36. Generate blocks

Background

A complex design modeled in Verilog may have several levels of hierarchy, where one module contains instances of other modules. Even small designs are often represented as a top-level module that connects together several lower level modules. One example might be building a larger 64-bit adder from several smaller 8-bit adders.

At the structural level of modeling, Verilog-1995 provides three ways to instantiate multiple modules or primitives:

- Separate instance statements, such as:

```
adder8 u0 (sum[7:0],   co[1],  a[7:0],   b[7:0],   ci[0]);
adder8 u1 (sum[15:8],  co[2],  a[15:8],  b[15:8],  ci[1]);
adder8 u2 (sum[23:16], co[3],  a[23:16], b[23:16], ci[2]);
...
adder8 u7 (sum[63:56], co[8],  a[63:56], b[63:56], ci[7]);
```

- A comma-separated list of instances, such as:

```
adder8 u0 (sum[7:0],   co[1],  a[7:0],   b[7:0],   ci[0]),
       u1 (sum[15:8],  co[2],  a[15:8],  b[15:8],  ci[1]),
       u2 (sum[23:16], co[3],  a[23:16], b[23:16], ci[2]),
       ...
       u7 (sum[63:56], co[8],  a[63:56], b[63:56], ci[7]);
```

- An instance array, such as:

```
adder8 u[7:0] (sum, co[7:0], a, b, ci[7:0]);
```

The array of instances construct is a convenient short cut for creating multiple instances of a module or primitive, but it does have limitations. The construct works well when each instance is connected to the same signals. It becomes convoluted, if not impossible, to use an array of instances to create a netlist of components with more complex interconnections between them.

At a more abstract level of modeling, Verilog-1995 does not have a method for replicating the same block of logic several times. Behavioral and RTL level models are represented with `always` procedures and continuous assignments. The following RTL example models a gray-code to binary converter using continuous assignments (the example assumes an 8-bit gray code input):

```
assign bin[7] = gray[7];
assign bin[6] = bin[7] ^ gray[6];
assign bin[5] = bin[6] ^ gray[5];
assign bin[4] = bin[5] ^ gray[4];
assign bin[3] = bin[4] ^ gray[3];
assign bin[2] = bin[3] ^ gray[2];
assign_bin[1] = bin[2] ^ gray[1];
assign bin[0] = bin[1] ^ gray[0];
```

Verilog-1995 does not provide a method to create multiple procedures or continuous assignments. Each one must be created separately, often by typing the code by hand.

What's new

Verilog-2001 adds a powerful construct for creating multiple instances of an object within a module, the *generate block*. The following new reserved words have been added: **`generate`**, **`endgenerate`** and **`genvar`**.

Most types of objects that can be placed within a Verilog module can also be generated from a generate block, making it easy to create any number of those objects. The primary items which can be generated are:

- Any number of module and primitive instances
- Any number of `initial` or `always` procedural blocks
- Any number of continuous assignments
- Any number of net and variable declarations
- Any number of parameter redefinitions
- Any number of task or function definitions

Items that are not permitted in a generate statement include: port declarations, constant declarations, and specify blocks.

Within the generate block, **`for`** loops can be used to create any number of objects. In addition, **`if`**—**`else`** decisions and **`case`** decisions can be used to selectively control what objects are generated.

The genvar variable

The **`genvar`** variable is a special integer variable for use as the index control variable by generate `for` loops. The value of a `genvar` variable can only be assigned a positive number or 0; values with X or Z, or negative values cannot be assigned. A `genvar` variable can only be used during elaboration, and cannot be accessed during

simulation—it no longer exists at that point. A `genvar` can only be assigned a value as part of a generate `for` loop control statement. Its value can be read within a generate block any place that a parameter constant could be used.

A `genvar` variable can be declared outside of a generate block, or within a generate block. When declared outside of the generate block, any number of generate blocks can use the same variable.

Generate loops

A generate **`for`** loop is used to create one or more instances of items that can be placed within a Verilog module. The loop is essentially the same as a regular Verilog HDL for loop, but with these limitations:

- The index loop variable must be a `genvar`.
- Both assignments in the for loop control must assign to the same `genvar`.
- The contents of the loop must be within a named `begin`—`end` block.

The reason the contents of a generate loop must be within a named `begin`—`end` block is so that a unique name can be created for each generated name, such as a module instance name or net name. The name of the block becomes a prefix to the generated names, followed by the loop variable count in square brackets.

The example having 8 instances of an 8-bit adder module shown on the previous page can easily be generated as follows:

```
generate
  genvar i;
  for (i=0; i<=7; i=i+1)
    begin: u
      adder8 add (sum[(i*8)+:8], co[i+1],
                  a[(i*8)+:8], b[(i*8)+:8], ci[i]);
    end
endgenerate
```

The generated instance names in the preceding example are:

```
u[0].add
u[1].add
...
u[7].add
```

Note: the square brackets in the generated names are regular characters. They do not represent indexes into an array, and cannot be represented with a variable. (i.e. `ff[i].u1` is illegal).

The gray-code to binary converter example, with eight continuous assignment statements, can also be generated with just a few lines of code, as follows.

```
generate
  genvar i;
  assign bin[7] = gray[7];
  for (i=6; i>=0; i=i-1)
    begin: gray2bin
      assign bin[i] = bin[i+1] ^ gray[i];
    end
endgenerate
```

The number of objects generated can also be based on constants. If parameter constants are used, the value of the constant can be redefined at elaboration, allowing the number of objects generated to be easily configured. For example:

```
parameter SIZE = 64;
generate
  genvar i;
  assign bin[SIZE-1] = gray[SIZE-1];
  for (i=SIZE-2; i>=0; i=i-1)
    begin: gray2bin
      assign bin[i] = bin[i+1] ^ gray[i];
    end
endgenerate
```

The generate block can be used to generate procedural code as well as structural code and continuous assignments. The following example illustrates using a generate statement to create multiple `always` procedures. The example creates a sequential logic gray-code to binary converter, using a slightly different algorithm for the conversion.

```
parameter SIZE = 8;
genvar i;
generate
  for (i=0; i<=SIZE-1; i=i-1)
    begin: proc
      always @(posedge clock or negedge reset)
        if (reset == 0)
          bin[i] = 1'b0;
        else
          bin[i] = ^gray[SIZE-1:i];
    end
endgenerate
```

Conditional generation

Verilog **if—else** and **case** statements can be used within a generate block to control what objects are generated. The following example examines the width of the input busses to determine what type of multiplier should be instantiated.

```
module multiplier (a, b, product);
  parameter a_width = 8, b_width = 8;
  localparam product_width = a_width + b_width;
  input   [a_width-1:0]       a;
  input   [b_width-1:0]       b;
  output  [product_width-1:0] product;

  generate
    if ((a_width < 8) || (b_width < 8))
      CLA_mult #(a_width, b_width) u1 (a, b, product);
    else
      WALLACE_mult #(a_width, b_width) u1 (a, b, product);
  endgenerate
endmodule
```

A **case** statement can be used when there are several conditions to test to determine what should be generated. The case expression and case items can only use constant expressions or `genvar` variables. That is, the values which are tested must be known values at elaboration time. Regular variables and nets will not have values during elaboration, and therefore cannot be used in a generate block.

The following example illustrates using a generate `case` statement to control which objects are generated:

```
parameter WIDTH = 1;
generate
  case (WIDTH)
    1:  adder_1bit x1(co, sum, a, b, ci);
    2:  adder_2bit x1(co, sum, a, b, ci);
    default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
  endcase
endgenerate
```

Using arrays with generate blocks

Generate blocks can be used to connect module or primitive instances up to a matrix of signals, represented by arrays of nets or variables. Verilog-2001 permits multidimensional arrays of any data type, and to select bits or parts from an array (see sections 15, 16 and 17). The combination of arrays and generate blocks makes it possible to create elaborate netlists with a minimal amount of code.

The following example illustrates declaring an array of net vectors, where each vector is 4 bits wide (using the default value of the SIZE parameter), with 3 vectors in the array (three internal nets are needed to connect the gates within each adder that is instantiated). Each pass of the generate loop instantiates the logic gates required to model a 1-bit adder, and connects the gates together using bits from the net array, **n**.

```
module Nbit_gate_adder #(parameter SIZE = 4)
                        (output wire [SIZE-1:0] sum,
                         output wire            co;
                         input  wire [SIZE-1:0] a, b,
                         input  wire            ci);

  wire   [SIZE  :0] c;   //internal carry bits between adders
  assign c[0] = ci;      //carry in
  assign co   = c[SIZE]; //carry out

  wire [SIZE-1:0] n [1:3];   //internal nets in each adder
  genvar            i;

  generate
    for(i=0; i<SIZE; i=i+1)
      begin: addbit
        xor g1 ( n[1][i],    a[i],    b[i]);
        xor g2 (  sum[i], n[1][i],    c[i]);
        and g3 ( n[2][i],    a[i],    b[i]);
        and g4 ( n[3][i], n[1][i],    c[i]);
        or  g5 (  c[i+1], n[2][i], n[3][i]);
      end
  endgenerate
endmodule
```

Synthesis considerations

This enhancement should be synthesizable.