浙江大学信电学院

Verilog　HDL 简明教程

第二版

目录

第 1 章　初识 Verilog HDL

1.1　概述

硬件描述语言（Hardware Description Language，简称HDL）是硬件设计人员和电子设计自动化（EDA）工具之间的界面。其主要目的是用来编写设计文件，建立电子系统行为级的仿真模型。即利用计算机的巨大能力对用Verilog HDL或VHDL建模的复杂数字逻辑进行仿真，然后再自动综合以生成符合要求且在电路结构上可以实现的数字逻辑网表（Netlist）， 根据网表和某种工艺的器件自动生成具体电路，然后生成该工艺条件下这种具体电路的延时模型。仿真验证无误后用于制造ASIC芯片或写入EPLD和FPGA器件中。

自从Iverson于1962年提出HDL以来，许多高等学校、科研单位和大型计算机厂商都相继推出了各自的HDL，但最终成为IEEE技术标准的仅有两个即Verilog HDL和VHDL。

1.2　Verilog HDL 的基本结构

1.2.1 Verilog HDL 与 C 语言的比较

Verilog HDL是在C语言基础上发展起来的，从语法结构上，Verilog HDL语言与C语言有许多相似之处，并借鉴C语言的多种操作符和语法结构。为了便于对Verilog HDL有个初步认识，在此先将Verilog HDL与C语言的异同作一比较：

1) C语言由函数组成的，而Verilog HDL则是由与其相应的模块（module）组成。不过，Verilog HDL的模块就是实现一定功能的硬件电路。

2) C语言对函数的调用通过函数名相关联，函数之间的传值是通过端口变量实现的。相应地，Verilog HDL中的模块调用也是通过模块名相关联，模块之间的联系同样通过端口的连接实现，所不同的Verilog HDL反映的是硬件之间的真实物理连接。

3) C语言中有个main主函数，整个程序的执行从main函数开始。Verilog HDL没有相应命名的模块，且每一个模块都是等价的，但必定存在一个顶层模块，它的端口中包含芯片系统与外界的所有I/O信号。这个顶层模块从程序的组织结构上讲，类似于C语言的主函数，但Verilog HDL中的所有模块都是并发运行的，因为在实际硬件中许多操作都是在同一时刻发生的，这一点必须从本质上与C语言加以区别。

4) Verilog HDL语言有时序的概念，因为在硬件电路中从输入到输出总是有延迟存在的。

5) Verilog HDL语言既包含一些高层次程序设计语言的结构形式，同时也兼顾描述硬件电路具体的线路连接。

1.2.2 模块的概念和结构

作为一种高级语言，Verilog HDL语言以模块集合的形式来描述数字系统。模块是Verilog HDL语言的基本单元，由两部分组成，一部分描述接口，即与其它模块通信的外部输入/输出

端口；另一部分描述逻辑功能。一般说来一个文件就是一个模块，这些模块是并行运行的，但常用的做法是用包括测试数据和硬件描述的高层模块来定义一个封闭的系统，并在这一模块中调用其它模块的实例。

模块结构基本语法如下：

module <模块名>(<端口列表>)

参数定义（可选）

端口说明（input，output，inout）

数据类型定义

持续赋值语句（assign）

过程块（always和initial）--行为描述语句

低层模块实例—调用底层模块

任务和函数

延时说明块

endmodule

其中<模块名>是模块唯一性的标识符；<端口列表>是输入、输出和双向端口的列表，这些端口用来与其它模块进行连接；数据类型定义部分用来指定数据对象为连线型、寄存器型或存储器型；过程块包括always过程块和initial过程块两种，行为描述语句只能出现在这两种过程块内；延时说明块用来对模块的各个输入和输出端口之间的路径延时进行说明。

下面用图1.1所示的数据选择器的Verilog HDL描述来说明模块结构。

例1-1 数据选择器的Verilog HDL描述

```
module   mux2to1 (y, in0,in1, sel); //  模块名、端口列表
    //  参数定义，N表数据的位数，默认为1位。
    parameter   N=1;
     //端口说明。
     output [N-1:0]   y;
     input   [N-1:0]    in0,   in1;
     input     sel;
     //数据类型定义
     reg y;
     //过程块，描述逻辑功能
     always @(in0 or in1 or sel) //括号内为敏感信号表达式
       begin
         if    (sel)      y=in1;
         else             y=in0;
       end
endmodule
```

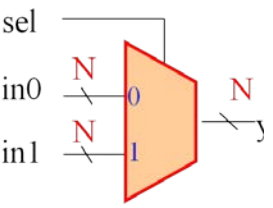


图 1.1 数据选择器

从上面的例子可以看出Verilog HDL语言的基本结构：

①Verilog HDL模块的内容包含在“module”和“endmodule”两个语句之间。

②首先要给出模块名，之后的括号内给出的是端口列表。本例模块取名mux2to1，参照图1.1，端口列表中端口即为硬件中外接引脚，模块通过这些端口与外界联系。

③参数定义，本例定义一个参数N，表示输入、输出数据的位数，默认N为1。这样，以后，其它模块可调用该模块时，可通过参数传递来改变参数N的值。参数传递的方法留在后面章节交代。

④端口说明，说明端口类型（input）、输出(output)和双向端口(inout)。

⑤数据类型定义，定义端口或变量的数据类型（wire、reg等）。

⑥描述逻辑功能描述，本例有用always过程块描述数据选择器的逻辑功能。

⑦用“//”可以进行单行注释，用“/*……*/”可以进行多行注释。另外，begin-end相当于C语言中的｛｝。

图1.2是例1-1的综合的RTL级电路结果，它与我们期望得到的结果是一到的。强调一下，虽然，Verilog HDL与C语言相似，但其用来描述硬件电路，因此在编写Verilog HDL代码时，要有与硬件对应起来的思维。

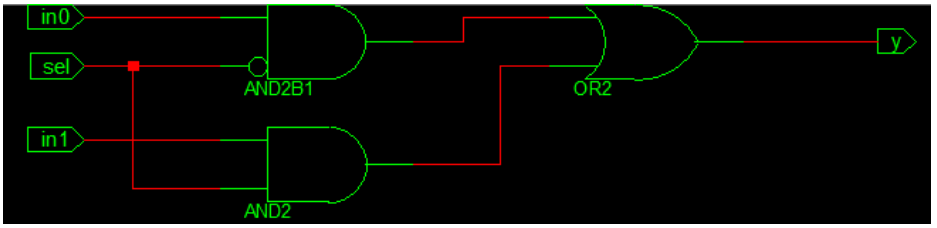


图 1.2　对例 1-1 用 ISE 软件综合结果

1.2.3 模块的描述方法

模块代表硬件上的逻辑实体，其范围可以从简单的门到整个大的系统，比如一个计数器、一个存储子系统、一个微处理器等。模块可以根据所采用的不同描述方法而分为三种：结构描述、数据流描述和行为描述，也可采用以上几种方式的组合。下面分别介绍模块的这几种描述方式。

1. 结构型描述

通过实例进行描述的方法，将Verilog HDL预定义的基本元件实例嵌入到语言中，监控实例的输入，一旦其中任何一个发生变化便重新运算并输出。

在Verilog HDL中可使用如下结构部件：

（1）　用户自定义的模块(创建层次结构)。

（2）　用户自定义元件UDP(在门级)。

（3）　内置门级元件(在门级)。

（4）　内置开关级元件(在晶体管级)

下面用内置门级元件为例介绍结构描述形式。Verilog HDL定义了26个内置门级元件，其中比较常用的有：

（1） 多输入门：and（与门）、nand（与非门）、or（或门）、nor（或非门）、xor（异或门）、xnor（同或门）等。

（2） 多输出门：buf（缓冲器）、not（非门）等。缓冲器、非门允许多个输出，但输出逻辑是一致的，这与集成门电路有不同。

（3） 三态门：bufif1（高电平使能）,bufif0（低电平使能）,notif1（高电平使能，三态非门） ,notif0（低电平使能，三态非门）。

门级元件的调用格式为：

（1） 多输入门：门类型 <实例名>(输出，输入1,输入2,…)

（2） 多输出门：门类型 <实例名>(输出1，输出2, …输入N,输入)

（3） 三态门： 门类型 <实例名>(输出，输入,使能输入)

1位数据选择器（N=1），可由图1.3所示的逻辑电路实现，例1-2即结构描述1位数据选择器的Verilog HDL代码。

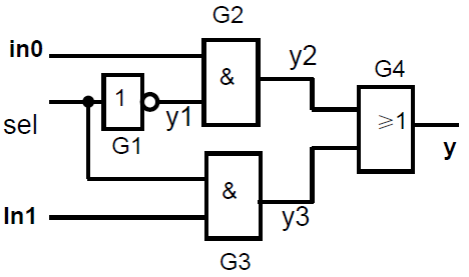


图1.3 MUX2_1的模块电路示意图

例1-2 结构型描述的例子

```
module mux2_1 (in0, in1, sel, y);          //端口定义
    input in0, in1, sel;                   //输入、输出列表
    output    y;
    wire   y1,y2,y3;                       //信号变量定义
    not G1(y1,select);                     //非门实例，实例名G1
    and G2(y2,in0,y1);                     //共有与门实例，实例名G2、G3
    and G3(y3,in1,select);
    or   G4(y,y2,y3);                      //或门实例，实例名G4
endmodule
```

2. 数据流型描述

是一种描述组合逻辑功能的方法，用assign持续赋值语句来实现。持续赋值语句完成如下的组合功能：等式右边的所有变量受持续监控，每当这些变量中有任何一个发生变化，整个表达式被重新赋值并送给等式左端。这种描述方法只能用来实现组合功能。例7-3为图7.1所示

数据选择器的数据流型描述。

例1-3 数据流型描述的例子

```
module   mux2to1 (out, in0,in1, sel); //  模块名、端口列表
    parameter   N=1; //  参数定义
    output [N-1:0]   y; //端口说明。
     input   [N-1:0]     in0,   in1;
     input     sel;
     assign y= select ? in1 : in0; //  数据流方式描述逻辑功能
endmodule
```

3. 行为型描述

是一种使用高级语言的方法,它和用软件编程语言描述没有什么不同,具有很强的可读性、通用性和有效性，它是通过描述硬件的行为特性来实现，它的关键词是always，其含义是一旦敏感信号表达式发生变化，就重新进行一次赋值，有无限循环之意。这种描述方法常用来实现时序电路，也可用来描述组合功能。例7-1为数据选择器的行为型描述。

4. 混合型描述

在模块中，用户可以混合使用上述三种描述方法。但需特别说明，模块中的门的实例、模块实例语句、assign语句和always语句是并发执行的，即执行顺序跟书写次序无关。

第 2 章 Verilog HDL 的基础知识

同其他高级语言一样，Verilog HDL具有自身固有的语法说明与定义格式，包括数字、字符串、标识符、运算符、数据类型和关键字等，这些有与C语言相同或相似之处，也有Verilog作为硬件描述所特有的地方，应注意比较Verilog HDL与C语言的区别。

2.1 词法

Verilog HDL 的源文本文件是由一串词法标识符构成的，一个词法标识符包含一个或若干个字符。源文件中这些标识符的排放格式很自由，也就是说，选择间隔符来分隔词法标识符。Verilog HDL 语言中词法标识符的类型有以下几种：

（1）间隔符与注释符
（2）数值
（3）字符串
（4）标识符和关键词

1. 间隔符

Verilog HDL 的间隔符包括空格、制表符、换行以及换页符。这些字符起到分割作用，使文本错落有致，可方便用户阅读和修改。

2. 注释符

Verilog HDL 支持两种形式的注释。

单行注释：以 “//”起始，以新的一行作为结束；

多行注释：是以“/*”起始，以“ */”结束。

3. 标识符

标识符是赋给对象的唯一的名字，用这个标识符来提及相应的对象。标识符可以是字母、数字、符号“$”和下划线“_”的任意组合序列，但它必须以字母（大小写）或下划线开头，不能是数字或“$”符开头。例如：temp_a、_shift 和 CLEAR 等都是合法的，而 2q_out，a*b 为不合法。

标识符是区分大小写，例 qout 与 QOUT 是不同的标识符。

扩展标识符以反斜杠“\”开始,以空格结束,这种命名可以包含任何可印刷的 ASCII 字符,反斜杠和空格不属于名称的一部分。

4. 关键字

关键字也称保留字，它是 Verilog HDL 语言的专用字，所有的关键字都是用小写形式，不能将关键字用作标识符。表 2.1 结出了 Verilog HDL 中常用的关键字，其中最后一个 mux 被 ISE、ModelSim 等 EDA 软件列为保留字。

表 2.1 Verilog HDL 中常用的关键字

always	endfunction	medium	real	time
and	endprimitive	module	realtime	tran
assign	endspecify	nand	reg	tranif0
begin	endtable	negedge	release	tranif1
buf	endtask	nmos	repeat	tritri0
bufif0	event	nor	rnmos	tri1
bufif1	for	not	rpmos	triand
case	force	notif0	rtran	trior
casex	forever	notif1	rtranif0	trireg
casez	fork	or	rtranif1	vectored
cmos	function	output	scalared	wait
deassign	highz0	parameter	small	wand
default	highz1	pmos	specify	weak0
defparam	if	posedge	specparam	weak1
disable	inout	primitive	strong0	while
edge	input	pull0	strong1	wire
else	integer	pull1	supply0	wor
end	join	pullup	supply1	xnor
endcase	large	pulldown	table	xor
endmodule	macromodule	rcmos	task	mux

2.2 常量

Verilog HDL有下列四种基本的值：

1) 0：逻辑0或“假”

2) 1：逻辑1或“真”

3) x：不定值

4) z：高阻态

Verilog HDL有整数、实数两类数值常量和字符串常量。

1. 整数常量

在Verilog HDL中，整数有二进制（b或B）、八进制（o或O）、十六进制（h或H）或十进制（d或D）四种进制表示方式，格式为：

<位宽>'<进制><数值>

其中，位宽为对应二进制数的位数，该项可选。当位宽小于相应数值的实际位数时，相应的高位部分被忽略，这一点值得注意；当位宽大于数值的实际位数时，相应的高位部分补0（数值的最高位为1或0）或补x（数值的最高位为x）或补z（数值的最高位为z）。如：

```
8'b1100_0101    //位宽为8位的二进制数11000101
8'hb5           //位宽为8位的十六进制数b5
'b01x           //缺省位宽的3位二进制数01x
6'hb5           //位宽为6位的十六进制数35，因为位宽不够，高位忽略（1011_0101）
8'b100101       //位宽为8位的二进制数00100101，位宽太多，高位补0
2a0             //非法的整数表示（十六进制数字表示需'h）
```

另外，对于十进制数，可缺省位宽和进制说明，如：

```
98              //简单的十进制数格式，代表十进制数98。
```

注意：

- x（或z）在十六进制值中代表4位x（或z），在八进制中代表3位x（或z），在二进制中代表1位x（或z），x（或z）不能出现在十进制中。
- 数值常量中的下划线“_”是为了增加可读性，不会影响数值大小，但不能放在数值常量的首位。

2. 实数常量

Verilog HDL中实数可以用下列两种形式定义：

（1） 十进制计数法，但数点两侧都必须至少有一位数字。例如

```
2.0
5.678
0.1
2.      // 非法，小数点两侧必须有1位数字
```

（2） 科学计数法； 这种形式的实数举例如下：

23_5.1e2 其值为23510.0;下划线可忽略

3.6E2 其值为360.0 (e与E相同)

5 E-4 其值为0 . 0 0 0 5

注意，Verilog HDL语言定义了实数如何隐式地转换为整数。实数通过四舍五入被转换为最相近的整数。

3. 字符串常量

字符串常量是一行写在双引号之间的字符序列串，若字符串用作Verilog HDL表达式或操作数，则字符串被看作8位ASCII值的序列，即一个字符对应8位ASCII值。例如：字符串“ab”等价于16'h57_58。

字符串变量是寄存器型变量，它具有与字符串的字符数乘以8相等的位宽，为存储字符串 ”Hello” 变量需要8*5，即40位的寄存器。

Verilog HDL支持C语言中的转意符，如\n （换行符）、\ t （制表符）、\\ （字符\）、\" 和\%。

2.3 数据类型和变量

Verilog HDL的数据类型集合表示在硬件数字电路中数据进行存储和传输的要素。Verilog HDL不仅支持抽象数据类型的变量如整型变量、实型变量等；同时也支持物理数据类型的变量，可代表真实的硬件。

Verilog HDL中的变量的数据类型分为连线型（nets type）和寄存器型（register type）两大类，共19种数据类型，本章节只介绍常用的5种 。

1 . wire 型

wire型属连线型变量，对应的是硬件电路中的物理信号连线，没有电荷保持作用。它的值始终根据驱动的变化而更新。有两种方法对它进行驱动：一是在结构描述值把它连接到门或模块的输出端；二是用持续赋值语句assign语句对其赋值。如果没有驱动源对其赋值，连线型的缺省值为z。

wire型类型信号的定义格式如下：

wire [msb:lsb] net1, net2, . . . , net N;

其中， msb和lsb 是用于定义连线型位宽的常量表达式；范围定义是可选的；如果没有定义范围，缺省的为1位wire型信号。下面是连线型类型说明实例。

wire *a*,*b*; //定义2个1位的wire型变量a，b。

wire [7:0] *Addrbus*; //定义一个8位的wire型地址向量*Addrbus*。

2. reg 型

reg型属寄存器变量,寄存器类型数据对应的是具有状态保持作用的硬件电路，如触发器、锁存器等。若寄存器类型的变量未被初始化，缺省值为x。寄存器类型与连线型的区别在于：

寄存器类型数据保持最后一次赋值，而连线型数据需持续驱动。

寄存器类型数据只能在always语句和initial语句中被赋值。反之，在always语句和initial语句中被赋值每一个信号必须定义寄存器类型。

寄存器数据类型re g定义形式如下：

reg [msb: lsb] reg1, reg2, . . . reg N;

msb和lsb 定义了位宽，并且均为常数值表达式。范围定义是可选的；如果没有定义范围，缺省值为1位寄存器。例如：

reg [3:0] count; //定义一个4 位寄存器向量count。

reg a,b; //定义两1 位寄存器a和b。

寄存器可以取任意长度。寄存器中的值通常被解释为无符号数, 但寄存器类型的值可取负数，但若该变量用于表达式的运算中，则按无符号类型处理 例如：

reg [3:0] A;

...

A = －2; // A总按无符号数14来看待， 14（1110）是-2的补码。

特别要说明一下，reg类型不等同于的硬件电路的触发器和锁存器，组合电路的输出也可定义为reg类型。例7.1中数据选择器输出就定义为reg类型，因为其在always过程块中被赋值。

3. 存储器型变量

若干个相同位宽的寄存器向量构成寄存器数组即为存储器变量（memory type）。存储器型变量使用如下方式说明：

reg [msb: 1sb] mem1 [upper1:lower1],mem2[upper2:lower2],... ;

例如：

reg [7:0] MyMem [1023 : 0] ;

上面的语句定义了一个1024字节的8位存储器MyMem。

数组的维数不能大于2。注意存储器属于寄存器数组类型。连线型数据类型没有相应的存储器类型，因此也没有多维连线型数据的数组。

4. 参数常量

在Verilog HDL中，用paramete语句来定义常量，即用paramete语句来定义一个标识符，代表一个常量，称为参数常量。参数经常用于定义时延和变量的宽度，提高程序的可读性和可维护性。

参数常量只能被赋值一次，参数说明形式如下：

parameter param1=const_ expr1, param2=const_expr2,...,;

下面为具体实例：

parameter CounterBits=4; // 定义了计数器的位数CounterBits

parameter S0=3, S1=1, S2=0, S3=2; //定义了四个状态

Parameter InDataWidth=8, OutDataWidth _size= in_size*2;// 定义输入、输出数据宽度

5. integer 型

integer型属寄存器变量、是纯数学抽象的数据类型，不对应任何具体的硬件电路。integer

型为32位带符号整数型变量，常用作循环控制变量。

2.4 模块端口类型

模块端口是模块与外界交互信息的接口，包括3种类型。

（1） input：输入信号端口，在模块内不能对input信号赋值。

（2） output：输出信号端口。

（3） inout：双向I/O端口。

2.5 运算符及优先级

2.5.1 运算符

Verilog HDL定义了许多运算符中，按功能可以分为算术运算符、逻辑运算符、关系运算符等九大类。表2.2详细给出了Verilog HDL中定义的运算符分类及简单功能说明

表2.2 Verilog HDL中的运算符分类及功能说明

类型	运算符	简单说明
算术运算符	+、 _、 *、 / %	加、减、乘、除 取模
逻辑运算符	！、 &&、 ‖	逻辑非、逻辑与、逻辑或
按位运算符	~、 &、|、 ^ ^~ 或 ~^	按位非、按位与、按位或、按位异或 按位异或非
关系运算符	<、 <=、 >、 >=	小于、小于等于、大于、大于等于
等式运算符	==、 != ===、 !==	逻辑相等、逻辑不等 全等、非全等
归约运算符	&、 ~& |、 ~| ^、 ^~ 或~^	归约与、归约与非 归约或、归约或非 归约异或、归约异或非
移位运算符	<<、>>	左移、右移
条件运算符	?:	条件
位拼接运算符	{ }	连接

1.算术运算符

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

（1） + （加法运算符，或正值符号，如 a+b，+5）

（2） - （减法运算符，或负值符号，如 a-b，-5）

（3） * （乘法运算符，如 a*2）

（4） / （除法运算符，如 a/5）

（5） % （求模运算符，或称求余运算符，要求%两侧均为整型数据，如 8%3 的值为 2）

注意：

（1）在进行算术运算操作时，如果某一操作数有不确定的值， 则运算结果也是不定值。

（2）在进行整数除法运算时，结果值要略去小数部分，只取整数部分例8/3=2 。

2.关系运算符

关系运算符为二目运算符，共有 4 种：

（1） > （大于）

（2） > = （大于等于）

（3） < （小于）

（4） < = （小于等于）

在进行关系运算时，如果操作数之间的关系成立返回值为 1； 反之关系不成立则返回值为 0； 若某一个操作数的值不定 x 或高阻 z，则关系是模糊的，返回值是不定值 x。

3.等式运算符

等式运算符为二目运算符，有 4 种，分别为：

（1） = = 相等

（2）! = 不相等

（3）= = = 全等

（4）! = = 非全等

注意：

（1）在= =（相等）运算或=（不相等）运算时，如果任何一个操作数中的某一位为不定值x或高阻z，则结果为不定值x。例：

1z1x01 = =1z1x01的结果为x；

1z1x01 ! =1z1x01的结果为x；。

（2）在= = =（全等）运算时，其比较过程与= =（相等）相同，全等运算时将不定值x或高阻z看作是逻辑状态的一种参与比较，因此，全等运算返回的结果只有逻辑0或逻辑1两种。! = =（非全等）与= = =（全等）正好相反，例：

1z1x01= = =1z1x01 的结果为 1；

101x01= = =1z1x01 的结果为 0；

101x01= = =101z01 的结果为 0。

（3）! = =（非全等）与= = =（全等）正好相反，这里不再赘述。

4. 逻辑运算符

在Verilog HDL语言中有3 种逻辑运算符：

（1） ! 逻辑非

（2） && 逻辑与

（3） || 逻辑或

&&和|| 是二目运算符，要求有两个操作数，如（a > b） && （b > c）。而!是单目运算符只要求一个操作数，！（a > b）。

5. 按位逻辑运算符

在Verilog HDL语言中有5 种按位逻辑运算符

（1） ~ 按位取反

（2） & 按位与

（3） | 按位或

（4）^ 按位异或

（5）^~ ，~^ 按位同或

按位逻辑运算符对其操作数的每一位进行操作，例如表达式a&b 的结果是a和b的对应位相与的值。对具有不定值的位进行操作，视情况而定会得到不同的结果。例如;x和0相或得结果x; x和1相或得结果1 。 如果操作数的长度不相等，较短的操作数将用0来补高位，逐位运算将返回一个与两个操作数中位宽较大的一个等宽的值。

在此需要注意的是不要将逻辑运算符和按位运算符相混淆，比如，!是逻辑非，而~是按位取反，例如，对于前者!(5= =6) 结果是1，后者对位进行操作，~1011=0100。

6. 缩减运算符

缩减运算符是单目运算符，它包括下面几种：

（1） & 缩减与

（2） ~& 缩减与非

（3） | 缩减或

（4） ~| 缩减或非

（5）^ 缩减异或

（6）^~ ，~^ 缩减同或

缩减运算的运算过程是：先将操作数的第 1 位与第 2 位进行与、或、非运算，然后将运算结果与第 3 位进行与、或、非运算，依次类推直至最后一位；最后的运算结果是 1 位的二进制数。例如：

```
reg[3:0]  a;
b=&a;        //等效为b=a[0] & a[1] & a[2] & a[3] 。
```

7. 移位运算符

在Verilog HDL语言中有两种移位运算符（左移位运算符）和>>（右移位运算符）。其用法为：

A << n 或>> n

表示是将第一个操作数A向左或右移n位，同时用0 来填补移出的空位。举例如下：

若A=5’11001 则 A << 2为的值5’ 00100。

8. 条件运算符

Verilog HDL条件运算符为：

?:

条件运算有三个操作数,其定义同 C 语句定义一样，格式如下：

signal = *condition _expr* ? true_ *expr* : false_*expr2*

当条成立（condition _expr为true）时，信号（signal）取第一个表达式的值，即true_ expr；反之取第二个表达式的值，即false_expr2。

例如对2选1的数据选择器，可描述如下：

out = select ? in1 : in0 ;　　// select=1时out = in1；select=0时out = in0。

9.拼接运算符

Verilog　HDL语言中有一个特殊的运算符：位拼接运算符{}。这一运算符可以将两个或更多个信号的某些位拼接起来进行运算操作。用法如下：

{信号1的某几位,信号2的某几位,……信号n 的某几位}

例 2-1 拼接符的 Verilog 实例—左移移位寄存器

```
reg[15:0] q;
always @(posedge clk)
  q[15:0]={ q[14:0],DataIn}
```

2.5.2 运算符优先级排序

运算符优先级顺序如下表 2.3 所示，为避免出错，同时为增加程序可读性，在书写程序时可用括号()来控制运算的优先级。

表2.3　运算符的优先级

优先级	运算符	简单说明
高优先级	！、~	逻辑非、按位非
	& 、 ~&、| 、 ~|、^ 、 ^~ 或~^	缩减运算符
	+、 _	正、负号
	{ }	位拼接运算符
	*、 /、%	算术运算符
	+、 _	
	<< 、>>	移位运算符
	< 、 <= 、 > 、 >=	关系运算符
	== 、 !=、=== 、 !==	等式运算符
	&	按位与
	^ 、 ^~ 或~^	按位异或、按位同或
	|	按位或
	&&	逻辑与
	||	逻辑或
低优先级	?:	条件运算符

2..6 编译预处理指令

与 C 语言的编译指令相似，Verilog HDL 中允许在程序中使用特殊编译指令。编译时，通常先对这些编译指令进行“预处理”，然后再与源程序一起进行编译。Verilog HDL 语言

提供了十多条的编译指令，本节只介绍常用的`define、`include 和`timescale 三条常用编译指令。在详细介绍之前，有必要先进行以下几点说明：

1）编译预处理指令以`（反引号）开头。

2）编译预处理指令非 Verilog HDL 的描述，因而编译预处理指令结束不需要加分号。

3）编译预处理指令不受模块与文件的限制。在进行 Verilog HDL 语言编译时，已定义的编译预处理指令一直有效，直至有其它编译预处理指令修改它或取消它。

1.宏编译指令 `define

`define 指令用于文本替换，它很像 C 语言中的#define 指令，语法格式如下:

`define 宏名 字符串

例`define wordsize 8

以上语句中，用易懂的宏名 wordsize 来代替抽象的数字 8；要采用了这样的定义后，在编译过程，一旦遇到 wordsize 则用 8 来代替。

2.文件包含指令 `include

`include 编译器指令用于嵌入内嵌文件的内容。文件既可以用相对路径名定义，也可以用全路径名定义, 例如:

` include " . . / . . /primitives.v "

编译时，这一行由文件“../ ../ primitives . v” 的内容替代。

3.时间定标指令`timescale

在 Verilog HDL 语言的模型中，所有时延都用单位时间表述的，且是一个相对的概念。`timescale 编译指令用于定义计时单位与精度单位。实际时间相关联。`timescale 编译指令的格式为：

`timescale <计时单位> / <精度单位>

其中时间度量有：s、ms、us、ns、ps(10^{-12}s)和 fs(10^{-15}s)，计时单位和精度单位时延精度精度间精度只能取 1，10 或 100，且计时单位必须大于精度单位。例如：

` timescale 1ns / 100 ps //表示计时单位为 1ns, 精度单位 100ps。

` timescale 1ps / 10 ps //非法定标，因为时单位必须小于精度单位。

第 3 章 verilog HDL 语言的描述语句

在Verilog HDL中，可以使用多种建模的方法，这些建模方法称为描述风格。最常用的三种描述风格为：数据流描述、行为描述和结构描述。大多采用它们组合成的混合描述。

3.1 数据流描述风格：assign 语句

数据流描述采用持续赋值语句即assign语句，它用于给wire等nets型变量进行赋值。当组合电路已有表达式或逻辑电路图，适合用assign语句描述。

下面是采用数据流方式描述一位全加器，一位全加器逻辑表达如式3.1表示。

$$\begin{cases} s = a \oplus b \oplus ci \\ co = a \bullet b + a \bullet ci + b \bullet ci \end{cases} \tag{3.1}$$

例3-1 一位全加器的数据流描述

```
module   full_adder_1bit (a, b, s,cin, co ) ;
  input    a, b, cin;
  output   s, co ;
  assign   s = a ^ b ^ cin;
  assign   co = (a && cin) || (b && cin )|| (a && b);
endmodule
```

在本例中，有两个持续赋值语句。这些赋值语句是并发的，与其书写的顺序无关。只要持续赋值语句右端表达式中操作数的值变化(即有事件发生), 持续赋值语句即被执行。

3.2 行为描述风格及主要描述语句

Verilog HDL支持许多高级行为语句，使其成为结构性和行为性语言。行为描述主要包括过程结构、语句块、时序控制和流控制的模块。

3.2.1 过程结构

下述两种语句是为一个设计的行为建模的主要机制。

1) initial 语句

2) always语句

一个模块中可以包含任意多个initial或always语句。这些语句相互并行执行，即这些语句的执行顺序与其在模块中的顺序无关。一个initial语句或always语句的执行产生一个单独的控制流，所有的initial 和always语句在0时刻开始并行执行。

initial或always语句是不能嵌套使用。

1.always 语句

always语句语法如下：

```
always   @(<敏感信号表达式>)
begin
   //过程赋值
   //if语句
   //case语句
   //while,repert,for 语句
   //task,functiony调用
end
```

1）敏感信号表达式

敏感信号表达式又称敏感事件表，只要该表达式的值发生变化，就会执行块内的语句，

因此always语句有无限循环意义。因此敏感信号表达式列出所有影响块内变量取值的输入信号，若有两个或两个以上敏感信号时，它们之间用or连接。

例如四选一数据选择器，只要四个输入信号in0、in1、in2、in3和地址信号sel[1：0]有一个发生改变，输出就会改变，所以四选一数据选择器的敏感信号表达式为：

always @(in0 or in1 or in2 or in3 or sel)

组合电路的敏感信号表达式可用“*”代替，表示只要等号右边的值发生变化，就进行赋值运算，上述四选一数据选择器的敏感信号表达式可表示为：

always @(*)

2）posedge与negedge关键字

对于时序电路，事件是往往由时钟边沿触发的。为表达边沿这个概念，Verilog HDL语言提供了posedge与negedge两个关键字来描述。如用posedge clk表示时钟信号clk的上升沿，而negedge clk表示时钟信号clk的下降沿。对于异步的清零/置数，应按以下格式书写敏感信号表达式：

always @(posedge clk or negedge clrn) //低电平清零有效

always @(posedge clk or posedge clr) //高电平清零有效

例3-2就是用if语句描述的带异步清零的D触发器，清零输入r低电平有效。

例3-2 带异步清零的D触发器

```
module dffr(q,d,clk,r);
    input d, clk, r;
    output q;
    reg     q ;
    always @(posedge clk or posedge r ) //异步清0，高电平有效。
    begin
        if (r)          q=0 ;
         else  q =d;  //同步输入。
    end
endmodule
```

在这个例子中，敏感信号表达式中没有列出输入信号d，这是因为d为同步输入信号，只能在时钟上升沿到来时起作用。从语句描述可看出，只要r为高电平，q就为0；而同步输入信号d，只有在clk上升沿且r无效时才能赋值给q。

注意，对于异步的清零/置数控制时，块内逻辑描述要与敏感信号表达式的有效电平一致。例3-3描述是错误的。

例3-3：带异步清0的上升沿D触发器

```
always  @(posedge clk or negedge r) //低电平清零有效
      begin
      if（r）     //与敏感信号表达式的有效电平矛盾，应改为if(!r)
            qout=0；
```

```
    else
        qout=in;
    end
```

在高速数字系统中，很少采用异步清零/置数操作，更多采用同步清零/置数操作，因此，同步时序电路的的敏感信号表达式大多采用下面两种其中之一。

```
always  @(posedge clk)    //时钟上升沿触发
always  @(negedge clk)    //时钟下降沿触发
```

3)语句块

Verilog HDL的语句分为两种：

串行语句块（begin...end语句组），begin...end作用类似C语言中的｛｝，用来组合需要顺序执行中的语句，串行语句块内的各条语句是按它们出现的次序逐条顺序执行。

并行语句块（fork ...join语句组），fork ...join作用类似C语言中的｛｝，用来组合需要并行执行中的语句，即语句并行块内的各条语句同时执行的。

2. initial 语句

initial过程赋值语句只执行一次，即在0时刻开始执行。主要面向功能仿真模拟，通常不具有可综合性。initial语句通常用来描述测试模块的初始化、监视、波形生成等功能行为。initial过程赋值语句的语法如下：

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

initial语句多用于测试代码的编写，下面就是例3-1所描述的一位全加器的测试代码，其输入波形如图3.1所示。

例3-4 全加器的测试代码

```
`timescale 1ns / 1ps
module full_adder_tb_v;
    // Inputs
    reg a,b,ci;
    // Outputs
    wire s;
    wire co;
    //  全加器实例
    full_adder_1bit    uut (.a(a),.b(b),.s(s),.ci(ci),.co(co));
    initial begin
        //  每隔100ns给输入a,b,ci赋一组值
```

```
            a = 0;b = 0;ci = 0;
        #100    a = 0;b = 0;ci = 1;
        #100    a = 0;b = 1;ci = 0;
        #100    a = 0;b = 1;ci = 1;
        #100    a = 1;b = 0;ci = 0;
        #100    a = 1;b = 0;ci = 1;
        #100    a = 1;b = 1;ci = 0;
        #100    a = 1;b = 1;ci = 1;
        #100    $stop;
    end
endmodule
```

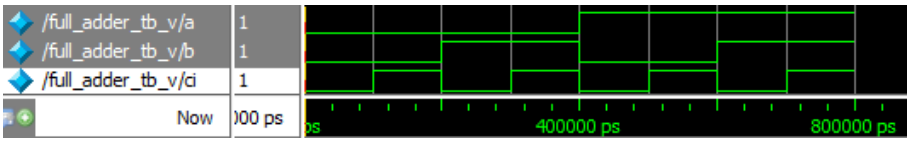


图3.1 例3-4所描述的输入信号波形

initial过程赋值语句也可以用为硬件功能模块中的reg变量赋初值。下面给出的例子，initial语句用于对reg变量和存储器变量初始化。

例3-5 initial语句用于对reg变量和存储器变量初始化

```
parameter   SIZE = 1024;
reg [7:0]   RAM [SIZE- 1 : 0] ;
initial
begin:
    integer   i;        //reg变量初始化
    for   (i = 0; i< SIZE; i =i + 1)
    RAM [i] = 0;       //存储器变量初始化
end
```

这个例子的作用是用initial语句在仿真开始时刻对reg变量和存储器赋0值，从而完成对各个变量的初始化。

3.2.2 过程赋值语句

过程赋值是在initial语句或always语句内的赋值，过程赋值只能对寄存器型的变量（reg、integer、memory型等）赋值。过程赋值分两类：

(1) 阻塞性过程赋值

赋值操作符是“=”， 表达式的右端可以是任何表达式。阻塞赋值在该语句结束时执行赋值，前面的语句没有完成前，后面的语句是不能执行，因此begin...end语句组内的阻塞赋值语句是顺序执行。

(2) 非阻塞性过程赋值

在非阻塞性过程赋值中，使用赋值符号“ <=”。在begin...end语句组内，一条非阻塞赋

值语句的执行是不会阻塞下一条语句的执行，也就是说本条非阻塞赋值语句的执行完毕前，下一条语句也开始执行。

下面两个例子可说明阻塞赋值和非阻塞赋值的区别。

例3-6 非阻塞赋值

```
module non_block (c, a,b,clk);
output   c,b;
input    a,clk;
reg      c,b;
always @(posedge clk)
    begin
        b<=a;// 非阻塞赋值
        c<=b;
    end
endmodule
```

例3-7 阻塞赋值

```
module   block (c, a,b,clk);
output   c,b;
input    a,clk;
reg      c,b;
always @(posedge clk)
    begin
        b=a;// 阻塞赋值
        c=b;
    end
endmodule
```

将上面两个例子用ModelSim进行仿真，可分别得到图3.2（非阻塞赋值）和图3.3（阻塞赋值）和所示的仿真波形。

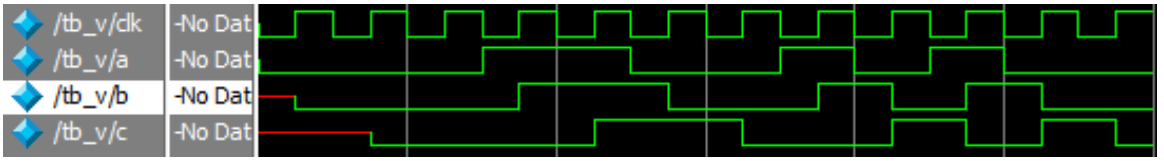


图 3.2 例 7-10 非阻塞赋值的仿真波形

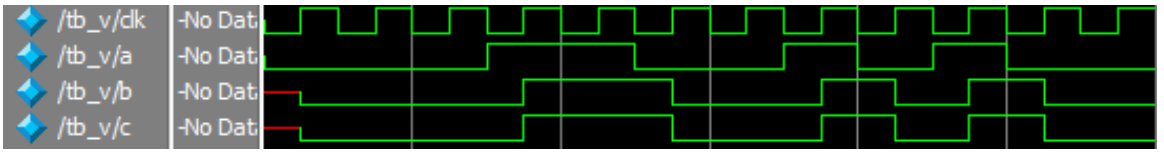


图 3.3 例 7-11 阻塞赋值的仿真波形

由仿真波形可得出，非阻塞赋值的两条语句是同时执行，而阻塞赋值的两条语句是顺序执行。相应地，这两种描述所对应的电路如图3.4和图3.5所示。

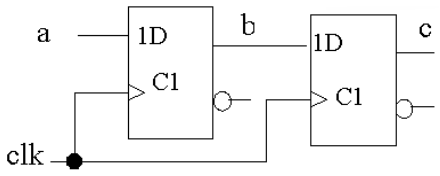


图3.4 非阻塞赋值描述的电路

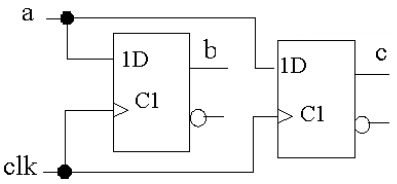


图3.5 阻塞赋值描述的电路

为避免对这两种赋值的错误应用，建议同学们尽量使用阻塞性过程赋值“=”，因为它类似C语言的赋值方式。

3.2.3 条件分支语句

条件分支语句有 if-else 语句和 csse 语句两种。下面将对这两种分支语句进行讨论。

1.if-else 语句

if-else语句是两分支语句，使用方法与C语言类似，语法有以下三种：

1）if （<条件表达式>）语句或语句块；

2）if （<条件表达式>）语句或语句块1 ；

else 语句或语句块2 ；

3）if （<条件表达式1>）语句或语句块1 ；

else if （<条件表达式2>）语句或语句块2 ；

……

else if （<条件表达式n>）语句或语句块n ；

else 语句或语句块n+1；

这三种方式中，“条件表达式”一般为逻辑表达式、关系表达式或1位逻辑变量。系统对表达式的值进行判断，若为0、不定值x、高阻z，作“假”处理；若为1，按“真”处理。

2. case 语句

case语句是一个多路条件分支语句，多用于描述多条件译码电路，如译码器、数据选择器、状态机及微处理机的指令译码等。case语句有case、casex和casez三种形式，下面分别予以介绍。

1） case语句

case语句其语法如下：

```
case （<控制表达式>）
        值1：语句或语句块1 ；      //case分支项。
        值2：语句或语句块2 ；
         ……
        值n：语句或语句块n ；
        default：语句或语句块n+1；       // default语句不是必需。
endcase
```

当“控制表达式”的值为值1时，执行语句或语句块1 ；为值2时，执行语句或语句块2；……依此类推；当“控制表达式”的值与所列出的值都不相等时，则执行default后面的语句。例3-8为用case语句描述4选1数据选择器。

例3-8 用case语句描述数据选择器

```
module mux4to1(out , in0 , in1 , in2 , in3 , sel);
          parameter   N=8;
          output[N:1] out;
          input[N:1]     in0,in1, in2,in3;
          input [1:0]   sel;
          reg[N:1] out;
          always @(in0 or in1 or in2 or in3 or sel)
```

```
    begin
      case (sel)
          0:      out = in0;
          1:      out = in1;
          2:      out = in2;
          default:   out = in3;        //可用2'b11: out = in3;代替。
      endcase
     end
endmodule
```

图3.6为用ISE软件综合的RTL级结果，符合设计结果。

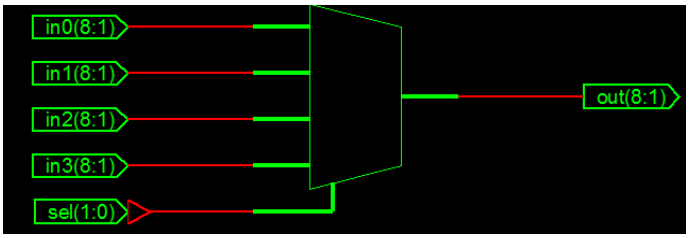


图3.6　数据选择器的ISE综合结果

2）casez与casex语句

除关键字casex和casez以外，casex和casez这两种形式的语法结构与case语句完全一致。case、casex和casez的区别在于对x和z值使用不同的解释，即在比较控制表达式或分支表达式的值时，在casez语句中，对取值为z的某些位比较不予考虑，因此只需关注其它位的比较结果；而在casex语句中，对取值为z和x的某些位的比较不予考虑。

例3-9为用casex语句描述的4线-2线高优先编码器。图3.7为编码器的端口示意图，输入d3~d0高电平有效。outcode为编码输出，而none_on为输入无效标记，one_on高电平表示输入无效，即d3~d0均为低电平。

例3-9　　4线-2线高优先编码器

```
module encode4to2(none_on,outcode, d3,d2,d1,d0);
output[1:0]          outcode;//编码输出
output               none_on;
input                d3,d2,d1,d0;
reg[3:0]             out_temp;
assign {none_on, outcode} = out_temp;
always @(*)
      begin
        casex ({d3,d2,d1,d0})
        4'B1???     : out_temp=3'b0_11;   //可用?来标识x或z。
        4'B01??     : out_temp=3'b0_10;
        4'B001?     : out_temp=3'b0_01;
```

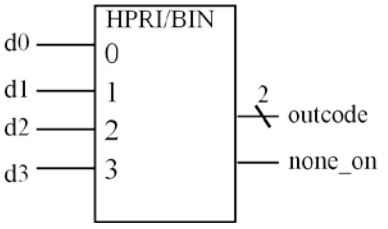


图 3.7.　高优先编码器

```
        4'B0001    : out_temp=3'b0_00;
        4'B0000    : out_temp=3'b1_00; //输入无效
        endcase
    end
endmodule
```

图3-8为高优先编码器的ISE软件综合结果。

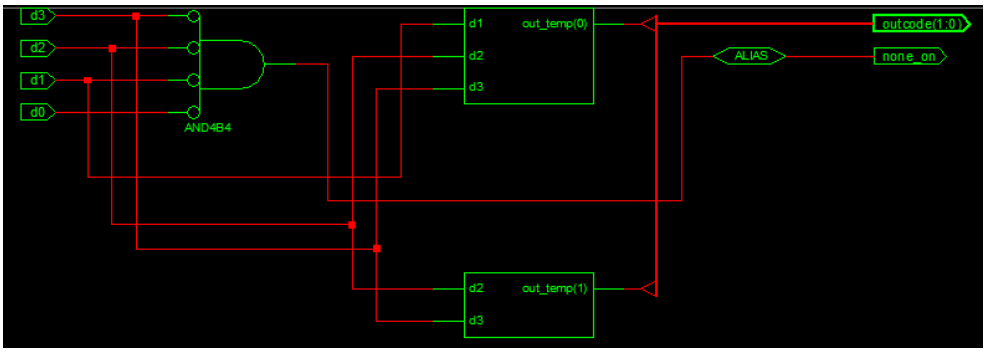


图3.8　高优先编码器的ISE综合结果

3. 使用条件语句注意事项

在使用条件语句设计时，应注意列出所有条件分支，否则编译器会认为条件不满足时，会引入触发器来保持原值。时序电路设计时，可利用这一点来进行状态保持；而在设计组合电路时，应避免这种隐含触发器的存在。

另外，有些情况下，很难列出所有条件分支，因此可在if语句最后加上else；在case语句的最后加上default语句。

3.2.4 循环控制语句

循环控制语句包括for循环语句、while循环语句、repeat循环语句和forever循环语句，循环控制语句多用于测试代码的编写。在逻辑电路描述中适合用循环语句范围较小，且不易掌握，因此，同学们应**慎用循环语句**描述逻辑电路。下面介绍常用两种：for循环语句和repeat循环语句。

1. for 循环语句

循环语句与C语言的for循环语句非常相似，只是Verilog　HDL中没有增1++和减1- -运算符，因此要使用i=i+1 的形式。for 循环语句的形式如下：

for(循环变量赋初值；循环结束条件；循环变量增值)

循环体语句或语句块；

for循环语句执行过程可分如下几步：

1）执行“循环变量赋初值”

2）判断“循环结束条件”表达式：若“循环结束条件”取值为真，则执行“循环体语句或语句块”,然后继续执行3；若“循环结束条件”取值为假，则循环结束，退出for循环语句的执行。

3)执行“循环变量增值”语句,转到第2步继续执行。

下面通过设计一个8位串行加法器，来说明for循环语句的使用。图3.9 为8位串行加法器原理图，由8个一位全加器级联组成，即低位的进位向高位传递的方法。

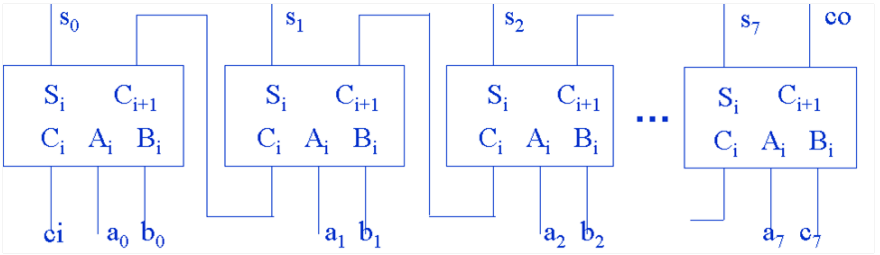


图3.9 串行加法器原理

例3-10 8位串行加法器

```
module   full_adder (a, b, s, ci, co);
   parameter N=8; //加法器的位数 N
   input [N-1:0] a;
   input [N-1:0] b;
   output [N-1:0] s;
   input ci;
   output co;
   wire co;
   integer i;//循环变量
   reg   [N-1:0] s;
   reg   [N:0] c; //暂存一位加法器的进位
   assign co=c[N];
   always @(*)
     begin
        c[0]=ci;
        for (i=0;i<=N-1;i=i+1)
          begin
            s[i]=a[i]^b[i]^c[i];
            c[i+1]=a[i]&& b[i] ||a[i]&&c[i] ||b[i]&&c[i];
          end
     end
endmodule
```

用 ISE 软件综合串行加法器，其 RTL 级的综合结果如图 3.10 所示，从中可看出，综合结果符合设计要求。

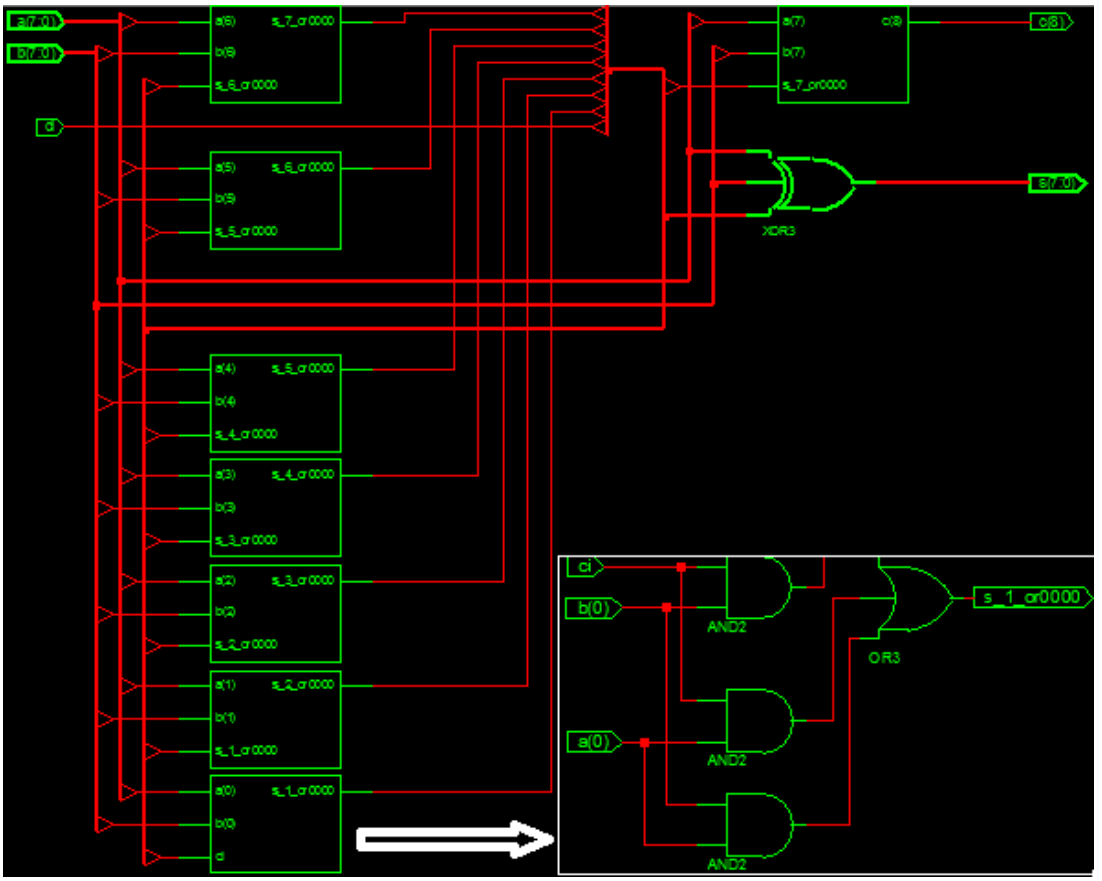


图 3.10 串行加法器 RTL 级综合结果

2. repeat 循环语句

repeat循环语句实现的是一种循环次数预先指定的循环， Repeat循环语句的格式如下：

repeat（<循环次数表达式>）语句或语句块;

其中，“<循环次数表达式>”用于指定循环次数，它可以是一个整数、变量或一个数值表达式。如果是变量或数值表达式，其取值只在第一次进入循环时得到计算，从而得以事先确定循环次数。“语句或语句块”是要被重复执行的循环部分。下面范例产生四变量组合电路的激励波形，波形如图7.所示。

例3-11 用 repeat语句产生四变量组合电路的激励波形

```
reg a,b,c,d;
initial
   begin
      {a,b,c,d}= 0 ;
       repeat (16)    begin    #100    {a,b,c,d} = {a,b,c,d}+1 ;   end
       #100 $stop;
   end
```

3.2.5 任务（task）与函数（function）

task 和 function 是存在于模块的一种类似 “子程序”结构，目的是为了对需要多次执行的语句进行描述，便于理解和调试。

1.任务

1) 任务的定义在模块说明部分中编写，其定义的形式如下：

```
task 任务名称;
    端口及数据类型说明
    局部变量说明
    行为语句或语句块;
 endtask
```

任务可以没有或有一个或多个参数。值通过参数传入和传出任务。除输入参数外（参数从任务中接收值），任务还能带有输出参数（从任务中返回值）。例如：

```
task counter;
    output [3:0] count;
    input     reset;
    begin
        if (reset) count = 0;       // 异步清 0
        else     count = count + 1;
    end
endtask
```

2)任务调用格式

<任务名>（ 端口 1，端口 2，…… ）

语句中参数列表必须与任务定义中的输入、输出和输入输出参数说明的顺序匹配。因为任务调用语句是过程性语句，所以任务调用中的输出参数必须是寄存器类型的。例调用上面定义的任务 count，可使用下面的语句：

```
input     clrn,clk;
output[3:1]     q_out;
reg [3:1]        q_out;
…
always @(posedge clk)
counter(q_out, clrn);
```

使用任务时，还需注意以下几点： 任务的定义和调用必须在同一模块 module 中，且任务的的调用只能在过程中，不能出现在 assign 语句中。另外，在一个任务中可以调用其它的任务和函数，也可以调用该任务本身。

2.函数

1）函数定义的形式如下：

function <返回值位宽或类型>　　<函数名>
　　输入端口与类型说明；
　　局部变量说明；
　　语句或语句块；
endfunction

另外，函数定义在函数内部隐式地声明一个寄存器变量，该变量与函数同名并且取值范围相同，故也称 “函数名变量”。函数调用时通过这个函数名变量”来传递函数值。

“返回值位宽或类型”是可选的，其作用是指定函数取值范围和类型，缺省的函数值为一位寄存器型。

函数至少要有一个输入。要注意在函数定义中，不允许出现 output 和 inout 端口。

2）函数调用的形式为：

<函数名>（ <输入表达式 1>， <输入表达式 2>，……）

注意，输入表达式的排列顺序及类型与定义结构中的输入端口顺序及类型一致；与任务调用不同，函数调用既要出现在过程语句中，也可出现在 assign 语句；函数可调用其它的函数，但不能调用任务。

下面先给出例子来说明函数的使用方式。在例 3-12 中，函数 Reverse 的作用是将输入信号（一个字节长度）的高、低位对调。

例 3-12　函数的定义和调用

```
module Reverse(date_in,date_out);
input [7:0]       date_in;
output [7:0]      date_out;
//函数定义
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
//函数调用
assign     date_out=ReverseBits(date_in);
endmodule
```

3.3 结构描述风格及 verilog 层次化设计

Verilog HDL 中的结构建模共有三种描述方式：门级建模、开级关建模和模块级建模。门级建模已在前面介绍。由于篇幅有限，下面只介绍种模块级建模。

复杂数字系统一般采用“自顶而下”的设计方法，即将复杂数字系统划分为几个子系统，再将子系统划分为若干子系统和功能模块，模块还可划分为若干个子模块，直至分成最基本

的模块组成。这就要求一个模块能够调用另外一个子模块中，这样就建立了 Verilog HDL 描述的层次关系，被调用的模块称为这个模块底层模块。

调用模块方法有“位置对应调用法”和“端口名称对应调用法”两种方法，由于“端口名称对应调用法”不易出错，且代码可读性强。因此，我们要求采用“端口名称对应调用法”，其调用格式如下：

模块名 ＜#(.参数 1（值 1），.参数 1（值 2）…)＞ 实例名（

.端口 1（与端口 1 相连的信号），

.端口 2（与端口 2 相连的信号），

…… ）；

上面的调用格式中，参数传递非必须的，当模块没有定义参数时，显然不存在参数传递问题；而当模块有定义参数，也可省略，此时参数采用默认值（即模块中的参数值）。

下面以 4 位无符号数乘法器为例说明模块的调用方法，乘法器的“移位加”算法是模拟笔算的一种比较简单的算法。*A* 与 *B* 相乘过程如图3.11 所示。*A* 与 *B* 都是 4 位二进制无符号数，每一行称为部分积，它表示左移的被乘数根据对应的乘数数位乘以 0 或 1，所以二进制数乘法的实质就是部分积的移位和相加。

				A_3	A_2	A_1	A_0	= A
			×)	B_3	B_2	B_1	B_0	= B
				A_3B_0	A_2B_0	A_1B_0	A_0B_0	部分积 pp0
			A_3B_1	A_2B_1	A_1B_1	A_0B_1		部分积 pp1
		A_3B_2	A_2B_2	A_1B_2	A_0B_2			部分积 pp2
+)	A_3B_3	A_2B_3	A_1B_3	A_0B_3				部分积 pp3
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	= P

图 3.11　乘法运算过程

图3. 12 为四位乘法器的电路结构图，主要用两个四位加法器（a 0 、a 1 ）和一个六位加法器（a 0 、a 1 ）组成。加法器 a 0 计算部分积 pp0 和 pp1 之和 pcs0；加法器 a1 计算部分积 pp2 和 pp3 之和 pcs1；而加法器 a2 则实现 pcs0 和 pcs1 之和，即最后的乘积。

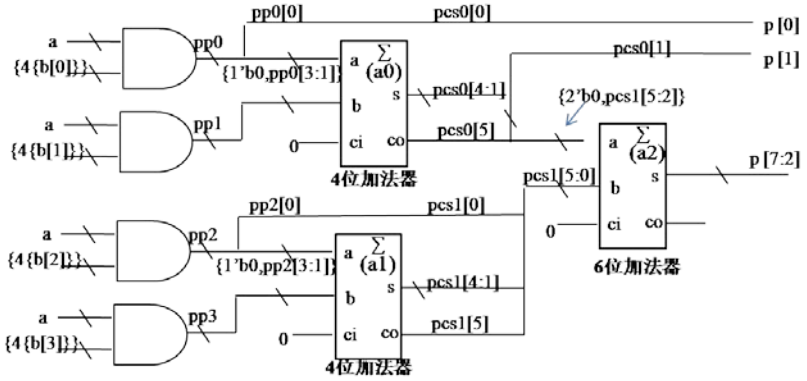


图 3.12　四位乘法器的电路结构图

我们调用例 3-10 的N位加法器模块来实现，乘法器顶层 Verilog HDL 描述如下。

例 3-13 四位乘法器的顶层 Verilog HDL 描述

```
module multipier(a,b,p) ;
    input [3:0] a,b ;
    output [7:0] p ;
    // 产生部分积
    wire [3:0] pp0 = a & {8{b[0]}} ; // x1
    wire [3:0] pp1 = a & {8{b[1]}} ; // x2
    wire [3:0] pp2 = a & {8{b[2]}} ; // x4
    wire [3:0] pp3 = a & {8{b[3]}} ; // x8
    wire[5:0] pcs0, pcs1;
    //调用 full_adder 模块求 pp0、pp1 之和，
    assign pcs0[0]=pp0[0];
  full_adder #(.N(4))   a0(    //将数值 4 传递给参数N
      .a({1'b0,pp0[3:1]}),
      .b(pp1),
      .ci(1'b0),
      .s(pcs0[4:1]),
      .co(pcs0[5]));
       //调用 full_adder 模块求 pp2、pp3 之和
    assign pcs1[0]=pp2[0];
    full_adder #(.N(4))     a1(      //将数值 4 传递给参数N
      .a({1'b0,pp2[3:1]}), //向量可拆开使用
      .b(pp3),
      .ci(1'b0),//  允许空脚输入常量
      .s(pcs1[4:1]),
      .co(pcs1[5]));
         //调用 full_adder 模块求乘积 p
    assign p[1:0]=pcs0[1:0];
    full_adder #(.N(6))     a2(      //将数值 6 传递给参数N
      .a({2'b0,pcs0[5:2]}),
      .b(pcs1),
      .ci(1'b0),
      .s(p[7:2]),
      .co());//允许空脚
endmodule
```

同学需从本例中，认真体会参数的作用，掌握模块调用及参数传递的方法是数字系统设计的基本技能，请同学重视。最后强调一下，层次化描述非常符合“自顶而下”的数字系统设计方法，是现在数字系统设计推崇的方法，目前比较流行**“小模块多层次”**的系统结构。

第 4 章　有限状态机的描述

4.1 状态机的结构

数字系统由控制器和数据通道组成，描述控制器功能常用有限状态机(FSM)或算法流程图（ASM）两种方法，这两种方法等价，可以相互转换。所谓状态机就是通过不同的状态转移来完成一些特定的顺序逻辑，图 4.1 是数字系统设计的最常用的同步状态机电路结构， 状态机是数字电路设计中的常用模块，组成元素有：输入信号、状态信号（state）、驱动信号（next_state，因为采用 D 型触发器，所以驱动信号即为下一状态信号 next_state）及输出信号。

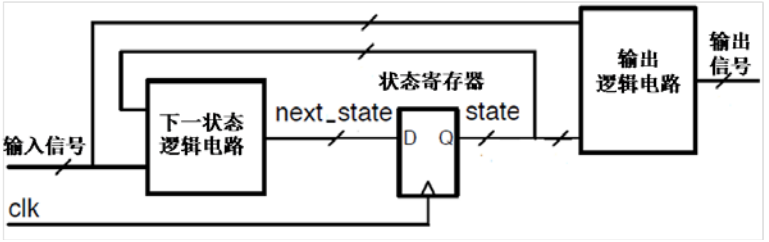


图 4.1　状态机的电路结构组成

状态机分 Mealy、Moore 两类，Mealy 状态机的输出信号不仅取决于当前状态，还与输入信号有关；而 Moore 状态机的输出信号只与当前状态有关，与输入信号无关；

4.2 状态机的 Verilog HDL 描述方法

状态机的描述方法一段式、 二段式和三段式一般有三种写法，他们在速度、面积、代码可维护性等各个方面互有优劣。

一段式：只有一个 always 过程块，把驱动、输出和状态转换等所有的逻辑都在一个 always 过程块中实现过程块；这种描述方法简洁，且由于输出采用寄存器方法，因此不会产生毛刺，所以电路稳定性较好。但这种描述方法可读性略差、难于理解和维护，如果状态复杂一些就很容易出错。严格来说，一段式只适用 Moore 状态机描述，不能准确地描述 Mealy 状态机。

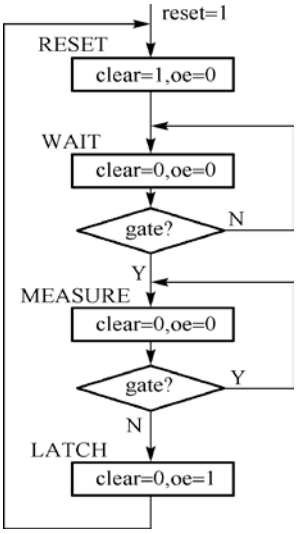
二段式： 有两个 always 过程块，把时序逻辑和组合逻辑分隔开来。一个 always 过程块描述时序逻辑，即状态寄存器；另一个 always 过程块描述组合逻辑，即下一个状态逻辑电路和输出逻辑电路；这种描述方法不仅便于阅读、理解、维护，而且利于综合器优化代码，利于用户添加合适的时序约束条件，利于布局布线器实现设计。但在两段式描述中，当前状态的输出用组合逻辑实现，可能存在竞争和冒险，产生毛刺。二段式可以准确地描述 Mealy 状态机和 Moore 状态机，是目前较为流行此种描述方法，推荐同学采用这种方法。

二段式的另一种方法是把状转换逻辑和输出逻辑分隔开来。一个 always 过程块描述状态转换，即状态寄存器和下一个状态逻辑用一个 always 过程块描述；另外，一个 always 过程块描述输出逻辑。

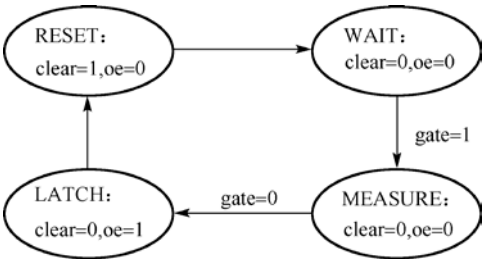
三段式： 有三个 always 过程块，一个 always 过程块采用同步时序的方式描述状态寄存器，第二个采用组合逻辑的方式描述下一个状态逻辑、描述状态转移规律，第三个使用同步时序的方式描述输出逻辑。代码容易维护，时序逻辑的输出解决了两段式组合逻辑的毛刺问

题，但是从资源消耗的角度上看，三段式的资源消耗多一些，且输出比另外两种会延时一个时钟周期。建议初学者不要采用三段式状态机描述方法。

下面先以频率测量系统的控制器为例，说明介绍状态机的描述方法，图 4.2 为控制器的 FSM 图和 ASM 图，控制器共有复位（RESET）、等待（WAIT）、测量（MEASURE）和结果输出锁相（LATCH）四个状态。控制器有一个输入信号 gata、两个输出信号 oe 和 clear。另外，一般来说，状态机必须有一个复位信号，复位信号不一定体现在状态机图中。



（a）ASM 图



（a）FSM 图

图 4.2 频率测量控制器的 FSM 图和 ASM 图

下面分别用一段式和二段式两种方式描述频率测量系统的控制器，并比较两者的差别。

例 4-1 一段式描述频率测量系统的控制器

```
module control(clk,reset,gate,oe,clear);
   input    clk,reset,gate;
   output reg    oe,clear;
   parameter    RESET=0,WAIT=1,MEASURE=2,LATCH=3; //状态编码
   reg [1:0] state; //四个状态，两位状态
   always @(posedge clk)
      if (reset) begin state=RESET;clear=1;oe=0; end
      else
          case(state)
             RESET:          begin    state=WAIT; clear=0; oe=0;end
             WAIT:           if(gate) begin    state=MEASURE; clear=0; oe=0;end
             MEASURE:        if(~gate) begin    state=LATCH;    clear=0; oe=1;end
             LATCH:          begin    state=RESET; clear=1; oe=0; end
        endcase
   endmodule
```

一段式状态机代码经 ISE 综合可得到的 RTL 级结果如图 4.3 所示，从中可看出，输出采用寄存器输出，电路结构与图 1 不太一样。

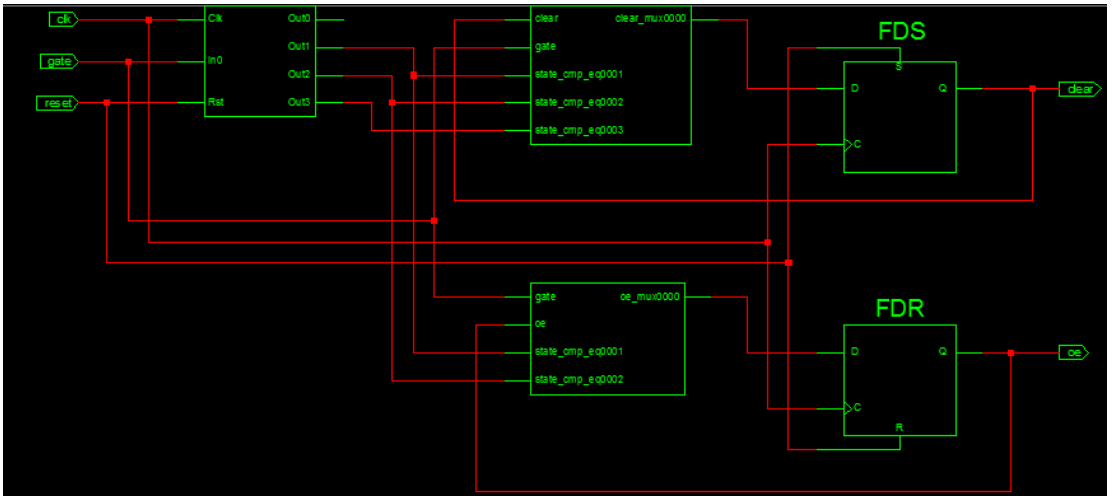


图 4.3 一段式状态机经 ISE 综合

例 4.2 二段式描述频率测量系统的控制器

```
module control(clk,reset,gate,oe,clear);
    input   clk,reset,gate;
    output  reg   oe,clear;
    parameter RESET=0,WAIT=1,MEASURE=2,LATCH=3; //状态编码
    reg [1:0] state,nextstate;
    //第一段-时序电路：D 型寄存器
    always @(posedge clk)
      if (reset) state=RESET; else state=nextstate;
    //第二段-组合电路：下一状态和输出电路
    always @(*)
     begin
       oe=0;clear=0;//默认值设为 0
       case(state)
          RESET:       begin nextstate=WAIT; clear=1; end
          WAIT:        if(gate) nextstate=MEASURE; else nextstate=WAIT;
          MEASURE:   if(gate) nextstate=MEASURE;   else nextstate=LATCH;
          LATCH:       begin nextstate=RESET; oe=1; end
       endcase
      end
endmodule
```

二段式状态机代码经 ISE 综合可得到的 RTL 级结果如图 4.4 所示，从中可看出，电路结构与图 4.1 一致，输出采用组合输出。

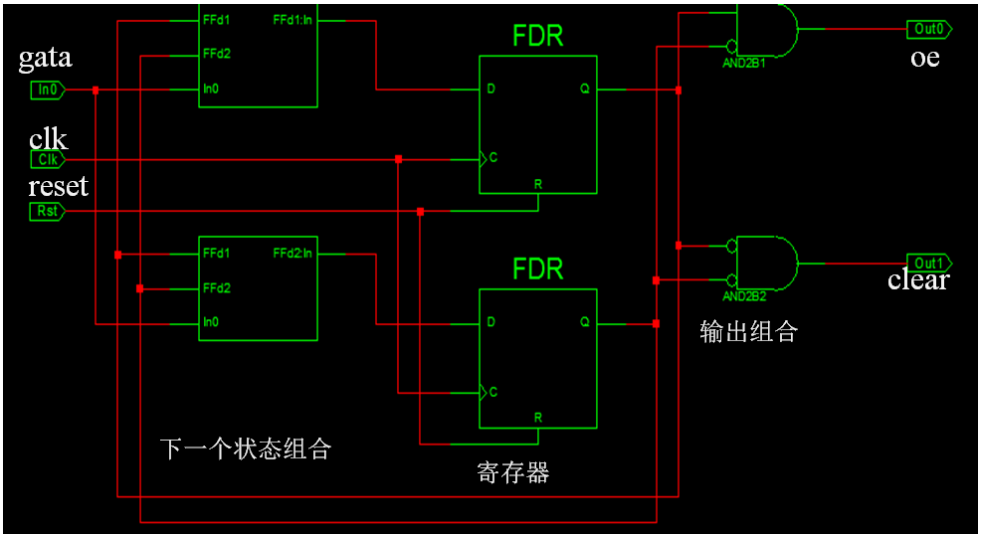


图 4.4 一段式状态机经 ISE 综合

例 4. 3 二段式的另外一种描述方式

```
module control2(clk,reset,gate,oe,clear);
    input   clk,reset,gate;
    output     oe,clear;
    parameter   RESET=0,WAIT=1,MEASURE=2,LATCH=3; //状态编码
    reg [1:0] state;
    //第一段：完成状态转换
    always @(posedge clk)
        if (reset)     state=RESET;
        else
          case(state)
            RESET:                  state=WAIT;
            WAIT:                 if(gate)     state=MEASURE;
            MEASURE:               if(~gate)  state=LATCH;
            LATCH:                state=RESET;
          endcase
       //第二段：完成状态转换，下面两条（两段）语句，用 always 语句只需一段
      assign clear=(state==RESET);
      assign oe=(state==LATCH);
endmodule
```

例 4.3 综合与例 4.2 综合结果完全一样。一段式还是要从本质上（组合或时序）看问题，不是形式上几段。例我们用一个时序 always 过程块描述寄存器，用二个组合 always 过程块描述下一个状态电路和输出电路，形式上是三段式，但本质上是二段式，因为三段式输出要求时序输出。

另外，也不需局限描述风格，如状态寄存器也可用 D 触发器实例完成。

第 5 章 设计举例与技巧

本节结合作者的工作实践，对常用的 Verilog HDL 的实例给出提示，在这些实例中，尽量使用多种方法，以便读者更好地掌握 Verilog HDL 语言。

5.1 常用组合电路的设计

数字电路中，描述组合电路常用有真值表（卡诺图）、表达式、电路图和功能表等方法，各种方法也有相应的较为合适的 Verilog HDL 描述方法：

1) 真值表： case 语句描述
2) 表达式： assign 赋值
3) 电路图：结构描述或 assign 赋值
4) 功能表： if-else 语句或 case 语句

下面提供一些常用的组合电路模块的 Verilog HDL 参考代码。

1.加法器和比较器

加法器和比较器是常用的组合电路，由于可以采用级联方法实现，因此，常采用结构描述实现。前面已给出几个加法器例子，下面两个就是给出的两个比较器例子。

例 5-1 n 位无符号数比较器的程序

```
module compare_n (great, equal, little,ina,inb);
parameter   n= 1 ;    //n 为比较器的位数
output              great,   equal,   little;
input[n-1:0]       ina, inb;
      assign   great=(ina>inb);
      assign   equal=(ina==inb);
      assign   little=(ina<inb);
endmodule
```

例 5-2：用加法器实现 n 位有符号数比较器的代码

```
module signed_compare_n(great, equal, little,ina,inb);
    parameter   n=4;    //n 为比较器的位数
    output              great,   equal,   little;
    input[n-1:0]       ina, inb;//有符号数，补码形式
  //计算 ina-inb，ina-inb= (ina)+(-inb),防止溢出，扩展一位。
  wire[n:0]   result;
  full_adder #(.N(n+1)) a1(    //将数值传递给参数 N
  .a({ina[n-1],ina}), //符号扩展
  .b ({~inb[n-1], ~inb}), //b 的反码
  .ci(1'b1),
  .s(result),
 .co());
     //两数相减结果：正、零和负分别大于、等于和小于
```

```
    assign   great= (~result[n] )&&( | result); //差为正
    assign   equal=~| result; //差为 0
    assign   little=result[n]; //差为负
endmodule
```

图 5.1 为 4 位有符号数比较器仿真波形，仿真结果正确，符合设计要求。

图 5.1　4 位有符号数比较器仿真结果

2.译码器

用 case 语句描述译码器最为合适。

1） 二进制译码器

例 5-3　3 线-8 线译码器（输出低电平有效）

```
module edcoder_38(out, in);
  output[7:0]   out;
  input[2:0]    in;
  reg[7:0]      out;
  always @(*)
    begin
      case(in)
        0: out = 8'b1111_1110;
        1: out = 8'b1111_1101;
        2: out = 8'b1111_1011;
        3: out = 8'b1111_0111;
        4: out = 8'b1110_1111;
        5: out = 8'b1101_1111;
        6: out = 8'b1011_1111;
        7: out = 8'b0111_1111;
      endcase
    end
endmodule
```

2） BCD 码-七段译码器

例 5-4 BCD 码-七段共阴数码译码器

```
module decode4_7(a,b,c,d,e,f,g,din);
  output   a,b,c,d,e,f,g;
  input[3:0]   din;
  reg   a,b,c,d,e,f,g;
```

```
always   @(*)
  begin
    case(din)
      0      : {a,b,c,d,e,f,g}=7'b1111110;
      1      : {a,b,c,d,e,f,g}=7'b0110000;
      2      : {a,b,c,d,e,f,g}=7'b1101101;
      3      : {a,b,c,d,e,f,g}=7'b1111001;
      4      : {a,b,c,d,e,f,g}=7'b0110011;
      5      : {a,b,c,d,e,f,g}=7'b1011011;
      6      : {a,b,c,d,e,f,g}=7'b1011111;
      7      : {a,b,c,d,e,f,g}=7'b1110000;
      8      : {a,b,c,d,e,f,g}=7'b1111111;
      9      : {a,b,c,d,e,f,g}=7'b1111011;
      default        : {a,b,c,d,e,f,g}=7'bx;
   endcase
  end
endmodule
```

3.数据选择器

数据选择器是数字电路最常用的电路之一，在 Verilog HDL 中，描述也最为简便，可用门级建模、if-else 语句、case 语句描述，也可条件运算符?:描述，前面已给出。

4.双向三态端口的描述

双向端口多用在总线结构中，也多用通信接口电路，例键盘、鼠标等接口也采用双向端口。图 5.2 所示的电路为双向 I/O 的端口，其 Verilog HDL 描述如下。

例 5-5 双向三态端口的 Verilog HDL 描述

```
module tri_inout(tri_inout,out,data,en,clk);
    input              en,clk;
    input              data;
    inout [7:0]        tri_inout; //双向端口
    wire   [7:0]       tri_inout;
    //双向端口的赋值
    assign tri_inout= en?data:8'bz;
endmodule
```

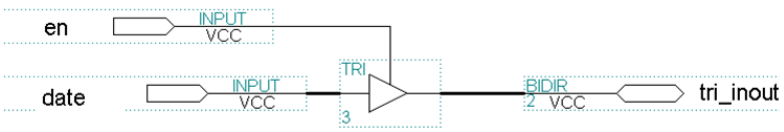


图 5.2　三态双向驱动 I/O 口

5.2 常用时序电路的设计

1.触发器和寄存器

在 Verilog HDL 中，最常用为 D 型触发器和寄存器，下面给出最常用的 D 型寄存器

例 5.6 最基本 D 型寄存器

```
module dff(clk,d,q);
    parameter WIDTH = 1;// 寄存器位数，当宽度 WIDTH 取 1 时即为 D 触发器
    input clk;
    input [WIDTH-1:0] d;
    output [WIDTH-1:0] q;
    reg [WIDTH-1:0] q;
    always @ (posedge clk)   q = d;
endmodule
```

例 5.7 带同步清 0 的 D 型寄存器，

```
module dffr (d, r, clk, q);
    parameter WIDTH = 1;
    input   r, clk;
    input [WIDTH-1:0] d;
    output [WIDTH-1:0] q;
    reg [WIDTH-1:0] q;
    always @ (posedge clk)
      if ( r )   q = {WIDTH{1'b0}};
      else       q = d;
endmodule
```

例 5.8 带同步清 0、输入使能的 D 型寄存器，

```
module dffre(d,en,r,clk,q);
    parameter WIDTH = 1;
    input   en，  r,   clk;
    input [WIDTH-1:0] d;
    output [WIDTH-1:0] q;
    reg [WIDTH-1:0] q;
    always @ (posedge clk)
      if ( r )   q = {WIDTH{1'b0}};
      else   if (en)   q = d;
             else   q = q;//该条语句也可省略
endmodule
```

2.计数器

计数器为最常用的时序电路，种类很多，这里只介绍最常用的几种。

例 5.9 最简单的 n 位二进制计数器（带同步清 0）

```
module counter_n (q, cout, cin , r, clk);
    parameter   n=1;
    output reg[n-1:0] q;
    output     cout;
    input     cin, r, clk;
    assign cout=&q && cin; //  进位输出
    always @(posedge clk)
          if(r)     q=0; //同步步清 0
          else   if(cin)   q =q + 1;   //计数或保持
               else   q = q;//该条语句也可省略
endmodule
```

例 5.10 任意进制计数器/分频器

```
module   counter_n(clk,r,ci,q, co);
     parameter   n=2; //计数器的模
     parameter   counter_bits=1;   //计数器的位数
     input     clk,r,ci;
     output   co;
     output [counter_bits:1]   q;
     reg [counter_bits:1]   q=0;
     assign   co=(q==(n-1))&& ci; //计数器为 Mealy 时序电路，进位必须为组合输出
     always @(posedge clk)
       begin
        if(r)   q=0 ;         //同步步清 0
        else     if(ci)        //cin=1,计数；cin=0,保持
                    begin     if (q==(n-1))   q=0 ; else q=q+1; end
               else   q = q;//该条语句也可省略
       end
endmodule
```

为了更好理解任意进制计数器/分频器，我们对其进行仿真测试，测试代码如下。

例 5.11 计数器测试代码

```
`timescale 1ns / 1ps
module counter_n_tb_v;
  // Inputs
     reg clk,r,ci;
     // Outputs
     wire [2:0] q;
     wire co;
```

```
// 5 进制计数器实例
counter_n #(.counter_bits(3),.n(5))   counter_5 (
      .clk(clk),.r(r),.ci(ci),.q(q),.co(co));
initial begin
            clk = 0; r = 1;     ci = 0;
      #152   r=0; ci=1;
      #1100   r=1;
      #100   r=0;ci=0;
      repeat(6)
       begin
        #100 ci=0;
        #200 ci=1;
       end
      $stop;
   end
   always #50 clk=~clk ;
endmodule
```

仿真结果如图 5.3 所示，当计数使能 ci=1 时，计数器对时钟信号计数或分频，而 ci 为脉冲信号时，计数器对脉冲信号计数或分频。注意，脉冲信号宽度必须为一个时钟周期。

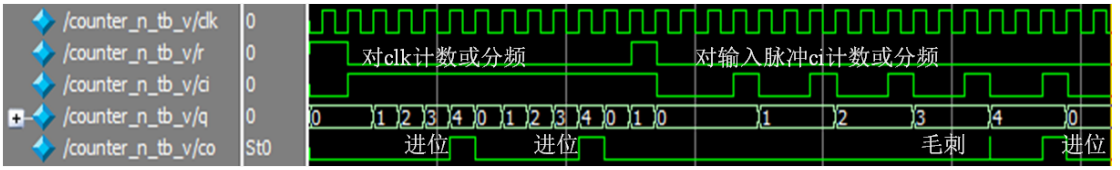


图 5.3　5 进制计数器仿真结果

BCD 码计数器在日常生活中极为常用，如日期、时间等，下面以 24 进制为例说明多位 BCD 码计数器实现方法，24 进制 BCD 计数器是由两个十进制计数器级联而成，代码中底层模块 counter_n 即为例 5-10 定义的模块 counter_n。

例 5-12　2~100 进制 8421BCD 计数器

```
module counter_bcd(q,co,ci,r,clk);
    parameter   MODULUS=8'h23;   //计数器最大值，计数器的模即 MODULUS+1
    output[7:0] q;
    output   co;
    input   ci,r,clk;
    //进位，该信号也为同步清零信号
    assign   co=(q==MODULUS)&ci;
    //个位
    wire co1;
    counter_n   #(.counter_bits(4),.n(10)) counter1 (
          .clk(clk),.r(co),.ci(ci),.q(q[3:0]),.co(co1));
```

```
    //十位
    counter_n #(.counter_bits(4),.n(10)) counter2 (
            .clk(clk),.r(co),.ci(co1),.q(q[7:4]),.co());
endmodule
```

计数器的综合结果如图 5.4 所示，从图中可看出电路结构，与设计思想相符。仿真结果如图 5.5 所示，符合设计要求。

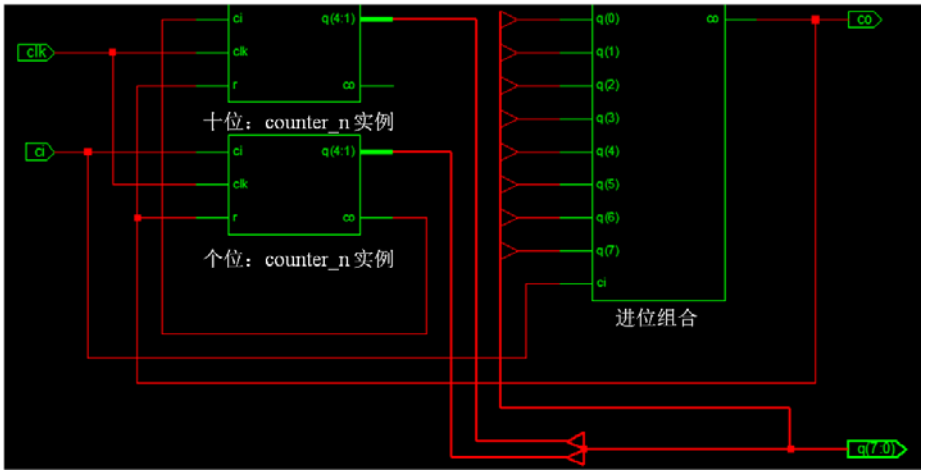


图 5.4 24 进制 BCD 计数器综合结果

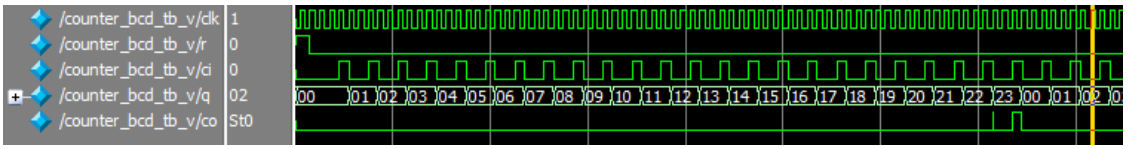


图 5.5　24 进制 BCD 计数器仿真结果

5.3 数字系统设计实例

设计并制作一个篮球比赛 24 秒计时系统。篮球比赛中，为了加快比赛节奏，规则要求进攻方在 24 秒内有一次投篮动作，否则视为违例。

1.设计要求

1）具有显示 24s 倒计时：用两个 LED 数码管显示，其计时间隔为 1s。
2）设置“复位”键：按此键可随时返回初始状态，即计时器返回到“24”并停止计数。
3）设置“启动/暂停”键，其作用为：

- 当处于初始状态或暂停状态，按此键，开始计时或继续计时；
- 当处计时状态，按此键暂停计数；

4）计时器递减计数到“0”时，给出一秒报警信号后使 计时器跳回“24”，并停止计数。

2.系统设计

根据设计要求，系统分解为输入按键处理模块、控制器模块、24 秒到计时模块（包括分频器）、显示模块、1 秒报警信号产生模块等模块组成，其框图如图 5.6 所示，系统时钟 clk 频率为 100MHz。

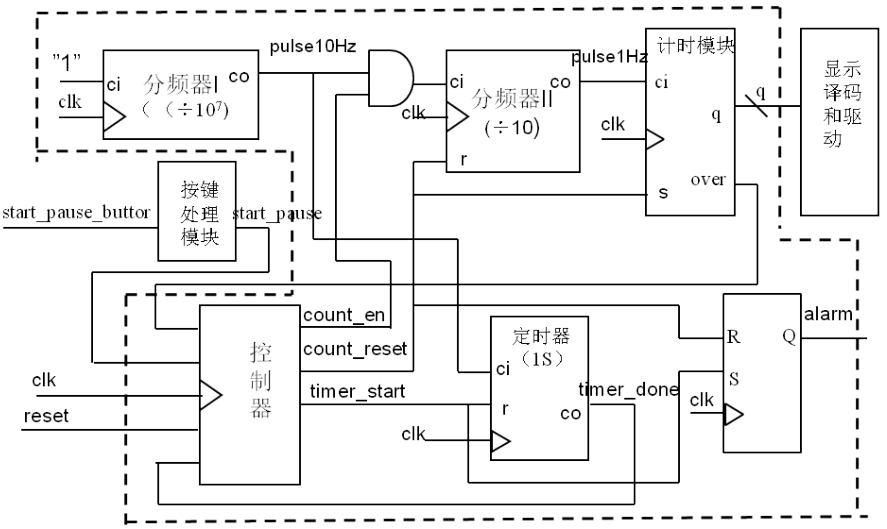
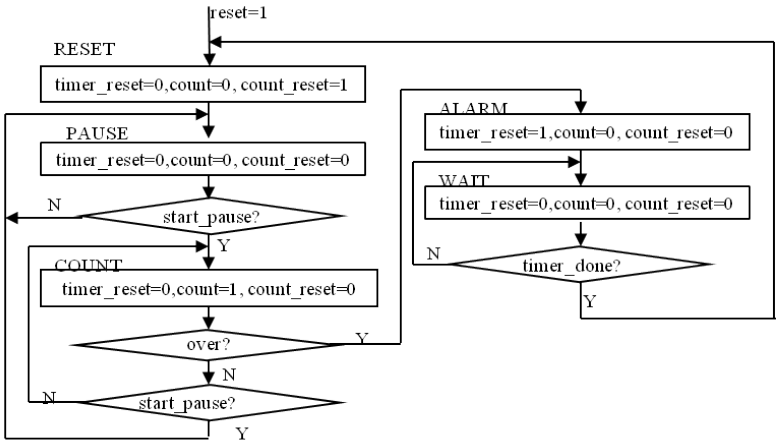
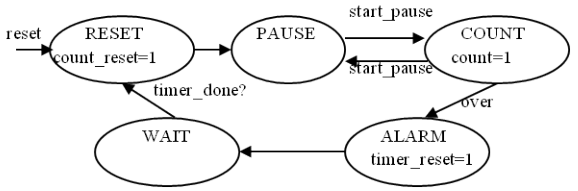


图 5.6　系统框图

控制器是电路核心，其 ASM 和 FSM 如图 5.7 所示。复位后 ,即计时器返回到初始值“24”并进入暂停 PAUSE 状态停止计时。在 PAUSE 状态时，如此时一次进攻开始，按“启动/暂停”键进入 COUNT 状态，开始倒计时。在 COUNT 状态下，分三种情况：一是 24 秒内投球、进攻方违例或进攻方丢球，按复位键进入初始状态；二是防守方违例，按“启动/暂停”停止计时（PAUSE），等进攻方重新发球，再按“启动/暂停”接着计时；三是 24 秒违例进入 ALARM 状态启动报警。随即进入 WAIT 状态等待报警结束。报警结束后，进入 RESET 状态复位，为下一次进攻准备。



（a）　ASM 图



（b）　FSM 图

图 5.7　系统的算法流程图和有限状态机图

2.Verilog HDL 描述

由于篇幅有限，我们对图 5-6 中的虚框内子系统设计描述。

例 5-13 系统顶层 Verilog HDL 代码

```
module timer_24s_top(clk, reset, start_pause, alarm, q);
    parameter sim=0;
    input clk;
    input reset;
    input start_pause;
    output alarm;
    output [7:0] q;
    //控制器模块实例
    wire over,timer_done;
    wire count_en,count_reset,timer_start;
    control control_inst(
      .clk(clk),
      .reset(reset),
      .start_pause(start_pause),
      .over(over),
      .timer_done(timer_done),
      .timer_reset(timer_start),
      .count(count_en),
      .count_reset(count_reset));
    /*分频器 I 模块实例,仿真时，2 分频；综合实现时，因 sim=0，分频比 10**7
    分频器调用例 5-10 定义的模块 counter_n*/
    wire pulse10HZ;
    counter_n   #(.n(sim?2:10**7),.counter_bits(sim?1:24)) divI(
                  .clk(clk),.r(1'b0),.ci(1'b1),.q(), .co(pulse10HZ));
    //分频器 II 模块实例，调用例 5-9 定义的模块 counter_n
    wire pulse1HZ;
    counter_n   #(.n(10),.counter_bits(4)) divII(
           .clk(clk),.r(count_reset),.ci(count_en && pulse10HZ),.q(), .co(pulse1HZ));
    //计时模块设计
    assign over=(q==1) &&   pulse1HZ;
    wire bo;
    //个位，调用十进制减法计数器模块 counter_down
    counter_down   #(.start_n(4))   counter_down_inst1(
        .clk(clk),.s(count_reset),.ci( pulse1HZ ),.q(q[3:0]),.co(bo));
    //十位，调用十进制减法计数器模块 counter_down
    counter_down #(.start_n(2)) counter_down_inst2(
     .clk(clk),.s(count_reset),.ci(bo),.q(q[7:4]),.co());
    //1s 定时器实例，调用定时计数器模块 timer_counter
    timer_counter    #(.n(9),.counter_bits(4))   timer_inst(
     .clk(clk),.r(timer_start),.ci(pulse10HZ),.co(timer_done));
```

```
    //RS 触发器实例
      srff    srff_alarm(.clk(clk),.r(count_reset),.s(timer_start),.q(alarm));
 endmodule
```

例 5-14 控制器 Verilog HDL 代码

```
 module control(clk,reset,start_pause,over,timer_done,timer_reset,count, count_reset);
    input    clk,reset,start_pause,over,timer_done;
    output    reg    timer_reset,count, count_reset;
    parameter RESET=0,PAUSE=1,COUNT=2,ALARM=3,WAIT=4; //状态编码
    reg [2:0] state,nextstate;
    //D 寄存器
    always @(posedge clk)        if (reset) state=RESET; else state=nextstate;
    //下一状态和输出电路
    always @(*) begin
        timer_reset=0;count=0;count_reset=0;//默认值设为 0
        case(state)
          RESET:    begin nextstate=PAUSE; count_reset=1; end
          PAUSE:    if(start_pause) nextstate=COUNT;    else nextstate= PAUSE;
          COUNT:    begin       count=1;
                    if(over) nextstate=ALARM;
                    else    begin
                              if(start_pause) nextstate= PAUSE;    else nextstate=COUNT;end
                            end
          ALARM:    begin nextstate=WAIT; timer_reset=1; end
          WAIT: if(timer_done) nextstate=RESET; else nextstate=WAIT;
          default: nextstate=RESET;
        endcase
      end
endmodule
```

例 5-15 十进制减法计数器 Verilog HDL 代码

```
module counter_down(clk,s,ci,q,co);
      parameter    start_n=10;//  减法计数起始值
      input      clk,s,ci;
      output    co;
      output [3:0]    q;
      reg [3:0]    q=0;
      assign    co=(~|q) && ci;//  借位
      always @(posedge clk)
        begin
          if(s)    q=start_n ;    //置减法计数起始值
          else    if(ci)         //cin=1,减法计数；cin=0,保持
                    begin      if (q==0)    q=9 ;    else q=q-1;    end
        end
endmodule
```

例 5-16 定时计数器的 Verilog HDL 代码

```
module timer_counter(clk,r,ci,co);
    parameter   n=10;
    parameter   counter_bits=4;
    input     clk,r,ci;
    output    co;
    reg [counter_bits:1]   q=0;
    assign    co=(q==(n-1))&& ci;       //进位
    always @(posedge clk)
      begin
       if(r)   q=0 ;
       else if(ci) q=q+1;       //cin=1,计数；cin=0,保持
      end
endmodule
```

例 5-17 RS 触发器 Verilog HDL 代码

```
module srff(clk, r, s, q);
    input clk,r,s;
    output reg q;
   always @(posedge clk)    if(r)q=0;else if(s) q=1;
endmodule
```

例 5-18 顶层测试代码

```
`timescale 1ns / 1ps
module timer_24s_tb_v;
 // Inputs
 reg clk,reset,start_pause;
 // Outputs
 wire alarm;
 wire [7:0] q;
 // 注意，给顶层传递 sim=1,降低分频比
 timer_24s_top   #(.sim(1))   timer_24s_inst (
      .clk(clk),
      .reset(reset),
      .start_pause(start_pause),
      .alarm(alarm),
      .q(q));
 initial   begin
      // Initialize Inputs
      clk = 0;
      reset = 1;
      start_pause = 0;
      #16 reset = 0; //reset
       #20 start_pause = 1;//start
      #10 start_pause = 0;
```

```
        #1000 start_pause = 1;//pause
        #10 start_pause = 0;
        #400 start_pause = 1;//start
        #10 start_pause = 0;
        #4100    start_pause = 1;//start
        #10       start_pause = 0;
        #2000      reset = 1;
        #10      reset = 0;//reset
        #200       $stop;
    end
      always #5 clk=~clk;
endmodule
```

注意：由于顶层存在分频比为 10^7 的分频器，基本不可能进行实时仿真，也没必要实时仿真。所以在顶层模块中，设置参数 sim，默认 sim=0 不影响电路综合和实现。而仿真时，测试文件通过参数传递置 sim=1，此时，将分频器 I 的分频比设为 2，这样大大减少仿真时间。

最后仿真结果如图 5.8 所示，符合设计要求。

图 5.8　24 秒计时系统的仿真结果

请同学们不要上传到网络