Verilog generate block

A `generate` block allows to multiply module instances or perform conditional instantiation of any module. It provides the ability for the design to be built based on Verilog parameters. These statements are particularly convenient when the same operation or module instance needs to be repeated multiple times or if certain code has to be conditionally included based on given Verilog parameters.

A `generate` block cannot contain port, parameter, `specparam` declarations or `specify` blocks. However, other module items and other generate blocks are allowed. All generate instantiations are coded within a `module` and between the keywords `generate` and `endgenerate`.

Generated instantiations can have either modules, continuous assignments, `always` or `initial` blocks and user defined primitives. There are two types of generate constructs - loops and conditionals.

- Generate for loop
- Generate if else
- Generate case

Generate for loop

A half adder will be instantiated `N` times in another top level design module called `my_design` using a `generate` for loop construct. The loop variable has to be declared using the keyword `genvar` which tells the tool that this variable is to be specifically used during elaboration of the generate block.

```verilog
 // Design for a half-adder
module ha ( input   a, b,
            output  sum, cout);

  assign sum  = a ^ b;
  assign cout = a & b;
endmodule

// A top level design that contains N instances of half adder
module my_design
    #(parameter N=4)
        (   input [N-1:0] a, b,
            output [N-1:0] sum, cout);

    // Declare a temporary loop variable to be used during
    // generation and won't be available during simulation
    genvar i;

    // Generate for loop to instantiate N times
    generate
        for (i = 0; i < N; i = i + 1) begin
          ha u0 (a[i], b[i], sum[i], cout[i]);
        end
    endgenerate
endmodule
```

Testbench

The testbench parameter is used to control the number of half adder instances in the design. When `N` is 2, `my_design` will have two instances of half adder.

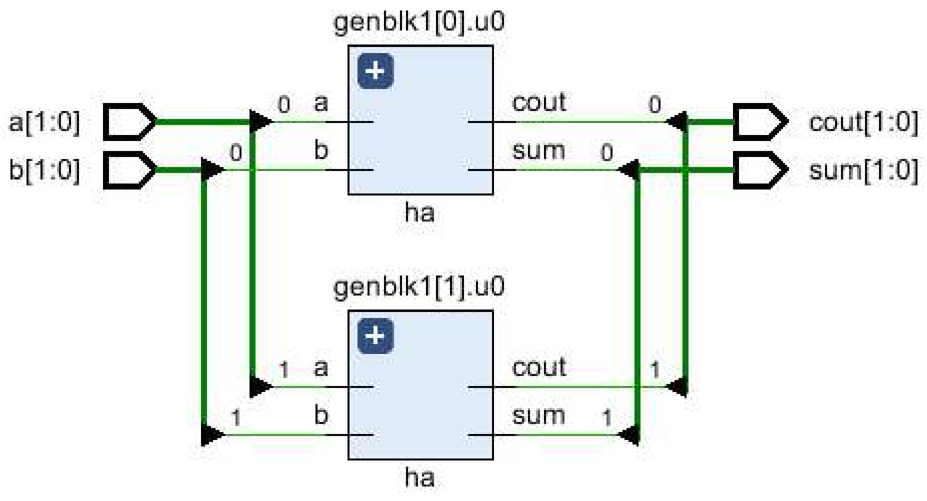
```
module tb;
    parameter N = 2;
  reg  [N-1:0] a, b;
  wire [N-1:0] sum, cout;

  // Instantiate top level design with N=2 so that it will have 2
  // separate instances of half adders and both are given two separate
  // inputs
  my_design #(.N(N)) md( .a(a), .b(b), .sum(sum), .cout(cout));

  initial begin
    a <= 0;
    b <= 0;

    $monitor ("a=0x%0h b=0x%0h sum=0x%0h cout=0x%0h", a, b, sum, cout);

    #10 a <= 'h2;
            b <= 'h3;
    #20 b <= 'h4;
    #10 a <= 'h5;
  end
endmodule
```

`a[0]` and `b[0]` gives the output `sum[0]` and `cout[0]` while `a[1]` and `b[1]` gives the output `sum[1]` and `cout[1]`.

Simulation Log

```
ncsim> run
a=0x0 b=0x0 sum=0x0 cout=0x0
a=0x2 b=0x3 sum=0x1 cout=0x2
a=0x2 b=0x0 sum=0x2 cout=0x0
a=0x1 b=0x0 sum=0x1 cout=0x0
ncsim: *W,RNQUIE: Simulation is complete.
ncsim> exit
```

See that elaborated RTL does indeed have two half adder instances generated by the `generate` block.



(https://www.chipverify.com/images/verilog/schematic/generate_block_for_loop_ha_schematic.png)

Generate if

Shown below is an example using an `if else` inside a `generate` construct to select between two different multiplexer implementations. The first design uses an `assign` statement to implement a mux while the second design uses a `case` statement. A parameter called USE_CASE is defined in the top level design module to select between the two choices.

```verilog
 // Design #1: Multiplexer design uses an "assign" statement to assign
// out signal
module mux_assign ( input a, b, sel,
                   output out);
  assign out = sel ? a : b;

  // The initial display statement is used so that
  // we know which design got instantiated from simulation
  // logs
  initial
    $display ("mux_assign is instantiated");
endmodule

// Design #2: Multiplexer design uses a "case" statement to drive
// out signal
module mux_case (input a, b, sel,
                 output reg out);
  always @ (a or b or sel) begin
    case (sel)
        0 : out = a;
        1 : out = b;
    endcase
  end

  // The initial display statement is used so that
  // we know which design got instantiated from simulation
  // logs
  initial
    $display ("mux_case is instantiated");
endmodule

// Top Level Design: Use a parameter to choose either one
module my_design (  input a, b, sel,
                    output out);
  parameter USE_CASE = 0;

  // Use a "generate" block to instantiate either mux_case
  // or mux_assign using an if else construct with generate
  generate
    if (USE_CASE)
      mux_case mc (.a(a), .b(b), .sel(sel), .out(out));
    else
      mux_assign ma (.a(a), .b(b), .sel(sel), .out(out));
  endgenerate

endmodule
```

Testbench

Testbench instantiates the top level module `my_design` and sets the parameter `USE_CASE` to 1 so that it instantiates the design using `case` statement.

```
module tb;
    // Declare testbench variables
  reg a, b, sel;
  wire out;
  integer i;

  // Instantiate top level design and set USE_CASE parameter to 1 so that
  // the design using case statement is instantiated
  my_design #(.USE_CASE(1)) u0 ( .a(a), .b(b), .sel(sel), .out(out));

  initial begin
    // Initialize testbench variables
    a <= 0;
    b <= 0;
    sel <= 0;

    // Assign random values to DUT inputs with some delay
    for (i = 0; i < 5; i = i + 1) begin
      #10 a <= $random;
          b <= $random;
          sel <= $random;
      $display ("i=%0d a=0x%0h b=0x%0h sel=0x%0h out=0x%0h", i, a, b, sel, out);
    end
  end
endmodule
```

When the parameter `USE_CASE` is 1, it can be seen from the simulation log that the multiplexer design using `case` statement is instantiated. And when `USE_CASE` is zero, the multiplexer design using `assign` statement is instantiated. This is visible from the display statement that gets printed in the simulation log.

Simulation Log

```
// When USE_CASE = 1
ncsim> run
mux_case is instantiated
i=0 a=0x0 b=0x0 sel=0x0 out=0x0
i=1 a=0x0 b=0x1 sel=0x1 out=0x1
i=2 a=0x1 b=0x1 sel=0x1 out=0x1
i=3 a=0x1 b=0x0 sel=0x1 out=0x0
i=4 a=0x1 b=0x0 sel=0x1 out=0x0
ncsim: *W,RNQUIE: Simulation is complete.

// When USE_CASE = 0
ncsim> run
mux_assign is instantiated
i=0 a=0x0 b=0x0 sel=0x0 out=0x0
i=1 a=0x0 b=0x1 sel=0x1 out=0x0
i=2 a=0x1 b=0x1 sel=0x1 out=0x1
i=3 a=0x1 b=0x0 sel=0x1 out=0x1
i=4 a=0x1 b=0x0 sel=0x1 out=0x1
ncsim: *W,RNQUIE: Simulation is complete.
```

Generate Case

A generate case allows modules, initial and always blocks to be instantiated in another module based on a `case` expression to select one of the many choices.

```verilog
 // Design #1: Half adder
module ha (input a, b,
           output reg sum, cout);
  always @ (a or b)
  {cout, sum} = a + b;

  initial
    $display ("Half adder instantiation");
endmodule

// Design #2: Full adder
module fa (input a, b, cin,
           output reg sum, cout);
  always @ (a or b or cin)
  {cout, sum} = a + b + cin;

    initial
      $display ("Full adder instantiation");
endmodule

// Top level design: Choose between half adder and full adder
module my_adder (input a, b, cin,
                 output sum, cout);
  parameter ADDER_TYPE = 1;

  generate
    case(ADDER_TYPE)
      0 : ha u0 (.a(a), .b(b), .sum(sum), .cout(cout));
      1 : fa u1 (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
    endcase
  endgenerate
endmodule
```

Testbench

```verilog
module tb;
  reg a, b, cin;
  wire sum, cout;

  my_adder #(.ADDER_TYPE(0)) u0 (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));

  initial begin
    a <= 0;
    b <= 0;
    cin <= 0;

    $monitor("a=0x%0h b=0x%0h cin=0x%0h cout=0%0h sum=0x%0h",
             a, b, cin, cout, sum);

    for (int i = 0; i < 5; i = i + 1) begin
      #10 a <= $random;
      b <= $random;
      cin <= $random;
    end
  end
endmodule
```

Note that because a half adder is instantiated, `cin` does not have any effect on the outputs `sum` and `cout`.

Simulation Log

```
ncsim> run
Half adder instantiation
a=0x0 b=0x0 cin=0x0 cout=00 sum=0x0
a=0x0 b=0x1 cin=0x1 cout=00 sum=0x1
a=0x1 b=0x1 cin=0x1 cout=01 sum=0x0
a=0x1 b=0x0 cin=0x1 cout=00 sum=0x1
ncsim: *W,RNQUIE: Simulation is complete.
```