1 The Verilog Language—A Learner's subset.

This short document is intended to contain all you need to know to write synthesisable Verilog for basic hardware designs. It was written to help users of the old CSYN compiler, so please ignore references to that.

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 taxononomy of four levels is handy for the current purposes.

1. Structural Verilog: a hierarchic netlist form of the type generated by the CSYN compiler.
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`.

Figure 1 shows some Verilog code and the diagrammatic representations of the hardware resulting from compilaton with CSYN.

1.1 Verilog Lexicography and Comments

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

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.
```

1.2 Preprocessor

Verilog has a preprocessor which can be used to define textual macros as follows

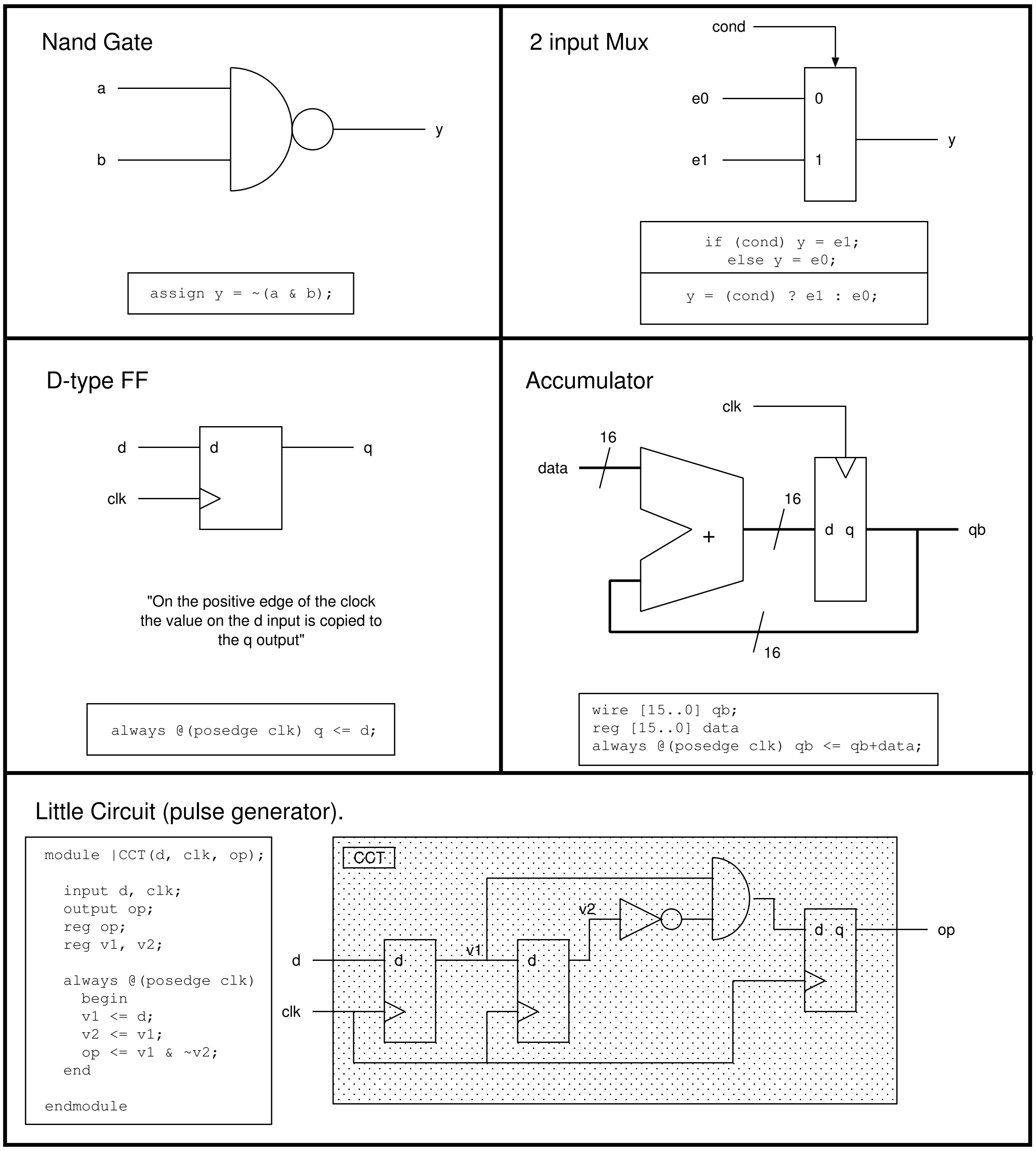
```
  ‘define RED 3
  ‘define BLUE 4

      if (a== ‘RED) b <= ‘BLUE;
```

Note that the reverse quote is used both in macro definitions and in their use.

The preprocessor can also be used to control conditional inclusion of parts of the source file. This is useful is variations are needed depending on whether the design is to be simulated or synthesised.

```
‘ifdef SYNTHESIS
```


Nand Gate
a
b
y
2 input Mux
cond
e0
e1
0
1
y
D-type FF
d
clk
d
q
"On the positive edge of the clock the value on the d input is copied to the q output"
Accumulator
clk
16
data
16
+
d q
qb
16
Little Circuit (pulse generator).
CCT
d
clk
d
v1
d
v2
d q
op

```
assign y = ~(a & b);
```

```
if (cond) y = e1;
else y = e0;
```

```
y = (cond) ? e1 : e0;
```

```
always @(posedge clk) q <= d;
```

```
wire [15..0] qb;
reg [15..0] data
always @(posedge clk) qb <= qb+data;
```

```
module |CCT(d, clk, op);

  input d, clk;
  output op;
  reg op;
  reg v1, v2;

  always @(posedge clk)
    begin
    v1 <= d;
    v2 <= v1;
    op <= v1 & ~v2;
  end

endmodule
```

Figure 1: Verilog Code and Corresponding Circuits

```
   ...
   Lines to be included when compiling
   ...
‘else
   ...
   Lines to be included when simulating
   ...
‘endif
```

Note that the cv2 compiler defines the macro SYNTHESIS automatically.

1.3 Identifiers

Identifier names consist of alphanumeric strings which can also contain underscores and must start with a letter. Case is significant. In this manual, capitalised 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.

1.4 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 at the end of atomic statements, but not composite statements. Hence there is never one after an `endmodule` (or an `end` or an `endcase`).

Useful module declarative items include the following statements `input`, `output`, `inout`, `wire`, `reg`, `tri`, `assign`, `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.

1.5 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.

1.6 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 refered 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 width is not given in the formal parameter list, just its identifier.

The most significant bit is always specified first and the least significant bit index second. This convention is understood by certain built-in arithmetic operators, such as addition. Users, such as die-hard IBM employees who wish to number their bits starting with '1' as the most significant are free to do so, but normally, the first index is the number of bits in the bus minus one and the second index is zero. The least significant index does not have to be zero and this is helpful sometimes - e.g. the least significant two bits are normally missing in an A32, byte addressed address bus. In summary, the numbers use to index a bus are just labels on the bit lanes and do not introduce any implied shifts or bit-reversals when the busses are used in expressions.

1.7 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, 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.
    ...
```

Local nets which will be driven from the outputs of instantiated modules and by continuous assignments should be of type `wire`.

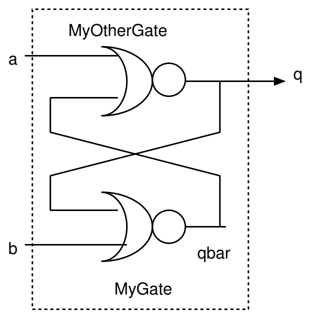


Figure 2: Example Verilog Module: FREDs

Local nets which are to be assigned to from behavioural statements should be of type `reg`, wherease those assigned in continuous assignments should be type `wire`.

Outputs which are to assigned to from behavioural statements should be defined twice, first as outputs and then with type `reg`.

1.8 Structural Specification

One further statement type, beyond the wire and port declarations, is all we need for structural specification of a hierarcical 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.

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 recommened 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 explict 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.

1.9 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 through 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> ( <parameter-expression> )
```

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

I use the term 'parameter 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 signal expression which results in a signal of the correct width. Signal expressions are explained shortly in section 1.10. [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 1.12.

1.10 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 (vertical bar) 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. Logic minimisation 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.

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.

1.11 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

```
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;
```

1.12 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.

1.13 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.

All operators can be applied to simple nets or to busses. When a diadic operator 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 1.14) their arguments and then combine them in a unit width AND or OR gate.

1.14 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. 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 x = (& (mybus));
```

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

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
/	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.

Symbol	Function
&	and
|	or
^	xor
~&	and with final invert
~|	or with final invert
~^	xor with final invert

Table 2: Verilog Unary Reduction Operators

1.15 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;
```

All comparison operators in Verilog treat their arguments as unsigned integers. There is no explicit support for basic signed operations on two's complement or sign extensions.

1.16 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 let 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
```

1.17 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) ignores the contents of `initial` statements which simply assign to a variable: these may be present and are useful when simulating the source file.

The following form of the Verilog `always` construct is the most useful for RTL designs. It has the syntax

```
always @(posedge <clocksignal>) <behavioural-statement>
```

This statement causes execution of the behavioural statement each time the clock goes from zero to one. An alternative keyword `negedge` has the expected effect.

1.18 `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`.

1.19 `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>
```

1.20 `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.

1.21 `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. Here is an example

```
 always @(aal_counter) case (aal_counter)  // full_case
        0:  aal_header <= 8'h00;
        1:  aal_header <= 8'h17;
        2:  aal_header <= 8'h2D;
        3:  aal_header <= 8'h3A;
        default:  aal_header <= 8'h74;
        endcase
```

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.)

1.22 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 1.10).

1.22.1 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. CSYN supports only one such assignment to a variable to be made within a program, except that multiple asssignments can be made if they are in separate clauses of an `if-then-else` construct. For example, to swap to registers we may use

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

A variable may only be updated by one delayed assignment per clock cycle.

1.22.2 Blocking or 'immediate' assignment

Verilog also supports assignments using the '=' operator. These take effect immediately, rather than at the next timestep.

1.23 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;       // This is the clock input
  output ex;

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

  always @(posedge ck)
    begin
    if (ez) begin
           if (q == 2) ex <= 1; else ex <= ~ex;
           q <= q + 3'd1;
           end
    ez <= ~ez;
    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. Since `q` is a three bit register, it will count 0, 1, 2, ... 7, 0, 1 ...

2 Unsynthesisable Constructs

Several statements cannot normally be turned into hardware, but are useful for generating test wrappers to exercise a design under simulation.

Hash Delays

The clock modules found in the libraries and the back-annotated modules from XNFTOV contain hash delays. These actually have three forms:

1. In a continuous assignment: '`assign #n v = exp;`'
2. Inside a behavioural assignment: '`v <= #n exp;`'
3. Between behavioural statements: '`begin s1; #n s2; ... end`'

The first two forms introduce a delay between calculating the value of an expression and the assigned variable taking on the new value. The third form actually pauses the flow of a 'thread' of execution. These constructs should not be fed into the synthesiser and should only be used for test harnesses in simulation.

Simulator Meta-commands.

The `$display` command instructs the simulator to print output. This command and others like it are ignored entirely by the synthesiser. The first argument is a string. The string contains tokens introduced with a percent sign that control printing of the remaining arguments. The percent operators available are

- h : print a hex value
- d : print a decimal value
- b : print a binary value
- m : print the surrounding module instance name (does not consume an argument from the list
- t : print the simulation time (must tie up with $time in the argument list).

Here is an example

```
always @(posedge clk) $display("module %m: time %t, bus value=%h",
             $time, mybus);
```

This will give output that might look like

```
module SIMSYS_mychip_gate1: time 10445, bus value=4DF
```

A clock Generator

To generate a free running clock, one can use

```
 module CLOCK(q);

   output q; reg q;

   // Toggle ever 50 or so time units.
   initial begin q = 0; forever #50 q = !q; end

endmodule
```

A Reset Pulse

To generate a reset pulse at the start of a simulation, one can use

```
 module POWER_ON_RESET(q);

   output q; reg q;

   // Toggle ever 50 or so time units.
   initial begin q = 1; #200 q = 0; end

endmodule
```

End of document.