CHAPTER 4 CSL Interconnect

TABLE 4.1 Chapter Outline

4.1 Definitions

Complete the table with definitions

TABLE 4.2 Definitions Table

Terms	Definitions
RTL interconnect	Circuit components at RTL level that route data between datapath units. Consist of data transfer wires, clock distribution network, steering logic(ex. multiplexers).
Intermediate connect	A connection between two units at different non consecutive hierarchy levels.
Leaf level file	File discrobing modules at the lowest hierarchy level.

4.2 CSL Interconnect Overview

4.2.1 Purpose/Description

The CSL language is divided into different categories one of which is the CSL interconnect specification. This chapter explains the CSL interconnect specification and how the user creates a CSL interconnect specification file. The user writes CSL interconnect code which specifies the RTL design, the hierarchy and the connection instances. The CSL interconnect specification files are compiled and analyzed by the CSL compiler (cslc). The cslc compiles the CSL code and creates a design database, analyzes the design database, generates RTL code (Verilog or VHDL), C++ code and generates documentation.

4.2.2 Benefitsj

The CSL interconnect specification and the cslc are used to eliminate interface differences between modules and models.

<INTEGRATE THIS TEXT>

4.2.3 Usage modules

Centralized interconnect specification and generation. Is this a title? or what?
One person creates the centrally generated RTL interconnect files.

FIGURE 4.1

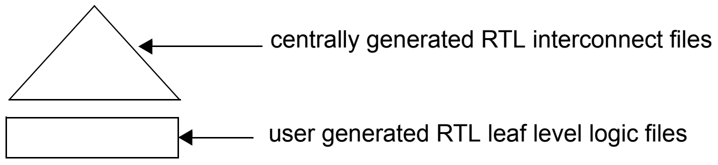
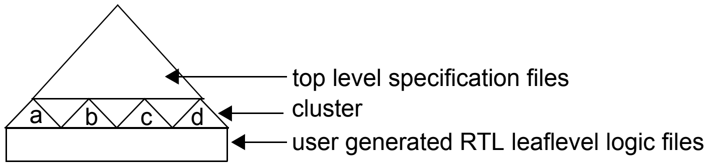

centrally generated RTL interconnect files
user generated RTL leaf level logic files

FIGURE 4.2


top level specification files
cluster
a
b
c
d
user generated RTL leaflevel logic files

The lines from the figures are not straight.
You have to explain the figures properly. The sentences aren’t connected.
Cluster owners create their own csl interconnect specification files.
A top level integration person creates the top level specification files.
Users create csl connection statements to connect signals.
There are no ports defined in the CSL Verilog models. Instead, CSL statements are inlined in Verilog modules. Then, the auto router creates the ports in each module based on the CSL statements which specify the connections through modules interfaces.
<INTEGRATE THIS TEXT>

4.2.4 Compilation Process

The **cslc** compiles CSL files. The **cslc** compilation process is described in chapter CSL Language or overview(not sure yet).

The **cslc** interconnect syntax is documented in the CSL Interconnect language summary. The cslc interconnect compilation steps include the following:

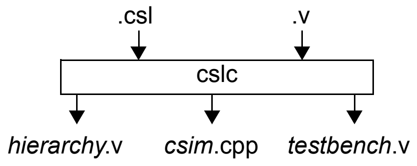
- interconnect generation
- generation of a set of RTL and C++ files
- building the interconnect
  - each module contains an interface which is used to connect instances of this module to other module instances
  - modules contain instantiations of modules
  - wires which create the connections between the instances in the module
  - the connections between the clocks and the state elements(SE) are also checked for glitches
  - suffixes
  - all SE’s are labeled with a valid phase suffix and pipestage - this can be used later/later on to check if the SE’s are correctly connected(move to cls_clk synch overview analysis)

The CSL interconnect is used to eliminate interface differences between modules.

4.2.5 Flow

Figure 4.3 gives the flow for creating hierarchical interconnections between the modules of a design. Cslc receives at input .csl and .verilog files that define interconnections between modules. The compiler generates a hierarchy file, a C++ model (csim.cpp) and a testbench file that can be used to the design from the point of view of interconnections.

FIGURE 4.3 cslc interconnect flow



4.2.6 Input Files

The user creates the CSL interconnect specification file (.csl) which contains the following design information:

- the design hierarchy:
  - the names of the units
  - the names of the units instances
- each unit interface
- connectivity:
  - the signals which connect the unit instances to each other
  - the formal to actual name mappings in the unit’s instantiations

4.2.7 Output Files

- RTL csl interconnect files
- RTL port list & declaration files
- C++ code
- Documentation
- Reports
- Interconnect files contents What are these?!?

The cslc reads the CSL interconnect specification and generates Verilog output files. The Verilog files can contain the following elements//what elements?

4.2.8 Analysis

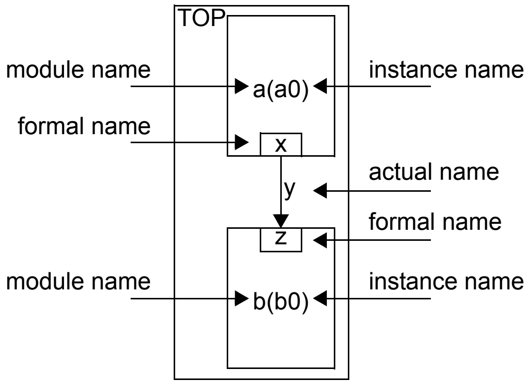
A user creates the CSL interconnect specfication. The specification may contain CSL syntax errors, design errors (e.g. unconnected signals, width mismatches between signals), and other errors. The cslc will read and parse CSL language files and check the CSL interconnect specification for correctness, and report errors.

- the analyzer will check connection for correctnesss and will issue following signal/ port instance reports:
  - each instance’s formal to actual mappings
  - each signal’s fanin and fanout
  - usage information for each bit in each port

4.3 CSL Concepts

4.3.1 What is specified using this part of the CSL language

FIGURE 4.4



Signal connectionsYou can replace 'what it is specified...' with "Signal Connections" / "Port and Signals Connections" because this is what it is about.
Signals are connected in many different ways In RTL languages.What is the connection between the sentence about ports and the one about signals?
An output port can connect to one or more ports in other instances. A signal which has more than one bit is called a multi-bit signal. Multi-bit signals can connect to other multi-bit signals of the same width. Parts of multi-bit signals can connect to other single or multi-bit signals.
You can place here the sentence about ports.The RHS of an assignment or actual name in a formal to actual port connection can be one of the following:

- signal with or without a bit range
- concatenation
- constant
- expression

The LHS of an assignment can be one of the following:

- signal with or without a bit range
- concatenation

The interconnect generation tool guarantees that the interfaces between modules will be consistent and that all the bits in each signal are connected (to another bit in another signal).
The CSL interconnect specification is built using the following components:

- signals (nets)
- bit ranges
- concatenations
- port lists
- formal to actual mappings
- modules (units)
- assignments

4.3.2 Language usage complete

4.3.2.1 Connecting the generated Verilog files

First of all , are the leaf level and hierarchical files 'generated Verilog files'? If they do, then say it before passing to the way in which they connect.
There are three ways to connect leaf level and hierarchical files.

1. Use CSL statements in the RTL to create ports and connection signals

2. Write a separate csl specification file that creates the design hierarchy and the design interconnect. CSL compiler can produce an include file for the leaf level files

which contains the port list for the leaf level module. The leaf level module includes the generated port list file.

3. Read the port list from a Verilog instance and connect it to CSL signals in other files.

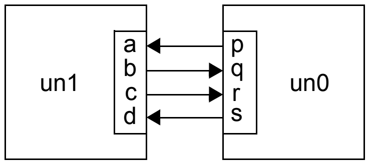
4.3.2.2 Unit (module) specification

The CSL unit is a similar construct to the Verilog module. The CSL unit contains an interface, signals (wires, regs) and instances of CSL objects and other units. 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.
add outlines about units, ports, csl_objects here

4.3.2.3 Connections between units

Interfaces
An interface is a container for the port list of a unit.

FIGURE 4.5 Unit Interface



Units can be connected to other units (un1 to un0) explicitly by using the CSL connection command or the job can be done automatically by the auto router. (connected explicitly = are directly connected (?)) What a,b,p... represent? There are three situations to deal with when connecting units within a design hierarchy:

- parent to child connection
- child to parent connection

- sibling to sibling connection

TABLE 4.3 port inference rules

driver and receiver connection type	action	description
parent to child connection or child to parent connection	preserve port direction	the port direction is preserved because the output of a parent unit is connected to the output of a child unit and the input of a parent unit is connected to the input of a child unit and vice versa
sibling to sibling connection	reverse port in direction	the port direction is reversed because the output of a unit is connected to the input of a unit in the same scope and the input of a unit is connected to the output of a unit in the same scope

CSL Parts
Signal declaration The design can be divided into multiple csl specification files.(Nonetheless)The same signal names can be used in many different files.These signals will be connected by the CSL compiler using the autoconnect feature. Thus, the signals in the design database are automatically connected.

4.3.2.3.1 Autorouting Complete this

4.3.2.3.2 Connectivity reports

A connectivity report is generated by the cslc after the input files have been compiled. The connectivity report analyzes the connectivity database and reports the number of times each bit in each signal in each interface is used.
The report looks like this:

```
signalname range bit number 76543210
foo [7:0]                   10211111
```

In the example shown above, the bit 6 is not connected and the bit 5 is connected twice. All other bits are connected once.
The list of files in a design is stored in a *design_name*.vc file. The CSL compiler generates the RTL interconnect files , a .vc file and interface files for each leaf used verilog module(in?).
The interface files are included in each designers leaf level module:

`'include "leaf_module.port"`

The leaf level files contain logic. The RTL interconnect files connect the leaf level files.If there are several hierarchical levels of interconnect and tree files, the interconnect can be built in a bottom-up manner. This has the advantage of breaking down the problem into manageable chunks.

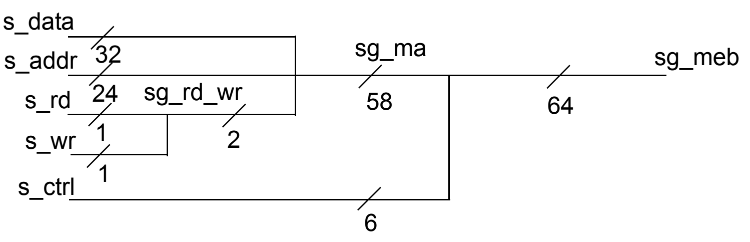
4.3.2.4 Creation of groups and vectors of signals

Signals correspond to nets or ports in a design. Signals have the following characteristics:

- signal type (wire, reg, tristate, wand, ...)
- width
- attribute type (reset, enable, clock)

Signal groups and interfaces. Signals can be grouped together to form a single object which is more efficient to manipulate than performing the same operation on individual signals. Signals or interfaces can be added to vectors which get values in algorithmic or behavioral simulators. The signal groups and vectors can be used later to construct testbenches.

FIGURE 4.6 Signal Group



//CSL code

```
 csl_signal_group sg_rd_wr {
 csl_signal s_rd;
 csl_signal s_wr;
};

csl_signal_group sg_ma {
 csl_signal s_data(32);
 csl_signal s_addr(24);
```

```
     sg_rd_wr sg_rd_wr_i;
    };

    csl_signal_group sg_meb {
     csl_signal s_ctrl(6);
     sg_ma sg_ma_i;
    };
```

The **units** contain other units, logic, and signals. A unit corresponds to a Verilog module or a VHDL entity.
A basic operation in the CSL interconnect specification includes the following:

- commands to execute
- reports generation
- code generation

4.3.2.5 Formal and Actual Name Mapping Examples

The instances have a formal port list which holds the names of the ports in the module port list, and an actual port list which contains the signals or expressions that are connected to each of the formal names. This is called the formal to actual mapping. CSL supports formal to actual name mappings.(this should have been explained earlier because the “formal to actual mapping” was used before in the text.)
The ports in instances are connected to upper level signals and expressions.
The following is a list of actual types of formal to actual mapping.

- expression
- constant
- concatenation
- rename
- port select

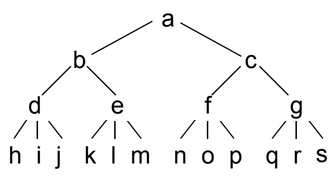
The following is a list of formal to actual mappings

TABLE 4.4 Formal to Actual connection

Actual arg	Example types
expression	w w0 (.x(a&b))
constant	w w0 (.x(1’b0))
concatenation	w w0 (.x({a,b,c,d}))
rename	w w0 (.x(y))
port select	w w0 (.x(p[7:4]))

Hierarchy Slice
A hierarchy slice is a set of modules that has the structure of a tree with n levels(Figure 4.7).

FIGURE 4.7



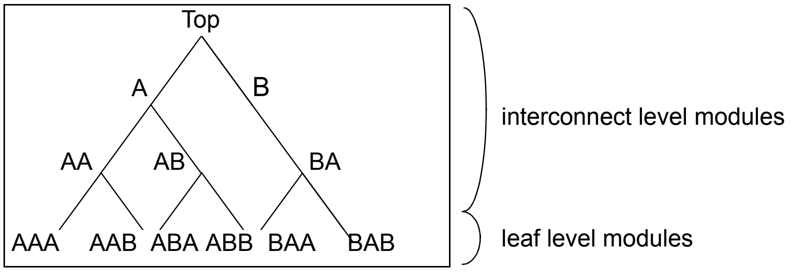
4.3.2.5.1 Interconnect and leaf level modules

The leaf level modules include:

- module name
- port list
- csl include file directives
- include files directives
- local signal declarations and logic
- modules interfaces

Figure 4.8 shows the design hierarchy for an example design.The interconnect level modules and the leaf level modules are shown.

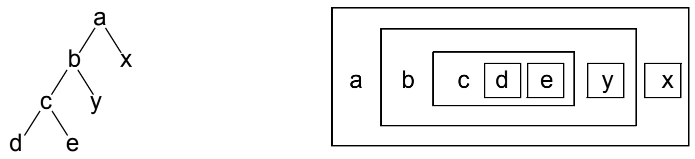
FIGURE 4.8 Interconnect and leaf level modules



4.3.2.5.1.1 SubTree

A subtree is a portion the design hierarchy (tree), containing a set modules that can be viewed as a complete hierarchy (tree) in itself. The subtree extends from the root module of the subtree to the bottom of the design hierarchy tree. All modules between the root of the subtree and the bottom of the subtree are included in the subtree.

FIGURE 4.9 Design hierarchy and block diagram example



In Figure 4.9 is given a design hierarchy and the corresponding blobk diagram. Unit a contains units b and x.Unit b contains units c and y. Unit c contains units d and e.(before the figure)
The following Verilog code implements the module hierarchy in Figure 4.9

```
module a();
  b b0();
  x x0();
endmodule

module b();
  c c0();
  y y0();
endmodule

module c();
  d d0();
  e e0();
endmodule

module d();
endmodule

module e();
endmodule

module y();
endmodule

module x();
endmodule
```

4.3.2.5.1.2 Instance Hierarchies

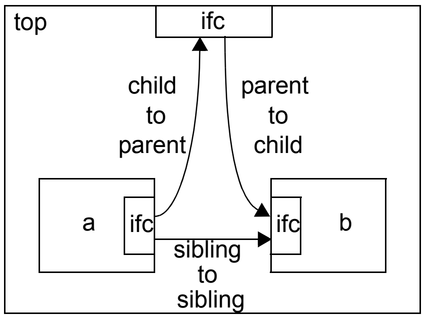
The hierarchical description of a design can be created using the CSL language.

4.3.2.5.1.3 Creating Module Instances

Top is a module which is declared using CSL. We add other modules as children to their parent modules using the **add_instance** keyword.

Figure 4.10 shows a simple unit hierarchy: a and b are siblings in unit top. When connecting unit a to top (child to parent connection) or top to b (parent to child connection), the port directions in the interfaces of the two units must be preserved. When making a sibling to sibling (eg. a to b) connection the port directions must be reversed.

FIGURE 4.10 Units hierarchy



ifc = interface

To connect unit ports, the connect method has to be used for port objects:

```
port_name.connect(port_name);(move to examples APP note)
```

4.3.3 CSL Interconnect Analysis

The variable CSL_SIGNAL_NOT_DEFINED_ERROR_LEVEL sets the warning/error level for CSL connection warnings and errors.

There are several different kinds of conditions that the different levels will check.

- not defined and used on RHS
- not defined and used on LHS
- defined and not assigned on LHS and used on RHS
- defined and assigned on LHS and used on RHS
- check for meta language grammar errors
- report errors
- report in connected to out
- report out connected to in , etc

- report width mismatches
- report formal to actual name mapping errors
- report name duplication
- report signal declared but not used
- report signal used but not declared

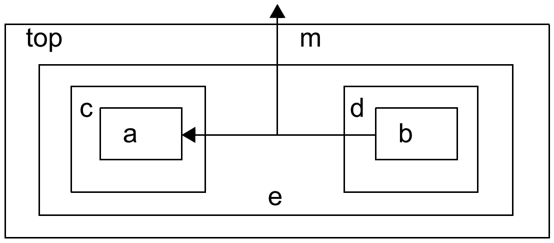
TABLE 4.5 Signal checker truth table

declared	asigned	used	warn/error
yes	no	no	warning
yes	yes	no	warning
no	yes	yes	error

- report continuous assigns
- find a directive to extract interface from a Verilog file and insert it into data base
- auto connect signals

Checker will make a spanning tree between all modules in the design to ensure they can be connected.
Search each module for child modules. Detect if there is another level of hierarchy below the current level.

FIGURE 4.11 Interconnect Analyses

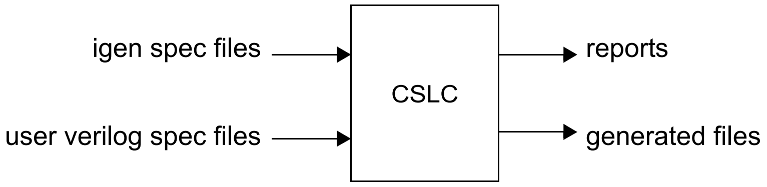


The user specifies the leaf level and the top modules that the signal(s) are connected to.

There will be nets which are unconnected/undriven. The interconnect checker will flag those nets as unconnected. An ignore command may be executed by the CSL interconnect model to mark the unconnected signal with an ignore attribute. The checker will ignore the signal and not report it unconnected. Note that a range in a signal may be ignored.

4.3.3.1 Checking the interconnect in the CDOM

FIGURE 4.12 CSL Interconnect Flow



The CDOM will be checked to ensure the following:

- unique instance names with no duplications inside the same scope;
- unique template names, every module being instantieted3. should have a unique name;
- unique signal names with no duplicate identifiers within the same scope;
- check the formal to actual mappings;
- check for formal to actual width mismatches;
- check for width mismatches in assignment statements;
- check for constants driving inputs of the right width;
- check multiple drivers of a bit; if the signal is not a tristate bus then this is a bug;
- check for no driver of a bit;
- an output is not connected to anything;
- flag any intermediate hierarchical blocks that are driving a bit. In other words only the leaf level modules should drive outputs;
- check that all bits in a vector are driving;
- signal route checker is checking if signals can be routed from a module to a another module.

The CSL interconnect a width checker (Width Ck) will check the connections between signals

- Count number of instances.
- Check the width of formals.
- Divide the actual bit range (if its vector). In this case, it is not a vector but it is a regular expression which declares there are 5 signals(q0 to q4). It needs to map those signals one at a time.

(The examples don’t exist - create them or delete this sentence?)In each of the following examples that the CSL code with one or more errors is shown along with the generated verilog code which will either not compile or simulate correctly.

(This text needs a figure or delete altogether)The output x of block q and the input x of the block s. Ths results in a contention error in the signal router since an input from an upper level block cannot be connected to the output of a lower level block.

4.3.4 CSL Interconnect Reports

CSLC can generate connectivity reports for the design:

- Information on the interfaces which have been mapped.
- Information on how many bits out of the bus are used and how many times each bit is used.
- Input port.
- Output port.
- Word width.
- Power estimate of the number of total bits for the word.
- Number of total bits for the entire chip

4.3.5 Inferring interface signals

1. The width of the signal is one bit unless a range is used with the signal.

```
wire bar;
assign foo = bar;  foo is not defined so it is an output. bar is
defined so it is a local input
```

infers:

```
output foo;
```

2. Width of a variable is explicitly defined in the assign statement

```
wire [3:0] b;
assign f[3:0] = b;  the wdith of f is inferred from the width of b
```

infers:

```
output [3:0] f;
```

Note that defines/expressions that reduce to a constant can be used in place of numbers

3. width inference
width_inference_on

```
wire [3:0] b;
assign f = b;  the width of f is inferred from the width of b
```

infers:

```
output [3:0] f;
```

4.3.6 Signal naming convention

Prefix and suffix information can be added to signal names based on CSL directives.
The order for the inter instance signal naming convention may be specified using the following variable:

INTER_SIGNAL_NAMING_CONVENTION

The default word separator character for the signal naming convention is the underscore character "_".
The word separator character for the signal naming convention can be specified in single quotes using the variable

The default naming convention for interfunctional unit block signal names is as follows:
from_unit"_"to_units"_"signal_name"_"signal_type"_"pipe_suffix
The "*from_unit*" and "*to_unit*" can be automatically included on the signal name based on the declared functional unit prefix.
class_prefix(*unit_name*, ID);

4.3.6.1 Verilog Code

- include files
- module contents
    - compile time directives
    - module name
    - module interface
    - signal declarations
    - assign statements

4.3.6.2 CSL interconnect specification wire types

CSL interconnect specification creates all of the Verilog and VHDL net types:

- integer
- reg
- wire
- tri
- tri0
- tri1
- triand
- trior
- trireg

- wand
- wor
- supply0
- supply1
- real
- realtime
- time

CSL can specify the memory type and instantiate memories:

- macro’s
- environment variables
- formal to actual mappings in instance port lists
- actual expressions
    - const
    - concat
    - bit or bit range
    - scalars
    - vectors
- multiple instances
- declarations
- assignments
- compiler directives

4.3.6.2.1 Connecting Signals

Signals are connected in many different ways In RTL languages.

The RHS of an assignment, a select, or an actual name can be one of the following:

- bus split
- group
- vector
- bit
- bus
- bus_part_select
- expr_concat
- expr_constant
- expr

The LHS of an assignment can be one of the following:

- bit
- bus

- bus_part_select

4.3.6.2.2 Instantiating user defined or IP modules

CSL interconnect can instantiate vendor supplied RTL IP files or user defined RTL files. The cslc will parse RTL files and the file will be added to the design hierarchy.

4.3.6.3 Signal specifications

Signals can be grouped together into a named object. CSL has structures that can aggregate a number of signals together. CSL groups can be used in interfaces and are treated as a type.
The CSL output code generator will create code in the target language which uses the most efficient representation in the target language or will the aggregate structures into scalar types (wires) if directed to do so by the user.(this makes no sense)
The following is the list of aggregate data structures that the CSLC can parse:

- verilog 2001 grouping constructs
- signal groups (a list of signals that are assigned group name).

The generated RTL code can create leaf level modules or can create interfaces which are included in designer's leaf level modules:

- leaf module interfaces.
- all module interfaces.
- specified interfaces.

If the signal is connected to a CSL object that has type information associated with the port that the signal is connected to then the CDOM will check that the signal type and the port type match.
We will now define the grammar for the CSL connection statements. Note that signal can be a **wire**, **reg,** or **port**. The connect statement is used to tie together signals that are in the same scope or are in different scopes. The signals can be **wires**, **regs**, or **ports**. The LHS and RHS expressions can contain one or more signals. Regular expressions will be used to refer to **signal_object_names** and **instance_object_names** that have already been declared.

4.3.7 Port inference

Some designers may choose not to explicitly define the inputs and outputs of their modules. Instead they will rely on the cslc. The auto router is a port name and direction inference engine used to connect the port signals with the same names in different units.

There will be different modes of operation for the CSL compiler. The first mode will allow dangling nets (unconnected nets) in the design database. The second mode of operation will allow the designer to add an override attribute in the code which will cause the clsc to ignore a dangling net. The third mode of operation will flag all dangling nets and any dangling net will cause a cslc compilation failure.

4.3.8 Adding parameters to instances

Instance parameters can be specified in the cslc

```
    module_name.parameter(parameter_name, default_value);
```

4.3.9 Module parameters and module parameters override

Instance parameters can be overriden by CSL directives.

4.3.10 Rules for interfaces

1.The CSL interface is not declared in any scope and cannot be referenced until is added to a unit.

2.The interface may be referenced with an HID(hierachical identifier).

3.The instances of the interfaces are added to the units.

4.More than one interface of the same or different types may be added to one unit. The interfaces may be instances of the same or different interfaces.

EXAMPLE :
Add two instances of the same interface to a unit.

CSL CODE

```
    csl_unit b;
    csl_interface myFirstInterface;
    myFirstInterface.add_port(intput, x);
    myFirstInterface.add_port(output, y, 32);
    b.add_interface(myFirstInterface, interface0);
    b.add_interface(myFirstInterface, interface1);
    // access methods
    b.interface0.x;
    b.interface0.y;

    b.interface1.x;
    b.interface1.y;
```

EXAMPLE :
Add two instances of the different interfaces to a unit.

CSL CODE

```
    csl_unit b,top;
    csl_interface myFirstInterface;
    myFirstInterface.add_port(intput, x);
    myFirstInterface.add_port(output, y, 32);

    csl_interface mySecondInterface;
    mySecondInterface.add_port(intput, m);
    mySecondInterface.add_port(output, n, 32);

    b.add_interface(myFirstInterface, interface0);
    b.add_interface(mySecondInterface, interface1);

    // access methods
    b.interface0.x;
    b.interface0.y;

    b.interface1.m;
    b.interface1.n;
```

5.The interface is a container which holds ports and interfaces. Interfaces are recursive.

EXAMPLE :

CSL CODE

```
    csl_interface a0, a1, a2;
    csl_unit b;

    a0.add_port(x);
    a1.add_port(y);
    a2.add_port(z);

    b.add_interface(a0, a0_0);
    a0.add_interface(a1, a1_0);
    a1.add_interface(a2, a2_0);

    // the ports in each interface are accessed with HID's
    a0_0.x;
    a0_0.a1_0.y;
    a0_0.a1_0.a2_0.z;
```

6.Modifying the contents(adding, deleting or changing the port names) of the interface container modifies the contents of all instances of the interface.

7.Interfaces contain only ports and interfaces.

8.The port directions of all signals in an instance of an interface may be reversed using the *reverse()* method except if the signal connected to the port has a *dont_modify_direction* attribute.

EXAMPLE :

A signal with a *tree_driver* attribute means that the signal is the root of a tree.
The tree driver signal drives a fan out tree of signals.
If the tree driver signal is an output of a unit it cannot have its direction changed.
If the tree driver signal is a local signal driving inputs of other units the *dont_modify_direction* attribute applied to ports in the other units.
Examples of signal trees are clock, reset, and control signals such as stall.
Some signals have a *dont_modify_direction* attribute which means that the direction of the signal should not be modified.
The *dont_modify_direction* attribute is set automatically on signals which fan_out from a signal with a *tree_driver* attribute.
The clock should not have its direction changed.
The reset should not have its direction changed.
Control signals should not have their direction changed.

9.Interfaces are automatically connected using the CSL auto router.

10.The CSL HID's can point to either interfaces or ports in interfaces

11.CSL interfaces can be connected using the connect command to ports in Verilog code.

12.The prefix of an interface may be set.
The prefix string is used to prefix the interface instance name in the generated Verilog RTL code.

EXAMPLE :

```
csl_interface a,b,c;
csl_unit i;
i.add_port(x);
a.add_interface(i, i_0);
// prefixes are added to instances
a.i_0.prefix(rs_);
b.add_interface(i, i_0);
// prefixes are added to instances
b.i_1.prefix(ra_);
c.add_interface(i, i_0);
// prefixes are added to instances
c.i_2.prefix(xb_);
```

13.The prefix of an interface may be used in the upper scope and deleted in the lower level scope.

EXAMPLE :

CSL CODE

```
    csl_unit a, b;
    csl_interface i;
    i.add_port(x);
    a.add_interface(i, i_0);
    // prefixes are added to instances
    a.i_0.prefix(rs_);
    a.add_instance(b, b_0);
    b.add_interface(i, i_0);
    // remove the prefix in the lower level
    b.i_0.remove_prefix();
```

If there are two or more instances of the same interface which have been added to a unit then the interface instance names are prefixed to the port name.
If a prefix is added to an interface instance then the prefix is added before the instance name if required and the port name.
Prefixes of output ports of interfaces propagate the prefix to all signals and ports and interfaces which are connected to the output signals of the interface. The prefix propagation stops when a unit or interface is encountered which contains a *no_prefix* attribute which is set with the *no_prefix()* method or the unit or interface has a prefix already set.

Unit prefixes propagate to lower level instances. The lower level instances can only be propagated in the current scope.

(signal, port, signal_group, interface, unit, instance).remove_prefix()
Remove the prefix from an object connected to a object with a prefix.

(signal, port, signal_group, interface, unit, instance).set_local_prefix(optional mask)
Default set the prefix for all local lower level signals, lower level unit instances, ports of lower level units (note that the unit can only be instantiated in the current unit).

(signal, port, signal_group, interface, unit, instance).set_output_prefix()
The prefix for the input signals should be set by the prefix associated with the signal from the upper scope.

14.The ports and signals in the interface group must be referenced with an HID which uses the Instance name of the interface and the port name in the scope of the unit or the fully qualified HID path to the port in the interface instance.

unit scope: *interface_instance_name.port_name*

outside unit scope: *unit_instance_name.interface_instance_name.port_name*

15.The unit's ports in the interface group must be referenced with an HID to avoid conflicts in the case where the same interface is added more than once to a unit.

EXAMPLE :

CSL CODE

```
csl_unit b, top;
csl_interface myFirstInterface;
myFirstInterface.add_port(intput, x);
myFirstInterface.add_port(output, y, 32);

top.add_interface(myFirstInterface, interface0);
top.add_interface(myFirstInterface, interface1);

b.add_interface(myFirstInterface, interface0);
b.add_interface(myFirstInterface, interface1);

// access methods
b.interface0.x;
b.interface0.y;

b.interface1.x;
b.interface1.y;
```

16.Interfaces which have been added to adjacent units (siblings) or to a parent and child in the same scope are automatically connected. No connect signal is required. The same semantic rules for verilog instance declarations and port list is used.

EXAMPLE :

No connect statement is necessary in the example below.
The interface only contains an output.
The interface of unit c drives the interface of unit a and unit b.

CSL CODE

```
csl_unit top,a,b,c;

top.add_instance(a,a_0);
top.add_instance(b,b_0);
top.add_instance(c,c_0);

csl_interface i;
i.add_port(input, x);
a.add_interface(i,i_0);
b.add_interface(i,i_0);
```

```
    c.add_interface(i,i_0);
    c.i_0.reverse();
```

However, if two or more units have an instance of the interface and two or more units have a reversed instance of the interface then the connect command needs to be used.

CSL CODE

```
    csl_unit top, a,b,c,d;

    top.add_instance(a,a_0);
    top.add_instance(b,b_0);
    top.add_instance(c,c_0);
    top.add_instance(d,d_0);

    csl_interface i;

    i.add_port(x);
    a.add_interface(i, i_0);
    b.add_interface(i, i_0);
    b.i_0.reverse();

    c.add_interface(i, i_0);
    d.add_interface(i, i_0);
    d.i_0.reverse();

    a_0.connect(b_0);
    c_0.connect(d_0);
```

17.The RTL code generator will convert the *interface_instance_name.port_name* to *interface_instance_name_port_name* where the period is replaced with an underscore.

18.Interfaces may be copied and modified using the *remove_signal* and *add_port*, *add_interface* methods.

19.Interface instance names are used to auto connect interfaces just as signal and port names with the same name are auto connected by the auto router interfaces in different units with the same *interface_formal_name* and *interface_instance_name* are auto connected by the auto router.

CSL CODE

```
    csl_interface formal_interface_name;
    top.add_interface(interface_formal_name,interface_instance_name);
```

4.3.11 CSL classes - declaration and definition

Following a pattern established by well known object oriented languages, CSL HDL uses a similar concept as C++ classes for various language constructs. In this chapter **csl_unit**, **csl_interface** and **csl_signal_group** classes are discussed, while other chapters deal with other types of classes which are built in the compiler. Common to all CSL classes are the declaration and the definition.
A CSL class declaration is given in the format:

```
    csl_class_type class_name;
```

where `csl_class_type` can be any of the following: **csl_unit, csl_interface**, **csl_signal_group** or other built in classes. A class declaration allows the user to make certain references to a class before its definition as long as the size of the class is not required.

The CSL class definition format is shown below:

```
    csl_class_type class_name { ... };
```

The above syntax defines a new type and enclosed in curly braces are different language constructs depending on the specified csl_class_type.

4.3.11.1 CSL unit class - declaration, defintion, instantiation

CSL unit classes are user defined types. The Verilog direct correspondent is a module. Objects declared or instantiated inside a unit class (signals, ports, interfaces) will output in the Verilog equivalent code as module members. The **csl_unit** declaration is detailed in the commands section: See "csl_unit unit_name;" on page 208.
The definition format for this class is as follows:

```
    csl_unit unit_name {
      csl_declaration
      unit_name(){ //optional CSL constructor
        csl_function_call //optional CSL function calls
      }
    };
```

In the above example csl_declaration can be a signal, port or bitrange declaration or it can be an instantiation of another unit, interface, signal group or built in class. Optionally a unit class constructor can be defined. The unit class constructor holds function calls for the current unit and it's the only scope where such function calls can be made inside the unit definition.
A unit instantiation inside another unit is declared at the csl_declaration level inside a csl_unit definition. The syntax for unit instantiation has similarities to the Verilog instantiation syntax, allowing for formal to actual connections specifier and unit instance parameters override:

```
    unit_name [#((parameter_override_value)+)] (unit_instance_name
    [((.formal_connection_object(actual_connection_object))*)])+
```

The above is the syntax for the csl unit class instantiation described using BNF (Backus-Naur Form). The CSL language reserved symbols are represented in bold, whle user defined variables in italic: *unit_name* is the name of the unit class to be instantiated, *unit_instance_name* is the name of the unit instance, *parameter_override_value* is a numeric expression and *connection_object* ranges from signal and port objects to signal_group and interface objects.

4.3.11.2 CSL signal group class - declaration, definition, instantiation

CSL signal groups classes are scopes that hold references to existing signal objects
The **csl_signal group** declaration is detailed in the commands section: See "csl_signal_group signal_group_name;" on page 204.
A signal group is a scope holder that bundles signals together.
In general terms, a signal group is similar to an interface,
only it deals with signals instead of ports.

USER ACCESS:
A signal group can only be built inside the scope of the design.
It can be instantiated inside a unit or inside another signal group
(therefore signal groups are hierarchical just like interfaces).
Another available option is to build a signal group using an interface:
all the ports in the interface are transformed into signals inside
the signal group.

There is no direct verilog equivalent of a signal group. The verilog
output will only contain the nets and vars built by the adapter out of the
csl signals inside the group. However, the user can use an option
called generateIndividualRtlSignals to control the number of signals generated
into verilog(see comments for m_generateIndividualRtlSignals).

CSL CODE

```
    // csl example:
     csl_signal_group sg1 {
       csl_signal s1;
       csl_signal s2;
       sg1() {
  generate_individual_rtl_signals(on);}
     };
     csl_signal_group sg2 {
       sg1 sg1instance;
       csl_signal s3;
       sg2() {}
     };
     csl_interface ifc1 {
       csl_port p1(input);
       csl_port p2(output);
       ifc1() {}
     };
     csl_signal_group sg3 {
           sg3() {}
     };
     csl_unit u {
       sg2 sg2instance;
       u() {}
     };
```

VERILOG CODE

```
    module u();
      wire sg2instance_sg1instance_s1;
      wire sg2instance_sg1instance_s2;
      wire sg2instance_s3;
    endmodule
```

4.3.11.3 CSL interface class - declaration, definition, instantiation

In CSL, just like in Verilog, communication with the outside of the unit is achieved using ports. However, CSL differes from Verilog by introducing the interface class which acts as a holder for port objects. The interface class allows for multiple ports to be manipulated at once in connection statements, or other function calls. Port names within an interface are prefixed in the Verilog output with the interface instance name. Note that port objects can be declared inside a unit class definition without having to be part of a user specified interface - in this case the ports are part of the units' default interface and do not receive any prefix. The **csl_interface** declaration is detailed in the commands section: See "csl_interface interface_object_name;" on page 233.
The csl_interface class defintion format is as follows:

4.3.12 The interfaces instance in a signal group

More than one interface can be added to a signal group.All signals in an interface that is added to a signal group have to have the same direction. Signal groups that are connected to a register preserve their signal names and attributes across the register sg-->register->sg .Signal groups that are connected to an interface must have the correct direction ifc_out->sg-->ifc_in.

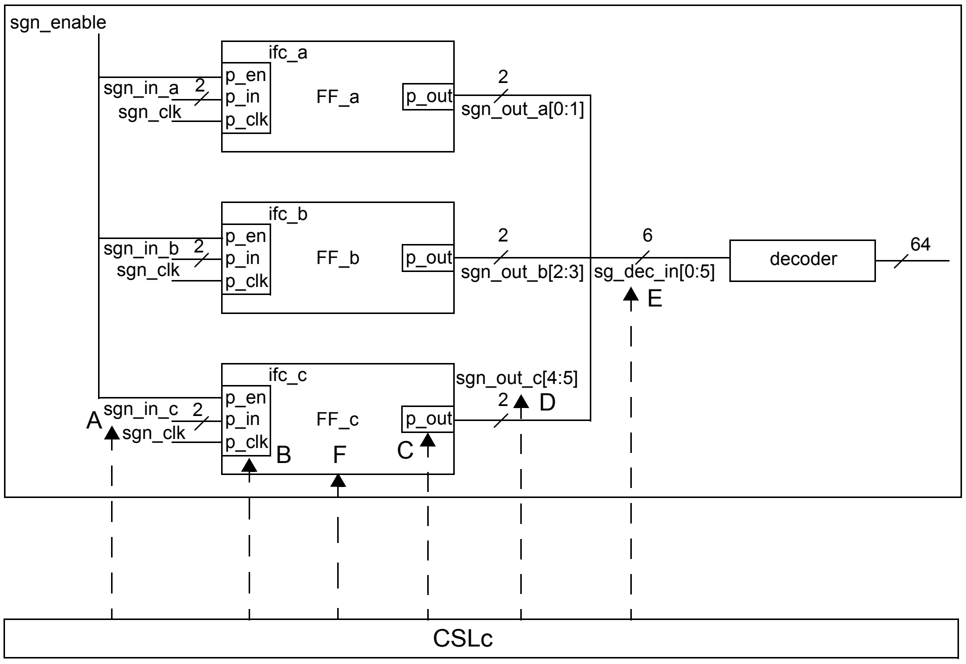
CSL CODE

```
     csl_interface ifc {
     port a (input, 5);
     port b (input, 15);
     ifc() { }
     };
    csl_signal_group sg{
    ifc ifc0;
    sg() {}
```

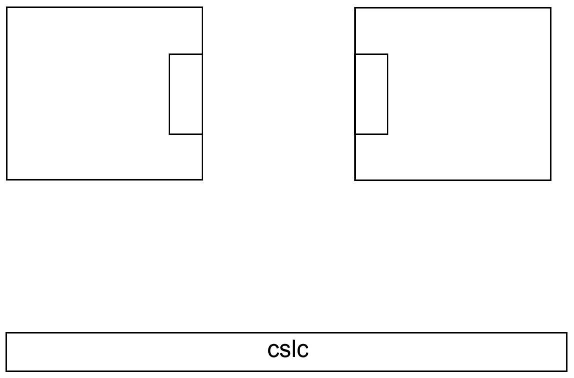
```
};

// to access the interface in the signal group use an hid:

 csl_unit u{
 ifc ifcin;
 sg  sg;
 u() {
 ifcin.connect(sg); // connects the signal group to the interface
 }
 };
```

More than one interface cannot be added to a signal group using inheritance then we will use instantiations.

FIGURE 4.13 Design hierarchy and interconnect



A - signal with 2 bits width
B - interface with three ports
C - output port
D - signal with bitrange associated
E - signal group
F - FF_c unit


cslc