第3章 Verilog硬件描述语言

CHAPTER 3

硬件描述语言(Hardware Description Language,HDL)是一种国际上流行的描述数字电路和系统的语言,可以在EDA工具的支持下,快速实现设计者的设计意图。

常用的硬件描述语言有Verilog HDL和VHDL两种。本章介绍Verilog语言的语法和使用规则。

3.1 硬件描述语言概述

Verilog HDL是由GDA(Gateway Design Automation)公司的Philip R. Moorby于1983年首创的,最初只设计了一个仿真与验证工具,之后又陆续开发了相关的故障模拟与时序分析工具。1985年Moorby推出商用仿真器Verilog-XL,获得了巨大的成功,从而使得Verilog HDL迅速得到推广应用。1989年CADENCE公司收购了GDA公司,Verilog HDL成为该公司的独家专利。1990年CADENCE公司公开发表了Verilog HDL,成立OVI(Open Verilog International)组织,并推动Verilog HDL的发展。IEEE于1995年制定了Verilog HDL的IEEE标准,即Verilog HDL1364-1995,2001年发布了Verilog HDL1364-2001,目前已发布Verilog HDL 2003。

VHDL是VHSIC Hardware Description Language的缩写,其中VHSIC是Very High Speed Integrated Circuit的缩写,美国国防部为解决项目的多个承包人的信息交换困难和设计维修困难的问题,提出了VHDL构想,由TI、IBM和INTERMETRICS公司完成,并于1987年作为IEEE标准,即IEEE Std 1076-1987[LRM87],后来又进行一些修改,成为新的标准版本,即IEEE Std 1076-1993[LRM93]。

VHDL和Verilog HDL这两种语言的主要功能差别并不大,它们的描述能力也类似,相比于Verilog HDL,只是VHDL的系统描述能力稍强,而Verilog HDL的底层描述能力则更强。

3.1.1 硬件描述语言特点

硬件描述语言(HDL)有不同于其他软件语言的特点:

(1) 功能的灵活性。HDL支持设计者从开关、门级、RTL、行为级等不同抽象层次对电路进行描述,并支持不同抽象层次描述的电路组合为一个电路模型,HDL支持系统的层次化设计,支持元件库和功能模块的可重用设计。用HDL设计数字电路系统是一种贯穿于

设计、仿真和综合的方法。

(2) HDL 支持高层次的设计抽象,可应用于设计复杂的数字电路系统。HDL 设计和传统的原理图输入方法的关系如同高级语言和汇编语言。原理图输入的可控性好、实现效率高,比较直观,但在设计大规模 CPLD/FPGA 时显得很烦琐,有时甚至无法理解。而设计者使用 HDL 进行设计,可以在非常抽象的层次上对电路进行描述,将烦琐的实现细节交由 EDA 工具辅助完成,实现"自顶向下"的层次化设计,缩短开发周期。

(3) HDL 设计可不依赖厂商和器件,移植性好。设计者在设计时,只需在寄存器传输级(RTL 级)对电路系统的功能和结构用 HDL 进行描述,电路系统如需实现在不同器件上,也不用重复设计,只需选择相应 FPGA/CPLD 芯片的综合、布局布线的库函数,由相应的设计工具对设计描述进行重新转换即可。

3.1.2 层次化设计

随着现代控制、通信等电子行业的发展,数字电路复杂度也越来越高。集成电路制造业和 EDA 工具的快速发展,使复杂数字系统的设计实现成为可能。复杂系统的设计必然要使用层次化、结构化的设计方法,其设计思想就是"自顶向下",即"化繁为简,逐步实现",在数字系统的功能指标和端口基础上,将系统分解成多个子模块构成,然后对各个子模块作进一步分解,直到将模块分解到适中的实现复杂度或者可使用的 EDA 元件库中已有的基本元件实现为止,在设计的后期将各子模块组合起来构成一个系统。自顶向下设计示意图如图 3-1 所示。

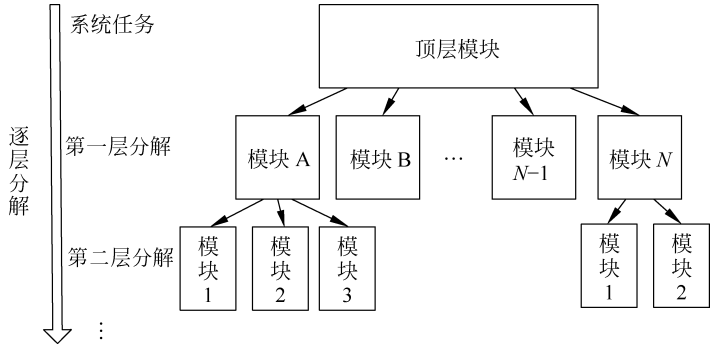


图 3-1 自顶向下设计示意图

本章介绍 Verilog 语言,将按照"先框架,再细节"的模式,即先介绍 Verilog HDL 程序的基本结构,然后介绍常用的语法,最后进行一些数字系统设计练习。

3.2 Verilog HDL 程序的基本结构

Verilog 语言作为一种用于设计数字系统的工具,可以完成以下功能:

(1) 描述数字系统的逻辑功能。

(2) 描述多个数字系统模块之间的连接,组合成为一个系统。

(3) 建立测试激励信号文件,在仿真环境中,对设计好的系统进行调试验证。

根据对电路描述的抽象程度不同,Verilog 语言描述有四个层次的模型类型。

（1）系统级或算法级：这是 Verilog 语言支持的最高抽象级别，设计者对系统行为进行描述，关注算法的实现，不关心具体的硬件实现细节，几乎可以使用 Verilog 语言提供的所有语句。

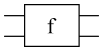
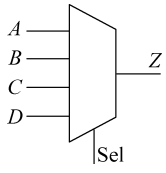
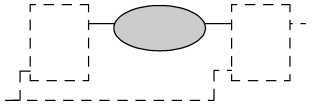
（2）寄存器传输级（RTL）：通过描述模块内部状态转移的情况来表征该逻辑单元的功能，设计者关注数据的处理及其如何在线网上、寄存器间的传递。

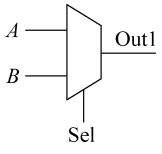
（3）逻辑级：调用已设计好的逻辑级门电路的基本单元（原语），如与门、或门、异或门等，描述逻辑门之间的连接，以实现逻辑功能。

（4）开关级：这是 Verilog 语言支持的最低抽象层次，通过描述器件中的晶体管、存储节点及其互连来设计模块。

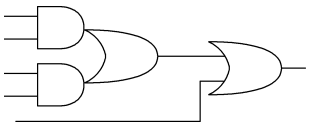
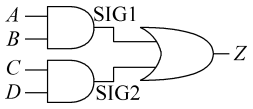
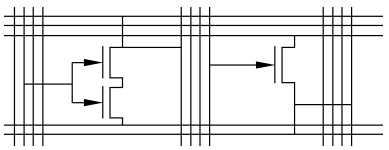
上述四个抽象级别的特性、描述方法和相关的问题在表 3-1 中给出。

表 3-1　Verilog HDL 的抽象等级

模　型	特　　性	描　　述	说　　明
系统级	 功能模型	利用两类过程语句表征： (1) initial 语句：常用于建立行为（仿真）模型，只运行一次； (2) always：用于行为描述和 RTL 级编码，可持续运行。 具体内容见 3.4 节	不是所有的行为模型都是可综合的
		例：always (A or B or C or D or Sel) begin case (Sel) 2`b00: Z = A; 2`b01: Z = B; 2`b10: Z = C; 2`b11: Z = D; default: Z = 1`bx; endcase end	注意 case 语句与 if-else if 语句的区别
RTL 级	 典型的 RTL	为逻辑综合目的，可以描述组合电路的数据运算，也可描述在时钟沿之间组合逻辑的运行。数据流和行为结构连续赋值是数据流模型的基本结构，其中的表达式可利用大多数运算符。连续赋值在每个仿真周期会重新估值	连续赋值中时间延迟将被综合工具忽略

		例：module Mux2_1 (A,B,Sel,Out1);
		output Out1;
		input A,B,Sel;
		wire N1,N2;
		assign N1 = (A & Sel);
		assign N2 = (B & ～Sel);
		assign Out1 = (N1 | N2);
		endmodule
		可用 assign out1=(A & Sel)|(B &～Sel);
			隐含的连续赋值提供更简练的编码

续表

模　型	特　性	描　述	说　明
逻辑级	 库、宏单元	Verilog 语言中，逻辑级直接利用预先定义的门电路原语构筑系统，逻辑级模型含有行为仿真时序信息，但只适应小系统的应用，对多数系统设计而言太详尽并费时	任何逻辑级模块都是可综合的
		例：module AND_OR(A,B,C,D,Z); input A,B,C,D; output Z; wire SIG1,SIG2; and (SIG1,A,B); and (SIG2,C,D); or (Z,SIG1,SIG2); … endmodule;	任何延时规定，综合时将被忽略
开关级		CMOS 开关电路	用 FPGA 实现数字系统，一般不采用开关级描述

一般来说，设计的抽象程度越高，设计的灵活性就越好，和工艺的无关性就越高，随着抽象程度降低，设计的灵活性和工艺的无关性变差，可移植性变差。

3.2.1　模块结构分析

下面通过一个简单的 Verilog HDL 程序来分析 Verilog HDL 程序的基本结构。

例 3-1　设计一个半加器，如图 3-2 所示。

```
module halfadder(A,B,CO,S);
input A,B;
output S,CO;
wire S,CO;
assign S = A ^B;
assign CO = A & B;
endmodule
```

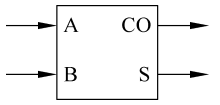


图 3-2　半加器模块图

从例 3-1 可以看到

(1) 程序文件位于关键字 module 和 endmodule 之间，module、endmodule 是关键词；

(2) 每个模块必须由一个模块名进行标识，如 halfadder；

(3) 模块的输入端口是 A 与 B，是相加的两个加数，输出端口是 S 与 CO，S 是相加的和，CO 是向高位的进位；

(4) 第 5 行、第 6 行语句描述了模块的功能；

(5) 模块中的每一条语句都以分号(;)结束，在模块末尾 endmodule 后不加分号。

模块(module)是 Verilog HDL 设计的基本功能单元，模块可以是一个元件，也可以是多个低层次模块的组合。一般而言，模块包含以下信息：端口名的模块声明、I/O 端口声明、各类型变量声明、模块功能说明和模块结尾。模块结构如表 3-2 所示。

表 3-2　Verilog HDL 模块结构

代码行	描　　述		举　　例
1	module 模块名(端口 1,端口 2,端口 3,…,端口 n)；		module halfadder(A,B,CO,S);
2	输入/输出端口声明；		input A,B; output S,CO;
3	wire、reg 等各类型变量声明；		wire S,CO;
4	模块 功能说明	数据流语句(assign)； 低层模块例化； always、initial 语句； 任务(task)和函数(function)	assign S = A ^B; assign CO = A & B;
5	endmodule		endmodule

1. 端口名的模块声明

```
module 模块名(端口 1,端口 2,端口 3,…,端口 n);
```

为便于工程管理，模块命名一般应和其功能相关，如 halfadder(半加器)、adder(加法器)、top(顶层模块)、testbench(测试模块)等。命名的字符应符合 Verilog HDL 对字符串的规定。

端口是模块和外界进行信息交互的接口，有些模块与外界无信息交互，则无端口列表，例如包含了待测模块和激励信号等完整的测试模块，可声明为

```
module testbench();
```

如有信息交互，则注意在括号中各端口应用逗号隔开。对于外界环境来说，模块内部是一个“黑盒子”，对模块的调用(例化)都是通过对端口的操作进行的。

2. I/O 端口声明

所有声明的端口都必须说明其端口类型、位宽等信息。根据信号的方向，端口类型有三类：

(1) 输入端口：声明为 input [width－1:0]端口名 1,端口名 2,…,端口名 n。

(2) 输出端口：声明为 output [width－1:0]端口名 1,端口名 2,…,端口名 n。

(3) 输入/输出端口：声明为 inout [width－1:0]端口名 1,端口名 2,…,端口名 n。

如果输入、输出端口无位宽的说明，系统将默认位宽为 1。

例 3-2

```
input[7:0] data_in;                    //一个名为 data_in 位宽为 8 位的输入数据
output S,CO;                           //两个名为 S 和 CO 位宽为 1 位的输出数据
```

3. 数据类型说明

对端口的信号、模块内部使用变量的数据类型说明详见 3.3.2 节。Verilog HDL 常用两大类数据类型：

(1) 线网类型(net type)：表示 Verilog HDL 结构化元件间的物理连线。它的值由驱动元件的值决定。如果没有驱动元件连接到线网，线网的默认值为高阻值 z。

(2) 寄存器类型(register type)：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。寄存器类型变量的默认值为不确定值 x。

4. 模块功能说明

这是模块中最重要的部分，常有四类方法可以选用以完成模块编辑功能的表述，简要介绍如下：

(1) 用连续赋值语句 assign 进行数据流建模。

例 3-3
```
assign a = b & c;                    //描述了一个二输入的与门
```

(2) 对已定义好的元件进行调用。

例 3-4
```
halfadder u1(a,b,s,co);              //调用半加器,例化名是 u1
```

(3) 用结构说明语句 always、initial、task 和 function 进行行为级描述。

例 3-5
```
always @ (posedge clk)               //描述了一个 D 触发器
  begin
    Q <= d;
  end
```

例 3-6
```
initial                              //产生信号 a,b 的波形
 begin
   a <= 0;
   b <= 0;
  #10 begin
   a <= 1;
   b <= 1;
   end
end
```

例 3-7
```
task writeburst;                     //定义一个任务 writeburst
  input [7:0] wdata;
    …
endtask
    …
writeburst(123);                     //调用任务
```

例 3-8
```
function max(a,b)                    //定义一个函数 max,求出 a,b 两数的最大值
  …
endfunction
assign c = max(data_1,data_2);  //函数的调用,将 data_1,data_2 的最大值赋给 c
```

assign、always、initial、task 和 function 的语法和应用详见 3.4 节。

5. 模块结尾

在每个模块的末尾用 endmodule 结束，其后不加分号。

从上面的分析可以看出，每个 Verilog HDL 模块实现特定的功能，其中 module、模块名和 endmodule 这三部分是模块必需的，其余如端口列表、端口声明、数据类型说明、assign 描

述、always、initial、task 和 function 等结构说明语句根据设计要求选用。

Verilog HDL 模块可分为两种类型：一种是功能模块，用来描述某种电路系统结构和功能，以综合或者提供仿真模型为目的；另一种是测试模块，为功能模块的测试提供信号源激励、输出数据监测。3.6 节中将进行介绍。

3.2.2　模块的实例化

Verilog HDL 支持层次化设计，按“自顶向下”的思路，大型的数字电路的设计可分解成多个小型模块的设计，在每个小模块的设计实现后，用顶层模块调用低层模块的方式实现整体系统功能。模块调用（也称模块实例化）的基本格式为

```
<模块名> <例化名>(<端口列表 >);
```

根据被调用的低层模块端口与在上层模块的连接端口的不同，有两种实例化方法：

（1）按端口顺序连接：低层模块定义时声明的端口顺序与上层模块相应的连接端口顺序保持一致。

格式：

```
模块名例化名(PORT_1,PORT_2,…,PORT_N);
```

（2）按端口名称连接，被调用的低层模块和上层模块是通过端口名称进行连接的。

格式：

```
模块名 例化名(.port_1(PORT_1),.port_2(PORT_2),…,.port_n(PORT_N));
```

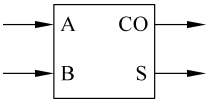
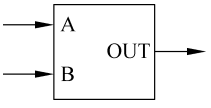
其中，port_1，port_2，…，port_n 为被调用模块设计声明的各个端口；PORT_1，PORT_2，…，PORT_N 为上一层模块调用时对应端口名称。

这种连接端口的顺序可以是任意的，只要保证上层模块的端口名和被调用模块端口的对应即可，如果被调用模块有不需要连接的端口，该端口悬空，则可以将此端口忽略或者写成.port_n()。

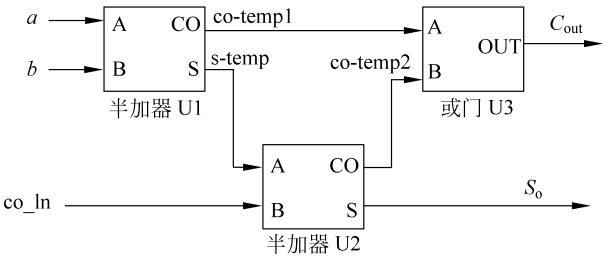
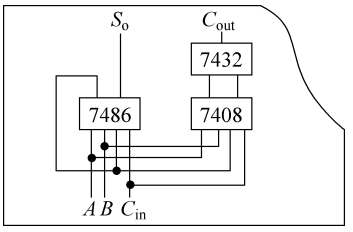
例 3-9　通过调用半加器模块和或门模块来实现一位全加器。

（1）使用电路板的元件装配与 Verilog 模块例化进行类比，清楚理解模块名，例化上层模块的对应端口以及低层模块设计时声明的端口，如表 3-3 所示。

表 3-3　Verilog 语言调用和电路板元件装配的类比关系

Verilog 语言的模块调用	电路板(PCB)元件装配
已设计好的模块： (1) 半加器(halfadder)  (2) 或门  	已制造好的元件： (1) 异或门 7486 7486 (2) 与门 7408 7408 (3) 或门 7432 7432

续表

Verilog 语言的模块调用	电路板(PCB)元件装配
实现的一位全加器：  	装配好的电路板(局部) 

说明：

(1) 半加器模块中标识的 A、B、CO、S 和或门 A、B、OUT 是设计该模块时声明的输入、输出端口，可以类比于实现全加器的异或门、与门和或门电路元件相应的端口。

(2) 一位全加器中的 a，b，co_temp1，s_temp，……是上层模块调用低层模块时的连接端口，可以类比于 PCB 中的上述元件之间相互的连接。

(3) 一位全加器图中的 U1、U2、U3 称为半加器、或门等低层模块的例化名，可以类比于 PCB 中的电路元件，如异或门 7486、与门 7408 和或门 7432。

(4) 在 Verilog 中，逻辑级例化名是可选的(如 U3)，而用户定义的模块例化时必须指定名字(如 U1、U2)。

由于全加器的输出与输入信号有如下关系：

$$S = A \oplus B \oplus \text{co\_in}$$

$$\text{Co\_out} = A \,\&\, B + A \,\&\, \text{co\_in} + B \,\&\, \text{co\_in} = A \,\&\, B + \text{co\_in} \,\&\, (A \oplus B)$$

所以，一位全加器可以调用两个半加器模块和一个或门模块来实现，连接方式以电路板的元件装配来做类比，帮助理解。

(2) 调用设计好的低层模块，搭建上层系统。

半加器设计见例 3-1。

```
module halfadder(A,B,CO,S);
       …
endmodule
```

如果采用第一种按模块端口顺序连接的方法例化模块则全加器写成

```
module fulladder(a,b,co_in,co_out,s);
…
//调用半加器模块两次,例化名分别为 u1,u2
halfadder u1(a,b, co_temp1, s_temp);
halfadder u2(s_temp,co_in, co_temp2, s);
//调用两输入与门,例化名为 u3
and2 u3(co_temp1,co_temp2,co_out);
endmodule
```

如果采用第二种按模块端口名称连接，则

```
halfadder u1(.A(a),.B(b),.CO(co_temp1),.S(s_temp));
halfadder u2(.A(s_temp),.B(co_in),.CO(co_temp2),.S(s));
```

如果没有 co_in 输入，则半加器 u2 的调用可写为

```
halfadder u2(.A(s_temp),.CO(co_temp2),.S(s));
```

或

```
halfadder u2(.A(s_temp),.B(),.CO(co_temp2),.S(s));
```

按端口名称连接方式较按端口顺序连接有以下优点：

(1) 在被调用模块有较多引脚时，根据端口名字进行信号连接，可避免因记错端口顺序而出错。

(2) 在被调用模块的端口顺序发生变化时，只要端口名字含义不变，模块调用就可以不更改调整。

3.3　Verilog HDL 词法、数据类型和运算符

本节讨论 Verilog HDL 的基本词法约定、数据类型和常用运算符，这些是组成 Verilog 语言的基本元素，是后续学习的基础。

3.3.1　词法约定

Verilog HDL 中的基本词法约定与 C 语言类似，可以有空白、注释、分隔符、数字、字符串、标识符和关键字等。

1. 注释

为加强程序的可读性和文档管理，设计程序中应适当地加入注释内容。注释有两种方式：单行注释和多行注释。

(1) 单行注释以“//”开始，只能写在一行中。

(2) 多行注释以“/*”开始，以“*/”结束，注释的内容可以跨越多行。

例 3-10

单行注释：

```
assign c = a + b;                    //c 等于 a,b 的和
```

多行注释：

```
assign c = a + b;       /* c 等于 a,b 的和,本语句可综合成一个加法器,实现加法的组合逻辑 */
```

2. 数字和字符串

数字的表达方式：

```
<位宽>`<进制><数值>
```

说明：

(1) <位宽>：用十进制表示的数字位数，如果默认，则位宽由具体机器系统决定(至少为 32 位)。

(2) <进制>：可以表示二进制(b 或 B)、八进制(o 或 O)、十进制(d 或 D)、十六进制(h 或 H)。默认为十进制。

(3) <数值>：可以是所选进制内的任意有效数字，包括不定值 x 和高阻态 z。当<数值>位宽大于指定的大小时，截去高位。

例 3-11

```
8`b11001100              //位宽为 8 的二进制数，`b 表示二进制
6`O23                    //位宽为 6 的八进制数，`O 表示八进制
`hff23                   //十六进制数，采用机器的默认位宽
  123                    //十进制数 123，采用机器的默认位宽
2`b1101                  //表示的是 2`b01，因为当数值大于指定的大小时，截去高位
4`b110x                  //四位二进制数，最低位为不定值 x
6`o1x                    //位宽为六位的八进制，其值的二进制表示为 6`b001xxx
16`h1z0x                 /* 位宽为 16 位的十六进制数，其值的二进制表示为
                         16`b0001zzzz0000xxxx */
```

在书写过程中，可在数字之间使用下画线“_”对数字进行分隔，以增加数字的可读性，下画线不能作为数字的首字符，下画线在编译阶段将被忽略，如 8`b1100_1100。

字符串是双引号内的字符序列，字符串不能分成多行书写。如"INTERNAL ERROR"和"REACHED—> HERE"等。

3. 标识符

标识符(identifier)用于定义模块名、端口名、连线、信号名等。标识符可以是任意一组字母、数字、符号 $ 和_(下画线)符号的组合，但标识符的第一个字符必须是字母或者下画线，字符数不能多于 1024 个。此外，标识符区分大小写。

例 3-12

```
adder
data_in
state,State              //这两个标识符是不同的
2and,&write              //非法格式
```

4. 空白符

空白符由空格、制表符和换行符定义而成，除了出现在字符串中，Verilog HDL 中的空白符仅用于分隔标识符，在编译阶段被忽略。

5. 关键字

Verilog 语言内部已经使用的词称为关键字或保留字。关键字必须使用小写字母，说明见附表 B。

3.3.2 数据类型

Verilog HDL 中共有 19 种数据类型。Verilog HDL 允许信号具有逻辑值和强度值，以尽可能反映真实硬件电路的工作情况。逻辑值有 0、1、x、z。其中，x 表示未初始化或者未知的逻辑值；z 表示高阻状态。逻辑强度值从最强到最弱分为几种强度等级。

类似于 C 语言，数据类型也有常量和变量之分。在程序运行中，其数值不能改变的量称为常量；数值可以改变的量称为变量。下面对常用的变量数据类型 wire 型、reg 型、memory 型和常量数据类型 parameter 型进行介绍。其他数据类型的详细情况，请查阅 Verilog HDL 的相关手册。

1. 线网型(wire)

wire 型是连线型(net)中最常用的数据类型,它表示硬件单元之间的连接,常用于表示以 assign 为关键字的组合逻辑。

格式:

```
wire [width-1 :0]变量名 1,变量名 2,…,变量名 n;
```

说明:

(1) [width-1:0]指明了变量的位宽,缺省此项时默认变量位宽为 1;

(2) wire 为关键字;

(3) wire 数据默认值是 z;

(4) 模块输入、输出信号的类型默认为 wire 型。

例 3-13

```
wire [7:0] a ,b;          //位宽为 8 的 wire 型变量 a 和 b
wire c;                   //wire 型变量 c,位宽为 1;
wire [3:1] data;          //位宽为 3 的 wire 型变量 data,分别为 data[3],data[2],data[1]
```

2. 寄存器型(reg)

寄存器是数据存储单元的抽象,寄存器中的数据可以保存,直到被赋值语句赋予新的值。reg 型变量只能在 initial 语句和 always 语句中被赋值。reg 的默认值是不定值 x。

格式:

```
reg[width-1:0]变量名 1,变量名 2,…,变量名 n;
```

例 3-14

```
reg [7:0] b,c;            //两个位宽为 8 的寄存器变量 b 和 c
reg a;                    //寄存器变量 a,位宽为 1
reg[3:1] d;               //位宽为 3 的寄存器变量 d,由 d[3],d[2],d[1]组成
```

3. 存储器型(memory)

memory 型数据常用于寄存器文件、ROM 和 RAM 建模等,是寄存器型的二维数组形式,它是将 reg 型变量进行地址扩展而得到的,一般格式为

```
reg[n-1 : 0] 存储器名[N-1 : 0];     //定义位宽为 n,深度为 N 的寄存器组
```

例 3-15

```
reg[7:0] mem[255 : 0];     //每个寄存器位宽为 8,共有 256 个寄存器的存储器组
```

对一组存储单元进行读写,必须指定该单元的地址。例如,对例 3-15 的 mem 寄存器的第 200 个存储单元进行读写操作,格式为

```
men[200] = 0;              //对存储器 mem 的第 200 个存储单元赋值 0
```

要注意的是,虽然 memory 型数据是将 reg 型变量进行地址扩展而得到的,但是 memory 和 reg 型数据有很大区别。

例 3-16

```
reg mem [N－1 : 0];          //N 个一位的寄存器组 men
reg [N－1:0] a;              //一个 N 位的寄存器变量 a
```

4. 参数型(parameter)

在 Verilog HDL 中可以使用 parameter 为关键词,指定一个标识符(即名字)来代表一个常量,参数的定义常用在信号位宽定义、延迟时间定义等位置,增加程序的可读性,方便程序的更改。

格式:

```
parameter 标识符 1 = 表达式 1,标识符 2 = 表达式 2,…,标识符 n = 表达式 n;
```

表达式可以是常数,也可以是以前定义过的标识符。

例 3-17

```
parameter width = 8;                    //定义了一个常数参数
input[width－1:0] data_in;              //输入信号 data_in 的位宽为 8

parameter a = 1,b = 3;                  //定义了两个常数参数
parameter c = a + b;                    //c 的值是前面定义的 a,b 的和
```

3.3.3 运算符

运算符按照功能分为表 3-4 所列的类型,表 3-4 也总结了运算符的优先级关系。

表 3-4 运算符的分类和优先级

分 类	运 算	运 算 符	操作数	优 先 级
逻辑/按位运算符	双目运算符(或,与)	|, &, ^	2	最高
	单目运算符(非)	～	1	
算术运算符	乘,除,取模	*, /, %	2	
	加,减	+, －	2	
移位运算符	移位	<<, >>	2	
关系运算符	关系	<, <=, >, >=	2	
等价运算符	等价	==, !=, ===, !==	2	
按位/缩减运算符	缩减	&, ～&	1	
		^, ～^	1	
		|, ～|	1	
逻辑运算符	逻辑	&&	2	
		||	2	
		!	1	
条件运算符	条件	?␣:␣	3	最低
拼接运算符	拼接	{,} {,{ }}	≥2	

根据参加运算的操作数数目,运算符可分为:

(1) 单目运算符:对一个操作数进行操作的运算符,例如 clock = ～clock;

(2) 双目运算符:对两个操作数进行运算的运算符,例如 $a = b \& c$;

(3) 三目运算符：对三个操作数进行运算的运算符，例如

```
D_out = condition ? D_in1 : D_in2.
```

下面简要介绍常用的运算符。

1. 算术运算符

算术运算符有加法(+)、减法(-)、乘法(*)、除法(/)和取模(%)。

例 3-18　设 a= 4`b0101，b = 4`b0010。

```
a + b                          //a 和 b 相加，等于 4`b0111
a * b                          //a 和 b 相乘，等于 4`b1010
a / b                          //a 除以 b，等于 4`b0010，余数部分舍弃，取整
a % b                          //a 对 b 取模，即求 a，b 相除的余数部分，结果等于 1
```

在算术运算时，如果有一个操作数为不定值 x，则运算结果全部为不定值 x。

2. 逻辑运算符

逻辑运算符有逻辑与(&&)、逻辑或(‖)、逻辑非(!)。

说明：

(1) && 和 ‖ 是双目运算符，！是单目运算符。

(2) 逻辑运算符的计算结果是逻辑假(0)、逻辑真(1)、不确定(x)3 种情况 1 位的值。

(3) 当操作数为具体数值时：①操作数不等于 0，则等价于逻辑真(1)；②操作数等于 0，则等价于逻辑假(0)；③操作数的任何一位为不确定值 x 或者高阻态 z，则等价于不确定值 x。

例 3-19　设 a=2，b=0。

```
a && b                         //等于 0，相当于(逻辑 1 && 逻辑 0)
a ‖ b                          //等于 1，相当于(逻辑 1 ‖ 逻辑 0)
!a                             //等于 0，相当于逻辑 1 取反
(a == 3)&&(b == 0)             /* 等于 0，相当于两个表达式是否成立(为真)，即如果
                                  a = 3 成立，则(a == 3)为逻辑 1，否则为逻辑 0 */
x && a                         //等于 x，相当于(x && 逻辑 1)
```

3. 按位运算符

按位运算符有取反(~)、与(&)、或(|)、异或(^)和同或(~^，^~)。

说明：

(1) 取反运算是单目运算符，其余是双目运算符。

(2) 按位运算对操作数中的每一位进行按位操作，如果两个数的位宽不相同，系统先将两个操作数右对齐，较短的操作数左端补 0，然后再按位运算。

(3) 注意按位运算和逻辑运算的差别，逻辑运算结果是一个 1 位的逻辑值，按位运算产生一个与较长位宽操作数等宽的数值。

例 3-20　设 a= 4`b0011，b=4`b1010，c=3`b011，d = 4`b11x0。

```
~a                             //按位取反，结果等于 4`b1100
b & c                          //按位与运算，结果等于 4`b0010
a ^~ d                         //按位同或运算，结果等于 4`b00x0
a & b                          //按位与运算，结果等于 4`b0010
a && b                         //逻辑与运算，等价于 1 && 1，结果等于 1
```

4. 关系运算符

关系运算符包括大于(>)、小于(<)、大于或等于(>=)以及小于或等于(<=)。

在运算中：①如果表达式成立，运算结果是真(1)；②如果表达式不成立，运算结果为假(0)；③如果操作数中某一位是不确定的，则表达式的结果是 x。

例 3-21 设 a = 4`b1010，b=4`b0001，c= 4`b1xz0。

```
a > b                    //结果等于逻辑 1
a < b                    //结果等于逻辑 0
a >= b                   //结果等于逻辑 1
a <= c                   //结果等于逻辑值 x
13 - a > b               /* 由于算术运算优先级较高，先进行 13 - a 的计算，得到 3，再和
                            b 进行比较，结果等于逻辑值 1 */
13 - (a > b)             /* 由于括号表明了关系运算的优先级，a > b 成立，结果是真值为
                            1，所以算术结果等于 12 */
```

5. 等式运算符

等式运算符包括 4 种：逻辑等(==)，逻辑不等(!=)，case 等(===)，case 不等(!==)。

说明：

(1) 如果两个操作数位宽不等，则先对两个操作数右对齐，用 0 填充较短数的左边。

(2) 逻辑等(==)和逻辑不等(!=)中，如果两操作数中某一位是不确定的，则返回值是 x；如果两个数相同，则返回逻辑 1；如果不相同，则返回逻辑 0。

例 3-22 设 a = 4`b1010，b= 4`b1100，d = 4`b101x。

```
a == b;                  //逻辑等，结果为逻辑值 0
a != b                   //结果为逻辑 1
a == d                   //结果为逻辑 x
```

(3) case 等(===)、case 不等(!==)与逻辑等式运算符不同，在对两个操作数进行逐位比较时，即使有 x，z 位，也要进行精确比较，只有在二者完全相等的情况下结果为 1，否则为 0，case 等式运算符的结果不可能为 x。

例 3-23 设 a = 4`b1010，b = 4`b1xzz，c = 4`b1xzz，d = 4`b1xzx。

```
a === b                  //结果为逻辑值 0
b === c                  //结果为逻辑值 1(两个数每一位都相同，包括 x,z)
b === d                  //结果为逻辑值 0 (最低的一位不同)
b !== d                  //结果为逻辑值 1
```

6. 缩减运算符

缩减运算符包括缩减与(&)、缩减与非(~&)、缩减或(|)、缩减或非(~|)、缩减异或(^)和缩减同或(~^)。

这类操作符将对操作数由左向右进行操作，它们的运算规则和按位操作符相同。注意，缩减运算符只有一个操作数，按位运算符有两个操作数。

例 3-24 设 a= 4`b1010。

```
&a                       //结果是 1 & 0 & 1 & 0 = 0
|a                       //结果是 1 | 0 | 1 | 0 = 1
```

```
^a                                    //结果是 1 ^ 0 ^ 1 ^ 0 = 0
```

可以看到，缩减异或、缩减同或可以产生一个向量的奇偶校验位。

7. 移位运算符

移位运算符有右移(>>)、左移(<<)。右移(>>)、左移(<<)将操作数向右、向左移动指定的位数，空出的位置用 0 补足。

例 3-25　设 a = 4`b1010。

```
b = a >> 1;                           //右移 1 位,结果是 b = 4`b0101
b = a << 2;                           //左移 2 位,结果是 b = 4`b1000
```

使用移位运算符可以将乘法转换成移位相加来完成，还可以进行移位寄存器的移位操作等，这在具体设计中有很多应用。

8. 拼接运算符

位拼接运算符{}可以将两个或多个操作数的某些位拼接起来成为一个操作数。进行拼接的每个操作数必须是确定位宽的，因为系统进行拼接时必须要确定拼接结果的位宽。

拼接运算时，将需要拼接的操作数按照顺序罗列出来，其间用逗号隔开，操作数的类型可以是线网变量、寄存器、向量线网或寄存器、有确定位宽的常数等。

例 3-26　设 a = 1`b1，b = 3`b101，c = 4`b1010。

```
X = {a,b,c}                           //结果是 8`b11011010
Y = {a,b,2`b01}                       //结果是 6`b110101
Z = {b[1:0],c[1],c[0]}                //结果是 4`b0110
```

位拼接可以使用重复操作、嵌套的方式来简化表达式。

例 3-27

```
{1`b0,3{1`b1}} = 4`b0111
{1`b0,{3{2`b01}} = 7`b0010101
```

9. 条件运算符

条件运算符是一个三目运算符，格式为

```
条件表达式?  表达式 1 : 表达式 2
```

判断过程是首先计算条件表达式：①如果条件表达式为真，则计算表达式 1 的值；②如果条件表达式为假，则计算表达式 2 的值；③如果表达式为不确定 x，且表达式 1 和表达式 2 的值不相等，则输出结果为不确定值 x。

例 3-28　`assign c = a > b? a: b  //如果 a 大于 b(即(a > b)为真),c = a; 反之,c = b`

3.4　Verilog HDL 行为语句

本节重点介绍 Verilog 语言的常用行为级建模编程语句。

(1) 赋值语句，包括过程赋值和连续赋值，注意理解阻塞赋值和非阻塞赋值；

(2) 顺序块和并行块语句；

(3) 过程模块的结构说明语句，即 always 语句、initial 语句、task 语句、function 语句；

(4) 条件语句：if-else 语句和 case 语句；

(5) 循环语句：for，forever，repeat，while；

(6) 命令语句：系统任务和系统函数，以及编译预处理命令。

3.4.1 赋值语句

Verilog HDL 赋值语句中，赋值符号左边是赋值目标，右边是表达式。常用赋值方式有过程赋值和连续赋值两种。

过程赋值语句的更新对象是寄存器、整数、实数等，这些类型变量在被赋值后，可以保持不变，直到赋值进程又被触发，变量才被赋予新值。过程赋值常出现在 initial 和 always 语句内。过程赋值方式有两种：阻塞赋值和非阻塞赋值。它们在功能和特点上有很大不同。

连续赋值语句中，任何一个操作数的变化都会重新计算赋值表达式，重新进行赋值。

1. 过程赋值——阻塞赋值

阻塞赋值操作符用“=”表示。

例 3-29

```
always @(posedge clk)          //当时钟上升沿到来时，触发 always 块执行
begin
  a = b + 1;
  c = a;
end
```

当上面的 always 块被触发执行时，先求解 b+1 的值，将结果赋给 a；然后再执行将 a 的值赋给 c 的操作；最后 a 和 c 的值都是 b+1。

综合出的参考电路结构如图 3-3 所示。

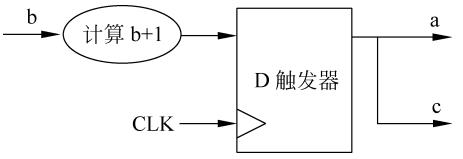


图 3-3 综合的参考电路结构

可以看到：

(1) 阻塞赋值的执行期间不允许其他 Verilog HDL 语句的执行干扰，必须是阻塞赋值完成后，才进行下一条语句的执行。

(2) 赋值一旦完成，等号左边的变量值立刻发生变化(如例 3-29 的 a 和 c)。

(3) 使用阻塞赋值可能会得到意想不到的结果，如例 3-29，可能设计者希望得到两个触发器，现在却只得到了一个。

2. 过程赋值——非阻塞赋值

非阻塞赋值操作符用“<=”表示。

例 3-30

```
always @(posedge clk)          //当时钟上升沿到来时，触发 always 块执行
  begin
    a <= b + 1;                //语句 1
  c <= a;                      //语句 2
end
```

执行时，根据时钟上升沿来到时，采样到 a 和 b 的值，并计算 b+1 的值，在 always 块结束之前，将 b+1 和 a 的值分别赋给 a 和 c，最后 a 的值是 b+1，c 的值是时钟上升沿采样的 a 的值。

综合出的参考电路结构如图 3-4 所示。

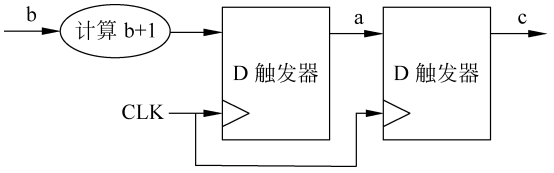


图 3-4　综合的参考电路结构

可以看到：

(1) 非阻塞赋值的符号(<=)与小于或等于运算符相同，但是这二者的意义是完全不一样的，在使用中应根据使用环境、相关语句含义进行区分。

(2) 非阻塞赋值在赋值开始时计算表达式右边的值，到了本次仿真周期结束时才更新被赋值变量(即赋值不立刻生效)。非阻塞赋值允许块中其他语句的同时执行。

(3) 在同一个顺序块中，非阻塞赋值表达式的书写顺序不影响赋值的结果。

3. 连续赋值语句

连续赋值常用于数据流行为建模。在连续赋值中，常以 assign 为关键词。

assign 赋值语句执行将数值赋给线网，可以完成逻辑级描述，也可从更高的抽象角度对线网电路进行描述，多用于组合逻辑电路的描述。连续赋值操作符是“=”。

语句格式：

```
assign 赋值目标线网 = 表达式;
```

例 3-31

```
assign a = b | c;                                  //描述的是两输入的或门
assign {c,sum[3:0]} = a[3:0] + b[3:0] + c_in; //描述一个加法器
assign c = max(a,b);                               //调用了求最大值的函数,将函数返回值赋给 c
```

说明：

(1) 式子左边的“赋值目标线网”只能是线网变量，而不是寄存器变量。

(2) 式子右边表达式的操作数可以是线网，可以是寄存器，也可以是函数。

(3) 一旦等式右边任何一个操作数发生变化，右边的表达式就会立刻被重新计算，再进行一次新的赋值。

(4) assign 可以使用条件运算符进行条件判断后赋值，例如：

```
assign data_out = sel? a : b;      /* 如果 sel 等于 1,将 a 赋给 data_out,否则将 b 赋给 data_out,这
                                      实现了一个二选一的选择器电路描述 */
```

4. 过程赋值与连续赋值的区别

在 Verilog HDL 程序编写过程中，要注意过程赋值与连续赋值的区别，避免出现问题。表 3-5 列出了二者的区别。

表 3-5　过程赋值与连续赋值的区别

过 程 赋 值	连 续 赋 值
无关键字(过程连续赋值除外)	关键字 assign
用“=”和“<=”赋值	只能用“=”赋值
只能出现在 initial 和 always 语句中	不能出现在 initial 和 always 语句中
用于驱动寄存器	用于驱动网线

3.4.2　顺序块和并行块语句

Verilog HDL 中使用块语句将多条语句组合成一条复合语句。块语句分为顺序块语句和并行块语句。

1. 顺序块

顺序块中的语句按书写顺序执行,由 begin-end 标识。顺序块的格式为

```
begin
  执行语句 1;
  执行语句 2;
  ⋮
end
```

或

```
begin 块名
  块内变量、参数定义;
  执行语句 1;
  执行语句 2;
  ⋮
end
```

说明:

(1) 块名是可选的,是一个块的标识名。

(2) 块内可以根据需要定义变量,声明参数,但这些内容只能在块内使用,类似于“局部变量”和“局部声明”。

(3) 顺序块内的语句是按照语句的书写顺序执行的。在仿真开始时执行第一条语句,后面语句开始的执行时间和前一个语句的执行时间是相关的,如有延时,延时也是相对于前一个语句执行完的仿真时间而言的。当块内的最后一条语句执行完,才跳出该顺序块。

2. 并行块

并行块中的语句并行执行,由 fork-join 标识。并行块的格式如下:

```
fork
  执行语句 1;
  执行语句 2;
  ⋮
join
```

或

```
fork 块名
  块内变量、参数定义语句;
  执行语句 1;
  执行语句 2;
  ⋮
join
```

说明：

(1) 块名、块内声明语句的理解与顺序块相同。

(2) 并行块的语句是同时执行的，可以将每一条语句看成一个独立的进程，语句的书写顺序不会影响语句的执行结果。应注意避免并行块中的多条语句对同一个变量进行改变，否则可能会引起竞争。

(3) 块内每条语句的起始执行时间是相同的，当块中的执行时间最长的语句执行完，跳出并行块的执行。

3. 顺序块和并行块程序执行过程的区别

下面通过例子来说明顺序块和并行块语句的执行过程。

例 3-32

顺序块：

```
begin
s = 0;
#2 s = 1;
#2 s = 0;
#3 s = 1;
#1 s = 0;
end
```

并行块：

```
fork
s = 0;
 #2 s = 1;
 #4 s = 0;
 #7 s = 1;
 #8 s = 0;
join
```

在仿真中，“#”后的数字表示仿真时间，在顺序块中是通过顺序累加得到当前仿真时间，而并行块中是用绝对时间表示仿真时间。例 3-32 中的顺序块和并行块完成了对变量 s 相同的赋值过程，假设块语句都从仿真时刻 0 开始执行，实现的赋值过程如表 3-6 所示。

以上两个程序都产生了如图 3-5 所示的波形。

表 3-6　顺序块和并行块执行结果

仿真时刻	s 的数值
0	0
2	1
4	0
7	1
8	0

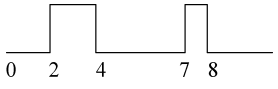


图 3-5　仿真波形

3.4.3　结构说明语句

Verilog 语言中的过程模块常使用四种结构的说明语句：always 语句、initial 语句、task 语句和 function 语句。

一个模块中可以有多个 initial 和 always 语句。每个 initial 块和 always 块代表一个独

立工作的单元，不管这两种语句在模块中书写的先后顺序，在仿真一开始就同时开始执行。initial 语句只能执行一次，always 语句只要满足触发条件，就不断地重复执行，直到仿真过程结束。task 语句和 function 语句定义后，可以在模块的多处进行调用。

在较大的系统中，常需要在不同的地方实现相似的功能、操作，为了程序的简洁、易懂，在 Verilog 语言中，可以用任务和函数的形式描述这些相似的功能模块，上一层模块可以根据需要对任务和函数进行调用。任务和函数都必须在模块内部进行定义和调用，其作用范围只局限于定义它们的模块内部。熟练编写、使用任务和函数的能力是设计大型系统的基础。

下面对这四种语句分别进行介绍。

1. initial 语句

语句格式：

```
initial
 begin
   语句 1;
   语句 2;
    ⋮
 end
```

说明：

(1) 一个模块中可以包含多个 initial 语句，所有的 initial 语句都同时从 0 时刻开始并行执行，但是只能执行一次。

(2) initial 语句常用于测试文本中对信号的初始化，生成输入仿真波形、监测信号变化等。

(3) 也可以使用 fork…join 对语句进行组合。

例 3-33

```
reg [1:0] a,b;
reg c;
initial                          //第一个 initial 语句
  begin
   a = 1; b = 0;
   #10 begin a = 2; b = 3; end   //10 个单位时间后,对 a,b 进行再次赋值
   #10 begin a = 0; b = 2; end   //20 个单位时间后,对 a,b 进行再次赋值
  end
initial                  //第二个 initial 语句,只有一条执行语句,不需要顺序块语句 begin-end
  c = 1;
```

执行结果如表 3-7 所示。

表 3-7 仿真执行结果

仿真时刻(以单位时间为基准)	变 量 值
0	a=1，b=0，c = 1(信号初始化不占用仿真时间)
10	a=2，b=3，c = 1
20	a=0，b=2，c = 1

2. always 语句

语句格式：

```
always @ <触发事件> 语句或语句组;
```

说明：

(1) always 的触发事件可以是控制信号变化、时钟边沿跳变等。always 块的触发控制信号可以是一个，也可以是多个，其间用 or 连接，例如：

```
always @ (posedge clock or posedge reset)
/ * 当 clock 上升沿或者 reset 上升沿时，触发 always 模块执行块中的语句 * /
always @ (a or b or c or d)
/ * 当 a,b,c,d 四个信号的任意一个电平发生变化时，触发 always 模块 * /
```

(2) 只要 always 的触发事件产生一次，always 就会执行一次。在整个仿真过程中，如果触发事件不断产生，则 always 中的语句将被反复执行。

例 3-34

```
reg q;
always @ (posedge clock)        //当时钟上升沿到来时，将 d 的值赋给 q
  q <= d;
```

(3) 一个模块中可以有多个 always 语句，每个 always 语句只要有相应的触发事件产生，对应的语句就执行，这与各个 always 语句书写的前后顺序无关。

例 3-35

```
always @ (posedge clk)          //第一个 always 语句，当时钟上升沿来到时触发执行
  if(rst)
   counter <= 4`b0000;          //当复位信号等于 1，计数器 counter 置 0
  else
   counter <= counter + 1;      //当复位信号等于 0，对输入时钟上升沿进行计数
always @ (counter)              //第二个 always 语句，只要 counter 发生变化就触发执行
    $display ("the counter is = %d",counter);      //显示计数器 counter 的值
    …
```

通过使用两个 always 块，程序实现了一个对时钟计数到 16 的有同步复位控制的计数器及其仿真器显示。这两个 always 模块只要其触发条件成立，就执行各自相关的操作，而两个 always 块的书写顺序上的先后，不会影响执行结果，体现了 Verilog 语言描述的“并行性”。

3. 任务(task)

语句格式：

```
task 任务名;
    <输入输出端口声明>;
    <任务中数据类型声明>;
    语句 1;
    语句 2;
```

```
    …
endtask
```

说明：

(1) 任务定义在关键字 task 和 endtask 之间，任务中可以包括延迟、时序控制和事件触发等时间控制语句。任务只有在被调用时才执行。

(2) 任务调用与变量的传递。

格式：

```
任务名(端口 1,端口 2,…,端口 n);
```

任务的输入/输出都是局部的寄存器，执行完之后才返回结果。

(3) 当任务被调用时，任务被激活。同时，一个任务可以调用其他任务或函数。

例 3-36 通过定义，调用任务，将数据送入 fifo 中。

```
module top;
  …
//例化一个 fifo
fifoctlr_cc u1 (.clock_in(clockin), .read_enable_in(read_enable), …);
/* 以下定义第一个任务 writeburst,功能是完成写入一个 8 位的数据到 fifo 中的任务 */
task writeburst;                    //定义任务 writeburst
   input [7:0] wdata;               //此 task 中局部信号
   begin
     always @(posedge clockin)
       begin
         write_enable = #2 1;       //将写控制信号置 1,有效
         write_data = #2 wdata;     //接收 wdata 数据,将其传送到 fifo 的 write_data 端口
       end
   end
endtask

initial
 begin
      writeburst(128);              //调用任务 writeburst,将数值 128 送入 fifo
   …
 end
endmodule
```

(4) 任务也可以没有参数的输入，只完成执行操作。

例 3-37 将输入信号的与和或的结果分别输出。

```
module result(data_in1,data_in2,data_out1,data_out2);
input data_in1,data_in2;
output data_out1,data_out2;
reg data_out1,data_out2;
task example;                       //定义任务 example
begin
    data_out1 <= data_in1 & data_in2;
    data_out2 <= data_in1 | data_in2;
 end
 endtask
always @ (data_in1 or data_in2)
```

```
    example;                          //调用任务,任务没有输入参数
endmodule
```

4. 函数(function)

语句格式:

```
function <返回值的位宽,类型说明> 函数名;
  <输入端口与类型说明>;
  <局部变量说明>;
  begin
   语句;
  end
endfunction
```

说明:

(1) 函数定义在关键字 function 和 endfunction 之间,函数的目的是返回一个用于表达式的值。

例 3-38

```
function[3:0] max;                    //函数名为 max,作用: 返回两个数中的最大值
input[3:0] a,b;
 begin
   if (a > b)
    max = a;
   else
    max = b;
 end
endfunction
```

函数的定义使得在模块中定义了一个和函数同名、位宽相同的寄存器类型变量,如例 3-38 中的 max。如果函数的返回值的位宽缺省时,这个变量位宽为 1。

(2) 函数的调用是以函数名同名的寄存器变量作为表达式的操作数来进行调用,并根据函数输入数据的要求,携带、传送数据。如对例 3-38 中函数的调用可写成

```
c = max(10,5);                        //运行结果 c = 10
```

5. 任务和函数的比较

(1) 任务和函数的定义和引用都应位于模块内部,而不是一个独立的模块。

(2) 函数不能启动任务,任务可以调用函数或其他任务。

(3) 任务可以没有输入变量或有任意类型的 I/O 变量,而函数允许有输入变量且至少有一个,输出则由函数名自身担当。

(4) 函数通过函数名返回一个值,任务名本身没有值,只是实现某种操作,传递数值通过 I/O 端口实现。

(5) 函数还可以出现在连续赋值语句的右端表达式中。

(6) 任务可以用于组合电路、时序电路的描述;函数只能用于组合电路的描述,函数的定义不能包含任何的时间控制语句。

3.4.4 条件语句

1. if 语句

if 语句和它的变化形式是条件语句的常见形式，用于判断给定的条件是否满足(为真)，根据判定结果，执行相应的操作。

语句格式：

```
(1) if(表达式)
      <语句>;
(2) if(表达式)
      <语句 1>;
    else
      <语句 2>;
(3) if(表达式 1)
      <语句 1>;
    else if(表达式 2)
      <语句 2>;
              ⋮
    else if(表达式 n)
      <语句 n>;
    else
      <语句 n + 1>;
```

说明：

(1) if 后面的表达式，可以是逻辑表达式或关系表达式。如果表达式的值是真(1)，执行紧接在后的语句；如果是假(0)，执行 else 后的语句。

(2) if 后的表达式还可以是操作数。如果操作数是 0、x、z，等价于逻辑假，反之为逻辑真。下式是一种表达式的简化写法：

```
if(reset)  等价于  if(reset == 1)
```

(3) else 不能作为单独的语句使用，必须与 if 语句配对使用。

(4) 如果 if 和 else 后有多个执行语句，可以用 begin-end 块将其整合在一起。

例 3-39

```
if(a > b)
    begin
      data_out1 <= a;
      data_out2 <= b;
    end
else
    begin
      data_out1 <= b;
      data_out2 <= a;
    end
```

(5) if 语句可以嵌套使用,但是在嵌套使用过程中,应注意与 if 配对的 else 语句。通常,else 与最近的 if 语句配对。

例 3-40

```
if(a > b)                          //第一个 if 语句
  if ( c )                         //第二个 if 语句
    data_out <= c + 1;
  else                             //第二个 if 语句的配对 else
    data_out <= a + 1;
  else                             //第一个 if 语句的配对 else
    data_out <= b;
```

特别是当 if-else 数目不一致、if 嵌套使用等情况时,可使用 begin-end 块,如同算术表达式的括号一样,确定 if-else 的配对关系,避免逻辑描述错误。

```
if(a > b)
  begin
   if( )
     执行语句;
   else
     执行语句;
  end
else   …
```

(6) 如果不正确使用 else,可能会生成不需要的锁存器。

例 3-41

```
always @ (a or b)                  //当 a 和 b 的数值发生变化时,触发此 always 块
begin
 if(a)
  data_out <= a;
end
```

如果设计者的设计意图是,当 a 不为 0 时,data_out 赋值为 a,否则赋值为 b。但例 3-41 的描述是:在 a 等于 0 时,没有 else 语句描述分支的操作,data_out 的值将锁定为 a。

(7) if-else 表达了一个条件选择的设计意图,它与条件操作符有重要的区别:

条件操作符可以出现在一个表达式中,而这个表达式可以使用在过程赋值中或者连续赋值中,可进行行为建模,也可以进行逻辑级建模。

if-else 只能出现在 always、initial 块语句,或函数、任务中,一般只能在行为建模中使用。

2. case 语句

if-else 语句提供了选择操作,如果选项数目较多,使用会很不方便,当 if 条件是同一个表达式时,而 case 是一种多分支选择语句,使用它就很简便。

语句格式:

```
case(控制表达式)
```

```
    分支表达式 1:   语句 1;
    分支表达式 2:   语句 2;
        ⋮
    分支表达式 n:   语句 n;
    default:   默认语句;
endcase
```

控制表达式常表示为控制信号的某些位，分支表达式表述的是这些控制信号的具体状态值。语句执行时，先计算 case 后的控制表达式，然后将得到的值与后面的分支表达式的值进行比较，当控制信号的值与某分支表达式的值相等时，执行该分支表达式后的语句；如果没有匹配的分支表达式，则执行 default 后的默认语句。

case 语句的作用类似于多路选择器。用 case 语句可以容易、简洁地实现四选一、八选一、十六选一等电路描述。

例 3-42 用 case 语句实现四选一电路。

```
module mux4(clk,rst,data_in1,data_in2, data_in3,data_in4,select,data_out);
input[3:0] data_in1,data_in2;
input[3:0] data_in3,data_in4;
input[1:0] select;
input clk,rst;
output[3:0] data_out;
reg [3:0] data_out;
always @ (posedge clk)
   if(rst)
  data_out <= 4`b0000;
   else
   case(select)
    2`b00 : data_out <= data_in1;  //如果 select = 2`b00,输出 data_in1 的值
    2`b01 : data_out <= data_in2;  //如果 select = 2`b01,输出 data_in2 的值
    2`b10 : data_out <= data_in3;  //如果 select = 2`b10,输出 data_in3 的值
    2`b11 : data_out <= data_in4;  //如果 select = 2`b11,输出 data_in4 的值
    default : $display("the control signal is invalid");
   endcase
endmodule
```

说明：

(1) 每个分支表达式的值必须是互不相同的，否则，将产生同一个控制表达式的值有多种执行语句，从而产生矛盾。

(2) case 语句执行中，逐位对控制表达式的值和分支表达式的值进行比较，每一位的值可能是 0、1、x、z。如果二者的位宽不一致，将用 0 加在数值左端的方法调整使二者位宽相等。如果多个不同的状态值有相同的执行语句，可以用逗号将各个状态隔开。

例 3-43

```
case(select)
  2`b00 : data_out <= data_in1;
```

```
  2`b01 : data_out <= data_in2;
  2`b10 : data_out <= data_in3;
  2`b11 : data_out <= data_in4;
  //如果 select 中有不确定位 x,输出值是 x
  2`b0x,2`bx0,2`b1x,2`bx1,2`bxx: data_out <= 4`bxxxx;
  //如果 select 中有高阻态位,输出值是 z
  2`b0z,2`bz0,2`b1z,2`bz1,2`bzz: data_out <= 4`bzzzz;
  default : $display("the control signal is invalid");
 endcase
```

(3) default 语句是可选的,但是一个 case 语句中只能有一个 default 选项。一般而言,建议在 case 语句中加入 default 分支,以避免因分支表达式未能对控制表达式的所有状态进行穷举而生成不需要的锁存器。如例 3-43 的四选一行为描述中没有 default 选项,如果有效控制信号中出现了 x 或 z 数值,输出端将不会产生相应的变化,原数据锁存。

(4) case 语句为表达式值存在不定值 x 和高阻值 z 位的情况提供了逐位比较和执行对应语句的操作。

case 语句还有两种变形,关键字为 casez 和 casex。这可以对比较中的不关心的值进行忽略。其中,casex 将条件表达式或者分支表达式中的不定态 x 的位视为不关心的位,casez 则将高阻态 z 视为不关心的位。这样,设计者就可以根据具体要求,只对信号的某些位进行比较。具体请查阅 Verilog HDL 相关手册。

5.1.1 节将分析 if-else 语句和 case 语句在使用上带来的不同的综合结果。

3.4.5　循环语句

循环语句只能在 initial、always 块中使用。Verilog HDL 中有 for、while、forever 和 repeat 四种循环语句,它们的语法规则类似于 C 语言的循环语句。下面分别进行介绍。

1. for 语句

语句格式:

```
for(表达式 1; 表达式 2; 表达式 3)语句;
```

说明:

(1) 表达式 1 是初始条件表达式,表达式 2 是循环终止条件,表达式 3 是改变循环控制变量的赋值语句。

(2) 语句执行过程:

步骤 1:求解表达式 1;

步骤 2:求解表达式 2,如果其值为真(非 0),执行 for 语句中内嵌的语句组,然后执行步骤 3;如果为假(等于 0),结束循环,执行 for 语句后的操作;

步骤 3:求解表达式 3,得到新的循环控制变量的值,转到步骤 2 继续执行。

例 3-44　用 for 语句对存储器组进行初始化。

```
reg[7:0] my_memory[511:0];        //定义一个寄存器型数组,共有 512 个变量,每个变量的位宽为 8
integer i;
initial
```

```
  begin
     for(i = 0; i < 512; i = i + 1)      //把数组中所有变量赋 0 值
      my_memory[i] <= 8'b0;
   end
```

2. while 语句

语句格式：

```
while(条件表达式)语句;
```

说明：

语句执行过程先求解条件表达式的值，如果值为真(等于 1)，执行内嵌的执行语句(组)，否则结束循环。如果一开始就不满足条件表达式，则循环一次都不执行。

例 3-45 用 while 语句求从 1 加到 100 的值，加法完成后打印结果。

```
module count(clk, data_out);
input clk;
output[12:0] data_out;
reg [12:0] data_out;
integer j;

initial                              //data_out 和 j 赋初值为 0
begin
  data_out = 0;
  j = 0;
end

always @ (posedge clk)
begin
    while(j <= 100)                  /* 如果 j 小于或等于 100,则执行循环内容,执行 100 次后,即
                                        j 大于 100 后,跳出循环 */
      begin
       data_out = data_out + j;
       j = j + 1;
      end
  $display ("the sum is %d,j = %d",data_out,j);
end
endmodule
```

在内嵌语句中应该包含使循环控制变量变化的语句，如例 3-45 的 j=j+1；如果没有此类语句，循环控制变量的值始终不变，循环将永不结束。

3. forever 语句

语句格式：

```
forever 语句;
```

说明：

forever 表示永久循环，无条件地无限次执行其后的语句，相当于 while(1)，直到遇到系统任务 $finish 或 $stop，如果需要从循环中退出，可以使用 disable。

forever 不能独立写在程序中，必须写在 initial 块中。

例 3-46　使用 forever 语句生成一个周期为 20 个时间单位的时钟信号。

```
reg clock;
initial
 begin
  clock = 0;
  forever #10 clock = ~clock;
end
```

这段程序常用于编写的测试程序中。

4. repeat 语句

语句格式：

```
repeat(表达式)语句;
```

说明：

repeat 语句执行其表达式所确定的固定次数的循环操作，其表达式通常是常数，也可以是一个变量，或者一个信号，如果是变量或者信号，循环次数是循环开始时刻变量或信号的值，而不是循环执行期间的值。

例 3-47　用加法和移位操作来完成两个 4 位数值的乘法运算。

```
module mux(data_in1,data_in2,data_out);
input [3:0] data_in1,data_in2;
output[7:0] data_out;
reg [7:0] data_out;
reg[7:0] data_in1_shift,data_in2_shift;

initial
begin
data_in1_shift = data_in1;
data_in2_shift = data_in2;
data_out = 0;
  repeat(4)
    begin
      if(data_in2_shift[0])
      data_out = data_out + data_in1_shift;
      data_in1_shift = data_in1_shift << 1;
      data_in2_shift = data_in2_shift >> 1;
     end
end
     endmodule
```

3.4.6　系统任务和系统函数

Verilog HDL 的系统任务和系统函数主要用于仿真，标准的系统任务和系统函数提供了显示、文件输入/输出、时间标度和仿真控制等各种功能。系统任务和系统函数前都有一个标志符 $ 加以确认，执行中如果有返回值为系统函数，否则为系统任务。

下面简单介绍几个常用的系统任务和系统函数，包括模块信息的屏幕显示 $display、信号的动态监控 $monitor、暂停 $stop、结束仿真 $finish、数据读入 $readmemb 和 $readmemh、文件打开 $fopen 以及文件关闭 $fclose 等。这些操作在系统的调试和测试过程中非常有用。其他系统任务和系统函数请参阅 Verilog HDL 手册。

1. $display

$display 用于变量、字符串、表达式的屏幕显示，格式如下：

```
$display(p1,p2,…,pn);
```

其中，p1，p2，…，pn 可以是字符串、变量名、表达式等，它的应用类似于 C 语言的 printf。

说明：

$display 可以根据显示格式的要求显示字符串、数值的内容，常用的显示格式说明见表 3-8。

Verilog 语言提供了一些特殊的字符，可以对显示格式进行调整，如表 3-9 所示。

表 3-8 $display 常用显示格式

格　式	显示结果
%h 或%H	以十六进制格式输出
%d 或% D	以十进制格式输出
%o 或% O	以八进制格式输出
%b 或%B	以二进制格式输出
%c 或%C	以 ASCII 格式输出
%s 或%S	显示字符串
%t 或%T	显示当前时间
%f 或%F	输出十进制形式的实数

表 3-9 特殊字符

字　符	显示结果
\n	换行
\t	横向跳格，输出到下一个输出区
\\	显示字符\
%%	显示百分符号%

例 3-48

```
$display("TESTED COMPLETE PN SEQUENCE rolling over to test again. ");
```

显示结果如下：

```
TESTED COMPLETE PN SEQUENCE rolling over to test again.
$display("a = %d , b = %2.2f",a ,b);     //数值的显示，设 a = 5,b = 2.345
```

显示结果如下：

```
a = 5 ,b = 2.35
$display("hello \nworld);                //特殊字符的显示
```

显示结果如下：

```
hello
world
```

2. $monitor

$monitor 函数提供了对信号变化进行监控的功能，格式为

```
$monitor(p1,p2,…,pn);
```

其中，p1，p2，…，pn 可以是字符串、变量、表达式、时间函数 $time 等。

说明：

(1) 在整个仿真过程中，在任意一个时刻，只要监测的一个或多个变量发生变化，就会启动 $monitor 函数，输出这一时刻的数值情况。

(2) $monitor 函数一般书写在 initial 块中，即只需调用一次 $monitor 函数，在整个仿真过程中都有效，这与 $display 不同。

(3) 在仿真过程中，如果在源程序中调用了多个 $monitor 函数，只有最后一个调用有效。

(4) Verilog 语言提供了两个用于控制监控函数的系统任务 $monitoron 和 $monitoroff。其中，$monitoron 用于启动监控任务；$monitoroff 用于关闭监控任务。在默认情况下，仿真开始时即启动了监控任务，在多模块联合调试时，为在适当的时候对各个模块的信号进行监控，就有必要将不需要的信号监控用 $monitoroff 关闭，而用 $monitoron 打开需要的信号监控。

3. $stop 和 $finish

$stop 和 $finish 常用于文本编写中的测试，它们的格式分别为

```
$stop;
$finish;
```

说明：

$stop 暂停仿真，进入一种交互模式，将控制权交给用户；$finish 结束仿真过程，返回主操作系统。

例 3-49

```
initial
begin
#100 begin a = 1;b = 1;end
#100  $stop;                          //在 200ns 时暂停仿真,交由用户控制
#200  $finish;                        //在 400ns 时,退出仿真
end
```

4. $readmemb 和 $readmemh

$readmemb 和 $readmemh 提供了将文件中的数据读到存储器阵列中的有效手段，这两个任务可以完成读取二进制或十六进制的数据。格式如下：

```
$readmemb("文件名",存储器名,起始地址,终止地址);
$readmemh("文件名",存储器名,起始地址,终止地址);
```

其中，文件名、存储器名是必须有的，起始地址、终止地址是可选项，这两类地址信息用十进制数表示。

说明：

(1) “文件名”是被读取文件的 ASCII 码形式，还可以增加该文件的位置信息，例如“c:/test/my_project/simulus.dat”。

(2) 文件中的内容中只允许有空白(包括空格、换行)、Verilog 注释行、数据地址(为

十六进制格式)、二进制数或十六进制数,数中不能有位宽、进制的说明,对 $readmemb 而言,应为二进制数值;对 $readmemh 而言,应为十六进制数值。

例 3-50

```
module test();
reg[1:0] my_mem[7:0];                    //定义了 8 个宽度为 2 的存储器组
integer i;
initial
begin
  //将位于 D:/test 目录下的 my_data.dat 中的数据读入 my_mem 中
    $readmemb("D:/ test/my_data.dat",my_mem);
    for(i = 0;i < 8;i = i + 1)
     $display("my_mem[ % d] = % b",i,my_mem[i]);
    end
endmodule
```

其中,my_data.dat 的内容如下:

```
00
01
10
11
00
01
10
11
```

执行结果如下:

```
memory[       0] = 00
memory[       1] = 01
memory[       2] = 10
memory[       3] = 11
memory[       4] = 00
memory[       5] = 01
memory[       6] = 10
memory[       7] = 11
```

(3) 数据文件中可以用@<地址>将数据存入存储器的指定位置,地址用十六进制数表示,@和<地址>间不能有空格。

例 3-51 如果例 3-50 中的 my_data.dat 为:

```
@1
00
00
@6
1x
z1
```

或

```
@1
00 00
@6
1x z1
```

执行结果如下:

```
my_mem[       0] = xx
my_mem[       1] = 00
my_mem[       2] = 00
my_mem[       3] = xx
my_mem[       4] = xx
my_mem[       5] = xx
my_mem[       6] = 1x
my_mem[       7] = z1
```

一般情况下,数据文件中指定的地址空间应该在存储器定义的空间范围之内,否则将提示出错信息。数据文件中的数据可以包括 x、z,存储器未赋值的位置默认为 x。

(4) 语句中的起始地址和终止地址的不同定义对数据装载有如下影响:

① 未指定终止地址，执行时，将数据从指定的起始地址开始载入，如果数据文件的数据个数多于存储器可以装载的单元个数，则数据存入直到该存储器的结束地址为止；反之，未能赋值的存储器单元的值默认为 x。

② 如果任务中指定了起始地址和终止地址，在执行中，将数据从起始位置开始载入，直到该指定的结束地址。如果指定的地址空间超出了定义的存储器空间，将提示出错信息。

③ 如果在数据文件和任务定义中都给出了地址信息，那么数据文件中指定的地址必须在任务定义中的地址声明范围之内，否则执行中将提示出错信息。

5. $fopen 和 $fclose

Verilog 语言支持将仿真结果输出到指定的文件中，使用系统函数 $fopen 打开一个可以写入数据的文件；再使用系统任务 $fclose 将前面打开的文件关闭。$fopen 格式为

```
<file_descriptor> = $fopen("文件名");
```

说明：

$fopen 返回的 file_descriptor 是一个 32 位(bit)的整数，称为无符号多通道描述符，该描述符每次只有一位被设置成 1，其余位为 0，表示一个独立的输出文件通道，如果文件不能正常打开，描述符的值是 0。最低位(第 0 位)置 1 表示标准输出，第 1 位被置 1 表示第一个被打开的文件，第 2 位被置 1 表示第二个被打开的文件，依次类推。

例 3-52

```
integer file_descriptor1,file_descriptor2,file_descriptor3;
initial
begin
  file_descriptor1 = $fopen("file1.out");
  file_descriptor2 = $fopen("file2.out");
  file_descriptor3 = $fopen("file3.out");
   $display("file_descriptor1 = %h",file_descriptor1);
   $display("file_descriptor2 = %h",file_descriptor2);
   $display("file_descriptor3 = %h",file_descriptor3);
  end
endmodule
```

执行结果如下：

```
file_descriptor1 = 00000002
file_descriptor2 = 00000004
file_descriptor3 = 00000008
```

当要关闭打开的文件，采用如下格式：

```
$fclose;
```

说明：

用 $fclose 将某文件关闭后，不能再写入，无符号多通道描述符的相应位设置为 0，下次 $fopen 的调用可以再使用这一位。

3.4.7 编译预处理命令

Verilog 语言和 C 语言一样提供了一些特殊的命令，在进行 Verilog 语言综合时，综合工具首先对这些特殊命令进行“预处理”，然后将得到的结果和源程序一起再进行通常的编译处理。

编译预处理指令前有一个标志符“`”(反引号)加以确认，其作用范围是：命令定义后直到本文件结束或其他命令替代或取消该命令为止。

接下来介绍常用的`define、`ifdef、`elsif、`else、`endif、`include、`timescale 等命令，其余的如`default_nettype、`resetall、`unconnected_drive、`nounconnected_drive、`celldefine 和`endcelldefine 等命令请查阅 Verilog HDL 手册。

1. 宏定义命令`define

宏定义`define 指定一个宏名来代表一个字符串。

语句格式：

```
`define  宏名  字符串
```

说明：

(1) 为与变量名区别，建议使用大写字符定义宏名。

(2) 在源文件中引用已定义的宏名时，必须在宏名前加“`”。

(3) 宏定义在预编译时只将宏名和字符串进行简单的置换，不作语法检查，如有错，只在宏展开后的源程序编译时才报错。

(4) 宏定义不是 Verilog HDL 语句，不必在末尾加分号，如果加了分号，则分号也将作为宏定义的字符串的内容，进行置换。

(5) `define 命令可以出现在模块定义里，也可以出现在模块定义外。宏定义的有效范围是宏定义命令后到源文件结束。

(6) 宏定义可嵌套使用。

例 3-53

```
`define ADD a + b                    //用 a + b 表示 ADD 字符串
assign c =`ADD;                      //在引用宏名时，前加"`"，即 assign c = a + b;
`define ADD1 a + b
`define ADD2 `ADD1 + d
assign data_out =`ADD2;              //等同于 data_out = a + b + d;
`define data "the c = % b "          //用 the c = % b 表示字符串 data
```

2. 条件编译命令`ifdef、`elsif、`else 和`endif

一般情况下，源程序的所有行都进行编译，但在一些特定的应用场合下，对源程序中满足指定编译条件的语句才进行编译，或者满足某条件时，对一组语句进行编译，当条件不满足时编译另一组语句。

语句格式：

```
`ifdef 宏名
  程序段 1;
```

```
`elsif
  程序段 2;
`else
  程序段 3;
`endif
```

其中，`else 和`elsif 命令对于`ifdef 命令是可选的。

3. 文件包含命令`include

文件包含是指一个源文件可以将另外一个源文件的内容全部“复制”过来，作为一个源程序进行编译。可用`include 来实现文件包含。

语句格式：

```
`include "文件名"
```

说明：

(1) 该语句可以出现在 Verilog HDL 程序的任何地方。编译时，`include "文件名"这一行文字由被包含的文件内容全部代替。文件名中还可以指明该文件存放的路径名。设计中可将常用的宏定义、任务、函数组成一个文件，用`include 命令包含到源文件中。

(2) 一个`include 只能包含一个文件，如果要包含多个文件，需要写多个`include 命令。如果源文件 top.v 包含 source1，而 source1.v 又需要用到 source2.v 的内容，则可以将 source1.v 和 source2.v 用`include 包含到源文件中，但是 source2.v 应出现在 source1.v 之前，即

```
`include "source2.v"
`include "source1.v"
```

例 3-54

(1) 将系统设计需要的宏定义写在一个模块 my_define.v 中。

```
`define GENERIC_MULTP2_32X32
`define IC_1W_8KB
`define RAMB16
```

(2) 上层模块 top.v 的编译中使用 my_define.v。

```
`include "my_define.v"
 module top( );
  …
endmodule
```

编译预处理后，成为如下文件：

```
`define GENERIC_MULTP2_32X32
`define IC_1W_8KB
`define RAMB16
 module top( );
   …
 endmodule
```

4. 时间尺度命令`timescale

在 Verilog HDL 模型中，所有时延都用单位时间表述。使用`timescale 命令将单位时间与实际时间相关联。用于定义仿真时间、延迟时间的单位和时延精度。

语句格式：

```
`timescale  时间单位/时间精度
```

说明：

(1) 时间单位是指时间和延迟的测量单位，时间精度是指仿真过程中延迟值进位取整的精度，时间精度应该小于或等于时间单位。时间单位和时间精度由值 1、10、和 100 以及单位秒(s)、毫秒(ms，即 10^{-3}s)、微秒(μs，即 10^{-6}s)、纳秒(ns，即 10^{-9}s)、皮秒(ps，即 10^{-12}s)等组成。该命令末尾没有分号。

(2) `timescale 命令在模块说明外部出现，并且影响后面所有的时延值，直至遇到另一个`timescale 命令。当一个设计中的多个模块带有自身的`timescale 编译命令时，仿真的时间单位与精度采用所有模块的最小时延精度，并且所有时延都相应地换算为最小时延精度。

例 3-55

```
`timescale 1ns/100ps                    //表示时间单位为 1ns,时间精度为 100ps
module testbench();
  …
initial
 begin
                  a = 0; b = 0;         //在 0 时刻,a = 0,b = 0
 #10 begin a = 4; b = 2; end            //在 0 + 10 * 1 = 10ns 时,a = 4;b = 2
 #20 begin a = 2; b = 3; end            //在 10 + 20 * 1 = 30ns 时,a = 2;b = 3
 end
endmodule
```

3.4.8 Verilog HDL 可综合设计

用硬件描述语言进行程序设计的最终目的是进行硬件实现，在硬件描述语言中，许多基于仿真的语句虽然符合语法规则，但不能用硬件来实现，即不能映射到硬件逻辑电路单元。如果要最终实现硬件设计，则必须写出可以综合的程序。

Verilog HDL 允许用户在不同的抽象层次上对电路进行建模，这些层次从逻辑级、寄存器传输级、算法级直至系统级。因此同一个电路可以有多种不同的描述方式，但不是每一种描述都可以综合。事实上，Verilog HDL 原本是被设计成一种仿真语言，而不是一种用于“综合”的语言。结果导致有些语句只满足语法，而无法映射到具体逻辑元件结构，是不可综合的，例如 initial 语句、时间声明、系统任务和系统函数等。同样，也不存在用于寄存器传输级综合的 Verilog HDL 标准子集。

由于存在这些问题，不同的“综合”系统所支持的 Verilog HDL 综合子集不同。由于在 Verilog HDL 中不存在单个对象来表示锁存器或触发器，所以每一种综合工具都会提供不同的机制以实现锁存器或触发器的建模，因此各种综合工具都定义了自己的 Verilog HDL

可综合子集以及自己的建模方式。这一局限给设计者造成了严重的障碍，因为设计者不仅需要理解 Verilog HDL，还必须理解特定综合工具的建模方式，才能编写出可综合的模型。

表 3-10 列出了可被绝大多数综合工具支持的可综合语句。

表 3-10　Verilog HDL 可综合的运算符、数据类型和语句列表

语　句			可综合	说　明
运算符	逻辑/按位运算符	|，～，&	支持	
	算术运算符	/，%	受限支持	在/和%运算中必须是除以或模 2 的幂次方
		*，+，-	支持	
	移位运算符	<<，>>	支持	
	关系运算符	<，<=，>，>=	支持	
	按位/缩减运算符	&，～&	支持	
		^，～^	支持	
		|，～|	支持	
	逻辑运算符	&&	支持	
	条件运算符	？：	支持	
	拼接运算符	{ }	支持	
数据类型	网线数据类型	wire	支持	
	寄存器数据类型	reg，integer	支持	综合工具把 integer 综合成 32 位的寄存器型数据
	存储器型		受限支持	仅支持一维限定性数组
语句	连续赋值语句	assign	支持	赋值语句的左边是 wire 型，右边是 reg、integer 或 wire 型
	过程赋值语句	=，<=	支持	一般是在 always 块内，采用非阻塞赋值
	顺序块语句	begin-end	支持	
	并行块语句	fork-join	支持	
	结构说明语句	always	支持	
		function	支持	函数内循环变量的循环次数、步长和范围必须固定；不能进行函数递归调用；函数体内不能包含任何的时间控制语句；函数不能启动任务
		task	支持	任务内循环变量的循环次数、步长和范围必须固定；不能进行任务递归调用；任务体内不能包含任何的时间控制语句；任务可以启动其他任务和函数
	条件语句	if…else if	支持	
		case	支持	
	循环语句	for	受限支持	循环次数、步长和范围必须固定
		repeat	受限支持	循环次数、步长和范围必须固定
		while	受限支持	循环次数、步长和范围必须固定

3.5 Verilog HDL 设计举例

本节通过一些可综合的 Verilog HDL 设计实例介绍如何运用 Verilog 语言对组合逻辑电路和时序逻辑电路进行描述，并重点介绍有限状态机设计。

3.5.1 组合电路设计

用 assign 语句对 wire 型变量进行赋值，综合后的结果是组合逻辑电路。

用 always@（敏感信号表），即电平敏感的 always 块描述的电路综合后的结果也可以是组合逻辑电路。此时，always 块内赋值语句左边的变量是 reg 或 integer 型，块中要避免组合反馈回路。在生成组合逻辑的 always 块中被赋值的所有信号必须都在 always@（敏感信号表）的敏感电平列表中列出，否则在综合时将会为没有列出的信号隐含地产生一个透明的锁存器，这时综合后的电路已不是纯组合电路了，自 Verilog 2001 版本开始，敏感信号表可以用(*)代替，由 * 运算符自动识别所有敏感变量。

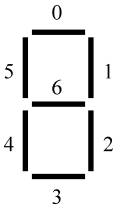
1. 编码器和译码器

编码器和译码器是常见的组合逻辑电路。组合逻辑可以使用连续赋值语句 assign，也可以使用 always 语句。在 Verilog HDL 设计中，它们有相似的设计思路，现以译码器为例讨论用 Verilog HDL 设计组合电路。

例 3-56 BCD 码将十进制数字转换为二进制，每一个十进制的数字(0～9)都对应着一个四位的二进制码，按照表 3-11 的转换关系，设计数字系统，其输出驱动信号至七段 LED 以显示相应信息。

表 3-11 十进制数与 BCD 码的转换关系

十进制数	8421BCD 码 data_in[3:0]	输出到七段 LED 的数据 data_out[6:0](共阴/共阳)	七段 LED 图例
0	0000	0111111/1000000	
1	0001	0000110/1111001	
2	0010	1011011/0100100	
3	0011	1001111/0110000	
4	0100	1100110/0011001	
5	0101	1101101/0010010	
6	0110	1111101/0000010	
7	0111	0000111/1111000	
8	1000	1111111/0000000	
9	1001	1100111/0011000	




```
module bin2bcd (data_in ,EN ,data_out );
input [3:0] data_in;
input EN;                                    //系统使能信号
output [6:0] data_out;
reg [6:0] data_out;
    always @(data_in or EN )    //当 data_in 或 EN 变化时触发 always 模块,可用 always @ ( * )
```

```
begin
        data_out = {7{1`b0}};
    if (EN == 1)
      begin
        case (data_in)                    //根据共阳接法译码
         4`b0000:data_out [6:0] = 7`b1000000;
         4`b0001:data_out [6:0] = 7`b1111001;
         4`b0010:data_out [6:0] = 7`b0100100;
         4`b0011:data_out [6:0] = 7`b0110000;
         4`b0100:data_out [6:0] = 7`b0011001;
         4`b0101:data_out [6:0] = 7`b0010010;
         4`b0110:data_out [6:0] = 7`b0000010;
         4`b0111:data_out [6:0] = 7`b1111000;
         4`b1000:data_out [6:0] = 7`b0000000;
         4`b1001:data_out [6:0] = 7`b0011000;
         default:data_out [6:0] = {7{1`b0}};
        endcase
      end
end
endmodule
```

2. 数据选择器

在数字信号的传输过程中，常需要从一组输入数据中选择一个输出，这就需要设计数据选择器或多路开关的逻辑电路。

例 3-57　设计一个数据选择器，功能描述：在选择信号 SEL、使能信号 EN 的控制下，从输入信号 IN0、IN1、IN2、IN3 中选择输出值 OUT，见表 3-12。

表 3-12　数据选择器真值表

EN	SEL	OUT
1	2`b00	OUT＝IN0
	2`b01	OUT＝IN1
	2`b10	OUT＝IN2
	2`b11	OUT＝IN3
0	任何值	OUT＝0

```
`define width 8
module mux(EN,IN0,IN1,IN2,IN3,SEL,OUT );
input EN;
input [`width－1:0] IN0,IN1,IN2,IN3;
input [1:0] SEL;
output [`width－1:0] OUT;
reg [`width－1:0] OUT;
always @(SEL or EN or IN0 or IN1 or IN2 or IN3 )     //或 always @ ( * )
    begin
     if (EN == 0)
OUT = {8{1`b0}};
     else
        case (SEL)
```

```
            2`b00:OUT = IN0;
            2`b01:OUT = IN1;
            2`b10:OUT = IN2;
            2`b11:OUT = IN3;
            default:OUT = {8{1`b0}};
          endcase
      end
endmodule
```

也可以去掉 always 块而写成：

```
wire[width - 1:0] OUT;
OUT = (EN == 0)?8`b0:(SEL == 2`b00) ? IN0
                    :(SEL == 2`b01) ? IN1
                    :(SEL == 2`b10) ? IN2
                    :(SEL == 2`b11) ? IN3
                    :8`b0;
```

3. 数值比较器

数值比较器是数字系统常用的比较两个数大小的逻辑电路，一位的数值比较器逻辑关系较为简单，可以用原理图设计方法或调用 Verilog 语言的逻辑门等完成。随着比较的数值的位数增加，如果仍用逻辑门搭建电路则较复杂，但采用 Verilog HDL 来描述这一电路还是很容易的。程序通过改变 define 宏定义的位宽值，可以被综合工具综合成不同位宽的比较器。

例 3-58 设计比较器电路，实现两个多位数的比较，并将结果显示如下：

当 a＞b 时置 a_great 为 1，其余输出端为 0；

当 a＝b 时置 a_equal_b 为 1，其余输出端为 0；

当 a＜b 时置 b_great 为 1，其余输出端为 0。

```
`define width 8                              //定义比较数值的位宽
module compare(a,b,a_great,a_equal_b,b_great);
input[`width - 1:0] a,b;
output a_great,a_equal_b,b_great;
reg a_great,a_equal_b,b_great;
always @ (a or b )     //如果 a 或者 b 任意一个发生变化，触发执行以下操作，可用 always @ ( * )
begin
   if(a > b)
        a_great = 1;
    else
       a_great <= 0;
     if(a == b)
        a_equal_b <= 1;
     else
        a_equal_b <= 0;
     if(a < b)
         b_great <= 1;
     else
         b_great <= 0;
end
endmodule
```

3.5.2　时序电路设计

对时序电路建模时，always 语句必须用时钟信号、复位或置位信号的边沿控制触发，如用 always @ (posedge clock)或 always @ (negedge clock)块描述的电路就可综合为同步时序逻辑电路。在 always 语句中对时序电路建模时一般采用非阻塞赋值。

1. 移位寄存器

当寄存器存储的代码能够在移位脉冲的作用下依次左移或右移，就构成了移位寄存器，移位寄存器不仅可以存储代码，而且可以用来实现数据的串行到并行的转换、数值的运算以及数据的处理等。

例 3-59　设计由边沿触发结构的 D 触发器组成的 4 位移位寄存器，如图 3-6 所示。

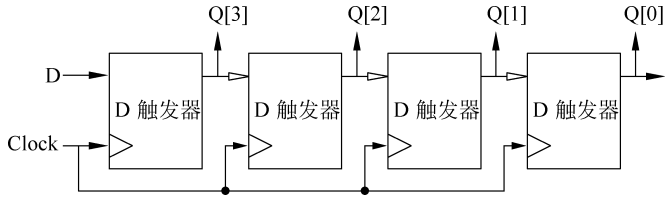


图 3-6　移位寄存器结构图

分析：当时钟 Clock 上升沿来到时，如果输入端的数据已经稳定，所有触发器的输出端按照输入端的状态翻转，加到寄存器输入端 D 的值被存入 D 触发器，按照图 3-6 的结构，相当于移位寄存器中原有的代码依次向右移了一位。经过 4 个周期时钟信号后，串行输入的 4 位代码全部移入了移位寄存器中，如果在 4 个触发器的输出端引出输出代码，就完成了串行到并行的转换。

```
module shift_flop(D,CLK,Q);
input D;
input CLK;
output [3:0] Q;
reg [3:0] Q;
always @ (posedge clk)
  begin
    Q[0] <= D;
    Q[1] <= Q[0];
    Q[2] <= Q[1];
    Q[3] <= Q[2];
    end
endmodule
```

2. 计数器

计数器是数字电路中使用广泛的时序电路，常用于脉冲计数，以控制程序执行时间，还可用于分频、定时、产生节拍脉冲等。

例 3-60　设计一个带异步复位的十六进制计数器。

```
module counter_16(clk,reset,counter_data);
input clk,reset;
```

```
reg [3:0] counter_data;
output
always @ (posedge clk or posedge reset);
 if(reset == 1)
    counter_data <= 4`b0000;              //当复位信号为1时,计数器置位为0
 else
   counter_data <= counter_data + 1;
 endmodule
```

3. 分频器

例 3-61 设有一个 50MHz 的时钟源,设计分频电路得到秒脉冲时钟信号。

分析：根据设计要求,可知分频系数应为 49999999。

```
module divider50m(inclk,outclk);
input inclk;
output outclk;
reg outclk;
reg [25:0] counter;
always @ ( posedge inclk )
    begin
        if ( count == 49999999 )
            count <= 0;
        else
            count <= count + 1;
    end
always @ ( count )
    begin
        if ( count == 49999999 )
            outclk <= 1;
        else
            outclk <= 0;
    end
endmodule
```

3.5.3 数字系统设计

在第 1 章中已指出数字系统是多个分层次嵌套的有限状态机组成,常分解成数据通道和控制单元两部分。数字系统的控制单元通常用传统的有限状态机或时钟模式的时序电路来建模。每个控制步骤可被看作一种状态,即现态,实现时该状态由一个状态寄存器来保存,与每一控制步骤相关的转移条件(即输入)确定了它将要转换的状态(即次态)。在每个时钟周期,状态寄存器中都要填入由现态和输入决定的下一个状态(次态)。

有限状态机根据其输出的逻辑关系可分为两类,当时序逻辑的输出是输入信号和当前状态的函数,且当前状态和输入信号决定了下一状态,称为 Mealy 状态机；当系统的输出只是当前状态的函数,称为 Moore 状态机。电路结构图分别如图 3-7 和图 3-8 所示。

除了根据输出信号的产生方式划分外,状态机还可以根据状态编码方式划分。常用的编码方式有二进制顺序编码、格雷码、随机码、一位有效(one-hot)编码等。

用 Verilog 语言描述有限状态机可使用多种风格。不同的风格会极大地影响电路性

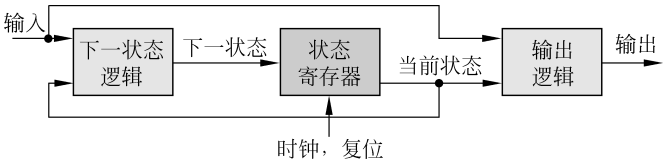


图 3-7　Mealy 状态机结构图

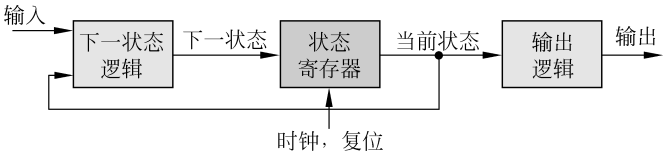


图 3-8　Moore 状态机结构图

能。通常有 3 种描述方式：单 always 块、双 always 块和三 always 块。

单 always 块把组合逻辑和时序逻辑用同一个时序 always 块描述，其输出是寄存器输出，无毛刺，但这种方式会产生多余的触发器，代码难以修改和调试，应该尽量避免使用。

双 always 块大多用于描述 Mealy 状态机和组合输出的 Moore 状态机，时序 always 块描述当前状态逻辑，组合逻辑 always 块描述下一状态（次态）逻辑并给输出赋值。这种方式结构清晰，综合后的时序性能好，资源占用少，节省面积。但组合逻辑输出往往会有毛刺，当输出向量作为时钟信号时，这些毛刺会对电路产生致命的影响。

三 always 块大多用于同步 Mealy 状态机，两个时序 always 块分别用来描述当前状态（现态）逻辑和对输出赋值，组合 always 块用于产生下一状态。这种方式的状态机也是寄存器输出，输出无毛刺，并且代码比单 always 块清晰易读，但是面积大于双 always 块。

先看两个双 always 块的 Mealy 机和 Moore 机实例，然后通过一个数字系统的设计来学习状态机的设计。

例 3-62　Mealy 型状态机。

```
…
always @(posedge clk)
    state <= next_state;
always @(state or in1 or in2)
begin
    case (state)
    2'd0: begin
        out <= in1 & in2;                  //输出
        if (in1) next_state <= 1;          //下一状态确定
        else next_state <= 2;
        end
    2'd1: begin
        out <= ~in2;
        if (in2) next_state <= 2;
        else next_state <= 3;
        end
    endcase
end
```

Mealy 型状态机的系统输出反映系统当前状态和系统的输入。

例 3-63 Moore 型状态机。

```
…
always @(posedge clk)
    state = next_state;
        always @ ( * )
    begin
    case (state)
    2`d0: begin
        out = 1;                          //输出
        if (in1) next_state = 1;          //下一状态确定
        else next_state = 2;
        end
    2`d1: begin
        if (in2) next_state = 0;
        else next_state = 3;
        end
    endcase
end
```

Moore 型状态机的系统输出反映系统当前状态，与系统的输入无关。

例 3-64 用状态机设计交通灯控制器。由一条主干道和一条支干道的汇合点形成十字交叉路口，主干道为东西向，支干道为南北向。为确保车辆安全、迅速地通行，在交叉道口的每个入口处设置了红、绿、黄 3 色信号灯。

要求：

(1) 主干道绿灯亮时，支干道红灯亮，反之亦然；主干道每次放行 35s，支干道每次放行 25s。每次由绿灯变为红灯的过程中，黄灯亮作为过渡，时间为 5s。

(2) 能实现正常的倒计时显示功能。

(3) 能实现总体清零功能：计数器由初始状态开始计数，对应状态的指示灯亮。

(4) 能实现特殊状态的功能显示：进入特殊状态时，东西、南北路口均显示红灯状态。

根据要求，交通灯的状态转换如表 3-13 所示。

表 3-13 交通灯控制器状态转换表

状 态	主 干 道	支 干 道	时间/s
0	红灯亮	红灯亮	
1	绿灯亮	红灯亮	35
2	黄灯亮	红灯亮	5
3	红灯亮	绿灯亮	25
4	红灯亮	黄灯亮	5

交通灯控制器系统框图如图 3-9 所示，包括置数模块、计数模块、译码器模块和主控制器模块。置数模块将交通灯的点亮时间预置到置数电路中，计数模块以秒为单位倒计时。当计数值减为零时，主控电路改变输出状态，电路进入下一个状态的倒计时。为了简化设计使结构清晰，将置数模块、计数模块和译码器模块视作整个系统的数据通道，与主控制器模

块构成了“数据通道＋控制器”的系统结构。因为将定时计数器划归到了数据通道，使得控制器的状态数大大减少，主控制部分可以按照有限状态机设计。下面设计主控制模块。

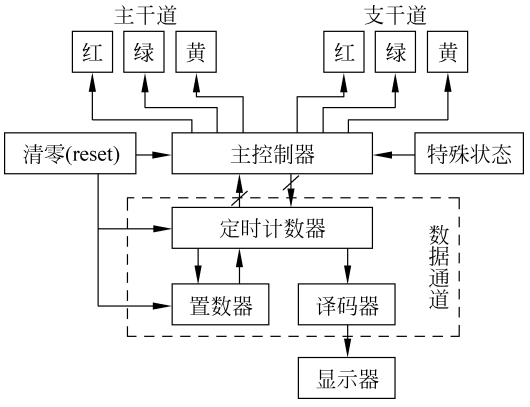


图 3-9　交通灯控制系统框图

根据对设计要求的分析，主控制单元的输入信号有：

(1) 时钟 clock。

(2) 复位清零信号 reset(reset＝1 表示系统复位)。

(3) 紧急状态输入信号 sensor1(sensor1＝1 表示进入紧急状态)。

(4) 定时计数器的输入信号 sensor2(由 sensor2[2]、sensor2[1]和 sensor2[0]三位组成，该信号为高电平时，分别表示 35s、5s、25s 的计时完成)。

输出信号有：

(1) 主干道控制信号(red1，yellow1，green1)。

(2) 支干道的控制信号(red2，yellow2，green2)。

(3) 控制状态信号 state(输出到定时计数器，分别进行 35s、25s、5s 计时)。

主控制单元的状态转移图如图 3-10 所示。

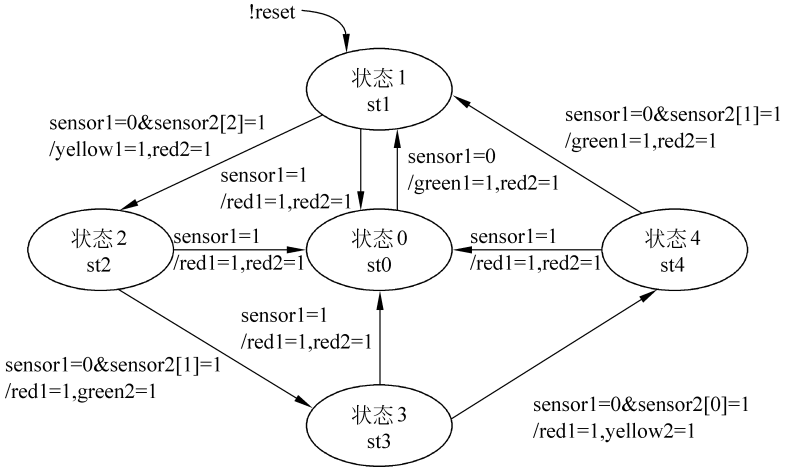


图 3-10　交通灯控制器状态转移图

注：未标出的信号灯值为 0，表示该信号灯关闭。

```
module traffic_control (clock, reset, sensor1, sensor2,
            red1, yellow1, green1, red2, yellow2, green2);

input clock, reset, sensor1, sensor2;
output red1, yellow1, green1, red2, yellow2, green2;

//定义各个状态
parameter st0 = 0, st1 = 1, st2 = 2, st3 = 3,st4 = 4;

reg [2:0] state, nxstate;
reg red1, yellow1, green1, red2, yellow2, green2;

//状态更新
always @(posedge clock) begin
    if (!reset)
     state = st1;
    else
     state = nxstate;
end

//根据当前状态和输入,计算下一个状态和输出
always @(state or sensor1 or sensor2)
begin
    case (state)                          //依据状态转移图,完成状态跳转
    st0: begin                            //状态 0
      if(sensor1)
        begin
         nxstate = st0;
            red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
            red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
            end
       else
        begin
         red1 = 1`b0; yellow1 = 1`b0; green1 = 1`b1;
         red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
         nxstate = st1;
        end
          end
    st1: begin                            //状态 1
        if(sensor1)
         begin
       red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
       red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
        nxstate = st0;
         end
         else if(sensor2[2])
            begin
              red1 = 1`b0; yellow1 = 1`b1; green1 = 1`b0;
              red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
```

```
                nxstate = st2;
                end
            end
        st2: begin                              //状态 2
          if(sensor1)
            begin
           red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
           red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
           nxstate = st0;
            end
            else if(sensor2[1])
             begin
           red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
           red2 = 1`b0; yellow2 = 1`b0; green2 = 1`b1;
            nxstate = st3;
             end
           end
        st3: begin                              //状态 3
           if(sensor1)
            begin
           red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
           red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
            nxstate = st0;
             end
            else if(sensor2[0])
              begin
           red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
           red2 = 1`b0; yellow2 = 1`b1; green2 = 1`b0;
            nxstate = st4;
             end
           end
        st4: begin                              //状态 4
             if(sensor1)
            begin
           red1 = 1`b1; yellow1 = 1`b0; green1 = 1`b0;
           red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
           nxstate = st0;
            end
           else if(sensor2[1])
             begin
           red1 = 1`b0; yellow1 = 1`b0; green1 = 1`b1;
           red2 = 1`b1; yellow2 = 1`b0; green2 = 1`b0;
            nxstate = st1;
             end
           end
    endcase
end
endmodule
```

3.5.4 数码管扫描显示电路

单个数码管显示器包括 7 个 LED 灯和 1 个小圆点，需要 8 个 I/O 口来进行控制。采用这种控制方式，当使用多个数码管进行显示时，每个数码管都将需要 8 个 I/O 口。在实际应用中，为了减少 FPGA 芯片 I/O 口的使用数量，一般会采用分时复用的扫描显示方案进行数码管驱动。以 4 个数码管显示为例，采用扫描显示方案进行驱动时，4 个数码管的 8 个段码并接在一起，再用 4 个 I/O 口分别控制每个数码管的公共端，动态点亮数码管。这样只用 12 个 I/O 口就可以实现 4 个数码管的显示控制，比静态显示方式的 32 个 I/O 口数量大大减少。

如图 3-11 所示，在最右端的数码管上显示“3”时，并接的段码信号为“01100001”，4 个公共端的控制信号为“1110”。这种控制方式采用分时复用的模式轮流点亮数码管，在同一时间只会点亮一个数码管，数码管扫描显示电路的时序如图 3-12 所示。分时复用的扫描显示利用了人眼的视觉暂留特性，如果公共端控制信号的刷新速度足够快，人眼就不会区分出 LED 的闪烁，认为 4 个数码管是同时点亮的。

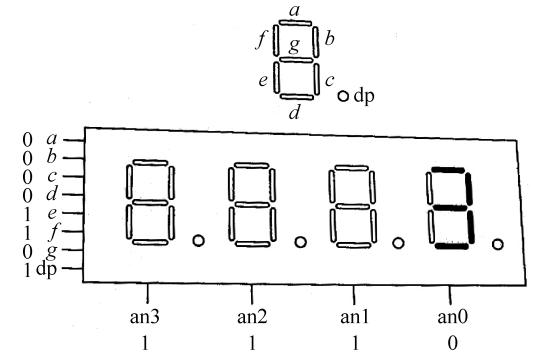


图 3-11 数码管扫描显示电路

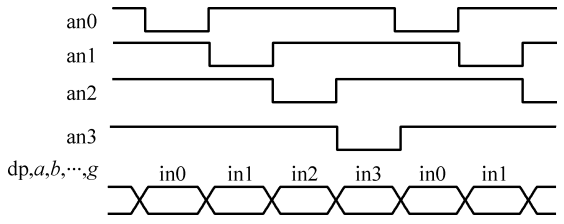


图 3-12 数码管扫描显示电路时序图

分时复用的数码管显示电路模块含有 4 个控制信号，即 an3、an2、an1 和 an0，以及与控制信号一致的输出段码信号 sseg。控制信号的刷新频率必须足够快才能避免闪烁感，但也不能太快，以免影响数码管的开关切换，最佳工作频率为 1000Hz 左右。在设计中，利用一个 18 位二进制计数器对系统输入时钟进行分频得到所需工作频率，分频器高两位用来作为控制信号，例如 an[0]的刷新频率为$(50\times10^6/2^{16})$Hz，约等于 800Hz。4 位数码管动态扫描显示电路的 Verilog 实现代码见例 3-65。

例 3-65　4 位数码管动态扫描显示电路的 Verilog HDL 描述。

```
module scan_led_disp
 (input clk,reset,
  input [7:0] in3,in2,inl,in0,
   output reg [3:0] an,
     output reg[7:0]sseg
  );
localparam N : 18;                          //对输入 50MHz 时钟进行分频(50 MHz/2^16)
reg[N-1:0] regN;
ahays @ (posedge clk,posedge reset)
if (reset)
  regN<= 0;
else
 regN<: regN+1,
always @ *
  case (regN[N-l:N-2])
2`b00:
begin
      an=4`b1110;
      sseg=in0;
 end
2`b01:
begin
      an:4`b1101;
      sseg:in1;
 end
2`b10:
begin
      an:4`b1011;
      sseg:in2;
 end
default:
 begin
       an=4`b0111;
       sseg=in3;
  end
endcase
endmodule
```

当采用分时复用电路,在七段式数码管上显示十六进制数字时,还需要 4 个译码电路,另外一个更好的选择是首先输出多路十六进制数据然后将其译码。这种方案只需要一个译码电路,使四选一数据选择器的位宽从 8 位降为 5 位(4 位十六进制数和 1 位小数点)。实现代码见例 3-66。除 clock 和 reset 信号之外,输入信号包括 4 个 4 位十六进制数 hex3、hex2、hex1、hex0 和 p_in 中的 4 位小数点。

例 3-66　4 位十六进制数的数码管动态显示电路 Verilog HDL 描述。

```
module scan_led_hex_disp
      (input clk, reset,
   input [3:0] hex3, hex2,hex1,hex0,
```

```
    input [3:0] dp_in,
    output reg [3:0] an,
    output reg [7:0]sseg
);
localparam N = 18;                          //对输入 50MHz 时钟进行分频(50 MHz/2^16)
reg[N-1:0] regN;
reg[3:0] hex_in;
  always @ (posedge clk,posedge reset)
 if ( reset)
  regN <= 0;
else
   regN <= regN + 1;
always @ *
  case (regN[N-1:N-2])
 2`b00:
  begin
            an = 4`b1110;
            hex_in = hex0;
            dp = dp_in[0];
   end
  2`b01:
  begin
            an = 4`b1101;
            hex_in = hex1;
            dp = dp_in[1];
   end
2`b10:
begin
       an = 4`b1011;
       hex_in = hex2;
       dp = dp_int[2];
 end
default:
begin
       an = 4`b0111;
       hex_in = hex3;
       dp = dp_in[3];
 end
endcase
always @ *
begin
    case ( hex_in)
         4`h0:sseg[6:0] = 7`b1000000;
         4`h1:sseg[6:0] = 7`b1111001;
         4`h2:sseg[6:0] = 7`b0100100;
         4`h3:sseg[6:0] = 7`b0110000;
         4`h4:sseg[6:0] = 7`b0011001;
         4`h5:sseg[6:0] = 7`b0010010;
         4`h6:sseg[6:0] = 7`b0000010;
         4`h7:sseg[6:0] = 7`b1111000;
         4`h8:sseg[6:0] = 7`b0000000;
```

```
            4`h9:sseg[6:0] = 7`b0011000;
            4`ha:sseg[6:0] = 7`b0001000,
            4`hb:sseg[6:0] = 7`b0000011;
            4`hc:sseg[6:0] = 7`b1000110;
            4`hd:sseg[6:0] = 7`b0100001;
            4`he:sseg[6:0] = 7`b0000110;
            default: sseg[6:0] = 7`b0001110; //4`hf
 endcase
   sseg[7] = dp;
end
endmodule
```

实际可在 FPGA 的电路中验证该设计，把 8 位开关数据作为两个 4 位无符号数据的输入，并使两个数据相加，将其结果显示在四位七段式数码管上。实现代码见例 3-67。

例 3-67　4 位十六进制数的数码管动态显示测试。

```
module scan_led_hex_disp_test
    (input clk,
    input[7:0] sw,
    output [3:0] an,
    output [7:0]sseg );
  wire [3:0] a,b;
  wire [7:0] sum;
  assign a = sw [3:0];
  assign b = sw [7:4];
  assign sum = {4`b0,a} + {4`b0, b};
    //实例化 4 位十六进制数动态显示模块
  scan_led_hex_disp_scan_led_disp_unit
    (.clk(clk), .reset(1`b0),
    .hex3(sum[7:4]),.hex2(sum[3:0]),.hexl(b),.hex0(a),
    .dp_in(4`b1011),.an(an),.sseg(sseg));
endmodule
```

许多时序逻辑电路一般工作在相对较低的频率，就像分时复用数码管电路中的使能脉冲一样。这可以通过使用计数器来产生只有一个时钟周期的使能信号。在这个电路中使用的是 18 位计数器：

```
localparam N = 18;
reg [N - 1:0] regN;
```

考虑计数器的位数，仿真这种电路需要消耗大量的计算时间（2^{18} 个时钟周期为一个周期）。因为主要工作在于分时复用那段代码，大部分模拟时间被浪费了。更高效的方法是使用一个较小的计数器进行仿真，可以通过修改常量声明来实现：

```
localparam N = 4;
```

这样就只需要 2^4 个时钟周期为一个仿真周期，节省了大量时间，并且可以更好地观察关键操作。

最好定义参数 N，而不是将其设置为一个常量，在仿真与综合时可以方便修改代码。同时在实例化过程中，也可以对于仿真和综合设置不同的值。

3.5.5 LED 通用异步收发电路设计

通用异步收发(UART)是一种通用串行数据总线,用于异步通信。UART 能实现双向通信,在嵌入式设计中,它常用于主机与辅助设备通信。UART 是异步串行通信口的总称,包括 RS232、RS449、RS423、RS422 和 RS485 等各种异步串行通信接口标准规范和总线标准规范,它规定了通信口的电气特性、传输速率、连接特性和接口的机械特性等内容,实际上是属于通信网络中的物理层(最底层)的概念,与通信协议没有直接关系。在本案例中采用的是 RS232 通信协议。UART 传输时序如图 3-13 所示。

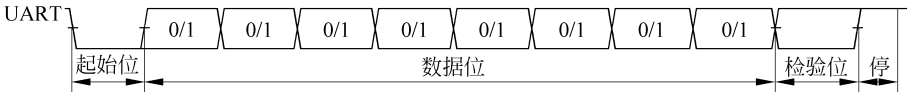


图 3-13 UART 传输时序

(1) 发送数据过程:空闲状态,线路处于高电平;当收到发送数据指令后,拉低电平一个数据位的时间(如图 3-13 起始位的时间);接着数据按低位到高位依次发送,数据发送完毕,接着发送奇偶校验位和停止位(停止位为高电平),一帧数据发送结束(本案例中,LED 电路仅处于接收状态,关于发送功能暂时不讨论)。

(2) 接收数据过程:空闲状态,电路处于高电平;当检测到电路的下降沿,说明电路有数据传输,按照约定的波特率从低位到高位接收数据,数据接收完毕;接着接收并比较奇偶校验位是否正确,如果正确,则通知接收端设备准备接收数据或存入缓存。

此案例中波特率默认设置为 115 200b/s,时钟频率默认为 50MHz,过采样率为 16 倍波特率,因为分频数至少为 2,所以时钟频率必须至少为 32 倍波特率。Rst 是输入系统的异步复位信号,在同步逻辑中利用之前,它必须同步到时钟域 clk_rx,可以利用一个简单的固化亚稳态的方法实现。此案例主要涉及模块如图 3-14 所示。

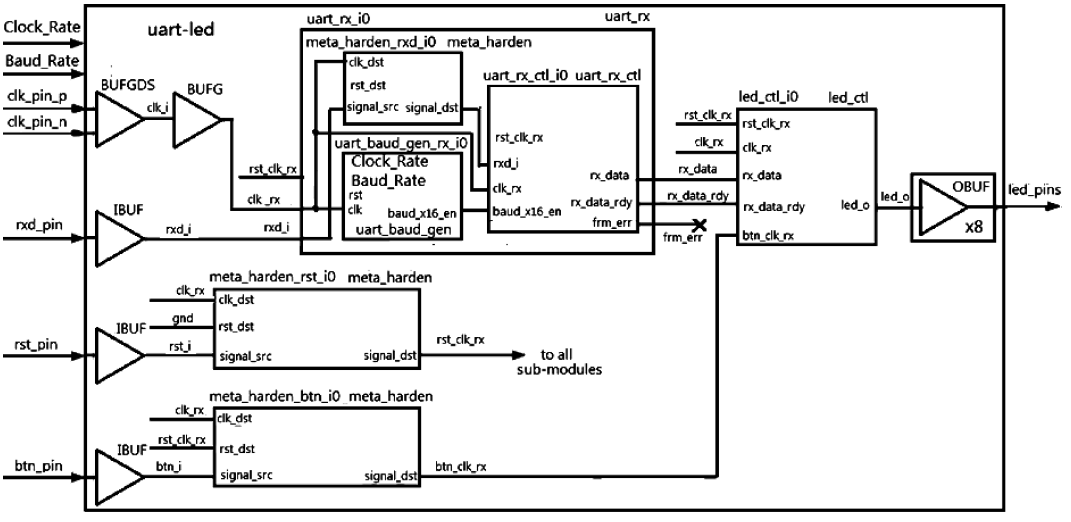


图 3-14 uart-led 组成框图

差分时钟由 IBUFGDS 缓冲把信号带进 FPGA，通过 BUFG 分布到低偏斜的内部时钟网线上。按钮的复位由 IBUF 缓冲，并馈送到 meta_harden 模块以减少其将成为亚稳态的风险，被“净化”的复位将提供其余模块。第二个按钮信号由 IBUF 缓冲，馈送到 meta_harden 模块，被同步的信号提供给 led_ctl 模块。uart_rx 捕获由 IBUF 缓冲 rxd_pin 的串行数据，将它存放到模块输出端的 rx_data 8 位总线上。信号 rx_data_rdy 持续一个时钟的脉冲为高。在这个设计中，来自 uart_rx 的帧错误信号没有利用。认定 rx_data_rdy 对 led_ctl 模块的作用，它捕获接收的字符，保持它在 LED 上显示。btn_clk_rx 交换 LED 的最高和最低有效位。最后，led_ctl 模块的输出通过 8 个 OBUF 驱动封装引脚。

时钟频率 CLOCK_RATE 和波特率 BAUD_RATE 都是通用类参数，前者以 Hz 规定，后者以波特率规定。前者的数值用于在推敲和编译期间计算与时钟频率有关的各种设置。后者的数值用于在推敲和编译期间分频时钟频率产生被选的波特率。

uart_rx 模块固化进入的串行信号防止亚稳态，传递此信号到 uart_rx_ctl 模块，同时进入的 200MHz 时钟信号被分频到 16 倍的波特率。uart_rx_ctl 模块是一个状态机，空闲的 IDLE 状态等待进入信号的下降沿，在检测到这个沿之后，它开始计数 8 个 baud_x16_en 的事件，查看固化的串行输入网线。时间延迟采样应该接近串行位的中点。如果是低电平，认为是起始位，并继续进行收集数据位；否则，认为串行线的变低数值是毛刺，返回 IDLE 状态。在串行线上采集数据并进入保持寄存器，当所有的数据被采集，在使能信号出现后或停止位的中央串行线再一次采样，一旦停止位被采集，则捕获的数据以并行的方式提供 rx_data，rx_data_rdy 持续一个时钟周期，表示新数据已经到达。在最后采样的事件中不是高电平，帧错误信号 frm_err 表示帧错误。

例 3-68 uart_led.v 通信总线模块设计。

```
`timescale 1ns/1ps
module uart_led (
  //Write side inputs
  input          clk_pin_p,          //来自引脚的时钟输入
  input          clk_pin_n,          //差分对
  input          rst_pin,            //来自引脚的高电平复位
  input          btn_pin,            //高低位变换按钮
  input          rxd_pin,            //直接来自引脚的 RS232 RXD
  output         [7:0] led_pins);    //8 LED 输出

   parameter BAUD_RATE = 115_200;
   parameter CLOCK_RATE = 200_000_000;

  //BUFG 的输出
   wire          clk_rx;
   wire          rst_clk_rx;         //同步复位
   wire          btn_clk_rx;         //同步按钮
  //Between uart_rx and led_ctl
  wire [7:0]     rx_data;            //uart_rx 的数据输出
  wire           rx_data_rdy;        //uart_rx 的数据准备输出

clk_core clk_core_inst (
```

```
        .clk_in1_p (clk_pin_p),
        .clk_in1_n (clk_pin_n),
        .clk_out1 (clk_rx));

  meta_harden meta_harden_rst_i0 (
    .clk_dst     (clk_rx),
    .rst_dst     (1'b0),                  //复位固化器上无复位
    .signal_src  (~rst_pin),  //针对 EG01 板卡复位的极性添加,如果复位不需要反相,则为 rst_pin
    .signal_dst  (rst_clk_rx));
  //输入按钮
  meta_harden meta_harden_btn_i0 (
    .clk_dst     (clk_rx),
    .rst_dst     (rst_clk_rx),
    .signal_src  (btn_pin),
    .signal_dst  (btn_clk_rx));
  uart_rx #(
    .CLOCK_RATE  (CLOCK_RATE),
    .BAUD_RATE   (BAUD_RATE))
    )
uart_rx_i0 (
    .clk_rx      (clk_rx),
    .rst_clk_rx  (rst_clk_rx),
    .rxd_i       (rxd_pin),
    .rxd_clk_rx  (),
    .rx_data_rdy (rx_data_rdy),
    .rx_data     (rx_data),
    .frm_err     () );
  led_ctl led_ctl_i0 (
    .clk_rx      (clk_rx),
    .rst_clk_rx  (rst_clk_rx),
    .btn_clk_rx  (btn_clk_rx),
    .rx_data     (rx_data),
    .rx_data_rdy (rx_data_rdy),
    .led_o       (led_pins));

endmodule
```

例 3-69 uart_rx.v 异步通信接收模块设计。

这是 UART 接收机的顶层,将同步 rxd_pin 的准稳态固化器、产生适合 x16 位使能的波特率产生器和 UART 自身的控制器放在一起组成一个子模块。波特率产生器生成 N 选 1 的脉冲,N 由波特率和系统时钟频率确定,此信号使能 uart_rx_ctl 模块中所有的触发器,对于波特率和系统频率的所有合理的组合,只要 $N>2$,uart_rx_ctl 模块中所有的路径都是多周期的。

```
`timescale 1ns/1ps
module uart_rx (
  //Write side inputs
  input           clk_rx,                  //时钟输入
```

```
    input           rst_clk_rx,          //高电平有效复位,与 clk_rx 同步
    input           rxd_i,               //直接来自焊盘的 RS232 RXD 引脚
    output          rxd_clk_rx,          //同步于 clk_rx 的 RXD 引脚
    output          [7:0] rx_data,       //8 位数据输出,在 rx_datardy 插入后有效
    output          rx_data_rdy,         //rx_data 的准备信号
    output          frm_err );           //未检测到 STOP 位

    parameter BAUD_RATE = 115_200;       //波特率
    parameter CLOCK_RATE = 50_000_000;

    wire            baud_x16_en;         //对 uart_rx_ctl 触发器 N 选 1 使能

    /* 将 RXD 引脚同步到 clk_rx 时钟域。因为 RXD 随采样时钟缓慢变化,一个简单的准稳态固化器
       就足够 */
    meta_harden meta_harden_rxd_i0 (
      .clk_dst      (clk_rx),
      .rst_dst      (rst_clk_rx),
      .signal_src   (rxd_i),
      .signal_dst   (rxd_clk_rx)
    );

    uart_baud_gen #
    ( .BAUD_RATE (BAUD_RATE),
      .CLOCK_RATE (CLOCK_RATE)
    ) uart_baud_gen_rx_i0 (
      .clk          (clk_rx),
      .rst          (rst_clk_rx),
      .baud_x16_en (baud_x16_en));

    uart_rx_ctl uart_rx_ctl_i0 (
      .clk_rx       (clk_rx),
      .rst_clk_rx   (rst_clk_rx),
      .baud_x16_en (baud_x16_en),
      .rxd_clk_rx   (rxd_clk_rx),
      .rx_data_rdy (rx_data_rdy),
      .rx_data      (rx_data),
      .frm_err      (frm_err));

endmodule
```

例 3-70　uart_baud_gen.v 波特率发生模块设计。

这个模块产生一个 16x 波特使能,当系统时钟频率和波特率的参数确定时,以 16 倍波特率产生这个信号。

```
//局部参数
//     OVERSAMPLE_RATE: 过采样率——16 x BAUD_RATE
//     DIVIDER     : 每个 baud_x16_en 的时钟数
//     CNT_WIDTH   : 计数器宽度
//说明:
//1) 分频器必须至少为 2 (因此 CLOCK_RATE 必须至少为 32x BAUD_RATE)
`timescale 1ns/1ps
```

```
module uart_baud_gen (
  //写端输入
  input         clk,                        //时钟输入
  input         rst,                        //高电平有效复位,与 clk 同步
  output        baud_x16_en                 //过采样波特率使能
);
// ****************************************************************************
//常数函数
// ****************************************************************************
  //产生基 2 对数的上限,即位数
  //要求持有 N 个值,大小为 clogb2(N)的向量将持有值为 0～N－1
  function integer clogb2;
    input [31:0] value;
    reg   [31:0] my_value;
    begin
      my_value = value - 1;
      for (clogb2 = 0; my_value > 0; clogb2 = clogb2 + 1)
        my_value = my_value >> 1;
    end
  endfunction
//参数定义
  parameter BAUD_RATE = 57_600;             //波特率
  parameter CLOCK_RATE = 50_000_000;
  //过采样波特率是波特率的 16 倍
  localparam OVERSAMPLE_RATE = BAUD_RATE * 16;
  //分频数是 CLOCK_RATE / OVERSAMPLE_RATE 的舍入
  //所以在整数除法之前,加 1/2 的过采样率
localparam DIVIDER = (CLOCK_RATE + OVERSAMPLE_RATE/2) / OVERSAMPLE_RATE;
  //计数器重新加载数值为 DIVIDER－1
  localparam OVERSAMPLE_VALUE = DIVIDER - 1;
  //要求的计算器宽度为 DIVIDER 的基 2 对数的上限
  localparam CNT_WID = clogb2(DIVIDER);
  reg [CNT_WID－1:0] internal_count;
  reg                baud_x16_en_reg;
  wire [CNT_WID－1:0] internal_count_m_1;  //减 1 计数
  assign internal_count_m_1 = internal_count－1`b1;
  //从 DIVIDER－1 到 0 计数, 当 internal_count = 0 时设置 baud_x16_en_reg
  //信号 baud_x16_en_reg 必须来自触发器(因为它是一个模块的输出)
  //当下一个计数为 1(即 internal_count_m_1 为 0)时,安排来设置它
  always @(posedge clk)
  begin
    if (rst)
    begin
      internal_count <= OVERSAMPLE_VALUE;
      baud_x16_en_reg <= 1`b0;
    end
    else
    begin
```

```
      baud_x16_en_reg <= (internal_count_m_1 == {CNT_WID{1'b0}});
      //OVERSAMPLE_VALUE 重复计数至 0
      if (internal_count == {CNT_WID{1'b0}})
      begin
        internal_count <= OVERSAMPLE_VALUE;
      end
      else                                    //internal_count 不为 0
      begin
        internal_count <= internal_count_m_1;
      end
    end
  end
  assign baud_x16_en = baud_x16_en_reg;

endmodule
```

例 3-71　led_ctl. v LED 主控模块程序设计。

```
module led_ctl (
  //写端输入
  input       clk_rx,                     //时钟输入
  input       rst_clk_rx,                 //高电平有效复位,与 clk_rx 同步
  input       btn_clk_rx,                 //高低位引脚变换按钮
  input       [7:0] rx_data,              //8 位数据输出——当 rx_data_rdy 有效时
  input       rx_data_rdy,                //rx_data 的准备信号
  output reg  [7:0] led_o);               // LED 输出
  reg         old_rx_data_rdy;
  reg [7:0]   char_data;

  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
    begin
      old_rx_data_rdy <= 1'b0;
      char_data <= 8'b0;
      led_o <= 8'b0;
    end
    else
    begin
      //获取边沿检测时 rx_data_rdy 的值
      old_rx_data_rdy <= rx_data_rdy;
      //如果 rx_data_rdy 为上升沿,捕获 rx_data 的值
      if (rx_data_rdy && !old_rx_data_rdy)
      begin
        char_data <= rx_data;
      end
          //输出正常的数据以及高低位变换后的数据
      if (btn_clk_rx)
        led_o <= {char_data[3:0],char_data[7:4]};
```

```
      else
        led_o <= char_data;
    end
  end
endmodule
```

例 3-72 分析下列 uart_rx_ctrl.v 程序的状态机描述。

```
module uart_rx_ctl (  //写输入
  input             clk_rx,              //输入时钟
  input             rst_clk_rx,          //高电平有效复位——同步于 clk_rx
  input             baud_x16_en,         //16x 过采样使能
  input             rxd_clk_rx,          //RS232 RXD 引脚——同步 clk_rx 之后
  output reg [7:0]  rx_data,             //8 位数据输出,当 rx_data_rdyc 插入时有效
  output reg        rx_data_rdy,         //为 rx_data 的 Ready 信号
  output reg        frm_err );           //STOP 位不被检测
 localparam         //State encoding for main FSM
    IDLE   = 2`b00,
    START  = 2`b01,
    DATA   = 2`b10,
    STOP   = 2`b11;
  reg [1:0]         state;               //主状态机
  reg [3:0]         over_sample_cnt;     //过采样计数器——每位 16
  reg [2:0]         bit_cnt;             //We Rx 的位计数器
  wire              over_sample_cnt_done; //处于一位的中间
  wire              bit_cnt_done;        //这是最后一个数据位
  always @(posedge clk_rx)               //主状态机
  begin
    if (rst_clk_rx)
       state <= IDLE;
    else  begin
      if (baud_x16_en) begin
    case (state)
       IDLE: begin                       //检测到 rxd_clk_rx 变低,转换到 START 状态
            if (!rxd_clk_rx)
            state <= START;
           end //IDLE state

       START: begin                      //在 1/2 位周期之后,重新确认 START 状态
          if (over_sample_cnt_done)
            if (!rxd_clk_rx)             //非毛刺的状态位
                state <= DATA;
              else                       //毛刺被拒绝
                state <= IDLE;
          end //START state
       DATA: begin                       //一旦最后一位已接收,对 stop 停止位检验
          if (over_sample_cnt_done && bit_cnt_done)
                state <= STOP;
            end                          //DATA state
```

```
            STOP: begin     //返回 idle
              if (over_sample_cnt_done)//过采样计数完成
                 state <= IDLE;
              end
          endcase
        end
      end
    end
    //过采样计数器:在 IDEL 状态检测到起始条件 rxd_clk_rx 为 0,预加到 7,这时处于第一位中间
    //在确认 START 状态,并在所有数据位之间,预加到 15
   always @(posedge clk_rx)
    begin
      if (rst_clk_rx)
          over_sample_cnt <= 4`d0;
      else  begin
        if (baud_x16_en)  begin
          if (!over_sample_cnt_done)
                over_sample_cnt <= over_sample_cnt - 1`b1;
          else if ((state == IDLE) && !rxd_clk_rx)
                 over_sample_cnt <= 4`d7;
          else if (((state == START) && !rxd_clk_rx) || (state == DATA)  )
                 over_sample_cnt <= 4`d15;
         end
      end
    end
    assign over_sample_cnt_done = (over_sample_cnt == 4`d0);

    //关于接收跟踪哪一位,当确认起始条件时,设置为 0,在整个 DATA 状态增量
    always @(posedge clk_rx)
    begin
      if (rst_clk_rx)
          bit_cnt <= 3`b0;
      else  begin
        if (baud_x16_en) begin
          if (over_sample_cnt_done)  begin
            if (state == START)
                 bit_cnt <= 3`d0;
            else if (state == DATA)
                 bit_cnt <= bit_cnt + 1`b1;
          end
        end
      end
    end

    assign bit_cnt_done = (bit_cnt == 3`d7);
  //捕获数据,产生 rdy 信号,一旦捕获最后一位数据,rdy 信号将被产生,
  //甚至 STOP 还没有确认,它就被插入,维持一个位周期(16 个 baud_x16_en 周期)
    always @(posedge clk_rx)
    begin
      if (rst_clk_rx)
```

```
    begin
      rx_data <= 8`b0000_0000;
      rx_data_rdy <= 1`b0;
    end
    else if (baud_x16_en && over_sample_cnt_done)
        if (state == DATA)
        begin
          rx_data[bit_cnt] <= rxd_clk_rx;
          rx_data_rdy <= (bit_cnt == 3`d7);
        end
        else
          rx_data_rdy <= 1`b0;
  end

    //帧错误产生,一旦帧位被假设采样,产生持续一个 baud_x16_en 周期
  always @(posedge clk_rx)
  begin
    if (rst_clk_rx)
       frm_err <= 1`b0;
    else
      if (baud_x16_en)
        if ((state == STOP) && over_sample_cnt_done && !rxd_clk_rx)
          frm_err <= 1`b1;
        else
          frm_err <= 1`b0;
  end
endmodule
```

请参考 5.3.4 节中双触发器技术,编写 meta_harden.v。

最后还包括一个时钟 IP 模块 clk_core,用于生成要求的时钟信号。

在第 4 章将会利用上述程序进行 uart_led 的设计,并应用仿真程序验证程序的功能是否正确,时序约束后,设计实现达到性能的要求,位流文件下载到硬件验证结果的正确。

3.6 Testbench 文件与设计

在采用 HDL 进行电路设计时,仿真是设计过程中不可或缺的环节。通过仿真能对设计的正确性进行验证,并及时发现问题和调整设计。针对较复杂的设计,要获得较好的工作效率,常用的方式是先由测试者编写测试平台(Testbench),再在仿真工具上运行来进行验证。仿真过程一般包括以下工作:

(1) 产生仿真激励(波形);

(2) 将仿真的输入激励加入被测试模块端口并观测其输出响应;

(3) 将被测模块的输出与期望值进行比较,验证设计的正确性。

1. 测试平台的搭建

Verilog 语言描述的模块中,其中功能模块可以完成特定的电路功能描述,如前面讨论

的半加器、全加器模块；测试模块描述变化的测试信号和监视输出信号，通过观察被测试模块的输出信号情况，对模块进行调试和验证。

测试平台的建立有两种模式，二者的不同之处在于模块的驱动设计。在设计中可以根据具体情况选择测试平台。

(1) 测试模块是顶层模块，它直接调用功能模块。这是一种较常用的测试平台，框图如图 3-15 所示。

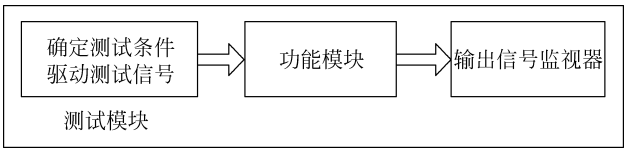


图 3-15　测试平台框图 1

例 3-73　用图 3-15 的测试平台对例 3-1 的半加器进行测试。

```
`timescale 1ns/100ps                    //指明 1 个时间单位是 1ns,其精度是 100ps
module testbench;                       //测试模块
    reg a,b;                            //激励信号名字可以和半加器模块一样,但数据类型不同
    wire co,s;
   //实例化半加器,半加器各端口和 testbench 的端口相连
    halfadder u1 ( .A(a),.B(b), .CO(co), .S(s));
   //产生各种可能的输入信号组合进行测试
   initial begin
        a = 0;
        b = 0;
    #10 begin a = 1;b = 1;end
    #10 begin a = 1;b = 0;end
    #10 begin a = 0;b = 1;end
    #10 begin a = 0;b = 0;end
    #10 $stop;                          //暂停仿真
     end
endmodule
```

对测试平台而言，输入端口 a、b 是寄存器型变量，如同信号发生器的输出端口，通过在初始化语句(initial)中对 a、b 进行赋值，驱动半加器的输入端口。输出端口 co、s 为线网型，所以名称相同的输入和输出类型与设计模块的类型正好相反。

(2) 将测试模块和设计模块分别设计完成，然后在一个虚拟的顶层模块中进行调用，将相应端口进行连接。

例 3-74　用图 3-16 的测试平台对例 3-1 描述的半加器进行测试。

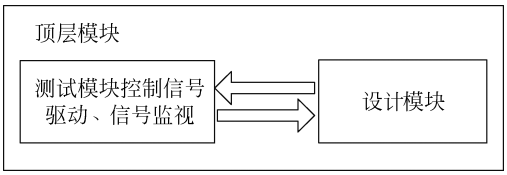


图 3-16　测试平台框图 2

第一步：编写针对半加器的测试模块。

```
`timescale 1ns/100ps
 module testhalfadder(a,b,co,s);
 input co,s;                          //注意这里的模块输入输出信号方向与半加器模块刚好相反
 output a,b;
//激励信号
 reg a;
 reg b;
 //输出
 wire s;
 wire co;
 initial begin
    a = 0;
    b = 0;
   #10 begin a = 1;b = 1;end
   #10 begin a = 1;b = 0;end
   #10 begin a = 0;b = 1;end
   #10 begin a = 0;b = 0;end
   #10 $stop;
 end
endmodule
```

第二步：实例化半加器及测试模块，建立两个模块在同一个层次上的连接。

```
module testbench;                     //这个顶层模块没有输入、输出端口
//实例化测试模块
testhalfadder u1 (.co(co),.s(s),.a(a),.b(b));
//实例化半加器模块
halfadder u2 (.A(a),.B(b),.S(s),.CO(co));
endmodule
```

2. Testbench 的时钟产生方法

测试文件中时钟波形的设计是最基本的设计，常利用 initial、always、forever、assign 等语句产生时钟信号。下面分别介绍产生周期性时钟和具有相移的时钟的方法。

(1) 要产生周期性的时钟方波，多采用 always 和 initial 结合的方式，其中 initial 进行初始赋值。

例 3-75 产生周期为 20ns 的时钟。

```
`timescale 1ns/100ps
module Gen_clock1 (clock1);
output clock1;
reg clock1;
parameter T = 20;
  initial
  clock1 = 0;
  always
  # (T/2) clock1 = ~clock1;
endmodule
```

波形如图 3-17 所示。

利用 forever 同样可以产生图 3-17 所示的周期为 20ns 的时钟。

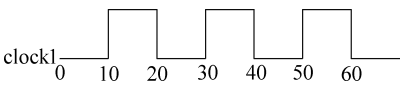


图 3-17　周期性的时钟

例 3-76

```
`timescale 1ns/100ps
…
Initial begin
 clock = 0;
 forever #10 clock2 = ~clock2;
end
endmodule
```

(2) 采用 always 语句产生高低电平持续时间不同的时钟。

例 3-77

```
module Gen_clock3 (clock3);
output clock3;
reg clock3;
always
begin
   # 4 clock3 = 0;                    //延时 4 个单位时间后,clock3 赋值 0
   # 6 clock3 = 1;                    //延时 4 个单位时间后,clock3 赋值 1
end
endmodule
```

波形如图 3-18 所示,高电平持续时间 4,低电平持续时间 6,初始值为不确定 x。

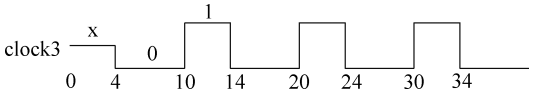


图 3-18　高低电平持续时间不同的时钟

例 3-78　利用 forever 也能产生如图 3-18 的时钟波形。

```
…
Forever begin
# 4 clock4 = 1;
# 6 clock4 = 0;
end
end
endmodule
```

(3) 利用前面的时钟产生模块,再通过添加连续赋值语句 assign 语句,就可以得到具有相移的时钟。

例 3-79

```
module Gen_clock1 (clock_pshift,clock1);
output clock_pshift,clock1;
reg clock1;
wire clock_pshift;
parameter T = 20;
```

```
parameter pshift = 2;
initial
  clock1 = 0;
always
  # (T/2) clock1 = ~clock1;
assign #PSHIFT clock_pshift = clock1;
endmodule
```

波形如图 3-19 所示，clock1 是周期为 20 的时钟，clock_pshift 是由 clock1 相移而来的。

3. Testbench 的描述方法

在测试平台上，除了时钟，还有多个输入信号值需要描述。随着数字电路系统的复杂性增加，测试时间可能占到设计总时间的 70%，测试的“完备性”对于降低设计的功能模块在使用中的风险起到了很大作用。下面介绍几种 Testbench 的描述方法。

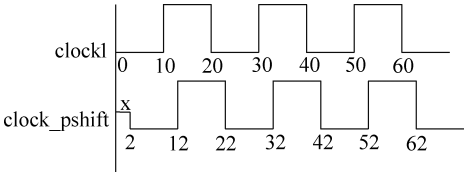


图 3-19 相移时钟

(1) 输入信号取值数据量较少时可以使用 initial 语句对输入信号的变化进行穷举式的描述。

```
`timescale 1ns/100ps
module testbench();
//定义输入、输出端口
reg 输入激励端口罗列;
wire 输出端口罗列;
…
//例化被测功能模块
…
//输入端加入激励信号
initial
begin
#延迟时间 begin
          输入激励信号赋值;
          end
#延迟时间 begin
          输入激励信号赋值;
          end
…
end
endmodule
```

例 3-80

```
`timescale 1ns / 100ps          //定义时间单位、时间精度
…
  Initial begin
        enable = 0;
   #4   enable = 1;             //延时 4 个时间单元后，enable 赋值为 1
   #10  enable = 0;             //延时 10 个时间单元后，enable 赋值为 0
   #5   enable = 1;             //延时 5 个时间单元后，enable 赋值为 1
  end
```

产生的波形如图 3-20 所示。

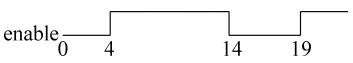


图 3-20　特定值的序列波形

(2) 激励数据量较多时,可以通过编写、调用任务和函数完成重复性操作,减少程序书写工作量。

例 3-81

```
`timescale 1ns / 100ps
module testbench;
…
//定义任务,以备重复调用,驱动被测试模块的输入端口
//进行一个字节数据的写入
task writeburst;
   input [7:0] wdata;
   begin
      @(posedge clockin) begin
         write_enable = #2 1;
         write_data = #2 wdata;
      end
   end
endtask
//可以调用低层任务,构筑更复杂的任务操作
//进行多个字节(本例中为 128 字节)的数据写入
task writeburst128;
begin
      writeburst(128); writeburst(129); writeburst(130); writeburst(131);
      writeburst(132); writeburst(133); writeburst(134); writeburst(135);
        …
end
…
//调用任务,进行较多测试数据的输入
Initial begin
     writeburst128;
 …
end
endmodule
```

(3) 如果输入的激励信号是视频码流等难以用手工进行输入的数据,就可以采用将需要输入的测试数据作为一个数据文件放于某文件目录下,调用系统函数读入数据,完成测试激励的输入,也可将得到的大量结果数据写到指定的文件中,以备后续分析。

例 3-82

```
`timescale 1ns / 100ps
module testbench;
integer i;
//例化被测功能模块
example u1(data_in,…);
 //定义一个寄存器组
 reg [width1 - 1:0] my_memory[width2 - 1:0];
//将数据文件中的值读入某寄存器中
initial begin
```

```
        $readmemb("mydata .dat", my_memory);
…
end
//使用数组中的数据作为输入激励
always @ (posedge clk)
begin
     data_in < = my_memory[i]
     i = < i + 1;
…
end
initial begin
   //打开一个文件,准备接收仿真的输出数据
   file_descriptor =  $fopen("simulus.dat");
…
$fwrite(file_descriptor , " % b\n",result);          //将仿真数据写入输出文件
…
$fclose(file_descriptor);
 end
```

本章小结

本章对 Verilog 语言的基本结构、基本语法进行了分类阐述,并采用 Verilog 语言进行电路设计举例,同时对测试文件的理解和设计进行了介绍。语言的学习应该在设计实践中去不断提高、体会。有以下学习建议:

(1) 明确设计目标后才开始编程。

任务的分析、分解是整个设计工作的核心和基础。在未充分理解设计目标、未完成任务的分析、分解时,就忙于编写代码,往往会做无用功,反而耽误开发进度,“磨刀不误砍柴工”。

(2) 用硬件电路系统的思想来编写 HDL。

首先要充分理解 HDL 语句和硬件电路的关系。HDL 就是在描述一个电路,每完成一段程序,就应当对生成的电路有一些大体上的了解;必须理解硬件“同时工作”的含义,在程序中描述的功能块往往是同时工作的,通常不会因为书写的顺序来决定其工作顺序(这和 C 语言是不同的)。

(3) 理解 HDL 的可综合性。

HDL 程序如果只用于仿真,那么几乎所有的 HDL 语句、函数、编程方法都可以使用。如果需要将文本描述转化为硬件实现,则必须保证程序“可综合”(即文本可以被综合工具转化成硬件电路)。不可综合的 HDL 语句在综合时将被忽略或者报错。“所有的 HDL 描述都可以用于仿真,但不是所有的 HDL 描述都能用硬件实现”。

(4) 语法掌握贵在精,不在多。

30%的基本 HDL 语句就可以完成 95%以上的电路设计,很多生僻的语句容易产生兼容性问题,也不利于其他人阅读和修改。学习中不需要花太多时间学全部的语句,而是着重理解常用的基本语句语法及其对应的硬件电路特点。本章只介绍了用 Verilog HDL 设计常用的语法,其他语法,读者可根据需要参考相关的手册。

习题

3.1　主要的 HDL 语言是哪两种？

3.2　Verilog HDL 语言的特点是什么？

3.3　定义以下 Verilog 变量：

(1) 一个名为 data_in 的 8 位向量线网；

(2) 一个名称为 MEM1 的存储器，含有 128 个数据，每个数据位宽为 8 位；

(3) 一个名为 data_out 的 16 位寄存器，其第 15 位为最低位。

3.4　设 A=4`b1010，B=4`b0011，C=1`b1，则下式运算结果是什么？

(1) &A

(2) ^A

(3) A >> 1

(4) {A,B[0],C}

(5) A & B

(6) A ^B

(7) A<B

3.5　设计一个时钟，要求：

(1) 可以对小时、分、秒进行计数；

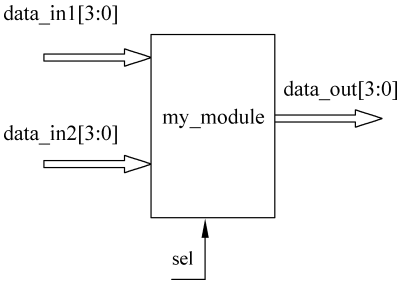
(2) 可以显示当前时间；

(3) 可以校对当前时间；

(4) 可以设置闹钟。

用“自顶向下”的设计思路分析系统，画出系统的模块组成情况(不必用语句进行具体设计)。

3.6　有一个模块名为 my_module，其输入、输出端口情况如题 3.6 图所示，试写出模块的定义、端口列表和端口定义(不必写出模块的内部语句)。



题 3.6 图

3.7　在下面的 initial 块中，每条语句在什么时刻开始执行？A、B、C、D 在仿真过程中和仿真结束时的值是什么？

```
initial
begin
```

```
    A = 1`b0; B = 1`b1; C = 2`b10; D = 4`b1100;
#10 begin
    A = 1`b1;B = 1`b0;end
#15 begin
    C = #5 2`b01;end
#10 begin
    D = #7 {A,B,C}; end
end
```

3.8 定义一个长度为 256、位宽为 2 的寄存器型数组，用 for 语句对该数组进行初始化，要求把所有的偶元素初始化为 0，所有的奇元素初始化为 1。

3.9 如果将例 3-62 中的非阻塞赋值改为阻塞赋值，程序实现的功能是否会发生改变？为什么？

3.10 设计一个移位函数，输入是一个 32 位的数 data 和一个左移、右移的控制信号 shift_ctrl，其输出是一个 32 位的数。

3.11 设计一个连加函数，输入的是起始数值和终止数值，输入和输出数据的位宽可由参数设定。

3.12 定义一个任务，该任务能计算出一个 8 位变量的偶校验位作为该任务的输出，计算结束后，经过三个时钟周期将该校验位赋给任务的输出。

3.13 设计一个周期为 40 个时间单位的时钟信号，其占空比为 20%，使用 always、initial 块进行设计，设初始时刻时钟信号为 0。

3.14 为什么应该尽量避免按照例 3-65 的方法用一个 always 模块来描述 Moore 型状态机？

3.15 设计例 3-64 中交通灯控制器的数据通道部分，并结合主控制模块的程序完成整个系统设计。

3.16 用 case、if-else、assign 语句分别设计四选一多路选择器，比较各种实现方式的特点。设本选择器的被选数据输入为 A、B、C、D，采用参数法定义这四个数据的位数，选择信号设为 sel，输出信号设为 data_sel，其关系如下表。

选择信号 sel[1:0]	输出信号 data_sel[width－1:0]
2`b00	data_sel[width－1:0] = A[width－1:0]
2`b01	data_sel[width－1:0] = B[width－1:0]
2`b10	data_sel[width－1:0] = C[width－1:0]
2`b11	data_sel[width－1:0] = D[width－1:0]

编写激励模块，对四选一选择器模块进行测试。