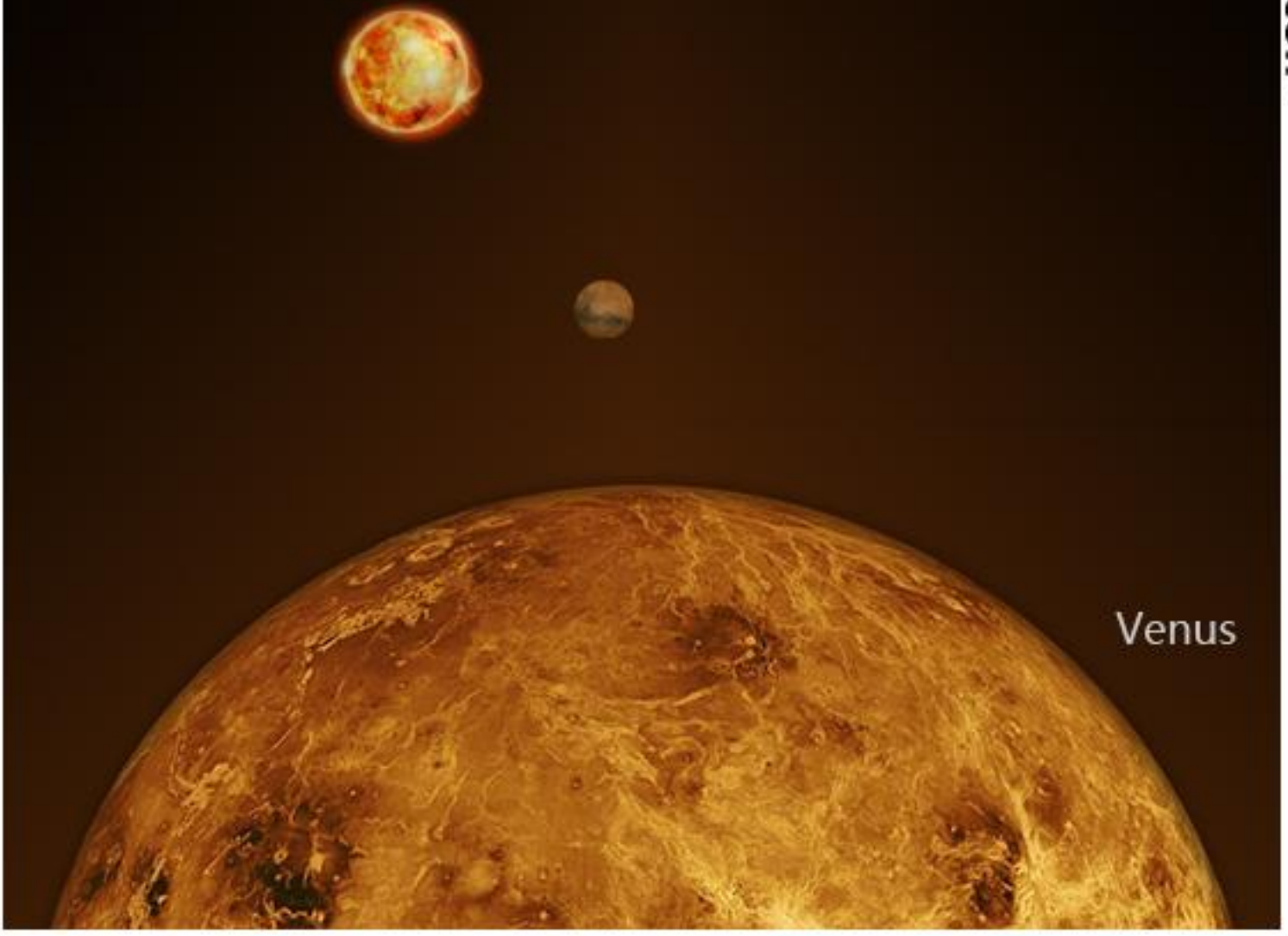
HELLO FPGA

硬件语法篇

锆石科技作品

zircon



前　言

为什么要学硬件语法篇：大家都知道软件设计使用软件编程语言，例如我们熟知的 C、Java 等等，而 FPGA 设计使用的是 HDL 语言，例如 VHDL 和 Verilog HDL。说的直白点，FPGA 的设计就是逻辑电路的实现，就是把我们从数字电路中学到的逻辑电路功能，使用硬件描述语言(Verilog/VHDL)描述出来，这需要设计人员能够用硬件编程思维来编写代码，以及拥有扎实的数字电路功底。

硬件语法篇包含了哪些内容：该篇不仅仅是介绍了 Verilog HDL 基本概念和语法，更着重讲解了 Verilog HDL 的基本设计思想及优良的代码书写规范和风格。

目　录

第一章

Verilog 的综述

第一章

Verilog 的综述

§1.1 什么是Verilog?

Verilog 是 Verilog HDL 的简称，Verilog HDL 是一种硬件描述语言（HDL：Hardware Description Language），硬件描述语言是电子系统硬件行为描述、结构描述、数据流描述的语言。利用这种语言，数字电路系统的设计可以从顶层到底层（从抽象到具体）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。接下去，再用专用集成电路 ASIC 或现场可编程门阵列 FPGA 自动布局布线工具，把网表转换为要实现的具体电路布线结构。

在 FPGA 的设计中，我们有多种设计方式，如绘制原理图、编写描述语言代码等。早起的工程师对原理图的设计方式情有独钟，这种输入方式能够很直观的看出电路的结构并快速理解电路。随着逻辑规模的不断攀升，逻辑电路也越来越复杂，这种输入方式就会显得力不从心，应付简单的逻辑电路还算实用，应付起复杂的逻辑电路就不行了。因此取而代之的便是编写描述语言代码的方式，现今的绝大多数设计都是采用代码来完成的。

目前，主流的硬件描述语言有两种：一种是 VHDL，另一种是 Verilog。VHDL 的全名 Very High Speed Integrated Circuit Hardware Description Language，即 VHSIC，译为超高速集成电路的硬件描述语言。VHDL 发展较早，语法严谨；Verilog 类似 C 语言，语法风格比较自由。Verilog 和 VHDL 这两种硬件描述语言都已成为了 IEEE 标准，VHDL 是在 1987 年成为 IEEE 标准，Verilog 则在 1995 年才正式成为 IEEE 标准。之所以 VHDL 比 Verilog 早成为 IEEE 标准，这是因为 VHDL 是美国军方组织开发的，而 Verilog 则是从一个普通的民间公司的私有财产转化而来，基于 Verilog 的优越性，才成为的 IEEE 标准，因而有更强的生命力。对于初入 FPGA 的新手而言，掌握一种硬件描述语言是必要的。

§1.2 Verilog与VHDL的区别

介绍完了 Verilog 和 VHDL 这两种语言，接下来我们再来看看 Verilog 与 VHDL 有着怎样的区别。Verilog 和 VHDL 作为最流行的硬件描述语言，从设计的角度来上来说，VHDL 要优于 Verilog，因为 VHDL 最初是为描述数字硬件的行为而设计的，而 Verilog 最初是为更简捷、更有效地描述数字硬件电路和仿真设计的，所以 VHDL 更适合描述更高层次的硬件电路。从学习的角度上来说，Verilog 则要优于 VHDL，Verilog 它是一种非常容易掌握的硬件描述语言，只要有 C 语言的编程基础，通过一段时间的学习，再加上一段实际操作，一般可在二至三个月内掌握这种设计技术。而掌握 VHDL 设计技术就比较困难。这是因为 VHDL 不很直观，需要有 Ada 编程基础，一般至少需要半年以上，才能掌握 VHDL 的基本设计技术。不管怎么说，Verilog 与 VHDL 语言本身并没有什么优劣之分，而是各有所长，我们使用这两种语言都是能够完成数字电路系统的设计任务的。

对于一个长期或者想要从事 FPGA 事业的工程师来说，只懂得一种硬件描述语言显然是不够的，这是由于不同项目的平台条件、环境因素以及合作模式等的不同所必然导致的。在这里我们推荐大家入门学习语言应当首选 Verilog 语言，更进一步，更上一层时，我们再去学习 VHDL 语言。这里我们需要说明的是，在后续的文字教程或者实例代码中，我们都是以 Verilog 为主。

§1.3 Verilog与C语言的区别

说完了 Verilog 与 VHDL 的区别，接下来我们还要说一说 Verilog 与 C 语言的区别。Verilog 的设计初衷是成为一种基本语法与 C 语言相近的硬件描述语言，这是因为 C 语言在 Verilog 设计之初，已经在许多领域得到广泛应用，C 语言的许多语言要素已经被许多人习惯。一种与 C 语言相似的硬件描述语言，可以让电路设计人员更容易学习和接受。

虽然 Verilog 语言是根据 C 语言设计而来的，并且 Verilog 语言还具备 C 语言简洁易用等特点，但是，Verilog 语言和 C 语言是有着本质的区别的。这里我们就以最容易理解的并发性为例进行说明：写过 C 语言代码的都知道，C 语言代码是一句一句串行执行的，它是不可以所有的代码语句并行执行的。而我们的 Verilog 语言作为硬件描述语言，它是可以实现所有的代码语句并行执行的，可以有效地地描述并行的硬件系统。不仅如此，我们在评价硬件描述语言写的好坏的标准和 C 编程语言的标准也是完全不同的。

这里需要注意的是，如果你有 C 语言软件编程经验，那么你在学习 Verilog 的时候，一定要放弃 C 编程的一些固有思路，学会用硬件的方式去解决问题。时刻提醒自己正在设计的是一个电路，而不是一行行空洞的代码。只有建立了硬件设计思想，才有更深入学习 FPGA 的可能。

第二章

Verilog 的基础知识

第二章

Verilog 的基础知识

§ 2.1 Verilog的四值逻辑系统

首先我们先来讲一下 Verilog 的四值逻辑系统，大家看，如图 2.1 所示。

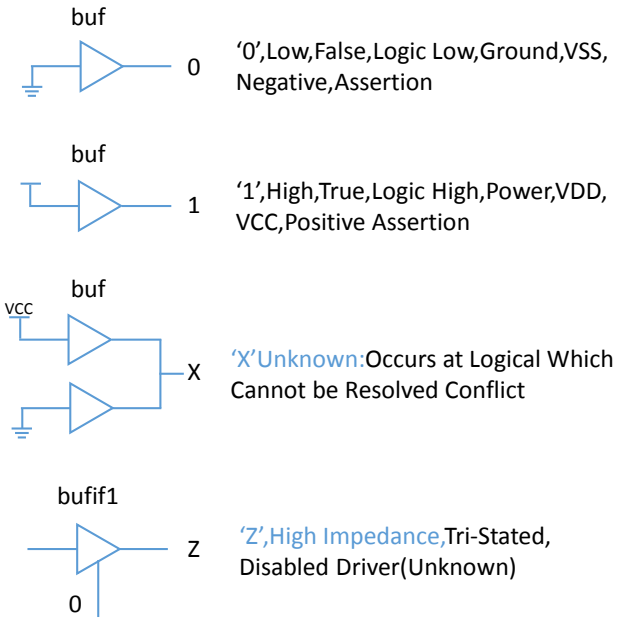


图 2.1 Verilog 的四值逻辑系统图

在 Verilog 的逻辑系统中有四种值，也即四种状态：逻辑 0：表示低电平，也就对应我们电路 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。

§ 2.2 Verilog的数据类型

在 Verilog 语言中，主要有三大类数据类型，即寄存器数据类型、线网数据类型和参数数据类型。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数据类型和线网数据类型，它们共同遵守 Verilog 的四值逻辑系统。

2.2.1 寄存器数据类型

首先我们介绍的是寄存器数据类型，所谓寄存器数据类型就是表示一个抽象的数据存储单元，它只能在 always 语句和 initial 语句等过程语句中被赋值，它的缺省值为 x。在实际的数字电路中，如果该过程语句描述的是时序逻辑，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；则该寄存器变量对应为硬件连线；如果该过程语句描述的是不完全组合逻辑，那么该寄存器变量也可以对应为锁存器。由此可见，寄存器类型的变量不一定会综合为寄存器。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 a;            //定义 a 为 1bit 的 reg 类型
reg [7:0] b;      //定义 b 为 8bit 的 reg 类型
reg [7:0] c,d,e;  //定义 c,d,e 为 8bit 的 reg 类型
```

2.2.2 线网数据类型

说完了寄存器数据类型，接下来我们再来看一下线网数据类型，所谓线网数据类型就是表示 Verilog 结构化元件间的物理连线。它的值由驱动元件的值决定，例如连续赋值与门的输出。如果没有驱动元件连接到线网，线网的缺省值为 z。线网数据类型同寄存器数据类型一样也是有很多种，如 tri、wand 和 wire 等，其中最常用的就是 wire 类型，它的使用方法如下：

```
wire a;           //定义 a 为 1bit 的 wire 类型
wire [7:0] b;     //定义 b 为 8bit 的 wire 类型
wire [7:0] c,d,e; //定义 c,d,e 为 8bit 的 wire 类型
```

2.2.3 参数数据类型

最后我们再来介绍一下参数数据类型。参数其实就是一个常量，通常出现在 module 内部，常被用于定义状态机的状态、数据位宽和延迟大小等，由于它可以再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    N = 8'd15;   //使用方法一
parameter    A = 4'b0001, //使用方法二
             B = 4'b0010,
             C = 4'b0100,
             d = 4'b1000;
```

§ 2.3 Verilog的基本运算符

介绍完了 Verilog 的数据类型，接下来我们再来看下 Verilog 的基本运算符，Verilog 硬件描述语言的运算符范围很广，其运算符按其功能可分为以下八类：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。下面我们分别进行介绍。

2.3.1 算术运算符

首先我们介绍的是算术运算符，所谓算术逻辑运算符就是我们常说的加、减、乘、除等，这类运算符的抽象层级较高，从数字逻辑电路实现上来看，它们都是基于与、或、非等基础门逻辑组合实现的，如表 2.1 所示。

表 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

2.3.2 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。如表 2.2 所示。

表 2.2 关系运算符

符号	使用举例	说明（返回结果真或者假）
>	a > b	a 大于 b
<	a < b	a 小于 b
<=	a >= b	a 大于等于 b
>=	a<= b	a 小于等于 b
==	a == b	a 等于 b
!=	a != b	a 不等于 b

2.3.3 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，如表 2.3 所示。

表 2.3 逻辑运算符

符号	使用举例	说明（返回结果真或者假）
!	!a	a 非
&&	a && b	a 与上 b
||	a || b	a 或上 b

2.3.4 条件运算符

Verilog 语言为了让连续赋值的功能更加完善，于是又从 C 语言中引入了条件操作符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，如表 2.4 所示。

表 2.4 条件运算符

符号	使用举例	说明
? :	a ? b : c	如果 a 为真，选择 b，否则选择 c

2.3.5 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。位运算符的与、或、非与逻辑运算符逻辑与、逻辑或、逻辑非，虽然它们处理的数据类型不一样，但是从硬件实现角度上来说，它们没有区别的，如表 2.5 所示。

表 2.5 位运算符

符号	使用举例	说明
~	~a	将 a 的每个位进行取反
&	a & b	将 a 的每个位与 b 相应的位进行相与
|	a | b	将 a 的每个位与 b 相应的位进行相或
^	a ^ b	将 a 的每个位与 b 相应的位进行异或
~^	a ~^ b	将 a 的每个位与 b 相应的位进行异或非
^~	a ^~ b	将 a 的每个位与 b 相应的位进行异或非

2.3.6 移位运算符

在 Verilog 中有两种移位运算符：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。如表 2.6 所示。

表 2.6 移位运算符

符号	使用举例	说明
<<	a << b	将 a 左移 b 位
>>	a >> b	将 a 右移 b 位

2.3.7 拼接运算符

在 Verilog 中有一个特殊的运算符，就是我们的位拼接运算符。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。如表 2.7 所示。

表 2.7 拼接运算符

符号	使用举例	说明
{}	{a,b}	将 a 和 b 连接起来，
{{}}	{a{b}}	将 b 重复 a 次

2.3.8 运算符的优先级别

运算符一多，必然涉及到优先级的问题，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，如表 2.8 所示。

表 2.8 运算符的优先级别

运算符	优先级
!、~	由高到低
*、/、%	
+、-	
<<、>>	
<、<=、>、>=	
==、!=、===、!==	
&	
^、^~	
|	
&&	
||	
?:	

第三章

Verilog 的基础语法

第三章

Verilog 的基础语法

讲完了 Verilog 基础知识，下面我们就来讲一下 Verilog 的基础语法。虽然 Verilog 硬件描述语言有很完整的语法结构和系统，这些语法结构的应用给设计描述带来很多方便。但是 Verilog 是描述硬件电路的，它是建立在硬件电路的基础上的。有些语法结构是不能与实际硬件电路对应起来的，比如 for 循环，它是不能映射成实际的硬件电路的，因此，Verilog 硬件描述语言分为可综合和不可综合语言。下面我们就来简单的介绍一下可综合与不可综合。

(1) 所谓可综合，就是我们编写的 Verilog 代码能够被综合器转化为相应的电路结构。因此，我们常用可综合语句来描述数字硬件电路。

(2) 所谓不可综合，就是我们编写的 Verilog 代码无法综合生成实际的电路。因此，不可综合语句一般我们在描述数字硬件电路时是用不到的，不过，我们可以用它来仿真、验证我们描述的数字硬件电路

§ 3.1 Verilog的关键字

说完了可综合和不可综合两个概念，接下来我们就继续讲解我们的 Verilog 基础语法，首先我们讲解的是 Verilog 的关键字，下面给出 Verilog 的所有关键字，如下表所示。

表 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	strentgh	strong0	strong1	supply0

supply1	table	task	tran	tranif0
tranif1	time	tri	triand	trior
trireg	tri0	tri1	vectored	wait
wand	weak0	weak1	while	wire
wor	xnor	xor		

学过 C 语言的朋友，一定会被该表给吓到，为什么这么说呢，我们知道，C 语言也才不过区区 32 个关键字，我们的 Verilog 虽然是根据 C 语言设计出来的，但是它的关键字却比 C 语言的关键字要多的多。看到了这么多的关键字，如果让你们一个一个的去学习，那么我相信大家肯定是要崩溃了。其实在这里大家并不需要为此担心，前面我们说过，Verilog 分为可综合和不可综合语句，由于我们是将 Verilog 用于描述数字硬件电路，所以我们只需要掌握可以被综合器综合的那部分关键字就可以了。下面我们列出了常用的可综合关键字，如表 3.2 所示。

表 3.2 Verilog 的常用的可综合关键字

module	endmodule	input	output	inout
wire	reg	parameter	always	assign
if	else	begin	end	case
endcase	posedge	negedge	or	default

该表格中所列出的代码是我们日后编写代码经常使用到的可综合关键字，我们只要掌握以上关键字，在语法方面可以说就已经足够了，无论多么牛的工程师，在他的代码里无非也就是上面的这些关键字。下面我们就来简单的介绍一下这些关键字：

首先我们介绍的是 module 和 endmodule，它们是成对使用的，模块是 Verilog 设计中基本功能块，一个最简单的模块是由模块命名、端口列表两个部分组成。整个模块是由 module 开头，endmodule 结尾，module 后面紧跟着的是模块名，每个模块都有它自己的名字。input、output 和 inout 用于端口定义，wire 和 reg 是用来声明数据类型，parameter 是用来声明参数类型，always 是过程赋值语句，assing 是连续赋值语句。if 和 else 成对使用，是条件判断语句，和 C 语言中的 if 和 else 是一样的功能，begin 和 end 也是成对使用，相当于 C 语言中的大括号，case、endcase 和 default 成对使用，是一个多分支条件语句，和 C 语言中的 switch 一样的功能，posedege、negedge 和 or 这三个关键字是和 always 关键字联合使用的，posedge 是上升沿触发，negedge 是下降沿触发，posedge or negedge 是既有上升沿又有下降沿。这里我们需要说明的是，上述关键字我们只是简单的介绍了一下，具体使用方法我们后续会结合实例进一步进行介绍的，这里大家只要混个面熟就可以拉。

§ 3.2 Verilog的基本程序框架

说完了 Verilog 的关键字，接下来我们再来说下 Verilog 的基本程序框架。学习 Verilog 的基本程序框架可以让我们对 Verilog 程序设计有一个整体的概念把握，进而在后续的 Verilog 语法学习中有更好的了解，就如同学习 C 语言时，首先接触到的第一个程序就是“Hello World”。既然 Verilog 是一种用于数字逻辑电路设计的语言，那么我们就以数字电路中最简单的与门为例，

来作为入门学习的第一个程序。与门的 Verilog 代码如代码 3.1 所示。

代码 3.1 与门的 Verilog 代码

```
1  module yumen
2  (
3    a, b, c
4  );
5
6  input a;
7  input b;
8  output c;
9
10 assign c = a & b;
11
12 endmodule
```

下面我们就来简单的介绍一下这个代码，在这个代码中，a 和 b 是与门的输入，c 是与门的输出，也就是说，该代码实现了一个 2 输入的与门电。这里我们需要说明的是，大家不必过分去苛求细节的语法，只要着眼于基本程序框架就行。通过上面的程序我们给大家总结如下：

- Verilog HDL 程序是由模块构成的，每个模块的内容都是嵌在 module 和 endmodule 两个语句之间。
- 每个模块要进行端口定义，并说明输入输出口，然后对模块的功能进行行为逻辑描述。
- Verilog HDL 程序的书写格式自由，一行可以写几个语句，一个语句也可以分写多行。
- 除了 endmodule 语句外，每个语句和数据定义的最后必须有分号。

通过上面与门的 Verilog 代码，想必大家已经初步了解了 Verilog 基本程序框架，下面我们就给出一个较为完整的 Verilog 语法结构，我们就以《项目实战篇》中的动态数码管显示为例进行讲解，如代码 3.2 所示。

代码 3.2 一般的 Verilog 语法结构

```
1  module Example_Segled
2  (
3    //输入端口
4    CLK_50M,RST_N,
5    //输出端口
6    SEG_DATA,SEG_EN
7  );
8
9  input                CLK_50M;          //时钟的端口,开发板用的 50M 晶振
10 input                RST_N;            //复位的端口,低电平复位
11 output reg   [ 5:0]  SEG_EN;           //数码管使能端口
12 output reg   [ 7:0]  SEG_DATA;         //数码管数据端口(查看管脚分配文档或者原理图)
13
14 reg          [15:0]  time_cnt;         //用来控制数码管闪烁频率的定时计数器
```

```
15  reg           [15:0]   time_cnt_n;       //time_cnt 的下一个状态
16  reg           [ 2:0]   led_cnt;          //用来控制数码管亮灭及显示数据的显示计数器
17  reg           [ 2:0]   led_cnt_n;        //led_cnt 的下一个状态
18
19  //设置定时器的时间为 1ms,计算方法为  (1*10^3)us / (1/50)us  50MHz 为开发板晶振
20  parameter SET_TIME_1MS = 16'd50_000;
21
22  //时序电路,用来给 time_cnt 寄存器赋值
23  always @ (posedge CLK_50M or negedge RST_N)
24  begin
25     if(!RST_N)                           //判断复位
26        time_cnt <= 16'h0;                //初始化 time_cnt 值
27     else
28        time_cnt <= time_cnt_n;           //用来给 time_cnt 赋值
29  end
30
31  //组合电路,实现 10ms 的定时计数器
32  always @ (*)
33  begin
34     if(time_cnt == SET_TIME_1MS)         //判断 10ms 时间
35        time_cnt_n = 16'h0;               //如果到达 10ms,定时计数器将会被清零
36     else
37        time_cnt_n = time_cnt + 27'h1;    //如果未到 10ms,定时计数器将会继续累加
38  end
39
40  //时序电路,用来给 led_cnt 寄存器赋值
41  always @ (posedge CLK_50M or negedge RST_N)
42  begin
43     if(!RST_N)                           //判断复位
44        led_cnt <= 3'h0;                  //初始化 led_cnt 值
45     else
46        led_cnt <= led_cnt_n;             //用来给 led_cnt 赋值
47  end
48
49  //组合电路,判断时间，实现控制显示计数器累加
50  always @ (*)
51  begin
52     if(time_cnt == SET_TIME_1MS)         //判断 10ms 时间
53        led_cnt_n = led_cnt + 1'h1;       //如果到达 10ms,计数器进行累加
54     else
55        led_cnt_n = led_cnt;              //如果未到 10ms,计数器保持不变
56  end
57
58  //组合电路,实现数码管的数字显示
```

```
59  always @ (*)
60  begin
61     case (led_cnt)
62        3'b000 : SEG_DATA = 8'b10111111;  //当计数器为 0 时,数码管将会显示 "0"
63        3'b001 : SEG_DATA = 8'b10000110;  //当计数器为 1 时,数码管将会显示 "1"
64        3'b010 : SEG_DATA = 8'b11011011;  //当计数器为 2 时,数码管将会显示 "2"
65        3'b011 : SEG_DATA = 8'b11001111;  //当计数器为 3 时,数码管将会显示 "3"
66        3'b100 : SEG_DATA = 8'b11100110;  //当计数器为 4 时,数码管将会显示 "4"
67        3'b101 : SEG_DATA = 8'b11101101;  //当计数器为 5 时,数码管将会显示 "5"
68        default: SEG_DATA = 8'b10111111;
69     endcase
70  end
71
72  //组合电路,控制数码管亮灭
73  always @ (*)
74  begin
75     case (led_cnt)
76        3'b000 : SEG_EN = 6'b111110;      //当计数器为 0 时,数码管 SEG1 显示
77        3'b001 : SEG_EN = 6'b111101;      //当计数器为 1 时,数码管 SEG2 显示
78        3'b010 : SEG_EN = 6'b111011;      //当计数器为 2 时,数码管 SEG3 显示
79        3'b011 : SEG_EN = 6'b110111;      //当计数器为 3 时,数码管 SEG4 显示
80        3'b100 : SEG_EN = 6'b101111;      //当计数器为 4 时,数码管 SEG5 显示
81        3'b101 : SEG_EN = 6'b011111;      //当计数器为 5 时,数码管 SEG6 显示
82        default: SEG_EN = 6'b111111;
83     endcase
84  end
85  endmodule
```

从代码中我们可以看出，该代码几乎包含了我们所有的可综合关键字，这个代码主要实现的功能是让我们的 A4 开发板上的六个数码管分别显示 012345。这里需要我们注意的是，我们应该把注意力放在关键字的使用上，而不是把注意力放在代码实现的功能上。比如，我们 always 关键字不会使用，那么我们就可以通过上面的代码，知道 alywas 关键字的两种使用方法。当然，其他的关键字也是可以依葫芦画瓢运用到其他的代码中。至此，我们的第三章 Verilog 的基础语法就讲解完了。

第四章

Verilog 的关键问题解惑

第四章

Verilog 的关键问题解惑

§ 4.1 Verilog的抽象级别

所谓抽象级别，实际上是指同一个物理电路，可以在不同的层次上用 Verilog 语言来描述它。Verilog 硬件描述语言支持以下五种级别：

（1）系统级；
（2）算法级；
（3）RTL 级；
（4）门级；
（5）开关级；

其中，系统级和算法级是属于行为级描述方式的， RTL 级又称为数据流描述方式，门级和开关级是属于结构化描述方式的，下面我们就对这三种方式分别进行介绍。

4.1.1 结构化描述方式

首先我们讲解是结构化描述方式，结构化描述方式是最原始的描述方式，也是抽象级别最低的描述方式，不过，它却是最接近于实际的硬件结构的描述方式。因为采用结构化的描述方式来编写 Verilog 代码，其思路就跟在面包板上搭建数字电路时一样的，唯一的不同点就是我们是通过 Verilog 的形式来描述数字电路都需要哪些元器件以及它们之间的连接关系是怎么样的罢了。为了能够让读者更直观、更容易的理解结构化描述方式，这里我们就以《数字电路篇》中的三人表决器为例，然后结合我们所给出的三人表决器的结构描述方式代码进行详细的讲解与说明。结构化描述方式代码，如代码 4.1 所示。

代码 4.1 结构化描述方式代码

```
1  module Example_Structure //Example_Structure,即模块的开始,
2  (
3    //输入端口
4    A,B,C,
5    //输出端口
6    L
7  );
8
9  input   A;              //模块的输入端口 A
10 input   B;              //模块的输入端口 B
11 input   C;              //模块的输入端口 C
12 output  L;              //模块的输出端口 L
13
14 wire AB,BC,AC;          //内部信号声明 AB,BC,AC
15
```

```
16  and U1(AB,A,B);        //与门（A,B 信号进入）（A 与 B 信号即 AB 输出）
17  and U2(BC,B,C);        //与门 同上
18  and U3(AC,A,C);        //与门 同上
19
20  or  U4(L,AB,BC,AC);    //或门 同上
21
22  endmodule              //模块的结束
```

从代码中我们可以看出，该代码主要是描述的三人表决器的逻辑电路图。

4.1.2 数据流描述方式

数据流描述方式要比结构化描述方式的抽象级别高一些，因为它不再需要清晰地刻画出具体的数字电路架构，而是可以比较直观地表达底层逻辑的行为。基于数据流的描述方式，形象点来说，每个模块就好比一个容器，大量外部信息从模块的输入端口流入，相应的，大量的处理后信息也会从模块的输出端口流出，因此，基于这种思路编写的 Verilog 代码被称为数据流描述方式。数据流描述方式又可称为 RTL 级描述方式，即寄存器传输级描述。下面我们同样也是以三人表决器为例，并给出三人表决器的数据流描述方式代码，如代码 4.2 所示：

代码 4.2 数据流描述方式代码

```
1   module Example_Dataflow //Example_Dataflow,即模块的开始
2   (
3     //输出端口
4     A,B,C,
5     //输入端口
6     L
7   );
8
9   input   A;          //模块的输入端口 A
10  input   B;          //模块的输入端口 B
11  input   C;          //模块的输入端口 C
12  output  L;          //模块的输出端口 L
13
14  assign L = ((!A) & B & C) | (A & (!B) & C) | (A & B & (!C)) | (A & B & C);
15
16  endmodule           //模块的结束
```

从代码中我们可以看出，该代码主要是描述的三人表决器的逻辑表达式。

4.1.3 行为级描述方式

和前面两种描述方式比起来，行为级描述方式的抽象级别最高，概括力也最强，因此规模稍大些的设计，往往都是以行为级描述方式为主。大家都知道高级语言的执行思路都是串行的，例如 C 语言。顺序执行的语句更容易帮助我们来表达我们的设计思想，尤其是使描述时序逻辑变得容易。虽然 FPGA 的设计思路都是并行的，但是 Verilog 中还是支持大量的串行语句元素。由

此可见，行为级描述方式的主要载体就是串行语句，同时辅以并行语句用于描述各个算法之间的连接关系。下面我们同样也是以三人表决器为例，并给出三人表决器的行为级描述方式代码，代码如代码 4.3 所示。

代码 4.3 行为级描述方式代码

```
1  module Example_Behavior    //Example_Behavior,即模块的开始
2  (
3    //输入端口
4    A,B,C,
5    //输出端口
6    L
7  );
8
9  input          A;          //模块的输入端口 A
10 input          B;          //模块的输入端口 B
11 input          C;          //模块的输入端口 C
12 output reg      L;         //模块的输出端口 L
13
14 always @ (A,C,B)           //always 在组合逻辑中的用法
15 begin                      //always @ (A,B,C)解析：只要 A,B,C
16    case({A,B,C})           //其中有一个信号有变化便会执行 begin 中的 case 语句
17       3'b000: L = 1'b0;    //也可以写成 always @ (*)，与 always @ (A,B,C)功能相同
18       3'b001: L = 1'b0;    //{A,B,C}解析：把 A,B,C 三条线合成一条总线
19       3'b010: L = 1'b0;    //举例说明：{1'b1,1'b0}=2'b10
20       3'b011: L = 1'b1;    //
21       3'b100: L = 1'b0;
22       3'b101: L = 1'b1;
23       3'b110: L = 1'b1;
24       3'b111: L = 1'b1;
25       default:L = 1'bx;    //不要省略
26    endcase                 //case 语句的结束
27 end                        //begin 语句的结束
28
29 endmodule                  //module 语句的结束
```

从代码中我们可以看出，该代码主要是描述的三人表决器的真值表。

§ 4.2 Verilog的模块化设计

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和仿真测试更加容易，使代码维护和升级更加便利。所谓模块化设计，就是将一个比较复杂的系统按照一定的规则划分为多个小模块，然后我们再分别对每个小模块进行设计，当这些小模块全都完成以后，我们再将这些小模块有机的组合起来，最终我们就能够完成整个复杂系统的设计。这里我们就以半加器为例进行说明，如图 4.1 所示。

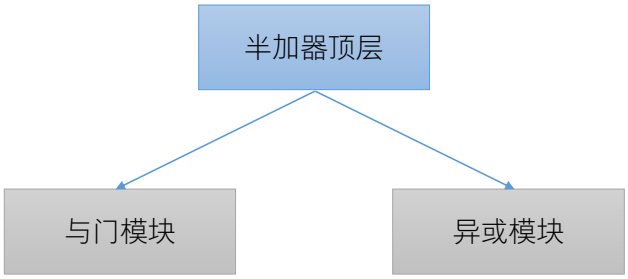


图 4.1 半加器的模块化设计

从该图中我们可以看出，我们将半加器分成了与门模块和异或模块，我们只需要完成实现与门模块和异或模块，然后我们再将实现的与门模块和异或模块相结合，最终我们就能实现半加器。下面我们给出与该图相对应的 Verilog 代码，首先我们给出的是半加器顶层模块代码，如代码 4.4 所示。

代码 4.4 半加器顶层模块代码

```
1  module Example_Module
2  (
3     input  a,
4     input  b,
5     output s,
6     output c
7  );
8
9  Example_yumen yumen_module
10 (
11    .yumen_a(a),
12    .yumen_b(b),
13    .yumen_c(c)
14 );
15
16 Example_yihuo yihuo_module
17 (
18    .yihuo_a(a),
19    .yihuo_b(b),
20    .yihuo_s(s)
21 );
22
23 endmodule
```

从该代码中我们可以看出，第 1 行是模块的开始，模块名为 Example_Module。第 2 至 7 行是端口声明，我们定义了 2 个输入端口 a 和 b，2 个输出端口 s 和 c。其中 a 和 b 代表两个加数，s 代表两个加数的和，c 代表进位。第 9 行是例化与门模块，其中 Example_yumen 是 Example_yumen.v 模块里相对应的模块名，yumen_module 可以任意命名，它主要是用来区分例化多个相同的模块。第 10 至 14 行是信号的例化，其中.yumen_a 是与门模块中的信号，它必

须和与门模块中的信号名一致才行，(a) 是顶层模块中的信号，它必须和顶层模块中的信号名一致才行。第 16 至 21 行是例化异或模块，与例化与门同理。看完了顶层模块，下面我们再来看下与门模块的代码，如代码 4.5 所示。

代码 4.5 与门模块代码

```
1  module Example_yumen
2  (
3     input  yumen_a,
4     input  yumen_b,
5     output yumen_c
6  );
7
8  assign yumen_c = yumen_a && yumen_b;
9
10 endmodule
```

从该代码中我们可以看出，该代码很简单，主要实现了一个 2 输入的与门功能。下面我们再来看下异或模块的代码，如代码 4.6 所示。

代码 4.6 异或模块代码

```
1  module Example_yihuo
2  (
3     input  yihuo_a,
4     input  yihuo_b,
5     output yihuo_s
6  );
7
8  assign yihuo_s = yihuo_a ^ yihuo_b;
9
10 endmodule
```

从该代码中我们可以看出，它和我们的与门模块基本上差不多。主要实现了一个 2 输入的异或门功能。由于我们给出的实例比较简单，即使我们将与门模块中的代码和异或模块中的代码都写在顶层模块中，我们也是感觉不到复杂的，我们这里主要是想让大家从该实例中能够更好的理解、明白模块化设计的概念及思想。最后我们再来看一下我们半加器的 RTL 视图，通过观察 RTL 视图，我们能够更深刻地感受到模块化带来的层次感，如图 4.2 所示。

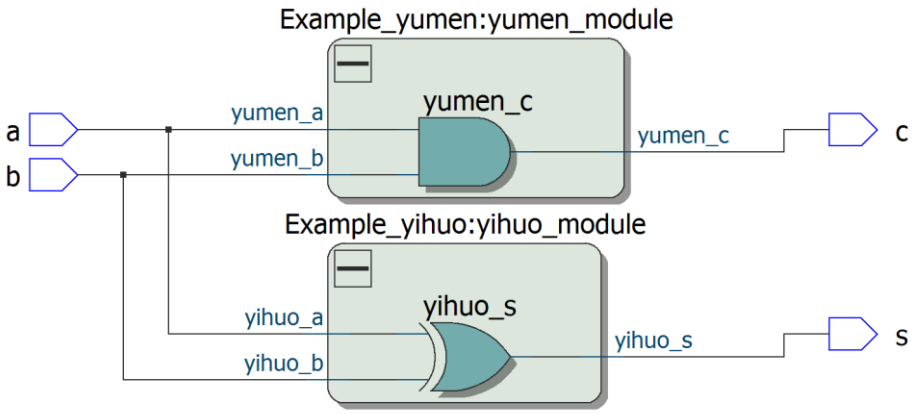


图 4.2 半加器的 RTL 视图

通过该图我们可以看到，我们的半加器模块不仅层次清晰，而且思路明确，代码写起来也是游刃有余。

§ 4.3 如何给端口选择正确的数据类型?

Verilog 中让人困惑的地方就是 reg 和 wire 的使用，虽然声明 reg 和 wire 的规则很简单，但是很多新手总是难以理解输入和输出端口到底是使用 reg 类型还是 wire 类型。因此，我们针对此情况进行一个详细讲解。大家看这是一个我们综合出的实际电路模块，如图 4.3 所示。

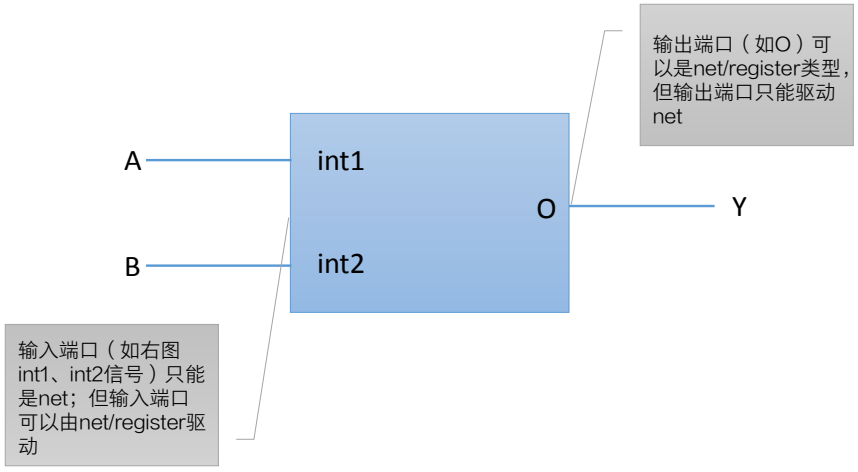


图 4.3 实际电路模块图

从图中我们可以看出，对于输入 A 和输入 B 这两个端口来说，我们只能使用线网类型，但是，输入 A 和输入 B 这两个端口可以由寄存器和线网所驱动。也就是说，寄存器和线网可以连接到这两个输入端口上作为输入源。对于输出端口 Y 来说，它可以是线网，也可以是寄存器类型，但是，对于输出端口 Y，它只能驱动线网类型。这样说比较抽象，下面我们结合具体实例进行说明，代码如代码 4.7 所示。

代码 4.7 数据类型模块代码

```
1  module Example_Datatype
2  (
3    a,b,c,d,o1,o2
```

```
4   );
5
6   input a,b,c,d;
7   output o1,o2;
8
9   reg c,d;
10  reg o1;
11  reg o2;
12
13  assign o2 = c && d;
14
15  always @ (a or b)
16  begin
17     if(a)
18        o1 = b;
19     else
20        o1 = 1'b0;
21  end
22
23  endmodule
```

首先我们先来简单的介绍一下该代码的，第 1 至 11 行是端口数据类型声明， 第 13 行使用 assign 连续赋值语句实现了一个 2 输入的与门电路。第 15 至 21 行使用 always 过程赋值语句实现了一个二选一数据选择器，通过端口 a 来选择 o1 是输出 b 还是输出 0。介绍完了该代码，下面我们再来看下该代码中的错误，如果你将该代码放入 Quartrus II 软件进行编译，那么 Quartus II 软件将会弹出如图 4.4 所示错误页面。

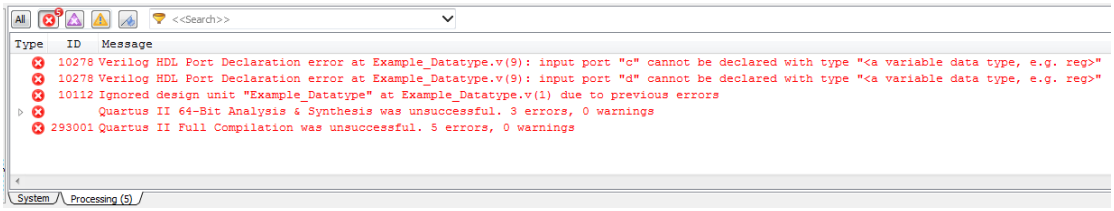


图 4.4 Quartus II 软件编译错误提示窗口

从该图中我们可以看出，c 不能被声明 reg 类型，也就是这个 variable data type，这里为什么会是 varibale，这是因为，在最新的 2001verilog 标准中，已经没有寄存器(register)类型，它们把 register 改成了 variable 变量类型，但是一般来说，我们还是比较习惯称为寄存器类型。为什么 c 和 d 不能被声明为寄存器类型？这是因为 c 和 d 它们是输入端口，前面我们已经说过了，输入端口，它不能被定义成寄存器类型，它只能被定义为 wire 类型。这里我们进行代码修改，将 reg c 和 reg d 注释掉就可以了，默认的类型就是 wire 类型。修改完成后，我们再通过 Quartus II 软件进行编译，我们可以看到，代码仍然是错的，错误提示信息如图 4.5 示。

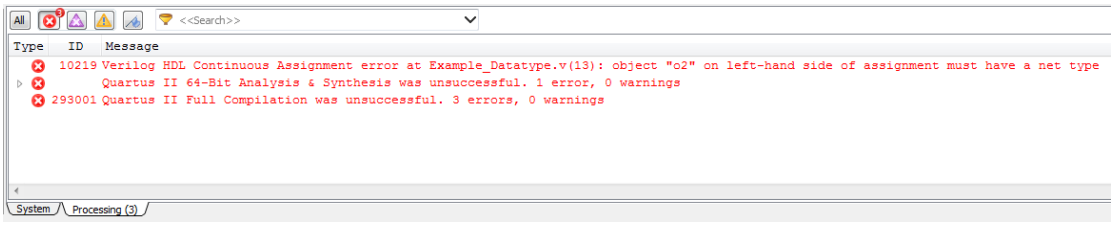


图 4.5 Quartus II 软件编译错误提示窗口

从该图中我们可以看出，o2 必须为线网类型，也许有的朋友会说，我们前面说对于输出端口来说，它可以是线网，也可以是寄存器类型，为什么这里的输出端口 o2 必须为线网类型不能为寄存器类型呢？这是因为 o2 是由关键字 assign 指定的，我们前面说过对于 assign 这种连续赋值语句，必须是 wire 类型，这一点大家一定要记清楚。对于 o1 它必须是定义为 reg 类型的，为什么呢？这是因为它是在 always 里面赋值的，所以必须为 reg 类型。我们将 o2 给注释掉，然后再一次通过 Quartus II 软件进行编译，这时，我们可以看到代码没有出现错误，通过了编译。至此，相信大家已经知道了如何给端口选择正确的数据类型。最后我们在补充说明一点，大家看如图 4.6 所示。

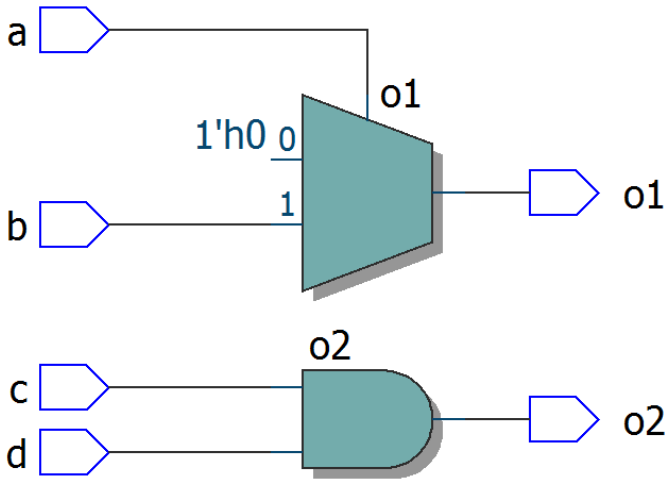


图 4.6 数据类型的 RTL 视图

从该图中我们可以看出，尽管我们将 o1 声明成了 reg 类型，但是它并没有综合成触发器，这是因为，对于寄存器 reg 类型，在 always 等过程块中被赋值的信号，如果该 always 模块描述的是时序逻辑电路，那么该信号常常被综合为 D 触发器，如果该 always 模块描述的是组合逻辑电路，那么该信号会被综合成连线。

§ 4.4 latch的产生

当你在 always 过程块中描述组合逻辑电路时，如果你的条件语句没有说明全部的条件，那么就会有可能产生锁存器，也就是我们所说的 latch。下面我们结合具体实例进行说明，代码如代码 4.8 所示。

代码 4.8 产生 latch 的电路模块代码

```
1  module Example_Latch
2  (
3    clk,q,data,enable
4  );
5
6  input    clk;
7  input    data;
8  input    enable;
9  output   q;
10
11 reg      q;
12
13 //always @ (posedge clk)
14 //begin
15 // if(enable)
16 //    q <= data;
17 // else
18 //    q <= q;
19 //end
20
21 always @ (*)
22 begin
23   if(enable)
24     q = data;
25 // else
26 //    q = 1'b0;
27 end
28
29 endmodule
```

我们可以看到，这就是一个典型的产生锁存器的电路。我们先来简单的介绍一下该代码，第 1 至 11 行是用来端口声明的，第 21 至 27 行是一个 always 模块，该 always 模块是一个组合电路，当 enable 为 1 时才会将 data 的值赋值给 q。由于 q 是在 always 中进行赋值的，所以 q 为 reg 类型。介绍完了代码，下面我们就来看一下 RTL 视图，如图 4.7 所示。

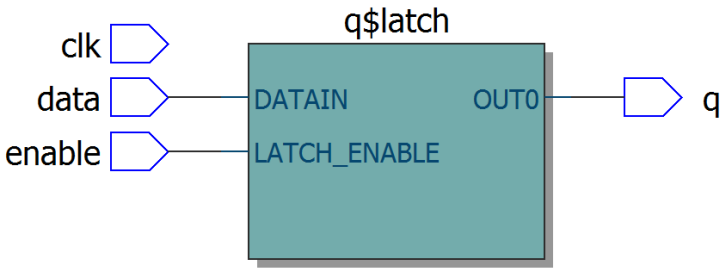


图 4.7 产生 latch 的 RTL 视图

从该图中我们可以看到，该电路最终生成了 latch。也许有的朋友会问，为什么会产生锁存器？这是因为当 enable 为 1 的时候，它把 data 端输入到 q，但是当 enable 为 0 的时候，我们没有执行任何程序，电路默认情况下也就保持原值不变，因此，就会产生一个锁存器，下面我们将第 25 和 26 行的代码注释给去掉，然后我们再一次查看 RTL 视图，我们可以看到如图 4.8 所示。

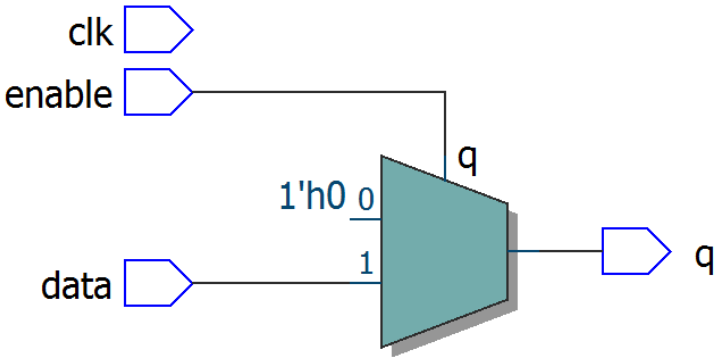


图 4.8 二选一数据选择器 RTL 视图

从该图中我们可以看出，当我们将条件语句补全以后，我们的电路不再生成 latch，而是生成了一个二选一的数据选择器。因此，我们建议大家在写组合逻辑的时候一定要注意，对于 if-else 语句尽量把它的每种条件都考虑进去，要把条件都写全，这样就会避免 latch 的产生。

最后我们再来补充说明一点，当你在 always 过程块中描述时序逻辑电路时，即使你的条件语句没有说明全部的条件，该电路也不会产生锁存器。我们在上面代码的基础上将用于描述组合逻辑电路的代码给注释掉，然后我们将已经准备好用于时序逻辑电路的代码注释给去掉，然后我们通过 Quartus II 软件进行编译并查看 RTL 视图，如图 4.9 所示。

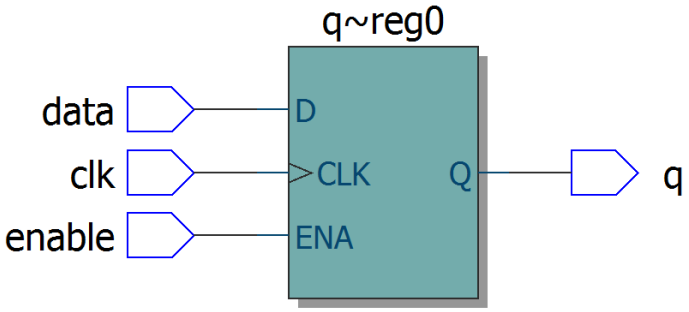


图 4.9 D 触发器的 RTL 视图

从该图中我们可以看出，我们的电路没有综合成 latch，而是综合成了 D 触发器。对于时序逻辑电路来说，这个 else 有和没有是完全一样的，因为对于 D 触发器来说，它默认情况下就是保持原值的，而 enable 这个端口相当于一个使能端口。q 是保持原值，还是打入新值，是由使能端口 enable 决定的。所以 Verilog 语言中 latch 的产生一般是组合电路中条件不完整造成的。

§ 4.5 组合逻辑反馈环

组合逻辑反馈环路是数字同步逻辑设计的大忌，它最容易因振荡、毛刺、时序违规等问题引

起整个系统的不稳定和不可靠。下面我们结合具体实例进行说明，如代码 4.9 所示。

代码 4.9 组合逻辑反馈环电路模块代码

```
1  module Example_Feedback
2  (
3    data_in1,data_in2,data_out
4  );
5
6  input    data_in1;
7  input    data_in2;
8  output   data_out;
9
10 assign data_out = (data_in2) ? data_in1 : (~data_out | data_in1);
11
12 endmodule
```

我们可以看到，这就是一个典型的组合逻辑反馈环电路。我们先来简单的介绍一下该代码，第 1 至 8 行是用来端口声明的，第 10 行是一个 assign 连续赋值语句，当 data_in 为 1 时，我们就将 data_in1 赋值给 data_out，当 data_in 为 0 时，我们便将(~data_out | data_in1)赋值给我们的 data_out。由于 data_out 是由 assign 关键字指定的，所以 data_out 为 wire 类型。介绍完了代码，下面我们就来看下它的 RTL 视图，如图 4.10 所示。

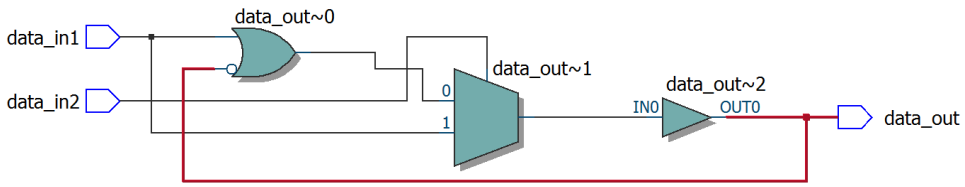


图 4.10 组合逻辑反馈环电路的 RTL 视图

从该图中我们可以看出，我们的输出端口 data_out 直接通过组合逻辑反馈到与门的输入端口上了，此时，我们假设 data_in1 和 data_in2 都为 1，那么毋容置疑，电路会输出 1。如果 data_in1 和 data_in2 都为 0，这时候，我们就需要判断 data_out 这个反馈信号，如果此时 data_out 为 1，输出就为 0，但是如果此时 data_out 为 0，那么输出就为 1，因此我们可以看出，它没有一个稳定的状态，电路存在不确定性。如果我们将此代码放入到 Quartus II 软件中进行编译，那么我们的 Quartus II 软件将会给出警告，如图 4.11 所示。

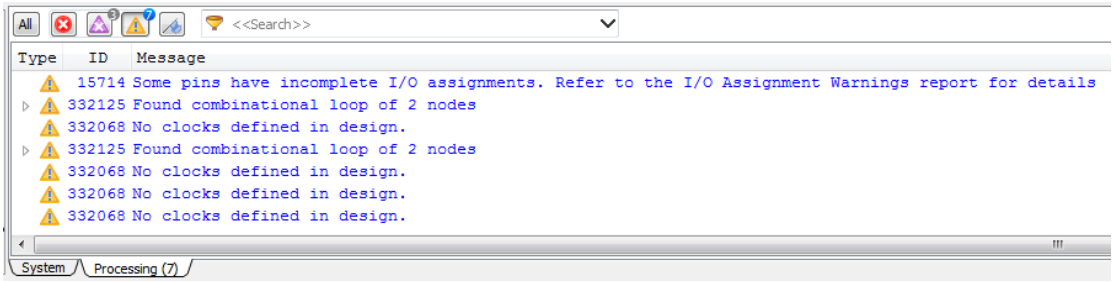


图 4.11 Quartus II 软件编译警告提示窗口

从该图中我们可以看到，Quartus II 软件提示我们说 Found combinational loop of 2 nodes。当然这种警告不是致命错误，它不影响我们编译。通过组合逻辑电路反馈环来实现这种功能的话是不允许这样做的，如果我们确实需要这样做的话，那么我们可以通过时序电路同步反馈来实现。下面我们将上述的组合逻辑电路改为时序逻辑电路，如代码 4.10 所示。

代码 4.10 避免组合逻辑反馈环电路模块代码

```
1  module Example_Feedback
2  (
3    data_in1,data_in2,data_out,clk,rst_n
4  );
5
6  input    clk;
7  input    rst_n;
8  input    data_in1;
9  input    data_in2;
10 output   data_out;
11
12 reg     data_out_r;
13
14 always @ (posedge clk or negedge rst_n)
15 begin
16   if(!rst_n)
17     data_out_r <= 1'b0;
18   else
19     data_out_r <= (data_in2) ? (data_in1) : (~data_out_r | data_in1);
20 end
21
22 assign data_out = data_out_r;
23
24 endmodule
```

由于该代码与我们前面的代码功能是一样的，所以我们这里就不再进行介绍了，这里我们需要说明的是，由于 data_out_r 是在 always 模块中赋值的，所以我们需要将 data_out_r 定义成 reg 类型，由于 data_out 是由 assign 指定的，所以我们需要将 data_out 定义成 wire 类型。如果将该代码放到 Quartus II 软件中进行编译，那么我们在警告窗口中是看不到 combinational loop 警告信息的。

§ 4.6 阻塞赋值与非阻塞赋值的不同

阻塞赋值和非阻塞赋值可以说是 Verilog 语言中最难理解概念之一。甚至有些很有经验的 Verilog 设计工程师也不能完全搞明白，究竟什么时候使用阻塞赋值什么时候使用非阻塞赋值。为此，我们也同样针对该问题尽可能地把阻塞赋值和非阻塞赋值的含义详细地解释清楚。在开始讲解阻塞赋值与非阻塞赋值的不同之前，我们先来说明一下，阻塞赋值我们使用等号（=）来表示，非阻塞赋值我们使用的是小于等于号（<=）来表示。我们知道了阻塞赋值和非阻塞赋值符号

以后，接下来我们就结合具体实例进行说明，如代码 4.11 所示。

代码 4.11 阻塞赋值与非阻塞赋值的顶层模块代码

```
1  module Example_Block
2  (
3    clk,block_in,block_out1,block_out2,no_block_out1,no_block_out2
4  );
5
6  input    clk;
7  input    block_in;
8  output   block_out1;
9  output   block_out2;
10 output   no_block_out1;
11 output   no_block_out2;
12
13 block           block_init
14 (
15   .clk           (clk         ),
16   .block_in      (block_in    ),
17   .block_out1    (block_out1  ),
18   .block_out2    (block_out2  )
19 );
20
21 no_block         no_block_init
22 (
23   .clk           (clk             ),
24   .no_block_in   (block_in        ),
25   .no_block_out1 (no_block_out1   ),
26   .no_block_out2 (no_block_out2   )
27 );
28
29 endmodule
```

从该代码中我们可以看到，我们的顶层模块 Example_Block 中包含了两个模块，一个模块是 block 也就是阻塞赋值，另一个模块是 no_block 也就是我们的非阻塞赋值，这个代码很简单，没有任何逻辑设计。下面我们再来看下 block 模块，如代码 4.12 所示。

代码 4.12 阻塞赋值模块代码

```
1  module block
2  (
3    clk,block_in,block_out1,block_out2
4  );
5
6  input    clk;
7  input    block_in;
```

```
8   output   block_out1;
9   output   block_out2;
10
11  reg     block_out1;
12  reg     block_out2;
13
14  always @ (posedge clk)
15  begin
16     block_out1 = block_in;
17     block_out2 = block_out1;
18  end
19
20  endmodule
```

在该代码中，我们可以看到，第 1 至 12 行是端口的声明，我们声明了 4 个端口，其中 2 个输入端口，2 个输出端口；第 14 至 18 行是一个 always 模块，在 always 模块中，我们先将输入端口 block_in 赋值给 block_out1，然后我们再将 block_out1 赋值给 block_out2。，这里我们需要注意的是，我们使用的是阻塞赋值（ = ）。看完了阻塞赋值模块，下面我们再来看下非阻塞赋值模块，如代码 4.13 所示。

代码 4.13 非阻塞赋值模块代码

```
1   module no_block
2   (
3     clk,no_block_in,no_block_out1,no_block_out2
4   );
5
6   input    clk;
7   input    no_block_in;
8   output   no_block_out1;
9   output   no_block_out2;
10
11  reg     no_block_out1;
12  reg     no_block_out2;
13
14  always @ (posedge clk)
15  begin
16     no_block_out1 <= no_block_in;
17     no_block_out2 <= no_block_out1;
18  end
19
20  endmodule
```

在该代码中，我们可以看到它和我们的阻塞模块代码基本上差不多，唯一不同的是，它在 always 模块赋值中使用的是非阻塞赋值（ <= ）。通过这两个代码的对比，尽管我们发现这两个代

码基本上一样，但是它们两个实现的功能却是完全不同。下面我们就来打开该模块的 RTL 视图来看下它们究竟有着怎样的不同。如图 4.12 所示。

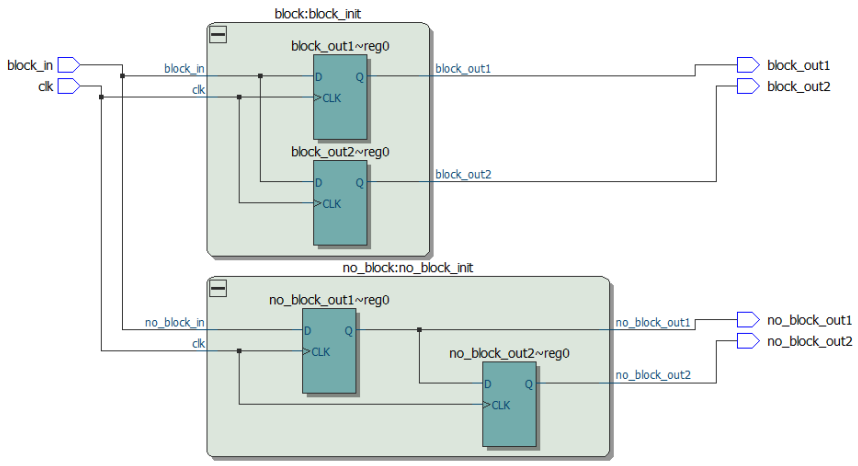


图 4.12 阻塞与非阻塞赋值模块的 RTL 视图

在 RTL Viewer 中我们可以很清楚的看到，使用阻塞赋值我们生成的电路是两个并联的 D 触发器，使用非阻塞赋值我们生成的电路是两个串联的 D 触发器，由此可以看到它们生成的电路是不一样的，我们可以想象出它们的产生的波形也是截然不同的，下面我们给出该模块的仿真波形图，如图 4.13 所示。

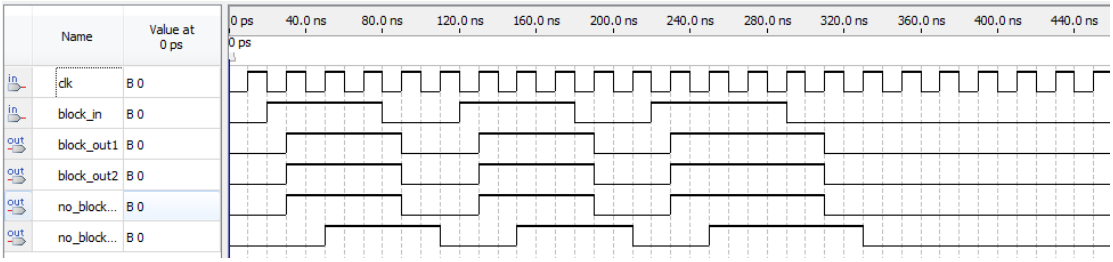


图 4.13 阻塞与非阻塞赋值模块的仿真波形图

从该仿真波形图中我们可以看出，block_out1、block_out2 和 no_block_out1，它们三个的波形是一致的，但是 no_block_out2 波形确实大大的不同。为什么会出现这种情况呢，下面我们就来详细说明：

(1) 阻塞赋值（=）是顺序执行的，也就是说，写在前面的语句先执行，写在后面的语句后执行，一般用于组合逻辑电路。

(2) 非阻塞赋值（<=）是并行执行的，也就是说，写在前面的语句与后面的语句同时执行，跟书写顺序没有关系，一般用于时序逻辑电路。

当我们带着这个想法再一次观看代码与波形的时候，我们可以发现确实是这个样子的。

§ 4.7 FPGA的灵魂，状态机

通过前面的学习，我们知道在 Verilog 中程序都是并行执行的，但是在一些在实际的工程应用中往往需要实现一些具有一定顺序的工作，那么我们应该怎么去实现呢？这就需要用到状态

机来解决上述问题。那么问题来了，什么是状态机？所谓状态机就是能够根据控制信号按照预先设定的状态进行状态转移，是协调相关信号动作、完成特定操作的控制中心。通过状态机这种方法，可降低抽象难度，同时也可提高代码的可读性及维护性。状态机简写为 FSM（Finite State Machine），主要分为 2 大类：

(1) 输出只和状态有关而与输入无关，则称为摩尔（Moore）状态机；
(2) 输出不仅和状态有关而且和输入有关系，则称为米里（Mealy）状态机；

状态机通常包括组合逻辑和时序逻辑，其中，时序逻辑由一组触发器组成，用来记忆当前的状态；而组合逻辑又可以分为次态逻辑和输出逻辑两部分，次态逻辑的功能是确定有限状态机的下一个状态，输出逻辑的功能是确定有限状态机的输出。

4.7.1 状态机的设计步骤

讲解完了什么是状态机，下面我们再来说一说如何设计一个状态机，通常设计一个状态机分为以下四个步骤：

(1) 根据实际使用需要来选择状态机的结构，确定采用摩尔型状态机还是米里型状态机。
(2) 根据实际情况分析设计要求并列出状态机的所有状态，然后对每个状态进行状态编码
(3) 根据状态转移关系和输出函数画出状态图。这里我们需要注意的是，对同一个设计问题来说，不同的人可能会构造出不同的状态图。状态图直观地反映了状态机各个状态之间的转换关系以及转换条件，因而有利于理解状态机的工作原理，但此时要求设计的状态个数不能太多对于状态个数较多的状态机一般采用状态表的方法列出状态机的转移条件。
(4) 根据所花的状态图，采用硬件描述语言对状态机进行描述。

4.7.2 状态机的状态编码

说完了状态机的设计步骤，接下来我们再来说一说状态机的状态编码。所谓状态编码就是通过不同的编码值去区分各个不同的状态，使得每一个状态都有唯一的标识。在使用 Verilog 描述状态机时，通常用参数定义语句 parameter 指定状态编码。常用的状态编码有三种分别是：递增二进制编码，格雷编码和 one-hot 编码，如表 4.1 所示。

表 4.1 状态机的三种编码方案

递进二进制编码	格雷编码	one-hot 编码
00	00	0001
01	01	0010
10	11	0100
11	10	1000

通过该表我们可以看出：

(1) 递增二进制编码：这种编码方式的效率很高，但是在译码过程中需要让所有的二进制位

参与译码，在状态较多且状态跳转条件比较复杂时会导致很大的组合逻辑。

(2) 格雷编码：这种编码方式能够避免进入错误的状态，常用于高可靠性设计。

(3) one-hot 编码:这种编码方式所占用的D触发器资源通常要比递增二进制编码多一些，状态数等于触发器的数据，冗余的触发器使得译码电路比较简单，因此它的速度非常快。

这里需要注意的是，不管使用哪种编码方式，状态机中的各个状态都应使用符号常量，而不应该直接使用编码数值，赋予各状态有意义的名字对与设计的验证和代码的可读性都是有益的。

4.7.3 状态机的描述方法

最后我们再来看下状态机的描述方法,状态机的描述代码风格有三种:分别是一段式状态机、二段式状态机、三段式状态机。对于这三种方法下面我们就来分别进行介绍。

(1) 一段式状态机：是将所有逻辑都写在了一个 always 模块中，虽然这种写法从功能上来说并没有错误，但是它的可读性差，在编写的时候容易出错，往往不利于维护。

(2) 二段式状态机：是将组合逻辑和时序逻辑分开，具有较好的可读性，相对容易维护，不过组合逻辑输出较易出现毛刺等常见问题。

(3) 三段式状态机：除了具有两段式的优点，还对状态输出进行了寄存，可以有效地滤除毛刺。因此，我们这里推荐读者使用三段式的写法。

为了能够让大家进一步了解三段式状态机，下面我们就以《数字电路篇》中的自动售货机为例，下面我们给出自动售货机的三段式状态机代码，如代码 4.14 所示。

代码 4.14 三段式状态机代码

```
1  module Example_State
2  (
3    X,Z,CLK_50M,RST_N
4  );
5
6  input   CLK_50M;
7  input   RST_N;
8  input   X;
9  output  Z;
10
11 reg     Z;
12 reg     Z_N;
13
14 parameter S0 = 2'b00;
15 parameter S1 = 2'b01;
16 parameter S2 = 2'b10;
17
18 reg [1:0] FSM_CS;
19 reg [1:0] FSM_NS;
```

```
20
21  always @ (posedge CLK_50M or negedge RST_N)
22  begin
23      if(!RST_N)
24          FSM_CS <= 2'b00;
25      else
26          FSM_CS <= FSM_NS;
27  end
28
29  always @ (*)
30  begin
31      case(FSM_CS)
32      S0 :
33      begin
34          if(X == 1'b1)
35              FSM_NS = S1;
36          else
37              FSM_NS = S0;
38      end
39      S1 :
40      begin
41          if(X == 1'b1)
42              FSM_NS = S2;
43          else
44              FSM_NS = S1;
45      end
46      S2 :
47      begin
48          if(X == 1'b1)
49              FSM_NS = S0;
50          else
51              FSM_NS = S2;
52      end
53      endcase
54  end
55
56  always @ (posedge CLK_50M or negedge RST_N)
57  begin
58      if(!RST_N)
59          Z <= 1'b0;
60      else
61          Z <= Z_N;
62  end
63
```

```
always @ (*)
begin
    if((FSM_CS == S2) && (X == 1'b1))
        Z_N = 1'b1;
    else
        Z_N = 1'b0;
end

endmodule
```

下面我们就根据代码结合自动售货机来给大家进行讲解，我们这里假设 X 为一元硬币输入，投入一元硬币 X 为 1，未投入一元硬币 X 就为 0，Z 为饮料输出，送出饮料 Z 为 1，未送出饮料 Z 就为 0.S0 为初态，S1 为投入一元后的状态，S2 为投入两元后的状态。首先自动售货机处于 S0 状态，这时，如果我们投入一个一元硬币，也就是说，X 等于 1，那么电路状态由 S0 转换到 S1 状态，表示已收到一枚硬币，又因为只收到一个一枚硬币，饮料不会送出，也就是说，Z 为 0。当电路处于 S1 状态时，我们再投入一个一元硬币，也就是说，X 等于 1，那么电路状态会由 S1 状态转换到 S2 状态，表示已连续收到两个一元硬币，但是由于只连续收到两个，饮料也不会送出，Z 也就为 0。当电路处于 S2 状态时，我们再投入一枚一元硬币，也就是说 X 等于 1，那么电路状态会由 S2 状态转换到 S0 状态，表示已连续收到三个一元硬币，此时，饮料便会送出，Z 也就为 1。介绍完了代码，下面我们再来看下该三段式状态机的状态图，如图 4.14 所示。

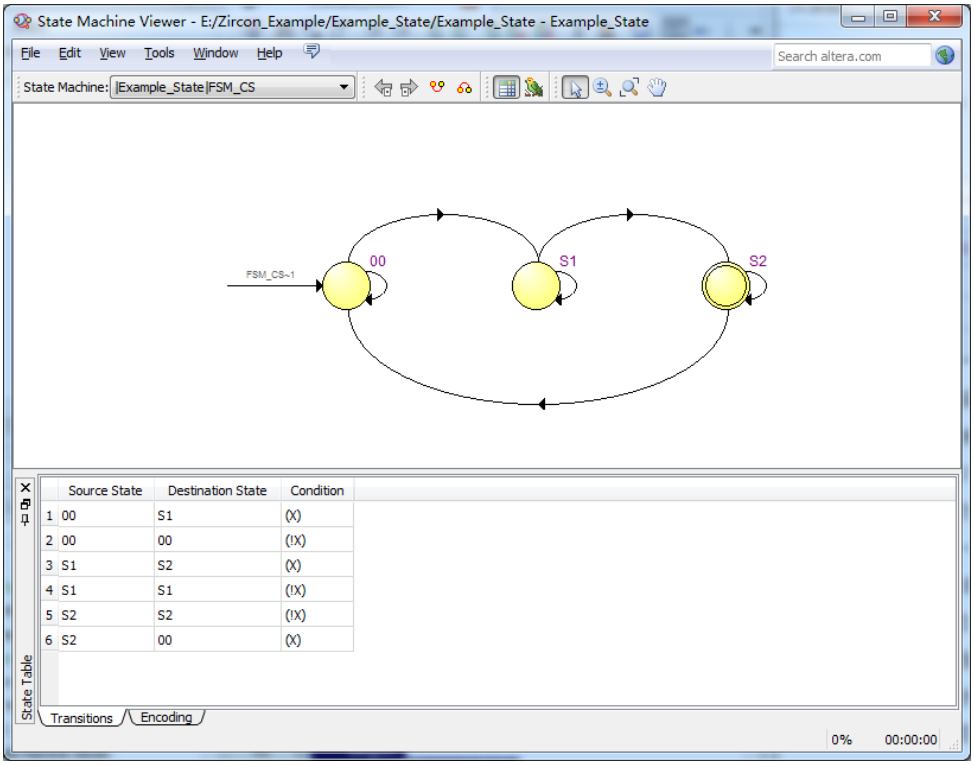


图 4.14 三段式状态机的状态图

从该图中，我们很容易就能看出三段式状态机的状态转移情况。这里我们需要说明的，该状态图是根据我们编写的 Verilog 代码由 Quartus II 软件自动生成的。

§ 4.8 代码风格的重要性

下面我们讲解的是代码风格的重要性，代码风格重要不重要，我们先不说，我们先来看，这里我们为大家准备了 4 种不同风格的代码，虽然它们的代码有多有少，但是它们所实现的功能是一样的，它们都实现一个 LED 闪烁，闪烁间隔时间是 1 秒。首先我们先来看下第一种风格，如代码 4.15 所示。

代码 4.15 第一种风格代码

```
1  module Example_Style3
2  (
3     a,b,c
4  );
5
6  //--------------------------------------------------------------------------------
7  //--    外部端口声明
8  //--------------------------------------------------------------------------------
9  input          a;
10 input          b;
11 output         c;
12
13 //--------------------------------------------------------------------------------
14 //--    内部端口声明
15 //--------------------------------------------------------------------------------
16 reg             e;
17 reg    [26:0]  d;
18
19 parameter f = 27'd50_000_000;
20
21 //--------------------------------------------------------------------------------
22 //--    逻辑功能实现
23 //--------------------------------------------------------------------------------
24 always @ (posedge a or negedge b)
25    begin
26       if(!b)
27          begin
28             e <= 1'b0;
29          end
30       else if (d == f)
31          begin
32             e <= ~e;
33             d <= 1'b0;
34          end
35       else d = d + 1'b1;
36    end
```

```
37
38  assign c = e;
39
40  endmodule
```

从这个代码中我们可以看出，这个代码确实够简洁，简洁到我们已经不知道该代码实现了什么功能。这个代码，它在语法结构和格式排版上可以说是非常的好，可是它犯的最大的错误就是在命名规则上，我们命名要做到简洁、清晰、有效，尽可能做到见名知意。从这个代码我们是完全看不出它命名的意义，我们只能看出该代码有三个端口，其中 a 和 b 是输入端口，c 是输出端口，如果我们不看 always 模块，我肯定猜不出 a 表示的是时钟端口，b 表示的是复位端口，而 c 表示的 LED 灯。说完了第一种代码风格，接下来我们再来看第二种代码风格，如代码 4.16 所示。

代码 4.16 第二种风格代码

```
1   module Example_Style2(CLK_50M,RST_N,LED1);
2   input CLK_50M;
3   input RST_N;
4   output LED1;
5   reg led_reg;
6   reg [26:0] time_cnt;
7   parameter SET_TIME_1S=27'd50000000;
8   always @ (posedge CLK_50M or negedge RST_N)
9   begin
10  if(!RST_N)
11  begin
12  led_reg<=1'b0;
13  end
14  else if(time_cnt==SET_TIME_1S)
15  begin
16  led_reg<=~led_reg;
17  time_cnt<=1'b0;
18  end
19  else time_cnt=time_cnt+1'b1;
20  end
21  assign LED1 =led_reg;
22  endmodule
```

从这个代码中，我们可以看到，命名规则是有了，但是没了语法结构和格式排版，所有代码揉成一团，这种代码风格我相信没有人会喜欢，使用这种代码风格编写的代码，即便是编写代码的本人，阅读起该代码也是很吃力的。当然，这里由于我们的代码量比较少，并且也不是特别的乱，所以我们感受不到这种代码的恐怖，大家可以想象一下，如果有一个成百上千行的代码使用的是这种风格编写的，然后让你去分析，我想此时你一定会崩溃。分析这种代码还不如直接重新编写来的方便。说完了第二种代码风格，接下来我们再来看第三种代码风格，如代码 4.16 所示。

代码 4.17 第三种风格代码

```
module Example_Style1
(
    //输入端口
    CLK_50M,RST_N,
    //输出端口
    LED1
);

//--------------------------------------------------------------------------------
//--    外部端口声明
//--------------------------------------------------------------------------------
input          CLK_50M;                  //时钟的端口,开发板用的 50M 晶振
input          RST_N;                    //复位的端口,低电平复位
output         LED1;                     //对应开发板上的 LED

//--------------------------------------------------------------------------------
//--    内部端口声明
//--------------------------------------------------------------------------------
reg             led_reg;                 //定义显示寄存器
reg    [26:0]   time_cnt;                //定义定时寄存器


//设置定时器的时间为 1s,计算方法为  (1*10^6)us / (1/50)us  50MHz 为开发板晶振
parameter SET_TIME_1S = 27'd50_000_000;

//--------------------------------------------------------------------------------
//--    逻辑功能实现
//--------------------------------------------------------------------------------
always @ (posedge CLK_50M or negedge RST_N)
    begin
        if(!RST_N)                       //判断复位
            begin
                led_reg <= 1'b0;         //初始化 led_reg 值
            end
        else if (time_cnt == SET_TIME_1S)//判断 1s 时间
            begin
                led_reg <= ~led_reg;     //如果到达 1s,显示寄存器将会被取反
                time_cnt <= 1'b0;        //如果到达 1s,定时计数器将会被清零
            end
        else time_cnt <= time_cnt + 1'b1;//如果未到 1s,定时计数器将会继续累加
    end

assign LED1 = led_reg;                   //最后,将显示寄存器的值赋值给端口 LED1
```

```
44
45  endmodule
```

如果学过 C 语言的读者,看到该代码可以说会倍感亲切，因为这就是利用 C 语言的编程思想来写的代码，我们可以从该代码中看出，该代码命名规则很好，做到了见名知意，格式排版也非常规范，并且还给每个代码添加上了注释。这种代码风格表面上看起来非常好，但是它却有很多缺点。比如说，通过这种代码风格编写出来的代码，很抽象，我们没有办法看出该代码所对应的硬件结构。不仅如此，通过这种代码风格编写出来的代码，我们对阻塞赋值和非阻塞赋值也是分不清楚的,在该代码中,我们不知道什么时候使用阻塞赋值,也不知道什么时候使用非阻塞赋值。当然了，这些还都不算严重，最严重的就是，通过这种代码风格编写的代码，我们使用的还是之前软件编程中的固有思路，也就是说，我们还没有真正的学会用硬件的方式去解决问题。

我们在这里要重点强调的是：虽然 Verilog 语言是根据 C 语言设计而来的，但是 Verilog 硬件描述语言和我们的 C 语言是大大的不同滴，我们写 C 语言代码是编程，我们写 Verilog 代码是描述。事实上，我们从硬件描述语言的名称中也是可以看出的，大家注意硬件描述语言名称中的“描述”两个字，它为什么是描述而不是设计呢？其主要原因就是硬件描述语言确实不是用来设计电路的，而仅仅是用来描述电路的。描述这个词精确地反映了硬件描述语言的本质。我们在写 Verilog 代码的时候，大脑里要先想好完成的具体功能应该用什么样的物理电路去实现，然后再用 Verilog 语言将该电路描述出来，而不能凭空地去写代码，只有存在的电路才是可物理实现的，Verilog 语言只不过是将这种设计转化为文字表达形式而已。我们只有这样，才能真正的做到代码在屏中，电路在心中。最终我们再来看下第四种代码风格，如代码 4.18 所示。

代码 4.18 第四种风格代码

```
1  module Example_Style
2  (
3     //输入端口
4     CLK_50M,RST_N,
5     //输出端口
6     LED1
7  );
8
9  //-------------------------------------------------------------------------
10 //--    外部端口声明
11 //-------------------------------------------------------------------------
12 input             CLK_50M;              //时钟的端口,开发板用的 50M 晶振
13 input             RST_N;                //复位的端口,低电平复位
14 output            LED1;                 //对应开发板上的 LED
15
16 //-------------------------------------------------------------------------
17 //--    内部端口声明
18 //-------------------------------------------------------------------------
19 reg     [26:0]    time_cnt;             //用来控制 LED 闪烁频率的定时计数器
20 reg     [26:0]    time_cnt_n;           //time_cnt 的下一个状态
```

```
21  reg               led_reg;              //用来控制 LED 亮灭的显示寄存器
22  reg               led_reg_n;            //led_reg 的下一个状态
23
24
25  //设置定时器的时间为 1s,计算方法为  (1*10^6)us / (1/50)us  50MHz 为开发板晶振
26  parameter SET_TIME_1S = 27'd50_000_000;
27
28  //-------------------------------------------------------------------------
29  //--    逻辑功能实现
30  //-------------------------------------------------------------------------
31  //时序电路，用来给 time_cnt 寄存器赋值
32  always @ (posedge CLK_50M or negedge RST_N)
33  begin
34      if(!RST_N)                          //判断复位
35          time_cnt  <=  27'h0;            //初始化 time_cnt 值
36      else
37          time_cnt  <=  time_cnt_n;       //用来给 time_cnt 赋值
38  end
39
40  //组合电路，实现 1s 的定时计数器
41  always @ (*)
42  begin
43      if(time_cnt == SET_TIME_1S)         //判断 1s 时间
44          time_cnt_n = 27'h0;             //如果到达 1s,定时计数器将会被清零
45      else
46          time_cnt_n = time_cnt + 27'h1;  //如果未到 1s,定时计数器将会继续累加
47  end
48
49  //-------------------------------------------------------------------------
50
51  //时序电路，用来给 led_reg 寄存器赋值
52  always @ (posedge CLK_50M or negedge RST_N)
53  begin
54      if(!RST_N)                          //判断复位
55          led_reg <=  1'b0;               //初始化 led_reg 值
56      else
57          led_reg <=  led_reg_n;          //用来给 led_reg 赋值
58  end
59
60  //组合电路，判断时间，控制 LED 的亮或灭
61  always @ (*)
62  begin
63      if(time_cnt == SET_TIME_1S)         //判断 1s 时间
64          led_reg_n = ~led_reg;           //如果到达 1s,显示寄存器将会改变 LED 的状态
```

```
65      else
66          led_reg_n = led_reg;              //如果未到 1s,显示寄存器将会将保持 LED 的原状态
67  end
68
69  assign LED1 = led_reg;                    //最后,将显示寄存器的值赋值给端口 LED1
70
71  endmodule
```

通过该代码，我们可以看出，该代码遵循组合逻辑电路和时序逻辑电路分开编写的一种风格。这种代码风格我们可以很清楚的知道每段代码所对应的硬件结构电路，也能够很清楚的知道在时序逻辑电路中，我们使用的是非阻塞赋值，在组合逻辑电路中，我们使用的是阻塞赋值。这种风格我们虽然不能说是最好，但是相对于其他的代码风格来说，那是要好上很多。下面我们就来简单的介绍一下整个代码。第 1 至 26 行是端口声明和参数定义。第 32 至 38 行是一个 always 模块，该 always 模块是一个时序电路，同时也是一个带有异步复位端口的边沿 D 触发器，它主要是用来存储定时器 time_cnt 的值。第 41 至 47 行也同样是一个 always 模块，不过该 always 模块是一个组合逻辑电路，它是用来判断定时器 time_cnt 有没有到达 1s，如果到达 1s，那么就将 time_cnt 清零，如果没有到达 1s，那么就继续累加。第 52 至 58 行，它和我们的第 32 至 38 行是非常类似的，不同的是，这里存储的是 led_reg 的值。第 61 至 67 行也和我们的第 41 至 47 行是类似的，不同的是，如果到达 1s，那么就将 led_reg 取反，如果没有到达 1s，那么就保持不变。介绍完了代码，下面我们就来看下它的 RTL 视图，如图 4.15 所示。

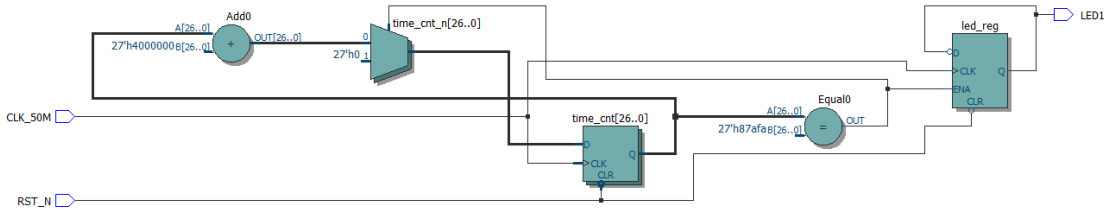


图 4.15 第四种风格的 RTL 视图

从该图中我们可以看到，该代码风格生成出来的 RTL 视图可以说非常的符合我们代码中的描述。该图中的 Add0 和二选一的数据选择器对应我们代码中的第 41 至 47 行。该图中的第一个 D 触发器 time_cnt 对应我们代码中的第 32 至 38 行。该图中的 Equal0 对应我们代码中的第 61 至 67 行。该图中的第二个 D 触发器 led_reg 对应我们代码中的第 52 至 58 行。至此，我们相信大家已经知道了代码风格到底重不重要。一个好的代码风格，能提高效率避免很多不必要的错误，一个好的代码风格，使代码阅读起来都是很舒畅的。希望大家能够养成一个好的代码风格习惯。

附　录

Verilog 语法手册

附　录

Verilog 语法手册

本附录部分内容参考并引用了——夏宇闻. 从算法设计到硬线逻辑的实现——实验练习与 Verilog 语法手册[M]. 北京：高等教育出版社，2001.

I. Verilog模块

在 Verilog 语言中，模块是层次的基本单元。模块中包括声明语句、功能描述和引用一些现存的硬件部件。有些模块只用来声明可被别的模块调用的参量，任务和函数。

关键字：module…endmodule

模块是 Verilog 设计中基本功能块，一个最简单的模块是由模块命名、端口列表两个部分组成。整个模块是由 module 开头，endmodule 结尾。每个模块都有它自己的模块名。

(1)　语法

```
module  模块名[（端口，……）];
模块条款
endmodule

模块条款 = {任取其一}
声明
参数定义
连续任务
实例引用
详细说明块
初始化块
总是执行块

声明 = {任取其一}
端口
网络
寄存器
参量
事件
任务
函数
```

(2)　规则

- 几个模块或几个 UDP (或它们的混合) 可以在一个文件中进行描述。(事实上，一个模块也可以分开在两个或更多的文件中描述，但不推荐这种做法)。
- 模块也可使用关键字 macromodule 来定义。其语法与用关键字 module 来定义模块是完全一样的。

- Verilog 编译器在编译宏模块时与编译一般模块时有所不同，比如不必为宏模块实例创建层次。这样，从仿真速度或存储介质的开销两方面来说，宏模块的编译更有效率。为了达到这个目的，宏模块的编译可能受制于实现时的某些特殊条件的限制。如果遇到这种情况，宏模块将被按一般模块编译。

(3) 注意

- 模块与宏摸块都以关键字 endmodule 作为结束标志。
- 尽量使每一个文件只包含一个模块。在大型设计中，这样做易于源代码的维护。在 module...endmodule 之间的语句都是并行执行的。

(4) 可综合性问题

- 每一个模块都被综合为一个独立的分层块，虽然有些工具的缺省配置规定把层次展平（为单层），但仍允许用户对综合后生成的网表层次进行控制。
- 不是所有的工具都支持宏模块的综合。

(5) 举例说明

```
module nand2(f,a,b);

input  a,
input  b,
output f

nand (f, a, b);

endmodule
```

II. Verilog端口

模块的端口是指硬件器件的引脚或接口的模型。

关键字：input、output、inout

端口列表也就是 I/O 的描述，而 input 也即是输入口的描述，output 是输出口的描述，inout 是双向端口，既可以利用它发送数据也可以利用它接受数据，主要在处理总线数据时使用，一般很少使用。

(1) 语法

```
input    [Range] Name1,Name2……;
output   [Range] Name1,Name2……;
inout    [Range] Name1,Name2……;
Range = [MSB : LSB] {MSB 表示最高位,LSB 表示最底位}
```

(2) 规则

- 在端口列表中列出的所有端口必须按次序排列或按端口名称排列，这两种排列方式是不同的，不能混合使用。

- 有端口的名称但没有端口表达式，如.A()，则表示在本模块中定义了不与任何东西相连的端口。
- 每个端口除了必须在端口列表中列出外，还必须声明该端口是输出（output）、输入（input）、还是双向端口（inout）。
- 每个端口不但要声明是输出、输入、还是双向端口，而且还要声明是连线（wire）还是寄存器(reg)类型，如果没声明，则会隐含地认为该端口是连线（wire）类型，且其位宽与相应的端口一致。如果某端口已被声明为一矢量，则其端口的方向和类型两个声明中的位宽必须一致。
- 输入和双向端口不能声明为寄存器类型。
- 输出端口的类型不能声明为实型（real）或实时型（realtime）。

(3) 注意

- 在测试模块中不要定义端口。
- 在模块定义时不建议使用命名的端口的列表，因为很少有人这样来定义模块端口，大家都不了解这种端口的定义形式。

(4) 举例说明

```
module Test1(a,b,c);

input            a;
input    [7:0]   b;
output   [7:0]   c;

Test2   Test2_Init
(
   .X  (a ),
   .Y  (b ),
   .Z  (c )
);

endmodule
```

III. Verilog线网

Net 是结构描述中为线路连接（连线和总线）建立的模型。net 的值是由 net 的驱动器所决定的。驱动器可以是门、UDP、实例模块或者连续赋值语句的输出。

关键字：wire、tri、wor、trior、wand、triand、tri0、tri1、supply0、supply1、vectored、scalared…

(1) 语法

```
NetType [ Expansion] [ Range] [ Delay] NetName,...;
线网有三个参数，1:Expansion,2:Range,3:Delay.参数可以省略
```

```
3
4  NetType = {下面类型任意选择}
5  wire,tri,wor,trior,wand,triand,tri0,tri1,supply0,supply1
6  Expansion = {下面类型任意选择}
7  vectored,scalared
8  Range = [MSB : LSB] {MSB 表示最高位,LSB 表示最底位}
9  Delay = #10; 延迟 10，仿真下使用
```

(2) 规则

- supply0 和 supply1 类型的 net 分别具有逻辑值 0 和 1，并可以为它定义驱动能力(supply strength)。
- tri0 和 tri1 类型的 nets，当没有驱动时，分别具有逻辑值 0 和 1，并可以为它定义驱动能力(pull strength)。
- 如果 net 的扩展（Expansion）选项选用了关键词 vectored，则不允许对它进行某位和某些位的选择，也不允许对它定义强度，PLI 会认为该 net 是不可扩展的；如果扩展（Expansion）选项选用了关键词 scalared，则允许对它进行某位和某些位的选择，也允许对它定义强度，PLI 将会认为该 net 是可扩展的，这些关键词是有参考价值的。
- 除了结构描述中的端口和标量连线不用声明其 net 类型外，其他类型的 net 变量在应用之前必须声明。
- 信号强度（从强到弱）及其属性:1、supply：驱动；2、strong：驱动；3、pull：驱动；4、large：存储；5、weak：驱动；6、medium：存储 7、small：存储 8；highz：高阻态。
- 数字电路中的信号可以用信号强度加以描述。当系统遇到信号之间的竞争时，需要考虑各组信号的状态和强度。如果驱动统一线网的信号强度不同，则输出结果是信号强度高的值；如果两个强度相同的信号之间连接到同一个线网，将会发生竞争，结果为不确定值 x。

(3) 注意

- 当 net 未被驱动时，对 tri0 或 tri1 类型的 net 的连续赋值不影响其值和强度，经常为强度（strength）保持为 pull，和逻辑值保持为 0（对 tri0）或 1（对 tri1）。
- 在 IEEE 标准和已成事实的 Cadence 公司标准中，扩展可选项的保留字 scalared 或 vectored 的位置有所不同，在 Cadence 标准中，保留字位于范围（range）选项的跟前。
- 在每个模块的块首明确地声明所有的 nets，即使是缺省的类型也应该明确地加以说明。通过清楚地说明设计意图，可以提高 Verilog 程序的可读性和可维护性。
- 只能用 supply0 和 supply1 来声明地和电源。

(4) 可综合性问题

- net 类型的变量被综合成线路连接，但是某些线路连接经优化后有可能被删去。
- 综合工具只支持 net 类型中 wire 型的综合，其它的 net 类型均不支持。

(5) 举例说明

```
module test(a,b);

input      a; //input 默认就是 wire
input      b; //input 默认就是 wire

wire  [7:0] b; //声明 b 为 wire 类型
wire  [7:0] c; //声明 c 为内部 wire 类型

endmodule
```

IV. Verilog寄存器

寄存器可存储在 initial、always、task 和 function 块中所赋的值，广泛地应用在行为建模中。

关键字：reg、integer、time、real、realtime…

(1) 语法

```
RegisterType [Range] MemoryName [Range]
RegisterType = {下面类型任意选择}
reg,integer,time
Range = [MSB : LSB] {MSB 表示最高位,LSB 表示最底位}

real RegisterName,...;
realtime RegisterName,...;
```

(2) 规则

- 寄存器类型变量只能用过程赋值语句赋值。
- 在具体实现时，整数（integer）类型的变量至小用 32 位，时间（time）类型的变量至小用 64 位寄存器。
- integer 或 time 类型的寄存器变量与位数相同的 reg 类型的寄存器变量行为是相同的。integer 和 time 型的寄存器变量也可像 reg 类型的寄存器变量一样对某位或某些位操作。而在表达式中，整数类型的值被当作有符号值，而 reg、time 类型的值被当作无符号值。
- 存储器类型数组中的每个元素作为整体可以进行读或写操作，如果要单独访问数组中某个元素的个别位，则必须先把这个元素的内容复制到某个位数相同的寄存器变量中才能进行。

(3) 注意

- 虽然 register 这个词指的是硬件寄存器（例如触发器），而寄存器（register）这个名字，在这里是指软件寄存器（即变量）。Verilog 寄存器常用于组合逻辑电路、锁存器、触发器和接口电路的描述和综合。
- realtime 类型寄存器变量是 Verilog 语言新增加的变量类型，目前还没有任何工具支持这种类型的变量。

- 有符号和无符号值的概念，不同版本的 Verilog 和用不同厂家的仿真器时，并不是完全一致的。因此，当使用位宽大于 32 位的有符号数或矢量时要特别注意。
- 运用 reg 类型变量来描述寄存器逻辑，integer 类型变量用于循环变量和计数，real 类型变量用于系统模块，time 和 realtime 类型变量用于测试模块中记录仿真时刻。

(4) 可综合性问题

- real, time 和 realtime 类型的寄存器变量是不可综合的。
- 在描述组合逻辑的 always 块中，寄存器被综合成 wire 型；如果存在不完整赋值的情况，则被综合成锁存器。在描述时序逻辑的 always 块中，寄存器根据块内语句的内容被综合成连线（wire）或者触发器。
- 运用目前的综合工具，整数被综合成 32 位，其值用二进制数表示，负数则用其二进制补码表示。
- 根据所用语句，存储器数组会被综合成触发器或连线，而不会被综合成 RAM 或 ROM 的器件。

(5) 举例说明

```
module test(a);

input        a;
reg          a;//声明 a 为外部 reg 类型
reg   [7:0]  b;//声明 b 为内部 reg 类型

endmodule
```

V. Verilog参数定义

参数是一种常量，通常出现在 module 内部，常被用于定义状态机的状态、数据位宽和延迟大小等，由于它可以再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。

关键字：parameter

(1) 语法

```
parameter   Name1 = 1, Name2 = 2,……;
localparam  Name3 = 3, Name4 = 4,……;
```

(2) 规则

- 参数是常量，在仿真期间更改参数的值是非法的。
- 在编译期间用 defparam 或者当包含参数的模块被引用时，可以改写其参数的值。
- localparam 关键字功能与 parameter 功能类似，唯一不同就是在上层模块中例化时不能修改参数的值。

(3) 注意

尽可能用参数给常数起一个有含义的名字。

(4) 可综合性问题

有些综合工具能把含有参数的模块当作模板，一旦读入模板，便能够用不同的参数值多次对该模板进行综合。所有的综合工具都支持不带改动参数的模块实例的综合。

(5) 举例说明

```
module adder(a,b,s);

input    a;
input    b;
output   s;

parameter N = 8;

wire     [N-1:0] a;
wire     [N-1:0] b;
reg      [N-1:0] s;

always @ (*)
begin
   s = a + b;
end

endmodule
```

关键字：specparam

类似于 parameter(参数)，但只能用在指定延时块中。

(1) 语法

```
specparam Name1 = 1, Name2 = 2,……;
```

(2) 规则

- specify 块中的常量表达式可以用数字和 specparam 来定义，但不能用参数(parameter)来定义，specparam 不能用在 specify 块（即指定延时模块）外。
- 利用 defparam 或在模块的实例引用时使用#，可以改写用 specparam 定义的延时参数值，用编程语言接口（PLI）也可以修改其值。

(3) 注意

- 在 specify 块中，用 specparam 来定义命名的延时参数比直接用数字要好。
- 这些延时参数应有一个命名的规则，这样便于对它们进行修改，如果有必要的话，也可以采用 PLI 的延时计数来进行修改。

(4) 举例说明

```
specify
   specparam tRise = 1.0,tFall = 1.0;
endspecify
```

关键字：defparam

编译时可重新定义参数值。如果是分层次命名的参数，可以在该设计层次内或外的任何地方重新定义参数。

(1) 语法

```
defparam Name1 = 1,Name2 = 2,...;
```

(2) 可综合问题

一般情况下是不可综合的。

(3) 注意

不要使用 defparam 声明语句!该声明语句过去常用于布线后的时延参数反标中，但现在时延参数反标一般用指定模块和编程语言接口（PLI）来做。在模块的实例引用时可用“#”号后跟参数的语法来重新定义参数。

VI. Verilog块语句

块语句的作用是将多条语句合并成一组，使它们像一条语句那样。

关键字：begin…end

串行语句的执行思路是顺序执行的，一般高级编程语言中的语句执行方式都是顺序执行的，C 语言就是如此，顺序执行的语句更容易帮助我们来表达我们的设计思想，尤其是使描述时序逻辑变得容易。所以，虽然 FPGA 的设计思想都是并行的，module 中仅支持并行语句的调用，但是为了方便设计者表达自己的思想，Verilog 中的一些并行语句中的字语句体允许是顺序执行的。而关键字 begin...end 就被赋予此使命。

(1) 语法

```
begin
   块内声明语句……
   语句……
end

块内声明语句 = {任选其一}
寄存器变量
参数
事件
```

(2) 规则

- begin...end 块必须包含至少一个声明语句。在 begin...end 中间的语句都是顺序执行的语句。定时控制是相对于前一声明语句的，当最后的声明语句执行完毕后，begin...end

块便结束。

- begin...end 和 fork...join 块可以自我嵌套或互相嵌套。如果 begin...end 块包含局部声明，则它必须被命名（即必须有一个标识）。如果要禁止（disable）某个 begin...end 块，那么被禁止的 begin...end 块必须有名字。

(3) 注意

- Verilog LRM 允许 begin...end 块在仿真时被交替执行。这就是说如果 begin...end 块包含两个相邻且其间没有时间控制的声明语句时，仿真器仍有可能在同一时刻在这两个语句之间执行另一个进程的部分语句（例如另一个 always 块中的语句）。这就是 Verilog 语言如果不加约束的话，便不能与硬件有确定对应关系的原因。
- 甚至在并不需要局部声明，也不想禁止 begin...end 块时，也可以对该 begin...end 块加标识命名，以提高其可读性。给不用在别处的寄存器作局部声明，能使声明的意图变得清楚。

(4) 举例说明

```
case(……)                  always @ (posedge clk)        if(……)
    ……: begin …… end      begin                         begin
    ……: begin …… end          ……                            ……
    ……: begin …… end          ……                            ……
    default:……;                ……                            ……
endcase                    end                           end
```

关键字：fork…join

恰恰与 begin…end 相反，fork…join 可将多个语句集合在一个块中，以使它们能被并发地执行。

(1) 语法

```
fork
   块内声明语句……
   语句……
join

块内声明语句 = {任选其一}
寄存器变量
参数
事件
```

(2) 规则

- fork...join 块必须至少包括一条语句。fork…join 块里的语句是并发执行的，因此 fork…join 块内语句的顺序是无所谓的。时间控制是相对于块的开始时刻的。 当 fork…join 块里所有的语 句执行完毕后，块也就执行完毕了。
- begin…end 和 fork…join 块可以自身嵌套或互相嵌套。如果想在某 fork…join 块内包

含块内局部声明语句，那么必须对该块命名（即该块必须有一个标识符号）。如果想要禁止某 fork…join 块的运行，则该块必须已被命名。

(3) 可综合性问题

fork 语句不可综合。

(4) 注意

fork…join 语句在描述并发形式的行为时很有用。

(5) 举例说明

```
initial
fork
   #20    Data = 8'hae;
   #40    Data = 8'hxx;// 本句最后执行
          Reset = 0;    // 本句最先执行
   #10    Reset = 1;
join      // 在第 40 个时间单位时结束
```

VII. Verilog比较语句

关键字：if…else

if 语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的两种操作之一。

(1) 语法

```
if(表达式)
   begin
      语句;
   end
else if(表达式)
   begin
      语句;
   end
else
   begin
      语句;
   end
```

(2) 规则

当表达式的值为非零时被认为是真，当值为 0、x 或 z 时被认为是假。

(3) 注意

- 如果在 if 或 else 分支中有超过一条的语句需要执行，则必须用 begin...end 或 fork...join 将其包括。
- 在使用嵌套的 if...else 语句时，当 else 分支省略时，要特别注意。else 只与离它最近

的前面的那个 if 相关联。Verilog 编译器不能判别源代码中省略的 else 分支。

- 如果对某些条件需要先进行测试，在这种情况下应选用嵌套的 if...else 语句。如果所有的条件优先权一致，则应选用 case 语句。

(4) 可综合性问题

- if 声明语句中的赋值语句通常被综合为多路器。在无时钟的 always 块中，当输入变化时而输出仍能保持不变的那些赋值语句，将被综合为透明锁存器，而在有时钟的 always 块中，它们则被综合为循环锁存器。
- 在某些情况下，嵌套的 if 语句会被综合为多层的逻辑。用 case 语句可以避免出现这种情况。

(5) 举例说明

```
if((a >= b) & (a >= c))    //如果 a 大于等于 b 与 a 大于等于 c
   max = a;                //那么最大值为 a
else if(b >= c)            //否则如果 b 大于等于 c
   max = b;                //那么最大值为 b
else                       //否则
   max = c;                //那么最大值为 c
```

关键字：case、casex、casez…default、endcase

case 语句是一种多分支选择语句，if 语句只有两个分支可供选择，而实际问题中常常需要用到多分支选择，Verilog 语言提供的 case 语句直接处理多分支选择。case 语句通常用于微处理器的指令译码。

(1) 语法

```
CaseType(表达式)
   表达式 1:begin 语句; end
   表达式 2:begin 语句; end
   表达式 3:begin 语句; end
   表达式 4:begin 语句; end
   default:begin 语句; end
endcase

CaseType = {任取其一}
case,casex,casez
```

(2) 规则

- 对于 casex 来说，它会将 x、z 视为不关心位；而对于 casez 来说，它会将 z 视为不关心位。
- 在 case 语句中最多只允许有一个 default 项。当没有一个分支标号表达式能与 case 表达 式的值相等时，便执行 default 项。(标号是位于冒号左边的一个表达式或用逗号隔开的几个表达式,标号也可以是保留字 default,在其后面可以跟冒号也可以不跟冒号。)
- 如果某标号是用逗号隔开的两个或两个以上表达式，只要其中任何一个表达式与 case

表达式的值相等时，就可执行该标号的操作。

- 如果没有一个标号表达式与 case 表达式的值相等，又没有 default 声明语句，该 case 声明语句没有任何作用。

(3) 注意

- 如果在标号分支中有一个以上的声明语句，这些声明语句必须放在一个 begin…end 或 fork…join 块中。
- 只有第一个与 case 表达式的值相等的标号分支才被执行。case 语句的标号并不一定是互斥的，所以当错误地重复使用相同的标号时，Verilog 编译器不会提示出错。
- casex 或 casez 声明语句的语法是用保留字 endcase 作为结束，而不是用 endcasex 或 endcasez 来结束。
- 在 casex 声明语句的表达式中的 x（不定值）或 z（高阻值）可以和任何值相等，casez 中的 z 也是如此。这有可能会给仿真结果带来混乱。
- 为了使仿真能顺利进行，常常用 default 作为 case 声明的最后一个分支，以控制无法与标号匹配的 case 变量。
- 通常情况下用 casez 比用 casex 更好一些，因为 x 的存在可能会导致仿真出现令人误解和混乱的结果。
- 在 casex 和 casez 声明的标号中用?来代替 z 比较好，因为这样做比较清楚，是一个无关项，而不是一个高阻项。

(4) 可综合性问题

- case 声明语句中的赋值语句通常被综合成多路器。如果变量（如寄存器或 net 类型）被用作 case 语句的标号，它就会被综合成优先编码器。
- 在一个无时钟触发的 always 块中，如有不完整的赋值（即对某些输入信号的变化其输出仍保持不变，未能及时赋值），它将被综合成透明锁存器。在一个有时钟触发的 always 块中，如有不完整的赋值，它将被综合成循环移位寄存器。

(5) 举例说明

```
1  module mux41c(c,s,z);
2
3  input wire  [3:0]   c;
4  input wire  [1:0]   s;
5  output reg          z;
6
7  always @ (*)
8  case(s)
9     0 : z = c[0];
10    1 : z = c[1];
11    2 : z = c[2];
12    3 : z = c[3];
13    default : z = c[0];
14 endcase
```

```
15
16 endmodule
```

VIII. Verilog循环语句

Verilog 中的循环语句有很多种，包括 for 循环、while 循环、repeat 循环以及 forever 循环等。这些循环语法中除了 for 循环有时候可以用来帮助我们简化一些代码的编写外，基本都是主要用于仿真激励的设计。

关键字：forever、disable

使一个或一个以上语句无限循环地执行。

(1) 语法

```
1 forever  语句;
2 forever begin 多条语句; end
```

(2) 注意

- forever 循环应包括定时控制或能够使其自身停止循环、否则循环将无限进行下去。
- forever 用在测试模块中描述时钟时，常用 disable 来跳出循环。

(3) 可综合性问题

forever 循环语句常用于产生周期性的波形，用来作为仿真测试信号。它与 always 语句不同之处在于不能独立写在程序中，而必须写在 initial 块中。

(4) 举例说明

```
1  initial
2  begin : Clocking
3    Clock = 0;
4    forever #10 Clock = !Clock;
5  end
6
7  initial
8  begin : Stimulus
9    ...
10   disable Clocking;   // 停止时钟
11 end
```

关键字：repeat

把一个或多个声明语句重复地执行指定的次数。

(1) 语法

```
1 repeat(表达式) 语句;
2 repeat(表达式) begin 多条语句; end
```

(2) 规则

重复执行的次数是由表达式的数值所决定的，如果该值为 0, X 或 Z，则不会有重复。

(3) 可综合性问题

只有部分综合工具可以综合 repeat 语句，而且只有当该循环中的每个循环的分支都被时钟事件，如被@(posedge Clock)，所中断时才有可能被综合成电路

(4) 举例说明

```
initial
begin
    Clock = 0;
    repeat (MaxClockCycles)
    begin
       #10 Clock = 1;
       #10 Clock = 0;
    end
end
```

关键字：while

只要控制表达式为真（即不为 0），循环语句就重复进行。

(1) 语法

```
while(表达式) 语句;

while(表达式)
begin
    多条语句;
end
```

(2) 可综合性问题

只有当循环块有事件控制（即@(posedge Clock)）才可综合。

(3) 举例说明

```
reg [15:0] word;

while(word)
begin
   if(word[0])
     time_cnt = time_cnt + 1;
   word = word >> 1;
end
```

关键字：for

一般用途的循环语句。允许一条或更多的语句能被重复地执行。

(1) 语法

```
for(循环变量赋初值;循环结束条件;循环变量增值)
   执行语句;
```

(2) 规则

当 for 循环开始执行时,循环计数变量已赋于初始值。在每一次循环执行之前(包括第一次),都必须首先检查表达式的值；如果它为假 (即为 0、x、或 z),则立刻退出循环。而在每一次循环重复执行之后，都要对迭代次数寄存器重新赋值。

(3) 注意

不要使用位宽小的 reg 类型变量作为循环变量。在测试存有负数值的寄存器变量时要格外注意。由于加减操作是可替换的，并且 reg 类型变量被看作是无符号数，所以循环表达式可能永远不会为假，从而导致循环无限止地进行。

(4) 可综合问题

如果循环的边界是固定的，那么在综合时该循环语句被认为是重复的硬件结构。

(5) 举例说明

```
integer i;

for(i = 0; i < 4; i = i+1)
begin
   a[i] = a[i] & b[3-i];
end
```

在描述设计时，for 循环一般不应该进行功能描述，而应该只进行结构描述。否则，由于 for 循环抽象级别比较高，编译器不一定能够正确给出对应的实现电路，而且有时候很可能就不存在能够对应该功能的电路。

IX. Verilog任务定义

任务常用于把模块代码分割成由若干声明语句构成的较大的块，便于模块代码的理解和维护，也可以从模块代码的不同位置执行这样一个常见的顺序声明语句块 3002

关键字：task…endtask

(1) 语法

```
task <任务名>
<端口及数据类型声明语句>;
<语句 1>;
<语句 2>;
  ……;
<语句 n>;
endtask
```

(2) 规则

- 若用于任务中的命名变量或参数没有在任务块中声明，则指的是在模块中声明的命名

变量或参数。

- 任务中的 input、output 和 inout 的个数不受限制（也可以为零个）。
- 任务中的变量（包括输入和双向端口（inout））可以声明为寄存器型。如果没有明确地声明，则默认为寄存器型，且其位宽与相应的变量匹配。
- 当启动任务时，相应于任务的输入和双向端口（inout）的变量表达式的值被存入相应的变量寄存器中。当任务结束时，输入和双向端口（inout）的变量寄存器中的值又被代入启动任务的语句中相应的表达式。

(3) 注意

- 和模块的端口定义不一样，任务的变量不能在任务名后的括号中定义。
- 任务中若包括一句以上的语句，必须要用 begin…end 或 fork…join 将其包含成块。
- 任务的输入、双向端口(inout)、输出和局部寄存器的值都是静态储存的，也就是说即使多次启动任务，也只有一份这些寄存器的拷贝。若第一次启动的任务还未完成，便第二次启动该任务，其输入、双向端口(inout)、输出和局部寄存器的值便会被覆盖。
- 当被启动的任务运行结束时，输出和双向端口(inout)的值被代入任务中相应的寄存器表达式。如果任务中的输出和双向端口(inout)在赋值后有时间的控制，则相应的寄存器只能在定时控制延迟后才被更新。
- 同样，对输出和双向端口(inout)寄存器变量的非阻塞赋值语句也不会起作用，因为当任务返回时，赋值语句可能还未生效
- 复杂 RTL 模块通常需要用多个 always 块来构造。建议最好不要采用一个 always 块运行多个任务的方案。
- 在测试块中可用任务来产生重复的激励序列。例如，对存储器的数据读写（见例）序列。
- 某任务如果被多个模块引用，可以把它定义为一个独立的模块（只包括该任务），并可用层次命名来引用它。

(4) 可综合问题

包含定时控制语句的任务是不可综合的。启动的任务往往被综合成组合逻辑。

(5) 举例说明

```
1  task Counter;
2
3  input        rst_n;
4  inout [3:0]  time_cnt;
5
6  if(rst_n)
7     time_cnt = 0;
8  else
9     time_cnt = time_cnt + 1;
10
11 endtask
```

X. Verilog函数定义

用于把多个语句组合在一起，来定义新的数学或逻辑函数。函数是在模块内部定义的，并且通常在本模块中调用，也能根据按模块层次分级命名的函数名从其他模块调用。

关键字：function…endfunction

(1) 语法

```
function <返回值的类型或范围> 函数名;
端口说明;
语句;
begin
   语句;
     ……;
   语句;
end
endfunction
```

(2) 规则

- 函数必须至少含有一个输入变量。它不能有任何输出或输入/输出双向变量。
- 函数不能包含时间控制语句（如延迟#，事件控制@ 或等待 wait）。
- 函数是通过对函数名赋值的途径返回其值的，（就好比它是一个寄存器）。
- 函数不能启动任务。
- 函数不能被禁用。

(3) 注意

- 函数的输入变量不能象模块的端口那样列在函数名后的括弧里；在声明输入时把这些输入端口列出即可。
- 如果函数包含一条以上的语句，这些语句必须包含在 begin...end 或 fork...join 块中。

(4) 可综合问题

函数的每一次调用都被综合为一个独立的组合逻辑电路块。

(5) 举例说明

```
function [7:0] ReverseBits;

input [7:0] Byte;
integer i;

begin
for (i = 0; i < 8; i = i + 1)
   ReverseBits[7-i] = Byte[i];
end

```

```
endfunction
```

XI. Verilog等待语句

关键字：@、#、wait

Verilog 中有三种等待语句，事件等待语句、直接时间等待语句、表达式等待语句。

(1) 事件等待语句

事件等待语句的语法如下：

```
@ (posedge clk_50m or negedge rst_n)
```

每个 always 程序块中必有一个事件等待语句，除此以外，事件等待语句还可以位于 always 程序块中，此时的 always 程序块主要是用于仿真。

(2) 直接时间等待语句

直接时间等待语句的语法如下：

```
#10 语句;
```

直接时间等待语句只能用于仿真。

(3) 表达式等待语句

表达式等待语句的语法如下：

```
wait(表达式);
```

当表达式为真的时候程序才往下执行，它也主要用于仿真。

XII. Verilog赋值语句

关键字：assign

连续赋值语句是针对线网变量的一种赋值语句，线网变量一般对应到 FPGA 中的一段连线，而连线的值时会随着它的驱动源的变化而不停变化的，因此也就称之为连续赋值语句。顾名思义，连续赋值语句肯定是一种描述组合逻辑的语句。

(1) 语法

```
assign wire_name = 表达式;
```

(2) 规则

在连续赋值声明语句之前，赋值语句左边的线网类型的变量必须明确声明。

(3) 注意

连续赋值并不等同于过程连续赋值语句，虽然它们相似。确保把 assign 放在正确的地方。连续赋值语句必须放在任何 initial 和 always 块外。过程连续赋值语句可放在该语句被允许放的地方执行（在 initial、always、task、function 等块内部）。

(4) 可综合问题

- 综合工具不能处理连续赋值语句中的延迟和强度，在综合中被忽略。请用综合工具指定的定时约束来代替。
- 连续赋值语句将被综合成为组合逻辑电路。

(5) 举例说明

```
module test(a,b,s,c);

input  a;
input  b;
output s;
output c;

assign s = a ^ b;
assign c = a & b;

endmodule
```

关键字：initial

在仿真一开始就执行并只执行一次的声明语句，可执行只包含一条语句或多条语句组成的块。

(1) 语法

```
initial
begin
    语句 1;
    语句 2;
       ……
    语句 n;
end
```

(2) 注意

- 包含多个语句的 initial 块需要用 begin...end 或 fork...join 块将这些语句合成一块。
- 在仿真测试文件中可使用 initial 语句来描述激励。

(3) 可综合问题

initial 语句是不可综合的。

(4) 举例说明

```
initial  //仿真下使用
begin
    Load      = 0; // Time 0
    Enable    = 0; // Time 0
    Reset     = 0; // Time 0
    #10 Reset = 1; // Time 10
```

```
    #25 Enable= 1; // Time 35
    #100 Load = 1; // Time 135 end
end
```

关键字：always、posedeg、negedge

与 initial 恰恰相反，always 程序块会不断地、循环地执行。因为，initial 程序块主要负责模块的初始化功能，而 always 程序块才主要负责模块的功能描述。使用方法如下：

```
always @ (posedge clk_50m or negedge rst_n)
begin
   语句;
end
```

always 共有三种基本类型：1、纯组合逻辑 always、2、纯时序逻辑 always、3、具有异步复位的同步时序逻辑 always。下面我们就对这三种基本类型分别进行介绍：

(1) 纯组合 always

纯组合 always 使用方法如下：

```
always @ (a)
begin
    a = ~a;
end
```

上述例子描述了一个非门的结构，关于纯组合 always 程序块，有以下三点需要注意：

- 纯组合 always 程序块中的语句强烈推荐值使用阻塞赋值符号，而时序 always 程序块中推荐只使用非阻塞赋值符号，否则会带来非常多的隐患。这样也更好地能区分开阻塞赋值和非阻塞赋值的区别。
- 虽然从字面上理解，always 是在变量 a 出现变化的情况下才触发执行的，但是不可自作聪明地将上例写出：

```
always @ (posedge a or negedge a)
begin
    a = ~a;
end
```

注意，只有时序逻辑才能用 posedge 和 negedge 关键字，虽然从代码时间解释来看上述两例好像功能相似，但是若出现沿事件关键字，则编译器会将程序块综合为时序逻辑，而这个世界上目前还没有既能够敏感一个信号上升沿有能敏感这个信号下降沿的触发器，所以综合会报错。

- 若括号中有多个变量，则可以用逗号“,”或者关键字 or 分隔开来。如果括号中的变量确实太多，Verilog 给大家提供了一个偷懒的方法，那就是使用匹配符号“*”，此时编译器将会完成括号中的元素推断。例如：

```
always @ (*)
```

(2) 纯时序 always

纯时序 always 的使用方法如下：

```
always @ (posedge clk)
begin
   a <= b;
end
```

(3) 具有异步复位的同步时序 always

具有异步复位的同步时序 always 使用方法如下：

```
always @ (posedge clk_50m or negedge rst_n)
begin
   if(!rst_n)
      n <= 8'b0;
   else
      n <= m;
end
```

XIII. Verilog多输入门

关键字：and、nand、nor、or、xor、xnor

这些逻辑门只能有一个输出端口，可以有多个输入端口，其中输出端口是第一个端口。使用方法如下：

```
and  (out,in1,in2,……);
nand (out,in1,in2,……);
nor  (out,in1,in2,……);
or   (out,in1,in2,……);
xor  (out,in1,in2,……);
xnor (out,in1,in2,……);
```

它们的真值表如下：

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

XIV. Verilog多输出门

关键字：buf、not

这些逻辑门只能有一个输入端口，可以有多个输出端口，其中输入端口是最后一个端口，使用方法如下：

```
1 buf (out1,out2...,in);
2 not (out1,out2...,in);
```

它们的真值表如下：

buf	0	1	x	z
out	0	1	x	z

not	0	1	x	z
out	1	0	x	x

XV. Verilog三态门

关键字：bufif1、bufif0、notif1、notif0

它们只能有一个数据输出端口、一个数据输入端口和一个控制输入端口，第一个端口是数据输出端口，第二个端口是数据输入端口，第三个端口是控制输入端口。对于 bufif1 和 notif1，当控制等于 1 时，数据通过；当控制等于 0 时，输出为高阻态 z。对于 bufif0 和 notif0，当控制等

于 0 时，数据通过；当控制等于 1 时，输出为高阻态 z。使用方法如下：

```
bufif1 (outa,inb,controlc);
bufif0 (outa,inb,controlc);
notif1 (outa,inb,controlc);
notif0 (outa,inb,controlc);
```

它们的真值表如下：

bufif0	0	1	x	z
in0	0	z	0/z	0/z
in1	1	z	1/z	1/z
inx	x	z	x	x
inz	x	z	x	x

bufif1	0	1	x	z
in0	z	0	0/z	0/z
in1	z	1	1/z	1/z
inx	z	x	x	x
inz	z	x	x	x

notif0	0	1	x	z
in0	1	z	1/z	1/z
in1	0	z	0/z	0/z
inx	x	z	x	x
inz	x	z	x	x

notif1	0	1	x	z
in0	z	1	1/z	1/z
in1	z	0	0/z	0/z
inx	z	x	x	x
inz	z	x	x	x

XVI. Verilog上拉和下拉电阻

关键字：pullup、pulldown

这类门设备没有输入只有输出，上拉电阻将输出置为 1；下拉电阻将输出置为 0，使用方法如下：

```
pullup   (a);
pulldown (a);
```

XVII. Verilog MOS开关

关键字：cmos、pmos、nmos、rcmos、rpmos、rnmos

这类门用来为单向开关建模。即数据从输入流向输出，并且可以通过设置合适的控制输入关闭数据流。pmos(p 类型 MOS 管)、nmos(n 类型 MOS 管)，rnmos(r 代表电阻)和 rpmos 开关有一个输出、一个输入和一个控制输入，使用方法如下：

```
pmos     (outa,datab,controlc);
nmos     (outa,datab,controlc);
rpmos    (outa,datab,controlc);
rnmos    (outa,datab,controlc);
```

如果 nmos 和 rnmos 开关的控制输入为 0，pmos 和 rpmos 开关的控制为 1，那么开关关闭，即输出为 z；如果控制是 1，输入数据传输至输出；与 nmos 和 pmos 相比， rnmos 和 rpmos 在输入引线和输出引线之间存在高阻抗(电阻)。因此当数据从输入传输至输出时，对于 rpmos 和 rmos，存在数据信号强度衰减。这些开关的真值表如下：

pmos rpmos	0	1	x	z
in0	0	z	0/z	0/z
in1	1	z	1/z	1/z
inx	x	z	x	x
inz	z	z	z	z

nmos rnmos	0	1	x	z
in0	z	0	0/z	0/z
in1	z	1	1/z	1/z
inx	z	x	x	x
inz	z	z	z	z

表中的某些项是可选项。例如，1/z 表明，根据输入和控制信号的强度，输出既可以为 1，也可以为 z。cmos(mos 求补)和 rcmos(cmos 的高阻态版本)开关有一个数据输出，一个数据输入和两个控制输入。使用方法如下：

```
cmos    (outa,datab,controlc,controld);
rcmos   (outa,datab,controlc,controld);
```

cmos(rcmos)开关行为与带有公共输入、输出的 pmos（rpmos）和 nmos(rnmos)开关组合十分相似。

XVIII. Verilog双向开关

关键字：tran、rtran、tranif0、rtranif0、tranif1、rtranif1

由于这些关键字无法综合，这里就简单介绍一下。这些开关是双向的，即数据可以双向流动，并且当数据在开关中传播时没有延时。后 4 个开关能够通过设置合适的控制信号来关闭。tran 和 rtran 开关不能被关闭。tran 或 rtran(tran 的高阻态版本)开关实例语句的语法如下：

```
tran    (signala,signalb);
rtran   (signala,signalb);
```

只有两个端口，并且无条件地双向流动，即从 signala 向 signalb，反之亦然。 其它双向开关的实例语句的语法如下：

```
tranif0  (signala,signalb,controlc);
rtranif0 (signala,signalb,controlc);
tranif1  (signala,signalb,controlc);
rtranif1 (signala,signalb,controlc);
```

前两个端口是双向端口，即数据从 signala 流向 signalb，反之亦然。第三个端口是控制信号。如果对 tranif0 和 tranif0，controlc 是 1；对 tranif1 和 rtranif1，controlc 是 0；那么禁止双向数据流动。对于 rtran、rtranif0 和 rtranif1，当信号通过开关传输时，信号强度减弱。

XIX. Verilog用户定义的原语

用户定义的原语是从英语 User Defined Primitives 直接翻译过来的，简称 UDP。利用 UDP 用户可以定义自己设计的基本逻辑元件的功能，也就是说，可以利用 UDP 来定义自己特色的用于仿真的基本逻辑元件模块并建立相应的原语库。这样，就可以与调用 Verilog HDL 基本逻辑元件同样的方法来调用原语库中相应的元件模块，并进行仿真。由于 UDP 是用查表的方法是来确定其输出的，用仿真器进行仿真时，对它的处理速度较对一般用户编写的模块块得多。与一般的用户模块比较，UDP 更为基本，它只能描述简单的能用真值表表示的组合或时序逻辑。UDP 模块的结构与一般模块类似，只是不用 module 而改用 primitive 关键字开始，不用 endmodule 而改用 endprimitive 关键字结束。在 Verilog 的语法中，还规定了 UDP 的形式定义和必须遵守的几个要点，与一般模块又不同之处，将在下面加以介绍。

关键字：primitive、endprimitive、table、endtable

(1) 语法

```
primitive 元件名(输出端口名,输入端口名 1,输入端口名 2,...);

output   输出端口名;
input    输入端口名 1;输入端口名 2;...;
reg      输出端口名;

initial
begin
   输出端口寄存器或时序逻辑内部寄存器赋初值(0,1 或 x);
end

table
  //输入 1  输入 2  输入 3  ...  :  输出
    逻辑值  逻辑值  逻辑值 ...  :  逻辑值;
    逻辑值  逻辑值  逻辑值 ...  :  逻辑值;
    逻辑值  逻辑值  逻辑值 ...  :  逻辑值;
     ...     ...    ...  ...  :   ...  ;
endtable

endprimitive
```

(2) 注意

- UDP 只能有一个输出端，而且必须是端口说明列表的第一项。
- UDP 可以由多个输入端，最多允许有 10 个输入端。
- UDP 所有端口变量必须是标量，也就是必须是 1 位的。
- 在 UDP 的真值表中，只允许出现 0,1,x 的 3 种逻辑值，高阻值状态时不允许出现的。
- 只有输出端才可以被定义为寄存器类型变量。
- initial 语句用于为时序电路内部寄存器赋初值，只允许赋 0,1,x 的 3 种逻辑值，默认值为 x。

对于数字系统的设计人员来说，只要了解 UDP 的作用就可以了；而对微电子行业的基本逻辑元器件设计工程师，必须深入了解 UDP 的描述，才能把设计的基本逻辑元件，通过 EDA 工具呈现给系统设计工程师。

XX. Verilog驱动强度

关键字：supply0、supply1、weak0、weak1、pull0、pull1、highz0、highz1、strong0、strong1、large、small、medium

除了逻辑值外，net 类型的变量还可以定义强度，因而可以更精确地建模。Net 的强度来自于动态 net 驱动器的强度。在开关级仿真时，当 net 由多个驱动器驱动且其值互相矛盾时，常用强度（strength）的概念来描述这种逻辑行为。

(1) 语法

```
strength0
strength1
chargeStrength

strength0 = {任选其一}
supply0,strong0,pull0,weak0,highz0
strength1 = {任选其一}
supply1,strong1,pull1,weak1,highz1
chargeStrength = {任选其一}
large,medium,small
```

(2) 规则

- 关键词 strength0 和 strength1 用于定义 net 的驱动器强度。其中 strength 表示强度，与紧跟着的 0 和 1 连起来分别表示输出逻辑值为 0 和 1 时的强度。
- 在强度声明中可选择不同的强度关键字来代替 strength，但(highz0，highz1)和 (highz1,highz0)这两种强度定义是不允许的，在 pullup（上拉）和 pulldown（下拉）门的强度声明中 highz0 和 highz1 是不允许的。
- 默认的强度定义为 (strong0,strong1)，但下述情况除外：
  - 对于 pullup and pulldown 门，默认强度分别为(pull1) 和 (pull0)。
  - 对于 trireg 的 net，默认强度为 medium
  - 强度定义为 supply0 和 supply1 的 Net，总是能提供强度。
- 在仿真期间，net 的强度来自于 net 上的主驱动强度（即具有最大强度值的实例或连续赋值语句）。如果 net 未被驱动，它会呈现高阻值，但以下情况除外：
  - tri0 和 tri1 类型的 net 分别具有逻辑值 0 和 1，并为 pull 度。
  - trireg 类型的 net 保持它们最后的驱动值。
  - 强度为 supply0 和 supply1 的 nets 分别具有逻辑值 0 和 1，并能提供驱动能力。
- 强度值有强弱顺序，可从 supply（最强的）依次减弱排列到 highz（最弱的），当需要

确定 Net 的确实逻辑值和强度时，或者当 Net 由多个驱动器驱动而且驱动相互间出现冲突时，出现冲突的两个强度值在强弱顺序表中的相对位置就会对该 Net 的真实逻辑值起作用。

- 信号强度（从强到弱）及其属性:1、supply：驱动；2、strong：驱动；3、pull：驱动；4、large：存储；5、weak：驱动；6、medium：存储 7、small：存储 8；highz：高阻态。

(3) 可综合性问题

不可综合

XXI. Verilog过程性连续赋值

关键字：deassign、force、release

连续赋值适用于线网，过程赋值适用于寄存器。但是还有一类赋值方式，它既能对线网赋值也能对寄存器赋值（但不能是寄存器的位选择或部分选择），这种赋值方式被称为过程性连续赋值。它属于过程性赋值而非连续赋值，所以它能出现在 always 和 initial 语句中（连续赋值语句不能出现在 always 和 initial 内）。但是它又有连续赋值的一些特性，就是在过程性连续赋值语句中，右端表达式中操作数的任何变化都会引起赋值语句的重新执行。过程性赋值语句有两种类型：assign...deassign(赋值-重新赋值)，force...release(强制-释放虽然它也可以用于对寄存器赋值，但主要用于对线网赋值）。assign 用于对寄存器赋值，deassign 用于取消之前由 assign 赋给某寄存器的值。也就是说，使用 assign 为寄存器赋值后，这个值将一直保存在寄存器上，直到遇到 deassign 为止。当 force 语句应用于寄存器时，寄存器的当前值被 force 语句的值覆盖；当 release 语句应用于寄存器时，寄存器中的当前值将保持不变，直到被赋予新值。

(1) 规则

- 过程连续赋值语句执行后，它会对指定的寄存器（组）强制地维持过程连续赋值直到解除赋值（deassign）语句的执行，或直到另一个过程连续赋值语句又对该寄存器（组）赋值。
- 用 force（强制）语句可以改写已由过程连续赋值语句赋值的寄存器类型变量，直到 release 语句的执行，此时强制赋值被解除而原过程连续赋值对该寄存器类型变量的作用又重新恢复。

(2) 注意

连续赋值语句与过程连续赋值语句尽管很相似，但并不是完全一致。在编写程序时，确认将 assign 写在正确的位置。过程连续赋值语句可以写在声明语句允许出现的位置（在 initial, always, task, function 等内部），而连续赋值语句则必须写在任何 initial 或 always 块之外。

(3) 可综合问题

无论用什么综合工具，过程连续赋值语句是都不能综合的。

(4) 举例说明

```
1  always @ (posedge Clock)
2  begin
```

```
3     Count = Count + 1;      //受下面 always 块控制的计时钟个数的计数器
4   end
5
6   always @ (Reset)          //异步复位
7   begin
8      if(Reset)
9         assign Count = 0;   //当 Reset 为高时，使 Count 为 0，不计数
10     else
11        deassign Count;     //当 Reset 为低时，解除 Count 为 0，
12  end                       //于是下一个时钟的上升沿又重新开始计数。
```

XXII. Verilog指定的块延时

specify 块（指定延时块）用于描述从模块的输入到输出的路径延时以及定时约束，例如信号的建立和保持时间。用指定延时块可以在设计时把模块的信号传输延时与行为或结构分开来进行描述。

关键字：specify、endspecify

(1) 语法

```
specify
   一些参数定义
   设置一些时序检查选项
   设置模块中组合逻辑管脚到管脚的时间延迟
   设置模块中时序逻辑时钟相关的时间延迟
   条件延迟语句，类似条件生成语句
endspecify
```

一般来说，各个 FPGA 厂商会针对自己的根据硬件相关的一些原语编写 specify，这样我们才能够对我们的设计进行时序仿真或者时序分析，因此基本上我们不需要在自己设计的模块中编写 specify。所以在这里仅对模块说明语句进行一些简单介绍，让大家对 specify 有个概念，做个了解即可。

版权声明

版权声明

(1) 南京锆石光电科技有限公司对其发行的或与合作公司共同发行的包括但不限于产品或服务的全部内容拥有版权等知识产权，受法律保护。

(2) 所有产品及资料内容仅供用户学习使用。

(3) 未经本公司书面许可，任何单位及个人不得以任何方式或理由对上述产品、服务、信息、材料的任何部分进行复制、修改、抄录或与其它产品捆绑使用、销售。

(4) 凡侵犯本公司版权等知识产权的，本公司必依法追究其法律责任。

声明单位：南京锆石光电科技有限公司