



Syntax and Basic Elements

Modules

Modules are the fundamental building blocks in Verilog.

```
module module_name (port_list);
    input input_port;
    output output_port;
    // Internal signals
    wire internal_signal;
    // Logic implementation
    assign output_port = input_port &
internal_signal;
endmodule
```

Instantiation:

```
module_name instance_name (
    .input_port (signal_a),
    .output_port (signal_b)
);
```

Data Types

wire	Represents a physical wire, cannot store values.
reg	Represents a variable that stores a value. Used in <code>always</code> blocks.
integer	General-purpose variable, typically 32 bits.
real	Floating-point number.
time_t	Unsigned 64-bit integer for simulation time.
logic	Four-state data type: 0, 1, X (unknown), Z (high impedance). SystemVerilog only.

Number Representation

`<size>'<base><value>`

- `<size>`: Size in bits.
- `<base>`: `b` (binary), `o` (octal), `d` (decimal), `h` (hexadecimal).
- `<value>`: The number.

Examples:

- `4'b1010` - 4-bit binary number 1010.
- `8'hFF` - 8-bit hexadecimal number FF (255 in decimal).
- `16'd255` - 16-bit decimal number 255.

Operators

Arithmetic Operators

<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>\</code>	Division
<code>%</code>	Modulo
<code>**</code>	Exponentiation (SystemVerilog)

Bitwise Operators

<code>&</code>	Bitwise AND
<code> </code>	Bitwise OR
<code>^</code>	Bitwise XOR
<code>~</code>	Bitwise NOT
<code>~&</code>	Bitwise NAND
<code>~ </code>	Bitwise NOR
<code>~^</code> or <code>^~</code>	Bitwise XNOR

Reduction Operators

<code>&</code>	Reduction AND
<code> </code>	Reduction OR
<code>^</code>	Reduction XOR
<code>~&</code>	Reduction NAND
<code>~ </code>	Reduction NOR
<code>~^</code> or <code>^~</code>	Reduction XNOR

Logical Operators

<code>&&</code>	Logical AND
<code> </code>	Logical OR
<code>!</code>	Logical NOT

Control Statements

Always Blocks

Describes sequential logic.

```
always @(posedge clock or negedge reset)
begin
    if (!reset) begin
        // Reset condition
        q <= 0;
    end else begin
        // Sequential logic
        q <= d;
    end
end
```

- `always @(*)`: Combinational logic (sensitivity list inferred).
- `always @(posedge clk)`: Positive edge-triggered sequential logic.
- `always @(negedge clk)`: Negative edge-triggered sequential logic.

Conditional Statements

if Basic conditional statement.

```
if (condition) begin
    // Code to execute if
    condition is true
end
```

else Alternative code block.

```
if (condition) begin
    // Code if true
end else begin
    // Code if false
end
```

else Multiple conditions.

if

```
if (condition1) begin
    // Code if condition1 is true
end else if (condition2) begin
    // Code if condition2 is true
end else begin
    // Default code
end
```

case Multi-way branching.

```
case (expression)
    value1: begin
        // Code for value1
    end
    value2: begin
        // Code for value2
    end
    default: begin
        // Default code
    end
endcase
```

Loop Statements

for For loop.

```
integer i;
always @(posedge clk) begin
    for (i = 0; i < 10; i = i + 1)
    begin
        // Code to repeat
    end
end
```

while While loop (less common in synthesizable code).

e

```
integer i;
always @(posedge clk) begin
    i = 0;
    while (i < 10) begin
        // Code to repeat
        i = i + 1;
    end
end
```

Advanced Topics

Functions and Tasks

Function: Returns a single value. Cannot contain timing controls or event triggers.

```
function integer my_function (input
integer a, b);
    my_function = a + b;
endfunction
```

Task: Can have multiple outputs and can contain timing controls.

```
task my_task (input integer a, output
integer result);
    #10; // Delay for 10 time units
    result = a * 2;
endtask
```

Generate Statements

Used for conditional or repetitive instantiation of modules or code blocks at compile time.

```
generate
  if (parameter_value) begin : label
    // Conditional instantiation
    module_instance instance1 (
      .portA(signalA)
    );
  end else begin : label2
    // Alternative instantiation
    module_instance instance2 (
      .portB(signalB)
    );
  end
endgenerate
```

```
generate
  for (i = 0; i < N; i = i + 1) begin :
loop_label
  // Repetitive instantiation
  module_instance
array_of_instances[i] (
  .portA(signalA[i])
);
end
endgenerate
```

SystemVerilog Constructs (Highlights)

Logic Data Type	Four-state data type (0, 1, X, Z), preferred over <code>reg</code> and <code>wire</code> for general use.
Interfaces	Bundles signals for easier module connections and verification.
Assertions	Used to check design properties dynamically during simulation.
Clocking Blocks	Synchronize signals to a specific clock edge for improved timing control.