1. 目的

本规范的目的是提高书写代码的可读性、可修改性、可重用性，优化代码综合和仿真的结果，指导设计工程师使用VerilogHDL规范代码和优化电路，规范化可编程技术部的FPGA设计输入，从而做到：① 逻辑功能正确，②可快速仿真，③ 综合结果最优（如果是hardware model)，④可读性较好。

2. 范围

本规范涉及Verilog HDL编码风格，编码中应注意的问题，Testbench的编码等。

本规范适用于Verilog model的任何一级（RTL，behavioral, gate_level)，也适用于出于仿真、综合或二者结合的目的而设计的模块。

3. 定义

Verilog HDL： Verilog 硬件描述语言

FSM ： 有限状态机

伪路径 ： 静态时序分析（STA）认为是时序失败，而设计者认为是正确的路径。

4. 规范内容

4.1.Verilog 编码风格

本章节中提到的Verilog编码规则和建议适应于 Verilog model的任何一级（RTL，behavioral, gate_level)，也适用于出于仿真，综合或二者结合的目的而设计的模块。

4.1.1. 命名的习惯

选择有意义的信号和变量名，对设计是十分重要的。命名包含信号或变量诸如出处、有效状态等基本含义，下面给出一些命名的规则。

- 用有意义而有效的名字

有效的命名有时并不是要求将功能描述出来，如

```
For ( I = 0; I < 1024; I = I + 1 )
    Mem[I] <= 32’ b0;
```

For 语句中的循环指针I 就没必要用loop_index作为指针名。

- 用连贯的缩写

长的名字对书写和记忆会带来不便，甚至带来错误。采用缩写时应注意同一信号在模块中的一致性。缩写的例子如下：

Addr	***address***
Pntr	***pointer***
Clk	***clock***
	reset

• 用最右边的字符下划线表示低电平有效，高电平有效的信号不得以下划线表示，短暂的引擎信号建议采用高有效。

如： ***Rst_ ，Trdy_，Irdy_， Idsel.***

•大小写原则

名字一般首字符大写，其余小写（但parameter, integer 定义的数值名可全部用大写），两个词之间要用下划线连接。

如： ***Packet_addr, Data_in, Mem_wr Mem_ce_***

•全局信号名字中应包含信号来源的一些信息。

如：D_addr[7:2]，这里的“D”指明了地址是解码模块(Decoder module)中的地址。

•同一信号在不同层次应保持一致性。

•自己定义的常数、类型等用大写标识

如：***parameter CYCLE=100；***

•避免使用保留字

如：***in，out，x，z***等不能够做为变量、端口或模块名

•添加有意义的后缀，使信号名更加明确，常用的后缀如下：

后缀	意义
_Clk	时钟信号
_next	寄存前的信号
_z	连到三态输出的信号
_f	下降沿有效的寄存器
_xi	芯片原始输入信号
_xo	芯片原始输出信号
_xod	芯片的漏极开路输出
_xz	芯片的三态输出
-xbio	芯片的双向信号
_reg	寄存后的信号

•一个module一个文件，且文件名能与module名对应起来

4.1.2. Modules

•顶层模块应只是内部模块间的互连。

Verilog设计一般都是层次型的设计，也就是在设计中会出现一个或多个模块，模块间的调用在所难免。可把设计比喻成树，被调用的模块就是树叶，没被调用的模块就是树根，那么在这个树根模块中，除了内部的互连和模块的调用外，尽量避免再做逻辑，如不能再出现对reg变量赋值等。这样做的目的是为了更有效的综合，因为在顶层模块中出现中间逻辑，Synopsys 的design compiler 就不能把子模块中的逻辑综合到最优。

•每一个模块应在开始处注明文件名、功能描述、引用模块、设计者、设计时间及版权信息等。代码中的所有说明、注释必须均为英文。需要特别说明的是，必须对***Revision History***要格外重视，必须将每次版本修改的信息按照时间一一详加叙述，以保持版本的可读性与继承性。

如： ***/* ＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝ ****

Filename ： RX_MUX.v

Author ：

Description ：

Called by ：Top module

Revision History ：99-08-01

Revision 1.0

Email ： M@sz.huawei.com.cn

Company ： Huawei Technology .Inc

**** ==========*/**

•不要对Input进行驱动, 在module 内不要存在没有驱动的信号，更不能在模块端口中出现没有驱动的输出信号，避免在elaborate和compile时产生warning，干扰错误定位。

•每行应限制在80个字符以内，以保持代码的清晰、美观和层次感。

一条语句占用一行，如果较长（超出80个字符）则要换行。

•电路中调用的 module名用Uxx标示， Cell名用Vxx标识。向量大小表示要清晰，采用基于名字（name_based）的调用而非基于顺序的（order_based）。另外，调用模块时调用关系要写全，否则，在用synplify综合过程会出warning。

```
Instance    UInstance2(
            .DataOut    (DOUT    ),
            .DataIn     (DIN     ),
            .Cs_        (Cs_     )
);
```

•用一个时钟的上沿或下沿采样信号，不能一会儿用上沿，一会儿用下沿。如果既要用上沿又要用下沿，则应分成两个模块设计。建议在顶层模块中对Clock做一非门，在层次模块中如果要用时钟下沿就可以用非门产生的Posedge Clk_， 这样的好处是在整个设计中采用同一种时钟沿触发，有利于综合。

•在模块中增加明了的英文注释。

对信号、参量、引脚、模块、函数及进程等加以说明，便于阅读与维护。

• Module 名要用大写标示，且应与文件名保持一致。

如：Module ***DFF_ASYNC_RST***(

Reset,

Clk,

Data,

Qout

);

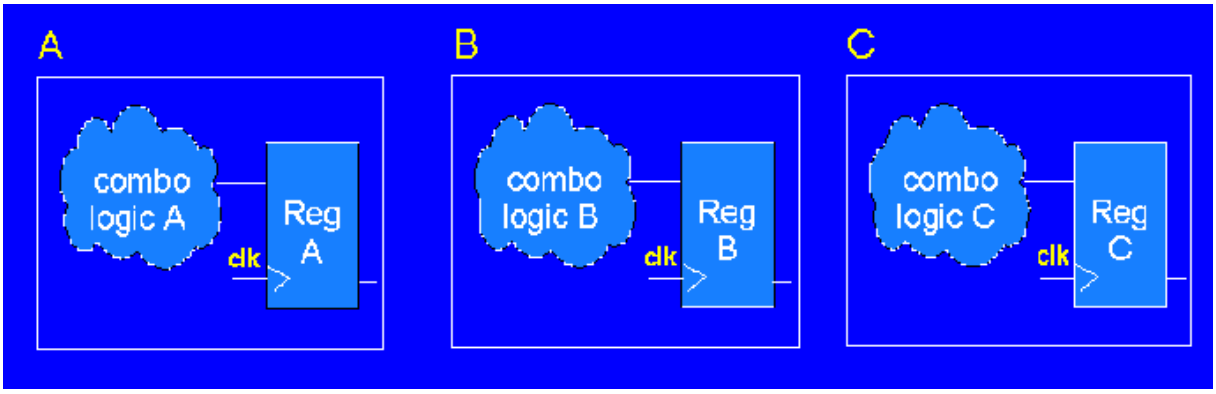
•严格芯片级模块的划分

只有顶层包括IO引脚(pads)，中间层是时钟产生模块、JTAG、芯片的内核(CORE)，这样便于对每个模块加以约束仿真，对时钟也可以仔细仿真。

•模块输出寄存器化

对所有模块的输出加以寄存（如图1），使得输出的驱动强度和输入的延迟可以预测，从而使得模块的综合过程更简单。

- 输出驱动的强度都等于平均的触发器驱动强度
- 输入延迟始终等于通过触发器的路径，近于相等



（图 1）

•将关键路径逻辑和非关键路径逻辑放在不同模块

保证DC可以对关键路径模块实现速度优化，而对非关键路径模块实施面积优化。在同一模块DC无法实现不同的综合策略。

•将相关的组合逻辑放在同一模块

有助于DC对其进行优化，因为DC通常不能越过模块的边界来优化逻辑

•ultraedit中的tab键设置为4个空格键

•输入输出的端口定义分行写，一行定义一个输入输出，顶格对齐

•module头中的括号内的输入输出的定义按输入输出分开，与括号对齐

•module用到的变量统一在输入输出的定义之后，按wire和reg型分开定义，并对重要的信号加注释说明

•module主体以//module begin作为起始标识

•子模块的调用顶格对齐；子模块中的输入输出的端口信号的名字尽量与调用的名字一致

•if...else一一对应，若无else时，加一个空语句

4.1.3. Net and Register

• 一个reg变量只能在一个always语句中赋值。

•向量有效位顺序的定义一般是从大数到小数。

尽管定义有效位的顺序很自由，但如果采用毫无规则的定义势必会给作者和读代码的人带来困惑，如Data[-4：0]，则LSB[0][-1][-2][-3][-4]MSB，或Data[0：4]，则LSB[4][3][2][1][0]MSB，这两种情况的定义都不太好，推荐Data[4：0]这种格式的定义。

•对net和register类型的输出要做声明。

如果一个信号名没做声明，Verilog将假定它为一位宽的wire变量。

•无用信号不要引入module内部，避免在elaborate和compile时产生warningr类型的输出要做声明。

如果一个信号名没做声明，Verilog将假定它为一位宽的wire变量。

4.1.4. Expressions

•用括号来表示执行的优先级

尽管操作符本身有优先顺序，但用括号来表示优先级对读者更清晰，更有意义。

If ((alpha < beta) && (gamma >= delta)).... 比下面的表达更合意

If (alpha < beta && gamma >= delta)...

•用一个函数(function)来代替表达式的多次重复

如果代码中发现多次使用一个特殊的表达式，那么就用一个函数来代替，这样在以后的版本升级时更便利，这种概念在做行为级的代码设计时同样使用，经常使用的一组描述可以写到一个任务(task)中。

4.1.5. IF 语句

•向量比较时，比较的向量要相等。

当比较向量时，verilog将对位数小的向量做 0 扩展以使它们的长度相匹配，它的自动扩展为隐式的。建议采用显示扩展，这个规律同样适用于向量同常量的比较。

```
Reg     Abc  [7:0];
Reg     Bca  [3:0];
.....
If (Abc = = {4’ b0, Bca})begin
.......
If (Abc = = 8’ b0) begin
```

•每一个If 都应有一个else 和它相对应

在做硬件设计时，常要求条件为真时执行一种动作而条件为假时执行另一动作，即使认为条件为假不可能发生。没有else可能会使综合出的逻辑和RTL级的逻辑不同。如果条件为假时不进行任何操作，则用一条空语句。

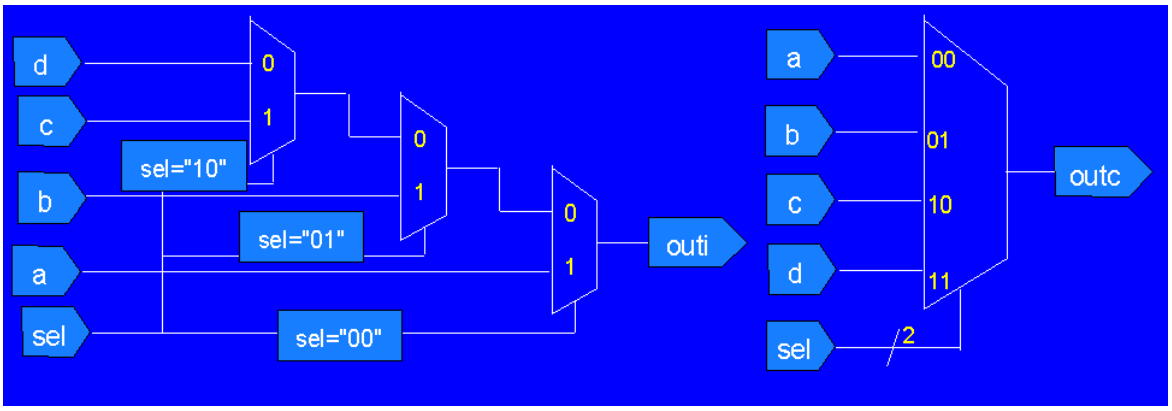
```
always @(Cond)
begin
   if (Cond)
     DataOut <= DataIn;
   Else ： ;
end
```

以上语句DataOut会综合成锁存器.

•应注意If ..else if ...else if ...else 的优先级

4.1.6 case 语句

•case语句通常综合成一级多路复用器（图的右边部分），而if-then-else则综合成优先编码的串接的多个多路复用器，如图的左边部分。通常，使用case 语句要比if语句快，优先编码器的结构仅在信号的到达有先后时使用。条件赋值语句也能综合成多路复用器，而case 语句仿真要比条件赋值语句快。



•所有的Case 应该有一个default case， 允许空语句

Default： ；

4.1.7 Writing functions

- 在function的最后给function赋值
- 函数中避免使用全局变量

否则容易引起HDL行为级仿真和门级仿真的差异。如：

```
function ByteCompare；
input [15:0] Vector1；
input [15:0] Vector2；
input [7:0] Length；

begin
  if (ByteSel)
     // compare the upper byte
  else
     // compare the lower byte
  。 。 。
end
endfunction // ByteCompare
```

中使用了全局变量ByteSel，可能无意在别处修改了，导致错误结果。最好直接在端口加 以定义。

4.1.8 Assignment

•Verilog 支持两种赋值：过程赋值(procedural) 和连续赋值(continuous)。过程赋值用于过程代码（initial, always, task or function)中给reg 和 integer变量赋值，而连续赋值一般给wire 变量赋值。

•Always @(敏感表），敏感表要完整，如果不完整，将会引起仿真和综合结果不一致

```
always @(d or Clr)
begin
    if (Clr)
        q = 1'b0;
    else if (e)
        q = d;
end
```

以上语句在行为级仿真时e的变化将不会使仿真器进入该进程,导致仿真结果错误

•Assign/deassign 仅用于仿真加速

•Force/release 仅用于debug

•避免使用Disable

•对任何reg赋值用非阻塞赋值代替阻塞赋值

4.1.9 Combinatorial Vs Sequential Logic

•如果一个事件持续几个时钟周期，设计时就用时序逻辑代替组合逻辑

如：
```
Wire      Ct_24_e4;          //it ccarries info. Last over several clock cycles
Assign    Ct_24_e4 = (count8bit[7:0] >= 8'h24) & (count8bit[7:0] <= 8'he4);
```

那么这种设计将综合出两个8 比特的加法器，而且会产生毛刺，对于这样的电路，要采用时序设计，代码如下：

```
Reg      Ct_24_e4;
Always @(poseddge Clk or negedge Rst_)
Begin
   If (!Rst_)
      Ct_24_e4  <= 1'b0;
   Else if (count8bit[7:0] = = 8'he4)
      Ct_24_e4 <=  1'b0;
  Else if (count8bit[7:0] = = 8'h23)
     Ct_24_e4 <=  1'b1;
   Esle    ;
End
```

•在simulation pattern 或 report file中，尽量不用内部信号，如果要用就把它们引到端口，这样做并不增加芯片面积。

•内部总线不要悬空。在default状态，要把它上拉或下拉。

```
Wire        OE_default;
Assign      OE_default = !(oe1 | oe2 | oe3);
```

```
Assign      bus[31:0] = oe1 ? Data1[31:0] :
                        oe2 ? Data2[31:0] :
                        oe3 ? Data3[31:0] :
                        oe_default ? 32'h0000_0000 :
                        32'hzzzz_zzzz;
```

4.1.10 Macros

•为了保持代码的可读性，常用“ `define” 做常数声明

•把“ `define” 放在一个独立的文件中

参数（parameter）必须在一个模块中定义，不要传替参数到模块，“ `define” 可以在任何地方定义，要把所有的“ `define” 定义在一个文件中，在编译原代码时首先要把这个文件读入。如果希望宏的作用域仅在一个模块中，就用参数来代替。

4.1.11 Comments

•对更新的内容更新要做注释

•在语法块的结尾做标记

```
//style  1
   If (~OE_ && (state != PENDING))
   begin
      ....
   End  // if enable = = ture and ready
//style  2  --- identical lables on begin and end
   If (~OE_ && (state != PENDING))
   begin    //drive data
      ....
   End      //drive data
//  Comment end<unit> with the name of the <unit>
   Function  Calcparity；//Data, ParityErr
      ....
   Endfunction      // Calcparity
```

•每一个模块都应在模块开始处做模块级的注释（参考前面标准模块头）

•在模块端口列表中出现的端口信号，都应做简要的功能描述

4.1.12 FSM

•VerilogHDL状态机的状态分配

VerilogHDL描述状态机时必须由parameter分配好状态，这与VHDL不同，VHDL状态机状态可以在综合时分配产生。

•组合逻辑和时序逻辑分开用不同的进程

组合逻辑包括状态译码和输出，时序逻辑则是状态寄存器的切换

• 必须包括对所有状态都处理，不能出现无法处理的状态，使状态机失控

• Mealy机的状态和输入有关,而Moore机的状态转换和输入无关。

Mealy 状态机的例子如下:

```
    ...
reg CurrentState, NextState, Out1;

Parameter S0=0,S1=1;

always @(posedge Clk or negedge Rst_)
// state vector flip-flops (sequential)
begin
  if (!Reset)
     CurrentState = S0;
  else
     CurrentState <= NextState;
end

always @(In1 or In2 or CurrentState)
// output and state vector decode (combinational)
begin
   case (CurrentState)
   S0:
   begin
     NextState <= S1;
     Out1 <= 1'b0;
   end
   S1:
    begin
      if (In1)
      begin
        NextState <= S0;
        Out1 <= In2;
      end
      Else
      begin
         NextState <= S1;
         Out1 <= !In2;
      end
  Endcase
end
endmodule
```

4.2 代码编写中容易出现的问题

•在for-loop中包括不变的表达式，浪费运算时间

```
for (i=0;i<4;i=i+1)
begin
   Sig1 = Sig2;
   DataOut[i] = DataIn[i];
end
```

for-loop中第一条语句始终不变,浪费运算时间.

•资源共享问题

条件算子中不存在资源共享，如

```
z = (cond) ? (a + b) : (c + d);
```

必须使用两个加法器;

而等效的条件if-then-else语句则可以资源共享，如

```
if (Cond)
    z =  a + b;
else
    z =  c + d;
```

只要加法器的输入端复用,就可以实现加法器的共享,使用一个加法器实现

•由于组合逻辑的位置不同而引起过多的触发器综合，如下面两个例子

```
module COUNT (AndBits, Clk, Rst);
Output      Andbits;
Input       Clk,
             Rst;
Reg         AndBits;
//internal reg
Reg         [2:0]       Count;

always @(posedge Clk)
begin
   begin
     if (Rst)
          Count <= 0;
      else
          Count <=  Count + 1;
     End       //end if
     AndBits <=  & Count;
```

```
End          //end always
endmodule
```

在进程里的变量都综合成触发器了,有4个;

```
module COUNT (AndBits, Clk, Rst);
Output          AndBits;
Input           Clk,
                Rst;
Reg             AndBits;
//internal reg
Reg             [2:0]       Count;

always @(posedge Clk)
begin //synchronous
  if (Rst)
    Count <= 0;
  else
    Count <= Count + 1;
End      //end always

always @(Count)
begin   //asynchronous
    AndBits = & Count;
End   //end always
Endmodule      //end  COUNT
```

组合逻辑单开,只有3个触发器.

•谨慎使用异步逻辑

```
module COUNT (Z,  Enable, Clk, Rst);
Output          [2:0]          Z;
Input           Rst,
                Enable,
                Clk;

reg [2:0] Z;

always @(posedge Clk)
begin
  if (Rst)
  begin
```

```
        Z <= 1'b0;
      end
      else if (Enable == 1'b1)
      begin
          If (Z == 3'd7)
          begin
              Z <= 1'b0;
          End
          Else
          begin
              Z <= Z + 1'b1;
          end
       End
       Else ;
    End       //end always
    Endmodule     //end COUNT
```

是同步逻辑,而下例则使用了组合逻辑作时钟,以及异步复位.实际的运用中要加以避免.

```
module COUNT (Z,  Enable, Clk, Rst);
Output          [2:0]          Z;
Input           Rst,
                Enable,
                  Clk;
Reg             [2:0]          Z;
 //internal wire
wire GATED_Clk = Clk & Enable;

always @(posedge GATED_Clk or posedge Rst) begin
begin
  if (Rst)
  begin
      Z <= 1'b0;
   end
   Else
  begin
      if (Z == 3'd7)
      begin
          Z <= 1'b0;
      end
```

```
        Else
        begin
            Z <= Z + 1'b1;
        end
    End       //end if
End       //end always

Endmodule       //end module
```

•对组合逻辑的描述有多种方式，其综合结果是等效的

```
c = a &b;
等效于
c[3:0] = a[3:0] & b[3:0];
等效于
c[3] = a[3] & b[3];
c[2] = a[2] & b[2];
c[1] = a[1] & b[1];
c[0] = a[0] & b[0];
等效于
for ( i=0; i<=3; i = i + 1)
    c[i] = a[i] & b[i];
```

可以选择简洁的写法.

•考虑综合的执行时间

通常会推荐将模块划分得越小越好，事实上要从实际的设计目标、面积和时序要求出发。好的时序规划和合适的约束条件要比电路的大小对综合时间的影响要大。要依照设计的目标来划分模块，对该模块综合约束的scripts也可以集中在该特性上。要选择合适的约束条件，过分的约束将导致漫长的综合时间。最好在设计阶段就做好时序规划，通过综合的约束scripts来满足时序规划。这样就能获得既满足性能的结果，又使得综合时间最省。

•避免点到点的例外

所谓点到点例外（Point-to-point exception），就是从一个寄存器的输出到另一个寄存器的输入的路径不能在一个周期内完成。多周期路径就是其典型情况。多周期路径比较麻烦，在静态时序分析中要标注为例外，这样可能会因为人为因素将其他路径错误地标注为例外，从而对该路径没有分析，造成隐患。避免使用多周期路径，如果确实要用，应将它放在单独一个模块，并且在代码中加以注释。

•避免伪路径(False path)

伪路径是那些静态时序分析（STA）认为是时序失败，而设计者认为是正确的路径。通常会人为忽略这些warning，但如果数量较多时，就可能将其他真正的问题错过了。

•避免使用Latch

使用Latch必须有所记录，可以用All_registers -level_sensitive来报告设计中用到的Latch。不希望使用Latch时，应该对所有输入情况都对输出赋值，或者将条件赋值语句写全，如在if语句最后加一个else，case语句加defaults。

当你必须使用Latch时，为了提高可测性，需要加入测试逻辑。

不完整的if和case语句导致不必要的latch的产生，下面的语句中，DataOut会被综合成锁存器。如果不希望在电路中使用锁存器，它就是错误。

```
always @(Cond)
begin
    if (Cond)
        DataOut <= DataIn；
end
```

•避免使用门控时钟

使用门控时钟(Gated clock)不利于移植，可能引起毛刺，带来时序问题，同时对扫描链的形成带来问题。门控钟在低功耗设计中要用到，但通常不要在模块级代码中使用。可以借助于Power compiler来生成，或者在顶层产生。

•避免使用内部产生的时钟

在设计中最好使用同步设计。如果要使用内部时钟，可以考虑使用多个时钟。因为使用内部时钟的电路要加到扫描链中比较麻烦，降低了可测性，也不利于使用约束条件来综合。

•避免使用内部复位信号

模块中所有的寄存器最好同时复位。如果要使用内部复位，最好将其相关逻辑放在单独的模块中，这样可以提高可阅读性。

•如果确实要使用内部时钟，门控时钟，或内部的复位信号，将它们放在顶层

将这些信号的产生放在顶层的一个独立模块，这样所有的子模块分别使用单一的时钟和复位信号。一般情况下内部门控时钟可以用同步置数替代。例如:

```
module COUNT (Reset， Enable， Clk， Qout)；
input Reset， Enable， Clk；
output [2:0] Qout；
reg [2:0] Qout；

wire GATED_Clk = Clk & Enable；


always @(posedge GATED_Clk or posedge Reset)
begin
    if (Reset)
    begin
        Qout = 1'b0；
    end
    Else
    begin
        if (Qout == 3'd7)
        begin
            Qout= 1'b0；
         end
        Else
```

```
        begin
          Qout = Qout + 1'b1；
        end
   end
end
endmodule


module COUNT (Reset， Enable， Clk， Qout)；

input Reset， Enable， Clk；
output [2:0] Qout；
reg [2:0] Qout；

always @(posedge Clk)
begin
    if (Reset)
    begin
        Qout = 1'b0；
    end
    else if (Enable == 1'b1)
    begin
        if (Qout == 3'd7)
        begin
            Qout = 1'b0；
        end
        Else
        begin
            Qout = Qout + 1'b1；
        end
    end
end
endmodule
```