Verilog HDL 代码规范

（仅内部使用）

文档作者：　冯仁委　　日期：　2013-8-5

审　　批：　　日期：

修订记录

日期	修订版本	描述	作者
2013-8-5	1.00	初稿完成	冯仁委

目录

1. 目的

为了提高 Verilog HDL 代码的可读性、可修改性、可重用性，方便成员间的交流使用，提升项目组的设计效率，现面向 FPGA 组制定本规范。

2. 范围

本规范包括可综合设计规范和仿真验证规范两部分。可综合设计规范是针对逻辑设计代码，适用于 RTL 的任何一种描述方式（数据流描述，行为描述，门级描述）。仿真验证规范针对 Testbench 代码。

本规范读者包括 FPGA 设计人员、验证人员和其他与 FPGA 开发相关人员。

3. 定义

Verilog HDL	Verilog 硬件描述语言
FSM	有限状态机
RM	参考模型
BFM	总线功能模型

4. 规范内容

1) 可综合设计

确保代码能在绝大多数综合工具所接受，且能无歧义快速综合，现制定可综合代码规范。

a) 命名规范

命名要做到简洁、清晰、有效，尽可能做到见名知意。

1. 使用有意义的名字。

使用有意义的的名字，使设计者更容易理解信号意义，理解模块功能，发现设计错误，修改错误，同时方便成员间交流。

2. 使用规范的缩写。

信号命名过长对设计带来麻烦，缩写就是为了简洁、清晰，所有缩写要求能基本表明本单词的含义。常用信号缩写见表格 2，其它信号名一般取其前三个字母。信号名长度在 20 个字母内，词与词之间用下划线连接。

3. 使用规范的大小写。

端口、信号、变量所有字母小写，函数名、模块名、宏定义、参数定义大写。

比如：

input sys_rst；

parameter CLK_PERIOD = 20。

4. 总线命名规则。

总线宽度的语句必须用 [N:0]，而不能用 [0:N]，标记 0 表示 LSB，标记 N 表示 MSB。

5. 低有效信号名加 _n。

比如：

sys_rst_n。

6. 信号打拍加 _Xd，打一拍加 _d 或者 _1d，打 2 拍加 _2d。

比如：

sck_ctrl_d，

sck_ctrl_2d。

7. 避免使用数字和 reg 作为后缀。

综合工具会给寄存器自动加上_reg，如果自己加上_reg，网表可读性变差。多 bit 信号综合后可能会改名成 bit 数后缀的单 bit 信号，造成网表理解困难。

比如：ctrl_reg 改为 reg_ctrl，

8. 避免使用关键字。

比如：in out case 等不能够做为变量端口或模块。

9. 同一信号在不同层次应保持一致性。

层次化设计中，同一信号相同名称可以确保不同层次信号处理正确，避免因为名称不同造成设计错误。

10. 全局信号名字中应包含信号来源的一些信息。系统信号应加上 sys。

比如：adc2378_sdo 、sys_clk。

11. 添加有意义的后缀使信号名更加明确常用的后缀。

如表 1

表格 1 **常用信号后缀表：**

信号后缀	意义
_clk	时钟
_n	低有效

_neg	下降沿
_pos	上升沿
_z	三态信号
_en	使能

表格 2 常见信号缩写：

全称	缩写	中文含义
acknowledge	ack	应答
adress	addr	地址
arbiter	arb	仲裁
check	chk	校验
clock	clk	时钟
configuration	cfg	配置
control	ctrl	控制
counter	cnt	计数器
current state	curr_st	当前状态
data in	din	数据输入
data out	dout	数据输出
decode	de	译码
decrease	dec	减一
delay	dly	延迟
disable	dis	不使能
error	err	错误
enable	en	使能
frame	frm	帧
generate	gen	生成
grant	gnt	同意，准予，承认
increase	inc	加一
input	in	输入
length	len	（帧、包）长
next state	next_st	下一状态
output	out	输出
packet	pkt	包
priority	pri	优先级
pointer	ptr	指针

read enable	rden	读使能
read	rd	读
ready	rdy	准备好
receive	rx	接收
request	req	请求
reset	rst	复位
segment	seg	片段 节
source	scr	源
statistics	stat	统计
timer	tmr	定时
switch	sf	转换
temporary	tmp 或者 temp	临时的
transmit	tx	发送
valid	vld	有效
write anable	wren	写使能
write	wr	写操作

b) 模块规范

模块的命名尽量用英文表达出其完整的功能，长度一般不少于 2 个字母。

1. 每文件最多只能包含一个模块。

每个模块必须用一个文件表示。在一个文件中不允许存在一个以上的模块。（该模块的测试文件必须用另外一个文件表示，因此一个经过初步验证的模块必须用一个包含可综合模块和测试文件的文件夹来表示）。

2. 顶层模块应只是内部模块间的互连，不允许顶层模块出现胶合逻辑。

在顶层模块中出现中间胶合逻辑，综合工具就不能把子模块中的逻辑综合到最优。

3. 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。

如下：

```
//------------------------------------------------------------------------------
// Title          : adc2378 data sample
// Project        : lb1101
// Filename       : adc2378_spi_din.v
```

```
//-------------------------------------------------------------------------------
// Description    :
// adc2378 sample circuit
//-------------------------------------------------------------------------------
// Revisions      :
// Date           : 2013-07-15
// Version        : 1.0
// Author         : XXX    XXX@linbo.com.cn
// Description    : create
// Platform       : Quartus II 12.0SP2
// Corporation    : LinBo Co.Ltd.
// Copyright(c)   : LinBo Co.Ltd    All right reserved
//-------------------------------------------------------------------------------
```

4. 对不要对 Inpu t 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 compile 时产生 warning 干扰错误定位。

5. 对每行应限制在 80 个字符以内以保持代码的清晰美观和层次感

对一条语句占用一行如果较长超出 80 个字符则要换行。

6. 对电路中调用的 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。

比如：

```
XXFIFO    U_XXFIFO_X(
        .aclr          (sys_rst     ),
        .data          (XXX_din     ),
        .rdclk         (sys_clk     ),
        ……
      );
```

例化名 U_XXFIFO_X 中，_X 中 X 用数字表示，如果需要例化多个子模块，则第一个子模块用_0 表示，第二个子模块用_1 表示，以此类推。

7. 模块例化时，例化模块输入必须有驱动，未使用的输出须有连接。

保证输入必须有驱动，否则工具报错；未使用的输出信号须有连接，建议使用“_nc”结尾。当出现有信号尚未连接的警告时，若信号的命名是以“_nc”结尾的，则可知该信号是设计者故意不连接的，不需要进行修改。

8. 不能同时使用一个时钟的上升沿或下降沿采样。

整个设计采用同一时钟采样有利于综合。

9. 内部模块端口避免 inout，尽量在最顶层模块处理双向端口。

10. 三态逻辑使用。

三态逻辑可以在顶层模块使用，子模块中禁止使用三态。如果能确保该信号不会被其它子模块使用，而是直接通过顶层模块输出要 I/O 口，可以在子模块中使用三态。

11. 在模块中增加明了的注释。

对信号、参量、引脚、模块、函数及进程等加以说明便于代码维护和组内沟通。

12. 模块中，禁止使用 tab 键，代码对齐只许使用空格键。

Tab 键一般工具显示 4 个空格，但是有些工具也会显示 3 个空格（可设置），如果使用 tab，代码在不同工具中可能显示混乱。

13. 对每一个 always、assign 语句，都从顶格写起，子层代码在其上一层上缩进 4 个空格键写起，每一层须有 begin……end，对于一层只有一句语句的可以灵活处理。

每一 always、assign 语句都独立处理一个信号，完成一个特定的功能。缩进处理则层次性更加明显，代码可读性更强，更容易发现和避免错误。

14. 宏定义必须放在模块内或者放在专门定义宏的文件中，统一管理。

由 `define 定义的宏文本声明语句必须放在模块名后，如：<顶层模块名>_<宏名>。放在模块定义语句外的宏定义有可能被其他层次模块的宏定义搞混，从而发生无意中对某个模块外的宏定义进行重新定义的错误。

15. 模块须先声明参数、输入端口、输出端口、本地变量，再写主代码，关键信号需要详细注释，没有参数可空开。

比如：

```
//*******************************************************
// PARAMETERS
//*******************************************************

//*******************************************************
//INPUTS
//*******************************************************
input sys_rst    ; // system rst,hign active
……
//*******************************************************
//OUTPUS
//*******************************************************
output      adc2378_cnv ; // adc2378_cnv up to 1M
```

```
……
reg         adc2378_cnv ;
//*****************************************************
//LOCAL VARIABLE
//*****************************************************
//REG
reg         sck_ctrl_neg ; //adc2378_sck_ctrl negedge,
//WIRE

//*****************************************************
// MAIN CODES
//*****************************************************
```

16. 对 module 名要用大写标示且应与文件名保持一致。

比如：

```
module ADC2261_DIN(
    sys_rst ,
    sys_clk ,
    ……
    dout
    );
```

17. 对输入、输出端口和信号的声明必须每个信号一行。

所有端口和信号的声明必须每个信号一行，必要时在端口信号后添加注释。增加可读性。简化用设计工具来对代码进行语法分析。

比如：

```
//*****************************************************
//INPUTS
//*****************************************************
input               sys_rst             ; // system rst,hign active
input               sys_clk             ; // system clk
input               adc2261_clk         ; // adc2261 clkin
//*****************************************************
//OUTPUS
//*****************************************************
output              adc2261_cs          ; // adc2261 chip select signal
```

```
output              adc2261_sck       ; // adc2261 spi interface
//*******************************************************
//LOCAL VARIABLE
//*******************************************************
//REG
reg                 adc2261_of_d      ; //One sys_clk cycle delay of adc2261_of signal
```

18. 模块中每个信号处理均需注释。

每一个 always、assign 语句处理一个信号，完成一个功能，在每个处理块前加几行注释对该代码加以描述，至少列出本节中所处理的信号，更详细的说明则更好（比如描述其功能）。

比如：

```
//adc2378_busy_neg
always @(*)begin
    adc2378_busy_neg = adc2378_busy_2d & (~adc2378_busy_d);
end
```

19. 一句 always 语句，只能使用一个时钟的一个沿（上升沿或者下降沿）。

在一个 module 中，在时钟信号的同一个沿动作。如果必须使用时钟上升沿和时钟下降沿，要分两个 module 设计。

20. 组合逻辑中推荐使用 always @(*)begin 替代 always @(aXX or bXX or cXX)begin。

避免实际电路与仿真结果不同。

21. 尽可能用 always 语句替代 assign 语句。

比如：

```
assign adc2378_busy_neg = adc2378_busy_2d & (~adc2378_busy_d) ;可替换成
always @(*)begin
    adc2378_busy_neg = adc2378_busy_2d & (~adc2378_busy_d);
end
```

22. 信号判断使用表达式替代信号直接判断。

比如：

if(sys_rst)begin 替换成 if(sys_rst == 1'b1)begin

23. 对于模块的任何修改均需注释。

比如：

```
// SPI programme interface
//always@(*)begin
```

```
//      adc2261_cs    = 1'b1;
//      adc2261_sck = 1'b0;
//      adc2261_sdi = 1'b0;
//end
// The value of the signal is X in modelsim
// modified by fengrenwei 2013.8.1
always@(posedge sys_clk)begin
    if(sys_rst == 1'b1)begin
        adc2261_cs     <= 1'b1;
        adc2261_sck    <= 1'b0;
        adc2261_sdi    <= 1'b0;
    end
    else begin
        adc2261_cs     <= 1'b1;
        adc2261_sck    <= 1'b0;
        adc2261_sdi    <= 1'b0;
    end
end
```

24. 模块划分须合理。

合理的模块划分使设计简单、清晰。

模块划分一般需遵循以下原则：

1、 对顶层模块，不允许有逻辑设计。

2、 要层次化设计，模块之间相互独立。

3、 时钟复位电路单独划分。

4、 对每个同步时序设计的子模块的输出使用寄存器。

5、 对相关逻辑和可以复用的逻辑划分在同一模块内。

6、 对将不同优化目标的逻辑分开。

7、 对相同约束的逻辑归到同一模块。

8、 将存储逻辑独立划分成模块。

9、 模块规模不能太大。

10、 关键路径逻辑和非关键路径逻辑放在不同模块。

c) Net and Register

1. 一个 reg 变量只能在一个 always 语句中赋值。

一个 reg 不能在不同 always 语句中赋值，造成多重驱动。

2. **同一信号赋值不能同时使用阻塞和非阻塞方式。**
3. **不允许出现定义的 parameter/wire/reg 没有使用。**
4. **对 net 和 register 类型的输出要做声明。**

没做声明信号 Verilog 将假定它为一位宽的 wire 变量。

d) Expressions

1. **用括号来表示执行的优先级。**
2. **一行中不能出现多个表达式。**
3. **不要给信号赋“x”态，以免 x 值传递。**
4. **设计中使用到的 0，1 常数采用基数表示法书写（即表示为 1'b0,1'b1,1'bz 或十六进制）。**
5. **端口申明、比较、赋值等操作时，数据位宽要匹配。**

e) For 语句

在 Verilog-2001 中使用 for 语句，必须使用 generate 语句。

1. **必须有 genvar 关键字定义 for 的变量。**
2. **必须给 for 语段起个名字。**

比如：

```
generate
//定义变量
genvar i;
for(i = 0; i < XX; i = i + 1)begin：FOR_NAME
    //表达式;
    //例化模块;
end
endgenerate
```

f) If 语句

3. **每一个 if 都应有一个 else 和它相对应。**

没有 else 可能会使综合出的逻辑和 RTL 级的逻辑不同（对于组合电路会生成锁存器），如果条件为假时不进行任何操作则用一条空语句。

4. **If…..else 语句是有优先级的。**
5. **不允许常数作为 if 语句的条件表达式。**

条件表达式必须是 1bit value。

6. **不推荐嵌套使用 5 级以上 if…else if…结构。**

g) Case 语句

1. **每一个 Case 应该有一个 default 对应，case 允许空语句。**

case 语句中所有变量在所有分支中都赋值。如果用到 case 语句，写上 default 项。没有 default 的组合逻辑会生成锁存器。

2. **禁止使用 casex 或者 casez。**
3. **case 没有优先级（不使用综合工具语句）。**

case 语句通常综合成一级多路复用器。

4. **case 语句 item 必须使用常数。**
5. **避免使用与综合工具紧密相关的语句，比如//synopsys parallel_case/ full_case 等。**

h) Writing functions

1. **在 function 的最后给 function 赋值。**
2. **函数中避免使用全局变量。**

i) 使用可综合的语句

1. **不要使用 disable、initial 等综合工具不可综合的语句，而应采用复位方式进行初时化。**
2. **不要使用 specify 模块。**
3. **不要使用===、！==等不可综合的操作符。**
4. **除仿真外，不要使用 fork-join、while、Repeat、forever、deassign、force-release、named -events 语句。**
5. **不要在连续赋值语句中引入驱动强度和延时。**
6. **禁止使用 trireg 型、tri1、tri0、triand 和 trior 型的连接。**

j) Comments

1. **对更新的内容更新要做注释（见命名规范）。**
2. **在块的结尾做标记。**
3. **每一个模块都应在模块开始处做模块级的注释。（见命名规范）**
4. **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。**

k) FMS

1. **VerilogHDL 状态机的状态必须由 parameter 分配。**
2. **组合逻辑和时序逻辑要分开。**

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。

3. **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。**
4. **Mealy 机与 Moore 机。**

Mealy 机的状态和输入有关,而 Moore 机的状态转换和输入无关。

5. **binary 和 gray-code 消耗较少的触发器，较多的组合逻辑；one-hot 消耗较多的触发器，较少的组合逻辑。**
6. **使用“三段式”状态机描述方式。**

两个时序 always 模块分别产生当前状态和输出，再用一个组合 always 产生下一状态。如下：

```
//第一个进程，同步时序 always 模块，描述状态转移。
……
always @ (posedge sys_clk or negedge sys_rst) begin
    if(sys_rst == 1'b1)
        curr_st <= IDLE;
    else
        curr_st <= next_st;
end

//第二个进程，组合逻辑 always 模块，描述状态转移条件判断
always @ (*) begin
    case(curr_st)
    stateA: begin
        next_st = stateB;
```

```
        end
        stateB: begin
            next_st = stateC;
        end
        ...
        default: next_st = stateA;
        endcase
    end

    //第三个进程，同步时序 always 模块，描述逻辑输出
    always @ (posedge sys_clk or negedge sys_rst) begin
        if(sys_rst == 1'b1)begin
            dout <= X'b0;
        end
        else begin
            // use case
            case(curr_st)
                stateA: begin
                    dout <= X'd1;
                end
                stateB: begin
                    dout <= X'd2;
                end
                default:
            endcase
            //use if….else
            if(curr_st == stateX)
                dout <= X'd1;
            else
                dout <= X'd2;
        end
    end
```

2) 仿真验证

1. **Testbench 要避免使用无限循环，时钟除外。**
2. **Testbench 不需要综合，所有符合语法的代码都可以使用。**
3. **被测试模块的输入激励设置为 reg 型，输出相应设置为 wire 类型，双向端口 inout 在测试中需要进行处理。**

方法 1：为双向端口设置中间变量 inout_reg 作为该 inout 的输出寄存，inout 口在 testbench 中要定义为 wire 型变量，然后用输出使能控制传输方向。

比如：

```
inout   [0:0]   bi_dir_port      ;
wire    [0:0]   bi_dir_port      ;
reg     [0:0]   bi_dir_port_reg ;
reg             bi_dir_port_oe ;
assign bi_dir_port = bi_dir_port_oe ? bi_dir_port_reg : 1'bz;
```

用 bi_dir_port_oe 控制端口数据方向，并利用中间变量寄存器改变其值。等于两个模块之间用 inout 双向口互连。

方法 2：使用 force 和 release 。

比如：

```
module XXX_tb();
wire   data_inout ;
reg    data_reg   ;
reg    link       ;
…
force data_inout=1'bx;
...
#xx;
release data_inout;
endmodule
```

4. **系统或者复杂模块须写 BFM，与参考模型（RM）的数据比对须打印 log。**
5. **从文本文件中读取和写入向量。**

1）读取文本文件：用 $readmemb 系统任务从文本文件中读取二进制向量（可以包含输入激励和输出期望值）。$readmemh 用于读取十六进制文件。

比如：

```
// 定义存储器 mem，位宽是 8bits，深度是 256 字节
reg [7:0] mem[1:256]
// 将.dat 文件读入寄存器 mem
initial begin
    #5；
    $readmemh ( "mem.data", mem );
end
//参数为寄存器加载
initial begin
    #5;
    $readmemh ( "mem.data", mem, 128, 1 ) ;
end
```

也可以使用$fread 读取

比如：

```
file_id = $fread("file_path/file_name", "r");
```

2）输出文本文件：打开输出文件用$fopen。

比如：

```
// out_file 是一个文件描述，需要定义为 integer 类型
integer out_file;
out_file = $fopen ( " cpu.data " );
```

3）关闭文件

```
$fclose(fjile_id);
```

6. 设计中的信号值可以通过$fmonitor, $fdisplay 打印关键信息。

```
//显示任务
$display();
//监视任务
$monitor();
```

7. Testbench 模板。

模板：

```
module xxx_tb; //定义一个没有输入输出的 module
// PARAMETERS
parameter ……;
//将 DUT 的输入定义为 reg 类型
```

```
reg ……
……
//将 DUT 的输出定义为 wire 类型
wire……
……
//例化 DUT
XXX   U_XXX(
    //INPUTS
  .sys_rst        (sys_rst     ),
  .sys_clk        (sys_clk     ),
    //OUTPUS
  .dout           (dout        )
 );
//产生时钟
initial begin
      sys_clk = 0;
      forever begin
            #(CLK_PERIOD / 2) sys_clk = ~sys_clk;
      end
end
//产生复位
initial begin
      #10;
      sys_rst = 1'b0;
      @(negedge sys_clk);
      sys_rst = 1'b1;
      repeat (10) @(negedge sys_clk);
      sys_rst = 1'b0;
      #200;
end

Initial begin
…… //在这里添加激励
end
```

```
initial
….//在这里添加比较语句(可选)
end
initial
//在这里添加输出语句(在屏幕上显示仿真结果)
$display(" XXX");
end
initial begin
    #10;
    //控制时间，避免跑死
    #(6000000*CLK_PERIOD);
    $stop;
    $finish;
end
endmodule
```