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.

```
// 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
```

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.

```
// 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.

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.

```
// 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

```
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
```

More on Generate Blocks in Verilog

The generate statement in Verilog is a very useful construct that generates synthesizable code during elaboration time dynamically. The simulator provides an elaborated code of the 'generate' block. It provides the below facilities:

1. To generate multiple module instances or code repetition.
2. Conditionally instantiate a block of code based on the Verilog parameter, however, the parameter is not permitted in the generate statement.

It basically provides control on variables, functions, tasks, and instantiation declarations. A generate block has been written within generate and endgenerate keywords.

Types of generate instantiation

1. Modules
2. Verilog gate primitives
3. Continuous assignments
4. Initial and always blocks
5. User-defined primitives

Let's see what is allowed within the scope of a generate block.

A. Data types

    1. integer, real
    2. net, reg
    3. time, realtime
    4. event

B. Function and task

Note: Function and task are not allowed within a generate loop, but they are allowed in generate block.

Below module items/declarations are not allowed within the scope of a generate block

1. Port declarations like input, output, and inout
2. specify blocks
3. parameters and local parameters

Methods to write generate statements

1. Generate loop
2. Generate conditional (includes generate if-else and generate case)

Generate loop

The generate loop is similar to the for loop statement, but it uses genvar keyword as a loop variable.

- The genvar keyword is only used during the evaluation of generate block and does not exist during the simulation of the design. It needs to be used by a generate loop.
- Generate loop provides flexibility to reduce code lines by replacing repetitive statements to a single statement like for loop.
- Similar to a for loop, generate loops also can be nested with different genvar as an index variable.

Example: Ripple Carry Adder

```
module full_adder(
  input a, b, cin,
  output sum, cout
);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module ripple_carry_adder #(parameter SIZE = 4) (
  input [SIZE-1:0] A, B,
  input Cin,
  output [SIZE-1:0] S, Cout);

  genvar g;

  full_adder fa0(A[0], B[0], Cin, S[0], Cout[0]);
  generate  // This will instantial full_adder SIZE-1 times
    for(g = 1; g<SIZE; g++) begin
      full_adder fa(A[g], B[g], Cout[g-1], S[g], Cout[g]);
    end
  endgenerate
endmodule
```

Generate conditional

A generate block allows conditionally instantiated using if-else-if construct and case keyword.

Example: generate If-else

In the below example, based on parameter sel full adder or half-adder design is instantiated. By default, parameter sel = 0 means half adder will be instantiated. But from the testbench code, parameter sel = 1 is passed to instantiate full adder. $display cannot be used within generate block without initial block, otherwise, it throws an error '$display' is an invalid generate scope construct.

```
module half_adder(
  input a, b,
  output sum, cout
);

  assign {sum, cout} = {a^b, (a & b)};
  //or
  //assign sum = a^b;
  //assign cout = a & b;
endmodule

module full_adder(
  input a, b, cin,
  output sum, cout
);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module gen_if_ex #(parameter sel = 0)(
  input A, B, Cin,
  output S, Cout);

  generate
    if(sel) begin
      initial $display("Full Adder is selected");
      full_adder fa(A, B, Cin, S, Cout);
    end
    else begin
      initial $display("Half Adder is selected");
      half_adder ha(A, B, S, Cout);
    end
  endgenerate
endmodule
```

Example: generate case

Similarly, the above example if-else generate block can alternatively use case statement as specified in the below example.

```
module half_adder(
  input a, b,
  output sum, cout
);

  assign {sum, cout} = {a^b, (a & b)};
  //or
  //assign sum = a^b;
  //assign cout = a & b;
endmodule

module full_adder(
  input a, b, cin,
  output sum, cout
);

  assign {sum, cout} = {a^b^cin, ((a & b) | (b & cin) | (a & cin))};
  //or
  //assign sum = a^b^cin;
  //assign cout = (a & b) | (b & cin) | (a & cin);
endmodule

module gen_if_ex #(parameter sel = 0)(
  input A, B, Cin,
  output S, Cout);

  generate
    case(sel)
      0: begin
           initial $display("Full Adder is selected");
           half_adder ha(A, B, S, Cout);
         end
      1: begin
           initial $display("Full Adder is selected");
           full_adder fa(A, B, Cin, S, Cout);
         end
    endcase
  endgenerate
endmodule
```