csl_unit a{
    csl_port x(output,16);
a(){} };
csl_unit b{
    csl_port y(input,16);
b(){}
};
csl_unit top{
    csl_signal data(wire);
    a a( .x(data));
    b b( .y(data));
    top(){
        data.set_width(16); }
};

```

VERILOG CODE

```

module a(x);
    output [15:0] x;
endmodule
module b(y);
    input [15:0] y;
endmodule
module top();
    wire [15:0] data;
    a a(.x(data));
    b b(.y(data));
endmodule

```

```
signal_object.set_bitrange(bitrange_object_name);
```

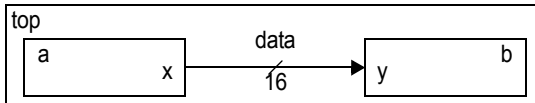
DESCRIPTION :

Sets the bitrange object of a signal. The bitrange object passed as a parameter to the `set_bitrange()` method must already be declared. After the assignment of the bitrange the signal contains a reference to the bitrange object *bitrange_object_name*.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.7 is illustrated the use of a signal to connect *a* and *b* units inside a *top* unit. The width of the signal is set using `set_bitrange` commands.

FIGURE 2.7**CSL CODE**

```

csl_bitrange br1(16);
csl_unit a{
    csl_port x(output,br1);
    a(){} };
csl_unit b{
    csl_port y(input,br1);
    b(){} };
csl_unit top{
    csl_signal data;
    a a( .x(data));
    b b ( .y(data));
    top(){
        data.set_bitrange(br1); }
    };

```

VERILOG CODE

```

module a(x);
    output [15:0] x;
endmodule
module b(y);
    input [15:0] y;
endmodule
module top();
    wire [15:0] data;
    a a(.x(data));
    b b(.y(data));
endmodule

```

```
signal_object.set_range(lower_limit, upper_limit);
```

DESCRIPTION :

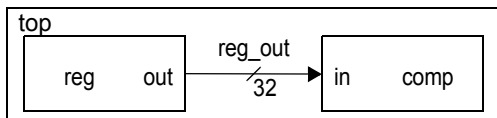
Set the bitrange for the *signal_object* using the *lower_limit* and *upper_limit* delimiters.

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example we have one 32-bit signal named *reg_out* which connects a register called *reg* and a comparator called *comp*. The width of the signal is set using *set_range* command.

FIGURE 2.8 A comparator and a register



CSL CODE

```

csl_unit reg1{
    csl_port out(output,32);
    reg1(){};
}
csl_unit comp{
    csl_port in(input, 32);
    comp(){}
};
csl_unit top{
    csl_signal reg_out;
    reg1 reg1( .out(reg_out));
    comp comp( .in(reg_out));
    top(){
        reg_out.set_range(0,31);
    };
};

```

VERILOG CODE

```

module reg1(out);
    output [31:0] out;
endmodule

module comp(in);
    input [31:0] in;
endmodule

module top();
    wire [31:0] reg_out;
    reg1 reg1(.out(reg_out));
    comp comp(.in(reg_out));
endmodule

```

```
signal_object.set_offset(numeric_expression);
```

DESCRIPTION :

Set the value to be added to both the lower and upper index of the signal. This method cannot be called if the signal has not set a width before. In this case will be an error.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the offset for the signal *s1*.

CSL CODE

```
csl_unit u1{
    csl_port p1(input,4);
    csl_signal s1(8);
    u1(){
        s1.set_offset(4);
    }
};
```

VERILOG CODE

```
module u1(p1,
          p2);
    input [3 : 0] p1;
    wire [11 : 4] s1;
endmodule
```

2.2.1.1 Signal Types

The CSL language will create additional type information for signals. The additional attributes information will be used to check that signals of compatible types are connected to each other. The type information in the CSL is declared. There are two different ways to set the signal type.

`signal_object.set_type(csl_signal_type);`

DESCRIPTION :

Assign a net type to *signal_object*. Applies to both signals and ports. *Csl_signal_type* specifies the type of signal and can take any of the values in table Signal Types. An input port cannot be register type. In this case the compiler shows an error.

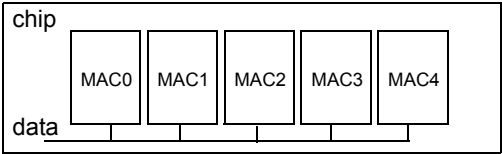
[*CSL Interconnect Command Summary*]

TABLE 2.12 Signal Types

Signal type	Description
wire	Creates a wire type signal
wand	Creates a wand type signal
wor	Creates a wor type signal
tri	Creates a tri type signal
triand	Creates a triand type signal
trior	Creates a trior type signal
tri0	Creates a tri0 type signal
tri1	Creates a tri1 type signal
supply0	Creates a supply0 type signal
supply1	Creates a supply1 type signal
reg	Creates a reg type signal.
integer	Creates a integer type signal.
time	Creates a time type signal.

This example illustrates the `set_type()` method applied on both signal objects and port objects

FIGURE 2.9 Connecting data to units



CSL CODE

```
csl_unit MAC{
    csl_port p_in(input);
    MAC () {
        p_in.set_type(integer);
    }
};
```



```
cs1_unit chip{
    cs1_signal data(8);
    MAC MAC0( .p_in(data));
    MAC MAC1( .p_in(data));
    MAC MAC2( .p_in(data));
    MAC MAC3( .p_in(data));
    MAC MAC4( .p_in(data));
    chip(){
        data.set_type(integer);
    }
};
```

VERILOG CODE

```
module MAC(p_in);
    input integer p_in;
endmodule

module chip();
    integer [7:0] data;
    MAC MAC0( .p_in(data));
    MAC MAC1( .p_in(data));
    MAC MAC2( .p_in(data));
    MAC MAC3( .p_in(data));
    MAC MAC4( .p_in(data));
endmodule
```

`signal_object.set_attr(csl_signal_attr);`

DESCRIPTION :

Assign an attribute to *signal_object*. Attributes describe what the signal is used for in the design. An attribute listing is given in Table 2.13

TABLE 2.13 signal attributes

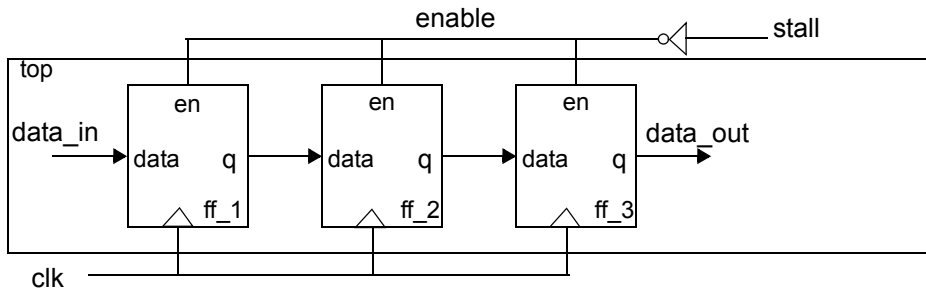
mnemonic	signal attribute
csl_enable	enable
csl_stall	stall
csl_mux_sel	mux select
csl_decode	decoded minterm/maxterm
csl_clock	clock signal
csl_reset	reset signal

[CSL Interconnect Command Summary]

EXAMPLE :

In this example three instances of a flop, *ff_1*, *ff_2* and *ff_3* are connected. The module *top* contains *ff_1*, *ff_2* and *ff_3* and additionally *st* and *data_in*, *data_out* signals.

FIGURE 2.10



CSL CODE

```

csl_unit ff{
    csl_port q(output), data(input), e(input), c(input);
    ff() {
        e.set_attr(csl_enable);
        c.set_attr(csl_clock);
    }
};

csl_unit top{
    csl_signal d_data1, d_data2, data_in, data_out, st, clk, en;
    ff ff_1(.data(data_in), .e(en), .q(d_data1), .c(clk));
    ff ff_2(.data(d_data1), .e(en), .q(d_data2), .c(clk));

```

```

    ff ff_3(.data(d_data2),.e(en),.q(data_out),.c(clk));
  top() {
    clk.set_attr(csl_clock);
    en.set_attr(csl_enable);
  }
};

```

VERILOG CODE

```

module ff(q,
           data,
           e,
           c);
  input data;
  input e;
  input c;
  output q;
endmodule

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

```

2.2.1.2 Grouping Signals

Signals can be grouped together and assigned a symbolic name. The great advantage of using sig-

nal groups is evident when connecting the objects in our design. Instead of connecting tens of wires to tens of ports we just connect a single object which contains group of signals (a signal group in CSL) to a group of ports (an interface in CSL).

This connection can be done manually (using the connect command) or automatically by the autorouter. In the first and second case, we must have the signals in the signal group and the ports in the interface put in the order of the connection preference so that the port and the signal we want to connect have the same index in the both lists.

```
csl_signal_group signal_group_name;
```

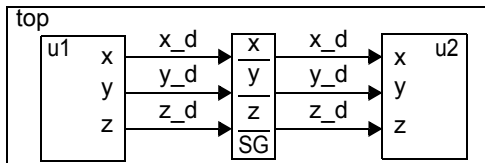
DESCRIPTION :

Creates a signal group named *signal_group_name*. *Csl_signal_group* is a scope delimited by curly braces and it contains a constructor. The signals are declared in the groups scope.

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example are three units: *u1*, *u2* and *top*. The *top* unit contains *u1* and *u2* units and a signal group named *SG* with signals *x_d*, *y_d*, *z_d*. The *u1* and *u2* units have the interfaces *ifc1* and *ifc2* which are connected with the signal group *SG*. The interface *ifc1* has 3 output ports called *x*, *y*, *z* and *ifc2* has 3 input ports called *x*, *y*, *z*.

FIGURE 2.11**CSL CODE**

```
csl_interface ifc1 {
csl_port x(output), y(output), z(output);
ifc1(){}
};

csl_interface ifc2 {
csl_port x(input), y(input), z(input);
ifc2(){}
};

csl_unit u1{
ifc1 ifc1;
u1(){}
};

csl_unit u2{
ifc2 ifc2;
u2(){}
};

csl_signal_group sg{
csl_signal x_d, y_d, z_d;
sg(){}
};

csl_unit top{
```

```

sg sg;
u1 u1( .ifc1(sg));
u2 u2( .ifc2(sg));
top(){ }
};

```

VERILOG CODE

```

module u1(ifc1_x,
          ifc1_y,
          ifc1_z);
    output ifc1_x;
    output ifc1_y;
    output ifc1_z;
endmodule
module u2(ifc2_x,
          ifc2_y,
          ifc2_z);
    input ifc2_x;
    input ifc2_y;
    input ifc2_z;
endmodule
module top();
    wire sg_x_d;
    wire sg_y_d;
    wire sg_z_d;
    u1 u1;
    u2 u2;
endmodule

```

2.2.1.3 Units

CSL Units act like Verilog modules. Units can be instantiated within other units. A unit represents a scope and every variable, signal, or port defined in that unit will have its name prepended with the unit name.

cs1_unit *unit_name*;

DESCRIPTION :

CSL unit class declaration. For definition see 2.2.1.3 Units above. It represents a scope delimited by curly braces and contains a constructor.

The CSL unit class declaration alone does not translate into a Verilog module. Only a unit definition results in a Verilog module.

[CSL Interconnect Command Summary]

EXAMPLE :

In the following example is a unit called *u_unit* .

u_unit

CSL CODE

```
cs1_unit u_unit{
  u_unit() { }
};

cs1_unit u_top{
  u_unit u1;
  u_top() { }
};
```

VERILOG CODE

```
module u_unit();
endmodule

module u_top();
  u_unit u1;
endmodule
```

2.2.1.4 Units:Ports

A port is a directed signal type. Ports are declared in the interface of a unit or in an unit and are used to make connections (input or output signals, or both).

csl_port

```
port_name(port_direction[,port_type][,range][,width][,upper_index,lower_index][,bitrange_object_name]);
```

DESCRIPTION :

Port object declaration. These types can be declared inside a unit class definition and are thus added to the unit's default interface, or inside an interface definition and become part of the respective interface. Parameters that can be passed to a port declaration are:

- port direction, specifying the direction of the signal passing through the port;

TABLE 2.14 Port direction specifiers

Port direction	Description
input	input port
output	output port
inout	inout port

- port type, details the signal type passing through the port

TABLE 2.15 Port types

Port type	Description
wire	Creates a wire type port
wand	Creates a wand type port
wor	Creates a wor type port
tri	Creates a tri type port
triand	Creates a triand type port
trior	Creates a trior type port
tri0	Creates a tri0 type port
tri1	Creates a tri1 type port
supply0	Creates a supply0 type port
supply1	Creates a supply1 type port
reg	Creates a reg type port. Note: only allowed for output ports
integer	Creates a integer type port. Note: only allowed for output ports
time	Creates a time type port. Note: only allowed for output ports

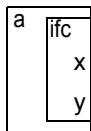
- range, specifies the range of the port and can be a bitrange object, a width numeric expression or a upper index, lower index pair of numeric expressions

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example is a unit called *a* which contains an interface called *ifc*. The interface has two ports : *x* and *y*;

FIGURE 2.12



CSL CODE

```

csl_interface ifc{
  csl_port x(input,wire,4), y(output,reg,4);
  ifc(){}
};

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

VERILOG CODE

```

module a(ifc_x,
         ifc_y);
  input [3:0] ifc_x;
  output reg [3:0] ifc_y;
endmodule
  
```

csl_port *port_object*(*port_hierarchical_identifier*);

DESCRIPTION :

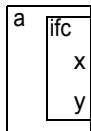
This acts like a copy constructor for a port object. The *port_object* object is created by copying all the properties found in the port object specified by the *port_hierarchical_identifier* (*port_direction*, *port_type* and *range*).

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example is a unit called *a* which contains an interface called *ifc*. The interface has two ports : *x* and *y*;

FIGURE 2.13



CSL CODE

```

csl_unit a{
csl_port x(output), y(output);
a(){}
};

csl_unit b{
csl_port x1(a.x), y1(a.y);
a a;
b(){}
};
  
```

VERILOG CODE

```

module a(x,y);
    output x;
    output y;
endmodule

module b(x1 , y1);
    output x1;
    output y1;
a a();
endmodule
  
```

```
port_object.reverse();
```

DESCRIPTION :

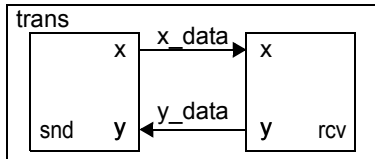
Reverse the direction of a port. This method is called on a port object and it reverses its direction. This method can only be called for input/output ports.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example the method is used to change the direction of the ports *x* and *y*.

FIGURE 2.14 A block named *trans* with two instances named *snd* and *rcv* interconnected

**CSL CODE**

```

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

csl_unit rcv{
    csl_port x(snd.x), y(snd.y);
    rcv(){
        x.reverse();
        y.reverse();
    }
};

csl_unit trans{
    csl_signal x_data, y_data;
    snd snd( .x(x_data), .y(y_data));
    rcv rcv( .x(x_data), .y(y_data));
    trans() {}
};

```

VERILOG CODE

```

module snd(x,
           y);
    output x;
    input y;
endmodule

module rcv(x,
           y);
    input x;

```

```

    output y;
endmodule

module trans();
    wire x_d;
    wire y_d;
    snd snd(.x(x_d),
            .y(y_d));
    rcv rcv(.x(x_d),
            .y(y_d));
endmodule

```

```
port_object.set_width(numeric_expression);
```

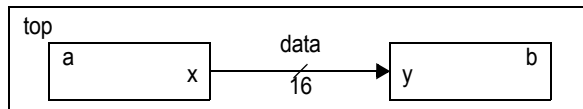
DESCRIPTION :

Sets the width of a port. The bitrange of the port is automatically created from the width. If the width has already been set the cslc will flag an error.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.6 is illustrated the use of signal *data* to connect *a* and *b* units inside a *top* unit. The *a* unit has an output port called *x* and the *b* unit has an input port called *y*. The width of the ports is set using *set_width* command.

FIGURE 2.15**CSL CODE**

```

csl_unit a{
  csl_port x(output);
  a() {
    x.set_width(16); }
};
csl_unit b{
  csl_port y(input);
  b() {
    y.set_width(16); }
};
csl_unit top{
  csl_signal data(wire,16);
  a a( .x(data));
  b b( .y(data));
  top() {}
};

```

VERILOG CODE

```

module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));
  b b(.y(data));

```

```
endmodule
```

```
port_object.set_bitrange(bitrange_object);
```

DESCRIPTION :

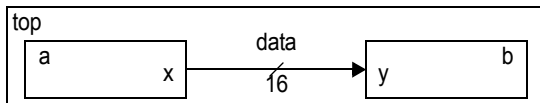
Sets the bitrange object of a port. The bitrange object passed as a parameter to the `set_bitrange()` method must already be declared. After the assignment of the bitrange the port contains a reference to the bitrange object *bitrange_object*.

[CSL Interconnect Command Summary]

EXAMPLE :

In the example from Figure 2.7 is illustrated the use of a signal and input/output ports to connect *a* and *b* units inside a *top* unit. The width of the ports is set using *set_bitrange* commands.

FIGURE 2.16



CSL CODE

```

csl_bitrange br1(16);
csl_unit a{
  csl_port x(output);
  a(){
    x.set_bitrange(br1);}
};
csl_unit b{
  csl_port y(input);
  b(){
    y.set_bitrange(br1);}
};
csl_unit top{
  csl_signal data(br1);
  a a( .x(data));
  b b ( .y(data));
  top(){};
};


```

VERILOG CODE

```

module a(x);
  output [15:0] x;
endmodule
module b(y);
  input [15:0] y;
endmodule
module top();
  wire [15:0] data;
  a a(.x(data));

```



```
b b(.y(data));  
endmodule
```

```
port_object.set_range(lower_limit, upper_limit);
```

DESCRIPTION :

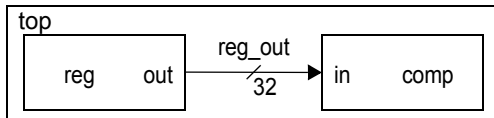
Set the bitrange for the *port_object* using the *lower_limit* and *upper_limit* delimiters.

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example we have one 32-bit signal named *reg_out* which connect a register called *reg* and a comparator called *comp*. The width of ports *out* and *in* is set using *set_range* command.

FIGURE 2.17 A comparator and a register



CSL CODE

```

csl_unit reg1{
  csl_port out(output);
  reg1(){
    out.set_range(0, 31);}};
csl_unit comp{
  csl_port in(input);
  comp(){
    in.set_range(0, 31);}
};
csl_unit top{
  csl_signal reg_out(32);
  reg1 reg1( .out(reg_out));
  comp comp( .in(reg_out));
  top(){};
}

```

VERILOG CODE

```

module reg1(out);
  output [0:31] out;
endmodule
module comp(in);
  input [0:31] in;
endmodule
module top();
  wire [0:31] reg_out;
  reg1 reg1(.out(reg_out));
  comp comp(.in(reg_out));
endmodule

```



```
port_object.set_offset(numeric_expression);
```

DESCRIPTION :

Sets the value to be added to both lower and upper index of the port. This method cannot be called if the port has not set a width before. In this case will be an error.

[CSL Interconnect Command Summary]

EXAMPLE :

Sets the offset for the port *p1*.

CSL CODE

```
csl_unit u1{
    csl_port p1(input,4);
    csl_port p2(output,8);
    u1(){
        p1.set_offset(32);
    }
};
```

VERILOG CODE

```
module u1(p1,
          p2);
    input [35 : 32] p1;
    output [7 : 0] p2;
endmodule
```

```
port_object.set_type(csl_port_type);
```

DESCRIPTION :

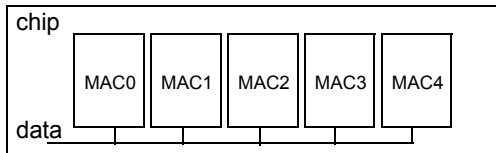
Assign a net type to *port_object*. Applies to both signals and ports. For input ports cannot be set the type register (reg). The *csl_port_type* can be one of the port types from Table 4.11Port types .

[*CSL Interconnect Command Summary*]

EXAMPLE :

This example illustrates the `set_type()` method applied on both port objects and signal objects

FIGURE 2.18 Connecting data to units



CSL CODE

```
csl_unit MAC{
    csl_port p_in(input);
    MAC() {
        p_in.set_type(integer);
    }
};

csl_unit chip{
    csl_signal data(8);
    MAC MAC0( .p_in(data));
    MAC MAC1( .p_in(data));
    MAC MAC2( .p_in(data));
    MAC MAC3( .p_in(data));
    MAC MAC4( .p_in(data));
    chip() {
        data.set_type(integer);
    }
};
```

VERILOG CODE

```
module MAC(p_in);
    input integer p_in;
endmodule

module chip();
    integer [7:0] data;
    MAC MAC0( .p_in(data));
    MAC MAC1( .p_in(data));
    MAC MAC2( .p_in(data));
```

```
MAC MAC3( .p_in(data));  
MAC MAC4( .p_in(data));  
endmodule
```

```
port_object.set_attr(csl_port_attr);
```

DESCRIPTION :

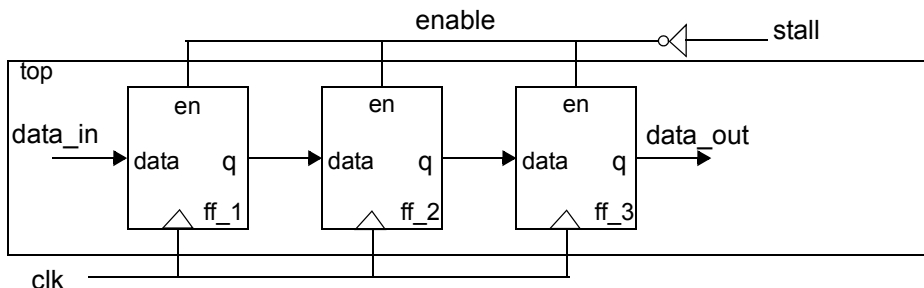
Assign an attribute to *port_object*. Attributes describe what the port is used for in the design. An attribute listing is given in Table 2.13

[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 *ff_1*, *ff_2* and *ff_3* and additionally stall *st* and *data_in*, *data_out* signals. When the *autorouter* is called, it connects the ports and signals which have the same attribute and pertain to units in a sibling relationship.

FIGURE 2.19



CSL CODE

```
csl_unit ff{
  csl_port q(output), data(input), e(input), c(input);
  ff(){
    e.set_attr(csl_enable);
    c.set_attr(csl_clock);
  }
};

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

VERILOG CODE

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

```

    input e;
    input c;
    output q;
endmodule

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

```

2.2.1.5 Units: Interface

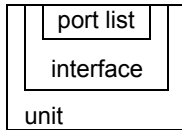
The interface is a container for other interfaces and ports. An interface is a description for the pin outs for the block. It is used to create groups of ports and other interfaces and can be instantiated in units. A unit has a default interface. Trees of interfaces can be built.

csl_interface *interface_object_name*;

DESCRIPTION :

Creates a new interface object named *interface_object_name*. This object holds the port list for a unit and vector descriptions for the port signals.

FIGURE 2.20 Interface organization

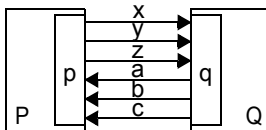


[CSL Interconnect Command Summary]

EXAMPLE :

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

FIGURE 2.21



CSL CODE

```

csl_interface p{
  csl_port x(output,1), y(output,1), z(outut,1), a(input,1), b(input,1),
  c(input,1);
  p(){ }
};

csl_interface q{
  csl_port a(output,1), b(output,1), c(output,1), x(input,1), y(input,1),
  z(input,1);
  q(){ }
};

csl_unit P{
  p p0;
  P(){ }
};

csl_unit Q{
  q q0;
  Q(){ }
};
  
```

VERILOG CODE

```

module P(p0_x,
          p0_y,
  
```

```
        p0_z,  
        p0_a,  
        p0_b,  
        p0_c);  
    input p0_a;  
    input p0_b;  
    input p0_c;  
    output p0_x;  
    output p0_y;  
    output p0_z;  
endmodule  
module Q(q0_a,  
        q0_b,  
        q0_c,  
        q0_x,  
        q0_y,  
        q0_z);  
    input q0_x;  
    input q0_y;  
    input q0_z;  
    output q0_a;  
    output q0_b;  
    output q0_c;  
endmodule
```

```
instance_interface_name.reverse()
```

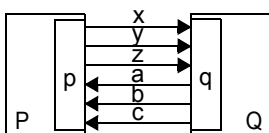
DESCRIPTION :

This method will invert the input ports and make them output ports, while the output ports will become input ports for *instance_interface_name*. The reverse method will not change inout, tri. The method cannot be used with interfaces that have wand or wor types.

[*CSL Interconnect Command Summary*]

EXAMPLE :

In this example two interface objects are created, modified and then assigned to different units. Using the method *reverse()* we will change the directions of ports from the instance *q0* of interface *p*. The output ports will become inputs and the input ports will become outputs.

FIGURE 2.22**CSL CODE**

```
csl_interface p{
  csl_port x(output,1), y(output,1), z(output,1), a(input,1),
  b(input,1), c(input,1);
  p(){}
};

csl_unit P{
  p p0;
  P(){}
};

csl_unit Q{
  p q0;
  Q(){
    q0.reverse(); }
};
```

VERILOG CODE

```
module P(p0_x,
         p0_y,
         p0_z,
         p0_a,
         p0_b,
         p0_c);

  input p0_a;
  input p0_b;
  input p0_c;
  output p0_x;
```



```
    output p0_y;
    output p0_z;
endmodule
module Q(q0_a,
        q0_b,
        q0_c,
        q0_x,
        q0_y,
        q0_z);
    input q0_x;
    input q0_y;
    input q0_z;
    output q0_a;
    output q0_b;
    output q0_c;
endmodule
```

2.2.2 Units: misc

Other options related to units are detailed in this paragraph.

`do_not_gen_cpp()` ;

DESCRIPTION :

This method specifies not to generate the C++ code for the unit and the sub-hierarchy below the unit. This method is used when there is existing C++ code (IP or legacy) from a customer and it's required to use that code by instantiating it in the design without generating a C++ unit from the CSL design. The table below shows the scopes where this method can be called.

TABLE 2.16

method	unit	register	fifo	rf	memory	memory_map	ifc	sg	vector	testbench
<code>do_not_gen_cpp()</code>	X	X	X	X	X	-	-	-	-	X

[CSL Interconnect Command Summary]

EXAMPLE :

In this example the `do_not_gen_cpp()` method is called in a counter register scope and in a unit scope, so for these scopes the C++ code will not be generated. C++ code will be generated only for `u_cnt` unit.

CSL CODE

```

csl_register reg1{
    reg1() {
        set_type(counter);
        set_width(8);
        add_logic(count_direction,up);
        do_not_gen_cpp();
    }
};

csl_unit u_cnt{
    csl_port p1(input,8);
    csl_port p2(output,8);
    csl_port p_clk(input);
    reg1 reg1_i(.reg_out(p2),.clock(p_clk));
    u_cnt() {
        p_clk.set_attr(clock);
    }
};

csl_unit u_top{
    csl_signal s1(8),s2(16),s3(8);
    u_cnt u_cnt(.p1(s1),.p2(s3));
    u_top() {
        s2={s1,s3};
        do_not_gen_cpp();
    }
}

```

```
};
```

C++ CODE

1) .h file

```
using namespace NSCsimLib;

namespace NSCsimGen {

class u_cnt : public CsimUnit {
public:
    RefCsimPortTChar p1;
    RefCsimPortTChar p2;
    RefCsimClock p_clk;
    RefCsimUnit reg1_i;
    //functions
    void defaultInitialize();
    void connect();
    virtual void allocate() = 0;
    virtual void initialize() = 0;
    virtual void execute() = 0;
    //constructor
    u_cnt() : CsimUnit(RefString(new std::string("u_cnt"))) {}
};
}
```

2) .cpp file

```
#include "u_cnt.h"
namespace NSCsimGen {

void u_cnt::defaultInitialize() {
    //port allocations
    p1 = CsimPortTChar::build(RefString(new std::string("p1")),8,
getThis(), PORT_DIR_INPUT);
    p2 = CsimPortTChar::build(RefString(new std::string("p2")),8,
getThis(), PORT_DIR_OUTPUT);
    //port registrations
    addConnectable(p1);
    addConnectable(p2);
    //building vector writers
    //initializers
}
```

```

allocate();
//instance registration
regl_i->setParent(getThis());
regl_i->setInstanceName(RefString(new std::string("regl_i")));
//clock connections
    p_clk->connectToClock(regl_i,
(boost::static_pointer_cast<regl>(regl_i)->clock, RefString(new
std::string("clock"))));
        //default initis
regl_i->defaultInitialize();
connect();
initialize();
}

void u_cnt::connect() {
    p2->connect(boost::static_pointer_cast<CsimPortTChar>(regl_i-
>getSignalByName(RefString(new std::string("reg_out"))));
}
}

```

```
do_not_gen_rtl();
```

DESCRIPTION :

This method specifies not to generate the RTL code for the unit and the sub-hierarchy below the unit. This unit is used when there is existing RTL code (IP or legacy) from a customer and it's required to use that code by instantiating it in the design without generating a RTL unit from the CSL design. The table below shows the scopes where this method can be called.

[*CSL Interconnect Command Summary*]

TABLE 2.17

method	unit	register	fifo	rf	memory	memory_map	ifc	sg	vector	testbench
do_not_gen_rtl()	X	X	X	X	X	-	-	-	-	X

EXAMPLE :

This example shows that unit *unit1* is instantiated in both units, *u1* and *u_top*, but RTL code is generated only for unit *u1* and for *u1* sub-hierarchy, because in *u_top* is used *do_not_gen_rtl()* method. RTL code for *fifo f1* will not be generated because the method is called inside of *fifo* scope.

CSL CODE

```

csl_fifo f1{
    f1() {
        set_width(16);
        set_depth(1<<8);
        do_not_gen_rtl();
    };
};

csl_unit unit1{
    csl_port p1_u1(input,2);
    csl_port p2_u1(output,4);
    csl_signal s1(2);
    unit1() {
        p2_u1= {p1_u1,s1};
    };
};

csl_unit u1{
    csl_port in0(input),data_in(input,2);
    csl_port in1(input);
    csl_port sel(input);
    csl_port out(output),data_out(output,4);
    unit1 ui(.p1_u1(data_in),.p2_u1(data_out));
    f1 fi;
    u1() {
        out= (sel)? in1: in0;
    };
};

csl_unit u_top{

```

```

csl_port p1(input,16);
csl_port p2(output,16);
csl_signal s1(16), s2;
f1 f1i;
unit1 unit_i;
u1 u1i;
u_top(){
    p2= s1 + p1;
    u1i.out.connect(s2);
    do_not_gen_rtl();
}
};

```

VERILOG CODE

```

module unit1(p1_u1,
             p2_u1);
    input [1:0] p1_u1;
    output [3:0] p2_u1;
    wire [1:0] s1;
    assign p2_u1 = {p1_u1,s1};
    `include "unit1.logic.v"
endmodule

module u1(in0,
          data_in,
          in1,
          sel,
          out,
          data_out);

    input in0;
    input [1:0] data_in;
    input in1;
    input sel;
    output out;
    output [3:0] data_out;
    assign out = sel ? in1 : in0;
    unit1 ui(.p1_u1(data_in),
             .p2_u1(data_out));

    f1 fi();
    `include "u1.logic.v"
endmodule

```

gen_unique_rtl_modules();

DESCRIPTION :

This method is used to specify that for each instance of the unit for which it is called there should be a unique RTL module declaration generated. Also the instantiation will be updated with the unit's new name. This is done by prefixing the unit name in the declaration with the name of the unit where the it is instantiated.

When gen_unique_rtl_module() and do_not_gen_rtl() methods are used together in the same scope, the compiler gives an error. These two methods should not be used together in the same scope.

The table below shows the scopes where this method can be called.

TABLE 2.18

method	unit	register	fifo	rf	memory	memory_ map	ifc	sg	vector	testbench
gen_unique_rtl_module()	X	X	X	X	X	-	-	-	-	X

[CSL Interconnect Command Summary]

EXAMPLE :

CSL CODE

```

csl_unit a{
    a() {
        gen_unique_rtl_modules();
    }
};

csl_unit top{
    a a0;
    a a1;
    top() {}
};

```

VERILOG CODE

```

module a0();
endmodule

module a1();
endmodule

module top();
    top_a0 a0();
    top_a1 a1();
endmodule

```

2.2.3 Clock domains

All connectivity elements can be tied with a clock domain. A clock domain is physically determined by a clock generator and all devices driven by that clock generator. The clock (clocks) enter a design through the top unit and propagate to all instances contained. A port carries a clock signal if it has the attribute clock set to it by the *set_attr(clock)* method.

Connectivity elements that can be associated with a clock name are:

- ports
- signals
- signal groups
- interfaces

This association does not imply a connection at this point - it only specifies the clock domain a connectivity element belongs to, so that it could be used later (e.g. in the *register_ios()* method, when a flip-flop is inferred to register a port it will be driven by the clock associated with that port).

Also, units can be associated with clocks, but this will not work if such a unit contains elements that are driven by more than one clock. It is useful however if a unit and all its components are driven by one clock - in this case the *set_clock()* command applies to all connectivity elements from that unit.


```
[object_name.]set_clock(clock_name);
```

DESCRIPTION :

Associates the object upon which it's called to the clock domain determined by the clock_name. clock_name is the identifier for a port/signal of type clock.

Objects on which *set_clock()* can be called upon:

- ports
- signals
- signal groups
- interfaces
- units

The method can be called inside an interface for a port or inside a signal group for a signal. If a port from an interface has clock *p_clk* set, the interface instance(s) can have a different clock associated.

[CSL Interconnect Command Summary]

EXAMPLE :

In this example the *set_clock()* method is called on unit instance *a0*. This unit will be associated to the clock domain determined by *clk*.

CSL CODE

```

csl_unit a {
    csl_port p_clk(input);
    a () {
        p_clk.set_attr(clock);
    }
};

csl_unit b {
    csl_port clk(input);
    a a0;
    b () {
        clk.set_attr(clock);
        set_clock(clk);
    }
};

```

VERILOG CODE

```

module a(p_clk);
    input p_clk;
endmodule

module b(clk);
    input clk;
    a a0(.p_clk(clk));
endmodule

```

EXAMPLE :

In this example units *a* and *b* are associated to different clock domains, *clka* and *clkb*.

CSL CODE

```

csl_unit a{
    csl_port p_clka(input);
    a(){
        p_clka.set_attr(clock);
    }
};

csl_unit b{
    csl_port p_clkb(input);
    b(){
        p_clkb.set_attr(clock);
    }
};

csl_unit top{
    csl_port clka(input);
    csl_port clkb(input);
    a a0;
    b b0;
    top(){
        clka.set_attr(clock);
        clkb.set_attr(clock);
        a0.set_clock(clka);
        b0.set_clock(clkb);
    }
};

```

VERILOG CODE

```

module a(p_clka);
    input p_clka;
endmodule

module b(p_clkb);
    input p_clkb;
endmodule

module top(clka,clkb);
    input clka;
    input clkb;
    a a0(.p_clka(clka));
    b b0(.p_clkb(clkb));
endmodule

```



```
(hid[.ps] | expression).connect_by_name((hid[.ps] | expression)
[,f2a_name]);
```

[CSL Interconnect Command Summary]

DESCRIPTION :

The method makes a connection between two endpoints. The complete path is given relative to the scope where the connect command is called:

hid = identifier(.identifier)*,

ps =part select,

expression = concatenation, replications, operators expression;

NOTE: More details about this method can be found in csl_auto_router.pdf document.

```
scope.connect_units(scope [,"f2a_prefix"]);
```

[CSL Interconnect Command Summary]

DESCRIPTION :

A connection between 2 scopes. The complete path is given relative to the scope where the connect command is called. The connection is made between connectivity elements with the same name from the two scopes.

scope = identifier(.identifier)*

NOTE: More details about this method can be found in csl_auto_router.pdf document.

```
hid[.ps].connect_by_pattern(hid_pattern[.ps] [,f2a_name]);
```

[CSL Interconnect Command Summary]

DESCRIPTION :

The method creates a connection between two endpoints. The hid is the path to the first endpoint relative to where the command is called. The second endpoint is given as a path pattern (Not the full path, but only a part of the path that has to be matched in the hierarchy).

hid = path to connectivity object;
 hid_pattern = path pattern to connectivity object;
 ps = part select;

NOTE: More details about this method can be found in csl_auto_router.pdf document.



