Summary of Verilog Syntax

1. Module & Instantiation of Instances

A **Module** in *Verilog* is declared within the pair of keywords `module` and `endmodule`. Following the keyword `module` are the **module name** and **port interface list**.

```
module my_module ( a, b, c, d );
  input a, b;
  output c, d;
  ...
endmodule
```

All **instances** must be **named** except the instances of primitives. Only primitives in *Verilog* can have **anonymous instances**, i.e. `and`, `or`, `nand`, `nor`, `xor`, `xnor`, `buf`, `not`, `bufif1`, `bufi0`, `notif1`, `notif0`, `nmos`, `pmos`, `cmos`, `tran`, `tranif1`, `tranif0`, `rnmos`, `rpmos`, `rcmos`, `rtran`, `rtranif1`, `rtranif0`.

Port Connections at Instantiations

In *Verilog*, there are 2 ways of specifying connections among ports of instances.

a) **By ordered list** (positional association)
This is the more intuitive method, where the signals to be connected must appear in the module instantiation in the same order as the ports listed in module definition.

b) **By name** (named association)
When there are too many ports in the large module, it becomes difficult to track the order. Connecting the signals to the ports by the port names increases **readability** and reduces possible errors.

```
module top;
  reg A, B;
  wire C, D;

  my_module m1 (A, B, C, D);                    // By order
  my_module m2 (.b(B), .d(D), .c(C), .a(A));    // By name
  ...

endmodule
```

Parameterized Instantiations

The values of parameters can be **overridden** during instantiation, so that each instance can be customized separately. Alternatively, `defparam` statement can be used for the same purpose.

```
module my_module ( a, b, c, d );
  parameter x = 0;

  input a, b;
  output c, d;

  parameter y = 0, z = 0;
  ...
endmodule

module top;
  reg A, B;
  wire C, D;

  my_module #(2, 4, 3) m1 (A, B, C, D);
                              // x = 2, y = 4, z = 3 in instance m1

  my_module #(5, 3, 1) m2 (.b(B), .d(D), .c(C), .a(A));
                              // x = 5, y = 3, z = 1 in instance m2

  defparam m3.x = 4, m3.y = 2, m3.z = 5;
  my_module m3 (A, B, C, D);  // x = 4, y = 2, z = 5 in instance m3
  ...
endmodule
```

2. Data Types

There are 2 groups of data types in *Verilog*, namely **physical** and **abstract**.

a) Physical data type
- Net (`wire`, `wand`, `wor`, `tri`, `triand`, `trior`). Default value is `z`. Used mainly in structural modeling.
- Register (`reg`). Default value is `x`. Used in dataflow/RTL and behavioral modelings.
- Charge storage node (`trireg`). Default value is `x`. Used in gate-level and switch-level modelings.

b) Abstract data type — used only in behavioral modeling and test fixture.
- Integer (`integer`) stores 32-bit signed quantity.
- Time (`time`) stores 64-bit unsigned quantity from system task `$time`.
- Real (`real`) stores floating-point quantity.
- Parameter (`parameter`) substitutes constant.
- Event (`event`) is only name reference — does not hold value.

Unfortunately, the current standard of *Verilog* does not support user-defined types, unlike *VHDL*.

3. Values & Literals

Verilog provides 4 basic values,
a) `0` — logic zero or false condition
b) `1` — logic one, or true condition
c) `x` — unknown/undefined logic value. Only for physical data types.

d) `z` — high-impedance/floating state. Only for physical data types.

Constants in *Verilog* are expressed in the following format:

width `'`*radix value*

width — Expressed in decimal integer. Optional, default is inferred from value.

`'`*radix* — Binary(`b`), octal(`o`), decimal(`d`), or hexadecimal(`h`). Optional, default is decimal.

value — Any combination of the 4 basic values can be digits for radix octal, decimal or hexadecimal.

```
4'b1011     // 4-bit binary of value 1011
234         // 3-digit decimal of value 234
2'h5a       // 2-digit (8-bit) hexadecimal of value 5A
3'o671      // 3-digit (9-bit) octal of value 671
4b'1x0z     // 4-bit binary.  2nd MSB is unknown.  LSB is Hi-Z.
3.14        // Floating point
1.28e5      // Scientific notation
```

There are 8 different strength levels that can be associated by values 0 and 1.

Strength Level	Abbreviation	Type	Degree
`supply0` `supply1`	`Su0` `Su1`	driving	strongest
`strong0` `strong1`	`St0` `St1`	driving	
`pull0` `pull1`	`Pu0` `Pu1`	driving	
`large0` `large1`	`La0` `La1`	charge storage	
`weak0` `weak1`	`We0` `We1`	driving	
`medium0` `medium1`	`Me0` `Me1`	charge storage	
`small0` `small1`	`Sm0` `Sm1`	charge storage	
`highz0` `highz1`	`HiZ0` `HiZ1`		weakest

In the case of **contention**, the **stronger signal dominates**. Combination of 2 opposite values of same strength results in a value of `x`.

`St0`, `Pu1` ⇒ `St0`
`Su1`, `La1` ⇒ `Su1`
`Pu0`, `Pu1` ⇒ `PuX`

4. Nets & Registers

Net is the **connection** between ports of modules within a higher module. Net is used in test fixtures and all modeling abstraction including behavioral. Default value of net is **high-Z** (`z`). Nets just only **pass values** from one end to the other, i.e. it does not store the value. Once the output device discontinues driving the net, the value in the net becomes high-Z (`z`). Besides the usual net (`wire`), *Verilog* also provides special nets (`wor`, `wand`) to resolve the

final logic when there is logic contention by multiple drivers. `tri`, `trior` and `triand` are just the aliases for `wire`, `wor` and `wand` for readability reason.

Register is the **storage** that retains (remembers) the value last assigned to it, therefore, unlike `wire`, it needs not to be continuously driven. It is only used in the test fixture, behavioral, and dataflow modelings. The default value of a register is **unknown** (`x`).

Other special nets in *Verilog* are the supplies like V_{CC}/V_{DD} (`supply1`), Gnd (`supply0`), pullup (`pullup`) and pulldown (`pulldown`), resistive pullup (`tri1`) and resistive pulldown (`tri0`), and charge storage/capacitive node (`trireg`) which has **storage strength** associated with it.

5. Vectors & Arrays

Physical data types (`wire`, `reg`, `trireg`) can be declared as **vector/bus** (multiple bit widths). An **Array** is a chunk of consecutive values of the same type. Data types `reg`, `integer` and `time` can be declared as an array. Multidimensional arrays are not permitted in *Verilog*, however, arrays can be declared for vectored register type.

```
wire [3:0] data;        // 4-bit wide vector
reg bit [1:8];          // array of 8 1-bit scalar
reg [3:0] mem [1:8];    // array of 8 4-bit vector
```

The **range** of vectors and arrays declared can start from any integer, and in either ascending or descending order. However, when accessing the vector or array, the **slice** (subrange) specified must be within the range and in the same order as declared.

```
data[4]                 // Out-of-range
bit[5:2]                // Wrong order
```

There is no syntax available to access a bit slice of an array element — the array element has to be stored to a **temporary variable**.

```
// Can't do mem[7][2]
reg [3:0] tmp;          // Need temporary variable
tmp = mem[7];
tmp[2];
```

6. Tasks & Functions

Tasks and functions in *Verilog* closely resemble the procedures and functions in programming languages. Both tasks and functions are **defined locally** in the module in which the tasks and functions will be invoked. No `initial` or `always` statement may be defined within either tasks or functions.

Tasks and functions are different — `task` may have 0 or more arguments of type `input`, `output` or `inout` ; `function` must have at least one input argument. Tasks do not return value but pass values through `output` and `inout` arguments; functions always return a single value, but cannot have `output` or `inout` arguments. Tasks may contain

delay, event or timing control statements; functions may not. Tasks can invoke other tasks and functions; functions can only invoke other functions, but not tasks.

```
module m;
  reg [1:0] r1;
  reg [3:0] r2;
  reg r3;

  ...
  always
  begin
    ...
    r2 = my_func(r1);           // Invoke function
    ...
    my_task (r2, r3);           // Invoke task
    ...
  end

  task my_task;
    input [3:0] i;
    output o;
    begin
      ...
    end
  endtask
  ...
  function [3:0] my_func;
    input [1:0] i;
    begin
      ...
      my_func = ...;            // Return value
    end
  endfunction
  ...
endmodule
```

7. System Tasks & Compiler Directives

System tasks are the **built-in tasks** standard in *Verilog*. All system tasks are preceded with $. Some useful system tasks commonly used are:

```
$display("format", v1, v2, ...); // Similar format to printf() in C
$write("format", v1, v2, ...);   // $display appends newline at the end,
                                 //   but $write does not.
$strobe("format", v1, v2, ...);  // $strobe always executes last among
                                 //   assignment statements of the same
                                 //   time.  Order for $display among
                                 //   assignment statements of the same
                                 //   time is unknown.

$monitor("format", v1, v2, ...); // Invoke only once, and execute (print)
                                 //   automatically when any of the
                                 //   variables change value.
$monitoron;          // Enable monitoring from here
$monitoroff;         // Disable monitoring from here

$stop;               // Stop the simulation
$finish;             // Terminate and exit the simulation

$time;               // Return current simulation time in 64-bit integer
$stime;              // Return current simulation time in 32-bit integer
```

```
$realtime;          // Return current simulation time in 64-bit real

$random(seed);      // Return random number.  Seed is optional.
```

Compiler directives are instructions to *Verilog* during **compilation** instead of simulation. All compiler directives are preceded with `.

```
`define alias text  // Create an alias.  Aliases are replaced/substituted
                    //  prior to compilation.

`include file       // Insert another file as part of the current file.

`ifdef cond         // If cond is defined, compile the following.
`else
`endif
```

8. Operators

Operator Symbol	Function	Group	Operands	Precedence Rank
!	logical negation	Logical	unary	1
~	bitwise negation	Bitwise	unary	
&	reduction and	Reduction	unary	
|	reduction or	Reduction	unary	
^	reduction xor	Reduction	unary	
~&	reduction nand	Reduction	unary	
~|	reduction nor	Reduction	unary	
~^	reduction xnor	reduction	unary	
+	unary positive	arithmetic	unary	
-	unary negative	arithmetic	unary	
*	multiplication	arithmetic	binary	2
/	division	arithmetic	binary	
%	modulus	arithmetic	binary	
+	addition	arithmetic	binary	3
-	subtraction	arithmetic	binary	
<<	left shift	shift	binary	4
>>	right shift	shift	binary	
<	less than	relational	binary	5
<=	less than or equal	relational	binary	
>	greater than	relational	binary	
>=	greater than or equal	relational	binary	
==	equality	equality	binary	6
!=	inequality	equality	binary	
===	case equality	equality	binary	
!==	case inequality	equality	binary	
&	bitwise and	bitwise	binary	7
^	bitwise xor	bitwise	binary	8
^~	bitwise xnor	bitwise	binary	

`|`	bitwise or	bitwise	binary	9
`&&`	logical and	logical	binary	10
`||`	logical or	logical	binary	11
`?:`	conditional		ternary	12
`=`	blocking assignment	assignment	binary	13
`<=`	non-blocking assignment	assignment	binary	
`[]`	bit-select			
`[ : ]`	part-select			
`{}`	concatenation			
`{ {} }`	replication			

Operators within the same precedence rank are associated **from left to right**.

Verilog has **special syntax restriction** on using both **reduction** and **bitwise** operators within the same expression — even though reduction operator has higher precedence, parentheses must be used to avoid confusion with a logical operator.

```
a & (&b)
a | (|b)
```

Since bit-select, part-select, concatenation and replication operators use **pairs of delimiters** to specify their operands, there is no notion of operator precedence associated with them.

9. Structured Procedures

There are 2 structured procedure statements, namely `initial` and `always`. They are the basic statements for behavioral modeling from which other behavioral statements are declared. They **cannot be nested**, but many of them can be declared within a module.

a) `initial` statement
`initial` statement executes **exactly once** and becomes **inactive** upon exhaustion. If there are multiple `initial` statements, they all start to execute concurrently at time 0.

b) `always` statement
`always` statement **continuously repeats** itself throughout the simulation. If there are multiple `always` statements, they all start to execute concurrently at time 0. `always` statements may be triggered by events using an **event recognizing list** `@( )`.

10. Sequential & Parallel Blocks

Block statements group **multiple statements** together. Block statements can be either sequential or parallel. Block statements can be **nested** or **named** for direct access, and **disabled** if named.

a) Sequential block
Sequential blocks are delimited by the pair of keywords `begin` and `end`. The statements in sequential blocks are executed in the **order** they are specified, except non-blocking assignments.

b) Parallel block
Parallel blocks are delimited by the pair of keywords `fork` and `join`. The statements in parallel blocks are executed **concurrently**. Hence, the order of the statements in parallel blocks are immaterial.

11. Assignments

a) **Continuous assignment**
Continuous assignments are always **active** — changes in RHS (right hand side) expression is assigned to is LHS (left hand side) net.

LHS must be a scalar or vector of **nets**, and assignment must be performed **outside** procedure statements.

```
assign #delay net = expression;
```

Delay may be associated with the assignment, where new changes in expression is assigned to net after the delay. However, note that such delay is called **inertial delay**, i.e. if the expression changes again within the delay after the 1st change, only the latest change is assigned to net after the delay from 2nd change. The 1st change within the delay is not assigned to net.

b) **Procedural assignment**
LHS must be a scalar or vector of **registers**, and assignment must be performed **inside** procedure statements (`initial` or `always`). Assignment is only active (evaluated and loaded) when control is transferred to it. After that, the value of register remains until it is reassigned by another procedural assignment.

There are 2 types of procedural assignments:

- **Blocking assignment**
  Blocking assignments are executed in the order specified in the sequential block, i.e. a blocking assignment waits for previous blocking assignment of the same time to complete before executing.
  ```
  register = expression;
  ```

- **Nonblocking assignment**
  Nonblocking assignments are executed concurrently within the sequential blocks, i.e. a nonblocking assignment executes without waiting for other nonblocking assignments of occurring at the same time to complete.
  ```
  register <= expression;
  ```

Intra-assignment delay may be used for procedural assignment.

```
register = #delay expression;
```

The expression is evaluated immediately, but the value is assigned to register after the delay. This is equivalent to

```
reg temporary;
temporary = expression;
#delay register = temporary;
```

c) **Quasi-continuous** (procedural continuous) **assignment**
The LHS must be a scalar or vector of **registers**, and assignment must be **inside** procedure statements.

Similar to procedural assignment, however quasi-continuous assignment becomes **active** and **stays active** from the point of the assignment until it is **deactivated** through deassignment. When active, quasi-continuous assignment **overrides** any procedural assignment to the register.

```
begin
    ...
  assign register = expression1;  // Activate quasi-continuous
    ...
  register = expression2;         // No effect.  Overridden by active
                                  //   quasi-continuous
    ...
  assign register = expression3;  // Becomes active and overrides
                                  //  previous quasi-continuous
    ...
  deassign register;              // Disable quasi-continuous
    ...
  register = expression4;         // Executed.
    ...
end
```

There is **no delay** associated with quasi-continuous assignment. . Only the activation may be delayed. However, once it is **activated**, any changes in expression will be assigned to the register **immediately.**

12. Timing Controls

a) **Delay-based**
Execution of a statement can be delayed by a fixed-time period using the # operator.

```
#num statement; // Delay num time from previous statement before
                //  executing
```

Intra-assignment delay
This evaluates the RHS expression immediately, but delays for a fixed-time period before assigning to LHS, which must be a register.

```
register = #num expr; // Evaluate expr now, but delay num time unit
                      //   before assigning to register
```

b) **Event-based**

Execution of a statement is triggered by the change of value in a register or a net. The `@` operator captures such change of value within its **recognizing list**. To allow multiple triggers, use `or` between each event.

```
@(signal) statement;        // Execute whenever signal changes values
@(posedge signal) statement;// Execute at positive edge of signal
@(negedge signal) statement;// Execute at negative edge of signal
register = @(signal) expr;  // Similar to intra-assignment
always @(s1 or s2 or s3)    // Enter always block when either s1, s2
  ...                       //  or s3 changes value
```

Level-sensitive
The `@` is edge-sensitive. To achieve level-sensitive, use additional `if` statement to check the values of each event.

```
always @(signal)
  if ( signal )
    ...
  else
    ...
```

Alternatively, combination of `always` and `wait` can be used. But, note that `wait` is a blocking statement, i.e. `wait` blocks following statement until the condition is true.

```
always
  wait (event) statement;   // Execute statement when event is true
```

c) **Named-event**
Event is **explicitly triggered** (with `->` operator) and **recognized** (with `@` operator). Note that the named event cannot hold any data.

```
event my_event;       // Declare an event

always @( my_event )  // Execute when my_event is triggered
begin
  ...
end

always
begin
  ...
  if (...)
    -> my_event;      // Trigger my_event
  ...
end
```

13. Conditional Statements

The body only allows a single statement. If multiple statements are desired, block statements may be used to enclose multiple statements in place of the body.

a) **If-Then-Else**

```
if ( expr )
  statement;

if ( expr )
```

```
    statement;
  else
    statement;

  if ( expr ) statement;
  else if ( expr ) statement;
  else if ( expr ) statement;
  else statement;
```

b) **Case**

```
  case ( expr )
    value1 : statement;
    value2 : statement;
    value3 : statement;
      ...
    default : statement;
  endcase
```

14. Loop Statements

The body only allows a single statement. If multiple statements are desired, block statements may be used to enclose multiple statements in place of the body.

a) **While**

```
  while ( expr )
    statement;
```

b) **For**

```
  for ( init ; expr ; step )
    statement;
```

c) **Repeat**

Iterations are based on a constant instead of conditional expression.

```
  repeat ( constant )         // Fix number of loops
    statement;
```

d) **Forever**

```
  forever                     // Same as while (1)
    statement;
```

References:

[1] "Verilog-XL Reference Manual ver 2.2." OpenBook, Cadence Design Systems, 1995.

[2] Samir Palnitkar. "Verilog HDL: A Guide to Digital Design and Synthesis." SunSoft Press, 1996.

[3] Donald Thomas, Phil Moorby. "The Verilog Hardware Description Language, 2nd ed." Kluwer Academic Publishers, 1994.

[4] Eli Sternheim, Rajvir Singh, Rajeev Madhavan, Yatin Trivedi. "Digital Design and Synthesis with Verilog HDL." Automata Publishing Company, 1993.