Introduction to Verilog

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction ..................................................</td>
<td>1</td>
</tr>
<tr>
<td>2. Lexical Tokens ..................................................</td>
<td>2</td>
</tr>
<tr>
<td>White Space, Comments, Numbers, Identifiers, Operators, Verilog Keywords</td>
<td></td>
</tr>
<tr>
<td>3. Gate-Level Modelling ...........................................</td>
<td>3</td>
</tr>
<tr>
<td>Basic Gates, buf, nor Gates, Three-State Gates, bufif1, bufifi1,notif1, notififi1</td>
<td></td>
</tr>
<tr>
<td>4. Data Types .......................................................</td>
<td>4</td>
</tr>
<tr>
<td>Value Set, Wire, Reg, Input, Output, Inout</td>
<td></td>
</tr>
<tr>
<td>Integer, Supply0, Supply1</td>
<td></td>
</tr>
<tr>
<td>Time, Parameter</td>
<td></td>
</tr>
<tr>
<td>5. Operators ..........................................................</td>
<td>6</td>
</tr>
<tr>
<td>Arithmetic Operators, Relational Operators, Bit-wise Operators, Logical Operators</td>
<td></td>
</tr>
<tr>
<td>Conditional Operator: &quot;?&quot; Operator Precedence</td>
<td></td>
</tr>
<tr>
<td>6. Operands ............................................................</td>
<td>9</td>
</tr>
<tr>
<td>Literals, Wires, Regs, and Parameters, Bit-Selects &quot;x[3]&quot; and Part-Selects &quot;x[5:3]&quot;</td>
<td></td>
</tr>
<tr>
<td>7. Modules .............................................................</td>
<td>10</td>
</tr>
<tr>
<td>Module Declaration, Continuous Assignment, Module Instantiations, Parameterized Modules</td>
<td></td>
</tr>
<tr>
<td>8. Behavioral Modeling ..............................................</td>
<td>12</td>
</tr>
<tr>
<td>Procedural Assignments, Delay in Assignment, Blocking and Nonblocking Assignments</td>
<td></td>
</tr>
<tr>
<td>begin ... end, for Loops, while Loops, forever Loops, repeat, disable, if ... else if ... else</td>
<td></td>
</tr>
<tr>
<td>case, cases, case</td>
<td></td>
</tr>
<tr>
<td>9. Timing Controls ...................................................</td>
<td>17</td>
</tr>
<tr>
<td>Delay Control, Event Control, @, Wait Statement, Intra-Assignment Delay</td>
<td></td>
</tr>
<tr>
<td>10. Procedures: Always and Initial Blocks ..................</td>
<td>18</td>
</tr>
<tr>
<td>Always Block, Initial Block</td>
<td></td>
</tr>
<tr>
<td>11. Functions .......................................................</td>
<td>19</td>
</tr>
<tr>
<td>Function Declaration, Function Return Value, Function Call, Function Rules, Example</td>
<td></td>
</tr>
<tr>
<td>12. Tasks ...............................................................</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>13. Component Inference ..........................................</td>
<td>22</td>
</tr>
<tr>
<td>Registers, Latches, Flip-flops, Counters, Multiplexers, Adders/Subtracters, Tri-State Buffers</td>
<td></td>
</tr>
<tr>
<td>Other Component Inferences</td>
<td></td>
</tr>
<tr>
<td>14. Finite State Machines .......................................</td>
<td>24</td>
</tr>
<tr>
<td>Counters, Shift Registers</td>
<td></td>
</tr>
<tr>
<td>15. Compiler Directives .........................................</td>
<td>26</td>
</tr>
<tr>
<td>Time Scale, Macro Definitions, Include Directive</td>
<td></td>
</tr>
<tr>
<td>16. System Tasks and Functions ..................................</td>
<td>27</td>
</tr>
<tr>
<td>Sdisplay, Sstrobe, Smonitor; Stime, Sstime, Srealtime; Sreset, Sstop, Sfinish; Sdeposit; Sscope, Sshowscope; Slist; Srandom, Sdumpfile, Sdumpvar, Sdumpoff, Sdumpall; Sshm_probe, Sshm_open, Sfopen, Sfdisplay, Sfstrobe, Sfmonitor.</td>
<td></td>
</tr>
<tr>
<td>17. Test Benches ....................................................</td>
<td>30</td>
</tr>
<tr>
<td>Test Benches, Synchronous Test Benches</td>
<td></td>
</tr>
<tr>
<td>18. Memories ..........................................................</td>
<td>32</td>
</tr>
<tr>
<td>Two-dimensional arrays, Initializing memory from a file.</td>
<td></td>
</tr>
</tbody>
</table>

1. Introduction

Verilog HDL is one of the two most common Hardware Description Languages (HDL) used by integrated circuit (IC) designers. The other one is VHDL. HDL’s allows the design to be simulated earlier in the design cycle in order to correct errors or experiment with different architectures. Designs described in HDL are technology-independent, easy to design and debug, and are usually more readable than schematics, particularly for large circuits.

Verilog can be used to describe designs at four levels of abstraction:

(i) Algorithmic level (much like c code with if, case and loop statements).
(ii) Register transfer level (RTL uses registers connected by Boolean equations).
(iii) Gate level (interconnected AND, NOR etc.).
(iv) Switch level (the switches are MOS transistors inside gates).

The language also defines constructs that can be used to control the input and output of simulation.

More recently Verilog is used as an input for synthesis programs which will generate a gate-level description (a netlist) for the circuit. Some Verilog constructs are not synthesizable. Also the way the code is written will greatly effect the size and speed of the synthesized circuit. Most readers will want to synthesize their circuits, so nonsynthesizable constructs should be used only for test benches. These are program modules used to generate I/O needed to simulate the rest of the design. The words “not synthesizable” will be used for examples and constructs as needed that do not synthesize.

There are two types of code in most HDLs:

Structural, which is a verbal wiring diagram without storage. assign a=b & c; /* "&" is an AND */
assign d = e & (~c);Here the order of the statements does not matter. Changing e will change a.

Procedural which is used for circuits with storage, or as a convenient way to write conditional logic. always @(posedge clk) // Execute the next statement on every rising clock edge. count <= count+1;
Procedural code is written like c code and assumes every assignment is stored in memory until overwritten. For synthesis, with flip-flop storage, this type of thinking generates too much storage. However people prefer procedural code because it is usually much easier to write, for example, if and case statements are only allowed in procedural code. As a result, the synthesizers have been constructed which can recognize certain styles of procedural code as actually combinational. They generate a flip-flop only for left-hand variables which truly need to be stored. However if you stray from this style, beware. Your synthesis will start to fill with superfluous latches.

This manual introduces the basic and most common Verilog behavioral and gate-level modelling constructs, as well as Verilog compiler directives and system functions. Full description of the language can be found in Cadence Verilog-XL Reference Manual and Synopsis HDL Compiler for Verilog Reference Manual. The latter emphasizes only those Verilog constructs that are supported for synthesis by the Synopsis Design Compiler synthesis tool.

In all examples, Verilog keyword are shown in boldface. Comments are shown in italics.
2. Lexical Tokens

Verilog source text files consist of the following lexical tokens:

2.1. White Space

White spaces separate words and can contain spaces, tabs, new-lines and form feeds. Thus a statement can extend over multiple lines without special continuation characters.

2.2. Comments

Comments can be specified in two ways (exactly the same way as in C/C++):
- Begin the comment with double slashes (/). All text between these characters and the end of the line will be ignored by the Verilog compiler.
- Enclose comments between the characters /* and */. Using this method allows you to continue comments on more than one line. This is good for “commenting out” many lines code, or for very brief in-line comments.

Example 2.1

```verilog
a = c + d; // this is a simple comment
/* however, this comment continues on more than one line */
assign y = temp_reg;
assign x = ABC /* plus its compliment*/ + ABC_
```

2.3. Numbers

Number storage is defined as a number of bits, but values can be specified in binary, octal, decimal or hexadecimal (See Sect. 6.1. for details on number notation). Examples are 3’b001, a 3-bit number, 5’d30, (=5’b1110), and 16’hSED4, (=16’d24276)

2.4. Identifiers

Identifiers are user-defined words for variables, function names, module names, block names and instance names. Identifiers begin with a letter or underscore (Not with a number or $) and can include any number of letters, digits and underscores. Identifiers in Verilog are case-sensitive.

Syntax

```verilog
allowed symbols
ABCD . . . abedef . . . 1234567890 _$ not allowed: anything else especially _ & @
```

Example 2.2

```verilog
adder // use underscores to make your
byithuber // identifiers more meaningful
_ABC_ /* is not the same as _abc_
Read_ // is often used for NOT Read
```

2.5. Operators

Operators are one, two and sometimes three characters used to perform operations on variables. Examples include >, * , & and module. Operators are described in detail in “Operators” on p. 6.

2.6. Verilog Keywords

These are words that have special meaning in Verilog. Some examples are assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers. Refer to the manual in this section for a complete listing of Verilog keywords. A number of them will be introduced in this manual. Verilog keywords also includes Compiler Directives (Sect. 15.) and System Tasks and Functions (Sect. 16.).
4. Data Types

4.1. Value Set
Verilog consists of only four basic values. Almost all Verilog data types store all these values:

- 0 (logic zero, or false condition)
- 1 (logic one, or true condition)
- x (unknown logic value) x and z have limited use for synthesis
- z (high impedance state)

4.2. Wire
A wire represents a physical wire in a circuit and is used to connect gates or modules. The value of a wire can be read, but not assigned to, in a function or block. See “Functions” on p. 19, and “Procedures: Always and Initial Blocks” on p. 18. A wire does not store its value but must be driven by a continuous assignment statement or by connecting it to the output of a gate or module. Other specific types of wires include:

- wire (wired-AND); the value of a wire depends on the logical AND of all the drivers connected to it.
- wor (wired-OR); the value of a wor depends on logical OR of all the drivers connected to it.
- tri (three-state); all drivers connected to a tri must be z, except one (which determines the value of the tri).

4.3. Reg
A reg (register) is a data object that holds its value from one procedural assignment to the next. They are used only in functions and procedural blocks. See “Wire” on p. 4 above. A reg is a Verilog variable type and does not necessarily imply a physical register. In multi-bit registers, data is stored as unsigned numbers and no sign extension is done for what the user might have thought were two’s complement numbers.

4.4. Input, Output, Inout
These keywords declare input, output and bidirectional ports of a module or task. Input and inout ports are of type wire. An output port can be configured to be of type wire, reg, or tri. The default is wire.

4.5. Integer
Integers are general-purpose variables. For synthesis they are used mainly loops, indices, parameters, and constants. See “Parameter” on p. 5. They are of implicitly of type reg. However they store data as signed numbers, whereas explicitly declared reg types store them as unsigned. If they hold numbers which are not defined at compile time, their size will default to 32-bits. If they hold constants, the synthesizer adjusts them to the minimum width needed at compilation.

4.6. Supply0, Supply1
Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power), respectively.

4.7. Time
Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

4.8. Parameter
A parameter defines a constant that can be set when you instantiate a module. This allows customization of a module during instantiation. See also “Parameterized Modules” on page 11.
5. Operators

5.1. Arithmetic Operators
These perform arithmetic operations. The + and - can be used as either unary (-z) or binary (x-y) operators.

\[
\text{Example 5.1} \\
\begin{array}{l}
\text{parameter } n = 4; \\
\text{reg}[3:0] a, c, f, g, count; \\
f = a + c; \\
g = c - n; \\
count = (\text{count } + 1)/16; \\
\end{array}
\]

\(\text{Can count 0 thru 15.}\)

5.2. Relational Operators
Relational operators compare two operands and return a single bit 0 or 1. These operators synthesize into comparators.

Wire and reg variables are positive. Thus -3'b001 \(=\) 3'b111 and -3d001 \(>\) 3d110. However for integers -1 < 6.

\[
\text{Example 5.2} \\
\begin{array}{l}
\text{if } (x == y) e = 1; \\
\text{else } e = 0; \\
\end{array}
\]

\(\text{Equivalent Statement} e = (x == y);\)

5.3. Bit-wise Operators
Bit-wise operators do a bit-by-bit comparison between two operands. However see “Reduction Operators” on p. 7.

\[
\text{Example 5.3} \\
\begin{array}{l}
\text{module and2 (a, b, c);} \\
\text{input [1:0] a, b;} \\
\text{output [1:0] c;} \\
\text{assign c = a & b;} \\
\text{endmodule}
\end{array}
\]

5.4. Logical Operators
Logical operators return a single bit 1 or 0. They are the same as bit-wise operators only for single bit operands. They can work on expressions, integers or groups of bits, and treat all values that are nonzero as “1.” Logical operators are typically used in conditional (if...else) statements since they work with expressions.

\[
\text{Example 5.4} \\
\begin{array}{l}
\text{wire}[7:0] x, y, z; \\
\text{x, y and z are multi-bit variables.} \\
\text{reg a;} \\
\text{if (x == y) \&& (z) a = 1; // if x equals y and z is nonzero} \\
\text{else } a = 1; \\
\text{if a = 0 if x is anything but zero.}
\end{array}
\]

5.5. Reduction Operators
Reduction operators operate on all the bits of an operand vector and return a single-bit value. These are the unary (one argument) form of the bit-wise operators above.

\[
\text{Example 5.5} \\
\begin{array}{l}
\text{module chk_zero (a, z);} \\
\text{input [2:0] a;} \\
\text{output z;} \\
\text{assign z = \sim a; // Reduction NOR} \\
\text{endmodule}
\end{array}
\]

5.6. Shift Operators
Shift operators shift the first operand by the number of bits specified by the second operand. Vacated positions are filled with zeros for both left and right shifts (There is no sign extension).

\[
\text{Example 5.6} \\
\text{assign c = a << 2; // c = a shifted left 2 bits; vacant positions are filled with 0's *)}
\]

5.7. Concatenation Operator
The concatenation operator combines two or more operands to form a larger vector.

\[
\text{Example 5.7} \\
\begin{array}{l}
\text{wire [1:0] a, b;} \\
\text{wire [2:0] x;} \\
\text{wire [3:0] y, Z;} \\
\text{assign x = {1'b0, a}; // x[2]=0, x[1]=a[1], x[0]=a[0]} \\
\text{assign y = {a, b}; // y[3]=a[1], y[2]=a[0], y[1]=b[1], y[0]=b[0]} \\
\text{assign {cout, y} = x + Z; // Concatenation of a result}
\end{array}
\]

5.8. Replication Operator
The replication operator makes multiple copies of an item.

\[
\text{Example 5.8} \\
\begin{array}{l}
\text{wire [1:0] a, b;} \\
\text{wire [4:0] x;} \\
\text{wire [3:0] y, Z;} \\
\text{assign x = {2(1'b0), a}; // Equivalent to x = (0,0,a) } \\
\text{assign y = {2(a), 3(b)}; // Equivalent to y = (a,a,b,b)}
\end{array}
\]

For synthesis, Synopsis did not like a zero replication. For example:-

\[
\text{Example 5.8} \\
\begin{array}{l}
\text{wire [1:0] a, b;} \\
\text{wire [4:0] x;} \\
\text{assign x = {2(1'b0), a}; // Equivalent to x = (0,0,a) } \\
\text{assign y = {2(a), 3(b)}; // Equivalent to y = (a,a,b,b)}
\end{array}
\]

\[
\text{assign x = ((m-n)(a))}
\]

5.9. Operators

Symbol
Description

\(+\) (addition)

\(-\) (subtraction)

\(*\) (multiplication)

\/% (division)

\& (reduction AND)

\| (reduction OR)

\~& (reduction NAND)

\~| (reduction NOR)

\^ (reduction XOR)

\~^ (reduction XNOR)

\% (modulus)

\(<\) (less than)

\<=(\) (less than or equal to)

\(>\) (greater than)

\=>(\) (greater than or equal to)

\(!\) (logical NOT)

\&\& (logical AND)

\|| (logical OR)
5.9. Conditional Operator: “?”

Conditional operator is like those in C/C++. They evaluate one of the two expressions based on a condition. It will synthesize to a multiplexer (MUX).

Example 5.9
assign a = (g) ? x : y;
assign a = (inc = = 2) ? a+1 : a-1;

* If (true), a = a+1, else a = a-1 *]

5.10. Operator Precedence

Table 5.1 shows the precedence of operators from highest to lowest. Operators on the same level evaluate from left to right. It is strongly recommended to use parentheses to define order of precedence and improve the readability of your code.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>bit-select or part-select</td>
</tr>
<tr>
<td>( )</td>
<td>parenthesis</td>
</tr>
<tr>
<td>!, ~</td>
<td>logical and bit-wise NOT</td>
</tr>
<tr>
<td>&amp;</td>
<td>reduction AND, OR, NAND, NOR, XOR, XNOR;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+, -</td>
<td>unary (sign) plus, minus; +17, -7</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>* ,</td>
<td>multiply, divide, modulus; * and % not be supported for synthesis</td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>&lt;, =</td>
<td>binary add, subtract.</td>
</tr>
<tr>
<td>&lt;=, &gt;, &gt;</td>
<td>comparisons. Reg and wire variables are taken as positive numbers.</td>
</tr>
<tr>
<td>= ,</td>
<td>logical equality, logical inequality</td>
</tr>
<tr>
<td>== , !=</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise AND, AND together all the bits in a word</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise XOR, bit-wise XNOR</td>
</tr>
<tr>
<td>[ ]</td>
<td>concatenation; [3’B101] + 6’B111110;</td>
</tr>
<tr>
<td>[ ] [ ]</td>
<td>replication; (3[3’B101]) = 9’B110110110</td>
</tr>
<tr>
<td>* ,</td>
<td>multiply, divide, modulus; * and % not be supported for synthesis</td>
</tr>
</tbody>
</table>

Table 5.1: Verilog Operators Precedence

6. Operands

6.1. Literals

Literals are constant-valued operands that can be used in Verilog expressions. The two common Verilog literals are:

(a) String: A string literal is a one-dimensional array of characters enclosed in double quotes (“ ”).

(b) Numeric: constant numbers specified in binary, octal, decimal or hexadecimal.

6.2. Wires, Regs, and Parameters

Wires, regs and parameters can also be used as operands in Verilog expressions. These data objects are described in more detail in Sect. 4.5.

6.3. Bit-Selects “x[3]” and Part-Selects “x[5:3]”

Bit-selects and part-selects are a selection of a single bit and a group of bits, respectively, from a wire, reg or parameter vector using square brackets “[ ]”. Bit-selects and part-selects can be used as operands in expressions in much the same way that their parent data objects are used.

6.4. Function Calls

The return value of a function can be used directly in an expression without first assigning it to a register or wire variable. Simply place the function call as one of the operands. Make sure you know the bit width of the return value of the function call. Construction of functions is described in “Functions” on page 19.
7. Modules

7.1. Module Declaration
A module is the principal design entity in Verilog. The first line of a module declaration specifies the name and port list (arguments). The next few lines specify the i/o type (input, output, or inout, see Sect. 4.4 ) and width of each port. The default port width is 1 bit.

Then the port variables must be declared wire, wand, . . . , reg (See Sect. 4. ) . The default is wire. Typically inputs are wire since their data is latched outside the module. Outputs are type reg if their signals were stored inside an always or initial block (See Sect. 10. ).

Syntax

module module_name (port_list);
  input [msb:lsb] input_port_list;
  output [msb:lsb] output_port_list;
  inout [msb:lsb] inout_port_list;
  . . . statements . . .
endmodule

Example 7.1

module add_sub (add, in1, in2, oot);
  input add; // defaults to wire
  input [7:0] in1, in2;
  output [7:0] oot;
endmodule

7.2. Continuous Assignment
The continuous assignment is used to assign a value onto a wire in a module. It is the normal assignment outside of always or initial blocks (See Sect. 10. ). Continuous assignment is done with an explicit assign statement or by assigning a value to a wire during its declaration. Note that continuous assignment statements are concurrent and do not matter. Any change in any of the right-hand-side inputs will immediately change a left-hand-side output.

Syntax

wire wire_variable = value;
assign wire_variable = expression;

Example 7.2

wire [1:0] a = 2’b01;  // assigned on declaration
assign b = c & d;  // using assign statement
assign d = 3’y;
  /* The order of the assignment statements does not matter. */

7.3. Module Instantiations
Module declarations are templates from which one creates actual objects (instantiations). Modules are instantiated inside other modules, and each instantiation creates a unique object from the template. The exception is the top-level module which is its own instantiation.

The instantiated module’s ports must be matched to those defined in the template. This is specified:
(i) by name, using a dot. ’ .template_port_name (name_of_wire_connected_to_port)’ .
or(ii) by position, placing the ports in exactly the same positions in the port lists of both the template and the instance.

Syntax for Instantiation

module_name
  instance_name_1 (port_connection_list),
  instance_name_2 (port_connection_list),
  . . .
  instance_name_n (port_connection_list);

Example 7.3

//MODULE INSTANTIATIONS
wire [3:0] in1, in2;
wire [3:0] c, d;
module and4 (a, b, c);
  input [3:0] a, b;
  output [3:0] c;
endmodule

//MODULE INSTANTIATION
and4 C1 (in1, in2, o1);
and4 C2 (c, d);

7.4. Parameterized Modules
You can build modules that are parameterized and specify the value of the parameter at each instantiation of the module. See “Parameter” on page 5 for the use of parameters inside a module. Primitive gates have parameters which have been predefined as delays. See “Basic Gates” on page 3.

Syntax

module_name #(parameter_values)
  instance_name(port_connection_list);

Example 7.4

// MODULE DEFINITION
module shift_n (it, ot);
  input [7:0] it;
  output [7:0] ot;
  parameter n = 2;  // default value of n is 2
endmodule

// PARAMETERIZED INSTANTIATIONS
wire [7:0] in1, in2, in3;
shift_n shift_2 (in1, ot1),  // used in module test_shift
  output [7:0] ot;
  parameter n = 2;  // default value of n is 2
  assign x = (it << n);
endmodule

// PARAMETERIZED INSTANTIATIONS
shift_n #(3) shift_3 (in1, ot2),  // shift by 3; override parameter 2.
  output [7:0] ot;
  parameter n = 3;  // default value of n is 3
endmodule

// PARAMETERIZED INSTANTIATIONS
shift_n #(5) shift_5 (in1, ot3),  // shift by 5; override parameter 2.
  output [7:0] ot;
  parameter n = 5;  // default value of n is 5
endmodule

Synthesis does not support the defparam keyword which is an alternate way of changing parameters.
8. Behavioral Modeling

Verilog has four levels of modeling:
1) The switch level which includes MOS transistors modeled as switches. This is not discussed here.
2) The gate level. See "Gate-Level Modeling" on p. 3
3) The Data-Flow level. See Example 7.4 on page 11
4) The Behavioral or procedural level described below.

Verilog procedural statements are used to model a design at a higher level of abstraction than the other levels. They provide powerful ways of doing complex designs. However small changes in coding methods can cause large changes in the hardware generated. Procedural statements can only be used in procedures. Verilog procedures are described later in "Procedures: Always and Initial Blocks" on page 18, "Functions" on page 19, and "Tasks, Not Synthesizable" on page 21.

8.1. Procedural Assignments

Procedural assignments are statement assignments used within Verilog procedures (always and initial blocks). Only reg variables and integers (and their bit/part-selects and concatenations) can be placed left of the '=' in procedures. The right hand side of the assignment is an expression which may use any of the operator types described in Sect. 5.

8.2. Delay in Assignment (not for synthesis)

In a delayed assignment Δt time units pass between the statement is executed and the left-hand assignment is made. With intra-assignment delay, the right side is evaluated immediately but there is a delay of Δt before the result is placed in the left hand assignment. If another procedure changes a right-hand side signal during Δt, it does not affect the output. Delays are not supported by synthesis tools.

Syntax for Procedures

Delayed assignment

Example 8.1

```verilog
reg [6:0] sum; reg h, ziltch;
ziltch = #15 ckz&h; /* ckz&h evaluated now; ziltch changed after 15 time units. */
#10 hat = b&c; /* 10 units after ziltch changes, b&c is evaluated and hat changes. */
```
8.6. for Loops

Similar to for loops in C/C++, they are used to repeatedly execute a statement or block of statements. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```verilog
define for (count = value1; count < value2; count = count +/- step)
begin
... statements ...
end
```

8.7. while Loops

The while loop repeatedly executes a statement or block of statements until the expression in the while statement evaluates to false. To avoid combinational feedback during synthesis, a while loop must be broken with an @posedge/negedge clock statement (Section 9.2). For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```verilog
define while (expression)
begin
... statements ...
end
```

8.8. forever Loops

The forever statement executes an infinite loop of a statement or block of statements. To avoid combinational feedback during synthesis, a forever loop must be broken with an @posedge/negedge clock statement (Section 9.2). For simulation a delay inside the loop will suffice. If the loop contains only one statement, the begin ... end statements may be omitted.

Syntax

```verilog
define forever
begin
... statements ...
end
```

8.9. repeat Not Synthesizable

The repeat statement executes a statement or block of statements a fixed number of times.

Syntax

```verilog
define repeat (number_of_times)
begin
... statements ...
end
```

8.10. disable

Execution of a disable statement terminates a block and passes control to the next statement after the block. It is like the C break statement except it can terminate any loop, not just the one in which it appears. Disable statements can only be used with named blocks.

Syntax

```verilog
define disable block_name;
```

8.11. if ... else if ... else

The if ... else if ... else statements execute a statement or block of statements depending on the result of the expression following the if. If the conditional expressions in all the if’s evaluate to false, then the statements in the else block, if present, are executed. There can be as many else if statements as required, but only one if block and one else block. If there is one statement in a block, then the begin ... end statements may be omitted. Both the else if and else statements are optional. However if all possibilities are not specifically covered, synthesis will generate extra latches.

Syntax

```verilog
define if (expression)
begin
... statements ...
end
else if (expression)
begin
... statements ...
end
else
begin
... statements ...
end
```

Example 8.11

```verilog
if (alu_func == 2'b00)
begin
aluout = a + b;
end
else if (alu_func == 2'b01)
aluout = a - b;
else if (alu_func == 2'b10)
aluout = a & b;
else // alu_func == 2'b11
aluout = a | b;
end
if (a == b) // This if with no else will generate
begin
a = b;
end
else // a latch for x and ot. This is so they
x = 1; // will hold there old value if (a != b).
ot = 4'b1111;
end
```
8.12. case
The case statement allows a multipath branch based on comparing the expression with a list of case choices. Statements in the default block executes when none of the case choices are true (similar to the else block in the if ... else if ... else). If no comparisons, including default, are true, synthesizers will generate unwanted latches. Good practice says to make a habit of putting in a default whether you need it or not. If the defaults are don’t cares, define them as ‘x’ and the logic minimizer will treat them as don’t cares. Case choices may be a simple constant or expression, or a comma-separated list of same.

Syntax
```verilog
case (expression)
    case_choice1:
        begin
            ... statements ...
        end
    case_choice2:
        begin
            ... statements ...
        end
... more case choices blocks ...
    default:
        begin
            ... statements ...
        end
endcase
```
Example 8.12
```verilog
case (alu_ctr)
    2'b00: aluout = a + b;
    2'b01: aluout = a - b;
    2'b10: aluout = a & b;
    default: aluout = 1'b0; // Treated as don’t cares for minimum logic generation.
endcase
```

Example 8.13
```verilog
case (x, y, z)
    2'b00: aluout = a + b; //case if x or y or z is 2'b00. 
    2'b01: aluout = a - b; 
    2'b10: aluout = a & b; 
    default: aluout = 1'b1; // Treated as don’t cares for minimum logic generation.
endcase
```

8.13. casex
In casex(a), the case choices constant “a” may contain z, x or or which are used as don’t cares for comparison. With case the corresponding simulation variable would have to match a tri-state, unknown, or either signal. In short, case uses x as a don’t care which can be used to minimize logic.

Syntax
```verilog
example as for case statement (Section 8.10)
```
Example 8.12
```verilog
casex (a)
    2'b1x: msb = 1; // msb = 1 if a = 10 or a = 11
    // If this were case (a) then only a = 1x would match.
    default: msb = 0;
endcase
```

Example 8.13
```verilog
casex (d)
    3'b17: b = 2'b11; // b = 11 if d = 100 or greater
    3'b011: b = 2'b10; // b = 10 if d = 010 or 011
    default: b = 2'b00;
endcase
```

8.14. casez
Case z is the same as case except only ? and x (not z) are used in the case choice constants as don’t cares. Casez is favored over case since in simulation, an inadvertent x signal, will not be matched by a 0 or 1 in the case choice.

Syntax
```verilog
example as for case statement (Section 8.10)
```
Example 8.12
```verilog
casez (a)
    2'b1x: msb = 1; // msb = 1 if a = 10 or a = 11
    // If this were case (a) then only a = 1x would match.
    default: msb = 0;
endcase
```

8. Timing Controls
9.1. Delay Control, Not Synthesizable
This specifies the delay time units before a statement is executed during simulation. A delay time of zero can also be specified to force the statement to the end of the list of statements to be evaluated at the current simulation time. See also “Intra-Assignment Delay, Not synthesizable” on p. 17

Syntax
```verilog
#delay statement;
```
Example 9.1
```verilog
#5 a = b + c; // evaluated and assigned after 5 time units
#0 a = b + c; // very last statement to be evaluated
```

9.2. Event Control, @
This causes a statement or begin-end block to be executed only after specified events occur. An event is a change in a variable, and the change may be: a positive edge, a negative edge, or either (a level change), and is specified by the keyword posedge, negedge, or no keyword respectively. Several events can be combined with the or keyword. Event specification begins with the character # and are usually used in always statements. See page 18.

For synthesis one cannot combine level and edge changes in the same list. For flip-flop and register synthesis the standard list contains only a clock and an optional reset. For synthesis to give combinational logic, the list must specify only level changes and must contain all the variables appearing in the right-hand-side of statements in the block.

Syntax
```verilog
@ (posedge variable or negedge variable) statement;
@ (variable or variable . . .) statement;
```
Example 9.2
```verilog
always @ (posedge clk or negedge rst) if (rst) Q=0; else Q=D; // Definition for a D flip-flop.
@ (a or b or c); // re-evaluate if a or b or c changes.
sum = a + b + c; // Will synthesize to a combinational adder
```

9.3. Wait Statement Not Synthesizable
Delay executing the statement(s) following the wait until the specified condition evaluates to true.

Syntax
```verilog
wait (condition_expression) statement;
```
Example 9.3
```verilog
wait (!c) a = b; // wait until c=0, then assign b to a
```

9.4. Intra-Assignment Delay, Not Synthesizable
This delay #s is placed after the equal sign. The left-hand assignment is delayed by the specified time units, but the right-hand side of the assignment is evaluated before the delay instead of after the delay. This is important when a variable may be changed in a concurrent procedure. See also “Delay in Assignment (not for synthesis)” on page 12.

Syntax
```verilog
variable = #delay expression;
```
Example 9.4
```verilog
assign a=1; assign b=0;
always @(posedge clk) b = #5 a; // a = b after 5 time units.
always @(posedge clk) c = #5 b; // b was grabbed in this parallel procedure before the first procedure changed it. *
10. Procedures: Always and Initial Blocks

10.1. Always Block

The always block is the primary construct in RTL modeling. Like the continuous assignment, it is a concurrent statement that is continuously executed during simulation. This also means that all always blocks in a module execute simultaneously. This is very unlike conventional programming languages, in which all statements execute sequentially. The always block can be used to imply latches, flip-flops or combinational logic. If the statements in the always block are enclosed within begin ... end, the statements are executed sequentially. If enclosed within the fork ... join, they are executed concurrently (simulation only).

The always block is triggered to execute by the level, positive edge or negative edge of one or more signals (separate signals by the keyword or). A double-edge trigger is implied if you include a signal in the event list of the always statement. The single edge-triggers are specified by posedge and negedge keywords.

Procedures can be named. In simulation one can disable named blocks. For synthesis it is mainly used as a comment.

**Syntax 1**

```verilog
always @(event_1 or event_2 or ...) begin ... statements ... end
```

**Example 10.1**

```verilog
always @(a or b) // level-triggered; if a or b changes levels 
always @(posedge clk); // edge-triggered: on +ve edge of clk 

see previous sections for complete examples
```

**Syntax 2**

```verilog
always @(event_1 or event_2 or ...) begin: name_for_block ... statements ... end
```

10.2. Initial Block

The initial block is like the always block except that it is executed only once at the beginning of the simulation. It is typically used to initialize variables and specify signal waveforms during simulation. Initial blocks are not supported for synthesis.

**Syntax**

```verilog
initial begin ... statements ... end
```

**Example 10.2**

```verilog
initial begin 
  clk = 0; // variables initialized at 
  clk = 1; // beginning of the simulation 
end 

initial // specify simulation waveforms
begin 
a = 2'bx00; // at time = 0, a = 00
u0 a = 2'bx11; // at time = 50, a = 01
u0 a = 2'bx10; // at time = 100, a = 10
end
```

11. Functions

Functions are declared within a module, and can be called from continuous assignments, always blocks, or other functions. In a continuous assignment, they are evaluated when any of its declared inputs change. In a procedure, they are evaluated when invoked. Functions describe combinational logic, and by do not generate latches. Thus an if without an else will simulate as though it had a latch but synthesize without one. This is a particularly bad case of synthesis not following the simulation. It is a good idea to code functions so they would not generate latches if the code were used in a procedure. Functions are a good way to reuse procedural code, since modules cannot be invoked from within a procedure.

11.1. Function Declaration

A function declaration specifies the name of the function, the width of the function return value, the function input arguments, the variables (reg) used within the function, and the function local parameters and integers.

**Syntax, Function Declaration**

```verilog
function [msb:lsb] function_name; 
input [msb:lsb] input_arguments; 
reg [msb:lsb] reg_variable_list; 
parameter [msb:lsb] parameter_list; 
integer [msb:lsb] integer_list; 
... statements ...
endfunction
```

**Example 11.1**

```verilog
function [7:0] my_func; // function return 8-bit value
input [7:0] i;
reg [4:0] temp;
integer n;
temp= i[7:4] | (i[3:0]);
my_func = {temp, i[[1:0]]};
endfunction
```

11.2. Function Return Value

When you declare a function, a variable is also implicitly declared with the same name as the function name, and with the width specified for the function name (The default width is 1-bit). This variable is “my_func” in Example 11.1 on page 19. At least one statement in the function must assign the function return value to this variable.

11.3. Function Call

As mentioned in Sect. 6.4., a function call is an operand in an expression. A function call must specify in its terminal list all the input parameters.

11.4. Function Rules

The following are some of the general rules for functions:
- Functions must contain at least one input argument.
- Functions cannot contain an inout or output declaration.
- Functions cannot contain time controlled statements (4, 8, wait).
- Functions cannot enable tasks.
- Functions must contain a statement that assigns the return value to the implicit function name register.
11.5. Function Example

A Function has only one output. If more than one return value is required, the outputs should be concatenated into one vector before assigning it to the function name. The calling module program can then extract (unbundle) the individual outputs from the concatenated form. Example 11.2 shows how this is done, and also illustrates the general use and syntax of functions in Verilog modeling.

**Syntax**

```plaintext
function_name = expression
```

**Example 11.2**

```verilog
module simple_processor (instruction, outp);
  input [31:0] instruction;
  output [7:0] outp;
  reg [7:0] outp; // so it can be assigned in always block
  reg func;
  reg [7:0] opr1, opr2;

  function [16:0] decode_add (instr) // returns 1 1-bit plus 2 8-bits
    input [31:0] instr;
    reg add_func;
    reg [7:0] opcode, opr1, opr2;
    begin
      opcode = instr[31:24];
      opr1 = instr[7:0];
      case (opcode)
        8'b10001000: begin
          add_func = 1;
          opr2 = instr[15:8];
        end
        8'b10001001: begin
          add_func = 0;
          opr2 = instr[15:8];
        end
        8'b10001010: begin
          add_func = 1;
          opr2 = 8'b00000001;
        end
        default: begin
          add_func = 0;
          opr2 = 8'b00000001;
        end
      endcase
      decode_add = {add_func, opr2, opr1}; // concatenated into 17-bits
    endfunction

  {func, op2, op1} = decode_add (instruction); // outputs unbundled
  if (func == 1)
    outp = op1 + op2;
  else
    outp = op1 - op2;
endmodule
```

12. Tasks, Not Synthesizable

A task is similar to a function, but unlike a function it has both input and output ports. Therefore tasks do not return values. Tasks are similar to procedures in most programming languages. The syntax and statements allowed in tasks are those specified for functions (Sections 11).

**Syntax**

```verilog
task task_name;
  input [msb:lsb] input_port_list;
  output [msb:lsb] output_port_list;
  reg [msb:lsb] reg_variable_list;
  parameter [msb:lsb] parameter_list;
  integer [msb:lsb] integer_list;
  ... statements ...
endtask
```

**Example 12.1**

```verilog
module alu (func, a, b, c);
  input [1:0] func;
  input [3:0] a, b;
  output [3:0] c;
  reg [3:0] c; // so it can be assigned in always block
  task my_and;
    input [3:0] a, b;
    output [3:0] andout;
    integer i;
    begin
      for (i=3; i >= 0; i = i-1)
        andout[i] = a[i] & b[i];
    end
endtask

always @(func or a or b) begin
  case (func)
    2'b00: my_and (a, b, c);
    2'b01: c = a | b;
    2'b10: c = a - b;
    default: c = a + b;
  endcase
end
endmodule
```
13. Component Inference

13.1. Latches
A latch is inferred (put into the synthesized circuit) if a variable, or one of its bits, is not assigned in all branch of an if statement. A latch is also inferred in a case statement if a variable is assigned to in only some of the branches.

To improve code readability, use the if statement to synthesize a latch because it is difficult to explicitly specify the latch enable signal using a case statement.

While in theory, a proper reset should be inferred from the Verilog code shown, Synopsys will not do a proper job without adding the //Synopsys comments shown.

13.2. Edge-Triggered Registers, Flip-flops, Counters
A register (flip-flop) is inferred by using posedge or negedge clause for the clock in the event list of an always block. To add an asynchronous reset, include a second posedge/negedge for the reset and use the if (reset) ... else statement. Note that when you use the negedge for the reset (active low reset), the if condition is (!reset).

13.3. Multiplexers
A multiplexer is inferred by assigning a variable to different variables/values in each branch of an if or case statement. You can avoid specifying each and every possible branch by using the else and default branches. Note that a latch ... the possible branch conditions. To improve readability of your code, use the case statement to model large multiplexers.

13.4. Adders/Subtracters
The +/- operators infer an adder/subtractor whose width depend on the width of the larger operand.

13.5. Tri-State Buffers
A tristate buffer is inferred if a variable is conditionally assigned a value of z using an if, case or conditional operator.

13.6. Other Component Inferences
Most logic gates are inferred by the use of their corresponding operators. Alternatively a gate or component may be explicitly instantiated by using the primitive gates (and, or, nor, inv ...) provided in the Verilog language.
14. Finite State Machines. For synthesis

Introduction to Verilog

When modeling finite state machines, it is recommended to separate the sequential current-state logic from the combinational next-state and output logic.

State Diagram

For lack of space the outputs are not shown on the state diagram, but are:
in state0: Zot = 000,
in state1: Zot = 101,
in state2: Zot = 111,
in state3: Zot = 001.

Using Macros for state definition

As an alternative for parameter

\[ \text{parameter} \ state0=0, \ state1=1, \ state2=2, \ state3=3; \]

one can use macros. For example after the definition below \texttt{2'd0} will be textually substituted whenever \texttt{`state0} is used.

\[ \text{`define state0 \ 2'd0} \]
\[ \text{`define state1 \ 2'd1} \]
\[ \text{`define state2 \ 2'd2} \]
\[ \text{`define state3 \ 2'd3}; \]

When using macro definitions one must put a back quote in front. For example:

\[ \text{case (state)} \]
\[ \text{`state0: Zot = 3'b000; } \]
\[ \text{`state1: Zot = 3'b101; } \]
\[ \text{`state2: Zot = 3'b111; } \]
\[ \text{`state3: Zot = 3'b001; } \]

Using Macros for state definition

As an alternative for parameter

\[ \text{parameter} \ state0=0, \ state1=1, \ state2=2, \ state3=3; \]

one can use macros. For example after the definition below \texttt{2'd0} will be textually substituted whenever \texttt{`state0} is used.

\[ \text{`define state0 \ 2'd0} \]
\[ \text{`define state1 \ 2'd1} \]
\[ \text{`define state2 \ 2'd2} \]
\[ \text{`define state3 \ 2'd3}; \]

When using macro definitions one must put a back quote in front. For example:

\[ \text{case (state)} \]
\[ \text{`state0: Zot = 3'b000; } \]
\[ \text{`state1: Zot = 3'b101; } \]
\[ \text{`state2: Zot = 3'b111; } \]
\[ \text{`state3: Zot = 3'b001; } \]

Example 14.1

Module my_fsm (clk, rst, start, skip3, wait3, Zot);
input clk, rst, start, skip3, wait3;
output [2:0] Zot;
reg [2:0] Zot; // Zot is declared reg so that it can be assigned in an always block.
parameter state0=0, state1=1, state2=2, state3=3;
reg [1:0] state, nxt_st;
always @ (state or start or skip3 or wait3)
begin :
next_state_logic // Name of always procedure.
case (state)
state0:
begin
if (start) nxt_st = state1;
else nxt_st = state0;
end
state1:
begin
nxt_st = state2;
end
state2:
begin
if (skip3) nxt_st = state0;
else nxt_st = state3;
end
state3:
begin
if (wait3) nxt_st = state3;
else nxt_st = state0;
end
default: nxt_st = state0;
endcase // default is optional since all 4 cases are end // covered specifically. Good practice says uses it.
always @(posedge clk or posedge rst)
begin :
register_generation
if (rst) state = state0;
else state = nxt_st;
end
always @ (state)
begin :
output_logic
case (state)
state0: Zot = 3'b000;
state1: Zot = 3'b101;
state2: Zot = 3'b111;
state3: Zot = 3'b001;
default: Zot = 3'b000; // default avoids latches endcase
end
endmodule

Example 14.1.1

Example 14.2

Binary Counter

Using toggle flip-flops

\[ \text{reg} \ [3:0] \ \text{count}; \ \text{wire} \ TC; \ // \ Terminal \ count \ (Carry \ out) \]
\[ \text{always @ (posedge clk or posedge rst)} \]
\[ \text{begin} \]
\[ \text{if (rst) count <= 0;} \text{else count <= count+1;} \]
\[ \text{end} \]
\[ \text{assign TC = & count; // See “Reduction Operators” on page 7} \]

Example 14.3

Shift Register

Shift registers are also best done completely in the flip-flop generation code. Use the nonblocking “<=” assignment operator.

\[ \text{reg} \ [3:0] \ \text{Q}; \]
\[ \text{always @ (posedge clk or posedge rst)} \]
\[ \text{begin} \]
\[ \text{if (rst) Q <= 0;} \text{else begin} \]
\[ \text{Q <= Q[3] << 1;} // Left shift 1 position \]
\[ \text{Q[0] <= Q[3]}; /* Nonblocking means the old Q[3] is sent to Q[0]. Not the revised Q[3] from the previous line.} \]
\[ \text{end} \]
\[ \text{end} \]

Example 14.4

Linear-Feedback Shift Register

\[ \text{reg} \ [3:0] \ \text{Q}; \]
\[ \text{always @ (posedge clk or posedge rst)} \]
\[ \text{begin} \]
\[ \text{if (rst) Q <= 0;} \text{else begin} \]
\[ \text{Q <= {Q[2:1]: Q[3] ^ Q[2]}; /* The concatenation operators \"{...}\" form the new Q from elements of the old Q. */} \]
\[ \text{end} \]
\[ \text{end} \]
15. Compiler Directives

Compiler directives are special commands, beginning with '!', that affect the operation of the Verilog simulator. The Synopsys Verilog HDL Compiler/Design Compiler and many other synthesis tools parse and ignore compiler directives, and hence can be included even in synthesizable models. Refer to Cadence Verilog-XL Reference Manual for a complete listing of these directives. A few are briefly described here.

15.1. Time Scale

timescale specifies the time unit and time precision. A time unit of 10 ns means a time expressed as say #2.3 will have a delay of 23.0 ns. Time precision specifies how delay values are to be rounded off during simulation. Valid time units include s, ns, ns (μs), ns, ps, fs. Only 1, 10 or 100 are valid integers for specifying time units or precision. It also determines the displayed time units in display commands like $display

Syntax

timescale time_unit / time_precision;

Example 15.1
	'timescale 1 ns/1 ps // unit = ns; precision=1/1000ns
	'timescale 1 ns / 100 ps // time unit = ns; precision = 1 / 10ns;

15.2. Macro Definitions

A macro is an identifier that represents a string of text. Macros are defined with the directive `define, and are invoked with the quoted macro name as shown in the example.

Syntax

define macro_name text_string;

Example 15.2

define add_lsb a[7:0] + b[7:0];
assign o = add_lsb; // assign o = a[7:0] + b[7:0];

15.3. Include Directive

Include is used to include the contents of a text file at the point in the current file where the include directive is. The include directive is similar to the C/C++ include directive.

Syntax

'include file_name;

Example 15.3

module x;
'include "delrec.v"; // contents of file "delrec.v" are put here

16. System Tasks and Functions

These are tasks and functions that are used to generate input and output during simulation. Their names begin with a dollar sign ($). The Synopsys Verilog HDL Compiler/Design Compiler and many other synthesis tools parse and ignore system functions, and hence can be included even in synthesizable models. Refer to Cadence Verilog-XL Reference Manual for a complete listing of system functions. A few are briefly described here.

System tasks that extract data, like $monitor need to be in an initial or always block.

16.1. Display Selected Variables; $display, $strobe, $monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like waves® or SignalTap®. $display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes. The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly where it is executed. The format string is like in C/C++, and may contain format characters. Format characters include %d (decimal), %h (hexadecimal), %b (binary), %c (character), %s (string) and %t (time), %m (memory level), %5d, %5b etc. would give exactly 5 spaces for the number instead of the space needed. Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

Syntax

$display("format_string", par_1, par_2, ...);
$strobe("format_string", par_1, par_2, ...);
$monitor("format_string", par_1, par_2, ...);

16.2. Stim, $stime, Srealtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively. Their use is illustrated in Examples 4.7. and 15.1.

16.3. Sreset, $stop, $finish

Sreset resets the simulation back to time 0; $stop halts the simulator and puts it in the interactive mode where the user can enter commands; $finish exits the simulator back to the operating system.

16.4. $deposit

$deposit sets a net to a particular value.

Syntax

$deposit (net_name, value);

16.5. $scope, $showscope

$scope(hierarchy_name) sets the current hierarchical scope to hierarchy_name. $showscope(n) lists all modules, tasks and block names in (and below, if n is set to 1) the current scope.

16.6. Slist

Slist (hierarchy_name) lists line-numbered source code of the named module, task, function or named-block.
16.7. $random

$random generates a random integer every time it is called. If the sequence is to be repeatable, the first time one invokes random give it a numerical argument (a seed). Otherwise the seed is derived from the computer clock.

Example 16.3

```verilog
reg [3:0] xz;
initial begin
    xz = $random(7); // Seed the generator so number
    forever xz = $random;
    // 4 lsb bits of the random integers will transfer into the
    // xz. Thus xz will be a random integer 0 ≤ xz ≤ 15.
end
```

16.8. $dumpfile, $dumpvar, $dumpoff, $dumpall

These can dump variable changes to a simulation viewer like cwaves®. The dump files are capable of dumping all the variables in a simulation. This is convenient for debugging, but can be very slow.

Example 16.4

```verilog
module testbench;
    reg a, b; wire c;
    initial begin
        $dumpfile("filename.dmp");
        $dumpvar("cwave_data.dmp");
        $dumpoff;
    end
endmodule
```

16.9. $shm_probe, $shm_open

These are special commands for the Simulation History Manager for Cadence cwaves® only. They will save variable changes for later display.

Example 16.5

```verilog
module testbench;
    reg a, b; wire c;
    initial begin
        $shm_open("cwave_dump.dmp");
        $shm_probe(var1, var2, var3);
        $shm_probe(a, b, c);
    end
```

16.10. Writing to a File; $fopen, $fwrite, $strobe, $monitor and $fwrite

These commands write more selectively to files.

$fopen opens an output file and gives the open file a handle for use by the other commands.

$fclose closes the file and lets other programs access it.

$fstrobe and $fwrite write formatted data to a file whenever they are executed. They are the same except $fstrobe inserts a new line after every execution and $fwrite does not.

$monitor writes to a file whenever any one of its arguments changes.

See "Display Selected Variables; $display, $strobe, $monitor" on page 27 for the meaning of %h, %b etc in the example.

Example 16.6

```verilog
module testbench;
    reg [15:0] a, reg clk; integer hand1;
    initial begin
        hand1=$fopen("filename1.suffix");
        forever @(posedge clk) begin
            $strobe(hand1, "time=\%t, a=\%h, c=\%b");
            $fwrite(hand1, "a=\%h, b=\%h, c=\%b");
            $fclose;
        end
    end
```

Example 16.7

```verilog
// Test Bench
submodule submod1(a, clk);
submodule submod(a, b, c);
```

Example 16.8

```verilog
// Test Bench
topmodule top(a, b, c);
```

These are special commands for the Simulation History Manager for Cadence cwaves® only. They will save variable changes for later display.
17. Test Benches

A test bench supplies the signals and dumps the outputs to simulate a Verilog design (module(s)). It invokes the design under test, generates the simulation input vectors, and implements the system tasks to view/format the results of the simulation. It is never synthesized so it can use all Verilog commands.

To view the waveforms when using Cadence Verilog XL Simulator, use the Cadence-specific Simulation History Manager (SHM) tasks `$shm_open` to open the file to store the waveforms, and `$shm_probe` to specify the variables to be included in the waveforms list. You can then use the Cadence `cwaves` waveform viewer by typing `cwaves &` at the UNIX prompt.

### Syntax

```bash
$shm_open(filename);
$ shm_probe(var1, var2, ...)
```

**Note also**

```bash
var=$random
```

### Example 17.1

```verilog
timescale 1 ns/100 ps // time unit = 1ns; precision = 1/10 ns;
module my_fsm_tb;
// Test Bench of FSM Design of Example 14.1
/* ports of the design under test are variables in the test bench */
reg clk, rst, start, skip3, wait3;
wire Button;
/***
DESIGN TO SIMULATE (my_fsm) INSTANTIATION
*****/
my_fsm dut1 (clk, rst, start, skip3, wait3, Button);
/***
SECTION TO DISPLAY VARIABLES
*****/
initial begin
$shm_open("sim.db"); //Open the SHM database file
$shm_probe(clk, reset, start);
// Use the qualifier dut1. to look at variables inside the instance dut1.
$ shm Probe(skip3, wait3, Button, dut1.state, dut1.nxt_st);
end
/***
RESET AND CLOCK SECTION
*****/
initial begin
clk = 0; rst=0;
#1 rst = 1; // The delay gives rst a posedge for sure.
#200 rst = 0; // Deactivate reset after two clock cycles +1 ns*
end
always @(posedge clk)
    if (I==9)
        $finish; // End simulation
begin:
begin
    #1; // Keeps data from changing on clock edge
    x<=data[I]; I<=I+1;
when synthesizing to flip-flops as in an In an @(posedge... procedure, always use nonblocking. Without that you will be racing with the flip-flops in the other modules.
wait3 = $random;
//Generate a random number, transfer lsb into wait3
$finish; // stop simulation. Without this it will not stop,
end
endmodule
```
18. Memorys

18.1. Two-Dimensional Arrays

Two-dimensional arrays can be declared and accessed by word. To get at a bit(s) one must equate the output to a register or wire and select the bits from this new variable. See Example 18.1

18.1.1 Initializing Memory From a File

The command $readmemb will read a file of binary numbers into the array. The data file consists of addresses and data. An address written in hex as @hhh...indicates the address of the first word in a block of data. It is followed by binary data words separated by blanks. Legal binary bits are "0 1 X x Z z". Data not included in the file will be given xxx... values. The data may be given in noncontiguous blocks if an address proceeds each block. If no initial address is given, @000 is assumed for the first data word. Comments are allowed in data files.

If start_addr is given the memory array will be filled starting at that address and continue until finish_addr (or the end of the array) is reached. One must have start address ≤ @hhh..., the initial address in the file.

The command $readmemh is similar except the data must contain hexadecimal numbers.

Syntax

```verilog
reg [wordsize-1:0] array [0:arraysize];
readmemb("file_name", array_name);
readmemb("file_name", array_name, start_addr);
readmemb("file_name", array_name, start_addr, finish_addr);
readmemh("file_name", array_name);
```

// start_addr and finish_addr are optional

Example 18.1

```verilog
reg [7:0] memry [0:31]; // 32 byte memory.
wire [7:0] memwrd;
wire x;
initial begin
    // Initialize memory contents from file.
    $readmemb("init.dat", memry, 8);
    // words 8 and 9 are not in the file and will default to x.
    end
    // Extract last word in memory.
    assign memwrd= memry[31];
    // Extract most sig bit in word #1
    assign x= memwrd[7];
------------------------------- file init.dat-------------------------------
    // Since start_addr =8 memory[0:9] will all be stored as x000000
    // @00 @01 @02 @03 @04 @05 @06 @07 @08 @09
    @00 10101100 11110000 1x000x11 11110101 01011010 01001100xXxxZzzz 00000000
    @01 1100_1010 00011_0001
```

Thursday, October 11, 2001 9:39 pm 32  Peter M. Nyasulu