Verilog HDL语言规范

主 讲：何宾

Email：hebin@mail.buct.edu.cn

2014.06

Verilog HDL层次化结构

Verilog HDL支持通过将一个模块嵌入到其它模块的层次化描述结构。

- 高层次模块创建低层次模块的例化，并且通过input、output和inout端口进行通信。
- 这些模块的端口为标量或者矢量。

Verilog HDL层次化结构
--模块和模块例化

在Verilog HDL的描述中对模块和模块例化的方法进行了简单的说明。下面介绍模块例化中涉及到一些其它问题。

Verilog HDL层次化结构
--覆盖模块参数值

Verilog HDL提供了两种定义参数的方法。

- 一个模块声明中可以包含一个或多个类型的参数定义
- 或者不包含参数定义。

Verilog HDL层次化结构
--覆盖模块参数值

模块参数可以有一个类型说明和一个范围说明。根据下面的规则，确定参数对一个参数类型和范围覆盖的结果：

- 默认地，对于一个没有类型和参数说明的参数说明，由最终参数值的值确定参数值的类型和范围。
- 一个带有范围说明，但没有类型说明的参数声明，其范围参数声明的范围，类型是无符号的。一个覆盖的值将转换到参数的类型和范围。

Verilog HDL层次化结构
--覆盖模块参数值

- 一个带有类型说明，但没有范围说明的参数声明，其类型是参数声明的类型。一个覆盖的值将转换到参数的类型。一个有符号的参数将默认到分配给参数最终覆盖值的范围。
- 一个带有有符号类型说明和范围说明的参数声明，其类型是有符号的，范围是参数声明的范围。一个覆盖的值将最终转换到参数的类型和范围。

Verilog HDL层次化结构
--覆盖模块参数值

参数的Verilog HDL描述的例子

```
module generic_fifo
#(parameter MSB=3, LSB=0, DEPTH=4)
//可以覆盖这些参数
(input [MSB:LSB] in,
input clk, read, write, reset,
output [MSB:LSB] out,
output full, empty );
localparam FIFO_MSB = DEPTH*MSB;
localparam FIFO_LSB = LSB;
// 这些参数是本地的，不能被覆盖
// 通过修改上面的参数，影响它们，模块将正常的工作。
```

Verilog HDL层次化结构
--覆盖模块参数值

```
reg [FIFO_MSB:FIFO_LSB] fifo;
        reg [LOG2(DEPTH):0] depth;
always @(posedge clk or reset) begin
    casex ({read,write,reset})
        // 实现fifo
    endcase
  end
endmodule
```

Verilog HDL层次化结构
--覆盖模块参数值

Verilog HDL提供了两种方法，用于修改非本地参数值：

- defparam语句

  允许使用层次化的名字，给参数分配值。

- 模块例化参数值分配

  允许在模块例化行内，给参数分配值。通过列表的顺序或者名字，分配模块例化参数的值。

注：如果defparam分配和模块例化参数冲突时，模块内的参数将使用defparam指定的值。

Verilog HDL层次化结构
--覆盖模块参数值

defparam分配参数值

参数定义对分配值类型和范围影响的Verilog HDL描述的例子。

```
module foo(a,b);
        real r1,r2;
        parameter [2:0] A = 3'h2;
        parameter B = 3'h2;
        initial begin
                r1 = A;
```

Verilog HDL层次化结构
--覆盖模块参数值

```
            r2 = B;
            $display("r1 is %f r2 is %f",r1,r2);
        end
endmodule       // foo

module bar;
            wire a,b;
            defparam f1.A = 3.1415;
            defparam f1.B = 3.1415;
            foo f1(a,b);
endmodule     // bar
```

Verilog HDL层次化结构
--覆盖模块参数值

- 该例子中，A指定了范围，B没有。所以，将f1.A=3.1415的浮点数，转换为定点数3。3的低三位赋值给A；而B由于没有说明范围和类型，所以没有进行转换。
- 使用defparam语句，通过在设计中，使用参数的层次化名字，在任何模块例化中，均可修改参数的值。defparm语句对于一个模块内一组参数值同时进行覆盖是非常有用的。

注意：一个层次内的defparam语句，或者在一个生成模块下面的defparam语句，或者一组例化的defparam语句，均不能改变当前层次外的参数值。

Verilog HDL层次化结构
--覆盖模块参数值

下面的Verilog HDL描述将不会修改参数的值。

```
genvar i;
generate
for (i = 0; i < 8; i = i + 1) begin : somename
        flop my_flop(in[i], in1[i], out1[i]);
        defparam somename[i+1].my_flop.xyz = i ;
end
endgenerate
```

注:对于多个defparam语句用于单个参数的情况，参数只使用最后一个defparam语句所分配的值。当在多个源文件中存在defparam时，未定义参数所使用的值。

Verilog HDL层次化结构
--覆盖模块参数值

下面的Verilog HDL将无法定义参数使用的值。

```
module top;
reg clk;
reg [0:4] in1;
reg [0:9] in2;
wire [0:4] o1;
wire [0:9] o2;
vdff m1 (o1, in1, clk);
vdff m2 (o2, in2, clk);
endmodule
```

```
module vdff (out, in, clk);
parameter size = 1, delay = 1;
input [0:size-1] in;
input clk;
output [0:size-1] out;
reg [0:size-1] out;
always @(posedge clk)
        # delay out = in;
endmodule
```

Verilog HDL层次化结构
--覆盖模块参数值

```
module annotate;
defparam
        top.m1.size = 5,
        top.m1.delay = 10,
        top.m2.size = 10,
        top.m2.delay = 20;
endmodule
```

模块annotate有defparam语句，其覆盖top模块中例化m1和m2的size和delay参数的值。模块top和annotate都将被认为顶层模块。

Verilog HDL层次化结构
--覆盖模块参数值

通过列表顺序分配参数值

采用这种方式分配参数，其分配的顺序应该和模块内声明参数的顺序一致。当使用这种方法时，没有必要为模块内的所有参数分配值。

然而，不可能跳过一个参数。因此，给模块内声明参数的子集分配值的时候，组成这个子集的参数声明将在剩余参数声明的前面。

Verilog HDL层次化结构
--覆盖模块参数值

一个可选的方法是，给所有的参数分配值，但是对那些不需要新值的参数使用默认的值（即与模块定义内的参数声明中所分配的值一样）。

Verilog HDL层次化结构
--覆盖模块参数值

按顺序分配参数值Verilog HDL描述的例子

```
module tb1;
wire [9:0] out_a, out_d;
wire [4:0] out_b, out_c;
reg [9:0] in_a, in_d;
reg [4:0] in_b, in_c;
reg clk;
// 测试平台和激励生成代码...
// 通过列表顺序对带有参数值分配的四个vdff例化
// mod_a 有新的参数值，size=10 and delay=15
```

Verilog HDL层次化结构
--覆盖模块参数值

```
// mod_b 为默认的参数值(size=5, delay=1)
// mod_c 有默认的参数值size=5和新的参数值delay=12
// 为了改变延迟的值，需要说明默认值。
// mod_d 有新的参数值size=10，延迟为默认参数值
    vdff #(10,15) mod_a (.out(out_a), .in(in_a), .clk(clk));
    vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
    vdff #(5,12) mod_c (.out(out_c), .in(in_c), .clk(clk));
    vdff #(10) mod_d (.out(out_d), .in(in_d), .clk(clk));
endmodule
```

Verilog HDL层次化结构
--覆盖模块参数值

```
module vdff (out, in, clk);
parameter size=5, delay=1;
output [size-1:0] out;
input [size-1:0] in;
input clk;
reg [size-1:0] out;
always @(posedge clk)
        #delay out = in;
endmodule
```

注：

不能覆盖本地参数值。因此，不能将其作为覆盖参数值列表的一部分。

Verilog HDL层次化结构
--覆盖模块参数值

对于本地参数处理的Verilog HDL描述的例子

```
module my_mem (addr, data);
parameter addr_width = 16;
localparam mem_size = 1 << addr_width;
parameter data_width = 8;
...
endmodule
module top;
...
my_mem #(12, 16) m(addr,data);
endmodule
```

Verilog HDL层次化结构
--覆盖模块参数值

在本例中，addr_width的值分配了12，data_width的值分配了16。由于列表的顺序，不能明确的为mem_size分配值，由于声明表达式，mem_size的值分配为4096。

覆盖模块参数值
--通过名字分配参数的值

通过名字分配参数，是将参数的名字和它新的值明确的进行连接。参数的名字是被例化模块内所指定参数的名字。

- 当使用这种方法的时候，不需要给所有参数分配值。只指定需要分配新值的参数。

覆盖模块参数值
--通过名字分配参数值的例子

```
module tb2;
wire [9:0] out_a, out_d;
wire [4:0] out_b, out_c;
reg [9:0] in_a, in_d;
reg [4:0] in_b, in_c;
reg clk;
// 测试平台和激励生成代码...
// 通过名字分配带有参数值的四个例化vdff
// mod_a有新的参数size=10和delay=15
```

覆盖模块参数值
--通过名字分配参数值的例子

```
// mod_b有默认的参数值 (size=5, delay=1)
// mod_c有默认的参数值size=5和新的参数值delay=12
// mod_d有一个新的参数值size=10，延迟保持它的默认参数值

vdff #(.size(10),.delay(15)) mod_a (.out(out_a),.in(in_a),.clk(clk));

vdff mod_b (.out(out_b),.in(in_b),.clk(clk));

vdff #(.delay(12)) mod_c (.out(out_c),.in(in_c),.clk(clk));

vdff #(.delay( ),.size(10) ) mod_d (.out(out_d),.in(in_d),.clk(clk));

endmodule
```

覆盖模块参数值
--通过名字分配参数值的例子

```
module vdff (out, in, clk);
parameter size=5, delay=1;
output [size-1:0] out;
input [size-1:0] in;
input clk;
reg [size-1:0] out;
always @(posedge clk)
        #delay out = in;
endmodule
```

相同顶层模块中，在例化模块时，使用参数重新定义的不同类型，这样做是合法的。

覆盖模块参数值
--不同参数定义类型混合使用的例子

```
module tb3;
// 声明和代码
// 使用位置参数例化和名字参数例化的混合声明是合法的
   vdff #(10, 15) mod_a (.out(out_a), .in(in_a), .clk(clk));
   vdff mod_b (.out(out_b), .in(in_b), .clk(clk));
   vdff #(.delay(12)) mod_c (.out(out_c), .in(in_c), .clk(clk));
endmodule
```

注：不允许在一个例化模块中，同时使用两种混合参数分配的方法，比如：

```
    vdff #(10, .delay(15)) mod_a (.out(out_a), .in(in_a), .clk(clk));
```

这个描述是非法的。

Verilog HDL层次化结构
--端口

端口提供了内部互联硬件描述的模块和原语的构成。比如

- 模块A可以例化模块B，通过适当的端口连接到模块A。
- 这些端口的名字可以不同于在模块B内所指定的内部网络和变量的名字。

Verilog HDL层次化结构
--端口

在顶层模块内的每个模块的声明中，用于端口列表中的每个端口的引用可以是：

- 一个简单的标识符或者转义标识符。
- 在模块内声明的一个向量的比特选择。
- 上面形式的并置。

Verilog HDL层次化结构
--端口

端口表达式是可选择的。这是由于定义的端口可能不会连接到模块内部。所以，一旦定义了端口，不能使用相同的名字定义其它端口。在例化模块内的端口声明，可以是明确的或者隐含的。

Verilog HDL层次化结构
--端口

端口声明

- 如果端口声明中包含一个网络或者变量类型，则将端口看作是完全的声明。
- 如果在一个变量或者网络数据类型声明中再次声明端口，则会出现错误。由于这个原因，端口的其它内容也应该在这样一个端口声明中进行声明，包括：有符号和范围定义（如果需要的话）。

Verilog HDL层次化结构
--端口

- 如果端口声明中不包含一个网络或者变量类型，则可以在变量或者网络数据类型声明中再次声明端口。
- 如果将网络或者变量声明为一个向量，则在一个端口中的两次声明中应该保持一致。一旦在端口定义中使用了该名字，则不允许在其它端口声明或者在数据类型声明中再次进行声明该名字。
- 实现可能限制一个模块定义中的最大端口数目，但是至少是256。

Verilog HDL层次化结构
--端口

端口不同声明方式Verilog HDL描述的例子

```
module test(a,b,c,d,e,f,g,h);
input [7:0] a;                  //没有明确的定义-网络a是无符号的
input [7:0] b;
input signed [7:0] c;
input signed [7:0] d;      //明确的网络定义-网络d是有符号的
output [7:0] e;                //没有明确的网络定义-网络e是无符号的
output [7:0] f;
output signed [7:0] g;
```

Verilog HDL层次化结构
--端口

```
output signed [7:0] h;      //明确的网络定义-网络h是有符号的
wire signed [7:0] b;        //从网络声明中，端口b继承有符号属性.
wire [7:0] c;               //从端口中，网络c继承有符号属性.
reg signed [7:0] f;         //从寄存器声明中，端口f继承有符号属性.
reg [7:0] g;                //从端口中，寄存器g继承有符号属性
endmodule
```

Verilog HDL层次化结构
--端口

```
module complex_ports ({c,d}, .e(f));
        //网络{c,d}接收到第一个端口比特位.
        //在模块内声明了名字'f'.
        //在模块外声明了名字'e'.
        //不能使用第一个端口命名的端口连接.
module split_ports (a[7:4], a[3:0]);
        //第一个端口是'a'的高四位.
        //第二个端口是'a'的底四位.
        //由于a是部分选择，不能使用命名的端口连接
```

Verilog HDL层次化结构
--端口

```
module same_port (.a(i), .b(i));
        //在模块内声明名字'i'为输入端口.
        //声明名字'a'和'b'用于端口连接.
module renamed_concat (.a({b,c}), f, .g(h[1]));
       // 在模块内声明名字'b', 'c', 'f', 'h'.
       // 声明名字'a', 'f', 'g'用于端口连接.
       // 能用于命名的端口连接.
module same_input (a,a);
       input a;       // 这是有效的. 将输入绑定到一起.
module mixed_direction (.p({a, e}));
       input a;      // p包含所有输入和输出的方向.
       output e;
```

Verilog HDL层次化结构
--端口

前面例子，端口定义的另一种形式：

```
module test (
input [7:0] a,
input signed [7:0] b, c, d,  // 可以一次声明多个相同属性的端口.
output [7:0] e,              // 必须在一个声明中，声明每个属性.
output reg signed [7:0] f, g,
output signed [7:0] h) ;
                             // 在模块的其它地方重新声明模块的
                                任何端口都是非法的.
endmodule
```

端口
--通过列表顺序连接模块例化

当使用这种方法例化模块的时候，端口连接的顺序和定义模块内端口的顺序相一致。

端口
--通过列表顺序连接模块例化的例子

```
module topmod;
wire [4:0] v;
wire a,b,c,w;
modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
inout wa, wb;
input c, d;

 tranif1 g1 (wa, wb, cinvert);
 not #(2, 6) n1 (cinvert, int);
 and #(6, 5) g2 (int, c, d);
endmodule
```

模块modB内定义的端口wa，连接到topmod模块的比特选择v[0]。

- ✓ 端口wb连接到v[3]。
- ✓ 端口c连接到w。
- ✓ 端口d连接到v[4]。

在仿真时，modB的例化b1，首先激活and门g2，在int上产生一个值。这个值触发not门n1，在cinvert上产生输出，然后激活tranif1门g1。

端口
--通过名字连接模块例化

另一种将模块端口和例化模块端口连接的方法是通过名字。

端口
--通过名字连接模块例化的例子1

例子

```
ALPHA instance1 (.Out(topB),.In1(topA),.In2());
```

- 例化模块将其信号topA和topB连接到模块ALPHA所定义的In1和Out端口。
- ALPHA提供的端口中，至少其中一个端口没有使用，该端口的名字为In2。
- 该例化中，没有提到例化中没有使用的其它端口。

端口
--通过名字连接模块例化的例子2

```
module topmod;
        wire [4:0] v;
        wire a,b,c,w;
        modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
        inout wa, wb;
        input c, d;
        tranif1 g1(wa, wb, cinvert);
        not #(6, 2) n1(cinvert, int);
        and #(5, 6) g2(int, c, d);
endmodule
```

Verilog HDL层次化结构
--端口

通过名字连接Verilog HDL描述的不合法例子。

```
module test;
a ia (.i (a), .i (b),    //非法连接输入端口两次.
.o (c), .o (d),          //非法连接输出端口两次.
.e (e), .e (f));         //非法连接输入输出端口两次.
endmodule
```

端口
--端口连接中的实数

real数据类型不能直接连接到端口上，应该采用间接的方式进行连接。

- 系统函数$realtobits和$bitstoreal用于在模块端口之间传递比特位。

端口
--通过系统函数传递实数的例子

```
module driver (net_r);
    output net_r;
    real r;
    wire [64:1] net_r = $realtobits(r);
endmodule

module receiver (net_r);
     input net_r;
     wire [64:1] net_r;
     real r;
     initial assign r = $bitstoreal(net_r);
endmodule
```

端口
--端口连接规则

一个模块的端口可看作两个条目（比如：网络、寄存器和表达式）之间的链路或者连接。即：

- 一个内部到模块实例
- 或者一个外部到模块实例

端口
--端口连接规则

下面的端口连接规则给出了条目通过端口接收的值（内部条目用于输入，外部条目用于输出）。

- 一个声明为input（output），但是用于output（input）或者inout，应该强制为inout。如果没有强制到inout，将产生警告信息。

端口
--端口连接规则

- 规则一
  - ✓ 一个input或者inout端口是网络类型。
- 规则二
  - ✓ 从源端口到目的端口的分配是连续的。用于input端口的分配是没有降低强度的晶体管连接。在一个分配中，只有目的端口是网络或者结构化的网络表达式。

端口
--端口连接规则

一个结构化的网络表达式是一个端口表达式，其操作数可以是：

- 一个标量网络
- 一个向量网络
- 一个向量网络的常数比特位选择
- 一个向量网络的部分选择
- 结构化网络表达式的并置

端口
--端口连接规则

下面的外部条目不能连接到模块的output或者inout端口

- 变量
- 不同于下面的其它表达式
  - ✓ 一个标量网络
  - ✓ 一个向量网络
  - ✓ 一个向量网络的常数比特位选择
  - ✓ 一个向量网络的部分选择
  - ✓ 上述表达式的并置

端口
--端口连接规则

规则三

- ✓ 如果一个端口两侧的任何一个网络类型是uwire，如果没有将网络合并为一个网络，则会出现警告信息。
- ✓ 如果通过一个模块端口将不同的网络类型连接到了一起，则两侧的端口会是同一种类型。下表给出了不同网络类型连接后最后类型的确定列表。

端口
--端口连接规则

内部网络	外部网络								
	wire, tri	wand, triand	wor, trior	trireg	tri0	tri1	uwire	supply0	supply1
wire，tri	ext	ext	ext	ext	ext	ext	ext	ext	ext
wand, triand	int	ext	ext warn	ext warn	ext warn	ext warn	ext warn	ext	ext
wor, trior	int	ext warn	ext	ext warn	ext warn	ext warn	ext warn	ext	ext
trireg	int	ext warn	ext warn	ext	ext	ext	ext warn	ext	ext
tri0	int	ext warn	ext warn	int	ext	ext warn	ext warn	ext	ext

端口
--端口连接规则

tri1	int	ext warn	ext warn	int	ext warn	ext	ext warn	ext	ext
uwire	int	int warn	int warn	int warn	int warn	int warn	ext	ext	ext
supply0	int	int	int	int	int	int	int	ext	ext warn
supply1	int	int	int	int	int	int	int	ext warn	ext

注释：

- ext表示将使用的外部网络类型。
- int表示将使用内部网络类型。
- warn表示产生一个警告

端口
--端口连接规则

规则四

- ✓ 符号属性不可以跨越层次。为了让符号类型跨越层次，在不同的层次级的对象声明中，使用signed关键字。

Verilog HDL层次化结构
--生成结构

Verilog HDL生成结构用于在一个模型中，有条件或者成倍的例化生成模块。

- 一个例化模块是一个或者多个模块条目的集合。
- 一个生成模块不包含端口声明、参数声明、指定模块或者specparam声明。
- 在一个生成块中，允许包含所有的模块条目，包括其它生成结构。生成结构提供了通过参数值影响模型结构的能力。这也允许更简单的描述带有重复结构的模块，这使得可以递归的实现模块例化。

Verilog HDL层次化结构
--生成结构

Verilog HDL提供了两种生成结构的类型：

- 循环生成结构

  允许单个生成模块可以被多次例化到一个模型。

- 条件生成结构

  包含if-generate或者case-generate结构。从一堆可以选择的生成模块中例化出最多一个生成模块。

Verilog HDL层次化结构
--生成结构

术语生成策略是指确定例化哪个生成模块或者生成多少个模块的方法。

- 它包含出现在一个生成结构中的条件表达式、case表达式和循环控制语句。

Verilog HDL层次化结构
--生成结构

在对模型进行详细说明（elaboration，计算机综合过程的一部分）的过程中，评估生成策略。当分析完HDL后，仿真之前进行详细的说明。

- 详细说明涉及到展开模块例化，计算参数值，解析层次的名字，建立网络连接和准备用于仿真的模型。
- 尽管生成策略的使用和行为语句类似的语法，但是重要的是要承认，在仿真的时候并不执行它们。**在生成策略中的所有表达式必须是常数表达式。**

Verilog HDL层次化结构
--生成结构

对生成结构的详细描述产生0个或者多个生成模块。

- 在某些方面，对一个生成模块的一个例化类似于一个模块的例化。
- 它创建了一个新的层次级。
- 它将模块内的对象、行为结构和模块例化引入到实体中。

Verilog HDL层次化结构
--生成结构

在模块内使用generate和endgenerate关键字定义生成区域。当使用一个生成区域时，在模块内没有语义上的差别。

生成结构
--循环生成结构

一个循环生成结构允许单个生成模块被多次例化。循环生成结构的语法如下：

```
genvar var;
  generate
      for (var=0; var < limit; var=var+1)
      begin: label（生成模块使用的标识符）
          instantiation（例化模块的描述）
      end
  endgenerate
```

生成结构
--循环生成结构

在一个循环生成结构的内部，有一个隐含的localparam声明。这是一个整数参数，它和循环索引变量有相同的名字和类型。在生成模块内该参数的值是当前详细描述中索引变量的值。

- 这个参数可以用于生成模块内的任何地方。在该生成模块中，可以使用带有一个整数值的普通参数。它可以被引用（带有一个参数名字）。

生成结构
--循环生成结构

由于这个隐含的localparam和genvar有相同的名字，任何对循环生成块内名字的引用都是对localparam的一个引用，而不是对genvar的引用。结果是，不可能使用相同的genvar，将其用于两个嵌套的循环生成结构中。

生成结构
--循环生成结构

可以命名或者不命名一个生成结构。他们可以只有一个条目，此时不需要begin/end关键字。即使没有begin/end关键字，其仍然为一个生成模块。

- 如果命名了一个生成模块，它是一组生成模块例化的声明。这个组内的索引值是详细描述期间所假定的值。

生成结构
--无效的循环生成结构的例子

```
module mod_a;
genvar i;
// 不要求"generate", "endgenerate"
for (i=0; i<5; i=i+1) begin:a
        for (i=0; i<5; i=i+1) begin:b
        ... // 错误 –使用"i"或为两个嵌套生成循环的索引
        end
end
endmodule
------
```

生成结构
--无效的循环生成结构的例子

```
module mod_b;
genvar i;
reg a;
for (i=1; i<0; i=i+1) begin: a
... // 错误-- "a"和寄存器类型"a"冲突
end
endmodule
------
```

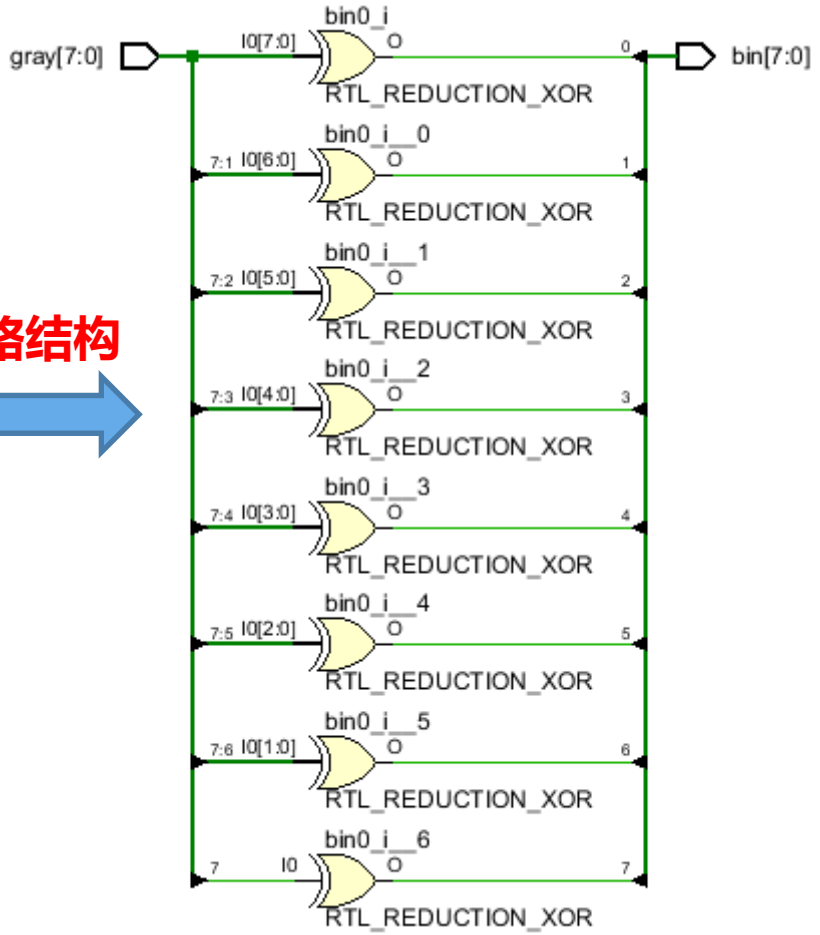
生成结构
--无效的循环生成结构的例子

```
module mod_c;
genvar i;
for (i=1; i<5; i=i+1) begin: a
...
end
for (i=10; i<15; i=i+1) begin: a
... //错误-- "a"和前面的名字冲突
end
endmodule
```

生成结构
--实现格雷码到二进制码转换的例子

```
module gray2bin1 (bin, gray);
    parameter SIZE = 8; //该模块参数化
    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
```

详细描述后的电路结构

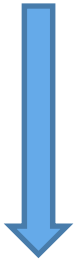


该设计保存在本书配套资源\eda_verilog\gray2bin目录下

生成结构
--生成逐位进位加法器的例子1

```
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;
```

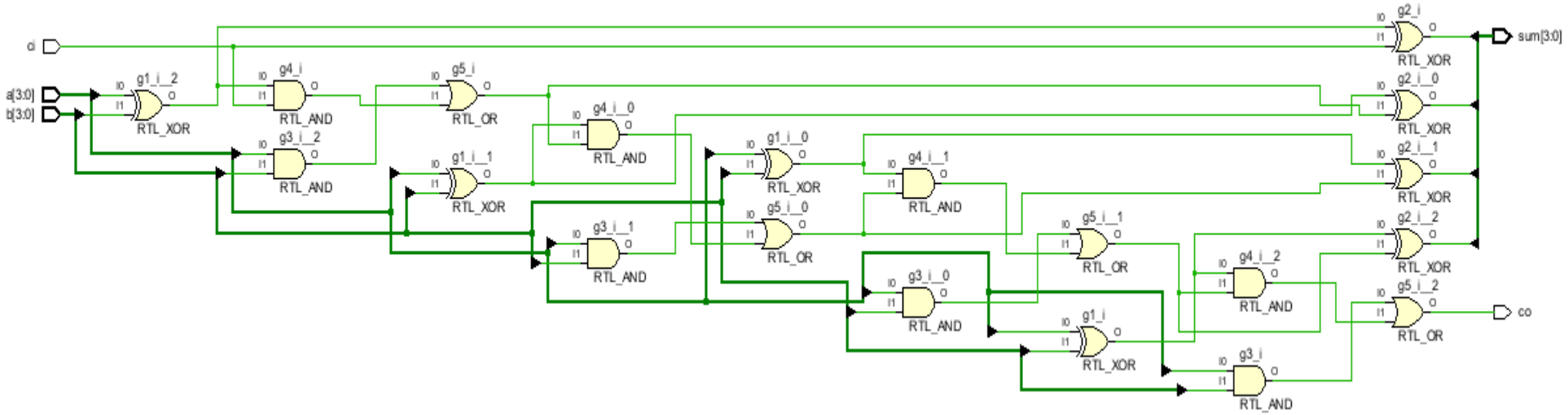
```
    for(i=0; i<SIZE; i=i+1) begin:bit
        xor g1 ( t[1][i], a[i], b[i]);
        xor g2 ( sum[i], t[1][i], c[i]);
        and g3 ( t[2][i], a[i], b[i]);
        and g4 ( t[3][i], t[1][i], c[i]);
        or g5 ( c[i+1], t[2][i], t[3][i]);
    end
  assign co = c[SIZE];
endmodule
```



详细描述后生成的电路结构

该设计保存在本书配套资源\eda_verilog\addergen1目录下

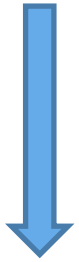
生成结构
--生成逐次进位加法器的例子1



思考与练习：查看每根连线的网络 名字，分析其和生成结构之间的关系

生成结构
--生成逐次进位加法器的例子2

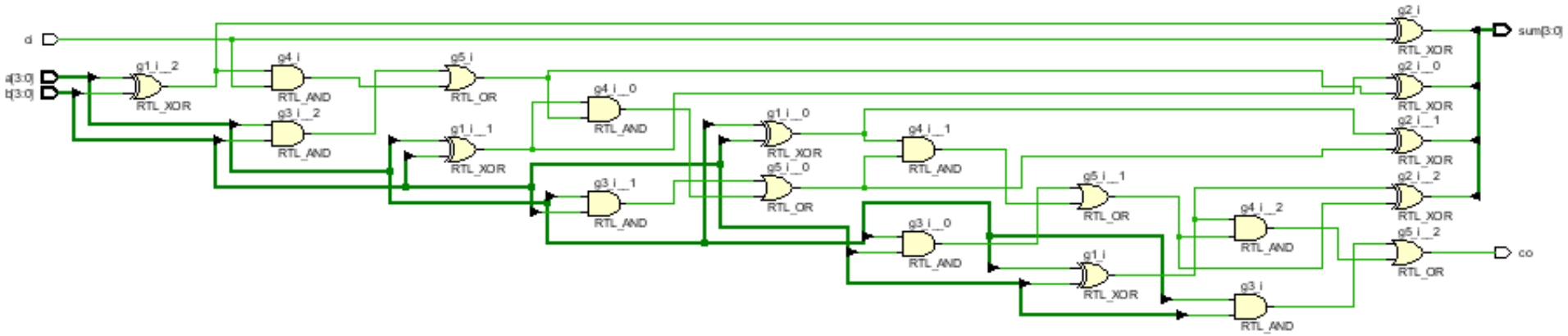
```
module addergen1 (co, sum, a, b, ci);
    parameter SIZE = 4;
    output [SIZE-1:0] sum;
    output co;
    input [SIZE-1:0] a, b;
    input ci;
    wire [SIZE :0] c;
    genvar i;
    assign c[0] = ci;
    for(i=0; i<SIZE; i=i+1) begin:bit
        wire t1, t2, t3;
        xor g1 ( t1, a[i], b[i]);
        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
    assign co = c[SIZE];
endmodule
```



详细描述后生成的电路结构

该设计保存在本书配套资源\eda_verilog\addergen1目录下

生成结构
--生成逐次进位加法器的例子2



思考与练习：查看每根连线的网络 名字，分析其和生成结构之间的关系

生成结构
--多层生成模块的例子

```
parameter SIZE = 2;
  genvar i, j, k, m;
  generate
    for (i=0; i<SIZE; i=i+1) begin:B1 // 范围B1[i]
      M1 N1(); // 例化B1[i].N1
        for (j=0; j<SIZE; j=j+1) begin:B2 // 范围 B1[i].B2[j]
          M2 N2(); // 例化B1[i].B2[j].N2
            for (k=0; k<SIZE; k=k+1) begin:B3 // 范围 B1[i].B2[j].B3[k]
              M3 N3(); // 例化B1[i].B2[j].B3[k].N3
            end
```

生成结构
--多层生成模块的例子

```
        end
        if (i>0) begin:B4 // 范围B1[i].B4
            for (m=0; m<SIZE; m=m+1) begin:B5 //范围B1[i].B4.B5[m]
                M4 N4(); //例化B1[i].B4.B5[m].N4
            end
    end
  end
endgenerate
```

生成结构
--条件生成结构

条件生成结构包含：if-generate和case-generate。

- **在工具详细描述的过程中，基于给出的常数表达式，从可替换的生成模块集合中，选择产生最多一个生成模块。**
- **如果存在选择需要生成的块，则将其例化到模型中。**

生成结构
--包含case语句的有条件生成结构

```
generate
    case (constant_expression)
      value: begin: label_1
                code
             end
      value: begin: label_2
                code
             end
      default: begin: label_3
                code
             end
    endcase
endgenerate
```

其中：

- ✓ constant_expression为常数表达式。
- ✓ value为case的取值。
- ✓ label为标号。

生成结构
--包含if-else语句的有条件生成结构

```
generate
    if (condition) begin: label_1
      code;
    end
    else if (condition) begin: label_2
      code;
    end
    else begin: label_3
      code;
    end
endgenerate
```

其中：

- ✓ condition为条件表达式。
- ✓ value为case的取值。
- ✓ label为标号。

生成结构
--包含if-else语句的有条件生成结构的例子

```
module test;
parameter p = 0, q = 0;
wire a, b, c;
//-------------------------------------------------------
// 代码或者生成u1.g1例化或者没有生成例化.
// u1.g1例化下面的一个门{and, or, xor, xnor) ，根据条件
// {p,q} == {1,0}, {1,2}, {2,0}, {2,1}, {2,2}, {2, default}
//-------------------------------------------------------
```

生成结构
--包含if-else语句的有条件生成结构的例子

```
if (p == 1)
    if (q == 0)
        begin : u1          // If p==1和q==0, 则例化
            and g1(a, b, c); // AND的层次名字为test.u1.g1
        end
    else if (q == 2)
        begin : u1          // If p==1和q==2, 则例化
            or g1(a, b, c); // OR的层次名字为test.u1.g1
        end
     // 添加"else"结束“(q == 2)" 的描述
     else ;                 // 如果p==1和q!=0或2, 则没有例化
```

生成结构
--包含if-else语句的有条件生成结构的例子

```
else if (p == 2)
    case (q)
    0, 1, 2:
        begin : u1       // 如果p==2和q==0,1, 或者2,则例化
            xor g1(a, b, c);// XOR层次名字为test.u1.g1
        end
     default:
        begin : u1       // If p==2 and q!=0,1或者 2，则例化
            xnor g1(a, b, c);// XNOR的层次名字为test.u1.g1
        end
  endcase
endmodule
```

生成结构
--一个参数化乘法器的例子

```
module multiplier(a,b,product);

parameter a_width = 8, b_width = 8;

localparam product_width = a_width+b_width;

        // 不能通过defparam描述或者例化语句直接修改#

input [a_width-1:0] a;

input [b_width-1:0] b;

output [product_width-1:0] product;
```

生成结构
--一个参数化乘法器的例子

```
generate
      if((a_width < 8) || (b_width < 8)) begin: mult
         CLA_multiplier #(a_width,b_width) u1(a, b, product);
         // 例化一个CLA乘法器
      end
      else begin: mult
    WALLACE_multiplier #(a_width,b_width) u1(a, b, product);
     // 例化一个Wallace树乘法器
    end
endgenerate
       // 层次话的例化名字为mult.u1
endmodule
```

生成结构
--一个case生成结构的例子

```
generate
    case (WIDTH)
        1: begin: adder //实现1比特加法器
            adder_1bit x1(co, sum, a, b, ci);
        end
        2: begin: adder //实现2比特加法器
            adder_2bit x1(co, sum, a, b, ci);
        end
        default:
            begin: adder //其它超前进位加法器
                adder_cla #(WIDTH) x1(co, sum, a, b, ci);
            end
    endcase
// 这个层次例化的名字是adder.x1
endgenerate
```

生成结构
--一个for生成结构的例子

```
module dimm(addr, ba, rasx, casx, csx, wex, cke, clk, dqm,
                    data, dev_id);
parameter [31:0] MEM_WIDTH = 16, MEM_SIZE = 8; // in mbytes
input [10:0] addr;
input ba, rasx, casx, csx, wex, cke, clk;
input [ 7:0] dqm;
inout [63:0] data;
input [ 4:0] dev_id;
genvar i;
```

生成结构
--一个for生成结构的例子

```
case ({MEM_SIZE, MEM_WIDTH})
   {32'd8, 32'd16}: // 8M x 16 位宽
begin: memory
      for (i=0; i<4; i=i+1) begin:word
         sms_08b216t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                                  .addr(addr), .rasb(rasx), .casb(casx),
                        .web(wex), .udqm(dqm[2*i+1]), .ldqm(dqm[2*i]),
                        .dqi(data[15+16*i:16*i]), .dev_id(dev_id));
   //层次化例化名字是memory.word[3].p,
   //memory.word[2].p, memory.word[1].p, memory.word[0].p,
   //和任务memory.read_mem
   end
```

生成结构
--一个for生成结构的例子

```
task read_mem;
    input [31:0] address;
    output [63:0] data;
    begin //在sms模块内调用read_mem
        word[3].p.read_mem(address, data[63:48]);
        word[2].p.read_mem(address, data[47:32]);
        word[1].p.read_mem(address, data[31:16]);
        word[0].p.read_mem(address, data[15: 0]);
    end
  endtask
end
```

生成结构
--一个for生成结构的例子

```
{32'd16, 32'd8}: // 16Meg x 8位宽度
begin: memory
    for (i=0; i<8; i=i+1) begin:byte
        sms_16b208t0 p(.clk(clk), .csb(csx), .cke(cke),.ba(ba),
                       .addr(addr), .rasb(rasx), .casb(casx),
                       .web(wex), .dqm(dqm[i]),
                       .dqi(data[7+8*i:8*i]), .dev_id(dev_id));
// 层次话的例化名字memory.byte[7].p,memory.byte[6].p, ... ,
memory.byte[1].p, memory.byte[0].p
// 和任务memory.read_mem
end
```

生成结构
--一个for生成结构的例子

```
task read_mem;
    input [31:0] address;
    output [63:0] data;
    begin //在sms module模块调用read_mem
        byte[7].p.read_mem(address, data[63:56]);
        byte[6].p.read_mem(address, data[55:48]);
        byte[5].p.read_mem(address, data[47:40]);
        byte[4].p.read_mem(address, data[39:32]);
        byte[3].p.read_mem(address, data[31:24]);
        byte[2].p.read_mem(address, data[23:16]);
        byte[1].p.read_mem(address, data[15: 8]);
```

生成结构
--一个for生成结构的例子

```
            byte[0].p.read_mem(address, data[ 7: 0]);
        end
    endtask
end
    // 其存储器情况
endcase
endmodule
```

生成结构
--用于未命名生成块的外部名字

- 尽管可以在层次化的名字中使用没有命名的生成块，但是需要有一个名字。
- 通过这个名字外部的接口可以指向该生成块。

生成结构
--用于未命名生成块的外部名字

对于一个给定范围内的每个生成结构，分配了一个数字。

- 在这个范围内，首先出现以文字形式出现的结构，其数字是1。
- 对于该范围的每个子生成结构其值递增1。
- 对于所有未命名的生成块，将其命名为genblk<n>。n为分配给结构的数字。如果这个名字和明确声明的名字冲突，则在数字前一直加0，直到没有冲突为止。

生成结构
--未命名生成块的例子

```
module top;
    parameter genblk2 = 0;
    genvar i;
    // 下面的生成块有隐含的名字genblk1
    if (genblk2) reg a; // top.genblk1.a
    else reg b; // top.genblk1.b
    //下面的生成块有隐含的名字genblk02，因为genblk2已经声
      明为标识符了
    if (genblk2) reg a; // top.genblk02.a
    else reg b; // top.genblk02.b
```

生成结构
--未命名生成块的例子

```
//下面的生成块有隐含的名字genblk3，但是有明确的名字g1
for (i = 0; i < 1; i = i + 1) begin : g1 //块的名字
        //下面的生成块有隐含名字genblk1
       // as the first nested scope inside of g1
     if (1) reg a; // top.g1[0].genblk1.a
end
//下面的生成块有隐含的名字genblk4，由于它输入top内的第四个
  生成块.
//如果没有明确命名为g1，前面的生成块命名为genblk3
for (i = 0; i < 1; i = i + 1)
   //下面的生成块有隐含名字genblk1
```

生成结构
--未命名生成块的例子

```
    //作为genblk4内第一个嵌套的生成块
    if (1) reg a;       // top.genblk4[0].genblk1.a
    //下面的生成块有隐含的名字genblk5
    if (1) reg a;       // top.genblk5.a
endmodule
```

Verilog HDL层次化结构
--层次化的名字

在Verilog HDL描述中，每个标识符应该有一个唯一的层次路径名字。

- 模块层次和模块内所定义的条目，比如：任务和命名块定义了这些名字。
- 名字的层次看作一个树形结构。在这个树形结构中，每个模块例化生成块例化、生成块例化、任务、函数或者命名的begin-end或者fork-join块，在一个特殊的树形分支上定义了一个新的层次级或者范围。

Verilog HDL层次化结构
--层次化的名字

一个设计描述包含一个或多个顶层模块。每个这样的模块构成一个名字层次的顶层。

- 在一个设计描述或者描述中，这个根或者并行根模块构成了一个或多个层次。
- 在命名的块和任务/函数内的命名块创建了新的分支。
- 没有命名的块是例外。它们所创建的分支只能从块内和由块例化的层次内看到。

Verilog HDL层次化结构
--层次化的名字

注：

- 对于简单的标识符可以一个字符或者下划线开始，必须包含有一个字符（a~z、Z~Z、0-9），这些字符之间不允许空格。
- 层次化名字通过“.”符号进行连接。

层次化的名字
--模块例化和命名模块的例子

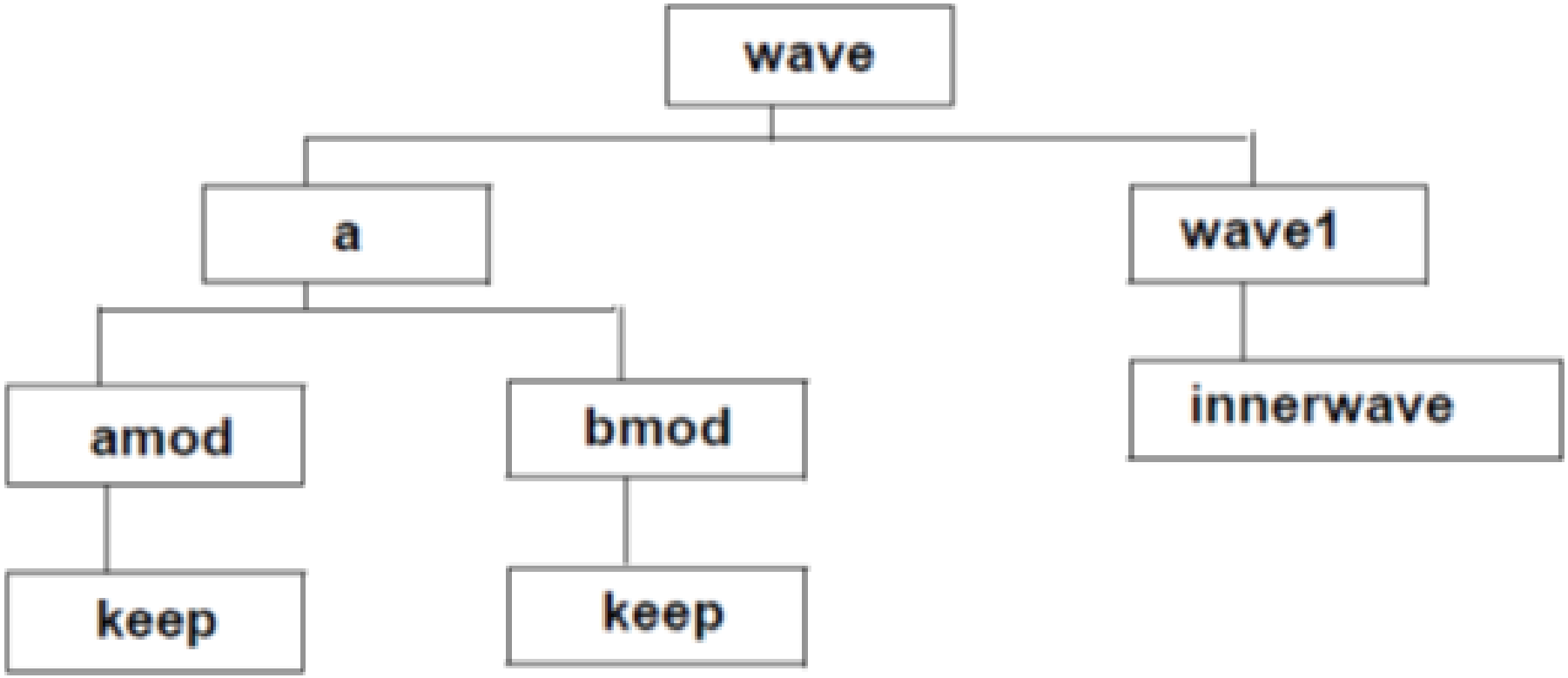
```
module mod (in);
input in;
always @(posedge in) begin : keep
reg hold;
        hold = in;
end
endmodule
module cct (stim1, stim2);
input stim1, stim2;
//例化mod
        mod amod(stim1), bmod(stim2);
endmodule
```

层次化的名字
--模块例化和命名模块的例子

```
module wave;
reg stim1, stim2;
cct a(stim1, stim2); // 例化cct
initial begin :wave1
        #100 fork :innerwave
                reg hold;
             join
        #150 begin
                stim1 = 0;
             end
          end
endmodule
```

Verilog HDL层次化结构
--层次化的名字

下图给出了该Verilog HDL所描述模型中的层次。



Verilog HDL层次化结构
--层次化的名字

下面给出了代码中，所有定义对象的层次形式列表

wave
wave.stim1
wave.stim2
wave.a
wave.a.stim1
wave.a.stim2
wave.a.amod
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
wave.a.bmod
wave.a.bmod.in
wave.a.bmod.keep
wave.a.bmod.keep.hold
wave.wave1
wave.wave1.innerwave
wave.wave1.innerwave.hold

Verilog HDL层次化结构
--层次化的名字

对层次化名字的引用允许自由访问层次内任何级对象的数据。如果知道一个条目唯一的层次化路径名字，则可以从描述中的任何位置采样或者修改它的值。

Verilog HDL层次化结构
--层次化的名字

在层次化结构中修改值Verilog HDL描述的例子

```
begin
      fork :mod_1
          reg x;
          mod_2.x = 1;
      join
      fork :mod_2
           reg x;
           mod_1.x = 0;
      join
end
```

Verilog HDL层次化结构
--向上名字引用

模块或者模块例化的名字对于识别模块和它在层次中的位置已经足够了。一个更低层的模块，能引用层次中该模块上层模块内的条目。

如果知道高层模块或者它的例化名字，则可以引用它的名字。对于任务、函数、命名的块和生成块，Verilog HDL将在检查模块内的名字，直到找到名字或者到达了层次的根部。

Verilog HDL层次化结构
--向上名字引用

向上名字引用的格式如下：

scope_name.item_name

其中：

- scope_name为一个模块例化名字或者一个生成块的名字。

Verilog HDL层次化结构
--向上名字引用的例子

```
module a;
integer i;
      b a_b1();
endmodule

module b;
integer i;
     c b_c1(), b_c2();
initial // 向下的路径引用，i的两个复制
          #10 b_c1.i = 2;  // a.a_b1.b_c1.i, d.d_b1.b_c1.i
endmodule
```

Verilog HDL层次化结构
--向上名字引用

```
module c;
integer i;
initial begin              // i的本地名字应用的四个复制:
        i = 1;             // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
                           // d.d_b1.b_c1.i, d.d_b1.b_c2.i
        b.i = 1;           // i的向上路径引用的两个复制
                           // a.a_b1.i, d.d_b1.i
end
endmodule
```

Verilog HDL层次化结构
--向上名字引用

```
module d;
integer i;
      b d_b1();
initial begin      //  i的每个复制的全路径名字引用
       a.i = 1;
       a.a_b1.i = 2;
       a.a_b1.b_c1.i = 3;
       a.a_b1.b_c2.i = 4;
       d.i = 5;
       d.d_b1.i = 6;
       d.d_b1.b_c1.i = 7;
       d.d_b1.b_c2.i = 8;
end
endmodule
```

Verilog HDL层次化结构
--范围规则

在Verilog HDL中，下面的元素定义了一个新的范围：

- 模块
- 任务
- 函数
- 命名块
- 生成块

Verilog HDL层次化结构
--范围规则

在一个范围内，一个标识符只声明一个条目。这个规则意味着下面情况是非法的：

- 在相同的模块内，为两个或者多个变量声明相同的名字，或者命名任务的名字和变量相同。
- 一个逻辑门的例化名字和连接到该逻辑门输出网络的名字相同。
- 对于生成块来说，上面规则也一样适用，而不考虑生成块是否被例化。对于一个条件生成结构内的生成模块，不适用这个规则。

Verilog HDL层次化结构
--范围规则

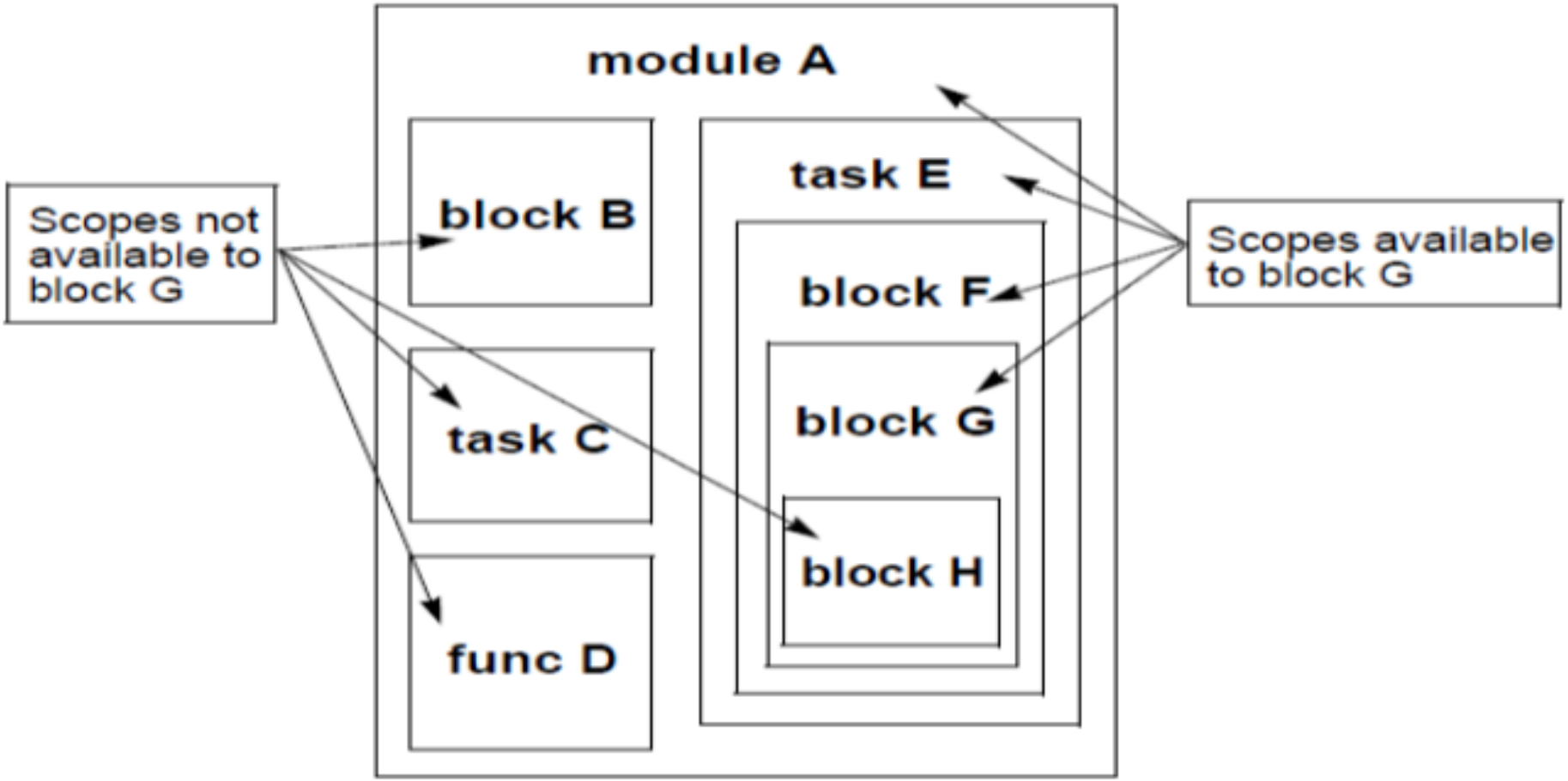
在一个任务、函数、命名块或者生成块内，如果直接引用一个标识符（没有层次化的路径），则应该在任务、函数、命名块、生成块内本地声明，或者在包含任务、函数、命名块、生成块的名字树的相同分支内较高层次的模块、任务、函数、命名块、生成块内声明。

Verilog HDL层次化结构
--范围规则

如果在本地直接声明，则可以使用本地条目；如果没有，则向上查找路径，直到找到该条目的名字或者到达模块边界。这也意味着，在包含模块内的任务和函数可以使用并且修改变量，而不需要通过它们的端口。

Verilog HDL层次化结构
--范围规则

下图内每个长方形表示一个本地范围。



Verilog HDL层次化结构
--范围规则

通过名字访问变量Verilog HDL描述的例子

```
task t;
reg s;
    begin : b
        reg r;
        t.b.r = 0;//这三行访问相同的变量r
        b.r = 0;
        r = 0;
        t.s = 0;// 这两行访问相同的变量s
        s = 0;
    end
endtask
```