

Contents

Appendix D	Verilog Summary	2
D.1	Basic Language Elements	2
D.1.1	Keywords	2
D.1.2	Comments	2
D.1.3	Identifiers	2
D.1.4	Numbers and Strings	3
D.1.5	Defining Constants	3
D.1.6	Data Types	3
D.1.7	Data Operators	4
D.1.8	Module	5
D.1.9	Module Parameter	6
D.2	Behavioral Model	7
D.2.1	always	7
D.2.2	Event Control	7
D.2.3	Assignment	8
D.2.4	begin-end	8
D.2.5	if-then-else	9
D.2.6	case, casex, casez	9
D.2.7	for	10
D.2.8	while	10
D.2.9	function	11
D.2.10	Behavioral Model Sample	11
D.3	Dataflow Model	12
D.3.1	Continuous Assignment	12
D.3.2	Conditional Assignment	12
D.3.3	Dataflow Model Sample	13
D.4	Structural Model	13
D.4.1	Structural Model Sample	14
Index	15

Appendix D Verilog Summary

The Verilog language is a hardware description language (HDL) for modeling digital circuits that can range from the simple connection of gates to complex systems. Verilog was originally designed as a proprietary verification and simulation tool. Later, logic and behavioral synthesis tools were also added. The language was first standardized in 1995 by IEEE, followed by a second revision in 2001. This appendix gives a brief summary of the basic Verilog elements and its syntax. Many advanced features of the language are omitted. Interested readers should refer to other references for detailed coverage.

D.1 Basic Language Elements

D.1.1 Keywords

The Verilog language is case sensitive, and all of the keywords are in lower case. Figure D.1 shows a partial list of the Verilog keywords.

always	and	assign	automatic	begin	buf
bufif0	bufif1	case	casex	casez	default
defparam	else	end	endcase	endfunction	endgenerate
endmodule	endtask	event	for	forever	function
generate	genvar	if	include	inout	input
integer	library	module	nand	negedge	nor
not	notif0	notif1	or	output	parameter
posedge	reg	signed	supply0	supply1	task
tri	tri0	tri1	unsigned	wand	while
wire	wor	xnor	xor	@	

Figure D.1 Partial list of Verilog keywords.

D.1.2 Comments

Single line comments are preceded by two consecutive slashes (//) and are terminated at the end of the line.

Example:

```
// This is a single line comment
```

Multiple line comments begin with the two characters /* and ends with the two characters */.

Example:

```
/* This is a
   multiple line comment
*/
```

D.1.3 Identifiers

Verilog identifiers are user given names. Verilog identifiers must use the following syntax:

- A sequence of one or more uppercase letters, lowercase letters, digits, and the underscore (_)
- Upper and lowercase letters are treated differently (i.e., case sensitive)
- The first character must be a letter or the underscore
- The length of the identifier must be 1024 characters or less

D.1.4 Numbers and Strings

Numbers

Number constants can be specified in any one of the four bases: decimal, hexadecimal, octal, or binary. An unsigned decimal number can also be specified using just the digits from 0 to 9.

Syntax:

a'sfn

where:

a is the number of bits (specified as an unsigned decimal number) of the constant.

s optionally specifies that the value is to be considered as a signed number.

f specifies the base of the number. It is replaced by one of the letters: d (decimal), h (hexadecimal), o (octal), and b (binary).

n is the value of the constant specified in the given base.

Example:

```
48          // an unsigned decimal number 48
4'b 1001   // a 4-bit binary number 1001
8'd 28     // an 8-bit decimal number 28
'o 537     // an unsigned octal number 537
12'h 7e9   // a 12-bit hexadecimal number 7e9
```

Strings

String constants are enclosed within double quotes.

Example:

```
reg [1:8] MyString;
MyString = "This is a string";
```

D.1.5 Defining Constants

Identifiers can be defined with a constant value. Once it is defined, the identifier can then be used in place of the constant. The compiler directive starting with the single opening quote (`), followed by the word **define** is used to define the identifier. When using the identifier, the single opening quote must always precede the identifier name.

Syntax—definition:

`define identifier constant

Syntax—usage:

`identifier

Example:

```
`define buswidth 'd8          // define buswidth to be constant 8
wire [ `buswidth-1:0 ] databus; // using buswidth
```

D.1.6 Data Types

Nets and registers are two main kinds of data types.

- Nets, defined with the **wire** keyword¹, are used to model electrical connections between components. It is used to connect instances together to transmit logic values between them. Nets do not store values and have to be continuously driven. An optional range [start:end] can be given for the bit width.
- Registers, defined with the **reg** keyword, are used to represent storage elements. Registers can store their values from one assignment to the next. An optional range [start:end] can be given for the bit width. Furthermore, the optional **signed** keyword can be used to denote that the data in the register is to be treated as a signed (2's complement) number.

Syntax:

```
wire [range] identifier1, identifier2, ...;
```

```
reg [signed] [range] identifier1, identifier2, ...;
```

Example:

```
wire x, y;           // two 1-bit wire
wire [1:4] bus;     // a 4-bit wire with bit 1 being the most significant
reg z;             // a 1-bit register
reg [7:0] s;       // an 8-bit register with bit 7 being the most significant
```

D.1.7 Data Operators

Some of the more commonly used Verilog operators are listed in Figure D.2.

Logical Operators	Operation	Example
&&	Logical AND	if ((a > b) && (c < d))
	Logical OR	If ((a > b) (c < d))
!	Logical NOT	If !(a > b)
&	Bitwise AND of individual bits	n = a & b
	Bitwise OR of individual bits	n = a b
~	Bitwise NOT of individual bits	n = ~a
^	Bitwise XOR of individual bits	n = a ^ b
Arithmetic Operators	Operation	Example
+	Addition	n = a + b
-	Subtraction	n = a - b
*	Multiplication (integer or floating point)	n = a * b
/	Division (integer or floating point)	n = a / b
%	Modulus; remainder (integer)	n = a % b
**	Power	n = a ** 2
Relational Operators	Operation	Example
==	Logical equal	if (a == b)
!=	Logical not equal	if (a != b)
<	Less than	if (a < b)
<=	Less than or equal	if (a <= b)
>	Greater than	if (a > b)
>=	Greater than or equal	if (a >= b)
===	Bitwise equal. All bits must match. May include x and z values.	if (a === b)
!==	Bitwise not equal. True if only one bit is different. May include x and z values.	if (a !== b)

¹ In addition to the **wire** net type, there are several other net data types defined in Verilog.

Shift and Other Operators	Operation	Example
<<	Logical left shift. Pad with zero	n = 7'b1001010 << 2
>>	Logical right shift. Pad with zero	n = a >> 1
<<<	Arithmetic left shift. Pad with zero	n = a <<< 3
>>>	Arithmetic right shift. Pad with sign bit	n = a >>> 2
{ , }	String concatenation	n = {{3{a}}, b, c}

Figure D.2 Verilog built-in data operators.

D.1.8 Module

In Verilog, a **module** represents a logical component in a digital system. Each module has an interface for specifying the signals for communication with other modules. These port signals are declared within parenthesis, and can be of types **input**, **output**, or **inout** (for bidirectional communication). A module's body contains statements which describe the actual operation of the logical component.

The operational description of the module can be written using one of three different models: behavioral, dataflow, or structural. Behavioral modeling is concerned with describing the abstract operation of the circuit using a high-level construct, and do not take into consideration of how the circuit is actually implemented. Dataflow modeling specifies the circuit in a form that is closely related to a Boolean equation. Structural modeling describes a circuit in terms of how the primitive gates are interconnected together.

Syntax:

```
module module_name
    (input port_name_list,
     output port_name_list,
     inout port_name_list);

    statements;

endmodule
```

Example: Behavioral model

```
// a 1-bit 2-to-1 multiplexer written in behavioral model
module multiplexer
    (input d0, d1, s,    // interface
     output reg y);

    always @(s, d0, d1) begin
        if (~s)
            y = d0;           // assign d0 to y
        else
            y = d1;
        end

endmodule
```

Example: Dataflow model

```
// a 1-bit 2-to-1 multiplexer written in dataflow model
module multiplexer
    (input d0, d1, s,    // interface
     output y);
```

```

assign y = (~s & d0) | (s & d1);

endmodule

```

Example: Structural model

```

// a 1-bit 2-to-1 multiplexer written in structural model
module multiplexer
  (input d0, d1, s, // interface
   output reg y);

  wire sn, snd0, sd1; // define 3 nets for connecting the components

  not U1(sn,s); // an instance of the NOT gate. sn is the output
  and U2(snd0,d0,sn); // an instance of the AND gate. snd0 is the output
  and U3(sd1,d1,s);
  or U4(y,snd0,sd1);

endmodule

```

D.1.9 Module Parameter

A **module** may have an optional parameter list. This list of parameters, with optional default values, allows us to define generic information about the module. The **parameter** keyword is used to specify a list of identifiers with optional default values assigned to them. The identifier is assigned an external value when the module is instantiated, or is assigned the default value when no external value is given. The identifier can then be used in place of the constant.

Syntax—Declaration:

```

module module_name
  #(parameter identifier = default_value, identifier = default_value)
  (input port_name_list,
   output port_name_list,
   inout port_name_list);

  statements;

endmodule

```

Syntax—Instantiation:

```

module_name #(constant) instance_name (parameter_list);

```

Example—Declaration:

```

// a default 8-bit 2-to-1 multiplexer written in dataflow model
module multiplexer
  #(parameter width = 8) // a parameter constant with a default value of 8

  (input [width-1:0] d0, d1,
   input s,
   output [width-1:0] y);

  assign y = (~s) ? d0:d1; // assigns d0 to y if s is 0,
                          // otherwise assigns d1 to y

endmodule

```

Example—Instantiation:

```
// instantiating a 4-bit 2-to-1 multiplexer
multiplexer #(4) U1(input1, input2, select, output);
```

D.2 Behavioral Model

The behavioral model allows statements to be executed sequentially very much like in a regular computer program. An **always** block, containing one or more sequential statements, forms the basis of the behavioral model. The **always** block is like a process with its independent thread of control, and continuously repeats executing the statements that are inside it. All sequential statements, including many of the standard constructs, such as variable assignments, if-then-else, and loops, must be written inside an **always** block.

D.2.1 always

The **always** block contains statements that are executed sequentially. However, the **always** statement itself is a concurrent statement. The **always** block continuously repeats executing the statements that are inside it. A behavioral module may contain multiple **always** blocks, and they all will be executed concurrently.

Syntax:

```
always
  statement;
```

Example:

```
Siren: PROCESS (D, V, M)
BEGIN
  term_1 <= D OR V;
  S <= term_1 AND M;
END PROCESS;
```

An **always** construct is usually used in conjunction with an event control (@) to create either a combinational or sequential logic.

D.2.2 Event Control

The event control statement, which uses the @ symbol, waits for the specified event to occur. As soon as the event occurs, the statement associated with it is executed. The event is specified in the form of a sensitivity list, which is a comma-separated list of nets. Whenever a signal in the sensitivity list changes value, the associated statement will be executed.

Syntax:

```
@ (sensitivity_list)
  statement;
```

If the sensitivity list contains every variable that are in the right-hand side or condition of the statement, then a combinational logic is created. The * symbol can be used as a shorthand notation to denote all of the variables.

Syntax:

```
@ (*)
  statement;
```

Example:

```
// synthesizes to a combinational logic
always @(*) // equivalent to always @(a, b, c)
  if (a == 1)
    x = b;
  else
    x = c;
```

The nets specified in the sensitivity list may be qualified with the keywords **posedge** or **negedge** so that the control statement watches only for the positive or negative transition, respectively, of the given signal before it executes the statement. In this case, a sequential logic is created.

Syntax:

```
@ (posedge signal)
  statement;
```

```
@ (negedge signal)
  statement;
```

Example:

```
// aD flip flop
module dFF
  (output reg q,
   input clock, data);

  always @(posedge clock)
    q <= data; // q gets the value of data at the next rising clock edge
endmodule;
```

D.2.3 Assignment

Variable assignments are performed using the symbol = for blocking, or <= for non-blocking.

Syntax:

```
register_identifier = expression; // blocking (immediate) assignment
register_identifier <= expression; // non-blocking (concurrent) assignment
```

Example:

```
q = q - 1;
zero = 1'b0;
```

D.2.4 begin-end

A block of sequential statements can be grouped together to form a single block with the use of the **begin** and **end** keywords.

Syntax:

```
begin
  statement1;
  statement2;
  ...
end
```

D.2.5 if-then-else

Syntax:

```

if (condition)
    statement1;
else
    statement2;

```

or

```

if (condition)
    statement1;
else if (condition)
    statement2;
else
    statement3;

```

Example:

```

if (count != 10) // not equal
    count = count + 1;
else
    count = 0;

```

D.2.6 case, casex, casez

Syntax:

```

case (expression)
    constant1: statement1;
    constant2: statement2;
    ...
    default: statement3;
endcase

```

The **casex**, and **casez** statements have the same syntax as the **case** statement, except for the replaced keyword. The **casez** statement allows for z values to be treated as don't-care values, while the **casex** statement allows for both x and z values to be treated as don't-cares.

Example:

```

module mux4
    #(parameter width = 8)
    (input [1:0] s,
    input [width-1:0] d3, d2, d1, d0,
    output reg [width-1:0] y);

    always @(s, d0, d1, d2, d3) begin
        case (s)
            2'b00: y = d0;
            2'b01: y = d1;
            2'b10: y = d2;
            2'b11: y = d3;
        endcase
    end

```

```
end  
endmodule
```

D.2.7 for

Syntax:

```
for (identifier = low_range; identifier < high_range; identifier = identifier + step)  
    statement;
```

Example:

```
module TestFOR  
    (sum);  
  
    inout reg [7:0] sum = 'd0;  
    reg [3:0] i;  
  
    always  
    begin  
        for (i = 0; i < 10; i = i + 1)  
        begin  
            sum = sum + i;  
        end  
    end  
  
endmodule
```

D.2.8 while

Syntax:

```
while (condition)  
    statement;
```

Example:

```
module TestWHILE  
    (sum);  
  
    inout reg [7:0] sum = 'd0;  
    reg [3:0] i;  
  
    always  
    begin  
        i = 0;  
        while (i < 10)  
        begin  
            sum = sum + i;  
            i = i + 1;  
        end  
    end  
  
endmodule
```

D.2.9 function

Syntax—Function definition:

```
function function_name (parameter_list);
    // register declarations
    // wire declarations
    begin
    statement;
    ...
    end
endfunction
```

Syntax—Function call:

```
function-name (parameters);
```

Example:

```
module TestFunction
  (input [7:0] bitstring,
   output [7:0] result);

  assign result = Shiftright(bitstring);      // function call

  // function to perform a shift right
  function [7:0] Shiftright
    (input [7:0] string);

    Shiftright = {1'b0,string[7:1]};
  endfunction

endmodule
```

D.2.10 Behavioral Model Sample

```
// BCD to 7-segment decoder written in behavioral code
module decoder
  (input [3:0] I,
   output reg a,b,c,d,e,f,g);

  always @(*) begin
    case(I)
      4'b0000: {a,b,c,d,e,f,g} = 7'b1111110; // 0 (1 = on; 0 = off)
      4'b0001: {a,b,c,d,e,f,g} = 7'b0110000; // 1
      4'b0010: {a,b,c,d,e,f,g} = 7'b1101101; // 2
      4'b0011: {a,b,c,d,e,f,g} = 7'b1111001; // 3
      4'b0100: {a,b,c,d,e,f,g} = 7'b0110011; // 4
      4'b0101: {a,b,c,d,e,f,g} = 7'b1011011; // 5
      4'b0110: {a,b,c,d,e,f,g} = 7'b1011111; // 6
      4'b0111: {a,b,c,d,e,f,g} = 7'b1110000; // 7
      4'b1000: {a,b,c,d,e,f,g} = 7'b1111111; // 8
      4'b1001: {a,b,c,d,e,f,g} = 7'b1110011; // 9
      4'b1010: {a,b,c,d,e,f,g} = 7'b1110111; // A
      4'b1011: {a,b,c,d,e,f,g} = 7'b0011111; // b
      4'b1100: {a,b,c,d,e,f,g} = 7'b1001110; // C
    endcase
  end
```

```

4'b1101: {a,b,c,d,e,f,g} = 7'b0111101;    // d
4'b1110: {a,b,c,d,e,f,g} = 7'b1001111;    // E
4'b1111: {a,b,c,d,e,f,g} = 7'b1000111;    // F
default: {a,b,c,d,e,f,g} = 7'b0000000;    // all off
endcase
end
endmodule

```

D.3 Dataflow Model

The dataflow model specifies the circuit in a form similar to Boolean algebra. Hence, this model is best suited for describing a circuit when given a set of Boolean equations.

D.3.1 Continuous Assignment

The **assign** statement is used to provide continuous assignment of values onto nets. The **assign** statement is evaluated anytime when any of its inputs changes, and the result of the evaluation is propagated to the output net.

Syntax:

```

assign net_identifier = expression;          // blocking (immediate) assignment
assign net_identifier <= expression;        // non-blocking (concurrent) assignment
assign net_identifier = function;

```

The expression on the right-hand side of the first two **assign** statements can be either a logical or arithmetic expression. It is evaluated anytime when any of its inputs changes. The result of the expression is assigned to the net on the left-hand side of the statement.

For the blocking assignment (=) statement, the assignment takes effect immediately and the value written to the register on the left-hand side of the = sign is available for use in the next statement.

However, the non-blocking assignment (<=) statements are executed concurrently. In other words, all of the right-hand side expressions in these non-blocking assignment statements will be evaluated *before* any of the left-hand side registers are updated. Hence, the ordering of these statements does not affect the resulting output, and all of the assignments will be synchronized so that they all appear to happen at the same time. Usually, the non-blocking assignment statements are used in **always** statements with an edge specification (i.e., **posedge** or **negedge**).

The function on the right-hand side of the last **assign** statement may contain procedural statements only if they describe a combinational logic function. These procedural statements may be case and loop statements, but not wait, or control event (@) statements. Wait and @event statements will produce a sequential logic function.

Example:

```

wire [7:0] y;
wire siren, master, door, vibration;
assign y = 8'b11111111; // assigns 8 bits of 1's to y
assign siren = master & (door | vibration);

```

D.3.2 Conditional Assignment

The conditional signal assignment statement selects one of two different values to assign to a net. This statement is executed whenever an input in any one of the expressions or condition changes.

Syntax:

```

assign net_identifier = (condition) ? expression1:expression2;

```

If the condition is true, then the result of expression1 is assigned to the net, otherwise the result of expression2 is assigned to the net.

Example:

```
assign out = (enable) ? in:1'bz;    // assigns in to out if enable is true
                                // otherwise assigns a z to out
```

D.3.3 Dataflow Model Sample

This example describes a full adder circuit using the dataflow model. Recall that the Boolean equations for describing the full adder circuit are:

$$c_{out} = xy + c_{in}(x \oplus y)$$

$$sum = x \oplus y \oplus c_{in}$$

The following code for the full adder circuit translates the above two equations into the corresponding two **assign** statements.

```
// Dataflow code for a full adder
module fa
  (input x,y,cin,
   output cout, sum);

  assign cout = (x & y) | (cin & (x ^ y));
  assign sum = x ^ y ^ cin;
endmodule
```

D.4 Structural Model

The structural model allows the manual connection of primitive gates and module components together using nets to build larger modules.

Syntax:

```
and instance_name (parameter_list); // implements the primitive logic function
nand instance_name (parameter_list);
nor instance_name (parameter_list);
or instance_name (parameter_list);
xor instance_name (parameter_list);
xnor instance_name (parameter_list);
buf instance_name (output, input); // implements the non-inverting buffer
not instance_name (output, input); // implements the inverter
bufif0 instance_name (output, input, enableN); // implements the tri-state buffer with active-low enable
bufif1 instance_name (output, input, enable); // implements the tri-state buffer with active-high enable
notif0 instance_name (output, input, enableN); // implements the tri-state inverter with active-low enable
notif1 instance_name (output, input, enable); // implements the tri-state inverter with active-high enable

user_defined_module_name instance_name (parameter_list); // a user defined module
```

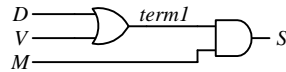
The syntax for using a gate or a user defined module is the same. Each instance of a gate or module can have an optional instance name. The parameter list consists of comma separated input and output signals. All of the input/output signals are nets of type **wire**. For the predefined primitive gates, the first parameter specified is always the output signal, followed by as many input nets as needed. For the user defined module, it depends on how the port list in the module definition is specified.

Example:

```
and    U1(out, in1, in2, in3); // a 3-input AND gate
bufif1 U2(out, in, enable); // a tri-state buffer with active-high enable
fa     U3(cout, sum, x, y, cin); // an instance of the full adder
```

D.4.1 Structural Model Sample

This example is based on the following circuit:



```
module siren
  (input D, V, M,
   output S);

  wire term1;

  or    U1(term1,D,V);
  and   U2(S,term1,M);
endmodule
```

Index

- @**
- @. *See* Event control
- {**
- { }. *See* String concatenation
- A**
- always, 7
- and, 13
- assign, 12
 - blocking, 12
 - conditional, 12
 - continuous, 12
 - non-blocking, 12
- Assignment, 8
- B**
- begin, 8
- Behavioral model, 5, 7
 - example, 11
- buf, 13
- bufif0, 13
- bufif1, 13
- C**
- case, 9
- casex, 9
- casez, 9
- Comments, 2
- Concatenation, 5
- Conditional assignment, 12
- Continuous assignment, 12
- D**
- Data operators, 4
- Data strings, 3
- Data types, 3
- Dataflow model, 5, 12
 - example, 13
- default, 9
- E**
- end, 8
- endcase, 9
- endmodule, 3, 5, 6
- Event control, 7
- F**
- for, 10
- function, 10
- I**
- Identifiers, 2
- if, 9
- inout, 3, 5, 6
- input, 3, 5, 6
- M**
- module, 5, 6
 - parameter, 6
- Multiplexer, 3, 5, 6, 7
- N**
- nand, 13
- negedge, 8
- nor, 13
- not, 13
- notif0, 13
- notif1, 13
- Number, 3
- O**
- or, 13
- output, 3, 5, 6
- P**
- posedge, 8
- Primitive gates, 13
- S**
- String, 3
 - concatenation, 5
- Structural model, 6, 13
 - example, 14
- V**
- Verilog
 - Basic language elements, 2
 - Behavioral model, 5, 7
 - Dataflow model, 5, 12
 - Structural model, 6, 13
- Verilog syntax
 - @. *See* Event control
 - { }. *See* String concatenation
 - always, 7

- and, 13
- assign, 12
- begin, 8
- buf, 13
- bufif0, 13
- bufif1, 13
- case, 9
- casex, 9
- casez, 9
- Comments, 2
- Conditional assignment, 12
- Continuous assignment, 12
- Data operators, 4
- Data types, 3
- default, 9
- end, 8
- endcase, 9
- endmodule, 3, 5, 6
- Event control, 7
- for, 10
- function, 10
- Identifiers, 2
- if, 9
- inout, 3, 5, 6
- input, 3, 5, 6
- module, 5, 6
 - parameter, 6
 - nand, 13
 - negedge, 8
 - nor, 13
 - not, 13
 - notif0, 13
 - notif1, 13
 - or, 13
 - output, 3, 5, 6
 - posedge, 8
 - Primitive gates, 13
 - String, 3
 - concatenation, 5
 - while, 10
 - xnor, 13
 - xor, 13
- Verilog syntax:, 3

W

- while, 10

X

- xnor, 13
- xor, 13