CHAPTER 1 CSL Language common features

TABLE 1-1CHAPTER OUTLINE.

1.1 CSL Language Overview
1.2 CSL Language Concepts
1.3 CSL Language Grammar
1.4 Memory elements
1.5 CSL Field - definition and instantiation
1.6 CSL Language Examples

1.1 CSL Language Overview

1.2 CSL Language Concepts

1.2.1 Scope

A scope defines an area of visibility for variables, methods or objects. In CSL a scope can be a memory map name, a unit, an instance or a namespace.

1.2.2 Multiple dimensions

When dealing with multi dimensional signals (multi dimensional bitranges) it is important to know that dimension indexing starts at 0. Thus, when defining, say the first dimension of a signal, it is referred as dimension 0.

```
scope.object.function(arg_list)
scope.function(arg_list)
if we are already inside the scope then:
function(arg_list)
```

TABLE 1.1 Valid combinations of objects

name-sapce	mem-ory map name	design name	unit name	group name	bitrang e name	regis-ter name	cell name	con-tainer object name	

1.2.2.1 Constants

The constants can be used to specify

- constants with widths
- wires with widths
- parameterized inference hardware (e.g. flip flop FF) using the constants as the module parameters

---

obj constr
stmt constr
obj eval
stmt eval
3. elaboration
-autorroute
-memory map construction
-others
csl analysis-semantic error detection

What is statement construction and how is it different from object construction?
An objects is a unit, bitrange, signal,... and so on. A stmt is a function call, something like signal.set_bitrange(br). The function calls modify the objects.When evaluating an object you eval all expresions linked to it. Object evaluation is separated from all stmt work with objs. If stmts don;t have a fully evaluated obj then bugs can appear.

What is stmt eval an how is that different from object evaluation?

If there is CSL code with a control statement
if (hh) sig.set_width(2) else sig.set_width(4)

The hh is a expresion is evaluated by obj eval. The if control statement
is then evaluated.
Then the if or the else branch of the control statement is evaluated.

constant expression evaluation
bitrange with width 2+4 becomes bitrange with width 6)

1.2.3 Strings

The string construct: **csl_string** *string_object_name* **(** *string_object* **);** generates a string object either by copying an existing one or by passing in an unnamed string objects (in the form **"***string contents***"**).

1.2.3.1 String operations

Strings are used to generate identifiers. A string may be declared using the csl_string command (in this case the string object will have a name) or it can be declared using " " (in this case the string object will be unnamed). String operations allow for changes to be made upon string objects and these are:

TABLE 1.2 String operators

Operator	Action
+	Concatenates two strings. This binary operator will create a string from two strings, or a string and an int, or a string and a char, or two chars.
*	Repeats the string pattern a number of times. n*str and str*n are allowed , n is an integer and m is a string
[ ]	Selects a substring from a string

The substring select operator (the square brackets) can be used in one of the following ways; see Table 1.3

TABLE 1.3 Substring select operator cases

Operator	Action
[n]	Select the n-th character from the string
[n:m]	Select the substring starting from the n-th character to the m-th character. If n and m are positive numbers the first character is 0. A negative index means that the count starts from the end of the string; -1 is the *length-1* character, -2 is *length-2* and so on.
[n:]	Select starts from the n-th character to the end of the string
[:n]	Select from the first character of the string to the n-th character

TABLE 1.3 Substring select operator cases

Operator	Action
[n:+m]	Select from the n-th character, m characters; m>=1
[n-:m]	from the n-m+1 to the n-th character (selects m characters); m>=1

1.2.3.2 C++ string type concatenate operations are supported in the construction of names

```
const string foo=”busx”;
reg ... foo +”_addr”...
reg ... foo +”_data”...
```

1.2.4 Regular Expressions

The CSL language will support the regular expressions in the [**put standard name here**] standard. The regular expressions cause CSL declaration and reference statements to expand into multiple statements. A CSL reference statement is a statement which contains one or more references to CSL symbols which have already been defined. Regular expressions in reference statements are used as a pattern match to find symbol matches in the CSL symbol tables.

A Regular Expression, commonly abreviated as RegEx, is a set of key combinations usually used to match a certain character pattern.The regular expressions supported by CSL can be divided in two main types: generate regex and search regex.

1.2.4.1 Generate Regex

One major problem in HDL languages is automatic language code generation. CSL offers a convenient way to generate variable names with regular expressions.The CSL generating regular expressions are simpler than usual regex because it makes no sense generating from the * for example, because it matches an infinite number of patterns. The following regex constructionsare supported for generation:

TABLE 1.4 CSL Generate Regular Expressions

Regex	Description
[ ]	This is a character class, the engine will generate an identifier from each possible value in the character class
|	Generates the left hand side of the operator and the right hand side of the operator. Note that the pipe operator has the lowest precedence and will only use to generate the closest left-hand and right hand patterns. If you would like to generate from larger patterns, use groupings.

TABLE 1.4 CSL Generate Regular Expressions

Regex	Description
(regex)	Round brackets group the the regex between them. They capture the generated text between them in a back reference and allow you to apply the regex operators to the entire group
(?:regex)	Non-capturing parenthesis group the regex so you can apply the regex operators, but they do not create a back reference.
\1 to \9	This will substitute the text in the generated identifier with the group that was captured in that back reference
(?#comment)	Everything between (?# and) is ignored by the regex generation engine.

The regular expression must be inside a **csl_regex** construct. Example:

```
csl_list names = csl_regex(regex);
```

1.2.4.2 Search Regex

If the user needs more than one regular expression in one command, back references can be used from one expression in another expression. A back reference is when a previous expression is reused without rewriting it, but by simply calling a number corresponding to the expression position. A example would be:

```
a.b.csl_regex(port_([0-9])).connect(a.b.csl_regex(wire_\1));
```

as you notice here we can use the back reference (`\1`) from the port in the wire expression, and the CSLC will generate:

```
a.b.port_0.connect(a.b.wire_0);
a.b.port_1.connect(a.b.wire_1);
a.b.port_2.connect(a.b.wire_2);
a.b.port_3.connect(a.b.wire_3);
a.b.port_4.connect(a.b.wire_4);
a.b.port_5.connect(a.b.wire_5);
a.b.port_6.connect(a.b.wire_6);
a.b.port_7.connect(a.b.wire_7);
a.b.port_8.connect(a.b.wire_8);
a.b.port_9.connect(a.b.wire_9);
```

The CSLC will not issue any errors if for example port_9 doesn't exist. The reason for this is that the regular expression here is a search pattern and it will match all the names in the a.b scope that match the regex and support the connect function. However if port_9 exists and wire_9 doesn't an error will be issued. An error will also be issued when the number of matches in the fist regex is different from the number of matches in the second regex, or the back reference doesn't exist. You can actually do probably the same thing by using re following syntax:

```
a.b.csl_regex(port_(.*)).connect(a.b.csl_regex(wire_\1));
```

Another possibility is:

```
a.b.csl_regex(port_.*).connect(a.b.csl_regex(wire_.*));
```

The CSLC supports this using the following rules:
- the number of matches in each regular expression must be the same
- the pairs of identifiers generated are combined in an alphabetical order (ASCII order actually)
It is also possible to create a concatenation from a regex. If a csl_regex is inside a concatenation the CSLC will automatically generate the concatenation items from the regex. Example:
{a, b, csl_regex(x_[0-9])};

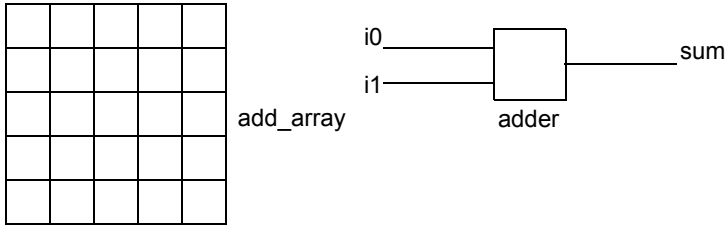
//outside regex back reference
```
foreach i in (a,b,c) {
  csl_regex(x([0-9])).$i.connect(y$1.$i);
}
```

the wires created for this must be named:
x0_y0_a
x0_y0_b
x0_y0_c
x1_y1_a
.....
x9_y9_c
this should be added to autorouter doc

1.2.5 Regular expressions example

Example 1

FIGURE 1.1 5 by 5 adder



A solution for representing in code the figure above is to make use of regular expressions:

```
csl_unit adder, add_array;
scope add_array{
  adder.add_port(input, i0, i1);
  adder.add_port(output, sum);
  add_array.add_instance(adder, adder_[0-4]_[0-4]);
  csl_signal i0_[0-4]_[0-4], i1_[0-4]_[0-4], sum_[0-4]_[0-4];
  adder_[0-4]_[0-4].i0.connect(i0_[0-4]_[0-4]);
  adder_[0-4]_[0-4].i1.connect(i1_[0-4]_[0-4]);
```

```
  adder_[0-4]_[0-4].sum.connect(sum_[0-4]_[0-4]);
  csl_unit adder_tree_5_1;
  adder_tree_5_1.add_port(input, i0_[0-4]);
  adder_tree_5_1.add_port(input, i1_[0-4]);
  adder_tree_5_1.add_port(output, adder_tree_5_1_sum);
  add_array.add_instance(adder_tree_5_1, adder_tree_5_1_[0-4]);
  adder_tree_5_1_[0-4].i0.connect(i0_[0-4]_[0-4]);
  adder_tree_5_1_[0-4].i1.connect(i1_[0-4]_[0-4]);
  adder_tree_5_1_[0-4].adder_tree_5_1_sum.con-
nect(adder_tree_5_1_sum_[0-4]);
}
```

Example 2

```
csl_signal bus_data(32);
csl_signal bus_valid;
csl_signal bus_op(3);
csl_signal bus_stall;
csl_signal bus_pc;

csl_signal_group bus;
//using a string operation
```

bus.add_signal(“bus_”+csl_regex([data, valid, op, stall, pc]));

1.2.6 Events

Events are used to control the behavior of components in a design. Events may be used to trigger other events or to set conditions.
Example of event usage:

CSL CODE

```
csl_event ev;
ev.set_event_expression(we && rf_addr > 0 && rf_addr < 500 );
```

1.3 CSL Language Grammar

1.3.1 CSL Operands

1.3.2 CSL Operators

1.3.2.1 Arithmetic Operators

Arithmetic operators can be *binary* or *unary* (See Table 1.5). Integer division truncates any fractional part. The result of a modulus operation takes the sign of the first operand. If any operand bit value is the unknown value x, then the entire result value is x. Register data types are used as unsigned values (Negative numbers are stored in two's complement form).

TABLE 1.5 Aritmetic Operators

Operator Name	Operator	Type	Description
Unary plus	+	unary	Used to specify the sign (plus)
Unary minus	-	unary	Used to specify the sign (minus)
Binary plus	+	binary	addition
Binary minus	-	binary	substraction
Mutiply	*	binary	multiplication
Divide	/	binary	division
Modulus	%	binary	modulus

EXAMPLE :

```
+5  =  5
-5  =  -5
5  +  10 =  15
5  -  10 =  -5
10 - 5 = 5
10 *  5  =  50
10 /  5  =  2
10 /  -5 = -2
10 %  3  =   1
```

1.3.2.2 Relational Operators

The result is a scalar value (example a < b).

1. 0 if the relation is false (a is bigger then b)

2. 1 if the relation is true (a is smaller then b)

3. x if any of the operands has unknown x bits (if a or b contains X). If any operand is x or z, then the result of that test is treated as false (0)

TABLE 1.6 Relational Operators

Operator Name	Operator	Type	Description
Less than	<	binary	first operand is less than the second operand
Greater than	>	binary	first operand is greater than the second operand
Less then or equal to	<=	binary	first operand is less than or equal to the second operand
Greater than or equal to	>=	binary	first operand is greater than or equal to the second operand

EXAMPLE :

```
5  <  10 = 1
5  >  10 = 0
1'bx  <  10 = x
1'bz  >  10 = x
5  <=  10 = 1
5  >=  10 = 0
10 >= 10 = 1
1'bx  <=  10 = x
1'bz  >=  10 = x
```

1.3.2.3 Equality Operators

There are two types of Equality operators. Case Equality and Logical Equality.
Operands are compared bit by bit, with zero filling if the two operands do not have the same lengt. Result is 0 (false) or 1 (true). For the == and != operators, the result is x, if either operand contains an x or a z. For the === and !== operators, bits with x and z are included in the comparison and must match for the result to be true.

TABLE 1.7 Equality Operators

Operator Name	Operator	Type	Description
Case equal to	===	binary	operand equal to operand2 , including x and z
Case not equal	!==	binary	operand not equal to operand2 , including x and z
Logical equal	==	binary	operand equal to operand2, result can be unknown
Logical not equal	!=	binary	operand1 not equal to operand2, result can be unknown

EXAMPLE :

```
5 == 10 = 0
5 == 5  = 1
5 != 5  = 0
5 != 6  = 1
4'bx001 ===  4'bx001 = 1
4'bx0x1 ===  4'bx001 = 0
4'bz0x1 ===  4'bz0x1 = 1
4'bz0x1 ===  4'bz001 = 0
4'bx0x1 !==  4'bx001 = 1
```

1.3.2.4 Logical Operators

Expressions connected by && and || are evaluated from left to right. Evaluation stops as soon as the result is known. The result is a scalar value: 0 if the relation is false, 1 if the relation is true and x if any of the operands has x (unknown) bits.

TABLE 1.8 Logical Operators

Operator Name	Operator	Type	Description
Logic negation	!	unary	logic negation of the operand
Logic and	&&	binary	logic and between operands
Logic or	||	binary	logic or between operands

EXAMPLE :

```
! 1'b1 = 0
! 1'b0 = 1
1'b1 && 1'b1 = 1
1'b1 && 1'b0 = 0
1'b1 && 1'bx = x
1'b1 || 1'b0 = 1
1'b0 || 1'b0 = 0
1'b0 || 1'bx = x
```

1.3.2.5 Bit-wise Operators

Bitwise operators perform a bit wise operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be extended on the left side with zeroes to match the length of the longer operand.
Computations include unknown bits, in the following way:

1. ~x = x
2. 0&x = 0
3. 1&x = x&x = x
4. 1|x = 1
5. 0|x = x|x = x
6. 0^x = 1^x = x^x = x
7. 0^~x = 1^~x = x^~x = x

When operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

TABLE 1.9 Bit-wise Operators

Operator Name	Operator	Type	Description
Bit-wise negation	~	unary	negation
Bit-wise and	&	binary	and
Bit-wise or	|	binary	inclusive or
Bit-wise exclusiv or	^	binary	exclusive or
Bit-wise exclusive nor	~^	binary	exclusive nor

EXAMPLE :

```
~4'b0001 = 1110
~4'bx001 = x110
~4'bz001 = x110
4'b0001 &  4'b1001 = 0001
4'b1001 &  4'bx001 = x001
4'b0001 |  4'b1001 = 1001
4'b0001 |  4'bx001 = x001
4'b0001 |  4'bz001 = x001
4'b0001 ^  4'b1001 = 1000
4'b0001 ^  4'bx001 = x000
```

```
4'b0001 ^  4'bz001 = z000
4'b0001 ~^ 4'b1001 = 0111
4'b0001 ~^ 4'bx001 = x111
4'b0001 ~^ 4'bz001 = x111
```

1.3.2.6 Reduction Operators

Reduction operators are unary. They perform a bit-wise operation on a single operand to produce a single bit result. Reduction unary NAND and NOR operators operate as AND and OR respectively, but with their outputs negated.

TABLE 1.10 Reduction Operators

Operator Name	Operator	Type	Description
Reduction and	&	unary	bit-wise and on a single operand
Reduction nand	~&	unary	bit-wise nand on a single operand
Reduction or	|	unary	bit-wise or on a single operand
Reduction nor	~|	unary	bit-wise nor on a single operand
Reduction xor	^	unary	bit-wise xor on a single operand
Reduction xnor	~^	unary	bit-wise xnor on a single operand

EXAMPLE :

```
&  4'b1001 = 0
&  4'b1111 = 1
&  4'bx111 = x
&  4'bz111 = x
~& 4'b1001 = 1
&  4'b1111 = 0
~& 4'bx001 = 1
~& 4'bz001 = 1
|  4'b1001 = 1
|  4'b0000 = 0
|  4'bx000 = x
|  4'bz000 = x
~|  4'b1001 = 0
~|  4'b0000 = 1
~|  4'bx000 = x
~|  4'bz000 = x
^  4'b1001 = 0
^  4'bx001 = x
^  4'bz001 = x
```

```
~^ 4'b1001 = 1
~^ 4'bx001 = x
~^ 4'bz001 = x
```

1.3.2.7 Shift Operators

The left operand is shifted by the number of bit positions given by the right operand. The vacated bit positions are filled with zeroes.

TABLE 1.11 Shift Operators

Operator Name	Operator	Type	Description
Left shift	<<, <<<	binary	Shift left the operand
Right shift	>>	binary	Shift right the operand
Right shift arithmetic	>>>	binary	Shift right arithmetic the operand

EXAMPLE :

```
4'b1001 << 1 = 0010
4'b1001 << 3 = 1000
4'b10x1 << 1 = 0x10
4'b10z1 << 1 = 0z10
4'b1001 >> 1 = 0100
4'b1001 >> 3 = 0001
4'b10x1 >> 1 = 010x
4'b10z1 >> 1 = 010z
4'b1001 >>> 1 = 1100
4'b0001 >>> 2 = 0001
4'b10x1 >>> 1 = 110x
4'b10z1 >>> 1 = 110z
```

1.3.2.8 Concatenation Operator

Concatenations are expressed using the brace characters { and }, with commas separating the expressions within. Unsized constant numbers are not allowed in concatenations.

EXAMPLE :

```
{4'b1001,4'b10x1}  = 100110x1
{4'b1001,6'b10z110}  = 100110z110
{4'b1001,5'b10010,2'b10}  = 10011001010
```

Syntax:

```
concatenation := { expression (, expression)* }
```

```
expression := port_hid | signal_hid |
              signal_group_hid | interface_hid |
              sized_constant
```

- There can be any number of expression arguments (at least 1).
- Arguments should be processed and concatenated in the given order.
- If container structures (like signal group or interfaces) are used, these should have all contained elements processed in the order they were declared in the container

EXAMPLE :

```
ifcB contains: port x
ifcA contains: port y, ifcB, port z
```

In a concatenation using ifcA (eg. { ifcA, t, 2'b01 }) the output will be equivalent to:

```
{ ifcA_y, ifcA_ifcB_x, ifcA_z, t, 2'b01 }
```

Where can concatenation be used:

- assignment:

```
x = {a,b,c} or {a,b,c} = x; (LHS and RHS)
```

- connect statement:

```
x.connect({a,b,c});
```

This is NOT legal:

```
{a,b,c}.connect(x);
```

- formal to actual:

```
(.x({a,b,c})); (RHS only)
```

1.3.2.9 Replication Operator

Replication operator is used to replicate a group of bits n times. Say you have a 4 bit variable and you want to replicate it 4 times to get a 16 bit variable: then we can use the replication operator. Repetition multipliers (must be constants) can be used: {3{a}} // this is equivalent to {a, a, a} Nested concatenations and replication operator are possible: {b, {3{c, d}}} // this is equivalent to {b, c, d, c, d, c, d}

EXAMPLE :

```
{4{4'b1001}       = 1001100110011001
{4{4'b1001,1'bz}  = 1001z1001z1001z1001z
```

Syntax:

```
replication := { constant_expression concatenation }
```

The constant_expression is a constant numeric_expression (eg: number, parameter) and has to have a positive, non-zero, non-z, non-x value. Replication is a joining together of as many concatenations as given by the constant_expression value.

Replications can be used in:
- assignment:

```
x = {2{a}};  (only RHS)
```

This is NOT legal

```
{2{a}} = x;
```

- connect statement:

```
x.connect({2{a,b,c}});
```

This is NOT legal:

```
{2{a,b,c}}.connect(x)
```

- formal to actual:

```
(.x({4{a,b,c}})); (RHS only)
```

As a generale rule:
Replications shall NOT be in expressions on the LHS of the assignment nor connected to output or inout ports

1.3.2.10 Conditional Operators

The conditional operator has the following C-like format: cond_expr ? true_expr : false_expr
The true_expr or the false_expr is evaluated and used as a result depending on what cond_expr evaluates to (true or false).

EXAMPLE :

CSL CODE

```
csl_unit conditional_operator;
scope conditional_operator {
    csl_signal out;
    csl_signal(reg, 1, csl_list(enable, data));
// Tri state buffer
    out = (enable) ? data : 1'bz;
}
```

VERILOG CODE

```
module conditional_operator();
wire out;
reg enable,data;
// Tri state buffer
assign out = (enable) ? data : 1'bz;
```

```
endmodule
```

Out value:

```
time enable data out
 0     0      0    z
 1     0      1    z
 2     0      0    z
 3     1      0    0
 4     1      1    1
 5     1      0    0
 6     0      0    z
```

1.3.3 CSL Statements

1.3.3.1 The Conditional Statement if-else

1.3.3.2 The Case Statement

1.3.3.3 Looping Statements

1.3.3.4 Continuous Assignment Statements

1.3.3.5 Prefix

At this point there are 2 CSL classes that supporting prefixing and suffixing in the generated code: interfaces and signal groups. For both types there is also default automatic prefixing activated. The default prefix is the instantiated container object's name prepended to the contained object's name.

Eg: If an interface ifc contains a port x and the interface is instantiated by the name ifc0, the generated port name will be:

```
ifc0x
```

Syntax
For set_prefix() method:

```
[ifc/sg_instance_name.]set_prefix("string");
```

Note that set_prefix() method can be called on both inside the class constructor and on the instance name due to the special nature of the interface and signal group instances.

Once a prefix is set, it will override the default prefix set by the compiler. Thus in the example given above:If an interface ifc contains a port x and the interface is instantiated by the name ifc0 and:
- we call ifc0.set_prefix("ixx_") the generated code will be ixx_x
- we call set_prefix("ixx_") in the interface constructor the generated code will be ixx_x on all instances of that interface unless a set_prefix() method called on an instance does not override this prefix.

No prefix situation:
Sometimes it may be required that there is no prefix in the generated code. There are two ways to accomplish this using the following syntax:
[ifc/sg_instance_name.]set_prefix("");
[ifc/sg_instance_name.]no_prefix();

If a port (x) belongs to an interface (ifc) in a unit the output name for the port will be ifcx
With no_prefix()/set_prefix("") on the ouput name for port x should be x.
Hierarchical interfaces:
Interfaces can be hierarchical. Example: ifc1_.ifc2_.ifc3_.x (ifc1_ifc2_ifc3_x)
If ifc3 has no_prefix()/set_prefix("") on,
than the output for port x should look like: ifc1_ifc2_x
If ifc2 has no_prefix()/set_prefix("") on,
than the output for port x should look like: ifc1_ifc3_x
Checks required:
when using no_prefix()/set_prefix("") on one interface, it's possible to have name conflicts with objects from the parent scope.

EXAMPLE :

```
ifc1_.x
ifc1_.ifc2_.x
```

If ifc2 has no_prefix()/set_prefix("") on, then the output would contain two variables with the same name:

```
ifc1_x
ifc1_x
```

So, when no_prefix()/set_prefix("") is used it should be checked if any object from the current scope has identical name with objects from the parent scope.

1.3.3.6 Suffix

Syntax
For set_suffix() method:

```
[ifc/sg_instance_name.]set_suffix("string");
```

Just like prefixing, objects can be suffixed with a user defined string. Unlike prefixes however there is no default prefixing done by the compiler.

Also, if the suffixing method is called as in:
set_suffix("")
A warning will be printed because the prefix supplied is null and nothing will happen (redundant command).

1.3.3.7 Pattern Generator

The pattern generator is a functionality implemented at preprocessor level and it's used to expand user code into multiple lines of code according to a specification syntax.

EXAMPLE :

```
iob  iob[[0---2]] (.in(x$1));
```

expands into:

```
iob  iob0 (.in(x0));
iob  iob1 (.in(x1));
iob  iob2 (.in(x2));
```

Syntax:

These are the specifiers for the pattern generator:

```
[[ pattern_specifier ]]
```

this generates a set of values according to the type of the pattern specifier. The type of the pattern specifier can be:
- range_pattern: This can be a numeric range (eg. 0---9 generates 0 to 9) or a character range (eg. a---z generates a to z).
- list_pattern: this can be a selection (eg. a,m,z will generate a, m and z)

The pattern specifier creates a backreference that can be called later in the code. Thus, the first pattern specifier creates backreference 1, the second creates backreference 2 and so on.. the backreferences can be later called using a specific syntax.

Note for Syntax specifications below: < > are user variable delimiters and are not part of the pattern generator syntax

backreference_number

Syntax:

```
\$<backreference_number>\
```

Eg. \$1\

this will insert generated code according to pattern specifier linked to backreference 1 as in:

-with range pattern

```
iob  iob[[0---2]] (.in(x\$1\));
```

which expands into:

```
iob  iob0 (.in(x0));
iob  iob1 (.in(x1));
iob  iob2 (.in(x2));
```

-with list_pattern

```
abc_[[a,c,e,g]] = b_\$1\;
```

generates

```
abc_a = b_a;
abc_c = b_c;
abc_e = b_e;
abc_g = b_g;
```

backreference_number with increment amount

Syntax:

```
\i<inc_amount>$<backreference_number>\
```

Eg. \i4$1\
this will insert generated code according to pattern specifier linked to backreference 1 incremented at each iteration with a value of 4

```
abc[[0----3]] = xyz\i4$1\;
```

generates

```
abc0 = xyz0;
abc1 = xyz4;
abc2 = xyz8;
abc3 = xyz12;
```

Note: this does not work for ASCII character ranges in pattern generator

`reversed backreference_number`

Syntax:

```
\r$<back_reference_number>\
```

Eg. \r$1\
counts backwards from the end of the range or list to the beginning of the range or list

-with range pattern

```
abc[[0----3]] = xyz\r$1\;
```

generates

```
abc0 = xyz3;
abc1 = xyz2;
abc2 = xyz1;
abc3 = xyz0;
```

-with list pattern

```
abc_[[a,c,e,g]] = b_\r$1\;
```

generates

```
abc_a = b_g;
abc_c = b_e
abc_e = b_c;
abc_g = b_a;
```

```
(?#ignore_sequence)
```
Everything between the (?# and) is the ignore sequence. The pattern generator will not affect the ignore sequence, leaving it as it is.
Eg:
```
    iob  iob[[(a---d($#_user)) , (e---g($_driver))]][[0---9]](.in(x$1$2));
```

1.3.3.8 Connection with actual name

When using the connect method across unit scopes, the autorouter infers intermediate connectivity elements (if a identical connectivity element cannot be located) with generated names.

The user can 'direct' a connection through selected connectivity elements by using the connect with actual name option. This is similar to connect method only that it specifies an additional connection element (the relaying connectivity element) as in the syntax:

```
    connect_object_hid.connect(connect_object_hid,
    relay_object_expression);
```

the relay_object can be a object of the same type as the objects involved in the connection or an width equivalent object (eg. interface that has the sum of the widths of all ports equal to the width of a port as in portz.connect(ifcx,ifcy)) or can be a part select on a port larger than the connect command scope as in:
```
    portz.connect(ifcx, porty[31:16]);
```

<HALTED: we need to be able to load Verilog to CSLOm for this>
NaN. Do not generate RTL

Syntax
```
    <unit_instance_name>.do_not_gen_rtl();
```

This method says do not generate the RTL code for the unit and the sub hierarchy below the unit.
This unit is used when there is existing Verilog IP code from a customer and they want to hook it up by instantiating the RTL unit but do not want to generate a leaf level unit.

</HALTED: we need to be able to load Verilog to CSLOm for this>

1.3.3.9 Clock domain

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 conatined. A port carries a clock signal if it has the attribute clock set to it by the set_attr(clock) method.

Connectivity elements like:
- ports
- signals
- signal groups
- interfaces

can be associated with a clock name (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: eg. in the register_ios() method, when a flipflop is inferred to register a port it will be driven by the clock associated with that port)

Also, units can be associated with clocks, however 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 it's components are driven by one clock - in this case the set_clock command applies to all connectivity elements from that unit

Synatx:

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

Can be called on:
- ports
- signals
- signal groups
- interfaces
- units

1.3.3.10 Register IO option

This command applies to units and allows for inputs/outputs to be registerd with flip flops:

Syntax:

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

The command can be called on all ports/interfaces of a unit or it can specify a certain interface and it will apply only to that particular interface and contained interfaces (if any).

Only input and output ports will be affected by this command inout type will not be registered.

Note: In order for this to work all connectivity elements involved need to have a clock associated to them with the set_clock() method (because each port in the unit's interface can be clocked by a different clock line) otherwise this will result in an error.

1.4 Memory elements

For some memory elements like register files or fifos, csl offer the possibility to access multiple memory location in same time (multiple writes/reads)

TABLE 1.12 Channels generated for register file

Port Name	Dir	W	Created by	Description
<ifc_name>_rd_en	i	1	csl_rd_interface ifc_name;	Port for ifc_name_rd_en signal
<ifc_name>_rd_addr	i	ud	csl_rd_interface ifc_name;	Port for ifc_name_rd_addr signal
<ifc_name>_rd_data	i	ud	csl_rd_interface ifc_name;	Port for ifc_name_rd_data signal
<ifc_name>_wr_en	i	1	csl_wr_interface ifc_name;	Port for ifc_name_wr_en signal
<ifc_name>_wr_addr	i	ud	csl_wr_interface ifc_name;	Port for ifc_name_wr_addr signal
<ifc_name>_wr_data	o	ud	csl_wr_interface ifc_name;	Port for ifc_name_wr_data signal

NOTE: Dir is the port direction, w is the port width, ud is user defined

1.5 CSL Field - definition and instantiation

1.5.1 Definition:

A csl_field definition looks in the following ways:

1.5.1.1 Hierchical:

```
csl_field field_name {
  (csl_field instantiation)+
  field_name() {
    (hierarchical field command)+
  }
}
```

The above syntax is used only for hierarchical fields. "`(csl_field instantiation)+`" means that we must have at least one instantiation of a field (hierarchical or not). "`(hierarchical field command)+`" will contain hierarchical field positioning commands(see commands for csl_field).

1.5.1.2 non-hierarchical: see CSL Language Commands Summary

1.5.1.3 Allowed command combinations

TABLE 1.13 Allowed command combinations

	set_enum()	set_enum_item()	set_value()	add instance	Note
undefined	Y	Y	Y	Y	any of the above commands will change the type from undef to enum/item/value/hierarchical etc
value	N	N	Y	N	any other command besided the allowed one will generate an error

TABLE 1.13 Allowed command combinations

	set_enum()	set_enum_item()	set_value()	add instance	Note
enum	Y	N	N	N	any other command besided the allowed one will generate an error
enum_item	N	Y	N	N	any other command besided the allowed one will generate an error
hierarchical	N	N	N	Y	any other command besided the allowed one will generate an error
undefined	Y	Y	Y	Y	any of the above commands will change the type from undef to enum/item/value/hier-archical etc

Where: Y = yes (allowed), N = no (not allowed)

See the commands for csl_field

1.5.2 Inheritance

CSL offers inheritance for classes like: units, interfaces, isa instruction formats, etc. (**BZ and DP see which classes do you want to support inheritance**)

1.5.2.1 ISA instruction format

TABLE 1.14 Allowed Inheritance

		instruction
instruction format	Y	N
instruction	Y	N

EXAMPLE :

```
csl_isa_instruction_format format2 : format1 {
  (..) //instantiations (see ISA chapter)
  format2() {
```

```
    (..)  // default constructor methods (see ISA chapter)
  }
};
```

The code above will define a new format (format2) inherited from format1. Basically all the fields and properties from format1 will be copied into format2. Aditional fields can be defined in the new formats if the user wants to.

1.5.3 Instantiation:

1.6 CSL Language Examples

1.6.0.1 Comments

Comments in the specifications are inserted into the generated code.

filename: constant_evaluators.rtf

Constant Classes and evaluation of constant classes
There will be seperate classes for the built in types and the user defined types

Constructing constant expression trees
The ast treewalk will only call Om class constructors and in linking phase there will be real evaluators that will work with om classes and not with ast subtrees

Constant Types
There are several kinds of constants
-built in constant (i.e. integer) with a width defined by the spec
-user constant defined with a bit width defined by the user
-preprocessor define
-parameter
-module parameter

Defines
The software will provide support for both #define and 'define' in input source files.
What is the diff between #define and 'define?
#define is a C/C++ define
`define is a verilog define
RTL'ers will use both types of defines depending on their build environment. The software needs to support both types of defines.

The SW will need several different constant evaluation modes
1. evaluate all #define's and `define`s
2. evaluate all #define's
3. Print out the files with the #define's and `define`s with the constant values in comments next to the defines where they are used

Users need to see what the values of the constants are and what the constant expressions evaluate to. The verilog code generators should in comments should print out the constant expression using the variable names and the numeric names in parens next to the variables.

For example a verilog code gen reads
wire [7:0] FOO = 20;
wire [7:0] FOO_Y = 15;

wire [7:0] x = FOO + FOO_Y;

and generates

wire [7:0] x = 35; // FOO(20) + FOO_Y(15) ?

At the cmd_shell they can type
cmd_shell> show FOO
20

Constant Expression sub-tree evaluation
This class will take an expression and produce a numeric result.
all scalar elements of the expression are either numbers or reduce to numbers

bit class is a class which holds a variable with width.

we need two different constant evaluators
-one for built in (i.e. integer, real)
-one for user defined variables (i.e. 8'h34)

5'10001 + 7'h78 is a bit class expression

Sub-Tree Expression Types
Each omExpr node in the OM expression tree should be marked with an attribute as they are being constructed.

enum{EXPR_TREE_BUILT_IN_CONSTANT, EXPR_TREE_USER_DEFINED_CONSTANT, EXPR_TREE_NON_CONSTANT}.

The software can tag the sub-tree with a TYPE as the tree is parsed. The type will tell the constant evaluation phase whether to evaluate the tree and which evaluator to call.Then the software can call the fast built in evaluator or the user defined variable evaluator. The constant expression evaluators will take the OM sub-tree as an argument.

Width Conversion rules
The specification width coersion rules need to be followed to correctly evaluate expression sub-trees with different width constants.

Command shell constant expression debugging
If the software evaluates the constants in the AST treewalker then the user can't trace the constants and debug problems. We want the user to be able to debug the constant expressions in a command shell. The command shell should support the evaluation of constant expressions. Each OM class will include line and source file so the user can see where problems are occuring. Programmers will use constants in their code that they did not define and they need to find out the widths and values to debug the code

for example omIntVal also holds a string
which is the str expression from the source file

cmd_shell> eval(FOO_ADDR + BASE_ADDR)
returns the value 35

then the user can see what the problem(s) is and they can fix the problem
If the software does the eval in the treewalker then the software can't support command shell evaluation.

CSL C++ language construct
section of
external_customer_code (ecd) external API for hardware
internal_customer_code (icd) internal API software team
project
hardware_team_code
extern

The attributes behave like a combination of namespaces and public, protected, private

Declaring name spaces

```
namespace <name>;
```

a hierarchy of modules may be declared as a namespace
each namespace module is declared with <name>
each <name> is prefixed with <name>
prefxing all ID's with a std prefix

CSL
Const Int
constants are declared with the "const" keyword

```
public:
protected:
private:
```

applied to variables

filename: csl_namespaces.fm

Declaring Namespaces

```
namespace <name>;
```

a hierarchy of modules may be declared as a namespace
each name in each module is declared with <name>
each <name> is prefixed with <name>
prefixing all ID's with a standard prefix

filename: editor_support_for_meta_languages.rtf

Emacs and vi/vim modes need to be developed which support the meta languages.

Emacs tag file generation also needs to be supported.

1.6.1 gen_interconnect macro commands

This section describe CSL macro's which are expanded into CSL code. We will explain how to use the different types of macros. They include the following:

- csl_for
- csl_foreach
- csl_if
- csl_else

1.6.1.1 for macro

```
csl_for (x=0; x<5; x++) {
 string ff = "foo" + x;
 csl_signal ff; // created in the current scope
}
```

This will create:

```
wire foo_0;
wire foo_1;
wire foo_2;
wire foo_3;
wire foo_4;
```

1.6.1.2 foreach macro

```
csl_foreach name (larry curly moe) {
 csl_signal name;
}
```

This will generate the following code:

```
wire larry;
wire curly;
```

```
wire moe;
```

1.6.1.3 csl_if

csl_else

Module Name Prefix

The variable MODULE_NAME_PREFIX is pre-pended to all module names. Module names may be omitted from the list of modules whic will have the prefix prepended by adding the module name to the NO_PREPEND_MODULE_NAME_PREFIX_LIST or by setting the LIBRARY_CELL or MEMORY_ARRAY attribute for the module

Naming conventions hierarchies

```
set_project_name <name>;
set_company_prefix <'prefix'>;
set-chip_prefix = <chip>
set_company_name <chip><unit><module><name>
list chip_prefix;
// show all names with <chip> prefix
set_<var>= string;
// sets the variable <var>=string <var> can be <level>_(name | prefix)
```

level can be :

- company
- project
- version
- chip
- cluster
- unit
- sub_unit

Types

```
csl_list <listname><type>=<list>:;
// support STL vector operations but do not use STL vector name to
avoid confusions
// with CSL test vector
<listname>.add (<element>);
csl_map <map name>=<map>;
<map name>.add (pair <key>, <value>)
```

support scalar types sucj as the following :

- int
- char
- string

Support a typedef operation

Support fixed point and floating point typedefs
Note that each bit position represents a power of two.
The fractional part of the fixed point format is 1/2, 1/4, 1/8, 1/16...

```
csl_fixed_point_typedef <fixed_point_typedef_name>;
<fixed_point_name>.integer (("signed" | "unsigned"));
// default is unsigned note that setting fixed point format to signed
adds one bit to the
// format
<fixed_point_typedef_name>.integer (<numeric_expression>);
// the number of bits in the integer part of the fixed point format
<fixed_point_typedef_name>.integer (<numeric_expression>);
// the number bits in the fractional part of the fixed point format
```

Support conditional instructions

- for
- if
- if else
- while
- case
- cond (one hot or one cold if else construct)
- foreach

1.6.2 Templates

The template contain the formal description of the object.
Templates are instantiated with parameters to define the data type and behaviour of the template.

```
templateNAME <template_item>"<"<list_of_type>">"
```

1.6.3 Range declaration shortcut

To avoid having to use the range syntax ([REGFILE_WIDTH-1:0]) a shortcut is supported which

allows the user to just use the $width operator.

```
reg [$width(signal)]...
which expands into
reg [signal_WIDTH-1:0]...
```

<file cdm_csl_interconnect.fm>

```
//many to 1 connection

//for each module to module connection a unique name is created
signal_declaration : <signal_name>
signal_declaration : {signal_prefix_override}<signal_name> signal_type
signal_range
```

Instead of using a constant in multiple places it is better to use a single bit range declaration

1.6.3.1 Name spaces

Name spaces correspond to functional units (i.e. a module in verilog).
The functional units in C/C++ are functions and in verilog RTL are modules.
We use the term namespace to set the current scope to a declared module name.
The term namespace Is borrowed from C++.
The CSL language has a variable called the **csl_current_name_space**

```
<module name = csl_current_name_space.get();
csl_current_name_space.set(<module name>);
```

If namespace is not defined then the parser will generate an error.
The following construct is used to create a namespace in a CSI design specification which applies to a specfic CSI statement:

```
csl_namespace:: namespace_name  <CSL statement>

csl_namespace:: namespace_name
csl_signal signal_name;
// creates signal_name with one bit In the namespace namespace_name
csl_endnamespace
```

1.6.3.2 For loops for string expansion

For loops can be used to expand strings.
CSL code

```
csl_unit top;
csl_signal top.input(x[7:0]);
csl_for (csl_int i = 0; i <= 7; i++) {
 csl_string vv = "v" + i;
 csl_unit vv;
 top.add_instance (v{vv});
 csl_connect top.x[i] = vv.x;
}
```

Nested loops

```
csl_for (csl_int i = 0; i <= 7; i++) {
 csl_for (csl_int j = 0; j <= 7; j++) {
   csl_string vv = "v" + i + "_" + j;
   csl_unit vv;
   top.add_instance (v{vv});
   csl_connect top.x[i] = vv.x;
  }
}
```

```
csl_for (csl_int i = 0; i <= 7; i++) {
 csl_for (csl_int j = 0; j <= 7; j++) {
   if (j !=4 || j != 6) {
     csl_string vv = "v" + i + "_" + j;
     csl_unit vv;
     top.add_instance (v{vv});
     csl_connect top.x[i] = vv.x;
    }
  }
}
```

```
Alternative for loop notation:
 csl_for (i in [0,7], j in [0,7]) { ... }
```

1.6.3.3 csl_if csl_else

1.6.4 Regular expressions

Regular expressions will be evaluated on the left hand side of the CSL statement first to determine how many statements to create. Then regular expressions on the right hand side will be evaluated.

there are the following cases for regular expression evaluation:

1. one regex on the LHS and no regexs on the RHS
2. no regexs on the LHS and one regex on the RHS
3. one regex on the LHS and one regex on the RHS
4. one regex on the LHS and two or more regexs on the RHS
5. two or more regexs on the LHS and one regex on the RHS
6. two or more regexs on the LHS and two or mor regexs on the RHS

We define the evaluation order to be the folllowing

```
csl_unit top, p;
top.add_instance (p);
csl_signal top.input( x"[0-4]" );  // creates x1, x2, x3, x4
csl_connect top.{x"[0-4]"} = p.z[4:0]; // creates a 5-bit signal
called z in unit p and connects it to {x0,x1,x2,x3,x4}

We either need two tokens of lookahead or syntactic predicates.

If the regular expression is inside of a concatenate then create a
list.
csl_connect top.{x"[0-4]"} = p.z[4:0];
creates a 5-bit signal called z in unit p and connects it to
{x0,x1,x2,x3,x4}

If the regular expression is inside of a assignment  then create n
assignment statements.
csl_connect top.x[4:0] = x"[0-4]";
```

```
expands into

assign x[0] = x4;
assign x[1] = x3;
assign x[2] = x2;
assign x[3] = x1;
assign x[4] = x0;


If the regular expression is inside of a assignment  then create n
assignment statements.
csl_connect top.x[4:0] = x"[4-0]";

expands into

assign x[4] = x4;
assign x[3] = x3;
assign x[2] = x2;
assign x[1] = x1;
assign x[0] = x0;



Multiple concatentations are allowed inside of a concatenation.

{x"[0-1]", y"[0-1]"}

expands to

{ x0, x1, y0, y1}
```

Width Mismatch example

```
sl_connect top.x[7:0] = x"[0-4]";
```

expands to

```
assign x[7] = x0;
assign x[6] = x1;
```

```
assign x[5] = x2;
assign x[4] = x3;
assign x[3] = x4;
```

The compiler should generate a warning that the indices and the sub-scripts are reversed.

Map columns into rows

We can create a group of 8 signals which are 14-bits each

```
csl_signal col0_"[0-7]"[13:0];
```

We can create a 4 groups of 4 signals (16 signal total) which are 14-bits each using a regular expression in a for loop.

```
csl_for(i in [0,3]){
  csl_string col =  "col_" +  i  +  "_" + "[0-3]";
  csl_signal col[13:0];
}
```

```
wire col_0_0[13:0]
wire col_0_1[13:0]
wire col_0_2[13:0]
wire col_0_3[13:0]

wire col_1_0[13:0]
wire col_1_1[13:0]
wire col_1_2[13:0]
wire col_1_3[13:0]

wire col_2_0[13:0]
wire col_2_1[13:0]
wire col_2_2[13:0]
wire col_3_3[13:0]

wire col_3_0[13:0]
wire col_3_1[13:0]
wire col_3_2[13:0]
wire col_3_3[13:0]
```

Now create 4 64-bit words

```
csl_for(i in [0,3]){
  csl_string col =  "row_" +  i  +  "_" + "[0-3]";
  csl_signal col[13:0];
}

csl_signal row"[0-3]"[63:0] = {col_"[$1]"_"[0-3]"};

Where $1 expands into the first regular expression and entire statement
expands into

wire row0_0_3[63:0] = {col_0_0, col_1_0, col_2_0, col_3_0};
wire row1_0_3[63:0] = {col_0_1, col_1_1, col_2_1, col_3_1};
wire row2_0_3[63:0] = {col_0_2, col_1_2, col_2_2, col_3_2};
wire row3_0_3[63:0] = {col_0_3, col_1_3, col_2_3, col_3_3};
```

1.6.4.1 Need to embed CSL into Verilog or VHDL

Do we have two parsers? Or do we mix the languagse into one parser.

1.6.5 Module Name Prefix

The variable MODULE_NAME_PREFIX is pre-period to all module names. Module names may be omitted from the list of modules which will have the prefix prepended by adding the module name to the NO_PREPEND_MODULE_NAME_PREFIX_LIST or by setting the LIBRARY_CELL or MEMORY_ARRAY attribute for the module

1.6.6 Naming convention hierarchies

```
set_project_name <name>;
set_company_prefix <prefix>;
set_chip_prefix <chip>;
set_company_name <unit><module><name>
list chip_prefix; // show all names with <chip> prefix
```

```
set_<var>=string; //sets the variable <var>=string
//<var>can be <level>_(name|prefix)
```

level can be :

- company
- project
- version
- chip
- cluster
- unit
- sub_unit

NAMESPACES HAVE BEEN REMOVED
Name spaces can be used to divide the memory element space into different scopes. The scopes are used to access memory elements in a local region. Name spaces can have base and limit addresses

The address space can be given a name . The address space name can be used as a prefix to constant names. Namespaces are hierarchical.

```
namespace top {
  prefix_names_with hierarchical path = true;
  // default = false;
  namespace middle {
    prefix_names_with namespace_name = true;
    // default = false;
  }
}
```

<BEGIN OF MOVED HERE>
<move to csl_language>

1.6.6.1 String concatenate operations are supported in the construction of names

```
const string foo = "busx";
const string bar = "unita";
const string reg_name = foo+bar;
reg reg_name;
```

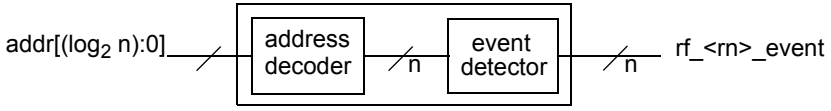
</move>
<moved from csl register file>

1.6.7 Register Files event detectors

- n is the width of the bus
- m is the number of unused addresses in the addresses spaces

Note: the verilog implementation of the register file address decoder is 1 << addr

FIGURE 1.2 Register File with an event detector



1.6.7.1

The width of the cells cannot exceed the width of register. If the cells are defined by the **rws** (*read*, *write* and *shadow*) bits then the cell uses the register's rws bits.

</moved>
</END OF MOVED HERE>