Verilog-2001 之 generate 语句的用法

Verilog-1995 支持通过以声明实例数组的形式对 primitive 和 module 进行复制结构建模。而在 Verilog-2001 里，新增加的 generate 语句拓展了这种用法（其思想来源于 VHDL 语言）。除了允许复制产生 primitive 和 module 的多个实例化，同时也可以复制产生多个 net、reg、parameter、assign、always、initial、task、function。

在 generate 中引入了一种新的变量类型：genvar，用以在 generate-for 语句中声明一个**正整数**的索引变量(如果将"X"或"Z"或者"负值"赋给 genvar 变量,将会出错)。genvar 变量可以声明在 generate 语句内，也可以声明在 generate 语句外。

generate 语句有 **generate-for**、**genreate-if** 和 **generate-case** 三种语句；

■ generate-for 语句

① generate-for 语句必须用 genvar 关键字定义 for 的索引变量；
② for 的内容必须用 begin…end 块包起来，哪怕只有一句；
③ begin…end 块必须起个名字；

例 1：一个参数化的 gray-code to binary-code 转换器；这里采用复制产生多个 assign 语句的形式来实现；

```
module gray2bin1 (bin, gray);
    parameter SIZE = 8;     // this module is parameterizable
    output [SIZE-1:0] bin;
    input [SIZE-1:0] gray;
    genvar i;
    generate
        for(i=0; i<SIZE; i=i+1)
        begin: bit
            assign bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

等同于下面的语句：

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

例 2：还是例 1 的 gray-code to binary-code 转换器；不过这里采用复制产生多个 always 语句的形式来实现；

```
module gray2bin2 (bin, gray);
    parameter SIZE = 8; // this module is parameterizable
    output [SIZE-1:0] bin;
```

```
    input [SIZE-1:0] gray;
    reg [SIZE-1:0] bin;
    genvar i;
    generate
        for(i=0; i<SIZE; i=i+1)
        begin: bit
            always @(gray[SIZE-1:i]) // fixed part select
                bin[i] = ^gray[SIZE-1:i];
        end
    endgenerate
endmodule
```

等同于下面的语句：

```
always @(gray[SIZE-1:0]) // fixed part select
    bin[0] = ^gray[SIZE-1:0];
always @(gray[SIZE-1:1]) // fixed part select
    bin[1] = ^gray[SIZE-1:1];
always @(gray[SIZE-1:2]) // fixed part select
    bin[2] = ^gray[SIZE-1:2];
always @(gray[SIZE-1:3]) // fixed part select
    bin[3] = ^gray[SIZE-1:3];
always @(gray[SIZE-1:4]) // fixed part select
    bin[4] = ^gray[SIZE-1:4];
always @(gray[SIZE-1:5]) // fixed part select
    bin[5] = ^gray[SIZE-1:5];
always @(gray[SIZE-1:6]) // fixed part select
    bin[6] = ^gray[SIZE-1:6];
always @(gray[SIZE-1:7]) // fixed part select
    bin[7] = ^gray[SIZE-1:7];
```

例 3：一个行波进位加法器，在 begin…end 内部定义局部变量，并且在 generate 语句内定义 genvar 变量；

```
module addergen1 (co, sum, a, b, ci);
    parameter SIZE = 4;
    output [SIZE-1:0] sum;
    output co;
    input [SIZE-1:0] a, b;
    input ci;
    wire [SIZE :0] c;
    assign c[0] = ci;
    generate
        genvar i;
        for(i=0; i<SIZE; i=i+1)
        begin: bit
            wire t1, t2, t3; // generated net declaration
```

```
            xor g1 (t1, a[i], b[i]);
            xor g2 (sum[i], t1, c[i]);
            and g3 (t2, a[i], b[i]);
            and g4 (t3, t1, c[i]);
            or g5 (c[i+1], t2, t3);
        end
    endgenerate
    assign co = c[SIZE];
endmodule
```

等同于下面的语句：

```
wire bit[0].t1, bit[0].t2, bit[0].t3;
xor bit[0].g1 (bit[0].t1, a[0], b[0]);
xor bit[0].g2 (sum[0], bit[0].t1, c[0]);
and bit[0].g3 (bit[0].t2, a[0], b[0]);
and bit[0].g4 (bit[0].t3, bit[0].t1, c[0]);
or  bit[0].g5 (c[0+1], bit[0].t2, bit[0].t3);

wire bit[1].t1, bit[1].t2, bit[1].t3;
xor bit[1].g1 (bit[1].t1, a[1], b[1]);
xor bit[1].g2 (sum[1], bit[1].t1, c[1]);
and bit[1].g3 (bit[1].t2, a[1], b[1]);
and bit[1].g4 (bit[1].t3, bit[1].t1, c[1]);
or  bit[1].g5 (c[1+1], bit[1].t2, bit[1].t3);

wire bit[2].t1, bit[2].t2, bit[2].t3;
xor bit[2].g1 (bit[2].t1, a[2], b[2]);
xor bit[2].g2 (sum[2], bit[2].t1, c[2]);
and bit[2].g3 (bit[2].t2, a[2], b[2]);
and bit[2].g4 (bit[2].t3, bit[2].t1, c[2]);
or  bit[2].g5 (c[2+1], bit[2].t2, bit[2].t3);

wire bit[3].t1, bit[3].t2, bit[3].t3;
xor bit[3].g1 (bit[3].t1, a[3], b[3]);
xor bit[3].g2 (sum[3], bit[3].t1, c[3]);
and bit[3].g3 (bit[3].t2, a[3], b[3]);
and bit[3].g4 (bit[3].t3, bit[3].t1, c[3]);
or  bit[3].g5 (c[3+1], bit[3].t2, bit[3].t3);
```

这样，复制产生的实例名称为：

```
xor gates:    bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
              bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
and gates:    bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
```

```
                bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
or gates:       bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
```

复制产生的 wire 组为：

```
bit[0].t1 bit[1].t1 bit[2].t1 bit[3].t1
bit[0].t2 bit[1].t2 bit[2].t2 bit[3].t2
bit[0].t3 bit[1].t3 bit[2].t3 bit[3].t3
```

可见，给 begin…end 块命名的用途！

例 4：一个行波进位加法器，使用外部定义的变量，这样就不会对外部变量做变动；

```
module addergen1 (co, sum, a, b, ci);
    parameter SIZE = 4;
    output [SIZE-1:0] sum;
    output co;
    input [SIZE-1:0] a, b;
    input ci;
    wire [SIZE :0] c;
    wire [SIZE-1:0] t [1:3];
    genvar i;
    assign c[0] = ci;
    generate
        for(i=0; i<SIZE; i=i+1)
        begin: bit
            xor g1 ( t[1][i], a[i], b[i]);
            xor g2 ( sum[i], t[1][i], c[i]);
            and g3 ( t[2][i], a[i], b[i]);
            and g4 ( t[3][i], t[1][i], c[i]);
            or g5 ( c[i+1], t[2][i], t[3][i]);
        end
    endgenerate
    assign co = c[SIZE];
endmodule
```

这样，复制产生的实例名称为：

```
xor gates:      bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
                bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
and gates:      bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
                bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
or gates:       bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
```

而外部变量 t 与 a,b,sum 等一样，没有复制产生新的变量组

例 5：多层 generate 语句所复制产生的实例命名方式；

```
parameter SIZE = 2;
genvar i, j, k, m;
```

```
generate
    for(i=0; i<SIZE; i=i+1)
    begin: B1     // scope B1[i]
        M1 N1();  // instantiates B1[i].N1
        for(j=0; j<SIZE; j=j+1)
        begin: B2     // scope B1[i].B2[j]
            M2 N2();  // instantiates B1[i].B2[j].N2
            for(k=0; k<SIZE; k=k+1)
            begin: B3     // scope B1[i].B2[j].B3[k]
                M3 N3();  // instantiates B1[i].B2[j].B3[k].N3
            end
        end
        if(i>0)
            for(m=0; m<SIZE; m=m+1)
            begin: B4     // scope B1[i].B4[m]
                M4 N4();  // instantiates B1[i].B4[m].N4
            end
    end
endgenerate
```

下面是复制产生的实例名称的几个例子：

```
B1[0].N1  B1[1].N1
B1[0].B2[0].N2   B1[0].B2[1].N2
B1[0].B2[0].B3[0].N3    B1[0].B2[0].B3[1].N3
B1[0].B2[1].B3[0].N3
B1[1].B4[0].N4   B1[1].B4[1].N4
```

■ **generate-if 语句**

根据条件不同产生不同的实例化，即根据模块参数（常量）的条件是否满足来选择其中一段代码生成相应的电路，不如`ifdef …`elsif …`else …`endif;

例 1：

```
module generate_if (a,b,c,y);
    input a,b,c;
    output y;
    localparam SIZE = 12;   // 参数常量

    generate
        if (SIZE < 8)
            assign y = a & b & c;
        else if (SIZE == 8)
            assign y = a & b | c;
        else
            assign y = a | b | c;   // 最后该语句生成电路;
    endgenerate
endmodule
```

例 2：

```
module multiplier(a,b,product);
    parameter a_width = 8, b_width = 8;
    localparam product_width = a_width+b_width;
// localparam can not be modified directly with the defparam
statement or the module instance statement，它是内部使用的局部参数 #
    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_multiplier #(a_width,b_width) u0(a, b, product);
            // instantiate a CLA multiplier
        else
            WALLACE_multiplier #(a_width,b_width)u1(a,b,product);
            // instantiate a Wallace-tree multiplier
    endgenerate
    // The generated instance name is u1
endmodule
```

注意：这里的条件都是常量条件，非常量条件不能综合；

■ **generate-case 语句**

跟 generate-if 语句一样的用法，只是采用 case 语句的形式。

例 1：

```
generate
    case(WIDTH)
        1: adder_1bit x1(co, sum, a, b, ci);
        // 1-bit adder implementation
        2: adder_2bit x1(co, sum, a, b, ci);
        // 2-bit adder implementation
        default: adder_cla #(WIDTH) x1(co, sum, a, b, ci);
        // others - carry look-ahead adder
    endcase
    // The generated instance name is x1
endgenerate
```

厦门大学电路所
huangshiwei
若有误，请联系作者，谢谢纠错
联系邮箱：win_hshiw@163.com