

## 第七章 Verilog HDL 语法

Verilog HDL (Hardware Description Language) 是在用途最广泛的 C 语言的基础上发展起来的一种硬件描述语言, 具有灵活性高、易学易用等特点。Verilog HDL 可以在较短的时间内学习和掌握, 目前已经在 FPGA 开发/IC 设计领域占据绝对的领导地位。

本章包括以下几个部分:

- 7.1 Verilog 概述
- 7.2 Verilog 基础知识
- 7.3 Verilog 程序框架
- 7.4 Verilog 高级知识点
- 7.5 Verilog 编程规范

## 7.1 Verilog 概述

本节主要描述了 Verilog HDL (以下简称 Verilog) 简介、Verilog 和 VHDL 以及和 C 语言的区别。

### 7.1.1 Verilog 简介

Verilog 是一种硬件描述语言, 以文本形式来描述数字系统硬件的结构和行为的语言, 用它可以表示逻辑电路图、逻辑表达式, 还可以表示数字逻辑系统所完成的逻辑功能。

数字电路设计者利用这种语言, 可以从顶层到底层逐层描述自己的设计思想, 用一系列分层次的模块来表示极其复杂的数字系统。然后利用电子设计自动化 (EDA) 工具, 逐层进行仿真验证, 再把其中需要变为实际电路的模块组合, 经过自动综合工具转换到门级电路网表。接下来, 再用专用集成电路 ASIC 或 FPGA 自动布局布线工具, 把网表转换为要实现的具体电路结构。

Verilog 语言最初是于 1983 年由 Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言。由于他们的模拟、仿真器产品的广泛使用, Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者所接受。在一次努力增加语言普及性的活动中, Verilog HDL 语言于 1990 年被推向公众领域。Verilog 语言于 1995 年成为 IEEE 标准, 称为 IEEE Std1364-1995, 也就是通常所说的 Verilog-95。

设计人员在使用 Verilog-95 的过程中发现了一些可改进之处。为了解决用户在使用此版本 Verilog 过程中反映的问题, Verilog 进行了修正和扩展, 这个扩展后的版本后来成为了电气电子工程师学会 Std1364-2001 标准, 即通常所说的 Verilog-2001。Verilog-2001 是对 Verilog-95 的一个重大改进版本, 它具备一些新的实用功能, 例如敏感列表、多维数组、生成语句块、命名端口连接等。目前, Verilog-2001 是 Verilog 的最主流版本, 被大多数商业电子设计自动化软件支持。

### 7.1.2 为什么需要 Verilog

在 FPGA 设计里面, 我们有多种设计方式, 如原理图设计方式、编写描述语言 (代码) 等方式。一开始很多工程师对原理图设计方式很钟爱, 这种输入方式能够很直观的看到电路结构并快速理解, 但是随着电路设计规模的不断增加, 逻辑电路设计也越来越复杂, 这种设计方式已经越来越不满足实际的项目需求了。这个时候 Verilog 语言就取而代之了, 目前 Verilog 已经在 FPGA 开发/IC 设计领域占据绝对的领导地位。

### 7.1.3 Verilog 和 VHDL 区别

这两种语言都是用于数字电路系统设计的硬件描述语言, 而且都已经是 IEEE 的标准。VHDL 1987 年成为标准, 而 Verilog 是 1995 年才成为标准的。这是因为 VHDL 是美国军方组织开发的, 而 Verilog 是由一个公司的私有财产转化而来。为什么 Verilog 能成为 IEEE 标准呢? 它一定有其独特的优越性才行, 所以说 Verilog 有更强大的生命力。

这两者有其共同的特点:

1. 能形式化地抽象表示电路的行为和结构;
2. 支持逻辑设计中层次与范围地描述;
3. 可借用高级语言地精巧结构来简化电路行为和结构;
4. 支持电路描述由高层到低层的综合转换;
5. 硬件描述和实现工艺无关。

但是两者也各有特点。Verilog 推出已经有 20 年了, 拥有广泛的设计群体, 成熟的资源, 且 Verilog 容易掌握, 只要有 C 语言的编程基础, 通过比较短的时间, 经过一些实际的操作, 可以在 1 个月左右掌握这种语言。而 VHDL 设计相对要难一点, 这个是因为 VHDL 不是很直观, 一般认为至少要半年以上的专业培

训才能掌握。

近 10 年来, EDA 界一直在对数字逻辑设计中究竟用哪一种硬件描述语言争论不休, 目前在美国, 高层次数字系统设计领域中, 应用 Verilog 和 VHDL 的比率是 80% 和 20%; 日本与中国台湾和美国差不多; 而在欧洲 VHDL 发展的比较好; 在中国很多集成电路设计公司都采用 Verilog。我们推荐大家学习 Verilog, 本教程全部的例程都是使用 Verilog 开发的。

## 7.1.4 Verilog 和 C 的区别

Verilog 是硬件描述语言, 在编译下载到 FPGA 之后, 会生成电路, 所以 Verilog 全部是并行处理与运行的; C 语言是软件语言, 编译下载到单片机/CPU 之后, 还是软件指令, 而不会根据你的代码生成相应的硬件电路, 而单片机/CPU 处理软件指令需要取址、译码、执行, 是串行执行的。

Verilog 和 C 的区别也是 FPGA 和单片机/CPU 的区别, 由于 FPGA 全部并行处理, 所以处理速度非常快, 这个是 FPGA 的最大优势, 这一点是单片机/CPU 替代不了的。

## 7.2 Verilog 基础知识

本节主要讲解了 Verilog 的基础知识, 包括 5 个小节, 下面我们分别给大家介绍这 5 个小节的内容。

### 7.2.1 Verilog 的逻辑值

我们先看下逻辑电路中有四种值, 即四种状态:

逻辑 0: 表示低电平, 也就是对应我们电路的 GND;

逻辑 1: 表示高电平, 也就是对应我们电路的 VCC;

逻辑 X: 表示未知, 有可能是高电平, 也有可能是低电平;

逻辑 Z: 表示高阻态, 外部没有激励信号是一个悬空状态。

如下图所示:

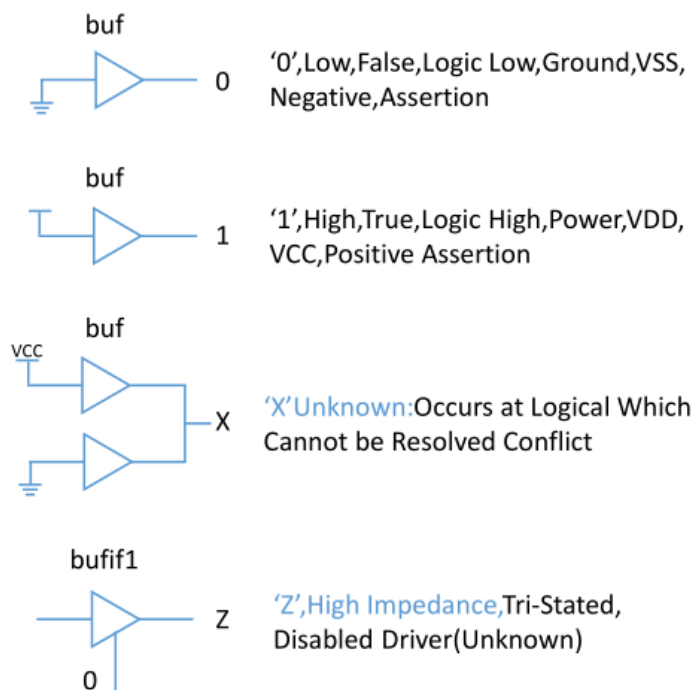


图 7.2.1 Verilog 逻辑值

### 7.2.2 Verilog 的标识符

#### 定义

标识符(identifier)用于定义模块名、端口名和信号名等。Verilog 的标识符可以是任意一组字母、数字、\$和\_(下划线)符号的组合,但标识符的第一个字符必须是字母或者下划线。另外,标识符是区分大小写的。以下是标识符的几个例子:

Count

COUNT //与 Count 不同。

R56\_68

FIVES

虽然标识符写法很多,但是要简洁、清晰、易懂,推荐写法如下:

count

fifo\_wr

不建议大小写混合使用,普通内部信号建议全部小写,参数定义建议大写,另外信号命名最好体现信号的含义。

#### 规范建议

以下是一些书写规范的要求:

- 1、用有意义的有效的名字如 sum、cpu\_addr 等。
- 2、用下划线区分词语组合,如 cpu\_addr。
- 3、采用一些前缀或后缀,比如:时钟采用 clk 前缀: clk\_50m, clk\_cpu; 低电平采用 \_n 后缀: enable\_n;
- 4、统一缩写,如全局复位信号 rst。
- 5、同一信号在不同层次保持一致性,如同一时钟信号必须在各模块保持一致。
- 6、自定义的标识符不能与保留字(关键词)同名。
- 7、参数统一采用大写,如定义参数使用 SIZE。

### 7.2.3 Verilog 的数字进制格式

Verilog 数字进制格式包括二进制、八进制、十进制和十六进制,一般常用的为二进制、十进制和十六进制。

二进制表示如下: 4'b0101 表示 4 位二进制数字 0101;

十进制表示如下: 4'd2 表示 4 位十进制数字 2 (二进制 0010);

十六进制表示如下: 4'ha 表示 4 位十六进制数字 a (二进制 1010),十六进制的计数方式为 0, 1, 2...9, a, b, c, d, e, f, 最大计数为 f (f: 十进制表示为 15)。

当代码中没有指定数字的位宽与进制时,默认为 32 位的十进制,比如 100,实际上表示的值为 32'd100。

### 7.2.4 Verilog 的数据类型

在 Verilog 语法中,主要有三大类数据类型,即寄存器类型、线网类型和参数类型。从名称中,我们可以看出,真正在数字电路中起作用的数据类型应该是寄存器类型和线网类型。

#### 1) 寄存器类型

寄存器类型表示一个抽象的数据存储单元,它只能在 always 语句和 initial 语句中被赋值,并且它的值从一个赋值到另一个赋值过程中被保存下来。如果该过程语句描述的是时序逻辑,即 always 语句带有时钟信号,则该寄存器变量对应为寄存器;如果该过程语句描述的是组合逻辑,即 always 语句不带有时钟信号,

则该寄存器变量对应为硬件连线; 寄存器类型的缺省值是 x (未知状态)。

寄存器数据类型有很多种, 如 reg、integer、real 等, 其中最常用的就是 reg 类型, 它的使用方法如下:

```
//reg define
reg [31:0] delay_cnt;    //延时计数器
reg      key_flag ;     //按键标志
```

## 2) 线网类型

线网表示 Verilog 结构化元件间的物理连线。它的值由驱动元件的值决定, 例如连续赋值或门的输出。如果没有驱动元件连接到线网, 线网的缺省值为 z (高阻态)。线网类型同寄存器类型一样也是有很多种, 如 tri 和 wire 等, 其中最常用的就是 wire 类型, 它的使用方法如下:

```
//wire define
wire      data_en;       //数据使能信号
wire [7:0] data ;       //数据
```

## 3) 参数类型

我们再来看下参数类型, 参数其实就是一个常量, 常被用于定义状态机的状态、数据位宽和延迟大小等, 由于它可以在编译时修改参数的值, 因此它又常被用于一些参数可调的模块中, 使用户在实例化模块时, 可以根据需要配置参数。在定义参数时, 我们可以一次定义多个参数, 参数与参数之间需要用逗号隔开。这里我们需要注意的是参数的定义是局部的, 只在当前模块中有效。它的使用方法如下:

```
//parameter define
parameter DATA_WIDTH = 8; //数据位宽为8位
```

## 7.2.5 Verilog 的运算符

大家看完了 Verilog 的数据类型, 我们再来介绍下 Verilog 的运算符。Verilog 中的运算符按照功能可以分为下述类型: 1、算术运算符、 2、关系运算符、3、逻辑运算符、 4、条件运算符、 5、位运算符、 6、移位运算符、 7、拼接运算符。下面我们分别对这些运算符进行介绍。

### 1) 算术运算符

算术运算符, 简单来说, 就是数学运算里面的加减乘除, 数字逻辑处理有时候也需要进行数字运算, 所以需要算术运算符。常用的算术运算符主要包括加减乘除和模除 (模除运算也叫取余运算) 如下表所示:

表 7.2.1 算术运算符

符号	使用方法	说明
+	a + b	a 加上 b
-	a - b	a 减去 b
*	a * b	a 乘以 b
/	a / b	a 除以 b
%	a % b	a 模除 b

大家要注意下, Verilog 实现乘除比较浪费组合逻辑资源, 尤其是除法。一般 2 的指数次幂的乘除法使用移位运算来完成运算, 详情可以看移位运算符章节。非 2 的指数次幂的乘除法一般是调用现成的 IP, QUARTUS/ISE 等工具软件会有提供, 不过这些工具软件提供的 IP 也是由最底层的组合逻辑(与或非门等)搭建而成的。

## 2) 关系运算符

关系运算符主要是用来做一些条件判断用的, 在进行关系运算符时, 如果声明的关系是假的, 则返回值是 0, 如果声明的关系是真的, 则返回值是 1; 所有的关系运算符有着相同的优先级别, 关系运算符的优先级别低于算术运算符的优先级别如下表所示。

表 7.2.2 关系运算符

符号	使用方法	说明
>	$a > b$	a 大于 b
<	$a < b$	a 小于 b
>=	$a \geq b$	a 大于等于 b
<=	$a \leq b$	a 小于等于 b
==	$a == b$	a 等于 b
!=	$a != b$	a 不等于 b

## 3) 逻辑运算符

逻辑运算符是连接多个关系表达式用的, 可实现更加复杂的判断, 一般不单独使用, 都需要配合具体语句来实现完整的意思, 如下表所示。

表 7.2.3 逻辑运算符

符号	使用方法	说明
!	!a	a 的非, 如果 a 为 0, 那么 a 的非是 1。
&&	$a \&\& b$	a 与上 b, 如果 a 和 b 都为 1, $a\&\&b$ 结果才为 1, 表示真。
	$a \ \  b$	a 或上 b, 如果 a 或者 b 有一个为 1, $a\ \ b$ 结果为 1, 表示真。

## 4) 条件运算符

条件操作符一般来构建从两个输入中选择一个作为输出的条件选择结构, 功能等同于 always 中的 if-else 语句, 如下表所示。

表 7.2.4 条件运算符

符号	使用方法	说明
? :	$a ? b : c$	如果 a 为真, 就选择 b, 否则选择 c

## 5) 位运算符

位运算符是一类最基本的运算符, 可以认为它们直接对应数字逻辑中的与、或、非门等逻辑门。常用的位运算符如下表所示。

表 7.2.5 位运算符

符号	使用方法	说明
~	~a	将 a 的每个位进行取反
&	$a \& b$	将 a 的每个位与 b 相同的位进行相与
	$a \ \  b$	将 a 的每个位与 b 相同的位进行相或
^	$a \wedge b$	将 a 的每个位与 b 相同的位进行异或

位运算符的与、或、非与逻辑运算符逻辑与、逻辑或、逻辑非使用时候容易混淆, 逻辑运算符一般用在条件判断上, 位运算符一般用在信号赋值上。

## 6) 移位运算符

移位运算符包括左移位运算符和右移位运算符,这两种移位运算符都用 0 来填补移出的空位。如下表所示。

表 7.2.6 移位运算符

符号	使用方法	说明
<<	a << b	将 a 左移 b 位
>>	a >> b	将 a 右移 b 位

假设 a 有 8bit 数据位宽,那么 a<<2,表示 a 左移 2bit, a 还是 8bit 数据位宽, a 的最高 2bit 数据被移位丢弃了,最低 2bit 数据固定补 0。如果 a 是 3 (二进制: 00000011), 那么 3 左移 2bit, 3<<2, 就是 12 (二进制: 00001100)。一般使用左移位运算代替乘法,右移位运算代替除法,但是这种也只能表示 2 的指数次幂的乘除法。

## 7) 拼接运算符

Verilog 中有一个特殊的运算符是 C 语言中没有的,就是位拼接运算符。用这个运算符可以把两个或多个信号的某些位拼接起来进行运算操作。如下表所示。


表 7.2.7 位拼接运算符

符号	使用方法	说明
{}	{a, b}	将 a 和 b 拼接起来,作为一个新信号

## 8) 运算符的优先级

介绍完了这么多运算符,大家可能会想到究竟哪个运算符高,哪个运算符低。为了便于大家查看这些运算符的优先级,我们将它们制作成了表格,如下表所示。

表 7.2.8 运算符的优先级

运算符	优先级
!, ~	最高
*, /, %	次高
+, -	
<<, >>	
<, <=, >, >=	
==, !=, ===, !==	
&	
^, ^^	
&&	
	次低
?	最低

## 7.3 Verilog 程序框架

在介绍 Verilog 程序框架之前,我们先来看下 Verilog 一些基本语法,基础语法主要包括注释和关键字。



### 7.3.1 注释

Verilog HDL 中有两种注释的方式, 一种是以 “/\*” 符号开始, “\*/” 结束, 在两个符号之间的语句都是注释语句, 因此可扩展到多行。如:

```
/* statement1 ,
statement2,
.....
statementn */
```

以上 n 个语句都是注释语句。

另一种是以 // 开头的语句, 它表示以 // 开始到本行结束都属于注释语句。如:

```
//statement1
```

我们建议的写法: 使用 // 作为注释。

### 7.3.2 关键字

Verilog 和 C 语言类似, 都因编写需要定义了一系列保留字, 叫做关键字 (或关键词)。这些保留字是识别语法的关键。我们给大家列出了 Verilog 中的关键字, 如下表所示。

表 7.3.1 Verilog 的所有关键字

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

虽然上表列了很多, 但是实际经常使用的不是很多, 实际经常使用的主要如下表所示。

表 7.3.2 Verilog 常用的关键字

关键字	含义
-----	----



<b>module</b>	模块开始定义
<b>input</b>	输入端口定义
<b>output</b>	输出端口定义
<b>inout</b>	双向端口定义
<b>parameter</b>	信号的参数定义
<b>wire</b>	wire信号定义
<b>reg</b>	reg信号定义
<b>always</b>	产生reg信号语句的关键字
<b>assign</b>	产生wire信号语句的关键字
<b>begin</b>	语句的起始标志
<b>end</b>	语句的结束标志
<b>posedge/negedge</b>	时序电路的标志
<b>case</b>	Case语句起始标记
<b>default</b>	Case语句的默认分支标志
<b>endcase</b>	Case语句结束标记
<b>if</b>	if/else语句标记
<b>else</b>	if/else语句标记
<b>for</b>	for语句标记
<b>endmodule</b>	模块结束定义

注意只有小写的关键字才是保留字。例如, 标识符 `always`(这是个关键词)与标识符 `ALWAYS`(非关键词)是不同的。

## 7.3.3 程序框架

我们以 LED 流水灯程序为例来给大家展示 Verilog 的程序框架, 代码如下所示(注意: 代码中前面的行号只是为了方便大家阅读代码与快速定位到行号的位置, 在实际编写代码时不可以添加行号, 否则编译代码时会报错)。

```

1  module led(
2      input          sys_clk , //系统时钟
3      input          sys_rst_n, //系统复位, 低电平有效
4      output reg [3:0] led      //4位LED灯
5  );
6
7  //parameter define
8  parameter WIDTH      = 25      ;
9  parameter COUNT_MAX = 25_000_000; //板载50M时钟=20ns, 0.5s/20ns=25000000, 需要25bit
10                                     //位宽
11
12 //reg define
13 reg [WIDTH-1:0] counter      ;
14 reg [1:0]      led_ctrl_cnt;
15

```

```
16 //wire define
17 wire                counter_en  ;
18
19 //*****
20 /**                                main code
21 //*****
22
23 //计数到最大值时产生高电平使能信号
24 assign counter_en = (counter == (COUNT_MAX - 1'b1)) ? 1'b1 : 1'b0;
25
26 //用于产生0.5秒使能信号的计数器
27 always @(posedge sys_clk or negedge sys_rst_n) begin
28     if (sys_rst_n == 1'b0)
29         counter <= 1'b0;
30     else if (counter_en)
31         counter <= 1'b0;
32     else
33         counter <= counter + 1'b1;
34 end
35
36 //led流水控制计数器
37 always @(posedge sys_clk or negedge sys_rst_n) begin
38     if (sys_rst_n == 1'b0)
39         led_ctrl_cnt <= 2'b0;
40     else if (counter_en)
41         led_ctrl_cnt <= led_ctrl_cnt + 2'b1;
42 end
43
44 //通过控制I0口的高低电平实现发光二极管的亮灭
45 always @(posedge sys_clk or negedge sys_rst_n) begin
46     if (sys_rst_n == 1'b0)
47         led <= 4'b0;
48     else begin
49         case (led_ctrl_cnt)
50             2'd0 : led <= 4'b0001;
51             2'd1 : led <= 4'b0010;
52             2'd2 : led <= 4'b0100;
53             2'd3 : led <= 4'b1000;
54             default : ;
55         endcase
56     end
57 end
```

58

59 `endmodule`

首先//开头的都是注释, 这个之前我们讲解过了。下面我们来看下具体的解释。

第 1 行为模块定义, 模块定义以 `module` 开始, `endmodule` 结束, 如 59 行所示。

其次 2 到 5 行为端口定义, 需要定义 `led` 模块的输入信号和输出信号, 此处输入信号为系统时钟和复位信号, 输出为 `led` 控制信号。

7 到 9 行为参数 `parameter` 定义, 语法如 7 到 9 行所示, 定义 `parameter` 的好处是可以灵活改变参数数字就能控制一些计数器最大计数值或者信号位宽的最大位宽。

12 到 14 行为 `reg` 信号定义, `reg` 信号一般情况下代表寄存器, 比如此处控制 0.5 秒使能信号的计数器 `counter`。

16 到 17 行为 `wire` 信号定义, `wire` 信号就是硬件连线, 比如此处的 `counter_en`, 代表计数到最大值时产生高电平使能, 本质上是一个硬件连线, 其实代表的是一些计数器/寄存器做逻辑判断的结果。

19 到 21 行为 `moudle` 开始的注释, 不添加工具综合也不会报错, 但是我们推荐添加, 作为一个良好的编程规范。

23 到 24 行为 `assign` 语句的样式, 条件成立选择 1, 否则选择 0。

26 到 34 行是 `always` 语句的样式, 27 行代表在时钟上升沿或者复位的下降沿进行信号触发。`begin/end` 代表语句的开始和结束。28 到 33 行为 `if/else` 语句, 和 C 语言是比较类似的。29 行的 “`<=`” 标记代表信号是非阻塞赋值, 信号赋值有非阻塞赋值和阻塞赋值两个方式, 这个我们后面会详细解释。

36 和 42 行也是一个 `always` 语句, 和 26 到 34 行类似。

44 和 57 行也是一个 `always` 语句, 不过这个 `always` 语句中嵌入了一个 `case` 语句, `case` 语句的语法如 49 到 55 行所示, 需要一个 `case` 关键字开始, `endcase` 关键字结束, `default` 作为默认分支, 和 C 语言也是类似的。当然 `case` 语句也可以用在不带时钟的 `always` 语句中, 不过本例子的 `always` 都是带有时钟的。不带时钟的 `always` 和带时钟的 `always` 语句的差异这个我们后面也会详细解释。

59 行是 `endmodule` 标记, 代表模块的结束。

在这里需要补充一点的是, 一些初学者可能会有这样一个疑问, 在 `always` 语句中编写 `if` 语句或 `else` 语句时, 后面需要加 `begin` 和 `end` 吗? 其实这个主要看 `if` 条件后面跟着几条赋值语句, 如果只有一条赋值语句时, `if` 后面可以加 `begin` 和 `end`, 也可以不加; 如果超过一条赋值语句时, 就必须加上 `begin` 和 `end`。

`if` 条件只有一条赋值语句时, 下面两种写法都是可以的, 这里更推荐第一种写法, 因为第二种写法会占用更多的行号, 代码如下所示:

```
if(en == 1'b1)
```

```
    a <= 1'b1;
```

或者

```
if(en == 1'b1) begin
```

```
    a <= 1'b1;
```

```
end
```

对于 `if` 条件超过一条赋值语句的情况, 必须添加 `begin` 和 `end`, 代码如下所示:

```
if(en == 1'b1) begin
```

```
    b <= 1'b1;
```

```
    c <= 1'b1;
```

```
end
```

好了, 程序框架就讲解完了, 大家是不是觉得也很简单呢? 这些都是基本的语法规则, 希望大家能记住这些基础的知识点。如果有些地方大家还是觉得比较抽象, 很难理解, 没有关系, 相信大家会在后面的

## 7.4 Verilog 高级知识点

前几节主要介绍了 Verilog 一些基础的知识点和程序框架, 本节给大家介绍一些高级的知识点。高级知识点包括阻塞赋值和非阻塞赋值、assign 和 always 语句差异、什么是锁存器、状态机、模块化设计等。

### 7.4.1 阻塞赋值 (Blocking)

阻塞赋值, 顾名思义, 即在一个 always 块中, 后面的语句会受到前语句的影响, 具体来说, 在同一个 always 中, 一条阻塞赋值语句如果没有执行结束, 那么该语句后面的语句就不能被执行, 即被“阻塞”。也就是说 always 块内的语句是一种顺序关系, 这里和 C 语言很类似。符号“=”用于阻塞的赋值(如: `b = a;`), 阻塞赋值“=”在 begin 和 end 之间的语句是顺序执行, 属于串行语句。

在这里定义两个缩写:

RHS: 赋值等号右边的表达式或变量可以写作 RHS 表达式或 RHS 变量;

LHS: 赋值等号左边的表达式或变量可以写作 LHS 表达式或 LHS 变量;

阻塞赋值的执行可以认为是只有一个步骤的操作, 即计算 RHS 的值并更新 LHS, 此时不允许任何其他语句的干扰, 所谓的阻塞的概念就是值在同一个 always 块中, 其后面的赋值语句从概念上来讲是在前面一条语句赋值完成后才执行的。

为了方便大家理解阻塞赋值的概念以及阻塞赋值和非阻塞赋值的区别, 我们这里以在时序逻辑下使用阻塞赋值为例来实现这样一个功能: 在复位的时候, `a=1`, `b=2`, `c=3`; 而在没有复位的时候, `a` 的值清零, 同时将 `a` 的值赋值给 `b`, `b` 的值赋值给 `c`, 代码以及信号波形图如下图所示:

```

always @(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
        a = 1;
        b = 2;
        c = 3;
    end
    else begin
        a = 0;
        b = a;
        c = b;
    end
end
    
```

图 7.4.1 阻塞赋值代码

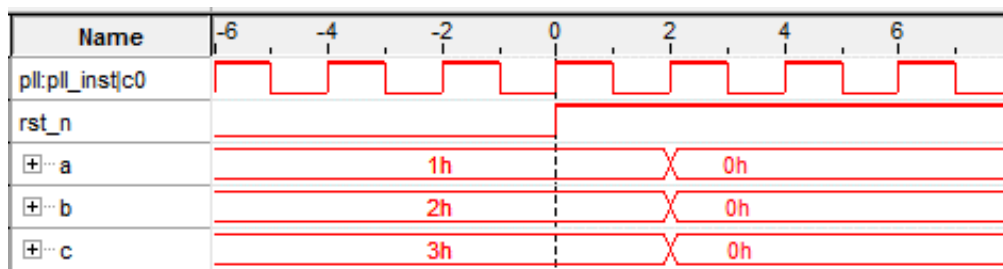


图 7.4.2 阻塞赋值的信号波形图

代码中使用的是阻塞赋值语句，从波形图中可以看到，在复位的时候（ $\text{rst\_n}=0$ ）， $a=1$ ， $b=2$ ， $c=3$ ；而结束复位之后（波形图中的 0 时刻），当  $\text{clk}$  的上升沿到来时（波形图中的 2 时刻）， $a=0$ ， $b=0$ ， $c=0$ 。这是因为阻塞赋值是在当前语句执行完成之后，才会执行后面的赋值语句，因此首先执行的是  $a=0$ ，赋值完成后将  $a$  的值赋值给  $b$ ，由于此时  $a$  的值已经为 0，所以  $b=a=0$ ，最后执行的是将  $b$  的值赋值给  $c$ ，而  $b$  的值已经赋值为 0，所以  $c$  的值同样等于 0。

#### 7.4.2 非阻塞赋值（Non-Blocking）

符号“ $\leq$ ”用于非阻塞赋值（如： $b \leq a$ ），非阻塞赋值是由时钟节拍决定，在时钟上升到来时，执行赋值语句右边，然后将  $\text{begin-end}$  之间的所有赋值语句同时赋值到赋值语句的左边，注意：是  $\text{begin—end}$  之间的所有语句，一起执行，且一个时钟只执行一次，属于并行执行语句。这个是和 C 语言最大的一个差异点，大家要逐步理解并行执行的概念。

非阻塞赋值的操作过程可以看作两个步骤：

- （1）赋值开始的时候，计算 RHS；
- （2）赋值结束的时候，更新 LHS。

所谓的非阻塞的概念是指，在计算非阻塞赋值的 RHS 以及 LHS 期间，允许其它的非阻塞赋值语句同时计算 RHS 和更新 LHS。

我们下面使用非阻塞赋值同样来实现这样一个功能：在复位的时候， $a=1$ ， $b=2$ ， $c=3$ ；而在没有复位的时候， $a$  的值清零，同时将  $a$  的值赋值给  $b$ ， $b$  的值赋值给  $c$ ，代码以及信号波形图如下图所示：

```

always @(posedge clk or negedge rst_n) begin
    if(!rst_n)begin
        a <= 1;
        b <= 2;
        c <= 3;
    end
    else begin
        a <= 0;
        b <= a;
        c <= b;
    end
end
    
```

图 7.4.3 非阻塞赋值代码

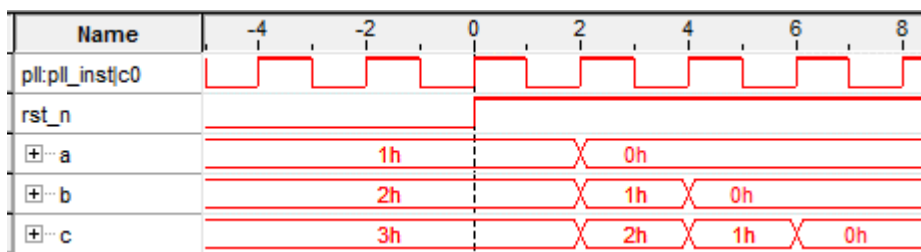


图 7.4.4 非阻塞赋值的信号波形图

代码中使用的是非阻塞赋值语句，从波形图中可以看到，在复位的时候（ $\text{rst\_n}=0$ ）， $a=1$ ， $b=2$ ， $c=3$ ；而结束复位之后（波形图中的 0 时刻），当  $\text{clk}$  的上升沿到来时（波形图中的 2 时刻）， $a=0$ ， $b=1$ ， $c=2$ 。这是因为非阻塞赋值在计算 RHS 和更新 LHS 期间，允许其它的非阻塞赋值语句同时计算 RHS 和更新 LHS。在波形图中的 2 时刻，RHS 的表达是 0、 $a$ 、 $b$ ，分别等于 0、1、2，这三条语句是同时更新 LHS，所以  $a$ 、

b、c 的值分别等于 0、1、2。

在了解了阻塞赋值和非阻塞赋值的区别之后,有些朋友可能还是对什么时候使用阻塞赋值,什么时候使用非阻塞赋值有些疑惑,在这里给大家总结如下。

在描述组合逻辑电路的时候,使用阻塞赋值,比如 assign 赋值语句和不带时钟的 always 赋值语句,这种电路结构只与输入电平的变化有关系,代码如下:

示例1: assign 赋值语句

```
assign data = (data_en == 1'b1) ? 8'd255 : 8'd0;
```

示例2: 不带时钟的 always 语句

```
always @(*) begin
    if (en) begin
        a = a0;
        b = b0;
    end
    else begin
        a = a1;
        b = b1;
    end
end
```

在描述时序逻辑的时候,使用非阻塞赋值,综合成时序逻辑的电路结构,比如带时钟的 always 语句;这种电路结构往往与触发沿有关系,只有在触发沿时才可能发生赋值的变化,代码如下:

示例 3:

```
always @(posedge sys_clk or negedge sys_rst_n) begin
    if (!sys_rst_n) begin
        a <= 1'b0;
        b <= 1'b0;
    end
    else begin
        a <= c;
        b <= d;
    end
end
```

### 7.4.3 assign 和 always 区别

assign 语句和 always 语句是 Verilog 中的两个基本语句,这两个都是经常使用的语句。

assign 语句使用时不能带时钟。

always 语句可以带时钟,也可以不带时钟。在 always 不带时钟时,逻辑功能和 assign 完全一致,都是只产生组合逻辑。比较简单的组合逻辑推荐使用 assign 语句,比较复杂的组合逻辑推荐使用 always 语句。示例如下:

```
24 assign counter_en = (counter == (COUNT_MAX - 1'b1)) ? 1'b1 : 1'b0;
45 always @(*) begin
49     case (led_ctrl_cnt)
50         2'd0 : led = 4'b0001;
```

```
51          2'd1      : led = 4'b0010;
52          2'd2      : led = 4'b0100;
53          2'd3      : led = 4'b1000;
54          default : led = 4'b0000;
55      endcase
57 end
```

#### 7.4.4 带时钟和不带时钟的 always

always 语句可以带时钟,也可以不带时钟。在 always 不带时钟时,逻辑功能和 assign 完全一致,虽然产生的信号定义还是 reg 类型,但是该语句产生的还是组合逻辑。

```
44 reg [3:0] led;
45 always @(*) begin
46     case (led_ctrl_cnt)
47         2'd0      : led = 4'b0001;
48         2'd1      : led = 4'b0010;
49         2'd2      : led = 4'b0100;
50         2'd3      : led = 4'b1000;
51         default : led = 4'b0000;
52     endcase
53 end
```

在 always 带时钟信号时,这个逻辑语句才能产生真正的寄存器,如下示例 counter 就是真正的寄存器。

```
26 //用于产生 0.5 秒使能信号的计数器
27 always @(posedge sys_clk or negedge sys_rst_n) begin
28     if (sys_rst_n == 1'b0)
29         counter <= 1'b0;
30     else if (counter_en)
31         counter <= 1'b0;
32     else
33         counter <= counter + 1'b1;
34 end
```

#### 7.4.5 什么是 latch

latch 是指锁存器,是一种对脉冲电平敏感的存储单元电路。锁存器和寄存器都是基本存储单元,锁存器是电平触发的存储器,寄存器是边沿触发的存储器。两者的基本功能是一样的,都可以存储数据。锁存器是组合逻辑产生的,而寄存器是在时序电路中使用,由时钟触发产生的。

latch 的主要危害是会产生毛刺 (glitch),这种毛刺对下一级电路是很危险的。并且其隐蔽性很强,不易查出。因此,在设计中,应尽量避免 latch 的使用。

代码里面出现 latch 的两个原因是在组合逻辑中,if 或者 case 语句不完整的描述,比如 if 缺少 else 分支,case 缺少 default 分支,导致代码在综合过程中出现了 latch。解决办法就是 if 必须带 else 分支,case 必须带 default 分支。

大家需要注意下,只有不带时钟的 always 语句 if 或者 case 语句不完整才会产生 latch,带时钟的语句 if



或者 case 语句不完整描述不会产生 latch。

下面为缺少 else 分支的带时钟的 always 语句和不带时钟的 always 语句, 通过实际产生的电路图可以看到第二个是有一个 latch 的, 第一个仍然是普通的带有时钟的寄存器。

<pre>always @(posedge clk)begin     if(enable)         q &lt;= data;     //else     // q &lt;= 0 ; end endmodule</pre>	<pre>always @(*)begin     if(enable)         q &lt;= data;     //     //     // else     //     q &lt;= 0 ; end endmodule</pre>
--	---

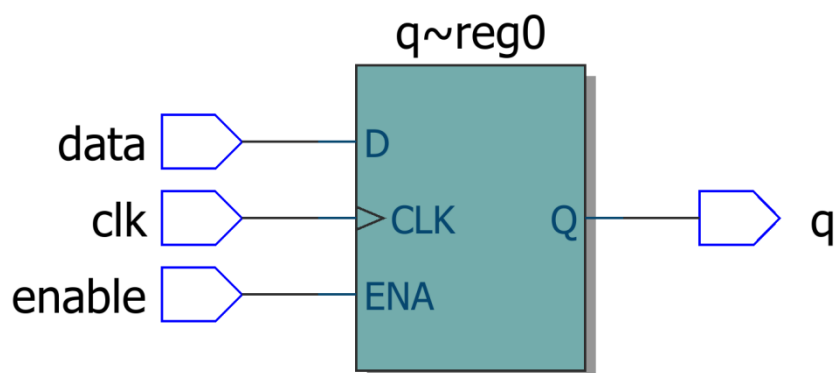


图 7.4.5 缺少 else 的带时钟的 always 语句电路图

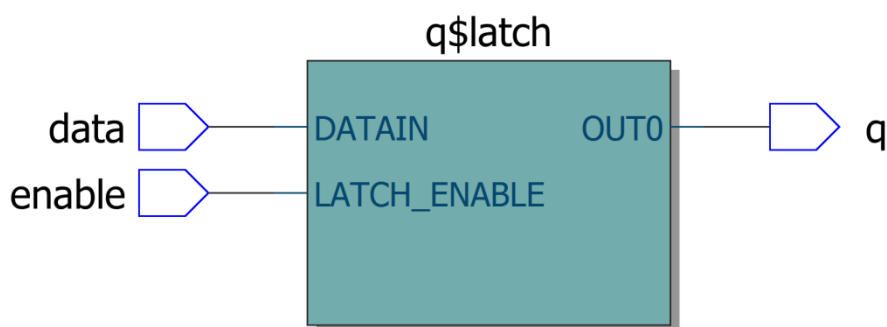


图 7.4.6 缺少 else 的不带时钟的 always 语句电路图

## 7.4.6 状态机

Verilog 是硬件描述语言, 硬件电路是并行执行的, 当需要按照流程或者步骤来完成某个功能时, 代码中通常会使用很多个 if 嵌套语句来实现, 这样就增加了代码的复杂度, 以及降低了代码的可读性, 这个时候就可以使用状态机来编写代码。状态机相当于一个控制器, 它将一项功能的完成分解为若干步, 每一步对应于二进制的状态, 通过预先设计的顺序在各状态之间进行转换, 状态转换的过程就是实现逻辑功能的过程。

状态机, 全称是有限状态机 (Finite State Machine, 缩写为 FSM), 是一种在有限个状态之间按一定规律转换的时序电路, 可以认为是组合逻辑和时序逻辑的一种组合。状态机通过控制各个状态的跳转来控制流程, 使得整个代码看上去更加清晰易懂, 在控制复杂流程的时候, 状态机优势明显, 因此基本上都会用到状态机, 如 SDRAM 控制器等。在本手册提供的例程中, 会有多个用到状态机设计的例子, 希望大家能够慢慢体会和理解, 并且能够熟练掌握。

根据状态机的输出是否与输入条件相关, 可将状态机分为两大类, 即摩尔(Moore)型状态机和米勒(Mealy)

型状态机。

- Mealy 状态机: 组合逻辑的输出不仅取决于当前状态, 还取决于输入状态。
- Moore 状态机: 组合逻辑的输出只取决于当前状态。

### 1) Mealy 状态机

米勒状态机的模型如下图所示, 模型中第一个方框是指产生下一状态的组合逻辑 F, F 是当前状态和输入信号的函数, 状态是否改变、如何改变, 取决于组合逻辑 F 的输出; 第二框图是指状态寄存器, 其由一组触发器组成, 用来记忆状态机当前所处的状态, 状态的改变只发生在时钟的跳边沿; 第三个框图是指产生输出的组合逻辑 G, 状态机的输出是由输出组合逻辑 G 提供的, G 也是当前状态和输入信号的函数。

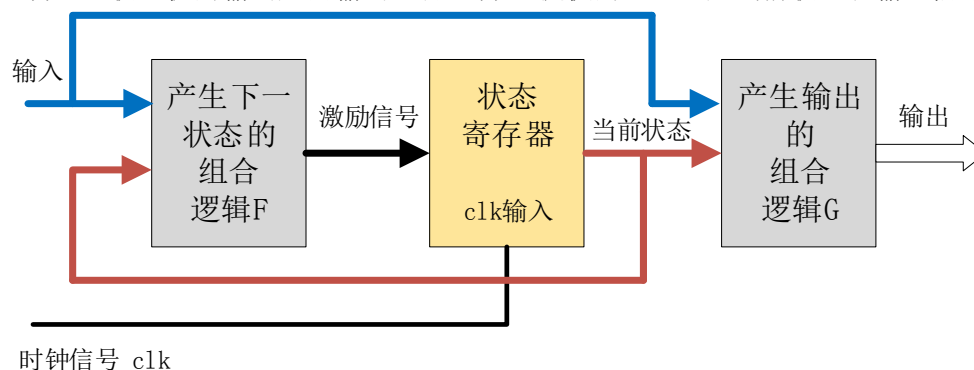


图 7.4.7 Mealy 状态机模型

### 2) Moore 状态机

摩尔状态机的模型如下图所示, 对比米勒状态机的模型可以发现, 其区别在于米勒状态机的输出由当前状态和输入条件决定的, 而摩尔状态机的输出只取决于当前状态。

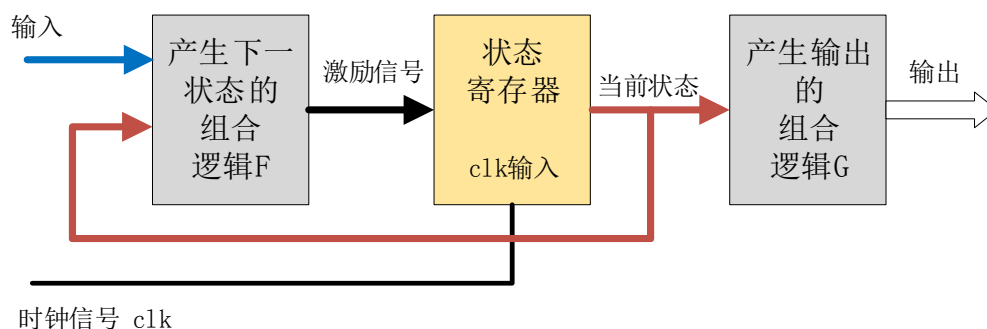


图 7.4.8 Moore 状态机模型

### 3) 三段式状态机

根据状态机的实际写法, 状态机还可以分为一段式、二段式和三段式状态机。

一段式: 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。不推荐采用这种状态机, 因为从代码风格方面来讲, 一般都会要求把组合逻辑和时序逻辑分开; 从代码维护和升级来说, 组合逻辑和时序逻辑混合在一起不利于代码维护和修改, 也不利于约束。

二段式: 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。不同于一段式状态机的是, 它需要定

义两个状态，现态和次态，然后通过现态和次态的转换来实现时序逻辑。

三段式：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 `always` 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` 模块描述状态输出(可以用组合电路输出，也可以时序电路输出)。

实际应用中三段式状态机使用最多，因为三段式状态机将组合逻辑和时序分开，有利于综合器分析优化以及程序的维护；并且三段式状态机将状态转移与状态输出分开，使代码看上去更加清晰易懂，提高了代码的可读性，推荐大家使用三段式状态机，本文也着重讲解三段式。

三段式状态机的基本格式是：

第一个 `always` 语句实现同步状态跳转；

第二个 `always` 语句采用组合逻辑判断状态转移条件；

第三个 `always` 语句描述状态输出(可以用组合电路输出，也可以时序电路输出)。

在开始编写状态机代码之前，一般先画出状态跳转图，这样在编写代码时思路会比较清晰，下面以一个 7 分频为例（对于分频等较简单的功能，可以不使用状态机，这里只是演示状态机编写的方法），状态跳转图如下图所示：

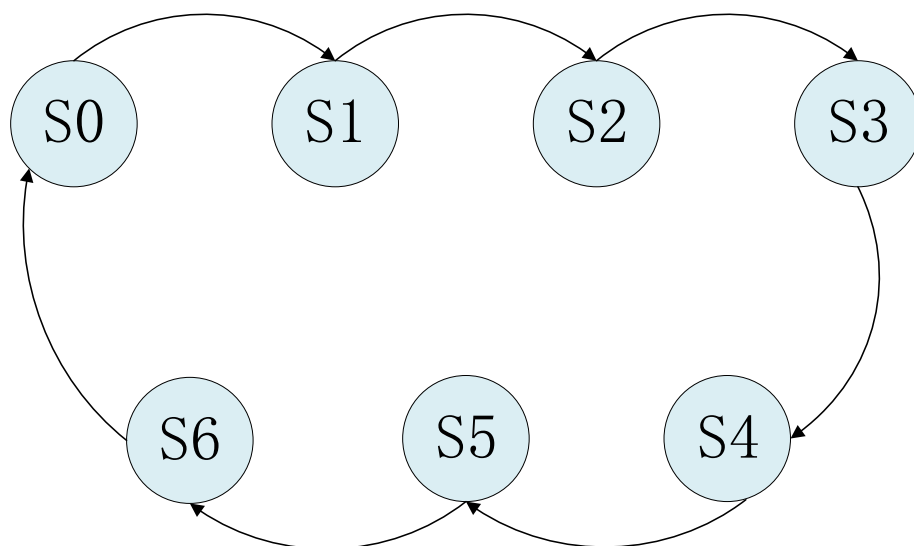


图 7.4.9 七分频状态跳转图

状态跳转图画完之后，接下来通过 `parameter` 来定义各个不同状态的参数，如下代码所示：

```

parameter S0 = 7'b0000001; //独热码定义方式
parameter S1 = 7'b0000010;
parameter S2 = 7'b0000100;
parameter S3 = 7'b0001000;
parameter S4 = 7'b0010000;
parameter S5 = 7'b0100000;
parameter S6 = 7'b1000000;
  
```

这里是使用独热码的方式来定义状态机，每个状态只有一位为 1，当然也可以直接定义成十进制的 0, 1, 2.....7。

因为我们定义成独热码的方式，每一个状态的位宽为 7 位，接下来还需要定义两个 7 位的寄存器，一个用来表示当前状态，另一个用来表示下一个状态，如下所示：

```

reg [6:0] curr_st ; //当前状态
  
```

```
reg [6:0] next_st ; //下一个状态
```

接下来就可以使用三个 `always` 语句来开始编写状态机的代码, 第一个 `always` 采用同步时序描述状态转移, 第二个 `always` 采用组合逻辑判断状态转移条件, 第三个 `always` 是描述状态输出, 一个完整的三段式状态机的例子如下代码所示:

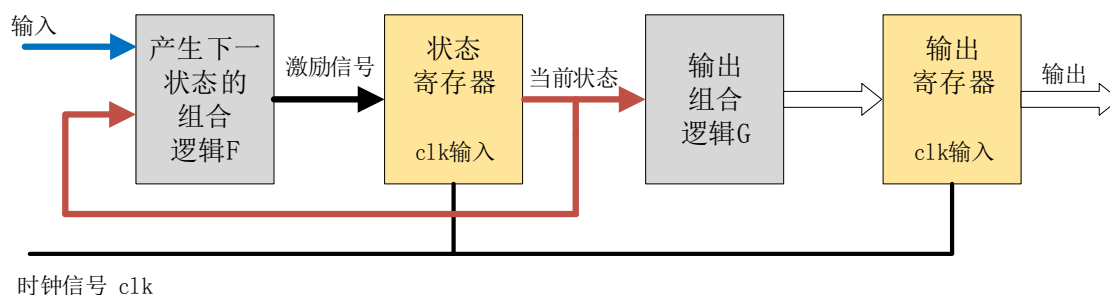
```
1 module divider7_fsm (
2     //系统时钟与复位
3     input    sys_clk      ,
4     input    sys_rst_n    ,
5
6     //输出时钟
7     output reg clk_divide_7
8 );
9
10 //parameter define
11 parameter S0 = 7'b0000001; //独热码定义方式
12 parameter S1 = 7'b0000010;
13 parameter S2 = 7'b0000100;
14 parameter S3 = 7'b0001000;
15 parameter S4 = 7'b0010000;
16 parameter S5 = 7'b0100000;
17 parameter S6 = 7'b1000000;
18
19 //reg define
20 reg [6:0] curr_st ; //当前状态
21 reg [6:0] next_st ; //下一个状态
22
23 //*****
24 /**                                main code
25 //*****
26
27 //状态机的第一段采用同步时序描述状态转移
28 always @(posedge sys_clk or negedge sys_rst_n) begin
29     if (!sys_rst_n)
30         curr_st <= S0;
31     else
32         curr_st <= next_st;
33 end
34
35 //状态机的第二段采用组合逻辑判断状态转移条件
36 always @(*) begin
37     case (curr_st)
38         S0: next_st = S1;
```

```
39      S1: next_st = S2;
40      S2: next_st = S3;
41      S3: next_st = S4;
42      S4: next_st = S5;
43      S5: next_st = S6;
44      S6: next_st = S0;
45      default: next_st = S0;
46  endcase
47 end
48
49 //状态机的第三段描述状态输出（这里采用时序电路输出）
50 always @(posedge sys_clk or negedge sys_rst_n) begin
51     if (!sys_rst_n)
52         clk_divide_7 <= 1'b0;
53     else if ((curr_st == S0) | (curr_st == S1) | (curr_st == S2) | (curr_st == S3))
54         clk_divide_7 <= 1'b0;
55     else if ((curr_st == S4) | (curr_st == S5) | (curr_st == S6))
56         clk_divide_7 <= 1'b1;
57     else
58         ;
59 end
60
61 endmodule
```

在编写状态机代码时首先要定义状态变量（代码中的参数 S0~S6）与状态寄存器（curr\_st、next\_st），如代码中第 10 行至第 21 行所示；接下来使用三个 always 语句来实现三段状态机，第一个 always 语句实现同步状态跳转（如代码的第 27 至第 33 行所示），在复位的时候，当前状态处在 S0 状态，否则将下一个状态赋值给当前状态；第二个 always 采用组合逻辑判断状态转移条件（如代码的第 35 行至第 47 行代码所示），这里每一个状态只保持一个时钟周期，也就是直接跳转到下一个状态，在实际应用中，一般根据输入的条件来判断是否跳转到其它状态或者停留在当前状态，最后在 case 语句后面增加一个 default 语句，来防止状态机处在异常的状态；第三个 always 输出分频后的时钟（如代码的第 49 至第 59 行代码所示），状态机的第三段可以使用组合逻辑电路输出，也可以使用时序逻辑电路输出，一般推荐使用时序电路输出，因为状态机的设计和其它设计一样，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。

从代码中可以看出，输出的分频时钟 clk\_divide\_7 只与当前状态（curr\_st）有关，而与输入状态无关，所以属于摩尔型状态机。状态机的第一段对应摩尔状态机模型的状态寄存器，用来记忆状态机当前所处的状态；状态机的第二段对应摩尔状态机模型产生下一状态的组合逻辑 F；状态机的第三段对应摩尔状态机产生输出的组合逻辑 G，因为采用时序电路输出有很大的优势，所以这里第三段状态机是由时序电路输出的。

状态机采用时序逻辑输出的状态机模型如下图所示：



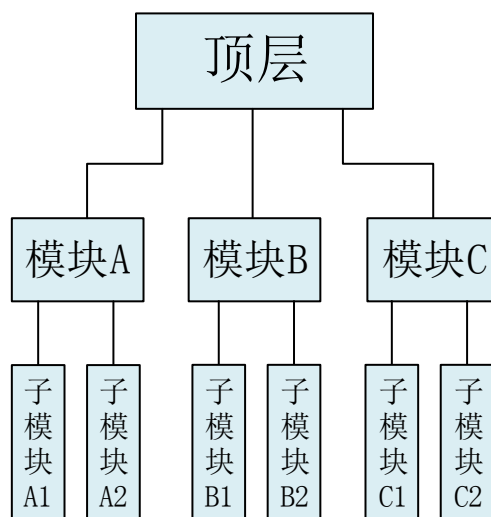
采用这种描述方法虽然代码结构复杂了一些，但是这样做的好处是可以有效地滤去组合逻辑输出的毛刺，同时也可以更好的进行时序计算与约束，另外对于总线形式的输出信号来说，容易使总线数据对齐，减小总线数据间的偏移，从而降低接收端数据采样出错的频率。

### 7.4.7 模块化设计

模块化设计是 FPGA 设计中一个很重要的技巧，它能够使一个大型设计的分工协作、仿真测试更加容易，代码维护或升级更加便利，当更改某个子模块时，不会影响其它模块的实现结果。进行模块化、标准化设计的最终目的就是提高设计的通用性，减少不同项目中同一功能设计和验证引入的工作量。划分模块的基本原则是子模块功能相对独立、模块内部联系尽量紧密、模块间的连接尽量简单。

在进行模块化设计中，对于复杂的数字系统，我们一般采用自顶向下的设计方式。可以把系统划分成几个功能模块，每个功能模块再划分成下一层的子模块；每个模块的设计对应一个 `module`，一个 `module` 设计成一个 Verilog 程序文件。因此，对一个系统的顶层模块，我们采用结构化的设计，即顶层模块分别调用了各个功能模块。

下图是模块化设计的功能框图，一般整个设计的顶层模块只做例化（调用其它模块），不做逻辑。顶层下面会有模块 A、模块 B、模块 C 等，模块 A/B/C 又可以分多个子模块实现。



在这里我们补充一个概念，就是 Verilog 语法中的模块例化。FPGA 逻辑设计中通常是一个大的模块中包含了一个或多个功能子模块，Verilog 通过模块调用或称为模块实例化的方式来实现这些子模块与高层模块的连接，有利于简化每一个模块的代码，易于维护和修改。

下面以一个实例(静态数码管显示实验)来说明模块和模块之间的例化方法。

在静态数码管显示实验中, 我们根据功能将 FPGA 顶层例化了以下两个模块: 计时模块 (time\_count) 和数码管静态显示模块 (seg\_led\_static), 如下图所示:

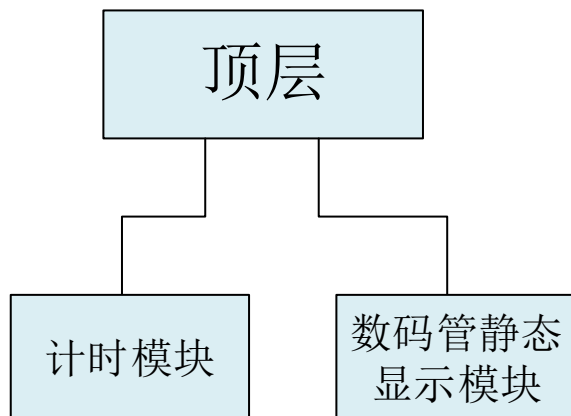


图 7.4.12 静态数码管显示模块框图

计时模块部分代码如下所示:

```

1 module time_count (
2     input      clk      ,    // 时钟信号
3     input      rst_n    ,    // 复位信号
4
5     output reg  flag     // 一个时钟周期的脉冲信号
6 );
7
8 //parameter define
9 parameter MAX_NUM = 25000_000; // 计数器最大计数值
10 .....
  
```

34 endmodule

数码管静态显示模块部分代码如下所示:

```

1 module seg_led_static (
2     input      clk      ,    // 时钟信号
3     input      rst_n    ,    // 复位信号 (低有效)
4
5     input      add_flag,    // 数码管变化的通知信号
6     output reg [5:0] sel   ,    // 数码管位选
7     output reg [7:0] seg_led // 数码管段选
8 );
9 .....
  
```

66 endmodule

顶层模块代码如下所示:

```

1 module seg_led_static_top (
2     input      sys_clk  ,    // 系统时钟
3     input      sys_rst_n,    // 系统复位信号 (低有效)
4
  
```



```

5      output    [5:0]    sel      ,          // 数码管位选
6      output    [7:0]    seg_led   // 数码管段选
7
8 );
9
10 //parameter define
11 parameter TIME_SHOW = 25'd25000_000; // 数码管变化的时间间隔0.5s
12
13 //wire define
14 wire          add_flag;           // 数码管变化的通知信号
15
16 //*****
17 /**                               main code
18 //*****
19
20 //例化计时模块
21 time_count #(
22     .MAX_NUM    (TIME_SHOW)
23 ) u_time_count(
24     .clk        (sys_clk ),
25     .rst_n      (sys_rst_n),
26
27     .flag       (add_flag )
28 );
29
30 //例化数码管静态显示模块
31 seg_led_static u_seg_led_static (
32     .clk        (sys_clk ),
33     .rst_n      (sys_rst_n),
34
35     .add_flag   (add_flag ),
36     .sel        (sel      ),
37     .seg_led    (seg_led  )
38 );
39
40 endmodule

```

我们上面贴出了顶层模块的完整代码，子模块只贴出了模块的端口和参数定义的代码。这是因为顶层模块对子模块做例化时，只需要知道子模块的端口信号名，而不用关心子模块内部具体是如何实现的。如果子模块内部使用 `parameter` 定义了一些参数，Verilog 也支持对参数的例化（也叫参数的传递），即顶层模块可以通过例化参数来修改子模块内定义的参数。

我们先来看一下顶层模块是如何例化子模块的，例化方法如下图所示：

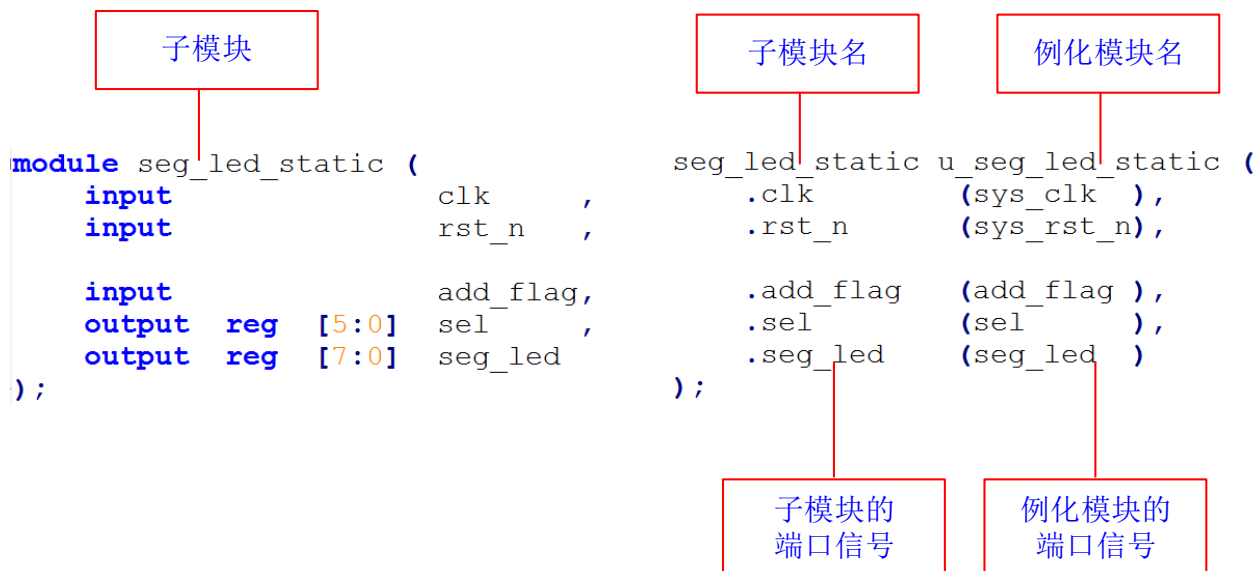


图 7.4.13 模块的例化

上图右侧是例化的数码管静态显示模块，子模块名是指被例化模块的模块名，而例化模块名相当于标识，当例化多个相同模块时，可以通过例化名来识别哪一个例化，我们一般命名为“u\_”+“子模块名”。信号列表中“.”之后的信号是数码管静态显示模块定义的端口信号，括号内的信号则是顶层模块声明的信号，这样就将顶层模块的信号与子模块的信号一一对应起来，同时需要注意信号的位宽要保持一致。

接下来再来介绍一下参数的例化，参数的例化是在模块例化的基础上，增加了对参数的信号定义，如下图所示：

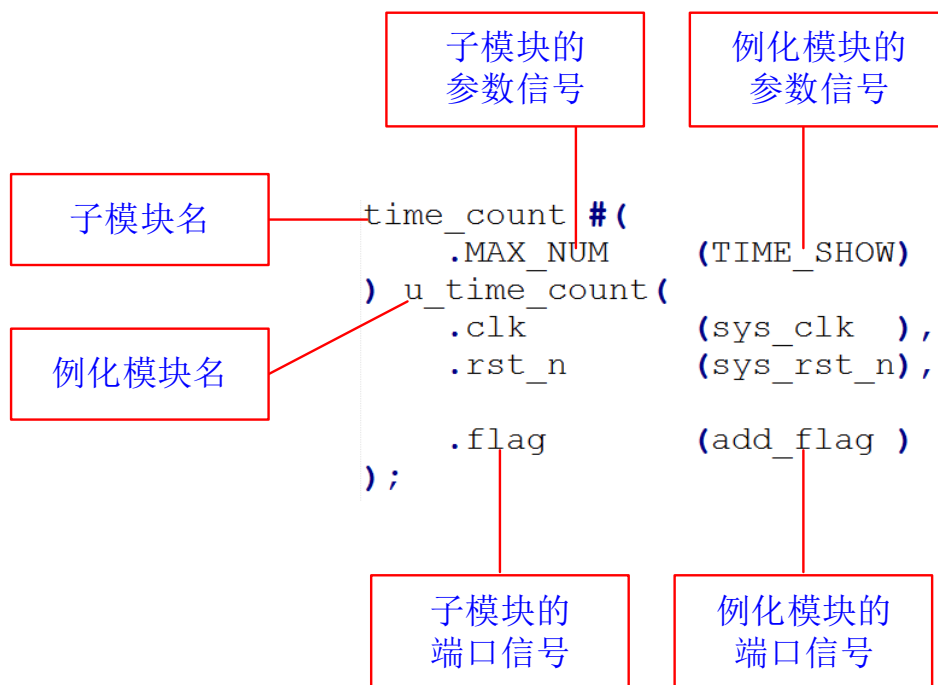


图 7.4.14 模块参数的例化

在对参数进行例化时，在模块名的后面加上“#”，表示后面跟着的是参数列表。计时模块定义的 MAX\_NUM 和顶层模块的 TIME\_SHOW 都是等于 25000\_000，当在顶层模块定义 TIME\_SHOW=12500\_000 时，那么子模块的 MAX\_NUM 的值实际上是也等于 12500\_000。当然即使子模块包含参数，在做模块的例化时也可以不添加对参数的例化，这样的话，子模块的参数值等于该模块内部实际定义的值。

值得一提的是, Verilog 语法中的 `localparam` 代表的意思同样是参数定义, 用法和 `parameter` 基本一致, 区别在于 `parameter` 定义的参数可以做例化, 而 `localparam` 定义的参数是指本地参数, 上层模块不可以对 `localparam` 定义的参数做例化。

## 7.5 Verilog 编程规范

本节主要给大家介绍下编程规范, 良好的编程规范是一个 FPGA 工程师必备的素质。

### 7.5.1 编程规范重要性

当前数字电路设计越来越复杂, 一个项目需要的人越来越多, 当几十号设计同事完成同一个项目时候, 大家需要互相检视对方代码, 如果没有一个统一的编程规范, 那么是不可想象的。大家的风格都不一样, 如果不统一的话, 后续维护、重用等会有很大的困难, 即使是自己写的代码, 几个月后再看也会变的很陌生, 也会看不懂 (您可能不相信, 不过笔者和同事交流发现大家都是这样的, 时间长不看就忘记了), 所以编程规范的重要性显而易见。

另外养成良好的编程规范, 对于个人的工作习惯、思路等都有非常大的好处。可以让新人尽快融入项目中, 让大家更容易看懂您的代码。

### 7.5.2 工程组织形式

工程的组织形式一般包括如下几个部分, 分别是 `doc`、`par`、`rtl` 和 `sim` 四个部分。

```
XX 工程名
|--doc
|--par
|--rtl
|--sim
```

`doc`: 一般存放工程相关的文档, 包括该项目用到的 `datasheet` (数据手册)、设计方案等。不过为了便于大家查看, 我们开发板文档是统一汇总存放在资料盘下的;

`par`: 主要存放工程文件和使用到的一些 IP 文件;

`rtl`: 主要存放工程的 `rtl` 代码, 这是工程的核心, 文件名与 `module` 名称应当一致, 建议按照模块的层次分开存放;

`sim`: 主要存放工程的仿真代码, 复杂的工程里面, 仿真也是不可或缺的部分, 可以极大减少调试的工作量。

### 7.5.3 文件头声明

每一个 Verilog 文件的开头, 都必须有一段声明的文字。包括文件的版权, 作者, 创建日期以及内容介绍等, 如下表所示。

```
/******Copyright (c)*****//
//原子哥在线教学平台: www.yuanzige.com
//技术支持: www.openedv.com
//淘宝店铺: http://openedv.taobao.com
//关注微信公众平台微信号: "正点原子", 免费获取 ZYNQ & FPGA & STM32 & LINUX 资料。
//版权所有, 盗版必究。
//Copyright(C) 正点原子 2018-2028
```

```
//All rights reserved
//-----
// File name:          led_twinkle
// Last modified Date: 2019/4/14 10:55:56
// Last Version:       V1.0
// Descriptions:       LED 灯闪烁
//-----
// Created by:         正点原子
// Created date:       2019/4/14 10:55:56
// Version:           V1.0
// Descriptions:       The original version
//
//-----
//*****
```

我们建议一个.V 只包括一个 module, 这样模块会比较清晰易懂。

## 7.5.4 输入输出定义

端口的输入输出有 Verilog 95 和 2001 两种格式, 推荐大家采用 Verilog 2001 语法格式。下面是 Verilog 2001 语法的一个例子, 包括 module 名字、输入输出、信号名字、输出类型、注释。

```
1 module led(
2     input          sys_clk , //系统时钟
3     input          sys_rst_n, //系统复位, 低电平有效
4     output reg [3:0] led      //4 位 LED 灯
5 );
```

我们建议如下几点:

- ① 一行只定义一个信号;
- ② 信号全部对齐;
- ③ 同一组的信号放在一起。

## 7.5.5 parameter 定义

我们建议如下几点:

- ① module 中的 parameter 声明, 不建议随处乱放;
- ② 将 parameter 定义放在紧跟着 module 的输入输出定义之后;
- ③ parameter 等常量命名全部使用大写。

```
7 //parameter define
8 parameter WIDTH      = 25      ;
9 parameter COUNT_MAX = 25_000_000; //板载50M时钟=20ns, 0.5s/20ns=25000000, 需要25bit
10                                //位宽
```

### 7.5.6 wire/reg 定义

一个 module 中的 wire/reg 变量声明需要集中放在一起, 不建议随处乱放。  
因此, 我们建议如下:

- ① 将 reg 与 wire 的定义放在紧跟着 parameter 之后;
- ② 建议具有相同功能的信号集中放在一起;
- ③ 信号需要对齐, reg 和位宽需要空 2 格, 位宽和信号名字至少空四格;
- ④ 位宽使用降序描述, [6:0];
- ⑤ 时钟使用前缀 clk, 复位使用后缀 rst;
- ⑥ 不能使用 Verilog 关键字作为信号名字;
- ⑦ 一行只定义一个信号。

```
12 //reg define
13 reg    [WIDTH-1:0] counter    ;
14 reg    [1:0]      led_ctrl_cnt;
15
16 //wire define
17 wire                counter_en ;
```

### 7.5.7 信号命名

大家对信号命名可能都有不同的喜好, 我们建议如下:

- ① 信号命名需要体现其意义, 比如 fifo\_wr 代表 FIFO 读写使能;
- ② 可以使用 “\_” 隔开信号, 比如 sys\_clk;
- ③ 内部信号不要使用大写, 也不要使用大小写混合, 建议全部使用小写;
- ④ 模块名字使用小写;
- ⑤ 低电平有效的信号, 使用\_n 作为信号后缀;
- ⑥ 异步信号, 使用\_a 作为信号后缀;
- ⑦ 纯延迟打拍信号使用\_dly 作为后缀。

### 7.5.8 always 块描述方式

always 块的编程规范, 我们建议如下:

- ① if 需要空四格;
- ② 一个 always 需要配一个 begin 和 end;
- ③ always 前面需要有注释;

- ④ beign 建议和 always 放在同一行;
- ⑤ 一个 always 和下一个 always 空一行即可, 不要空多行;
- ⑥ 时钟复位触发描述使用 posedge sys\_clk 和 negedge sys\_rst\_n
- ⑦ 一个 always 块只包含一个时钟和复位;
- ⑧ 时序逻辑使用非阻塞赋值。

```

26 //用于产生0.5秒使能信号的计数器
27 always @(posedge sys_clk or negedge sys_rst_n) begin
28     if (sys_rst_n == 1'b0)
29         counter <= 1'b0;
30     else if (counter_en)
31         counter <= 1'b0;
32     else
33         counter <= counter + 1'b1;
34 end
    
```

## 7.5.9 assign 块描述方式

assign 块的编程规范, 我们建议如下:

- ① assign 的逻辑不能太复杂, 否则易读性不好;
- ② assign 前面需要有注释;
- ③ 组合逻辑使用阻塞赋值。

```

23 //计数到最大值时产生高电平使能信号
24 assign counter_en = (counter == (COUNT_MAX - 1'b1)) ? 1'b1 : 1'b0;
    
```

## 7.5.10 空格和 TAB

由于不同的解释器对于 TAB 翻译不一致, 所以建议不使用 TAB, 全部使用空格。

## 7.5.11 注释

添加注释可以增加代码的可读性, 易于维护。我们建议规范如下:

- ① 注释描述需要清晰、简洁;
- ② 注释描述不要废话, 冗余;
- ③ 注释描述需要使用 “//”;
- ④ 注释描述需要对齐;
- ⑤ 核心代码和信号定义之间需要增加注释。

```

26 //用于产生 0.5 秒使能信号的计数器
    
```

```
27 always @(posedge sys_clk or negedge sys_rst_n) begin
28     if (sys_rst_n == 1'b0)
29         counter <= 1'b0;
30     else if (counter_en)                // counter_en 为 1 时, counter 清 0
31         counter <= 1'b0;
32     else
33         counter <= counter + 1'b1;
34 end
```

### 7.5.12 模块例化

模块例化我们建议规范如下:

- ① module 模块例化使用 u\_xx 表示。

```
20 //例化计时模块
21 time_count #(
22     .MAX_NUM    (TIME_SHOW)
23 ) u_time_count(
24     .clk        (sys_clk ),
25     .rst_n      (sys_rst_n),
26
27     .flag       (add_flag )
28 );
29
30 //例化数码管静态显示模块
31 seg_led_static u_seg_led_static (
32     .clk        (sys_clk ),
33     .rst_n      (sys_rst_n),
34
35     .add_flag   (add_flag ),
36     .sel        (sel      ),
37     .seg_led    (seg_led  )
38 );
```

### 7.5.13 其他注意事项

其他注意事项如下:

- ① 代码写的越简单越好, 方便他人阅读和理解;
- ② 不使用 repeat 等循环语句;
- ③ RTL 级别代码里面不使用 initial 语句, 仿真代码除外;
- ④ 避免产生 Latch 锁存器, 比如组合逻辑里面的 if 不带 else 分支、case 缺少 default 语句;



- ⑤ 避免使用太复杂和少见的语法, 可能造成语法综合器优化力度较低。

良好的编程规范是大家走向专业 FPGA 工程师的必备素质, 希望大家都能养成良好的编程规范。