CSYN Verilog Compiler

Professional Reference Manual

D.J. Greaves
(djg@cl.cam.ac.uk)

Release CV2.18 Summer 97

http://www.cl.cam.ac.uk/users/djg
University of Cambridge
Computer Laboratory
New Museums Site
Pembroke Street
Cambridge
CB2 3QG

About This Manual

This manual describes the CSYN-V2 Verilog Compiler and mentions the ancillary programs CVNL, CVABEL and XNFTOV. The manual also explains how to use CSYN with FPGA tools (such as those for Xilinx).
CSYN-V2 is a program which will compile a circuit description in the Verilog language into a gate-level, hierarchic, netlist. This netlist may then be fed into other tools for simulation or layout and fabrication.

Disclaimer and Copyright

This software and this manual are provided for use at your own risk. DJ Greaves accepts no responsibility for any copyright or patent infringement or any other direct or consequential loss which may arise from using this software.

Contents

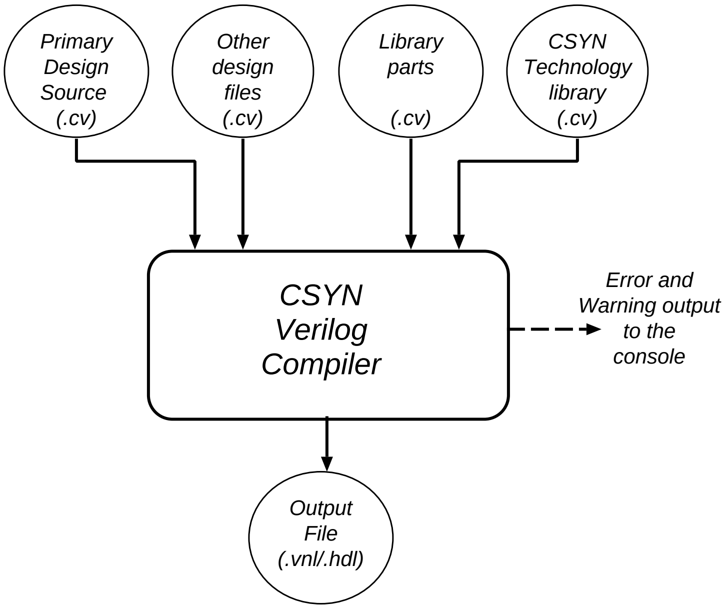

Figure 1: CSYN Design Flow

1 About the CSYN Verilog Compiler

Figure 1 shows that the CSYN program accepts a number (not necessarily four) of source files and produces a single object file, as well as console messages in the case of any warnings or errors.

All of the source files are in the Verilog language and normally have suffix '.cv'. One of the files contains a root module and the other files contain modules needed to synthesise the root module. These other files may be in the current directory or in libraries.

The output file is a net list in Verilog (suffix '.vnl') or Cambridge HDL ('.hdl').

Although not necessary, the tool should typically be used alongside a Verilog simulator which can simulate either the input or the output from CV2. Suitable simulators are CSIM (available within the department only) and Cadence Verilog-XL (for which we have on license).

2 CSYN Verilog Language Subset and Syntax

The CSYN-V2 compiler accepts a subset of the full Verilog language for compilation into a gate-level netlist. This section describes that subset. The subset is the main structural and RTL (register transfer level) set of the language, but excludes Verilog's built-in transistors and gates, and signal strengths.

CV2.18 is a basic RTL compiler and will not handle the following behavioural constructs: `while`, `for`, `fork-join` and more than one event control statement in the body of an `always` statement. The compiler will also not support Verilog's functions, and user-defined primitives. See section 11 for further limitations. CV2.19 has full behavioural capabilities.

The full Verilog language is described in the OVI (Open Verilog International) 'Language Reference Manual' available for ftp from various places,
including `ftp.chronologic.com/pub/ovi`. This section describes the language subset supported by the CSYN synthesiser. Other useful reference books include

> 'The Verilog Hardware Description Language' by Donald E. Thomas and Philip Moorby. Published by Kluwer Academic Publishers. ISBN 0–7923-9126-8.
>
> 'Digital Design and Synthesis with Verilog HDL' by Eli Sternheim, Rajvir Singh, Rajeev Madhaven and Yatin Trivedi. Published by Automata. ISBN 0-9627488-2-X. This book

comes with a demonstration disk of a Verilog simulator.

Verilog is quite a rich language and supports various levels of hardware specification of increasing abstraction using the various language constructs available. These levels can be freely mixed in one circuit. A taxonomy of four levels is handy for the current purposes, as follows:

1. Structural Verilog: a hierarchic netlist form.
2. Continuous Assignment: allows advanced combinatorial expressions, including adders and bus comparison, etc.
3. Synthesisable Behavioural Subset: allows clocked assignment, `if-then-else` and `case` statements, resulting in state machine synthesis.
4. The Unsynthesisable Behavioural Remainder: includes programming constructs such as unbounded `while` loops and `fork-join`.

The CSYN compiler (cv2) supports levels 1 through 3. *The next major version of CSYN (cv2.19/CV2) supports additional behavioural constructs, including event control within loops.*

2.1 Verilog Lexicography

A Verilog source file contains modules, comments and macro definitions. All whitespace characters (outside of macro definition lines) are optional and ignored, except where adjacent identifiers would elide.

2.1.1 Comments

One line comments are introduced with the revered double slash syntax and terminate at the end of line. Block comments are introduced with slash-star and terminate with star-slash. Block comments may not be nested.

```
/*
 * This is a block comment.
 */

// And this is a single line comment.
```

2.1.2 Identifiers

Identifier names consist of alphanumeric strings which can also contain underscores and must start with a letter. Case is significant. In this manual, uppercase identifiers are used by convention for module names, but this is not required by the language definition or CSYN.

CSYN maintains separate identifier spaces for modules, nets, macros and instances, allowing the same identifier to be used in more than one context. The net identifier space also includes other classes of identifier that could be used in expressions, such as parameters and integers.

Many postprocessing tools do not maintain such separate namespaces, so it is possible to have clashes within these tools that were not a problem for CSYN.

2.2 Module, Port and Net Definitions

A module is the top-level syntactic item in a Verilog source file. Each module starts with the word `module` and ends with `endmodule` and between the two are 'module declarative items'.

```
// Here is a module definition, called FRED
module FRED(q, a, b);

   input a, b;    // These are the inputs
   output q;      // Make sure your comments are helpful

  // ..... Guts of module go here .....

 endmodule
```

After the module name comes a list of formal parameters in parenthesis. These are also known as the 'ports'. In the module body, semicolons are used to terminate atomic statements, but not composite statements. Hence there is never one after an `endmodule` (or an `end` or an `endcase`).

Valid module declarative items include the following statements `parameter`, `input`, `output`, `inout`, `wire`, `reg`, `tri`, `pullup`, `pulldown`, `integer`, `event`, `time`, `assign`, `initial`, `always` and gate and module structural instantiations.

The order of the declarative items is unimportant semantically, except that nets need to be declared before they are referenced.

2.2.1 Input and Output definitions

Each of the ports of a module must be explained using the keywords

- `input` : signal coming into the module
- `output`: signal going out of the module
- `inout` : bidirectional signal.

These declarations take a comma separated list of ports which share a similar direction. When a modules are instantiated, CSYN checks that no two outputs are connected to each other and that all nets of type `wire` are are driven by exactly one output.

2.2.2 Pullups and Pulldowns

A net may be defined to float to logic zero or one by the inclusion of a pulldown or pullup, respectively. These statements should be used for inputs which are liable to be disconnected or to tristate bus lines.

```
wire a;
tri b;
pullup (a);
pulldown (b);
```

For synthesis, CSYN turns these Verilog language components into straightforward structural instances of a PULLUP or PULLDOWN. However, ignore the above, since with CV2, it is better to instantiate components called PULLUP and PULLDOWN instead of using the builtin pullup and pulldown declarators.

2.2.3 Bus definitions

A Verilog signal may either be a simple net or else a bus. When an identifier is introduced (using a declaration such as `input` or `tri` etc.), if it is given a range, then it is a bus, otherwise it is a simple net. When an identifier which denotes a bus is referred to without giving an index range, then the full range of the bus is implied. Most Verilog operators will operate on both busses and simple nets.

Busses are defined by putting a range in square brackets after the keyword which declares then. For example

```
// Here is a module definition with two input busses
module FRED(q, d, e);

 input [4:0] d, e;    // Ports d and e are each five bit busses
 output q;            // q is still one bit wide

endmodule
```

Note that when a module port is actually a bus, the bus range specification is not given in the formal parameter list, just its identifier. However, some dialects of Verilog allow the range specifications in the formals. In Verilog, width specifications may be from high to low or low to high. With CSYN however, the most significant bit (highest index) must be specified first and the least significant bit index second. Users who define their busses the other way around will later find that certain built-in operators (such as addition) may not work as expected. The least significant index does not have to be zero, which is helpful sometimes. Before logical operations on busses, such as addition or comparison, the numeric values of the bus indecies are normalised by effectively subtracting the least significant index from all bus wires.

2.2.4 Local signal names

Local signal names are introduced with the `wire`, `reg` and `tri` statements. There are slight variations in the way these types of nets can be used, as explained elsewhere in this manual, but the syntax of the three statements is the same. Here is an example using some wires:

```
// Here are some local nets being defined
module FRED(q, a);

   input a;
   output q;

   wire [2:0] xbus, ybus;    // Two local busses of three bits each
   wire parity_x, parity_y;  // Two simple local signals.

 endmodule
```

Verilog allows the words `vectored` *and* `scalared` *to be put before range definitions, but they are ignored by the synthesiser.*

2.3 Structural Specification

One further statement type, beyond the wire and port declarations, is all we need for structural specification of a hierarchical or flat netlist. This is the component instance statment. The syntax of a submodule or component instance is:

```
<modname> <instancename>(<portlist>)  [ , <instancename>(<portlist>) ] ;
```

where the square brackets are meta-syntax to indicate optional repetition and the angle brackets indicate that an identifier is not a keyword. The syntax allows several instances of the same submodule to be specified in one instance statement, if desired.

```
// Here is a module definition with submodule instances.
module FRED(q, a, b);
```

```
  input a, b;    // These are the inputs
  output q;      // Make sure your comments are helpful
  wire qbar;     // Local, unported net

  NOR2 mygate(q, a, qbar), myothergate(qbar, b, q);

endmodule
```

The submodule NOR2 must be defined somewhere, as a leaf module in a technology library, or in the same or another design-specific Verilog source file. Technology libraries are described in section 5.

Instance names, such as 'mygate' in the example, are optional. CSYN will make up instance names for instances in its output netlists if they are not provided in the source Verilog. However, this is not recommended since these names are likely to change each time a circuit is compiled and this is often inconvenient if a simulator or placement tool has retained instance name state from one run to another.

Note that explicit instantiation of this type at the low level of individual gates is not normally done by a human, unless this is an especially critical section of logic. Such gate-level Verilog is normally synthesised from the higher levels of Verilog source by CSYN or other tools.

2.3.1 Positional and Named port mapping

The component instance example above, for NOR2, uses positional port mapping. The writer of the Verilog had to know that in the definition of NOR2, the output net was put first. That is

```
// Here is the signature of NOR2
module NOR2(y, x1, x2);

  input x1, x2;    // These are the inputs
  output y;        // Make sure your comments are always helpful

endmodule
```

However, Verilog also allows named port mapping in the instance statement. Using named port mapping, the writer does not have to remember the order of the ports in the module definition, he must remember their names instead. The chance of misconnection though giving the wrong order is eliminated and the compiler is able to generate more helpful error messages when the wrong number of arguments is supplied to an instance.

```
// Here is a module definition with named port mapping.
module FRED(q, a, b);

  input a, b;    // These are the inputs
  output q;      // Make sure your comments are helpful
  wire qbar;     // Local, unported net

  NOR2 mygate(.x1(a), .y(q), .x2(qbar)),
       myothergate(.y(qbar), .x2(q), .x1(b));

endmodule
```

As can be seen, the syntax of a named port association is

```
. <formal-name> ( <signal-expression> )
```

whereas for the positional syntax, we just have a list of signal expressions.

I use the term 'signal expression' since the actual parameters, for both syntaxes, do not have to be simply signal names, but for inputs to the submodule, can be any expression which results in a signal of the correct width. Signal expressions are explained shortly in section 2.4. [It's the next section!]

It is often necessary to pass a single net of a bus, or a subgroup of a bus to or from a module instance. This can be done using the bus subranging operators described in section 2.4.2.

2.4 Continuous Assignment and Signal Expressions

The level of Verilog specification which comes above netlists and structural instantiation is continuous assignment. The syntax of a continuous assignment within CSYN is

```
assign <signal> = <signal-expression> ;
```

For example

```
wire p;
assign p = (q & r) | (~r & ~s);
```

The stile denotes Boolean OR, the ampersand denotes Boolean AND and the tilde denotes inversion. The signals `q`, `r` and `s` must be defined already using one of the declarations `input`, `output`, `inout`, `wire`, `tri` or `reg`.

A continuous assignment is synthesised into a set of gates which achieve the desired logic function. Partial logic minimisation[1] is applied to the signal expression on the right-hand side, but CSYN does not eliminate explicitly declared nets or their interconnection pattern across continuous assignments.

In Verilog, the 'hash' form of event control may be used as part of an assignment statement to introduce a delay. For example

```
assign #10 p = q;
```

CSYN ignores delay specifications when they are present in assignment statements, but flags a warning at the end of compilation if the delays were in non-primitive modules.

Verilog allows a shorthand that combines a `wire` declaration with an assignment. The above example can be written more concisely

```
wire p = (q & r) | (~r & ~s);
```

where the assignment to p is combined into the `wire` statement that introduced it. Multiple wires can be defined and initialised at once if they all have the same range. For example

```
wire p = ~q, r = ~s;
wire [3:0] bill = 15-ben, foo = 15-bar;
```

2.4.1 Constant numeric expressions

Numerical constants can be introduced in a number of bases using the syntax:

```
<width-expr> ' <base> <value-expr>
```

where the base is a single letter:

b	:	binary
o	:	octal
h	:	hexadecimal
d	:	decimal (the default)

For example

[1]Full logic minimisation is NP complete.

```
wire [7:0] bus1, clipbus;
assign clipbus = (bus1 > 8'd127) ? 8'd127 : bus1;
```

where 8'd127 is the decimal number 127 expressed to a width of eight bits.

If constants are given without a width, CSYN will attempt to provide an appropriate width to use. If the situation is ambiguous, an error is flagged. CSYN will also accept simple strings of digits and treat them as unsized decimal numbers.

Underscores are allowed in numbers. These are ignored by the compiler but can make the language more readable. For instance

```
wire [23:0] mybus = yourbus & 24'b11110000_11110000_00001111;
```

2.4.2 Bus subranging and concatenation

To have access to parts of a bus, indexing can be used with a bit field in square brackets. When the index is left out, which is the case in the examples until now, the default width of the bus is used.

examples

```
wire [31:0] dbus;
wire bit7 = dbus[7];
wire [7:0] lowsevenbits = dbus[7:0];
wire [31:0] swapbus = dbus[0:31];
```

It is possible to assign to part of a bus using a set of assignment statements provided that each net of the bus is assigned to exactly once (i.e. the ranges are disjoint). For example

```
wire [11:0] boo;
input [3:0] src;
assign boo[11:8] = src;
assign boo[7:4] = 4'b0;
assign boo[3:0] = src + 4'd1;
```

It is also possible to form anonymous busses using a concatenation of signals. Concatenations are introduced with curly braces and have width equal to the sum of their components. Hence, the previous example can be rewritten more briefly

```
assign boo = { src, 4'b0, src + 4'd1 };
```

Note that all items within a concatenation must have a clearly specified width. In particular, unsized numbers may not be used.

Items in a concatenation may be prefixed with a repeat count if desired. The following two example lines are therefore identical in meaning.

```
{ bill, bill, ben, ben, ben, weed }
{ 2 { bill }, 3 { ben }, weed }
```

Note that CSYN does not support assignment to concatenations, although many Verilog dialects do. Instead, multiple separate assignments are needed.

2.4.3 Verilog Operators

Table 1 defines the set of combinatorial operators available for use in signal expressions in order of binding power. Parenthesis can be used to overcome binding power in the normal sort of way.

Symbol	Function	Resultant width
~	monadic negate	as input width
−	monadic complement (*)	as input width
!	monadic logic not	unit
*	unsigned binary multiply	sum of arg widths minus one (?)
/	unsigned binary division (*)	difference of arg widths
%	unsigned binary modulus (*)	width of rhs arg
+	unsigned binary addition	maximum of input widths
−	unsigned binary subtraction	maximum of input widths
>>	right shift operator	as left argument
<<	left shift operator	as left argument
==	net/bus comparison	unit
!=	inverted net/bus compare operator	unit
<	bus compare operator	unit
>	bus compare operator	unit
>=	bus compare operator	unit
<=	bus compare operator	unit
&	diadic bitwise and	maximum of both inputs
^	diadic bitwise xor	maximum of both inputs
^~	diadic bitwise xnor (*)	maximum of both inputs
|	diadic bitwise or	maximum of both inputs
&&	diadic logical and	unit
||	diadic logical or	unit
? :	conditional expression	maximum of data inputs

Table 1: Verilog Operators in order of Binding Power. Asterisked operators are not supported in current release.

All operators can be applied to simple nets or to busses. When a diadic operator (i.e., one that takes two arguments; this is often called *binary*) is applied where one argument is a simple net and the other a bus, CSYN treats the simple net as the least significant bit of a bus with all other bits zero. This is also the case when busses of unequal size are combined. Verilog has no support for two's complement or sign extension built in—it treats all numbers as unsigned. When signed operation is needed, the user must create such behaviour around the built-in operators. (Of course, addition and subtraction work both with unsigned and two's complement numbers anyway).

There are both Boolean and logical versions of the AND and OR operators. These have different effects when applied to busses and also different precedence binding power. The Boolean operators '&' and or '|' produce bitwise ANDs and ORs of their two arguments. The logical operators '&&' and '||' first or-reduce (see section 2.4.4) their arguments and then combine them in a unit width AND or OR gate.

2.4.4 Unary Reduction

It is possible to reduce a bus to a wire under an operator: that is, to combine all of the wires of a bus using an associative Boolean diadic operator. This is called in Verilog *unary reduction* (in contrast to the mathematical use of a unary operator as a function that takes only one argument). The syntax of unary reduction required by CSYN insists on two sets of parenthesis.

```
( <unary-op> ( <signal-expr> ))
```

the unary-operator must be one of the ones in table 2. For example

```
wire [9:0] mybus;
wire ex = (& (mybus));
```

constructs a ten-input AND gate such that `ex` will be a logical one if the bus contains all ones.

<table>
<tr><th>Symbol</th><th>Function</th></tr>
<tr><td>&</td><td>and</td></tr>
<tr><td>|</td><td>or</td></tr>
<tr><td>^</td><td>xor</td></tr>
<tr><td>~&</td><td>and with final invert</td></tr>
<tr><td>~|</td><td>or with final invert</td></tr>
<tr><td>~^</td><td>xor with final invert</td></tr>
</table>

Table 2: Verilog Unary Reduction Operators

2.4.5 Logical Not and Bitwise Not

The logical not operator is '!' and has a different meaning from the bitwise not operator '~' when applied to a bus. The logical not operator will or-reduce the wires of the bus and then negate the result, giving a unit width output. The bitwise not operator simply gives a bus the same width as its input but with each wire negated.

2.4.6 Carry and borrow access

In order to access the carry (borrow) bit which results when two busses of identical size are added (subtracted), it is recommended to increase the size of one (or both) input busses first, as the follow example shows:

```
wire [3:0] x0, x1;
wire [4:0] sum_with_carry = x0 + { 1'd0, x1 };
```

This is because, in Verilog, the result width of an addition or subtraction is defined to be only the width of input arguments. CSYN takes the maximum of the input arguments, so that only one of the two busses `x0` and `x1` in the example needed extension.

In fact, CSYN is quite relaxed about carry access and the following will also work:

```
wire [3:0] x0, x1;
wire [4:0] sum_with_carry = x0 +  x1;
```

The following will not give access to the carry, and instead bit 4 of `sum_without_carry` will be always zeros and a width mismatch warning message will be given, since inside concatenations it is necessary to be strict about widths:

```
wire [3:0] x0, x1;
wire [4:0] sum_without_carry = { x0 + x1 };
```

2.4.7 Conditional Expressions

The conditional expression operator allows multiplexers to be made. It has the syntax

```
(<switch-expr>) ? <true-expr> : <false-expr>
```

The value of the expression is the false-expression when the switch-expression is zero and the true-expression otherwise. If the switch-expression is a bus, then it is unary-reduced under the OR operator. CSYN does not insist on the parenthesis around the switch-expression, but they are recommended. The association of the conditional expression operator is defined to enable multiple cases to be tested without multiple parenthesis. Examples

```
wire [7:0] p, q, r;
wire s0, s1;
```

```
wire [7:0] bus1 = (s0) ? p : q;
wire [7:0] bus2 = (s0) ? p : (s1) ? q : r;
```

The bus comparison predicates (== != < > <= >=) take a pair of busses and return a unity width result signal. For example

```
wire yes = p > r;
```

2.4.8 Dynamic Subscription

CSYN supports dynamic subscription (indexing) of a bus in a signal expression provided that the selected width from the bus is one bit wide. Dynamic subscription is not supported on the left hand side of any assignment. Here is an example of use

```
module TEST();
  wire [10:0] bus;
  wire [3:0] idx;
  ...
  wire bit = bus[idx];
  ...
endmodule
```

2.5 Behavioural Specification

Verilog HDL contains a full set of behavioural programming constructs, allowing one to write procedural style programs, with ordered assignments (as opposed to continuous assignments). The CSYN-V2 Verilog compiler supports a subset of the language's behavioural constructs for logic synthesis. These are described in this section.

Behavioural statements must be included within an `initial` declaration or an `always` declaration. CSYN (release cv2.18) ignores the contents of `initial` statements which simply assign to a variable: these may be present and are useful when simulating the source file with a Verilog simulator (e.g. Verilog-XL or CSIM). Statements of the form `initial forever` or `initial while (1)` are synonymous with `always` statements. Other forms of `initial` statements are flagged as unsynthesisable.

CSYN supports only the following form of the Verilog `always` construct. It has the syntax

```
always @( <sensitivity-list> ) <behavioural-statement>
```

This statement causes execution of the behavioural statement each time an event in the sensitivity list occurs. The sensitivity list is described in section 2.5.6, but we will first present the behavioural statements that are supported.

The sequences '`initial forever`' and '`initial while (1)`' are synonymous with '`always`'.

CSYN-V2 will synthesise the following behavioural statements into hardware: `begin-end`, `if-then-else`, `case repeat`, task invocation and both the blocking and non-blocking behavioural assignment.

2.5.1 `begin-end` Behavioural Statement

This statement is simply the sequencing construct required to extend the range of behavioural statements which include behavioural statements (such as `if` and `always`) to cover multiple behavioural statements. The syntax is

```
begin  <behav-statement> [ ; <behav-statement> ] end
```

where multiple instances of the contents of the square brackets may be present. The order of statements inside a `begin-end` block is important when multiple blocking assignments are made to the same variable. Multiple non-blocking assignments to the same variable are not allowed unless they are in different branches of an `if-then-else`.

2.5.2 `repeat` Behavioural Statement

The `repeat` statement is supported by CSYN provided it has a compile-time constant argument and is simply textually expanded into that many copies of its argument statement (which of course can be a block). The syntax is

```
repeat (<simple-numerical-expression>) <behav-statement>
```

2.5.3 `if-then-else` Behavioural Statement

The `if` statement enables selective execution of behavioural statements. The target behavioural statement (which can be any type of behavioural statement, including further `if` statements) is executed if the signal expression is non-zero.

```
if ( <signal-expr> ) <behav-statement> [ else <behav-statement>  ]
```

If the signal expression given for the condition is a bus it is or-reduced so that the target behavioural statement is executed if any of the wires in the bus are non-zero. When the optional `else` clause is present, the else clause is executed if the signal-expression condition is false.

2.5.4 `case` Behavioural Statement

The `case` statement enables one of a number of behavioural statements to be executed depending on the value of an expression. The syntax of the statement is

```
case ( <top-signal-expr> ) <case-items> endcase
```

where the case-items are a list of items of the form

```
<tag-signal-expr> [ , <tag-signal-expr> ] : <behav-statement>
```

and the list may include one default item of the form

```
default : <behav-statement>
```

The semantic is that the top signal expression is compared against each of the tag signal expressions and the behavioural statement of the first matching tag is executed. If no tags match, the statement provided as the default is executed, if present.

If the `parallel_case` flag is present in a comment immediately after the right hand parenthesis of the case top signal expression, then the semantic of the case statement is changed and all matching expressions are evaluated. This normally produces fewer gates than the sequential case semantic and is often helpful for one-hot state machines where the designer knows that only one state will actually match.

It may be observed that there are a number of variations from the case statements found in other languages, such as C. These are: the tag values do not have to be constants, but can be any signal expression; for each group of tag expressions there must be exactly one target statement (`begin-end` must be used to overcome this) and there is no 'fall-through' from one case section to the next (i.e. the 'break' of C is implied.)

There is an example traffic lights controller in section 9 which uses the case statement.

The variant case statments `casex` and `casez` are supported and have the same syntax as the normal case statement.

2.5.5 Behavioural Assignment Statements

Behavioural assignment is used to change the values of nets which have type 'reg'. These retain their value until the next assignment to them. CSYN supports two types of behavioural assignment (as well as the continuous assignment of section 2.4).

Non-blocking or 'delayed' assignment.
Using the `<=` operator, an assignment is 'non-blocking' (or is 'delayed') in the sense that the value of the assigned variable does not change immediately and subsequent references to it in the same group of statements (`always` block) will refer to its old value. This corresponds to the normal behaviour of synchronous logic, where each registered net corresponds to a D-type (or broadside register for a bus) clocked when events in the sensitivity list dictate. The syntax is

```
<regnet>  <=  <signal-expression> ;
```

The left hand side signal must have type reg. For example, to swap two registers we may use

```
x <= y;
y <= x;
```

Where a variable is updated by more than one delayed assignment per clock cycle, each new one simply cancels any previous one.

Blocking or 'immediate' assignment.
When assignment is made using the `=` operator, the assignment is 'blocking' and occurs immediately meaning that the values found by the right-hand-sides of subsequent behavioural statements in the block will use the newly assigned value.[2] The syntax is

```
<regnet>  =  <signal-expression> ;
```

Again, the left hand side signal must have type reg. CSYN supports multiple such assignment to a variable to be made within one group of assignments. For example, two registers may be swapped using an intermediate register `t` as follows

```
t = x;
x = y;
y = t;
```

This will result in a flip-flop called `t` being present in the CSYN output, but in fact it will not drive anything, and so it will normally be deleted by subsequent CAE tools.

2.5.6 Sensitivity lists

The full syntax of a sensitivity list is

```
 <edge> <signal> [ or <edge> <signal> ]
```

where the edge qualifier is either null or one of `posedge` or `negedge`. The signal is the name of any net. If the net is a bus, subscripts are not allowed within the sensitivity list and the whole bus is implied. The square brackets indicate that any number of nets may be listed, separated with the keyword `or`.

CSYN-V2 supports several forms of sensitivity list, including

1. `posedge` of a single net (which will become a clock),

[2]The term 'blocking' is not helpful in the current context, but is useful in general Verilog when the assignment also contains arbitrary event control statements. This is why I have introduced the two synonyms *delayed* and *immediate* for the two types of assignment.

2. `negedge` of a single net (which will become an inverted clock),
3. `posedge` of a single net (which will become an asynchronous reset or preset) together with `posedge` or `negedge` of a single net which will become a clock,
4. `negedge` of a single net (which will become an asynchronous active low reset or preset) together with `posedge` or `negedge` of a single net which will become a clock,
5. a list of nets with no edge qualifiers.

The first two variations make an `always` block represent a block of synchronous logic with a global clock of either polarity.

The third form is the normal Verilog construct for introducing asynchronous presets or resets to the flip-flops within the block, as illustrated in section 2.8.1.

The last form allows a complex combinatorial function to be expressed using behavioural constructs (`if` and `case` etc.). All of the inputs to the function (known as the `support`) must be listed in the sensitivity list. If a signal is missing from the support list, a transparent latch would be implied. Creation of these is normally not useful and CV2 instead reports an error.

2.5.7 Behavioural Example and Style

To support registered outputs from a module, it is allowed for the same net name to appear both in a reg statement and an output statement.

For example:

```
module EXAMPLE(ck, ex);
  input ck;
  output ex;

  reg ex, ez;        // ex is both a register and an output
  reg [2:0] q;

  always @(posedge ck)
    begin
    ex = 0;
    ez = ~ez;
    if (ez) begin
           q = q + 3'd1;
           if (q == 2) ex = 1;
           end
    end
endmodule
```

The example defines a divide by two flip-flop `ez` and a three bit counter with clock enabled by ez. The output `ex` is also a register. In addition `ex` is updated twice within the `always` block, but under different conditions.

It is a matter of style whether to use many simple `always` declarations or fewer `always` declarations with large `begin-end` blocks, each containing a greater number of assignments. However, the interleaving of assignments in a simulation is not guaranteed across always blocks and can lead to random behaviour or different behaviour between simulation and synthesis. Therefore, it is usual practice to only use one always statement per clock in each module.

For synthesis, a given net can only be updated in one `always` declaration.

2.6 `$display` and `$strobe` etc.

The Verilog meta-statements `$display`, `$write`, `$finish`, `$strobe`, `$finish` and some others may be embedded in the language accepted by CSYN. They are in the syntactic category of behavioural statements and so must be inside an `initial` or `always` statement. Since these statements are only significant during simulation, CSYN completely ignores them.

2.7 Module Parameters

A Verilog *parameter* is an identifier which takes on a numerical value which is typically used to provide such things as variable bus widths or variable division ratio in a divider.

In Verilog, parameters may be defined locally to a module with a constant value or passed in as a pseudo-dynamic value from a structural instantiation of the module. CSYN only supports the former case. For example:

```
parameter bussize = 10;
parameter delta = 4;
wire [bussize-1:0] mybus = yourbus + delta;
```

2.8 Tasks

The Verilog `task` system is supported for synthesis by CSYN. Tasks must be non-recursive. Task definitions occur inside a module definition after the declaration of any free parameters, wires, regs, events and other identifiers which are referenced inside the task.

The syntax of a task definitions is

```
task <name> ; <declarative-items>* <behav-statement> endtask
```

where the declarative items are restricted to input, output, inout, net, reg, integer and parameter declarations. Owing to the vagueries of Verilog syntax, unlike a module, the task does not have a signature in parenthesis. Arguments must be passed to it in the order of the declaration statments in the task definition. There is no semicolon after an `endtask`. Here is an example

```
module TEST();

  reg a, freev;
  wire  [2:0] b;

  task andingtask;
    output x;
    input [2:0] d;
    x = (&(d)) | freev;
   endtask

always @(posedge clk) andingtask(a, b);

endmodule
```

Note that the task has scope of the enclosing module and is able to make access to the local signals of the module, such as `freev` in the example, much like free variables in Pascal or Modula.

Functions are only supported in CV2.19.

2.8.1 Asynchronous Resets and Clock Enables

The flip-flops generated by CSYN have the following signature:

```
module DTYPE(q, d, c_enable, a_reset);
  input d, c_enable, a_reset;
  output q;
  reg q;
  always @(posedge clk or posedge a_reset)
        if (a_reset) q = 0;
        else if (c_enable) q = d;
endmodule
```

Synthesis of the above Verilog will result in one flip-flop, but whether the clock-enable input is actually used depends on compiler heuristics (which never produce a clock enable term in release CV2.10).

Here is an example of behavioural Verilog which CSYN will compile into flip-flops with asynchronous resets (for `z0`) and presets (for `z1`). Asynchronous presets are actually achieved with the same `DFF` leaf flip-flop and with inversion applied to the d input and q output.

```
  reg [1:0] z0, z1;
  always @(posedge r or negedge clk)
        begin
        z0 = (r) ? 0 : d;
        z1 = 1;
        if (r == 0) z1 = d;
        end
```

CSYN may not always spot that it can use an asynchronous reset to achieve a logic function and will stop with an error message. To instruct CSYN to use clock enables or resets, direct instantiation using either the DFF_AR or the DFF_ARCE modules. The difference between them is that the former does not have a clock enable input. They can be used as follows

```
DFF_AR x1(.q(...), .d(...), .ck(...), .ar(...));

DFF_ARCE x2(.q(...), .d(...), .ck(...), .ar(...), .ce(...));
```

where the ... is replaced with a signal expression.

Here is a 3 bit binary counter with asynchronous reset.

```
// 3 bit binary counter with asynchronous reset
    module CTR3(q, ck, clear);
       input ck;
       output [2:0] q;  reg [2:0] q;
       input clear;
       wire [2:0] e_d = q + 3'd1;
       DFF_AR ctr0(.q(q[0]), .d(e_d[0]), .ar(clear), .ck(ck));
       DFF_AR ctr1(.q(q[1]), .d(e_d[1]), .ar(clear), .ck(ck));
       DFF_AR ctr2(.q(q[2]), .d(e_d[2]), .ar(clear), .ck(ck));
    endmodule
```

And here is the same using behavioural resets.

```
    module CTR3(q, ck, clear);
       input ck;
       output [2:0] q;  reg [2:0] q;
       input clear;
       always @(posedge ck or posedge clear)
              q = (clear) ? 0 : q + 3'd1;
    endmodule
```

CSYN CV2.18 will use clock enables to flip flops if they are available in the library and the `-usece` flag is given on the command line.

2.9 Primitive Modules

CSYN allows certain modules to be flagged as primitive. Primitive modules may be included by the user in his main source or in libraries. The purpose of a primitive module is to provided a prototype for a component or subcircuit which is to be later substituted for an alternative implementation. The alternative implementation may be an abstract simulation model that cannot be passed to CSYN (e.g. because it is written in C or fdl) or it might be a leaf module in the target technology library, such as a gate, pad or macrocell.

The compiler does not examine the bodies of primitive modules, except for `input`, `output` or `inout` port specification statements, and it does not include primitive module definitions in the output netlist.

There are three ways of marking that a module definition is a primitive prototype. The prototypes may be placed in their own file, such as fdlprotos.cv or myfdlprotos.cv, and this file should contain the following line

```
primitive everything;
```

at some point. This line causes CSYN to treat as primitive all further modules in the file beyond that point. Modules up to that point in the file are treated normally.

Alternatively, the keyword `primitive` may be placed in front of the `module` keyword at the start of selected modules which are primitive.

For example

```
primitive module AND2(o, x, y);
  input x, y;
  output o;
  // The body is ignored and may be absent
  // in a primitive module
endmodule
```

Thirdly, the `-libs` command line separator may be used. Here no modification or marking of the contents of the library file is needed: instead the files become primitive as a result of being after the `-libs` statement on the command line.

3 Preprocessor

Verilog tools, such as CSYN, include a preprocessor which is similar to the preprocessor defined for the C language. The following preprocessor commands are handled by CSYN.

3.1 Macros

The `define` macro allows replacement of text with other text, for instance

```
‘define red 1
‘define amber 2
‘define green (‘red+‘amber)
```

introduces three identifiers whose scope is from their point of definition onwards, through all other subsequent files. The backquote must be given before the macro name at the point of macro expansion. The definition of ‘green’ uses the fact that ‘red’ and ‘amber’ have been defined already.

Macros may be defined from the command line with the `-D` flag.

The macro ‘SYNTHESIS’ is defined by default in CSYN.

3.2 Textual Inclusion

The `include` preprocessor command takes as an argument a file name (which must be in one of the directories on the CVPATH) and textually inserts it in the source file. Includes may be nested. Example of use:

```
`include mylibrary.cv
```

3.3 Conditional Compilation

The `if`, `ifdef` and `ifndef` macro commands turn on and off processing of the following text, up to an `endif`. There is also an `else` command. The `if` command takes an argument which must be '1' or '0' (after macro expansion) and processing continues if it is a 1. The other two look up the given argument in the list of defined macros and proceed with processing if or if not respectively the macro is defined. Ifs may be nested.

```
`ifdef synthesis
 ... lines included if defined
`else
 ... lines included if not defined
`endif
```

3.4 Default nettype

The directive `default_nettype wire` causes subsequent undefined net names to become automatically defined as type `wire` when first encountered.

```
`default_nettype wire
```

3.5 Ignored Macros

The `timescale` macro, used during simulation, may be present, but is ignored by CSYN.

4 CSYN Installation and Invocation

4.1 Programs

The CSYN-V2 system consists of the executable binary file *cv2* and several library files.

For Xilinx devices, the programs *cvnl* and *lcatov* are available to convert Verilog to and from Xilinx design files respectively. These programs also operate using library files.

The directory containing these programs must be on your PATH.

4.2 Other Environment Variables

The `CVPATH` is an environment variable which defines a search path for Verilog source files. It is a colon-separated list of directories. This list should normally include the current directory (dot), the CSYN library directory, and other technology and project specific libraries.

4.3 CSYN Command line arguments

CSYN is run with the following command

```
cv2  [ options ] filename [ filename ... ] [ -libs filename [ filename ... ]]
```

CSYN takes a list of one or more source files. Nothing is inferred from order of them, except for those that come before and after the `-libs` separator and the contents of all of the files are essentially concatenated during internal processing. The source files must either be in the current directory or in one of the directories given by the shell variable CVPATH.

The CSYN program expects Verilog source files to have the suffix '.cv' and will try to append such a suffix, unless the user-supplied filename already has a dot in it.

As well as the files that you specify, CSYN also reads a technology library specified using the `-tech` command line argument. This library defines the gates which the compiler will use in its synthesised logic and other leaf cells which are required for certain technologies, such as IO pads etc..

If the `-tech` command line argument is omitted, the standard library file 'cvgates' will be used. This contains a set of unit delay 'vanilla' gates. The library file must be on the CVPATH.

The source files must contain a Verilog module whose name is supplied in the `-root` command line option and also all modules structurally instantiated by the root and its instantiated modules. Any other module not used but present in the source files are checked for correct syntax by the compiler, but not processed otherwise.

4.3.1 Root command line option

The command line option

```
-root modname
```

must refer to the top level module that you wish synthesised. The arg string 'modname' refers to the name of the module. All leaf modules required for this module must be found and will be cross checked for contact rules.

4.3.2 The -I command line option

The `-I` option allows a string to be specified which overrides the (or any) `CVPATH` environment variable. The string may either be the next command line argument or else directly after the `-I` as part of the same textual argument.

4.3.3 Warn command line option

If the `-warn` option is given, missing modules will cause only a warning, allowing partial checking to be done.

4.3.4 Libraries command line separator

The token `-libs` may occur on the command line and it has the effect of making all of the contents of all of the Verilog source files listed after the `-libs` separator primitive. This is useful if the simulation models of library components are going to be used only for their signatures during compile. The use of `-libs` is now the recommended method of using primitives, as opposed to the `primitive everything` statement inside the library file itself.

4.3.5 Compile-only command line option

The `-c` command line option disables most of the compiler and enables it to be used as quick source file syntax checker.

4.3.6 Macro define command line option

The `-Dname` command line option causes 'name' to defined as a macro, so that `'ifdef name` will succeed.

4.3.7 Verilog output command line option

The `-vnl` command line causes the output from the compiler to be a Verilog netlist. A Verilog Netlist contains module definitions where the modules contain only wire definitions and instances of other more primitive modules.

4.3.8 Cambridge HDL output command line option

The `-cam` command line causes the output from the compiler to be a Cambridge HDL netlist.

4.3.9 Relax command line options

The `-relax` command line causes the compiler to disable certain warning and error messages. In particular it causes net names which have not be defined to become defined as single width wires when first encountered.

The `-relax-xz` command line causes the compiler to allow 'x' and 'z' as net names, although these are normally a bad idea.

The `-relax-ec` command line causes the compiler to allow and ignore hash style event control imbedded in assignments and between behavioural statements. There are few cases in Verilog where ignoring this style of event control causes differences between simulation and synthesis, and these are mostly due to not including hash delays in the source, therefore having them and ignoring them for synthesis is generally fine. This flag should perhaps be active by default ?

4.3.10 Arg file command line options `-f` and `-files`

The `-f <filename>` causes a file to be opened from which commands are read as though on the command line. The `-files` causes the file called "files" to be opened from which commands are read as though on the command line. The commands can be split over many lines, with all whitespace characters counting as delimiters. The file must lie in the current directory or a full path name given.

4.3.11 Verbose command line option

The `-verbose` command line option causes the compiler to augment certain error messages with additional information. This is useful if an error is hard to understand.

4.3.12 Verbose command line option

The `-usece` command line option causes the compiler to use clock enables if possible.

4.3.13 Output file name option

The `-o <filename>` option enables the output file name to be specified for synthesised netlists. The default is the standard output.

4.3.14 Listmods command line option

The `-listmods` option causes the compiler to print a list of the modules that have been defined in the set of input files.

4.3.15 Dot HDL command line option

When using the `-cam` output format, if filenames ending `.hdl` are supplied to the compiler they are not treated as Verilog source files but instead they are added to the list of imports included in synthesised Cambridge HDL object code.

4.3.16 The behavioural compiler command line option.

The `-bc` command line flag enables CV2's behavioural compiler. This compiler allows a much richer source language to be accepted. Documentation for this will be provided separately.

4.3.17 Other Command line options

The following command line options do not need to be used in normal use.

The lookahead command line option.
The `-lookahead` option causes adders and subtracters generated by the system to use full-lookahead carry chains. When it is missing, the compiler generates ripple carries. This option is not for normal use.

The dontcare command line option.
The `-dontcare0` option causes the value zero to be used in logic expressions where 'x', the don't care value is give. Otherwise '1' is used. If CSYN contained a fuller logic minimiser, dont care's would be properly handled and this option would be not needed.

Dontsimplify command line option
This option is designed primarily for advanced debugging and is not for normal use. The `-dontsimplify` command line option disables the internal logic simplifier. This causes verbose logic output. This option is not for normal use.

Dontshare command line option.
The `-dontshare` command line option disables the internal logic matcher which causes gates that have already been generated to produce a particular logic function, within a module, to be reused for parts of other functions in that module. This flag causes verbose logic output. This option is not for normal use.

Trace command line option.
This option is designed primarily for advanced debugging and is not for normal use. The `-trace n` option enables internal compiler tracing to be turned on. This feature is normally only used for debugging the compiler. The arg n is a decimal number where each bit in its binary representation enables a particular trace mode. The trace modes are

```
TRACE_TOKENS 1
TRACE_LRGEN 2
TRACE_REDUCS 4
TRACE_TREE 8
TRACE_DISPATCH 16
TRACE_BEHEV 32
TRACE_DRIVENETS 64
TRACE_BUILD 128
TRACE_OUTPUT 256
TRACE_CROSS 512
TRACE_ARGS 1024
TRACE_LCE 2048
TRACE_EXPR 4096
TRACE_PANDEX 8192
```

Working pin cost command line option.
The `-wpc n` command line option enables the working pin cost value used by the compiler to be modified. This is an integer which is used in an internal heuristic which determines what complexity of subexpression is worth maintaining rather than simplifying and merging into adjacent logic. Maintaining a subexpression is useful since its output can be fanned out to several other places. Its value should be appropriate to the complexity of combinatorial function generator available in the target technology. It has little effect unless `-dontshare` is used.

5 Technology Libraries (and other files)

The currently available technology libraries are: xi4000, xi3000 and cvgates.

The CSYN compiler generates these gates: BUF MUX2 AND2 OR2 INV XOR2 DFF TLATCH. These must be present in all technology libraries used with CSYN. They must have the following signatures.

```
CVMUX2(o, a, b, c);
DFF(q, d, ck, ce, ar, spare);
XOR2(o, i1,i2);
AND2(o, i1,i2);
OR2(o, i1,i2);
BUF(o, i);
INV(o, i);
```

CSYN can also output any other gate or module provided that it is structurally instantiated in the source file and a Verilog primitive prototype is provided.

5.0.18 Xi3000 Technology Library

This library contains

```
INFF3(p, i, ck);
CVMUX2(y, s, s1, s0);
CLBDELAY(o, i);
CLBMAP7(x, y, a, b, c, d, e);
GCLK(o, i);
ACLK(o, i);
IBUF(p, i);
INFF(p, direct, i, ck);
INLAT(p, q, g);
OUTFFZ(p, s, ck, t);
OUTFF(p, s, ck);
OBUFZ(p, s, en);
OBUF(p, s);
FROBUF(p, s, ck);
FOUTFF(p, s, ck);
FOBUF(p, s);
BUFZ(p, s, oe );
TRIPAD(p, s, oe);
IOBUFZ(p, s, oe, i);
DFF(q, d, ck, ce, ar, gr);
TBUF(o, i, t);
XOR2(o, i1,i2);
XOR3(o, i1, i2, i3);
XOR4(o, i1,i2,i3,i4);
AND2(o, i1,i2);
```

```
AND3(o, i1,i2,i3);
AND4(o, i1,i2,i3,i4);
AND5(o, i1,i2,i3,i4,i5);
OR2(o, i1,i2);
OR3(o, i1,i2,i3);
OR4(o, i1,i2,i3,i4);
OR5(o, i1,i2,i3,i4,i5);
INV(o, i);
BUF(o, i);
LO(lo);
HI(hi);
LONGLINE(i);
CRITICAL(i);
UNCRITICAL(i);
PULLUP(i);
```

5.1 Xi4000 Technology Library

This library contains

```
INFF3(p, i, ck);
CVMUX2(y, s, s1, s0);
CLBDELAY(o, i);
CLBMAP7(x, y, a, b, c, d, e);
BUFGP(o, i);
BUFGS(o, i);
GLOBAL_RESET(i);
IBUF(p, i);
INFF(p, direct, i, ck);
INLAT(p, q, g);
OUTFFZ(p, s, ck, t);
OUTFF(p, s, ck);
OBUFZ(p, s, en);
OBUF(p, s);
FROBUF(p, s, ck);
FOUTFF(p, s, ck);
FOBUF(p, s);
BUFZ(p, s, oe );
TRIPAD(p, s, oe);
IOBUFZ(p, s, oe, i);
DFF(q, d, ck, ce, ar, gr);
TBUF(o, i, t);
XOR2(o, i1,i2);
XOR3(o, i1, i2, i3);
XOR4(o, i1,i2,i3,i4);
AND2(o, i1,i2);
AND3(o, i1,i2,i3);
AND4(o, i1,i2,i3,i4);
AND5(o, i1,i2,i3,i4,i5);
OR2(o, i1,i2);
OR3(o, i1,i2,i3);
OR4(o, i1,i2,i3,i4);
OR5(o, i1,i2,i3,i4,i5);
INV(o, i);
BUF(o, i);
LO(lo);
HI(hi);
```

```
LONGLINE(i);
CRITICAL(i);
UNCRITICAL(i);
```

5.2 Xi4000macros Hard Macros Library

The Xi4000 hard macros library contains various counters and other components which use the Xi4000 fast carry chains.

A macro library can be used with something like the following set of command lines:

```
$ cv2 -tech xi4000 xi4000macros.cv -root MYCHIP -o a.vnl \
     -vnl macrotest.cv
$ cvnl -tech xi4000 xi4000macros.mas -device 4010pq160 \
     -root MYCHIP a.vnl -xnf -o mychip.xnf
```

The libraries x4ram and xcadders are useful.

6 CVABEL

CVABEL is a program that reads a Verilog source file and compiles it using logic synthesis into either an Abel FPGA description or a scalar, flattened Verilog module.

6.1 Abel output

Abel is a language used as input to many FPGA systems. By writing in Verilog and then converting to Abel via CVABEL, it is possible to keep all of your source code in one flexible form and also to do simulations at the system level all in Verilog.

6.2 Verilog Scalar output

Scalar Verilog is the opposite to a net list. There are no component instances, only behavioural expressions. To use this, all leaf components must contain a Verilog behavioural model. CVABEL will collect all of these behavioural statements together and reduce all bus operators (such as addition or comparison) into simple Boolean operators.

See DJG for more details.

7 CVNL

CVNL is program that will read in Verilog netlists (.vnl) files and write them out in a selected format. Before processing, the program cross-checks all input and output ports to ensure Verilog's wiring rules are met. The program will flatten the design starting from the module specified with `-root` on the command line and proceed downwards until encountering a module which is declared as primitive or a member of a technology library. The resulting flattened module is then output in the format selected. If the `-dontflatten` command line flag is given, all instances within the root module will be considered primitive and so not expanded (flattened), hence only the root module will be output.

Modules may be declared primitive with the old style 'primitive' keyword. However the preferred way of doing this now is to put them in a file whose name appears after the flag `-libs` in the command line.

Note that CVNL will only accept a subset of the Verilog language as its input. Full-blooded Verilog modules must first be compiled into vnl with CSYN or another program.

Note, the older programs CVAUX and CVXNF are subsumed by CVNL.

7.1 Description of CVNL command line and options

The format of the command line is

```
cvnl [ options ] fn [ fn ... ] -libs fn [ fn ...] \n", argv[0]);
```

The files before the `-libs` flag are the primary source files which will be processed and those after are library files which will be used only for cross-checking.

7.1.1 CVNL netlist selection options

The output formats available are as follows.

`-xnf` is used to request output in Xilinx netlist format.

`-xnfmodule` is used to request output in Xilinx netlist format with external nets defined using the XNF `SIG` construct.

`-cam` is used to request Cambridge HDL netlist format.

`-vnl` is used to request Verilog netlist format.

`-ranger3` is used to request Seetrax Ranger 3 netlist format.

`-edif` is used to request EDIF netlist format.

With ranger3 format, the `-powernames` option enables a list of comma-separated additional supply names to be supplied.

7.1.2 Other CVNL options

The '-root rootmodule' command line flag selects a module of those present in the input files to be used as the root for output and or flattening.

`-tech <technology-library>` specifies a technology library file. The library file must be on the CVPATH and has suffix '.mas'. For example, to use the xi3000 library the file 'xi3000.mas' must be on the CVPATH. Only the xnf output format currently requires a technology library to be given.

`-o <outputfile>` specifies the output file name. The '.xnf' suffix should be supplied explicitly for xnf outputs.

The `-device <part>` option enables the target device name string to be entered from the command line for inclusion in the output file. Not all output formats use this. CVNL does not check that a part is valid.

The `-ncd` command line option disables module interconnectivity checks, enabling inputs to be undriven and outputs to be joined, etc..

The `-autogen` option causes the program to render a used module in the output even when the module definition is not present in the target technology library (a prototype must be present in the included input files though).

The `-report` option causes the program to report how many gate inputs and outpus are in use and related statistics. Report is perhaps not implemented today.

If the `-dontflatten` command line flag is given, all instances within the root module will be considered primitive and so not expanded (flattened), hence only the root module will be output.

7.2 Using CVNL to generate EDIF

The '-edif' command line flag selects an EDIF netlist output format.

7.3 Using CVNL to generate Seetrax Ranger netlists.

CVNL can generate a Seetrax Ranger netlist when it is given the `-ranger3` command line option. A typical command line would be

```
$ cvnl -root PCB myfile.vnl -libs mylib1 mylib2 -ranger3 -o outputfn
```

Note that the root module may not have ports for a Ranger3 output.

Note that it is not possible to pass a bus into a component in the Ranger3 format: therefore every connection to a component from a bus must be spelled out separately. This does not mean that busses cannot be declared as such with ranged declarations: they simply may not be used as such.

The pin numbers generated in the Ranger3 output format are in the order found on the declaration of the component. This will be the order of the connections in a positional instantiation, but for an associative instantiation (using dot format) (which is preferred for large devices) the order in the instance is clearly unimportant.

7.4 CVNL used as a Verilog flattener

The cvnl program is able to read a Verilog hierarchic netlist and write out a flat Verilog netlist containing a single module. It may be run as follows:

```
$ cvnl -vnl -o <fn> -root <rootmodule> <srcfiles>+
```

7.5 CVNL used to generate XNF

CVXNF is a program which process a structural Verilog source file. The input is typically the output from CSYN where the `-vnl` option has been used.

To generate XNF for Xilinx series 3000 devices the program needs a command line with the following fields

```
cvnl -xnf -tech xi3000 -o <outputfile>
      [ -device <device> ] [ -root <root>] <sourcefiles> ...
```

The source files must contain Verilog definitions of the root module and all modules structurally instantiated in its components. Anything not defined, or defined as primitive, is assumed to be a leaf in the technology library in use. If such a module is not in the named library, then an error is generated, or if the `autogen` command line flag is given, it generates a module of the same name in the netlist (so that xnfmerge or similar have later to resolve them).

8 Making FPGA's with CSYN

CSYN has been used mostly to generate Xilinx designs.

There are many possible routes available. The main options are whether to put your top-level files through CSYN or whether to manually edit them and whether to use the Olivetti `xnf` program or the `cvnl` program.

In all cases we have the following basic steps:

1. Partition your Verilog to include a module which is a feasible Xilinx chip. This must include a pad for each port, or multiple pads for bus ports.
2. Compile your Verilog using CSYN to generate a structural Verilog netlist (.vnl file).
3. Generate XNF (Xilinx Netlist Format) from the netlist using CVNL.
4. Send the XNF to the Xilinx routing tools.

8.1 Xilinx Pads

Your Verilog must contain a module which is the Xilinx chip, such as MYCHIP. The ports of the module will become the pins of the Xilinx device.

```
  module MYCHIP(a_p, b_p, ...

    wire a;
    input a_p;
    IBUF pada (a_p, a);

    wire b;
    output b_p;
    OBUF padb (b_p, b);

    // more pads go here

    MYCHIPCORE mychipcore(.a(a), .b(b), ....);

 endmodule
```

The port names have, by convention, been given a suffix of _p, but this is not required. Each input or output must go through a pad.

If the pad instance name has a the form 'pNNN' where 'NNN' is an integer then the xnf tools will lock down the port to a pin on the chip. For first time layouts, normal practice will be not to lock down any ports to facilitate optimal pad placement by the Xilinx tools. For pin-grid array devices, the pad instance name must have the form 'p_XY', where 'X' and 'Y' are a letter and a one or two digit number.

All of the core logic of the device has, by convention, been placed in a module, but this is not required.

8.2 Busses

When busses are to be brought in and out of a chip, they must be manually split into separate pads. For example

```
  module MYCHIP(a_p, ...
    wire [2:0] a;
    input [2:0] a_p;
    IBUF pada0 (a_p[0], a[0]);
    IBUF pada1 (a_p[1], a[1]);
    IBUF pada2 (a_p[2], a[2]);
```

8.3 Global Clock Networks

Xilinx devices contain global clock nets and associated driver buffers. These can be used by explicitly routing the clock through its buffer and leaving the Xilinx tools to assign the routing to the special nets. Here is an example of how to use the global clock on an Xc3000 device.

```
input clock_p;
wire clock_in, clk;

IBUF p30 (clock_p, clock_in);
GCLK gclk (clk, clock_in);
```

```
always @(posedge clk) ....
```

On the 3000 series devices there is also the alternate clock buffer 'ACLK' for designs with two major clocks.

8.4 Xilinx Soft Macros

Xilinx soft macros have not been used to date, but can easily be entered into the system, preferably as non-primitive Verilog modules in their own library.

8.5 Xilinx Hard Macros

Xilinx hard macros are available as 'leaf' cells at the XNF level. Use of these macros is currently the preferred way to access the low-level Xilinx carry chains etc.. The files xi4000macros.cv and xi4000macros.mas contain definitions of the macros. The first file should be supplied to CSYN from synthesis and the second should be included on the command line of CVXNF during xnf generation.

9 A collection of examples

```
// first.cv

// A divide by 16 counter with an exciting output.
module CTR16(ck, o);
  input ck;
  output o;

  reg [3:0] count;
  always @(posedge ck)
     begin
     // Add a four bit decimal value of one to count
     count <= count + 4'd1;
     if (count == 14) count <= 1;
     end

  // Note ^ is exclusive or operator
  assign o = count[3] ^ count[1];

endmodule

// Top level simulation model
module SIMSYS();

  wire ck;
  wire boo;
  CLK10MHz clock(ck);
  CTR16 counter(ck, boo);
endmodule

// end of first.cv
```

Here is an example module which generates a digital bit stream equivalent to that found in many CD players.

```
// daudio.cv
```

```
module CDSRC(sds, aclk, adata);

  input aclk;  // audio clock
  output sds;  // side select
  output adata;// audio data bit stream
  wire clk = aclk;

  reg [4:0] phase;
  reg [15:0] audio_data;
  reg sds, adata;

  always @(posedge clk)        begin
        if (phase == 19) begin  // This value may vary from one cd to another
                phase <= 0;
                end
        else phase <= phase + 1;

        if (phase == 15)
                begin
                  // The 'sound' is just a count sequence generated here
                audio_data <= audio_data + 16'h0201;
                sds <= ~sds;
                end
  end

  always @(posedge clk)
        if (phase < 16) adata <= audio_data[phase];
        else adata <= 0;
endmodule
```

Here is a decoder for the HDB3 line code used in European trunk telehone systems.

```
// cbg HDB3DEC
module HDB3DEC(ck, rxpos, rxneg, dout);

input   ck;        // Clock input
input   rxpos;        // Positive pulse input
input   rxneg;        // Negative pulse input
output  dout;        // Decoded data out


  reg   R_a, R_b, R_c, R_pol;

  wire  ab = rxpos | rxneg;
  wire  delta = rxpos ^ R_pol;
  wire  mark = ab & delta;

  // This signal is not brought out, but could be.
  wire  violation = ab & ~delta;

always @(posedge ck)
  begin
        R_c <= mark;
        R_b <= R_c;
        R_a <= R_b;
        R_pol <= (~ab & R_pol) | (ab & rxpos);
  end

  assign dout = R_a & ~violation;

endmodule // cbg HDB3DEC
```

Here is the associated HDB3 encoder.

```
module HDB3ENC (ck, din, txpos, txneg);

  input ck, din;        // Input clock and data
  output txpos, txneg;        // Output positive and negative pair


  reg R_a, R_b, R_c, R_d;
  reg [1:0] R_ct;
  reg R_apout, R_amout;
  reg R_pol, R_lastv;

  wire firstof4, viol_needed, mark;

  assign viol_needed = (R_ct == 2'b11) & ~R_a;

  always @(posedge ck)
  begin

        // Input data shifted into four bit register.
        R_a <= R_b;
        R_b <= R_c;
        R_c <= R_d;
        R_d <= din;


        // Counter of number of zeros, reset on ones.
        R_ct <= (R_a) ? 2'b00 : R_ct + 2'1;

        // Polarity reverses on bona-fida marks;
        R_pol <= R_pol ^ mark;

        // Save polarity of last violation
        R_lastv <= (viol_needed) ? ~R_pol :  R_lastv;
  end


  // This is the magic rule defined in the spec.
  assign mark = R_a | (firstof4 & (R_lastv ^ R_pol));

  // Detect first zero of a group of 4.
  assign firstof4 = ~R_a & ~R_b & ~R_c & ~R_d & (R_ct == 2'b00);


// Generate the two outputs and register them.
  always @(posedge ck)
  begin
        R_apout <= ((mark & R_pol) | (viol_needed & ~R_pol));
        R_amout <= ((mark & ~R_pol) | (viol_needed & R_pol));
  end

  assign txpos = R_apout;
  assign txneg = R_amout;

endmodule  // hdb3 encoder;
```

Here is an example traffic light controller which uses the `case` statement and some simple macros.

```
// Traffic light controller
    `define red 1
    `define amber 2
    `define green 4

    module TRAFFIC_LIGHTS(clk, lights);
```

```
    input clk;
    output [2:0] lights;
    reg [2:0] lights;

    reg [1:0] phase;

    always @(posedge clk)
     begin
        case (phase)
         0 : lights = `red;
         1 : lights = `red + `amber;
         2 : lights = `green;
         3 : lights = `amber;
        endcase
        phase = phase + 1;
     end
endmodule /* TRAFFIC_LIGHTS */

module SIMSYS();
   wire clk;
   wire [2:0] lights;
   CLK1MHz clk(clk);
   TRAFFIC_LIGHTS traffic_lights(clk, lights);
endmodule
```

10 XNFTOV: Conversion of Xilinx .xnf file to Verilog

XNFTOV may be run as follows:

```
$ xnftov infn.xnf -o outfn
```

The input file must be an XNF file, typically generated by using the Xilinx tool 'lca2xnf' with the '-g' flag.

The output will be called '`outfn.vnl`' and will be a Verilog module containing one module called 'BACK_MODULE'. The module uses zero delay gates which may be found in the library 'zgates.cv'. The module will also instantiate Xilinx I/O pads, clock buffers and DFFs.

11 CSYN Restrictions and Extensions

CSYN does not support the built-in leaf gates defined in Verilog, such as 'and' and 'or', but a standard library supplies a set of radixed gates, including 'AND2', 'AND3', 'AND4', 'OR2', 'OR3', 'OR4' and 'XOR2', 'XOR3', 'XOR4'.

Multi-symbol macros are not supported by current CSYN.

Array types are not supported by current CSYN.

The drive strength and delay qualifiers of a full Verilog continuous assignment are not supported by CSYN.

Disable, specify, named blocks and functions are not supported by CSYN-V2.15.

End of document.