先记下来：

1、不使用初始化语句；

2、不使用延时语句；

3、不使用循环次数不确定的语句，如：forever，while 等；

4、尽量采用同步方式设计电路；

5、尽量采用行为语句完成设计；

6、always 过程块描述组合逻辑，应在敏感信号表中列出所有的输入信号；

7、所有的内部寄存器都应该可以被复位；

8、用户自定义原件（UDP 元件）是不能被综合的。

一：基本

Verilog 中的变量有线网类型和寄存器类型。线网型变量综合成 wire，而寄存器可能综合成 WIRE，锁存器和触发器，还有可能被优化掉。

二：verilog 语句结构到门级的映射

1、连续性赋值：assign

连续性赋值语句逻辑结构上就是将等式右边的驱动左边的结点。因此连续性赋值的目标结点总是综合成由组合逻辑驱动的结点。Assign 语句中的延时综合时都将忽视。

2、过程性赋值：

过程性赋值只出现在 always 语句中。

阻塞赋值和非阻塞赋值就该赋值本身是没有区别的，只是对后面的语句有不同的影响。

建议设计组合逻辑电路时用阻塞赋值，设计时序电路时用非阻塞赋值。

过程性赋值的赋值对象有可能综合成 wire，latch,和 flip-flop，取决于具体状况。如，时钟控制下的非阻塞赋值综合成 flip-flop。

过程性赋值语句中的任何延时在综合时都将忽略。

建议同一个变量单一地使用阻塞或者非阻塞赋值。

3、逻辑操作符：

逻辑操作符对应于硬件中已有的逻辑门，一些操作符不能被综合：===、!==。

4、算术操作符：

Verilog 中将 reg 视为无符号数，而 integer 视为有符号数。因此，进行有符号操作时使用 integer,使用无符号操作时使用 reg。

5、进位：

通常会将进行运算操作的结果比原操作数扩展一位，用来存放进位或者借位。如：

```
Wire [3:0] A,B;
Wire [4:0] C;
Assign C=A+B;
```

C 的最高位用来存放进位。

6、关系运算符：

关系运算符：<,>,<=,>=

和算术操作符一样，可以进行有符号和无符号运算，取决于数据类型是 reg，net 还是 integer。

7、相等运算符：==，！=

注意：===和！==是不可综合的。

可以进行有符号或无符号操作，取决于数据类型

8、移位运算符：

左移，右移，右边操作数可以是常数或者是变量，二者综合出来的结果不同。

9、部分选择：

部分选择索引必须是常量。

10、BIT 选择：

BIT 选择中的索引可以用变量，这样将综合成多路（复用）器。

11、敏感表：Always 过程中，所有被读取的数据，即等号右边的变量都要应放在敏感表中，不然，综合时不能正确

地映射到所用的门。
12、IF：
如果变量没有在 IF 语句的每个分支中进行赋值，将会产生 latch。如果 IF 语句中产生了 latch，则 IF 的条件中最好不要用到算术操作。Case 语句类似。Case 的条款可以是变量。
如果一个变量在同一个 IF 条件分支中先赋值然后读取，则不会产生 latch。如果先读取，后赋值，则会产生 latch。
13、循环：
只有 for-loop 语句是可以综合的。
14、设计时序电路时，建议变量在 always 语句中赋值，而在该 always 语句外使用，使综合时能准确地匹配。建议不要使用局部变量。
15、不能在多个 always 块中对同一个变量赋值
16、函数
函数代表一个组合逻辑，所有内部定义的变量都是临时的，这些变量综合后为 wire。
17、任务：
任务可能是组合逻辑或者时序逻辑，取决于何种情况下调用任务。
18、Z：
Z 会综合成一个三态门，必须在条件语句中赋值
19、参数化设计：
优点：参数可重载，不需要多次定义模块
四：模块优化
1、资源共享：
当进程涉及到共用 ALU 时，要考虑资源分配问题。可以共享的操作符主要有：关系操作符、加减乘除操作符。通常乘和加不共用 ALU，乘除通常在其内部共用。
2、共用表达式：
如：C=A+B;
D=G+(A+B);
两者虽然有共用的Ａ＋Ｂ，但是有些综合工具不能识别．可以将第二句改为：D=G+C；这样只需两个加法器．
3、转移代码：
如循环语句中没有发生变化的语句移出循环．
4、避免 latch：
两种方法：1、在每一个 IF 分支中对变量赋值。2、在每一个 IF 语句中都对变量赋初值。
5：模块：
综合生成的存储器如 ROM 或 RAM 不是一种好方法，只是成堆的寄存器，很费资源。最好用库自带的存储器模块。
五、验证：
１、敏感表：
在 always 语句中，如果敏感表不含时钟，最好将所有的被读取的信号都放在敏感表中。
２、异步复位：
建议不要在异步时对变量读取，即异步复位时，对信号赋以常数值。

Averilog 的流行,有两方面的原因;
B verilog 与 VHDL 相比的优点
C 典型的 verilog 模块
D verilog 语法要点

A) verilog 的流行,有两方面的原因:
1 它是 cadence 的模拟器 verilog-XL 的基础,cadence 的广泛流行使得 verilog 在 90 年代深入人心;
2 它在硅谷获得广泛使用;
B) verilog 与 VHDL 相比的优点二者的关系仿佛 C 与 FORTRAN,具体而言:

1 verilog 的代码效率更高:
比较明显的对比:
VHDL 在描述一个实体时采用 entity/architecture 模式,
verilog 在描述一个实体时只需用一个"module/edumodule"语句块.
此外 verilog 的高效性还在很多地方体现出来;
2 verilog 支持二进制的加减运算:
VHDL 在进行二进制的加减运算时使用 conv_***函数或者进行其他的定义,总之必须通知编译器;verilog 直接用形如"c=a+b"的表示二进制的加减运算;
3 综合时可控制性好:
VHDL 对信号不加区分地定义为"signal",
而 verilog 区分为 register 类型的和 wire 类型的;
但是也有人支持 VHDL,认为 verilog 和 VHDL 的关系仿佛 C 和 C++.C)典型的 verilog 模块
讨论以下典型电路的 verilog 描述:
*与非门;
*加法器; //即全加器
* D 触发器;
*计数器; //**分频的 counter
* latch;
*时序机;
*RAM; //用 synopsys 的
*模块引用;
*预编译;
*与非门的 verilog 描述如下:

```
//verilog 使用和 C 语言相同的注释方法
module nd02(a1,a2,zn);//一个 verilog 模块总是以 module 开始,以 endmodule 结束,nd02 是模块名,a1,a2,zn 是模块的 3 个输入输出信号
input a1,a2; //告诉编译器 a1,a2 对此模块而言是输入,并且数据类型是"bit"
output zn; //告诉编译器 zn 对此模块而言是输出,数据类型也是"bit"
nand (zn,a1,a2); //我理解 nand 是运算符,我们不必深究 verilog 中的正式术语是什么了吧, 总之这种形式表示 zn=~(a1 && a2);你一定已经想到类似的运算符还有"not","and","or","nor","xor"了吧;除了"not",括号里的信号数可以任意,例如 or (z,f,g,h)表示 z=f || g || h,并且延时是 3 个单位时间, #x 表示延时 x 个单位时间;
endmodule
```

*加法器的 verilog 描述如下:

```
module ad03d1(A,B,CI,S,CO) ;
input [2:0] A,B; //表示 A,B 是输入信号,并且是 3 位矢量,上界是 2,下界是 0
input CI;
output [2:0] S;
output CO;
assign {CO,S}=A+B+CI;//一对"{"和"}"表示链接,即将 CO 和 S 合并成 4 位矢量
endmodule
```

*带异步清零端的 D 触发器的 verilog 描述如下:

```
module dfctnb (d,cp,cdn,q,qn);
input d,cp,cdn;
output q,qn;
```

reg q,qn;　　//关键字"reg"表示 q 和 qn 是"register"类型的信号;verilog 中有两种类型的信号:"register"类型和"wire"类型.你可以简单地把 register 类型的信号想象为某个 D 触发器的输出,而 wire 类型的的信号是组合逻辑的输出.二者的最大区别在于:你可以对 register 类型的信号进行定时赋值(用 wait 语句在特定时刻的赋值,详见下面 always 语句),而对于 wire 类型的信号则不可.

always wait (cdn==0) //表示每当 cdn=0 时,将要对 D 触发器清零,"always"和"wait"嵌套，"wait"和"@"是 verilog 的两个关键字,表示一旦有某事发生；则执行下面的语句块,"always"有点象 C 语言中的"if ... then...", "wait"和"@"的区别:请参考本模块.wait 表示本语句块的进程停止,直到"cdn=0"的条件出现才继续；我理解在 verilog 中,每个最外层语句块都是一个***的进程;"@"(请看下个 always 语句)也表示本语句块的进程停止,直到后面定义"posedge cp"(即出现 cp 的上升沿)的事件出现才继续;也许 wait 和@可以合二为一吧,但至少到目前 verilog 中 wait 表示"条件",@表示"事件";具体运用中,wait 总是用于类似"wait(xxx=1)"之类的场合,@总是用于类似"@(xxx)"或"@(posedge/negedge xxx)"之类的场合整句话的意思是"每当 cdn 等于 0 时,则作以下事情"

```
begin          //begin...end 结构的用法类似于 pascal 语言
             q=0;
                qn=1;
               wait (cdn==1);
end
always @ (posedge cp)//"@(posedge cp)"中有两个关键字:"@ (x)"表示"每当事件 x 发
```

生","posedge x"表示"x 的上升沿,"negedge x"表示"x 的下降沿", 整句话的意思是"每当 cp 的上升沿,则作以下事情"

```
               if (cdn)    //如果 cdn=1(意味着清零端无效)
               begin
                                q=d;
                                qn=~q;//"~"表示反相
               end
endmodule
```

*计数器的 verilog 描述如下:

```
module count(in,set,cp,out) ;//此计数器,在 cp 的上升沿将输入赋给输出,在 cp 的上升沿使输出加一
input [15:0] in;
input set,cp;
output [15:0] out;
reg [15:0] out;
always @ (posedge set)
  out = in;
always @(posedge cp)
  out = out+1;    //verilog 容许一个信号同时出现在等号两端,只要它是 reg 类型的
endmodule
```

*latch 的描述如下:

```
always @(clk or d)
       if (clk) q = d;
```

*时序机的 verilog 描述如下:

```
always @(posedge CLK)    //D 是下一个状态,Q 是当前状态,e1,e2 是输入,a,b 是输出
Q=D;
always @(Q or othercase) begin //当 Q 变化或输入 e1,e2 变化时 D 要相应变化
D = Q; //note1
a = 0;
b = 0;
```

```
......
case(Q)
   q1:begin
     q1 action;
     if(e1)D=d1;
     if(e2)D=d2;
     else D=d3;
     a = 1; //note 2
     end
   q2:begin
     b = 1;
     ......
     end
   default:begin
     a = 0;
     b = 0;
     ......
end
end
---annotations---
note 1:
   This is a custom expression,after reset,D should be equal to Q;
note 2:
   In this state machine,a is only equal to 1 at state q1,in
   other state,a is equal to 0;
```

* RAM 的 verilog 描述如下:

```
module ram(din,ain,dout,aout,rd,wr);//这是一个双口 RAM,分别有:输入端:输入地址 ain;输入数据 din;上升沿有效的写信号 wr;/输出端:输出地址 aout;输出数据 dout;高电平有效的读信号 rd;
   inout [7:0] din;
   input [7:0] ain,aout;
   input rd,wr;
   output [7:0] dout;
   reg [7:0] memory [0:255];      //请注意这是存储阵列的描述方法,描述了一个共有 256 个字的存储阵列,每个字是 8 位
   assign dout = rd ? memory[aout] : 8'bz; //"assign"关键字表示并行赋值语句的开始"?"运算符的作用和在 C 语言中一样"8'bz"是一个常量,表示一个字节的高阻态,其中 8 表示长度是 8bit,"'"是固定分割符,"b"表示后面的数据是以比特形式给出的,"z"表示高阻;举例:4'ha 表示长 4bit 的数"1010"。类似的还可举出 5'b10111,6'o33 等等
   always @(posedge wr)
memory[ain] = din;
endmodule
```

*模块引用

假设在前面(可以是别的模块)定义了 module ram(din,ain,dout,aout,rd,wr),则引用此模块时只需写

```
ram myram(din_in_map,ain_in_map,dout_in_map,aout_in_map,rd_in_map,wr_in_map)
;
```

//其中"ram"是所引用的 module 名,"myram"是你起的 instance 名,"din_in_map"等等是图中的节点名,和器件(module)中的"din..."进行"虚实结合";

*预编译
类似 C 语言,只需写
`include "<pathname:filename>",反上撇号"`"是 verilog 的预编译符,类似 C 中的"#".

D) verilog 语法要点
*基本原则
设计时应该把你的系统划分为计数器,触发器,时序机,组合逻辑等等可综合的单元,对此不同的 IC 公司和 EDA 开发商可能根据自己的见解和经验提出不同的要求,并且对 verilog 程序的细节进行自己的规定,但有一点是对的:即写硬件描述语言不象写 C 语言那样符合语法就行.单单符合 verilog 语法的程序可能被拒绝综合,甚至被拒绝模拟;
*最外层可以写什么?
这里所说的最外层是指 module 语句后的第一层,在这一层可以写这些可执行语句:
assign 和 nand 等定义组合逻辑的语句,
always 语句,
模块引用语句,
一些以"$"开头的系统定义语句.
特别注意不可以写 if 语句.if 语句只能放在 always 内部.
不推荐写 wait 语句,因为不能综合.
*不可以在多个 always 语句中对一个信号赋值.

1. 强烈建议用同步设计
2. 在设计时总是记住时序问题
3. 在一个设计开始就要考虑到地电平或高电平复位、同步或异步复位、上升沿或下降沿触发等问题，在所有模块中都要遵守它
4. 在不同的情况下用 if 和 case，最好少用 if 的多层嵌套（1 层或 2 层比较合适，当在 3 层以上时，最好修改写法，因为这样不仅可以 reduce area，而且可以获得好的 timing）
5. 在锁存一个信号或总线时要小心，对于整个 design，尽量避免使用 latch，因为在 DFT 时很难 test。
6. 确信所有的信号被复位，在 DFT 时，所有的 FlipFlop 都是 controllable，
7. 永远不要再写入之前读取任何内部存储器（如 SRAM）
8. 从一个时钟到另一个不同的时钟传输数据时用数据缓冲，他工作像一个双时钟 FIFO（是异步的），可以用 Async SRAM 搭建 Async FIFO。
9. 在 VHDL 中二维数组可以使用，它是非常有用的。在 VERILOG 中他仅仅可以使用在测试模块中，不能被综合
10. 遵守 register-in register-out 规则
11. 像 synopsys 的 DC 的综合工具是非常稳定的，任何 bugs 都不会从综合工具中产生
12. 确保 FPGA 版本与 ASIC 的版本尽可能的相似，特别是 SRAM 类型，若版本一致是最理想的，但是在工作中 FPGA 版本一般用 FPGA 自带的 SRAM，ASIC 版本一般用厂商提供的 SRAM。
13. 在嵌入式存储器中使用 BIST
14. 虚单元和一些修正电路是必需的
15. 一些简单的测试电路也是需要的，经常在一个芯片中有许多测试模块
16. 除非低功耗不要用门控时钟，强烈建议不要在 design 中使用 gate clock
17. 不要依靠脚本来保证设计。但是在脚本中的一些好的约束能够起到更好的性能（例如前向加法器）
18. 如果时间充裕，通过时钟做一个多锁存器来取代用 MUX
19. 不要用内部 tri-state, ASIC 需要总线保持器来处理内部 tri-state，如 IO cell。
20. 在 top level 中作 pad insertion
21. 选择 pad 时要小心（如上拉能力，施密特触发器，5 伏耐压等），选择合适的 IO cell
22. 小心由时钟偏差引起的问题
23. 不要试着产生半周期信号
24. 如果有很多函数要修正，请一个一个地作，修正一个函数检查一个函数
25. 在一个计算等式中排列每个信号的位数是一个好习惯，即使综合工具能做

26.不要使用 HDL 提供的除法器

27.削减不必要的时钟。它会在设计和布局中引起很多麻烦，大多数 FPGA 有 1－4 个专门的时钟通道

良好代码编写风格可以满足信、达、雅的要求。在满足功能和性能目标的前提下，增强代码的可读性、可移植性，首要的工作是在项目开发之前为整个设计团队建立一个命名约定和缩略语清单，以文档的形式记录下来，并要求每位设计人员在代码编写过程中都要严格遵守。良好代码编写风格的通则概括如下：

（1） 对所有的信号名、变量名和端口名都用小写，这样做是为了和业界的习惯保持一致；对常量名和用户定义的类型用大写；

（2） 使用有意义的信号名、端口名、函数名和参数名；

（3） 信号名长度不要太长；

（4） 对于时钟信号使用 clk 作为信号名，如果设计中存在多个时钟，使用 clk 作为时钟信号的前缀；

（5） 对来自同一驱动源的信号在不同的子模块中采用相同的名字，这要求在芯片总体设计时就定义好顶层子模块间连线的名字，端口和连接端口的信号尽可能采用相同的名字；

（6） 对于低电平有效的信号，应该以一个下划线跟一个小写字母 b 或 n 表示。注意在同一个设计中要使用同一个小写字母表示低电平有效；

（7） 对于复位信号使用 rst 作为信号名，如果复位信号是低电平有效，建议使用 rst_n；

（8） 当描述多比特总线时，使用一致的定义顺序，对于 verilog 建议采用 bus_signal[x:0]的表示；

（9） 尽量遵循业界已经习惯的一些约定。如*_r 表示寄存器输出，*_a 表示异步信号，*_pn 表示多周期路径第 n 个周期使用的信号，*_nxt 表示锁存前的信号，*_z 表示三态信号等；

（10）在源文件、批处理文件的开始应该包含一个文件头、文件头一般包含的内容如下例所示：文件名，作者，模块的实现功能概述和关键特性描述，文件创建和修改的记录，包括修改时间，修改的内容等；

（11）使用适当的注释来解释所有的 always 进程、函数、端口定义、信号含义、变量含义或信号组、变量组的意义等。注释应该放在它所注释的代码附近，要求简明扼要，只要足够说明设计意图即可，避免过于复杂；

（12）每一行语句独立成行。尽管 VHDL 和 Verilog 都允许一行可以写多个语句，当时每个语句独立成行可以增加可读性和可维护性。同时保持每行小于或等于 72 个字符，这样做都是为了提高代码得可读性；

（13）建议采用缩进提高续行和嵌套语句得可读性。缩进一般采用两个空格，如西安交通大学 SOC 设计中心 2 如果空格太多则在深层嵌套时限制行长。同时缩进避免使用 TAB 键，这样可以避免不同机器 TAB 键得设置不同限制代码得可移植能力；

（14）在 RTL 源码的设计中任何元素包括端口、信号、变量、函数、任务、模块等的命名都不能取 Verilog 和 VHDL 语言的关键字；

（15）在进行模块的端口申明时，每行只申明一个端口，并建议采用以下顺序：

输入信号的 clk、rst、enables other control signals、data and address signals。然后再申明输出信号的 clk、rst、enalbes other control signals、data signals；

（16）在例化模块时，使用名字相关的显式映射而不要采用位置相关的映射，这样可以提高代码的可读性和方便 debug 连线错误；

（17）如果同一段代码需要重复多次，尽可能使用函数，如果有可能，可以将函数通用化，以使得它可以复用。注意，内部函数的定义一般要添加注释，这样可以提高代码的可读性；

（18）尽可能使用循环语句和寄存器组来提高源代码的可读性，这样可以有效地减少代码行数；

（19）对一些重要的 always 语句块定义一个有意义的标号，这样有助于调试。注意标号名不要与信号名、变量名重复；

（20）代码编写时的数据类型只使用 IEEE 定义的标准类型，在 VHDL 语言中，设计者可以定义新的类型和子类型，但是所有这些都必须基于 IEEE 的标准；

（21）在设计中不要直接使用数字，作为例外，可以使用 0 和 1。建议采用参数定义代替直接的数字。同时，在定义常量时，如果一个常量依赖于另一个常量，建议在定义该常量时用表达式表示出这种关系；

（22）不要在源代码中使用嵌入式的 dc_shell 综合命令。这是因为其他的综合工具并不认得这些隐含命令，从而导致错误的或较差的综合结果。即使使用 Design Compiler，当综合策略改变时，嵌入式的综合命令也不如放到批处理综合文件中易于维护。这个规则有一个例外的综合命令，即编译开关的打开和关闭可以嵌入到代码中；

（23）在设计中避免实例化具体的门级电路。门级电路可读性差，且难于理解和维护，如果使用特定工艺的门电路，

设计将变得不可移植。如果必须实例化门电路，我们建议采用独立于工艺库的门电路，如 SYNOPSYS 公司提供的 GTECH 库包含了高质量的常用的门级电路；
（24）避免冗长的逻辑和子表达式；
（25）避免采用内部三态电路，建议用多路选择电路代替内部三态电路。

规则 #1: 建立时序逻辑模型时，采用非阻塞赋值语句。
规则 #2: 建立 latch 模型时，采用非阻塞赋值语句。
规则 #3: 在 always 块中建立组合逻辑模型时，采用阻塞赋值语句。
规则 #4: 在一个 always 块中同时有组合和时序逻辑时时，采用非阻塞赋值语句。
规则 #5: 不要在一个 always 块中同时采用阻塞和非阻塞赋值语句。
规则 #6: 同一个变量不要在多个 always 块中赋值。
规则 #7: 调用$strobe 系统函数显示用非阻塞赋值语句赋的值。
规则 #8: 不要使用#0 延时赋值。
组合逻辑
1，敏感变量的描述完备性
Verilog 中，用 always 块设计组合逻辑电路时，在赋值表达式右端参与赋值的所有信号都必须在 always @(敏感电平列表)中列出,always 中 if 语句的判断表达式必须在敏感电平列表中列出。如果在赋值表达式右端引用了敏感电平列表中没有列出的信号，在综合时将会为没有列出的信号隐含地产生一个透明锁存器。这是因为该信号的变化不会立刻引起所赋值的变化，而必须等到敏感电平列表中的某一个信号变化时，它的作用才表现出来，即相当于存在一个透明锁存器，把该信号的变化暂存起来，待敏感电平列表中的某一个
信号变化时再起作用，纯组合逻辑电路不可能作到这一点。综合器会发出警告。
Example1:

```
input a,b,c;
reg e,d;
always @(a or b or c)
    begin
    e=d&a&b; /*d 没有在敏感电平列表中,d 变化时 e 不会立刻变化,直到 a,b,c 中某一个变化*/
    d=e |c;
    end
```

Example2:

```
input a,b,c;
reg e,d;
always @(a or b or c or d)
    begin
    e=d&a&b; /*d 在敏感电平列表中,d 变化时 e 立刻变化*/
    d=e |c;
    end
```

2,条件的描述完备性
如果 if 语句和 case 语句的条件描述不完备，也会造成不必要的锁存器。
Example1:

```
if (a==1'b1) q=1'b1;//如果 a==1'b0,q=? q 将保持原值不变，生成锁存器！
```

Example2:

```
if (a==1'b1) q=1'b1;
else         q=1'b0;//q 有明确的值。不会生成锁存器！
```

Example3:

```
  reg[1:0] a,q;
  ....
  case (a)
```

```
    2'b00 : q=2'b00;
    2'b01 : q=2'b11;//如果 a==2'b10 或 a==2'b11,q=? q 将保持原值不变，锁存器!
  endcase
Example4:
  reg[1:0] a,q;
  ....
  case (a)
    2'b00 : q=2'b00;
    2'b01 : q=2'b11;
    default: q=2'b00;//q 有明确的值。不会生成锁存器!
  endcase
```

Verilog 中端口的描述

1，端口的位宽最好定义在 I/O 说明中,不要放在数据类型定义中；

Example1:

```
module test(addr,read,write,datain,dataout)
input[7:0]   datain;
input[15:0] addr;
input        read,write;
output[7:0] dataout;  //要这样定义端口的位宽!
wire addr,read,write,datain;
reg   dataout;
```

Example2:

```
module test(addr,read,write,datain,dataout)
input   datain,addr,read,write;
output dataout;
wire[15:0] addr;
wire[7:0]   datain;
wire        read,write;
reg[7:0]    dataout;   //不要这样定义端口的位宽!!
```

2，端口的 I/O 与数据类型的关系：

端口的 I/O	端口的数据类型	
	module 内部	module 外部
input	wire	wire 或 reg
output	wire 或 reg	wire
inout	wire	wire

3，assign 语句的左端变量必须是 wire；直接用"="给变量赋值时左端变量必须是 reg!

Example:

```
assign a=b; //a 必须被定义为 wire!!
********
begin
  a=b; //a 必须被定义为 reg!
end
```

VHDL 中 STD_LOGIC_VECTOR 和 INTEGER 的区别

例如 A 是 INTEGER 型，范围从 0 到 255；B 是 STD_LOGIC_VECTOR，定义为 8 位。A 累加到 255 时，再加 1 就一直保持 255 不变，不会自动反转到 0，除非令其为 0；而 B 累加到 255 时，再加 1 就会自动反转到 0。所以在使用时要特别注意!

以触发器为例说明描述的规范性

1，无置位/清零的时序逻辑

```
always @( posedge CLK)
  begin
  Q<=D;
  end
```

2，有异步置位/清零的时序逻辑

异步置位/清零是与时钟无关的，当异步置位/清零信号到来时，触发器的输出立即 被置为 1 或 0，不需要等到时钟沿到来才置位/清零。所以，必须要把置位/清零信号 列入 always 块的事件控制表达式。

```
always @( posedge CLK or negedge RESET)
  begin
  if (!RESET)
    Q=0;
  else
    Q<=D;
  end
```

3，有同步置位/清零的时序逻辑

同步置位/清零是指只有在时钟的有效跳变时刻置位/清零，才能使触发器的输出分 别转换为 1 或 0。所以，不要把置位/清零信号列入 always 块的事件控制表达式。但是 必须在 always 块中首先检查置位/清零信号的电平。

```
always @( posedge CLK )
begin
  if (!RESET)
    Q=0;
  else
    Q<=D;
  end
```

结构规范性

在整个芯片设计项目中，行为设计和结构设计的编码是最重要的一个步骤。它对逻辑综合和布线结果、时序测定、校验能力、测试能力甚至产品支持 都有重要的影响。考虑到仿真器和真实的逻辑电路之间的差异，为了有效的进行仿真测试：

1，避免使用内部生成的时钟

内部生成的时钟称为门生时钟（gated clock）。如果外部输入时钟和门生时钟同时驱动， 则不可避免的两者的步调不一致，造成逻辑混乱。而且，门生时钟将会增加测试的难度 和时间。

2，绝对避免使用内部生成的异步置位/清零信号

内部生成的置位/清零信号会引起测试问题。使某些输出信号被置位或清零，无法正常 测试。

3，避免使用锁存器

锁存器可能引起测试问题。对于测试向量自动生成（ATPG）， 为了使扫描进行，锁存器需要置为透明模式（transparent mode）， 反过来，测试锁存器需要构造特定的向量，这可非同一般。

4，时序过程要有明确的复位值

使触发器带有复位端，在制造测试、ATPG 以及模拟初始化时，可以对整个电路进行 快速复位。

5，避免模块内的三态/双向

内部三态信号在制造测试和逻辑综合过程中难于处理.

近日读 J.Bhasker 的<verilog synthesis practical primer>，受益匪浅,理清了不少基础电路知识，记下一些 tips：

1. 过程赋值(always 中触发赋值)的变量,可能会被综合成连线 或触发器 或锁存器.

2.综合成锁存器的规则:

a. 变量在条件语句(if 或 case)中,被赋值.

b. 变量未在条件语句的所有分支中被赋值.

c. 在 always 语句多次调用之间需要保持变量值 .

以上三个条件必须同时满足.

3.综合成触发器的规则:

变量在时钟沿的控制下被赋值。

例外情况：变量的赋值和引用都仅出现在一条 always 语句中，则该变量被视为中

间变量而不是触发器。

4. 对于无时钟事情的 always 语句（即组合逻辑建模），其时间表应包括该 alwa 语

句引用的所有变量，否则会出现 RTL 与 Netlist 的不一致

芯片外部引脚很多都使用 inout 类型的，为的是节省管腿。一般信号线用做总线等双向数据传输的时候就要用到 INOUT 类型了。就是一个端口同时做输入和输出。 inout 在具体实现上一般用三态门来实现。三态门的第三个状态就是高阻'Z'。当 inout 端口不输出时，将三态门置高阻。这样信号就不会因为两端同时输出而出错了,更详细的内容可以搜索一下三态门 tri-state 的资料.
1 使用 inout 类型数据,可以用如下写法:
inout data_inout;
input data_in;
reg data_reg;//data_inout 的映象寄存器
reg link_data;
assign data_inout=link_data?data_reg:1’bz;//link_data 控制三态门
//对于 data_reg,可以通过组合逻辑或者时序逻辑根据 data_in 对其赋值.通过控制 link_data 的高低电平,从而设置 data_inout 是输出数据还是处于高阻态,如果处于高阻态,则此时当作输入端口使用.link_data 可以通过相关电路来控制.
2 编写测试模块时,对于 inout 类型的端口,需要定义成 wire 类型变量,而其它输入端口都定义成 reg 类型,这两者是有区别的.
当上面例子中的 data_inout 用作输入时,需要赋值给 data_inout,其余情况可以断开.此时可以用 assign 语句实现:assign data_inout=link?data_in_t:1’bz;其中的 link ,data_in_t 是 reg 类型变量,在测试模块中赋值.
另外,可以设置一个输出端口观察 data_inout 用作输出的情况:
Wire data_out;
Assign data_out_t=(!link)?data_inout:1’bz;

else，in RTL
inout use in top module(PAD)
dont use inout(tri) in sub module
也就是说，在内部模块最好不要出现 inout，如果确实需要，那么用两个 port 实现，到顶层的时候再用三态实现。理由是：在非顶层模块用双向口的话，该双向口必然有它的上层跟它相连。既然是双向口，则上层至少有一个输入口和一个输出口联到该双向口上，则发生两个内部输出单元连接到一起的情况出现，这样在综合时往往会出错。

对双向口，我们可以将其理解为 2 个分量：一个输入分量，一个输出分量。另外还需要一个控制信号控制输出分量何时输出。此时，我们就可以很容易地对双向端口建模。

例子：
CODE:
module dual_port (
....
inout_pin,
....
);

inout inout_pin;

wire inout_pin;

wire input_of_inout;
wire output_of_inout;
wire out_en;

assign input_of_inout = inout_pin;

assign inout_pin = out_en ? output_of_inout : 高阻;

endmodule

可见，此时 input_of_inout 和 output_of_inout 就可以当作普通信号使用了。

在仿真的时候，需要注意双向口的处理。如果是直接与另外一个模块的双向口连接，那么只要保证一个模块在输出的时候，另外一个模块没有输出（处于高阻态）就可以了。
如果是在 **ModelSim** 中作为单独的模块仿真，那么在模块输出的时候，不能使用 force 命令将其设为高阻态，而是使用 release 命令将总线释放掉

很多初学者在写 testbench 进行仿真和验证的时候，被 inout 双向口难住了。仿真器老是提示错误不能进行。下面是我个人对 inout 端口写 testbench 仿真的一些总结，并举例进行说明。在这里先要说明一下 inout 口在 testbench 中要定义为 wire 型变量。

先假设有一源代码为：

module xx(data_inout , ...);

inout data_inout;

.....

assign data_inout=(! link)?datareg:1'bz;

endmodule

方法一：使用相反控制信号 inout 口，等于两个模块之间用 inout 双向口互连。这种方法要注意 assign 语句只能放在 initial 和 always 块内。

module test();

wire data_inout;

reg data_reg;

reg link;

initial begin

.....

end

assign data_inout=link?data_reg:1'bz;

endmodule

方法二：使用 force 和 release 语句，但这种方法不能准确反映双向端口的信号变化，但这种方法可以反在块内。

module test();

wire data_inout;

```
reg data_reg;

reg link;

#xx;        //延时

force data_inout=1'bx;          //强制作为输入端口

.....

#xx;

release data_inout;         //释放输入端口

endmodule
```

很多读者反映仿真双向端口的时候遇到困难，这里介绍一下双向端口的仿真方法。一个典型的双向端口如图 1 所示。

其中 inner_port 与芯片内部其他逻辑相连，outer_port 为芯片外部管脚，out_en 用于控制双向端口的方向，out_en 为 1 时，端口为输出方向，out_en 为 0 时，端口为输入方向。

用 **Verilog** 语言描述如下：

```
module bidirection_io(inner_port,out_en,outer_port);
input out_en;
inout[7:0] inner_port;
inout[7:0] outer_port;
assign outer_port=(out_en==1)?inner_port:8'hzz;
assign inner_port=(out_en==0)?outer_port:8'hzz;
endmodule
```

用 **VHDL** 语言描述双向端口如下：

```
library ieee;
use IEEE.STD_LOGIC_1164.ALL;
entity bidirection_io is
port ( inner_port : inout std_logic_vector(7 downto 0);
out_en : in std_logic;
outer_port : inout std_logic_vector(7 downto 0) );
end bidirection_io;
architecture behavioral of bidirection_io is
begin
outer_port<=inner_port when out_en='1' else (OTHERS=>'Z');
inner_port<=outer_port when out_en='0' else (OTHERS=>'Z');
end behavioral;
```

仿真时需要验证双向端口能正确输出数据，以及正确读入数据，因此需要驱动 out_en 端口，当 out_en 端口为 1 时，testbench 驱动 inner_port 端口，然后检查 outer_port 端口输出的数据是否正确；当 out_en 端口为 0 时，testbench 驱动 outer_port 端口，然后检查 inner_port 端口读入的数据是否正确。由于 inner_port 和 outer_port 端口都是双向端口（在 VHDL 和 Verilog 语言中都用 inout 定义），因此驱动方法与单向端口有所不同。
验证该双向端口的 testbench 结构如图 2 所示。

这是一个 self-checking testbench，可以自动检查仿真结果是否正确，并在 Modelsim 控制台上打印出提示信息。图中 Monitor 完成信号采样、结果自动比较的功能。
testbench 的工作过程为

1）out_en=1 时，双向端口处于输出状态，testbench 给 inner_port_tb_reg 信号赋值，然后读取 outer_port_tb_wire 的值，如果两者一致，双向端口工作正常。
2）out_en=0 时，双向端口处于输如状态，testbench 给 outer_port_tb_reg 信号赋值，然后读取 inner_port_tb_wire 的值，如果两者一致，双向端口工作正常。

用 Verilog 代码编写的 testbench 如下，其中使用了自动结果比较，随机化激励产生等技术。

```
`timescale 1ns/10ps
module tb();
reg[7:0] inner_port_tb_reg;
wire[7:0] inner_port_tb_wire;
reg[7:0] outer_port_tb_reg;
wire[7:0] outer_port_tb_wire;
reg out_en_tb;
integer i;

initial
begin
out_en_tb=0;
inner_port_tb_reg=0;
outer_port_tb_reg=0;
i=0;
repeat(20)
begin
#50
i=$random;
out_en_tb=i[0]; //randomize out_en_tb
inner_port_tb_reg=$random; //randomize data
outer_port_tb_reg=$random;
end
end

//**** drive the ports connecting to bidirction_io
assign inner_port_tb_wire=(out_en_tb==1)?inner_port_tb_reg:8'hzz;
assign outer_port_tb_wire=(out_en_tb==0)?outer_port_tb_reg:8'hzz;

//instatiate the bidirction_io module
bidirection_io bidirection_io_inst(.inner_port(inner_port_tb_wire),
.out_en(out_en_tb),
.outer_port(outer_port_tb_wire));

//***** monitor ******
always@(out_en_tb,inner_port_tb_wire,outer_port_tb_wire)
begin
#1;
if(outer_port_tb_wire===inner_port_tb_wire)
begin
$display("\n **** time=%t ****",$time);
```

```
$display("OK! out_en=%d",out_en_tb);
$display("OK! outer_port_tb_wire=%d,inner_port_tb_wire=%d",
outer_port_tb_wire,inner_port_tb_wire);
end
else
begin
$display("\n **** time=%t ****",$time);
$display("ERROR! out_en=%d",out_en_tb);
$display("ERROR! outer_port_tb_wire != inner_port_tb_wire" );
$display("ERROR! outer_port_tb_wire=%d, inner_port_tb_wire=%d",
outer_port_tb_wire,inner_port_tb_wire);
end
end
endmodule
```

今天重新回顾了一下阻塞赋值和非阻塞赋值的概念，感觉又有所收获。

一、特点：

阻塞赋值：1、RHS 的表达式计算和 LHS 的赋值更新，这两个动作之间不能插入其他动作，即所谓计算完毕，立即更新。

2、所谓阻塞就是指在一个“begin...end”块中的多个阻塞赋值语句内，只有上一句完全执行完毕后，才会执行下一语句，否则阻塞程序的执行。

非阻塞赋值：RHS 的表达式计算和 LHS 的赋值更新分两个节拍执行，首先，应该是 RHS 的表达式计算，得到新值后并不立即赋值，而是放在事件队列中等待，直到

当前仿真时刻的后期才执行（原因下文会提到）。

二、Verilog 的分层事件队列：

在 Verilog 中，事件队列可以划分为 5 个不同的区域，不同的事件根据规定放在不同的区域内，按照优先级的高低决定执行的先后顺序，下表就列出了部分 Verilog 分层事件队列。其中，活跃事件的优先级最高（最先执行），而监控事件的优先级最低，而且在活跃事件中的各事件的执行顺序是随机的（注：为方便起见，在一般的仿真器中，对同一区域的不同事件是按照调度的先后关系执行的）。

活跃事件	阻塞赋值，非阻塞赋值的 RHS 计算……
非活跃事件	显式 0 延时的阻塞赋值……
非阻塞赋值更新事件	由非阻塞语句产生的一个非阻塞赋值更新事件，并被调入当前仿真时刻。
监控事件	$monitor 和$strobe 等系统任务
	被调度到将来仿真时间的事件

三、结论：

由上表就可以知道，阻塞赋值属于活跃事件，会立刻执行，这就是阻塞赋值“计算完毕，立刻更新”的原因。此外，由于在分层事件队列中，只有将活跃事件中排在前面的事件调出，并执行完毕后，才能够执行下面的事件，这就可以

解释阻塞赋值的第二个特点。

同样是由上表知，非阻塞赋值的 RHS 计算属于活跃事件，而非阻塞赋值更新事件排在非活跃事件之后，因此只有仿真队列中所有的活跃事件和非活跃事件都执行完毕后，才轮到非阻塞赋值更新事件，这就是非阻塞赋值必须分两拍完成的原因。

以上就是我今天的读书笔记，写得仓促，如有不对，敬请指出。

一. 强调 Verilog 代码编写风格的必要性。

强调 Verilog 代码编写规范，经常是一个不太受欢迎的话题，但却是非常有必要的。

每个代码编写者都有自己的编写习惯，而且都喜欢按照自己的习惯去编写代码。与自己编写风格相近的代码，阅读起来容易接受和理解。相反和自己编写风格差别较大的代码，阅读和接受起来就困难一些。

曾有编程大师总结说，一个优秀的程序员，能维护的代码长度大约在 1 万行数量级。代码的整洁程度，很大程度上影响着代码的维护难度。

遵循代码编写规范书写的代码，很容易阅读、理解、维护、修改、跟踪调试、整理文档。相反代码编写风格随意的代码，通常晦涩、凌乱，会给开发者本人的调试、修改工作带来困难，也会给合作者带来很大麻烦。

（实际上英文 Coding Style 有另一层涵义，更偏重的是，某一个电路，用那一种形式的语言描述，才能将电路描述得更准确，综合以后产生的电路更合理。本文更偏重的是，编写 Verilog 代码时的书写习惯。）

二. 强调编写规范的宗旨。

缩小篇幅

提高整洁度

便于跟踪、分析、调试

增强可读性，帮助阅读者理解

便于整理文档

便于交流合作

三. 变量及信号命名规范。

1. 系统级信号的命名。

系统级信号指复位信号，置位信号，时钟信号等需要输送到各个模块的全局信号；系统信号以字符串 Sys 开头。

2. 低电平有效的信号后一律加下划线和字母 n。如：SysRst_n；FifoFull_n；

3. 经过锁存器锁存后的信号，后加下划线和字母 r，与锁存前的信号区别。如 CpuRamRd 信号，经锁存后应命名为 CpuRamRd_r。

低电平有效的信号经过锁存器锁存后，其命名应在_n 后加 r。如 CpuRamRd_n 信号，经锁存后应命名为 CpuRamRd_nr

多级锁存的信号，可多加 r 以标明。如 CpuRamRd 信号，经两级触发器锁存后，应命名为 CpuRamRd_rr。

4. 模块的命名。

在系统设计阶段应该为每个模块进行命名。命名的方法是，将模块英文名称的各个单词首字母组合起来，形成 3 到 5 个字符的缩写。若模块的英文名只有一个单词，可取该单词的前 3 个字母。各模块的命名以 3 个字母为宜。例如：

Arithmatic Logical Unit 模块，命名为 ALU。

Data Memory Interface 模块，命名为 DMI。

Decoder 模块，命名为 DEC。

5. 模块之间的接口信号的命名。

所有变量命名分为两个部分，第一部分表明数据方向，其中数据发出方在前，数据接收方在后，第二部分为数据名称。两部分之间用下划线隔离开。

第一部分全部大写，第二部分所有具有明确意义的英文名全部拼写或缩写的第一个字母大写，其余部分小写。
举例：CPUMMU_WrReq，下划线左边是第一部分，代表数据方向是从 CPU 模块发向存储器管理单元模块（MMU）。下划线右边 Wr 为 Write 的缩写，Req 是 Request 的缩写。两个缩写的第一个字母都大写，便于理解。整个变量连起来的意思就是 CPU 发送给 MMU 的写请求信号。
模块上下层次间信号的命名也遵循本规定。
若某个信号从一个模块传递到多个模块，其命名应视信号的主要路径而定。

6. 模块内部信号：
模块内部的信号由几个单词连接而成，缩写要求能基本表明本单词的含义；
单词除常用的缩写方法外（如：Clock->Clk, Write->Wr, Read->Rd 等），一律取该单词的前几个字母（ 如：Frequency->Freq, Variable->Var 等）；
每个缩写单词的第一个字母大写；
若遇两个大写字母相邻，中间添加一个下划线（如 DivN_Cntr）；
举例：SdramWrEn_n；FlashAddrLatchEn;

四. 编码格式规范。

1. 分节书写，各节之间加 1 到多行空格。如每个 always,initial 语句都是一节。每节基本上完成一个特定的功能，即用于描述某几个信号的产生。在每节之前有几行注释对该节代码加以描述，至少列出本节中描述的信号的含义。

2. 行首不要使用空格来对齐，而是用 Tab 键，Tab 键的宽度设为 4 个字符宽度。行尾不要有多余的空格。

3. 注释。
使用//进行的注释行以分号结束；
使用/* */进行的注释，/*和*/各占用一行，并且顶头；
例：
// Edge detector used to synchronize the input signal；

4. 空格的使用：
不同变量，以及变量与符号、变量与括号之间都应当保留一个空格。
Verilog 关键字与其它任何字符串之间都应当保留一个空格。如：
Always @ (……)
使用大括号和小括号时，前括号的后边和后括号的前边应当留有一个空格。
逻辑运算符、算术运算符、比较运算符等运算符的两侧各留一个空格，与变量分隔开来；单操作数运算符例外，直接位于操作数前，不使用空格。
使用//进行的注释，在//后应当有一个空格；注释行的末尾不要有多余的空格。
例：
```
assign SramAddrBus = { AddrBus[31:24], AddrBus[7:0] };
assign DivCntr[3:0] = DivCntr[3:0] + 4’b0001;
assign Result = ~Operand;
```

5. 同一个层次的所有语句左端对齐；Initial、always 等语句块的 begin 关键词跟在本行的末尾，相应的 end 关键词与 Initial、always 对齐；这样做的好处是避免因 begin 独占一行而造成行数太多；
例：
```
always @ ( posedge SysClk or negedge SysRst ) begin
if( !SysRst ) DataOut <= 4'b0000;
else if( LdEn ) begin
DataOut <= DataIn;
End
else DataOut <= DataOut + 4'b0001;
end
```

6. 不同层次之间的语句使用 Tab 键进行缩进，每加深一层缩进一个 Tab；

8. 在 endmodule，endtask，endcase 等标记一个代码块结束的关键词后面要加上一行注释说明这个代码块的名称；

9. 在 task 名称前加 tsk 以示标记。在 function 的名称前加 func 以示标记。例如：
task tskResetSystem；
……
endtask //of tskResetSystem

五．小结：
以上列出的代码编写规范无法覆盖代码编写的方方面面，还有很多细节问题，需要在实际编写过程中加以考虑。并且有些规定也不是绝对的，需要灵活处理。并不是律条，但是在一个项目组内部、一个项目的进程中，应该有一套类似的代码编写规范来作为约束。
总的方向是，努力写整洁、可读性好的代码

二．reg 型

在“always”块内被赋值的每一个信号都必须定义成 reg 型。
reg 型数据的缺省初始值是不定值。
reg 型只表示被定义的信号将用在“always”块内，理解这一点很重要。并不是说 reg 型信号一定是寄存器或触发器的输出。虽然 reg 型信号常常是寄存器或触发器的输出，但并不一定总是这样。

三．memory 型

memory 型数据是通过扩展 reg 型数据的地址范围来生成的。其格式如下：

```
reg [n-1:0] 存储器名[m-1:0];
或 reg [n-1:0] 存储器名[m:1];
```

在这里，reg[n-1:0]定义了存储器中每一个存储单元的大小，即该存储单元是一个 n 位的寄存器。存储器名后的[m-1:0]或[m:1]则定义了该存储器中有多少个这样的寄存器。

```
reg [7:0] mema[255：0];
```

这个例子定义了一个名为 mema 的存储器，该存储器有 256 个 8 位的存储器。该存储器的地址范围是 0 到 255。
注意：对存储器进行地址索引的表达式必须是常数表达式。
尽管 memory 型数据和 reg 型数据的定义格式很相似，但要注意其不同之处。如一个由 n 个 1 位寄存器构成的存储器组是不同于一个 n 位的寄存器的。见下例：

```
reg [n-1:0] rega;     //一个 n 位的寄存器
reg mema [n-1:0];     //一个由 n 个 1 位寄存器构成的存储器组
```

一个 n 位的寄存器可以在一条赋值语句里进行赋值，而一个完整的存储器则不行。见下例：

```
rega =0;    //合法赋值语句
mema =0;    //非法赋值语句
```

如果想对 memory 中的存储单元进行读写操作，必须指定该单元在存储器中的地址。下面的写法是正确的。

```
mema[3]=0; //给 memory 中的第 3 个存储单元赋值为 0。
```

3.3.1.基本的算术运算符

在 Verilog HDL 语言中，算术运算符又称为二进制运算符，共有下面几种：

1) + (加法运算符,或正值运算符,如 rega＋regb，＋3)
2) －(减法运算符，或负值运算符，如 rega－3,－3)
3) ×(乘法运算符，如 rega*3)
4) / (除法运算符，如 5/3)
5) % (模运算符，或称为求余运算符，要求%两侧均为整型数据。如 7%3 的值为 1)

注意： 在进行算术运算操作时，如果某一个操作数有不确定的值 x，则整个结果也为不定值 x。

1) ~ //取反
2) & //按位与
3) | //按位或
4) ^ //按位异或
5) ^~ //按位同或(异或非)

在 Verilog HDL 语言中存在三种逻辑运算符:

1) && 逻辑与
2) || 逻辑或
3) ! 逻辑非

关系运算符共有以下四种：

a < b a 小于 b
a > b a 大于 b
a <= b a 小于或等于 b
a >= b a 大于或等于 b

3.3.5.等式运算符

3.3.6.移位运算符

3.3.7.位拼接运算符(Concatation)

3.3.10.关键词

在 Verilog HDL 中，所有的关键词是事先定义好的确认符,用来组织语言结构。关键词是用小写字母定义的,因此在编写原程序时要注意关键词的书写,以避免出错。下面是 Verilog HDL 中使用的关键词(请参阅附录：Verilog 语言参考手册)：

always, and, assign, begin, buf, bufif0, bufif1, case, casex, casez, cmos, deassign, default, defparam, disable, edge, else, end, endcase, endmodule, endfunction, endprimitive, endspecify, endtable, endtask, event, for, force, forever, fork, function, highz0, highz1, if, initial, inout, input, integer, join, large, macromodule, medium, module, nand, negedge, nmos, nor, not, notif0, notif1, or, output, parameter, pmos, posedge, primitive, pull0, pull1, pullup, pulldown, rcmos, reg, releses, repeat, mmos, rpmos, rtran, rtranif0, rtranif1, scalared, small, specify, specparam, strength, strong0, strong1, supply0, supply1, table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg, vectored, wait, wand, weak0, weak1, while, wire, wor, xnor, xor

(1).非阻塞(Non_Blocking)赋值方式(如 b <= a;)

1) 块结束后才完成赋值操作。
2) **b 的值并不是立刻就改变的。**
3) 这是一种比较常用的赋值方法。（特别在编写可综合模块时）

(2).阻塞(Blocking)赋值方式(如 b = a;)

1) 赋值语句执行完后,块才结束。
2) **b 的值在赋值语句执行完后立刻就改变的。**
3) 可能会产生意想不到的结果。

一. 顺序块

顺序块有以下特点：

1) 块内的语句是按顺序执行的，即只有上面一条语句执行完后下面的语句才能执行。
2) 每条语句的延迟时间是相对于前一条语句的仿真时间而言的。
3) 直到最后一条语句执行完，程序流程控制才跳出该语句块。

顺序块的格式如下：

```
begin
    语句 1;
    语句 2;
    ......
    语句 n;
end
```

其中：

- 块名即该块的名字，一个标识名。其作用后面再详细介绍。
- 块内声明语句可以是参数声明语句、reg 型变量声明语句、integer 型变量声明语句、real 型变量声明语句。

二. 并行块

并行块有以下四个特点：

1) 块内语句是同时执行的，即程序流程控制一进入到该并行块，块内语句则开始同时并行地执行。
2) 块内每条语句的延迟时间是相对于程序流程控制进入到块内时的仿真时间的。
3) 延迟时间是用来给赋值语句提供执行时序的。
4) 当按时间时序排序在最后的语句执行完后或一个 disable 语句执行时，程序流程控制跳出该程序块。

并行块的格式如下：

```
fork
    语句 1;
    语句 2;
    .......
    语句 n;
join
```

其中：

- 块名即标识该块的一个名字，相当于一个标识符。
- 块内说明语句可以是参数说明语句、reg 型变量声明语句、integer 型变量声明语句、real 型变量声明语句、time 型变量声明语句、事件(event)说明语句。

在 fork_join 块内，各条语句不必按顺序给出，因此在并行块里，各条语句在前还是在后是无关紧要的。见下例：

三. 块名

在 VerilgHDL 语言中，可以给每个块取一个名字，只需将名字加在关键词 begin 或 fork 后面即可。这样做的原因有以下几点。

1) 这样可以在块内定义局部变量，即只在块内使用的变量。

2) 这样可以允许块被其它语句调用，如被 disable 语句。
3) 在 Verilog 语言里，所有的变量都是静态的，即所有的变量都只有一个唯一的存储地址，因此进入或跳出块并不影响存储在变量内的值。

基于以上原因，块名就提供了一个在任何仿真时刻确认变量值的方法。

casez 语句用来处理不考虑高阻值 z 的比较过程，casex 语句则将高阻值 z 和不定值都视为不必关心的情况。

如果用到 if 语句，最好写上 else 项。如果用 case 语句，最好写上 default 项。遵循上面两条原则，就可以避免发生这种错误，使设计者更加明确设计目标，同时也增强了 Verilog 程序的可读性。

3.6.循环语句

在 Verilog HDL 中存在着四种类型的循环语句，用来控制执行语句的执行次数。

1) forever 连续的执行语句。
2) repeat 连续执行一条语句 n 次。
3) while 执行一条语句直到某个条件不满足。如果一开始条件即不满足(为假)，则语句一次也不能被执行。
4) for 通过以下三个步骤来决定语句的循环执行。
   a) 先给控制循环次数的变量赋初值。
   b) 判定控制循环的表达式的值，如为假则跳出循环语句，如为真则执行指定的语句后，转到第三步。
   c)

＃1:当为时序逻辑建模，使用“非阻塞赋值”。
＃2:当为锁存器（latch）建模，使用“非阻塞赋值”。
＃3:当用 always 块为组合逻辑建模，使用“阻塞赋值”
＃4:当在同一个 always 块里面既为组合逻辑又为时序逻辑建模，使用“非阻塞赋值”。
＃5:不要在同一个 always 块里面混合使用“阻塞赋值”和“非阻塞赋值”。
＃6:不要在两个或两个以上 always 块里面对同一个变量进行赋值。
＃7:使用$strobe 以显示已被“非阻塞赋值”的值。
＃8:不要使用＃0 延迟的赋值。
#9:在 VERILOG 语法中， if...else if ... else 语句是有优先级的，一般说来第一个 IF 的优先级最高，最后一个 ELSE 的优先级最低。如果描述一个编码器，在 XILINX 的 XST 综合参数就有一个关于优先级编码器硬件原语句的选项 Priority Encoder Extraction. 而 CASE 语句是"平行"的结构，所有的 CASE 的条件和执行都没有“优先级”。而建立优先级结构会消耗大量的组合逻辑，所以如果能够使用 CASE 语句的地方，尽量使用 CASE 替换 IF...ELSE 结构。
#10:XILINX 的底层可编程硬件资源叫 SLICE，由 2 个 FF 和 2 个 LUT 组成。 FF 触发器 LUT 查找表
ALTERA 的底层可编程硬件资源叫 LE，由 1 个 FF 和 1 个 LUT 组成。

#11:慎用锁存器(latch)，同步时序设计要尽量避免使用锁存器，综合出非目的性 latch 的主要原因在于不完全的条件判断句。另外一种情况是设计中有组合逻辑的反馈环路(combinatorial feedback loops)。
#12:状态机的一般设计原则，Biary, gray-code 编码使用最少的触发器，较多的组合逻辑。而 one-hot 编码反之。所以 **CPLD** 多使用 GRAY-CODE, 而 **FPGA** 多使用 ONE-HOT 编码。另一方面，小型设计使用 GRAY-CODE 和 BINARY 编码更有效，而大型状态机使用 ONE-HOT 更有效。
#13:业界主流 CPLD 产品是 lattice 的 LC4000 系列和 ALTERA 的 MAX3000 系列。
#14:复位使初始状态可预测，防止出现禁用状态。FPGA 和 CPLD 的复位信号采用异步低电平有效信号，连接到其全局复位输入端，使用专用路径通道,复位信号必须连接到 FPGA 和 CPLD 的全局复位管脚。。
#15:不要用时钟或复位信号作数据或使能信号，也不能用数据信号作为时钟或复位信号，否则 HDL 综合时会出现时序验证问题。信号穿过时钟的两半个周期时，要在前后分别取样；防止出现半稳定状态。
#16：fpga 设计中 不要使用门时钟（don't use gated clock）。时钟信号必须连接到全局时钟管脚上。
#17：不要使用内部三态信号，否则增加功耗。
#18：只使用同步设计，不要使用延时单元。
#19：避免使用负延触发的双稳态多谐振荡器（flip flop）。
#20:不要使用信号和变量的默认值（或初始值），用复位脉冲初始化
信号和变量。
#21:不要在代码中使用 buffer 类型的端口读取输出数据；要使用 out 类型，再增加另外变量或信号，以获取输出值。这是因为 buffer 类型的端口不能连接到其他类型的端口上，因此 buffer 类型就会在整个设计的端口中传播下去。
#22:对变量要先读后写；如果先写后读，就会产生长的组合逻辑和锁存器（或寄存器）。这是因为变量值是立即获取的。
#23:在组合逻辑进程中，其敏感向量标中要包含所有要读取得信号；这是为了防止出现不必要的锁存器。

近期，在 stephen Brown 的一本书数字逻辑基础与 **verilog 设计**一书中看到关于触发器电路的时序分析。以前一直没有搞明白这个问题，现在觉得豁然开朗。怕忘记了，特地摘抄与此与 edacn 网友分享。

触发器电路的时序分析：

图 7-84 给出了一个使用 D 触发器的简单电路。我们想要计算该电路能正常工作的最大的时钟频率 Fmax，并且想确定该电路的保持时间是否不够长。在**技术**文献中，这种类型的电路分析通常叫做**时序分析**。假设该触发器的时序参数为：tsu=0.6ns，th=0.4ns，0.8ns<=tcQ<=1.0ns。给 tcq 参数规定一个范围是因为延迟参数分布在一定范围内，这样处理是现成集成电路芯片常用的方法。为了计算最小的时钟信号周期 Tmin=1/Fmax，我们必须考虑在触发器中从开始到结束的所有路径。在这个简单的电路中，只有一条这样的路径，这条路径开始于数据被时钟信号的正跳变沿加载进入触发器，经过 tcQ 的延迟后传播到 Q 的输出端，再传播通过非门，同时必须满足 D 输入端的建立时间要求。因此：

Tmin=tcQ+tNOT+tsu

由于我们关注的只是计算出最长的延迟时间，所以应该用 tcQ 的最大值。为了计算出 tNOT，我们将假设通过任何逻辑门的延迟都可以用 1+0.1k 进行计算，其中 k 是该门的输入信号的个数。对非门而言，k=1，因此得到如下 Tmin 和 Fmax 的值：Tmin=1.0+1.1+0.6=2.7ns

Fmax=1/2.7ns=370.37MHz

当然，有必要检查电路中的保持时间是否违反规定。在这种场合，我们必须核查从时钟信号的正跳变沿到 D 输入值改变的最短延迟。该延迟由 tcQ+tNOT=0.8+1.1=1.9ns 给定。因为 1.9ns>0.4ns，所以保持时间够长，没有违反规定。再举一个触发器电路时序分析的例子，请考虑图 7-85 所示的计数器电路。假设所用的触发器的时序参数与图 7-84 中用过的触发器相同，请计算该电路能正常运行的最高频率。再次假设通过逻辑门的传播延迟可以用 1+0.1k 来计算。

在这个电路中，存在着四个触发器从开始到结束的许多路径。最长的路径从触发器 Q0 起到触发器 Q3 结束。在某个电路中最长的路径成为**关键路径**。关键路径的延迟包括触发器 Q0 的时钟信号到 Q 的延迟、通过三个与门的传播延迟和一个异或门的延迟。我们还必须考虑触发器 Q3 的建立时间。因此，得到

Tmin=tcQ+3(tAND)+tXOR+tsu

用 tcQ 的最大值，得到

Tmin=1.0+3(1.2)+1.2+0.6=6.4ns

Fmax=1/6.4ns=156.25MHz

该电路的最短路径是从每个触发器通过异或门反馈到该触发器本身的输入端。沿每个这样路径的最小延迟为 tcQ+tXOR=0.8+1.2=2.0ns。因为 2.0ns>th=0.4ns，因此保持时间足够长，没有违反规定。

在上面的分析中，假设时钟信号同时到达所有四个触发器。我们现在将重复这个分析，假设时钟信号同时到达触发器 Q0,Q1,Q2，但到达触发器 Q3 有一些延迟。始终到达不同的触发器之间的时间差称为**时钟偏差**(clock skew)，记作 tskew，时钟偏差可以由许多原因引起。

在图 7-85 中，电路的关键路径是从触发器 Q0 起到触发器 Q3。然而，Q3 的时钟偏差使得这个延迟减少，因为时钟偏差在数据被加载进该触发器前提供了附加的时间。如果考虑增加 1.5ns 的时钟偏差，则从触发器 Q0 到触发器 Q3 的路径延迟由 tcQ+3(tAND)+tXOR+tsu-tskew=6.4-1.5ns=4.9ns 给定。该电路现在还存在一个不同的关键路径，该路径从触发器 Q0 起到触发器 Q2 结束。这条路径的延迟为

Tmin=tcQ+2(tAND)+tXOR+tsu=1.0+2(1.2)+1.2+0.6ns=5.2ns

Fmax=1/5.2ns=192.31MHz

在这种场合，时钟偏差导致电路的最高时钟频率提高。但是，如果时钟偏差是负的，即触发器 Q3 的时钟到达时间比其他触发器更早一些，则会造成该电路的最高时钟频率 Fmax 降低。

因为数据加载到触发器 Q3 被时钟偏差延迟了，所以对所有起始于 Q0，Q1，Q2 而以 Q3 为结束点的路径，都会产生使触发器 Q3 的保持时间需要增加到 th+tskew 的影响。在该电路中，这种最短的路径是从触发器 Q2 到 Q3 的路径，其延迟时间为 TcQ+tAND+tXOR=0.8+1.2+1.2=3.2ns。因为 3.2ns>th+tskew=1.9ns，所以保持时间足够长，没有违反规定。

如果对时钟偏差值 tskew>=3.2-th=2.8ns，重复以上保持时间的分析，则会出现保持时间不够的情况。当 tskew>=2.8ns 时，该电路将不可能在任何频率下可靠地运行。由于时钟偏差的存在会引起电路时序问题，所以好的**电路设计**方法必须保证时钟信号到达所有触发器的偏差尽可能小。

最后是我的总结：确定最小周期是找关键路径即最长路径。确定 Th 是否违例是找最短路径。最短路径要大于 Th。如果有 Tskew 的情况则要大于 Th+Tskew（有 skew 的寄存器为最短路径的终点的时候）

还有就是我对有 Tskew 的情况的时候为什么防止违例要最短路径>Th+Tskew。因为 Q0，Q1 和 Q2 时钟比 Q3 早，以他们为起点的路径已经开始走了一段时间后 Q3 的时钟才到才开始打入数据。所以保持时间上要加上这段 skew

ISE 约束文件的基本操作

1．约束文件的概念

FPGA 设计中的约束文件有 3 类：用户设计文件（.UCF 文件）、网表约束文件（.NCF 文件）以及物理约束文件（.PCF 文件），可以完成时序约束、管脚约束以及区域约束。3 类约束文件的关系为：用户在设计输入阶段编写 **UCF** 文件，然后 UCF 文件和设计综合后生成 NCF 文件，最后再经过实现后生成 PCF 文件。本节主要介绍 UCF 文件的使用方法。

UCF 文件是 ASC 2 码文件，描述了逻辑设计的约束，可以用文本编辑器和 Xilinx 约束文件编辑器进行编辑。NCF 约束文件的语法和 UCF 文件相同，二者的区别在于： UCF 文件由用户输入，NCF 文件由综合工具自动生成，当二者发生冲突时，以 UCF 文件为准，这是因为 UCF 的优先级最高。PCF 文件可以分为两个部分：一部分是映射产生的物理约束，另一部分是用户输入的约束，同样用户约束输入的优先级最高。一般情况下，用户约束都应在 UCF 文件中完成，不建议直接修改 NCF 文件和 PCF 文件。

2．创建约束文件

约束文件的后缀是.ucf，所以一般也被称为 UCF 文件。创建约束文件有两种方法，一种是通过新建方式，另一种则是

利用过程管理器来完成。

第一种方法：新建一个源文件，在代码类型中选取“Implementation Constrains File”，在“File Name”中输入“one2two_ucf”。单击“Next”按键进入模块选择对话框，选择模块“one2two”，然后单击“Next”进入下一页，再单击“Finish”按键完成约束文件的创建。

第二种方法：在工程管理区中，将“Source for”设置为“Synthesis/Implementation”。“Constrains Editor”是一个专用的约束文件编辑器，双击过程管理区中“User Constrains”下的“Create Timing Constrains”就可以打开“Constrains Editor”，其界面如图所示：

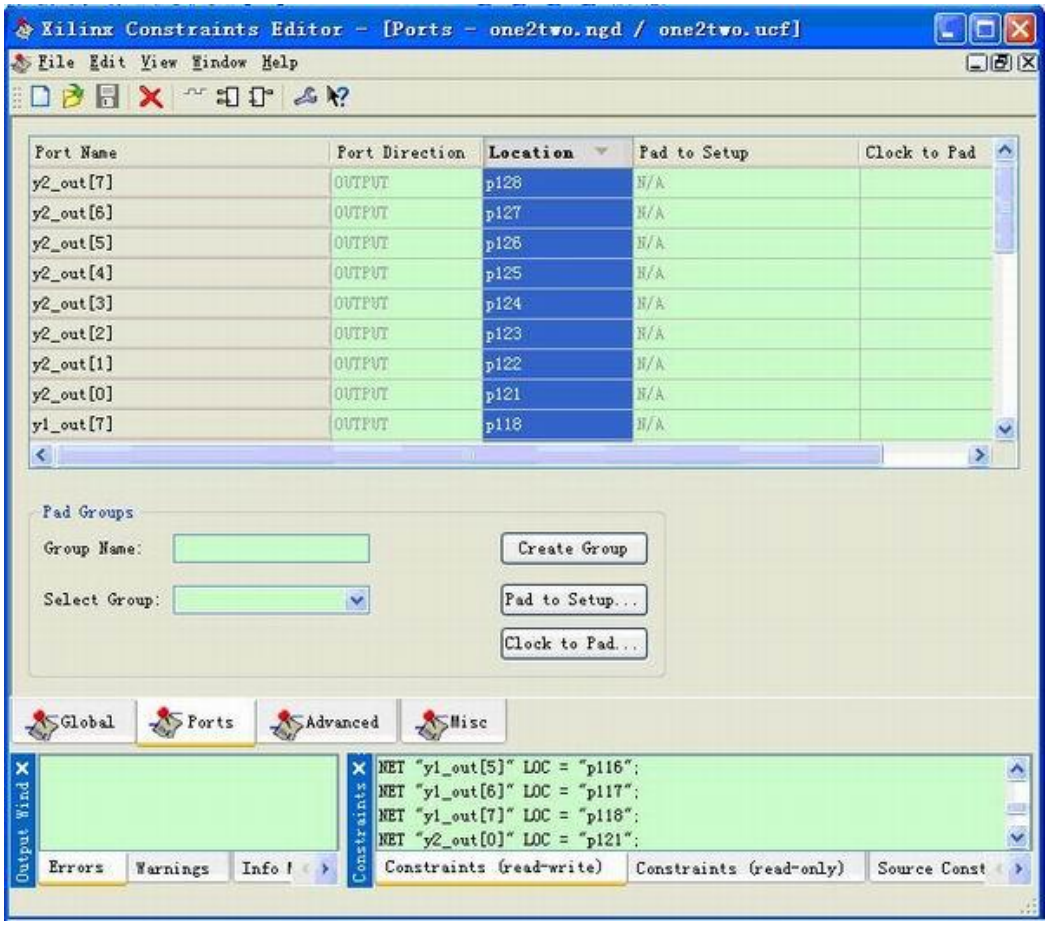


图 启动 Constrains Editor 引脚约束编辑

在“Ports”选项卡中可以看到，所有的端口都已经罗列出来了，如果要修改端口和 FPGA 管脚的对应关系，只需要在每个端口的“Location”列中填入管脚的编号即可。例如在 UCF 文件中描述管脚分配的语法为：

NET “端口名称” LOC = 引脚编号;

需要注意的是，UCF 文件是大小敏感的，端口名称必须和源代码中的名字一致，且端口名字不能和关键字一样。但是关键字 NET 是不区分大小写的。

3．编辑约束文件

在工程管理区中，将“Source for”设置为“Synthesis/Implementation”，然后双击过程管理区中“User Constrains”下的“Edit Constraints (Text)”就可以打开约束文件编辑器，如下图所示，就会新建当前工程的约束文件。

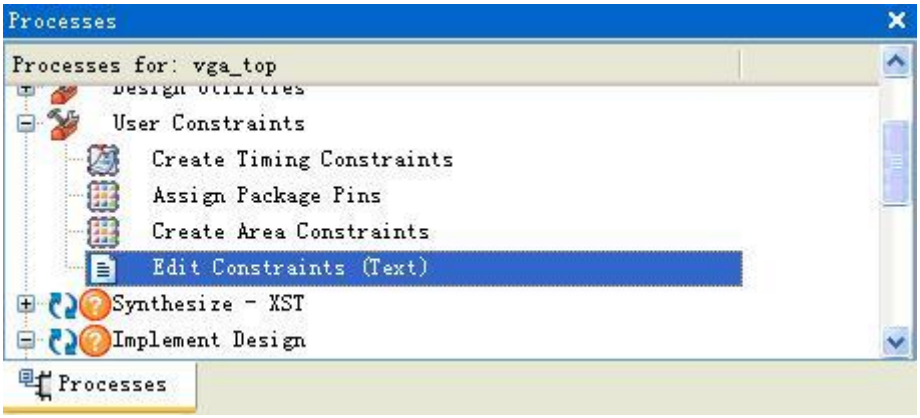


图 用户约束管理窗口

UCF 文件的语法说明

1．语法

UCF 文件的语法为：

{NET|INST|PIN} "signal_name" Attribute;

其中，“signal_name”是指所约束对象的名字，包含了对象所在层次的描述；“Attribute”为约束的具体描述；语句必须以分号“；”结束。可以用“#”或“/* */”添加注释。需要注意的是：UCF 文件是大小写敏感的，信号名必须和设计中保持大小写一致，但约束的关键字可以是大写、小写甚至大小写混合。例如：

NET "CLK" LOC = P30;

“CLK”就是所约束信号名，LOC = P30；是约束具体的含义，将 CLK 信号分配到 FPGA 的 P30 管脚上。

对于所有的约束文件，使用与约束关键字或设计环境保留字相同的信号名会产生错误信息，除非将其用" "括起来，因此在输入约束文件时，最好用" "将所有的信号名括起来。

2．通配符

在 UCF 文件中，通配符指的是“*”和“?”。“*”可以代表任何字符串以及空，“?”则代表一个字符。在编辑约束文件时，使用通配符可以快速选择一组信号，当然这些信号都要包含部分共有的字符串。例如：

NET "*CLK?" FAST;

将包含“CLK”字符并以一个字符结尾的所有信号，并提高了其速率。

在位置约束中，可以在行号和列号中使用通配符。例如：

INST "/CLK_logic/*" LOC = CLB_r*c7;

把 CLK_logic 层次中所有的实例放在第 7 列的 CLB 中。

3．定义设计层次

在 UCF 文件中，通过通配符*可以指定信号的设计层次。其语法规则为：

* 遍历所有层次

Level1/* 遍历 level1 及以下层次中的模块

Level1/*/ 遍历 level1 种的模块，但不遍历更低层的模块

例 4-5 根据图 4-75 所示的结构，使用通配符遍历表 4-3 所要求的各个模块。

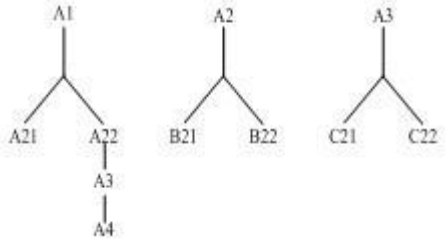


图 层次模块示意图

表 要求遍历的符号列表

要求遍历的符号	相应的约束语句
所有符号	INST * 或 INST/*
A1, B1, C1	INST/*/
A21, A22	INST A1/*/
A3	INST A1/*/*/
A3, A4	INST A1/*/*
A22, B22, C22	INST /*/*22/

管脚和区域约束语法

LOC 约束是 FPGA 设计中最基本的布局约束和综合约束，能够定义基本设计单元在 FPGA 芯片中的位置，可实现绝对定位、范围定位以及区域定位。此外， LOC 还能将一组基本单元约束在特定区域之中。LOC 语句既可以书写在约束文件中，也可以直接添加到设计文件中。换句话说，ISE 中的 FPGA 底层工具编辑器（FPGA Editor）、布局规划器（Floorplanner）和引脚和区域约束编辑器的主要功能都可以通过 LOC 语句完成。

- LOC 语句语法

INST "instance_name " LOC = location;

其中“location”可以是 FPGA 芯片中任一或多个合法位置。如果为多个定位，需要用逗号“,”隔开，如下所示：
LOC = location1,location2,...,locationx;

目前，还不支持将多个逻辑置于同一位置以及将多个逻辑至于多个位置上。需要说明的是，多位置约束并不是将设计定位到所有的位置上，而是在布局布线过程中，布局器任意挑选其中的一个作为最终的布局位置。

范围定位的语法为：
INST “instance_name” LOC=location:location [SOFT];

常用的 LOC 定位语句如表 4-4 所列。

表 常用的 LOC 定位语句

LOC 语句	说明	功能表述
INST "instance_name"LOC=P12;	单定位语句	将 I/O 管脚 P12 分配给实例信号
INST "instance_name"LOC=CLB_R3C5;	单定位语句	将逻辑置于坐标为行 3、列 5 的 CLB 中的任一个 SILCE
INST "instance_name" LOC=SLICE_X3Y2;	单定位语句	经逻辑置于 Slice xy 网格中的（3，2）位置上
INST "instance_name" LOC=TBUF_R1C2.*;	单定位语句	将逻辑至于 1 行、2 列位置的两个 TBUF 中
INST "instance_name" LOC=MULT18X18_X0Y6;	单定位语句	将乘法器逻辑置于乘法器 xy 网格中（0，6）位置上的 MULT18X18 乘法器中
INST "instance_name" LOC=clb_r4c5.s1,clb_r4c6.*;	多定位语句	将触发器置于 4 行 5 列 CLB 和 4 行 6 列的 CLB 中最右端的 Slice 中
INST "instance_name" LOC=SLICE_X2Y10, SLICE_X1Y10;	多定位语句	将逻辑置于 Slice xy 网格中（2，10）或（1，10）位置上的 Slice 中
INST "instance_name" LOC=SLICE_X3Y5:SLICE_X5Y20;	范围定位语句	将逻辑至于 4 行 4 列 CLB 左上角的任一 Silce 上
INST "instance_name" LOC=SLICE_X3Y5:SLICE_X5Y20;	范围定位语句	将逻辑至于 Slice xy 网格中，由（3，5）、（5，20）两点为对角线的矩形中的任一 Slice 中

使用 LOC 完成端口定义时，其语法如下：
NET "Top_Module_PORT" LOC = "Chip_Port";

其中，“Top_Module_PORT”为用户设计中顶层模块的信号端口，“Chip_Port”为 FPGA 芯片的管脚名。

LOC 语句中是存在优先级的，当同时指定 LOC 端口和其端口连线时，对其连线约束的优先级是最高的。例如，在图 4-76 中，LOC=11 的优先级高于 LOC=38。

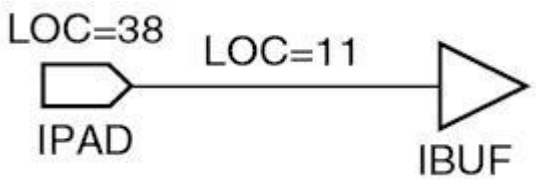


图 LOC 优先级示意图

2．LOC 属性说明

LOC 语句通过加载不同的属性可以约束管脚位置、CLB、Slice、TBUF、块 RAM、硬核乘法器、全局时钟、数字锁相环（DLL）以及 DCM 模块等资源，基本涵盖了 FPGA 芯片中所有类型的资源。由此可见，LOC 语句功能十分强大，表 4-5 列出了 LOC 的常用属性。

表 LOC 语句常用属性列表

约束类型	可用属性示例	属性含义
IO 管脚约束	P12	将信号置于由芯片引脚号定位的端口上
	A12	将信号置于由芯片引脚阵列号定位的端口上
	B, L, T, R	将信号定位到芯片特定边界中（从物理位置上划分的上、下、左、右4部分）的端口上。
	LB, RB, LT, RT, BR, TR, BL, TL	将信号定位到芯片特定边界上的一半（从物理位置上划分的上左 、上右 、下左、下右、左上、坐下、右上以及右下8部分）位置中的端口上。
	Bank0, Bank1, Bank2, Bank3, Bank4, Bank5,Bank6, Bank7	将信号置于特定管脚分组中的端口上。
CLBs	CLB_R4C3 (or .S0 or .S1)	指定确定位置的CLB来实现逻辑
	CLB_R6C8.S0 (or .S1)	指定确定位置的CLB中的Slice来实现逻辑
Slice	SLICE_X22Y3	直接通过Slice坐标来指定确定位置的Slice来实现逻辑
TBUF	TBUF_R6C7 (or .0 or .1)	指定确定位置的TBUF来实现逻辑
	TBUF_X6Y7	
块 RAM	RAMB4_R3C1	指定确定位置的块RAM来实现逻辑
	RAMB16_X2Y56	
硬核乘法器	MULT18X18_X55Y82	指定确定位置的硬核乘法器来实现逻辑
全局时钟	GCLKBUF0 (or 1, 2, or 3)	指定确定位置的全局时钟缓存器来实现逻辑
	GCLKPAD0 (or 1, 2, or 3)	指定确定位置的全局时钟端口来实现逻辑
DLL	DLL0P(or S) (or 1, 2, or 3)	使用确定位置的DLL来实现逻辑
DCM	DCM_X0Y0	使用确定位置的DCM模块来实现逻辑

Verilog HDL 代码描述对状态机综合的研究

2007-11-25 16:59

1 引言

Verilog HDL 作为当今国际主流的 **HDL 语言**,在芯片的前端设计中有着广泛的应用。它的语法丰富,成功地应用于设计的各个阶段：建模、仿真、验证和综合等。可综合是指综合工具能将 Verilog HDL 代码转换成标准的门级结构网表,因此代码的描述必须符合一定的规则。大部分数字系统都可以分为控制单元和数据单元两个部分,控制单元的主体是一个状态机,它接收外部信号以及数据单元产生的状态信息,产生控制信号,因而状态机性能的好坏对系统性能有很大的影响。

有许多可综合状态机的 Verilog 代码描述风格,不同代码描述风格经综合后得到电路的物理实现在速度和面积上有很大差别。优秀的代码描述应当易于修改、易于编写和理解,有助于仿真和调试,并能生成高效的综合结果。

2 有限状态机

有限状态机(Finite State Machine,FSM)在数字系统设计中应用十分广泛。根据状态机的输出是否与输入有关,可将状态机分为两大类：摩尔(Moore)型状态机和米莉(Mealy)型状态机。Moore 型状态机的输出仅与现态有关;Mealy 型状态机的输出不仅与现态有关,而且和输入也有关。图 1 是有限状态机的一般结构图,它主要包括三个部分,其中组合逻辑部分包括状态译码器和输出译码器,状态译码器确定状态机的下一个状态,输出译码器确定状态机的输出,状态寄存器属于时序逻辑部分,用来存储状态机的内部状态。

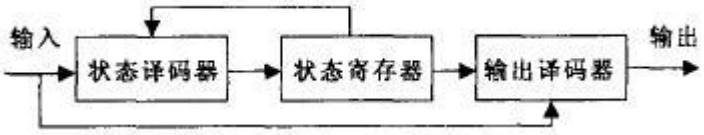


图 1 状态机的结构框图

2.1 好的状态机标准

好的状态机的标准很多,最重要的几个方面如下：

第一,状态机要安全,是指 FSM 不会进入死循环,特别是不会进入非预知的状态,而且由于某些扰动进入非设计状态,也能很快的恢复到正常的状态循环中来。这里面有两层含义。其一要求该 FSM 的综合实现结果无毛刺等异常扰动,其二要求 FSM 要完备,即使受到异常扰动进入非设计状态,也能很快恢复到正常状态。

第二,状态机的设计要满足设计的面积和速度的要求。

第三,状态机的设计要清晰易懂、易维护。

需要说明的是,以上各项标准,不是割裂的,它们有着直接紧密的内在联系。在芯片设计中,对综合结果评判的两个基本标准为：面积和速度。“面积”是指设计所占用的逻辑资源数量;“速度”指设计在芯片上稳定运行所能够达到的最高频率。两者是对立统一的矛盾体,要求一个设计同时具备设计面积最小,运行频率最高,这是不现实的。科学的设计目标应该是：在满足设计时序要求(包含对设计最高频率的要求)的前提下,占用最小的芯片面积,或者在所规定的面积下,使设计的时序余量更大,频率更高。另外,如果要求 FSM 安全,则很多时候需要使用“full case”的编码方式,即将状态转移变量的所有向量组合情况都在 FSM 中有相应的处理,这经常势必意味着要多花更多的设计资源,有时也会影响 FSM 的频率所以,上述的标准要综合考虑,根据设计的要求进行权衡。

2.2 状态机描述方法

状态机描述时关键是要描述清楚几个状态机的要素,即如何进行状态转移,每个状态的输出是什么,状态转移的条件等。具体描述时方法各种各样,最常见的有三种描述方式：

第一,整个状态机写到一个 always 模块里面,在该模块中既描述状态转移,又描述状态的输入和输出;

第二,用两个 always 模块来描述状态机,其中一个 always 模块采用同步时序描述状态转移;另一个模块采用组合逻辑判断状态转移条件,描述状态转移规律以及输出;

第三,在两个 always 模块描述方法基础上,使用三个 always 模块,一个 always 模块采用同步时序描述状态转移,一个采用组合逻辑判断状态转移条件,描述状态转移规律,另一个 always 模块描述状态的输出(可以用组合电路输出,也可以时序电路输出)。

一般而言,推荐的 FSM 描述方法是后两种。这是因为:FSM 和其他设计一样,最好使用同步时序方式设计,以提高设计的稳定性,消除毛刺。状态机实现后,一般来说,状态转移部分是同步时序电路而状态的转移条件的判断是组合逻辑。

第二种描述方法同第一种描述方法相比,将同步时序和组合逻辑分别放到不同的 always 模块中实现,这样做的好处不仅仅是便于阅读、理解、维护,更重要的是利于综合器优化代码,利于用户添加合适的时序约束条件,利于布局布线器实现设计。在第二种方式的描述中,描述当前状态的输出用组合逻辑实现,组合逻辑很容易产生毛刺,而且不利于约束,不利于综合器和布局布线器实现高性能的设计。第三种描述方式与第二种相比,关键在于根据状态转移规律,在上一状态根据输入条件判断出当前状态的输出,从而在不插入额外时钟节拍的前提下,实现了寄存器输出。

2.3 状态机的编码

二进制编码(Binary)、格雷码(Gray-code)编码使用最少的触发器,较多的组合逻辑,而独热码(One-hot)编码反之。独热码编码的最大优势在于状态比较时仅仅需要比较一个位,从而一定程度上简化了比较逻辑,减少了毛刺产生的概率。由于 **CPLD** 更多地提供组合逻辑资源,而 **FPGA** 更多地提供触发器资源,所以 CPLD 多使用二进制编码或格雷码,而 FPGA 多使用独热码编码。另一方面,对于小型设计使用二进制和格雷码编码更有效,而大型状态机使用独热码更高效。

3 实例说明

下面通过实例来说明 Verilog HDL 代码描述对状态机综合结果的影响。

设计一个序列检测器,用于检测串行的二进制序列,每当连续输入三个或三个以上的 1 时,序列检测器的输出为 1,其它情况下输出为 0。

假设初始的状态为 s0,输入一个 1 的状态记为 s1,连续输入二个 1 后的状态记为 s2,输入三个或以上 1 的状态记为 s3,不论现态是何种状态一旦输入 0 的话,就返回到初始状态。根据题意,可画出状态图如图 2 所示。

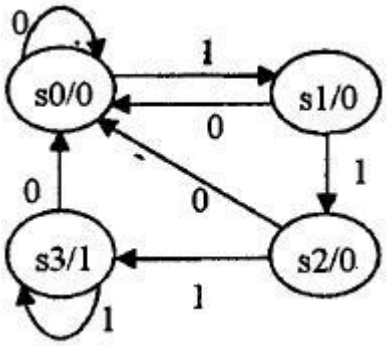
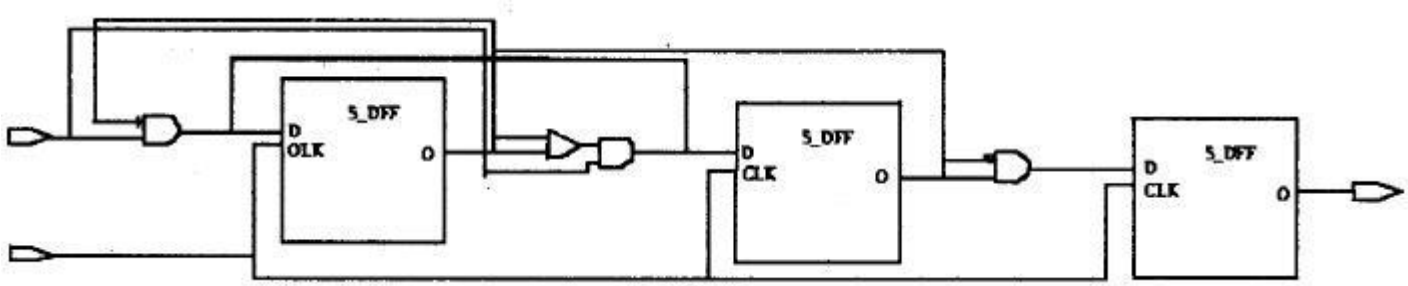


图 2 状态图

根据状态图以及前面状态机的介绍,可以采用一个 always 模块来描述,状态编码采用二进制编码,程序如下:

```
module fsm(clk,ina,out);
input clk,ina;
output out;
reg out;
parameter s0 = 3'bOO,s1 =3'b01,s2 =3'b10,s3=3'b11;
reg[0:1]state;
always @ (posedge clk)
begin
state<=s0;
out =0;
case(state)
s0:begin
state<=(ina)?s1:s0;out=0;
end
s1:begin
state<=(ina)?s2:s0;out=0;
end
s2:begin
state<=(ina)?s3:s0;out=0;
end
s3:begin
state<=(ina)?s3:s0;out=1;
end
endcase
end
endmodule
```

采用 Synplify Pro 工具在 **Altera** EPF10K10 系列器件上进行综合,其综合的结果如图 3 所示。



如果采用两个 always 来描述,程序的模块声明、端口定义和信号类型部分不变,只是改动逻辑功能描述部分,改动部分的程序如下:

```
alwys @ (posedge dk)
state fsm <=next_state;
always @ (state_fsm or ina)
begin
state<=s0;out =0;
case(state_fsm )
s0:begin
next_state=(ina)?s1:s0;out=0;
end
s1:begin
next state=(ina)?s2:s0;out=0；
end
s2:begin
next_state=(ina)?s3:s0;out=0;
end
s3:begin
next_state=(ina)?s3:s0;out=1;
end
endcase
end
endmodule
```

在相同的器件上其综合的结果如图 4 所示,比较图 3 与图 4 的综合结果,可以看出。两种综合结果都是采用了两个触发器来存储状态。其不同的地方是输出部分,采用一个 always 模块的输出结果是寄存器输出。采用两个 always 模块描述的是组合逻辑直接输出,这是因为代码中的输出赋值也放在了时钟的上升沿(always @ (posedge clk))。其综合的结果是寄存器,因此它比直接组合逻辑输出延迟一个时钟周期。

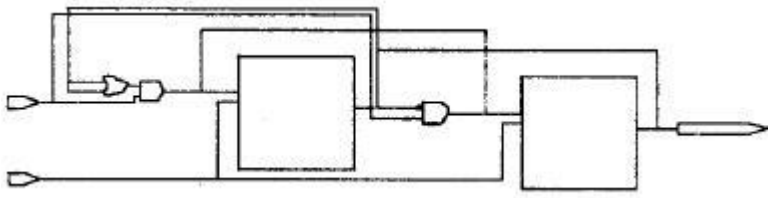


图 4

如果采用一位 hot 编码,仅改动参数设置的两行程序。采用一个 always 模块描述,改动部分的程序如下：

```
parameter s0 = 3'b0001,s1 =3'b0010,s2 =3'b0100,s3=3'b1000;
reg[0:3] state;
```

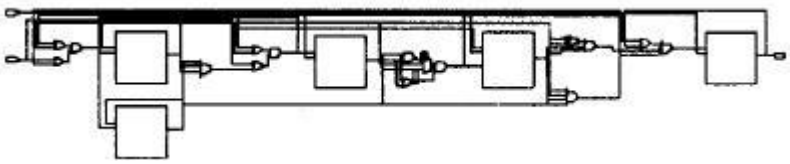


图 5

综合的结果如图 5 所示。将图 5 与图 3 相比,可以看出：

图 5 中状态寄存器采用了 4 个触发器来存储状态,而图 3 采用了两个触发器来存储状态,这是由于它们的状态编码的不同而得到的不同的综合结果,采用二进制编码综合得到的触发器要比采用独热码综合得到的触发器少。它们的共同之处都是采用了寄存器来输出的。

3 结束语

有多种可综合状态机的 Verilog HDL 代码描述风格。其综合的结果是不同的。其中广泛采用的是两个或三个 always 模块描述。组合逻辑输出型状态机不适合应用在高速复杂系统设计中,在高速系统中应当采用寄存器输出型状态机。寄存器类型信号不会产生毛刺,并且输出不含组合逻辑。会减少组合逻辑门延时。容易满足高速系统设计要求。总之,状态

机的设计是数字系统设计中的关键部分,设计时做到心中有电路。充分考虑其综合的结果,才能编写出高质量的综合代码。进而提高设计水平。

模块划分非常重要，除了关系到是否最大程度上发挥项目成员的协同设计能力，而且直接决定着设计的综合、实现时间。下面是一些模块划分的原则。

a.对每个同步设计的子模块的输出使用寄存器（registering）。也即用寄存器分割同步时序模块的原则。

使用寄存器的好处有：综合工具在编译综合时会将所分割的子模块中的组合电路和同步时序电路整体考虑。而且这种模块结构符合时序约束的习惯，便于使用时序约束熟悉进行约束。

b.将相关的逻辑或者可以复用的逻辑划分在同一模块内。

这样做的好处有，一方面将相关的逻辑和可以复用的逻辑划分在同一模块，可以最大程度的复用资源，减少设计消耗的面积。同时也更利于综合工具优化一个具体功能（操作）在时序上的关键路径。其原因是，综合工具只能同时考虑一部分逻辑，而所同时优化的逻辑单元就是模块，所以将相关功能划分在同一模块更有利于综合器的优化。

c.将不同优化目标的逻辑分开。

好的设计，在规划阶段，设计者就已经思考了设计的大概规模和关键路径，并对设计的优化目标有一个整体上的把握。对于时序紧张的部分，应该独立划分为一个模块，其优化目标为“speed”，这种划分方法便于设计者进行时序约束，也便于综合和实现工具进行优化。比如时序优化的利器 Amplify，使用模块进行区域优化更方便一些。另一类矛盾集中在面积的设计，也应该划分成独立的模块，这类模块的优化目标是“Area”，同样将他们规划到一起，更有利于区域布局与约束。这种根据优化目标进行优化的方法的最大好处是，对于某个模块综合器仅仅需要考虑一种优化目标和策略，从而比较容易达到较好的优化效果。相反的如果同时考虑两种优化目标，会使综合器陷入互相制约的困境。

d.将松约束的逻辑归到同一模块。

有些逻辑的时序非常宽松，不需要较高的时序约束，可以将这类逻辑归入同一模块，如多周期路径“multi-cycle”等。将这些模块归类，并指定松约束，则可以让综合器尽量的节省面积资源。

e.将 RAM/ROM/FIFO 等逻辑独立划分成模块。

这样做的好处是便于综合器将这类资源类推为器件的硬件原语，同时仿真时消耗的内存也会少些，便于提高仿真速度。（大多数仿真器对大面积的 RAM 都有独特的内存管理方式）

f.合适的模块规模。

规模大，利于“Resource Sharing”。但是对综合器同时处理的逻辑量太大，不利于多模块和增量编译模式。

关于约束，时序分析的问题汇总

很多人发贴，来信询问关于约束、时序分析的问题，比如：如何设置 setup，hold 时间？如何使用全局时钟和第二全局时钟（长线资源）？如何进行分组约束？如何约束某部分组合逻辑？如何通过约束保证异步时钟域之间的数据交换可靠？如何使用 I/O 逻辑单元内部的寄存器资源？如何进行物理区域约束，完成物理综合和物理实现？等等。。。

为了解决大家的疑难，我们将逐一讨论这些问题。

今天先讨论一下约束的作用？

有些人不知道何时该添加约束，何时不需要添加？有些人认为低速设计不需要时序约束？关于这些问题，希望下面关于约束作用的论述能够有所帮助！

附加约束的基本作用有 3：

(1)提高设计的工作频率

对很多数字电路设计来说，提高工作频率非常重要，因为高工作频率意味着高处理能力。通过附加约束可以控制逻辑的综合、映射、布局和布线，以减小逻辑和布线延时，从而提高工作频率。

(2)获得正确的时序分析报告

几乎所有的 FPGA 设计平台都包含静态时序分析工具，利用这类工具可以获得映射或布局布线后的时序分析报告，从而对设计的性能做出评估。静态时序分析工具以约束作为判断时序是否满足设计要求的标准，因此要求设计者正确输

入约束，以便静态时序分析工具输出正确的时序分析报告。
(3)指定 FPGA/CPLD 引脚位置与电气标准
FPGA/CPLD 的可编程特性使电路板设计加工和 FPGA/CPLD 设计可以同时进行，而不必等 FPGA/CPLD 引脚位置完全确定，从而节省了系统开发时间。这样，电路板加工完成后，设计者要根据电路板的走线对 FPGA/CPLD 加上引脚位置约束，使 FPGA/CPLD 与电路板正确连接。另外通过约束还可以指定 IO 引脚所支持的接口标准和其他电气特性。为了满足日新月异的通信发展，Xilinx 新型 FPGA/CPLD 可以通过 IO 引脚约束设置支持诸如 AGP、BLVDS、CTT、GTL、GTLP、HSTL、LDT、LVCMOS、LVDCI、LVDS、LVPECL、LVDSEXT、LVTTL、PCI、PCIX、SSTL、ULVDS 等丰富的 IO 接口标准。

另外通过区域约束还能在 FPGA 上规划各个模块的实现区域，通过物理布局布线约束，完成模块化设计等。

贴 2：时序约束的概念和基本策略！
时序约束主要包括周期约束（FFS 到 FFS，即触发器到触发器）和偏移约束（IPAD 到 FFS、FFS 到 OPAD）以及静态路径约束（IPAD 到 OPAD）等 3 种。通过附加约束条件可以使综合布线工具调整映射和布局布线过程，使设计达到时序要求。例如用 OFFSET_IN_BEFORE 约束可以告诉综合布线工具输入信号在时钟之前什么时候准备好，综合布线工具就可以根据这个约束调整与 IPAD 相连的 Logic Circuitry 的综合实现过程，使结果满足 FFS 的建立时间要求。
附加时序约束的一般策略是先附加全局约束，然后对快速和慢速例外路径附加专门约束。附加全局约束时，首先定义设计的所有时钟，对各时钟域内的同步元件进行分组，对分组附加周期约束，然后对 FPGA/CPLD 输入输出 PAD 附加偏移约束、对全组合逻辑的 PAD TO PAD 路径附加约束。附加专门约束时，首先约束分组之间的路径，然后约束快、慢速例外路径和多周期路径，以及其他特殊路径。

贴 3：周期（PERIOD）的含义
周期的含义是时序中最简单也是最重要的含义，其它很多时序概念会因为**软件**商不同略有差异，而周期的概念确是最通用的，周期的概念是 FPGA/ASIC 时序定义的基础概念。后面要讲到的其它时序约束都是建立在周期约束的基础上的，很多其它时序公式，可以用周期公式推导。
周期约束是一个基本时序和综合约束，它附加在时钟网线上，时序分析工具根据 PERIOD 约束检查时钟域内所有同步元件的时序是否满足要求。PERIOD 约束会自动处理寄存器时钟端的反相问题，如果相邻同步元件时钟相位相反，那么它们之间的延迟将被默认限制为 PERIOD 约束值的一半。
如下图所示，时钟的最小周期为：
TCLK = TCKO +TLOGIC +TNET +TSETUP －TCLK_SKEW
TCLK_SKEW =TCD2 －TCD1
其中 TCKO 为时钟输出时间，TLOGIC 为同步元件之间的组合逻辑延迟，TNET 为网线延迟，TSETUP 为同步元件的建立时间，TCLK_SKEW 为时钟信号延迟的差别。

这个帖子打算先澄清一些时序约束的基本概念，然后将在综合工具（Synplify Pro 为例），设计平台（ISE5.x 和 **Quartus** 2.2 为例）的具体约束方法和技巧，然后将如何利用时序分析工具分析关键路径。如果没有意外，应该 30 多个帖子吧。仿真时序本来是 Deve 的老本行，随时需要 Deve 加入一起把这个帖子办好。欢迎大家畅谈观点，本站的版主，冲锋啊，嘻嘻。

贴 4：数据和时钟之间的约束：OFFSET 和 SETUP、HOLD 时间。
为了确保芯片数据采样可靠和下级芯片之间正确的交换数据，需要约束外部时钟和数据输入输出引脚之间的时序关系（或者内部时钟和外部输入/输出数据之间的关系，这仅仅是从采用了不同的参照系罢了）。约束的内容为告诉综合器、布线器输入数据到达的时刻，或者输出数据稳定的时刻，从而保证与下一级电路的时序关系。
这种时序约束在 Xilinx 中用 Setup to Clock（edge），Clock（edge） to hold 等表示。在 **Altera** 里常用 tsu (Input Setup Times)、th (Input Hold Times)、tco (Clock to Out Delays)来表示。很多其它时序工具直接用 setup 和 hold 表示。其实他们所要描述的是同一个问题，仅仅是时间节点的定义上略有不同。下面依次介绍。

贴 5：关于输入到达时间，这一贴估计问题比较多，看起来也比较累，但是没有办法，这些都是时序的基本概念啊。搞不清楚，永远痛苦，长痛不如短痛了，呵呵。

Xilinx 的"输入到达时间的计算"时序描述如图所示：

定义的含义是输入数据在有效时钟沿之后的 TARRIVAL 时刻到达。则，
TARRIVAL=TCKO+TOUTPUT+TLOGIC　　公式 1
根据”贴 3“介绍的周期（Period）公式，我们可以得到：
Tcko+Toutput+Tlogic+Tinput+Tsetup-Tclk_skew=Tclk; 公式 2
将公式 1 代入公式 2：
Tarrival+Tinput+Tsetup-Tclk_skew=Tclk, 而 Tclk_skew 满足时序关系后为负，所以
TARRIVAL +TINPUT+TSETUP <TCLK 公式 3，
这就是 Tarrival 应该满足的时序关系。其中 TINPUT 为输入端的组合逻辑、网线和 PAD 的延迟之和，TSETUP 为输入同步元件的建立时间。

贴 6 数据延时和数据到达时间的关系：
TDELAY 为要求的芯片内部输入延迟，其最大值 TDELAY_MAX 与输入数据到达时间 TARRIVAL 的关系如图 2 所示。也就是说：
TDELAY_MAX+TARRIVAL=TPERIOD 公式 4
所以：
TDELAY<TDELAY_MAX=TPERIOD－TARRIVAL
帖 7 要求输出的稳定时间
从下一级输入端的延迟可以计算出当前设计输出的数据必须在何时稳定下来，根据这个数据对设计输出端的逻辑布线进行约束，以满足下一级的建立时间要求，保证下一级采样的数据是稳定的。
计算要求的输出稳定时间如图所示。
公式的推导如下：
定义：TSTABLE = TLOGIC +TINPUT +TSETUP
从前面帖子介绍的周期（Period）公式，可以得到(其中 TCLK_SKEW＝TCLK1－TCLK2)：
TCLK＝TCKO＋TOUTPUT+TLOGIC+TINPUT+TSETUP+TCLK_SKEW
将 TSTABLE 的定义代入到周期公式，可以得到：
TCLK=TCKO+TOUTPUT+TSTABLE+TCLK_SKEW
所以，
TCKO +TOUTPUT+TSTABLE<TCLK
这个公式就是 TSTABLE 必须要满足的基本时序关系，即本级的输出应该保持怎么样的稳定状态，才能保证下级芯片的采样稳定。有时我们也称这个约束关系是输出数据的保持时间的时序约束关系。只要满足上述关系，当前芯片输出端的数据比时钟上升沿提早 TSTABLE 时间稳定下来，下一级就可以正确地采样数据。
其中 TOUTPUT 为设计中连接同步元件输出端的组合逻辑、网线和 PAD 的延迟之和，TCKO 为同步元件时钟输出时间。

/**/
这里的概念介绍比较繁复，但是如果想掌握数据与时钟关系的基本约束，就必须搞清楚这些概念，下一帖介绍这些概念的具体应用，实施上述约束的方法和具体命令。

转贴 lipple 的问题：
请问斑竹上面几贴那些延时属于 setup，哪些属于 hold 啊
周期=Tsetup+Tlogic+Thold 这个公式对比斑竹的公式，区别在于是不是划分的不够细啊？
westor 的答复：
基本是哪个意思。这些公式描述的对象是意义的，只是每个变量的定义略有区别罢了，换句话说，变量定义的节点不

同。
这个公式是 altera 等采用的描述方法，一些工具为了便于理解用
周期=Tsetup+Tlogic+Thold 约束时序。
和我前面介绍的公式：
TCLK = TCKO +TLOGIC +TNET +TSETUP －TCLK_SKEW
相比，他把到寄存器前的所有组合逻辑 logic 和线延时都归在 Tsetup 里面了，而且上面公式忽略了 Tclk_skew。

帖 8 实施上述约束的方法和命令。
实施上述约束的基本方法是，根据已知时序信息，推算需要约束的时间值，实施约约束。具体的说是这样的，首先对于一般设计，首先掌握的是 TCLK,这个对于设计者来说是个已知量。前面介绍公式和图中的 TCKO 和 TSETUP（注：有的工具软件对 TCKO 和 TSETUP 的定义与前面图形不同，还包含了到达同步器件的一段 logic 的时延）是器件内部固有的一个时间量，一般我们选取典型值，对于 FPGA，这个量值比较小，一般不大于 1~2ns。比较难以确定的是 TINPUT 和 TOUTPUT 两个时间量。
约束输入时间偏移，需要知道 TINPUT，TINPUT 为输入端的组合逻辑、网线和 PAD 的延迟之和（详细定义见帖 5），PAD 的延时也根据器件型号也有典型值可选，但是到达输入端的组合逻辑电路和网线的延时就比较难以确定了，只能通过静态时序分析工具分析，或者通过底层布局布线工具量取，有很大的经验和试探的成分在里面。
约束输出时间偏移，需要知道 TOUTPUT，TOUTPUT 为设计中连接同步元件输出端的组合逻辑、网线和 PAD 的延迟之和（见帖 7），仍然是到达输出端的组合逻辑电路和网线的延时就比较难以确定，需要通过静态时序分析工具分析，或者通过底层布局布线工具量取，有很大的经验和试探的成分在里面。
约束的具体命令根据约束工具不同而异，首先说使用 Xilinx 器件的情况下，实施上述约束的命令和方法。Xilinx 把上述约束统称为：OFFSET 约束（偏移约束），一共有 4 个相关约束属性：OFFSET_IN_BEFORE、OFFSET_IN_AFTER、OFFSET_OUT_BEFORE 和 OFFSET_OUT_AFTER。
其中前两个属性叫做输入偏移（OFFSET_IN）约束，基本功能相似，仅仅是约束取的参考对象不同而已。后两个属性叫做输出偏移（OFFSET_OUT）约束，基本功能相似，也是约束取的参考对象不同而已。
为了便于理解，举例说明。
输入偏移约束例：时钟周期为 20ns，前级寄存器的 TCKO 选则 1ns，前级输出逻辑延时 TOUTPUT 为 3ns，中间逻辑 TLOGIC 的延时为 10ns，那么 TARRIVAL=14ns，于是可以在数据输入引脚附加
NET DATA_IN FFET=IN 14ns AFTER CLK
约束，也可以使用 OFFSET_IN_BEFORE 对芯片内部的输入逻辑进行约束，其语法如下：
NET DATA_IN FFET=IN TDELAY BEFORE CLK
其中 TDELAY 为要求的芯片内部输入延迟，其最大值与输入数据到达时间 TARRIVAL 的关系如帖 6 所述:TDELAY_MAX + TARRIVAL = TPERIOD,所以
TDELAY < TPERIOD - TARRIVAL = 20 - 14 =6 ns.
输出偏移约束例：设时钟周期为 20ns，后级输入逻辑延时 TINPUT 为 4ns、建立时间 TSETUP 为 1ns，中间逻辑 TLOGIC 的延时为 10ns，那么 TSTABLE=15ns，于是可以在数据输入引脚附加 NET DATA_OUT FFET=OUT 15ns BEFORE CLK
约束，也可以直接对芯片内部的输出逻辑直接进行约束，
NET DATA_OUT FFET=OUT TOUTPUT_DELAY AFTER CLK，
其中 TOUTPUT_DELAY 为要求的芯片内部输出延迟，其最大值与要求的输出数据稳定时间 TSTABLE 的关系为：
TOUTPUT_DELAY_MAX+TSTABLE= TPERIOD.
TOUT_DELAY< TPERIOD - TSTABLE = 20 - 15 = 5ns
/**/
这些概念和推导有些枯燥和乏味，但是如果要掌握好数据与时钟之间的约束，就要耐心看下去，明天介绍一下 Altera 的相关约束方法。

帖 9 Altera 对应的时序概念
这两天太忙了，帖子上的有些慢，请朋友们原谅，我会尽量按照计划写完这个主题的。
前面 8 个帖子介绍了一些时序概念，有的是 FPGA/ASIC 设计的一般性时序概念，有的为了方便叙述，主要介绍了 Xilinx

对应的这些时序概念，和具体的约束熟悉。下面几个帖子主要介绍 Altera 对应的这些时序概念和约束方法。
前面首先介绍的第一个时序概念是周期，Period，这个概念是 FPGA/ASIC 通用的一个概念，各方的定义相当统一，至多是描述方式不同罢了，所有的 FPGA 设计都首先要进行周期约束，这样做的好处除了在综合与布局布线时给出规定目标外，还能让时序分析工具考察整个设计的 Fmax 等。
Altera 的周期定义如图所示，公式描述如下：
Clock Period = Clk-to-out + Data Delay + Setup Time - Clk Skew
即，
Tclk = Tco + B + Tsu -(E-C)
Fmax = 1/Tclk
对比一下前面的介绍，只要理解了 B 包含了两级寄存器之间的所有 logic 和 net 的延时就会发现与前面公式完全一致。
一个设计的 Fmax 在时序报告，或者在图形界面观察。以 Quartus2 为例，在图形界面的观察方法是，编译实现完成后，展开 Compilation Report 下面的 Timing Analyses，单击 Fmax（not include delays to / from pins）即可。在详细报告窗口可以观察到影响周期恶化的 10 条最差时序路径，根据这些信息可以找出关键路径，进行时序分析。
关于时序分析和关键路径改进等内容在后面的帖子会有专门的讨论，暂时不做进一步介绍。
贴 10
Clock Setup Time (tsu)
要想正确采样数据，就必须使数据和使能信号在有效时钟沿到达前就准备好，所谓时钟建立时间就是指时钟到达前，数据和使能已经准备好的最小时间间隔。如图 1 所示：
注：这里定义 Setup 时间是站在同步时序整个路径上的，需要区别的是另一个概念 Micro tsu。Micro tsu 指的是一个触发器内部的建立时间，它是触发器的固有属性，一般典型值小于 1~2ns。在 Xilinx 等的时序概念中，称 Altera 的 Micro tsu 为 setup 时间，用 Tsetup 表示，请大家区分一下。
回到 Altera 的时序概念，Altera 的 tsu 定义如下：
tsu = Data Delay – Clock Delay + Micro tsu

贴 11
Clock Hold Time tH
时钟保持时间是只能保证有效时钟沿正确采用的数据和使能信号的最小稳定时间。其定义如图 2 所示。定义的公式为：
tH= Clock Delay – Data Delay + Micro tH
注：其中 Micro tH 是指寄存器内部的固有保持时间，同样是寄存器的一个固有参数，典型值小于 1～2ns。

贴 12
Clock-to-Output Delay（tco）
这个时间指的是当时钟有效沿变化后，将数据推倒同步时序路径的输出端的最小时间间隔。如图 3 所示。
tco ＝ Clock Delay + Micro tco + Data Delay
注：其中 Micor tco 也是一个寄存器的固有属性，指的是寄存器相应时钟有效沿，将数据送到输出端口的内部时间参数。它与 Xilinx 的时序定义中，有一个概念叫 Tcko 是同一个概念。

Pin to Pin Delay （tpd）
tpd 指输入管脚通过纯组合逻辑到达输出管脚这段路径的延时，特别需要说明的是，要求输入到输出之间只有组合逻辑，才是 tpd 延时。
7.Slack
Slack 是表示设计是否满足时序的一个称谓，正的 slack 表示满足时序（时序的余量），负的 slack 表示不满足时序（时序的欠缺量）。slack 的定义和图形如图 4 所示。
Slack = Required clock period – Actual clock period
Slack = Slack clock period – (Micro tCO+ Data Delay + Micro tSU)
8.Clock Skew
Clock Skew 指一个同源时钟到达两个不同的寄存器时钟端的时间偏移。如图 5 所示。