第3章 Verilog HDL层次化描述

3.1 设计方法学

数字系统设计中有两种基本的设计方法：自底向上和自顶向下，如图 3-1 所示。自底向上的开发模式是先编写出基础单元，然后再逐步扩大规模、不断补充和升级某些功能，最终构造出顶层模块的过程。这种模式的核心本质是“不断归纳”，直到形成稳定的系统。自顶向下的开发模式是不断地将相对复杂的大问题分解为相对简单的小问题，找出每个问题的构建方式，最终完成电路功能的过程。这一模式的核心本质是“不断分解”，直到每个问题都变成比较简单的小单元模块。

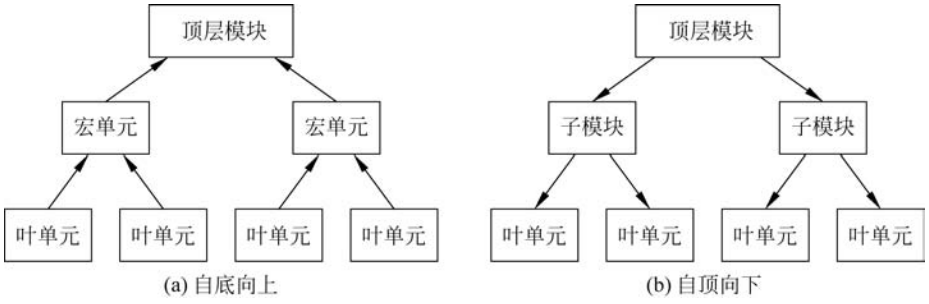


图 3-1　自底向上和自顶向下的设计方法

在典型的数字系统设计中，这两种方法往往是混合使用的，设计人员首先根据电路的体系结构定义顶层模块。逻辑设计者确定如何根据功能将整个设计划分子模块；与此同时，电路设计者对底层功能块电路进行优化设计，并进一步使用这些底层模块来搭建其高层模块。

因此，契合了数字系统设计的方法学理念，Verilog HDL 采用模块(module)的概念来代表一个基本的功能块，一个模块可以是一个元件，也可以是低层次模块的组合。模块通过接口(输入和输出)被高层的模块调用，但隐藏内部的实现细节。由此，设计者可以方便地对某一个模块进行修改，而不影响设计的其他部分。

根据设计需要，设计者可以从 5 个不同抽象层次对模块进行 Verilog 语言描述，而模块实现的功能相同。这 5 个抽象层次依据抽象层级由高到低定义如下。

1. 系统级

系统级(System Level)是用高级语言结构实现系统运行的模型,是针对整个系统性能的描述,是系统的最高层次的抽象描述。

2. 算法级

算法级(Algorithm Level)也称为行为级(Behavioral Level)或功能级(Functional Level),是高级语言结构实现设计算法的模型,对每一个功能模块完成行为描述。

3. 寄存器传输级

寄存器传输级(Register Transport Level,RTL)是描述数据如何在寄存器之间流动和如何处理、控制这些数据流动的模型,它将算法或功能用数字电路来实现。

4. 逻辑门级

逻辑门级(Logic Gate Level)是将逻辑门作为基本元件,描述逻辑门以及逻辑门之间的连接模型,与逻辑电路有一一对应的连接关系。

5. 晶体管开关级

晶体管开关级(Switch Level,即电路级)是把晶体管作为基本单元,从晶体管的层次描述电路,描述器件中晶体管以及它们之间连接的模型。

这5个抽象层次是如何在数字系统设计中体现的?首先来看Y图理论。Y图理论是Gajski和库恩于1983年提出的一种将数字集成系统的设计意见以及设计层次变得可视化的理论,它在硬件描述语言设计中广泛使用。如图3-2所示,Y图中三个轴分别表示行为域、结构域、物理/几何域,每一个同心圆则代表设计过程中各个层级的结构,由内往外,抽象层级越来越高。通过这个Y图,能够对数字电路的不同剖析角度建立联系。例如,假设

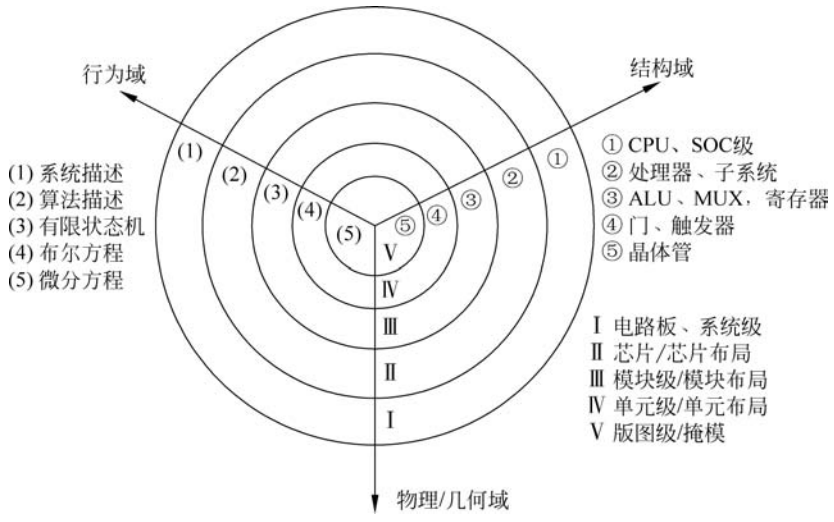


图3-2 Gajski的Y图

要完成一个 10 位的加法器,在系统级只要说明设计完成的功能和相关的约束,并不需要关心加法器的具体实现方式。如设计一个行波进位加法器或快速进位加法器,就没必要关心为了达到相应的速度,需要采用几级流水线,或者为了节约资源而采用什么实现方式。

将 Verilog 的五个抽象层级和 Y 图中三个领域,以及可以采用的描述方式的关系进行对应和归纳的结果如表 3-1 所示。

表 3-1 HDL 抽象层次描述表

抽象层次	行为领域	结构领域	物理领域	描述方式
系统级	性能描述	部件及它们之间的逻辑连接方式	芯片、模块、电路板和物理划分的子系统	行为建模
算法级	I/O 应答算法级	硬件模块数据结构	部件之间的物理连接、电路板、底盘	行为建模,数据流建模
寄存器传输级(RTL 级)	并行操作寄存器传输、状态表	算术运算部件、多路选择器、寄存器总线、微定时器、微存储器之间的物理连接方式	芯片、宏单元	行为建模,数据流建模
逻辑门级	用布尔方程描述	门电路	标准单元布图	结构化建模
晶体管开关级	微分方程式表达	晶体管、电阻、电容、电感元件	晶体管布图	结构化建模

为了实现在模块中从不同抽象层次进行描述,需要采用相对应的描述方式进行建模。Verilog HDL 的常用描述方式有三种,分别是:行为建模描述、数据流建模描述和结构化建模描述。行为建模描述通常采用 always 或 initial 为关键词的语句块,数据流建模描述通常采用 assign 为关键词的赋值语句,结构化建模则采用电路图实例化(也称为调用)已有的功能模块或原语。

对于大型系统,如 SoC 的整体设计,可以从系统级、算法级出发,采用行为建模的描述方式直接描述要实现的算法,这一过程类似于高级语言(如 C 语言)的实现流程。行为建模与具体的硬件实现没有任何关系,它们只是表述被描述的对象实现什么样的功能。针对中小规模的电路设计,可以在较低的抽象层次,通过描述电路的寄存器传输级(RTL)行为,说明各个系统的寄存器传输的时钟沿和先决条件。这一模式通常采用行为建模和数据流建模的描述方式。若从更低的抽象层次出发,可以通过结构化建模描述方式直接描述逻辑门或触发器的行为。设计人员可以采用自顶向下的设计方法,建立系统的行为模型并验证,然后再用结构模型的源代码替换行为模型的源代码,并对结构模型验证,用同一种语言完成各个设计阶段的任务。

下面将分别从数据流建模、行为建模和结构建模三个方面通过实例展示具体描述方式。

3.2 数据流建模描述方式

在数字电路中，信号经过逻辑电路的过程就像数据在电路中流动，即信号从输入流向输出。当输入变化时，总会在一定的时间之后在输出端呈现出效果。模拟数字电路的这一特性，对其进行建模的方式称为数据流建模描述方式。数据流描述说明数据在寄存器间移动，描述的是硬件的寄存器级实现，与硬件电路中的器件相对应。数据流建模最基本的方法就是用关键词 assign，通过连续赋值语句实现。

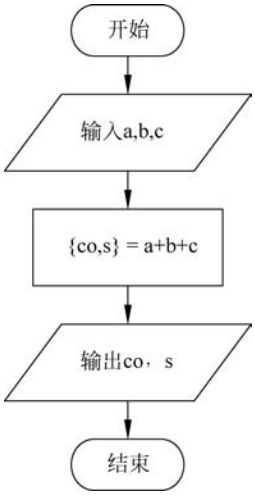


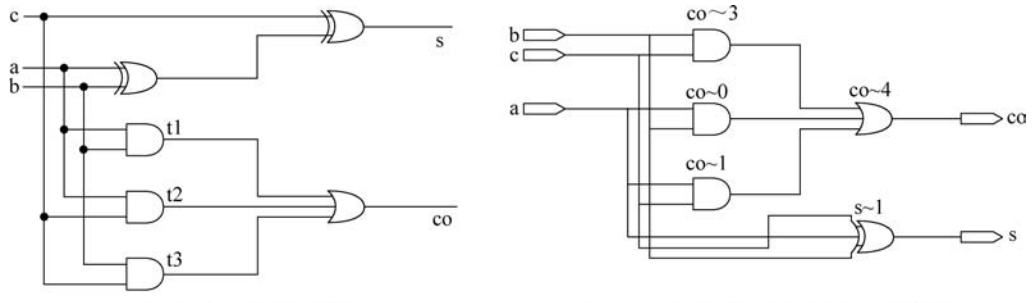
图 3-3 全加器的算法流程图

下面的例子将从算法级和寄存器传输级两种抽象层级出发，采用数据流建模描述方式，实现一个全加器设计。

首先，从算法级描述出发，图 3-3 给出了全加器的算法流程图。其中，a 和 b 是参与加运算的一位二进制数，c 是一位输入进位位。co 是输出进位位，s 为输出和。在图 3-3 后，给出了对应的 Verilog HDL 代码。从代码中可以看出，算法级描述更类似于高级语言描述方式，不考虑算法执行的结构细则，仅从抽象的数学概念上完成功能实现。

```
//从算法级抽象层级出发,通过数据流建模描述实现全加器
module FA_flow (a, b, c, s, co);
input a,b;
input c;
output s;
output co;
assign {co, s} = a + b + c;
endmodule
```

此外，还可以换一种思路，从 RTL 抽象层级出发，同样采用数据流建模描述方式实现全加器。因此，首先画出全加器的逻辑电路图如图 3-4(a)所示，之后写出相应的 Verilog HDL 程序。代码完成后，通过 Qutartus Ⅱ可以自动生成 RTL 分析电路，如图 3-4(b)所示。可以发现，图 3-4(a)中的电路与图 3-4(b)中的电路图几乎完全一致。



(a) 全加器的逻辑电路图　　(b) Quartus Ⅱ生成的全加器的RTL分析电路

图 3-4 全加器的逻辑电路图和 Quartus Ⅱ生成的全加器的 RTL 分析电路

```
//从 RTL 抽象层级出发,通过数据流建模描述实现全加器
module FA_flow (a, b, c, s, co);
input a, b;
input c;
output s;
output co;
assign s = a ^ b ^ c;
assign co = (a & b) | (a & c) | (b & c);
endmodule
```

【思考】 如何采用数据流建模描述方式分别从算法级和 RTL 级出发实现一个四位加法器?

3.3 行为建模描述方式

一个模块的行为模型是这个模块怎样工作的抽象描述,描述模块的输出在工作时与它的输入的关系,但是又不需要费力地描述模块是怎样通过具体电路实现的。

行为模型通常用在设计过程的初期。在这一时期,设计人员更关心的是模拟这个系统的预期行为以便了解系统总的性能特性,而较少考虑其如何具体实现。随后,才会考虑采用具有最终实现的准确细节的结构化模型,重新模拟以便说明功能和时序的正确性。设计过程中在选定模块的具体实现结构之前,采用行为描述方法来描述和模拟一个模块通常是很有用的。采用这种方法,设计人员可以在工作之前集中精力开发出正确的设计(即能够与系统的其他部分一起正确地工作并具有预期的行为)。然后可以将这个行为模型作为综合几个结构化实现方案的起点。

在表示方面,行为建模方式类似数据流的建模方式,但一般把用 initial 块语句或 always 块语句描述的内容归为行为建模方式。initial 块语句和 always 块语句在第 2 章中已详细介绍,二者是行为级建模的两种基本语句,其他所有的行为语句只能出现在这两个关键词对应的语句块里面。这两种语句块分别代表一个独立的执行过程,二者不能嵌套使用,但可并行执行。

在这里,同样从算法级和 RTL 级的抽象层级出发,利用行为描述方式分别实现全加器的 Verilog HDL 程序如下。

```
//从算法级抽象层级出发,通过行为建模描述实现全加器
module FA_behavior (a, b, c, s, co);
input a,b;
input c;
output s;
output co;
reg s, co;                          //这里需要定义为 reg 型
always @ (a,b,c)
{co, s} = a+b+c;
endmodule
```

```
//从 RTL 抽象层级出发,通过行为建模描述实现全加器
module FA_behavior (a, b, c, s, co);
input a, b;
input c;
output s;
output co;
reg s, co, t1, t2, t3;              //这里需要定义为 reg 型
always @ ( * )                      //当敏感信号表涵盖所有信号时,可以用 * 表示
begin
s = (a ^ b) ^ c ;
t1 = a & c;
t2 = b & c ;
t3 = a & b;
co = (t1 | t2) | t3;
end
endmodule
```

从上述两个 Verilog HDL 代码实例可以看出,建模描述方式的更改仅与 Verilog HDL 的代码描述方式有关,而抽象层级则涉及算法实现的方法和思路。两者的出发点完全不同。Verilog 建模的最大优点是与工艺无关,便于设计人员修改,极大提高了设计效率。通常用 Verilog 语言在 RTL 级描述一个设计,借助于自动综合工具,设计人员可以将 RTL 级代码快速地变换成逻辑级描述。目前,除了某些特定的应用系统,如数字信号处理,高层次的综合工具对行为级描述电路的综合效果没有对 RTL 级描述的电路那么好。因此,大部分的电路都是在 RTL 级进行描述的。

【思考】 如何采用行为描述方式从算法级和 RTL 级分别实现一个四位加法器?

3.4 结构化建模描述方式

抽象层级中更低级的门级和开关级描述需要通过结构化建模方式实现。结构化的建模描述方式通过对电路结构的描述来实现,即可通过实例化 Verilog HDL 中内置的基本门级(Gate Level Primitives,门级原语)和开关级(Switch Level Primitives,开关级原语)元件、实例化已有模块(module)、实例化用户定义的原语(User-Defined Primitive,UDP)三种不同方式,再使用线网来连接各器件,描述出逻辑电路图中的元件及元件之间的连接关系。

其中,基本门级和开关级元件是 Verilog HDL 中自带的一套标准原语,可以直接实例化;模块是设计者已经设计好的一个基本的功能块,在构建系统时可通过实例化该功能块(即在多个地方进行代码重用)以减少设计的工作量;用户定义原语是设计者根据电路的功能自己编写的原语,在设计电路、系统时可以被实例化。为了与之前的内容相衔接,这里首先介绍实例化已有模块的结构建模描述方式。

3.4.1 实例化已有模块

在 Verilog HDL 的语法中,把在当前模块内调用其他模块来完成设计的过程统称为模块的实例化。模块实例化语法结构如下。

模块名称 实例名称(端口连接);

模块名称是已定义好,在当前模块中拟调用的模块名;实例名称是在本模块内为其命名的新名称,主要用于同一被调用模块被多次调用时的命名区分。端口连接是在当前模块中把实例化的模块所包含的端口与当前模块的端口进行映射关系的连接,有两种写法,一种是按名称映射,另一种是按端口顺序映射。

例如,考虑设计电路实现两个闪灯组合,第一个灯每隔 6s 亮灭一次,第二个灯每隔 3s 亮灭一次。设立一个前提条件是,假设输入的时钟频率是 1Hz 的。

在这里,首先考虑电路整体包含一个输入:时钟 clk;两个输出:led1 和 led2。其次,分析实现闪灯的功能,需要对应两种不同的控制功能。这里采用一个模块控制 led1,另一个模块控制 led2 的方式实现。如图 3-5 所示,电路的顶层模块(实体)名是 led,其下一层(内部)包含两个模块,分别是 led_1,led_2。

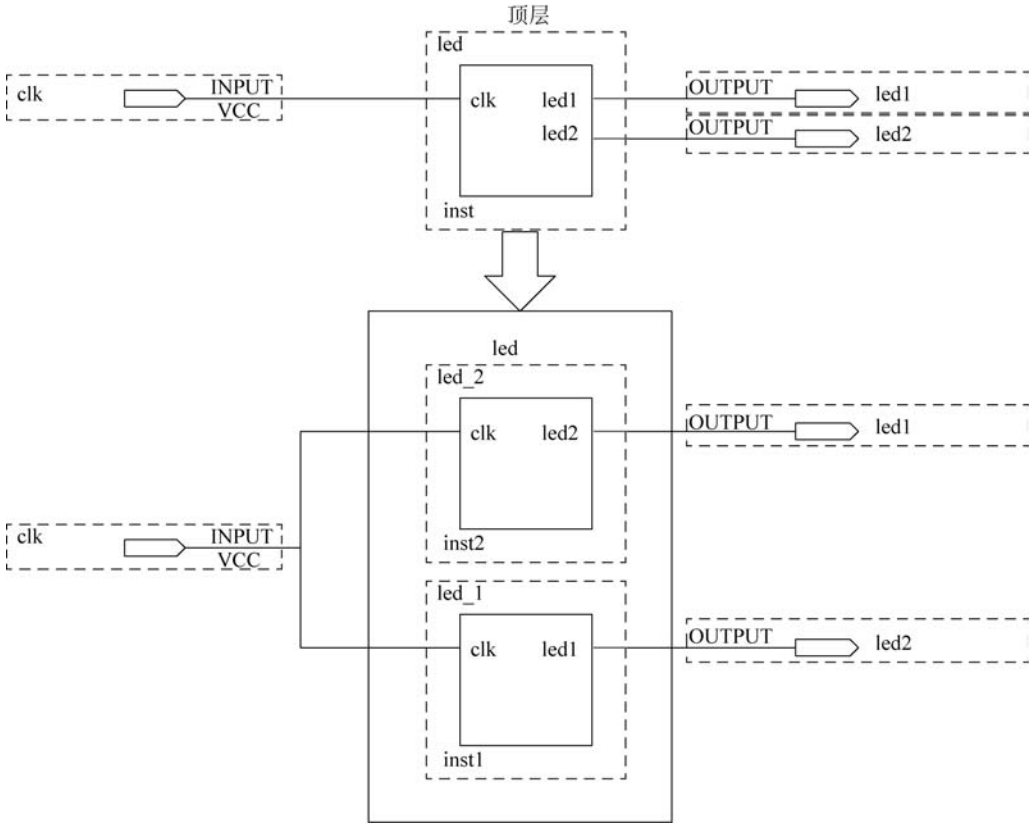


图 3-5 结构化建模示意图

首先,建立模块文件 led_1.v 和 led_2.v 分别实现 led_1 和 led_2。

```
module led_1(clk, led1);            //将 led_1 放在 led_1.v 中
input clk;
output led1;
reg clka;                           //中间变量
reg [1:0] count;                    //中间变量
reg led1;
```

```
always @(posedge clk)
begin
    count <= count + 1;
    if(count == 2)
    begin
        count <= 0;
        clka <= ~clka;
    end
end
always@(posedge clka)
    led1 = ~led1;
endmodule

module led_2(clk, led2);              //将 led_2 放入 led_2.v 中
input clk;
output led2;
reg clka;                             //中间变量
reg [1:0] count;                      //中间变量
reg led2;
always @(posedge clk)
begin
  count <= count + 1;
  if(count == 2)
  begin
    count <= 0;
    clka <= ~clka;
  end
  led2 <= clka;
end
endmodule
```

在顶层文件 led.v 的定义中,将输出端口 led1 和 led2 分别映射到 led_1 和 led_2 的输出端口,输入端口 clk 则同时映射到两个模块的输入端。

```
module led(clk, led1, led2);
input clk;
output led1,led2;
wire clk;                             //这里都可采用线网型
wire led1,led2;                       //这里都可采用线网型
led_1 M1 (.clk(clk),.led1(led1));  //端口映射采用名字关联方式
led_2 M2 (.clk(clk),.led2(led2));
endmodule
```

最终得到仿真结果如图 3-6 所示。可以看到,led1 的频率是 led2 的 1/2。led2 的频率是 clk 的 1/6。

该例题用结构化建模方式进行两输出的闪灯功能的设计,顶层模块 led 实例化了两个子模块 led_1 和 led_2。已有的设计模块 led_1 和 led_2 对于顶层实例模块 led 相当于一个已有器件模块,顶层模块只要对其进行引脚映射就可以了。其中,所有模块的书写顺序任

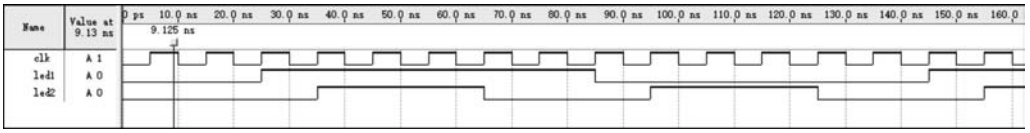


图 3-6 闪灯程序的 Quartus Ⅱ 仿真结果

意,可以看作所有模块并行执行。

在实例化中,端口映射(管脚的连线)采用名字关联,如 .clk(clk)中 .clk 表示被实例化器件 led_1 或 led_2 的管脚 clk,括号中的信号 clk 表示上层模块 led 接到 led_1 或 led_2 的 clk 管脚。

器件的端口映射采用名字关联方式实例化模块时端口的排列次序是任意的,如将上述实例化语句写成 led_1 (.led1(led1), .clk(clk)); 亦可。需要注意的是,采用这种书写方式实现管脚关联时,每个被映射管脚前要加上“.”。

另外,采用端口顺序的直接映射方式能够简化代码,这种方式要求必须按照被实例化模块 led_1 或 led_2 的端口排列顺序书写 led 的对应端口。如果排列顺序错误,则映射关系也将产生错误。这两种代码方式均可实现结构化建模描述功能。需要注意的是,实例化用户定义的模块不建议省略实例化名。

```
module led (clk, led1, led2);
input clk;
output led1, led2;
wire clk;
wire led1,led2;
led_1 M1 (clk, led1);                    //端口映射采用顺序对应的方式
led_2 M2 (clk, led2);
endmodule
```

在模块实例化过程中,如果有些端口没有被使用到,不需要进行连接,可以直接悬空。对于按照名称关联方式映射连接的情况,没有出现的端口名称就直接被认为是没有连接的端口;对于按顺序连接的方式,可以在不需要连接的端口位置直接留一个空格,以逗号表示这个端口在原模块中的存在。例如,如果不需要连接 clk,则可写成:

```
led_1M1 (led1(led1));                    //没有出现的 clk 就是没有连接的
led_1M2 (   , led1);                     //逗号前的空格表示没有连接的端口
```

此外,在实例化过程中,被实例化的各个模块之间连接输入和输出信号的数据连线,以及中间变量都是 wire 类型的。

```
wire led1, led2;
```

此部分的定义如果省略是符合语法规范的,因为 Verilog HDL 代码在编译过程中,凡是模块实例化中没有定义过的端口连接信号均被默认为 1 位的 wire 类型。但在正常设计中建议把中间变量信号显式地声明出来,避免出现位宽不匹配的现象。

3.4.2 实例化基本门级和开关级元件

抽象层级中的逻辑门级和逻辑开关级因为建模风格都是对电路结构的具体描述,所以

在 Verilog HDL 中都需要采用结构建模方法实现。其中，逻辑门级模型是对电路的“门”连接的具体描述，主要是描述与、或、非等基本电路的连接方式；开关级建模则更加接近“底层”，它把最基本的 MOS 晶体管连接起来实现电路功能。本节将首先介绍门级结构化描述，再介绍开关级结构化描述的使用方法。

1. 门级结构化描述

由于任何组合逻辑电路都可表示为与-或表达式，因此使用逻辑门的模型来描述组合逻辑电路是最为直观的一种方式。Verilog HDL 中内置了 12 种类型的基本门级元件模型，包括：多输入与门、多输入与非门、多输入或门、多输入或非门、多输入异或门、多输入异或非门、多输出缓冲器、多输出反相器、控制信号高/低电平有效的三态缓冲器、控制信号高/低电平有效的三态反相器等，详细符号及分类如表 3-2 所示。

表 3-2 Verilog HDL 中内置的 12 个基本门级元件

类型	逻辑门名称	功 能 说 明
多输入门	and	多输入端的与门
	nand	多输入端的与非门
	or	多输入端的或门
	nor	多输入端的或非门
	xor	多输入端的异或门
	xnor	多输入端的异或非门(同或门)
多输出门	buf	多输出端的缓冲器
	not	多输出端的反相器
三态门	bufif0	控制信号低电平有效的三态缓冲器
	bufif1	控制信号高电平有效的三态缓冲器
	notif0	控制信号低电平有效的三态反相器
	notif1	控制信号高电平有效的三态反相器

门级调用的语法格式与已有模块的调用方法类似，其一般形式为：

```
逻辑门类型 <实例名称(可省略)> (端口 1,端口 2,端口 3,…);
```

实例名称在门级建模中一般是不使用的，因为门级建模使用到的基本门较多，对其一一命名没有必要。可用的逻辑门类型见表 3-2 中所列共 12 种。具体写法范例如下。

```
nand  NA1 (out,in1,in2,in3);
xor    XOR1 (out,in1,in2,in3);
buf  B1 (out1,out2,out3,…, in);
not    N1 (out1,out2,out3,…, in);
bufif0  BU1 (out,in1,ctrl);
bufif1  BF1 (out,in1,ctrl);
notif0  NO1 (out,in1,ctrl);
notif1  NT1 (out,in1,ctrl);
```

需要注意的是，无论哪一种预定义逻辑门，其端口都是输出在前，输入在后。对于包含

控制信号的逻辑门,其控制端口写在最后。因此,进行端口映射时也必须按照这样的顺序书写。下面将对于每一种门的基本功能和逻辑符号进行列举。

1) 多输入门

Verilog HDL 中内置的 6 种类型多输入门的逻辑真值表如表 3-3 所示,A,B 表示任意输入信号。多输入门可能有多个输入,但只有一个输出。每个输入有四种可能:0,1,x,z。由于多输入门的输出不可能为高阻态 z,故只要任意一个输入为 x 或 z,则输出必定为 x。

表 3-3 多输入门的逻辑功能表

and		B			
		0	1	x	z
A	0	0	0	0	0
	1	0	1	x	x
	x	0	x	x	x
	z	0	x	x	x
nand		B			
		0	1	x	z
A	0	1	1	1	1
	1	1	0	x	x
	x	1	x	x	x
	z	1	x	x	x
or		B			
		0	1	x	z
A	0	0	1	x	x
	1	1	1	1	1
	x	x	1	x	x
	z	x	1	x	x
nor		B			
		0	1	x	z
A	0	1	0	x	x
	1	0	0	0	0
	x	x	0	x	x
	z	x	0	x	x
xor		B			
		0	1	x	z
A	0	0	1	x	x
	1	1	0	x	x
	x	x	x	x	x
	z	x	x	x	x

续表

xnor		B			
		0	1	x	z
A	0	1	0	x	x
	1	0	1	x	x
	x	x	x	x	x
	z	x	x	x	x

多输入门的元件模型见图 3-7。图中以 3 输入为示例，实际输入口个数可大于 3。

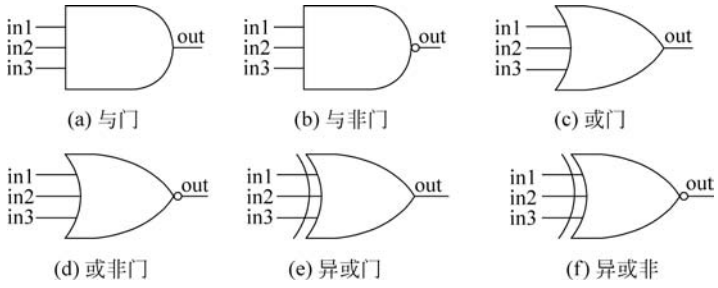


图 3-7 多输入门的元件模型

2）多输出门

Verilog HDL 中内置了两种类型的多输出门元件：缓冲器 buf 和反相器 not，它们可以有多个输出端 out1，out2，…，out$N$，多个输出端输出值相同，但只有一个输入端 in。图 3-8 为多输入缓冲器和反相器的元件模型的示意图。图中输出端的个数可根据实际应用的需要确定。缓冲器、反相器的逻辑真值表如表 3-4 所示。

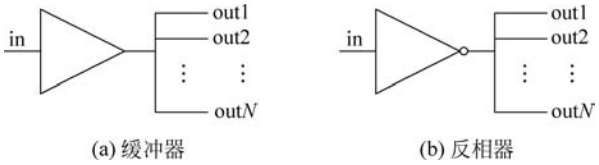


图 3-8 多输出缓冲器和反相器的元件模型

表 3-4 单输入多输出逻辑门功能表

	buf				not			
in	0	1	x	z	0	1	x	z
out	0	1	x	z	0	1	x	z

3）三态门

Verilog HDL 中内置了 4 种类型的三态门元件。控制信号低电平有效的三态缓冲器、控制信号高电平有效的三态缓冲器、控制信号低电平有效的三态反相器、控制信号高电平有效的三态反相器。它们有一个数据输入端 in、一个输出端 out 和一个控制输入端 ctrl。由于控制输入信号的控制作用，三态门的输出除了 0，1 外，还有可能是高阻态 z。图 3-9 为三态门元件模型的示意图。

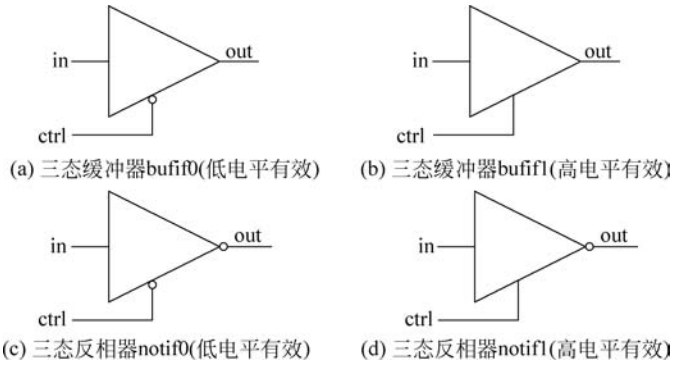


图 3-9　三态门元件模型

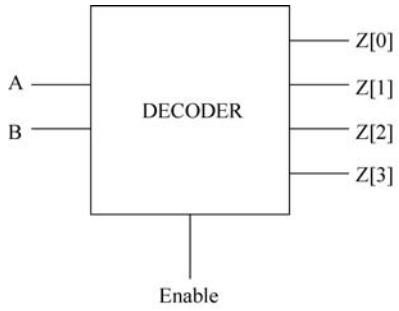
以上三态缓冲器、三态反相器的逻辑真值表如表 3-5 所示。表中 0/z 表明三态门的输出有可能是 0,有可能是 z,主要由输入的数据信号和控制信号的强度决定,同理可知 1/z 的含义。

表 3-5　三态门的逻辑真值表

bufif0		ctrl			
		0	1	x	z
in	0	0	z	0/z	0/z
	1	1	z	1/z	1/z
	x	x	z	x	x
	z	x	z	x	x
bufif1		ctrl			
		0	1	x	z
in	0	z	0	0/z	0/z
	1	z	1	1/z	1/z
	x	z	x	x	x
	z	z	x	x	x
notif0		ctrl			
		0	1	x	z
in	0	1	z	1/z	1/z
	1	0	z	0/z	0/z
	x	x	z	x	x
	z	x	z	x	x
notif1		ctrl			
		0	1	x	z
in	0	z	1	1/z	1/z
	1	z	0	0/z	0/z
	x	z	x	x	x
	z	z	x	x	x

下面采用结构化建模方式，使用基本的逻辑门对解码器进行设计。

解码器有两个输入端 A 和 B，一个使能端 Enable，四个输出端 Z[3]～Z[0]，其逻辑符号和真值表如图 3-10 所示。



En	A	B	C
0	0/1	0/1	1111
1	0	0	1110
1	0	1	1101
1	1	0	1011
1	1	1	0111

图 3-10 解码器逻辑符号和真值表

从真值表可以看出，解码器的使能端 Enable 为高电平有效，输出端 Z 为低电平有效。当 Enable＝0 时，无论输入为什么值，输出都为 1111。当 Enable＝1 时，根据 AB 的输入组合，对应 Z 的相应位输出为 0，其他位输出为 1。图 3-11 是采用基本门电路实现的解码器。

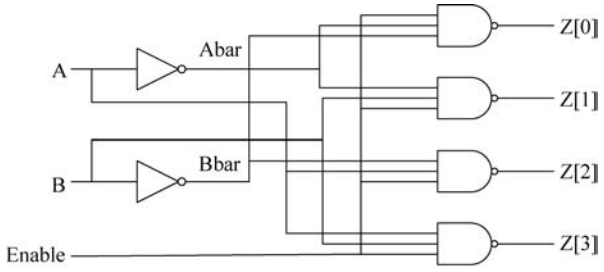


图 3-11 解码器的逻辑电路图

由图 3-11 可知，该解码器由两个反相器(not)、四个多输入端与非门(nand)组成，其 Verilog HDL 程序代码如下。

```
//采用结构化描述方式对解码器进行设计
module Dec (Z, A, B, Enable);
output[3:0] Z;                    //输入/输出端口声明
input A, B, Enable;
wire Abar, Bbar;                  //内部线网声明
//实例化逻辑门级原语
not N0 (Abar, A);                 //产生 Abar 和 Bbar 信号
not N1 (Bbar, B);
nand A0 (Z[3], Enable, A, B);     //实例化三输入与非门
nand A1 (Z[0], Enable, Abar, Bbar);
nand A2 (Z[1], Enable, Abar, B);
nand A3 (Z[2], Enable, A, Bbar);
endmodule
```

上述解码器的例子是从逻辑门级的抽象层级出发，通过结构化描述实现电路功能。下面进一步降低抽象层次，来看开关级的抽象描述。

2. 开关级结构化描述

数字电路系统中逻辑门是由一个个晶体管组成的。在 Verilog HDL 中,有用于直接描述 NMOS 和 PMOS 的原语,即 Verilog HDL 具有对 MOS 晶体管级进行设计的能力。Verilog HDL 目前仅提供逻辑值 0,1,x,z 和它们相关的驱动强度进行数字设计能力,没有模拟设计能力,因此在 Verilog HDL 中,晶体管也仅被当作导通或者截止的开关。

开关级建模处于最低的设计抽象层次,随着电路复杂性的增加以及先进 CAD 工具的出现,以开关级为基础进行的设计正在逐渐萎缩,只有在很少的情况下设计者需定制自己的最基本元件时才使用,因此本节只讨论开关级建模的基本原理。关于开关级建模的详细内容可参考 IEEE Standard Verilog HDL 文档。

Verilog HDL 中内置的开关级建模元件主要有 MOS 开关(包括 NMOS 开关和 PMOS 开关)、CMOS 开关、电源和地、双向开关、阻抗开关等,这里只介绍最常用的 MOS 开关、CMOS 开关、电源和地。

1) MOS 开关

MOS 开关元件的符号如图 3-12 所示。

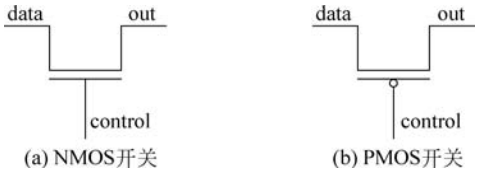


图 3-12 NMOS 开关和 PMOS 开关符号

MOS 开关元件可用关键字 nmos 和 pmos 声明,实例化形式为:

```
nmos N1 (out1,data,control);
pmos P1 (out1,data,control);
```

其中实例化名 N1,P1 可省略。

MOS 开关元件的逻辑真值表如表 3-6 所示,NMOS 开关在 control 信号为 1 时导通,在 control 信号为 0 时输出为高阻态。同理,PMOS 开关在 control 信号为 0 时导通,在 control 信号为 1 时输出为高组态。表中符号 L 代表 0 或 z,H 表示 1 或 z。

表 3-6 MOS 开关的逻辑真值表

(a) NMOS 开关

nmos		control			
		0	1	x	z
data	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	z	z	z

(b) PMOS 开关

pmos		control			
		0	1	x	z
data	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	z	z	z	z

2）CMOS开关

CMOS开关本质上是NMOS和PMOS两个开关组合而成，其符号如图3-13所示。给定data、ncontrol和pcontrol值，则可根据表3-6推断出CMOS开关的输出值。通常信号ncontrol和pcontrol是互补的。当ncontrol＝1且pcontrol＝0时，CMOS开关导通，输出值为data值；当ncontrol＝0且pcontrol＝1时，CMOS开关的输出为高阻值。

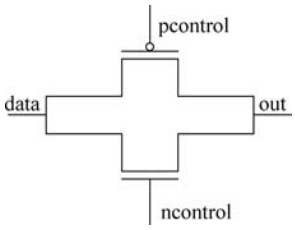


图3-13 CMOS开关符号

CMOS开关元件可用关键字cmos声明，实例化形式为：

```
cmos C1(out,data,ncontrol,pcontrol);
```

其中，实例化名C1可省略。

3）电源和地

设计晶体管级电路时需要源极（Vdd，逻辑值1）和地极（Vss，逻辑值0）。源极和地极分别用关键字supply1和supply0来声明，一般声明形式为：

```
supply1 vdd;                    //声明 vdd 为电源
supply0 gnd;                    //声明 gnd 为地
```

若某信号接电源或地，将该信号赋值为电源或地的值即可。

```
assign x = vdd;                 //x 接电源
assign y = gnd;                 //y 接地
```

例如，用晶体管开关级元件设计两输入与非门。两输入与非门的门级电路图和开关级电路图如图3-14所示。

```
module my_nor(out, a, b);
output out;
input a, b;
wire c;
//声明电源和地
supply1 pwr;                    //pwr 连接到电源 Vdd
supply0 gnd;                    //gnd 连接到地
pmos (out, pwr, a);             //实例化 PMOS 晶体管
pmos (out, pwr, b);
nmos (c,gnd , b);               //实例化 NMOS 晶体管
nmos (out, c, a);
endmodule
```

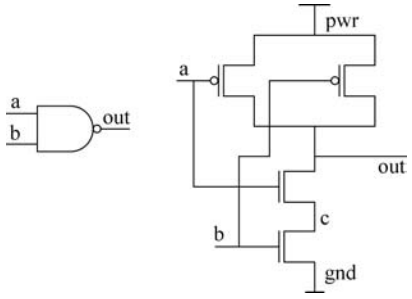


图3-14 两输入与非门的门级电路图和开关级电路图

3.4.3 用户定义的原语

Verilog 语言不仅为设计者提供了 26 条门级和开关级原语组成的集合实现数字系统的结构建模,还提供了一种扩展门级原语(又称为基元,UDP)的方法,使用户能够自己定义组合电路以及包含电平敏感和边沿敏感的时序电路。其应用有如下三点:首先,用户可以定义原语以简洁有效地实现各种逻辑块的描述;其次,可以减少由于可能存在不确定值 x 而导致的不现实性问题;最后,能够提高模拟效率。但是,原语无法用于逻辑综合,只能仿真实现。

举例:以类似于逻辑函数真值表的形式实现用户定义原语。

```
primitive carry (carryout, carryin, ain, bin);
input carryin, ain, bin;
output carryout;

table
    0  0  0  :  0;
    0  0  1  :  0;
    0  1  0  :  0;
    0  1  1  :  1;
    1  0  0  :  0;
    1  0  1  :  1;
    1  1  0  :  1;
    1  1  1  :  1;
endtable
endprimitive
```

原语是与模块同级的,因此,不能在模块中定义原语。在上面这个例子里,描述了一个产生一位全加器进位的原语。carryout 是输出,carryin,ain 和 bin 是输入。在列出的表中给出了各种输入组合所对应的输出值,采用冒号将输入和输出隔开。需要注意的是,表中的输入顺序必须与原语定义时,语句端口表中的顺序一致。例如,从表中最后一行可以看出,如果输入 carryin=1,ain=1,bin=1,则输出 carryout=1。

1. 用户定义原语的基本特征及规则

用户定义原语的五个部分组成是:原语名称和端口列表、端口说明语句、原语初始化、状态表、结束定义,具体格式如下。

```
primitive <udp_name> (<输出端口名>, <输入端口名>); //原语名和端口列表
output <输出端口名>;                  //端口说明语句
input <输入端口名>;
reg <输出端口名>;                     //可选,只有表示时序逻辑的原语才需要
initial <输出端口名> = <值>           //原语初始化(可选,只有表示时序逻辑的 UDP 才需要)
table                                 //状态表
    <状态表>
endtable
endprimitive                          //定义的结束
```

原语的定义以关键字 primitive 开始,以关键字 endprimitive 结束。与模块类似,在定义时序逻辑的原语时,输出端口必须被声明为 reg 类型,而且需要一条 initial 语句对时序逻辑的输出端口进行初始化;原语的状态表是实现其功能的最重要的部分。状态表以关键字

table 开始，以关键字 endtable 结束，定义了输入状态、当前状态和输出状态的对应关系，该表类似逻辑电路的真值表。

用户定义原语必须遵循以下几个规则。

(1) 原语有多个输入端口，但只允许有一个输出端口，而且输出端口必须出现在端口列表的第一个位置。

(2) 原语不支持 inout 双向端口。

(3) 表示时序逻辑的原语需要保持状态，因此其输出端口除了要用关键字 output 声明外，还必须声明为 reg 类型。

(4) 表示时序逻辑的原语中的状态可以用 initial 语句初始化，将一个 1 位的值赋给 reg 类型的输出，该语句是可选的。

(5) 状态表的语法格式为：＜input1＞ ＜input2＞ …＜input$N$＞ ：＜output＞；输入的顺序必须与端口列表中出现的顺序相同，输入与输出之间以冒号“：”分隔，每一行以分号“；”结束。状态表的内容与逻辑电路的真值表相似，可根据真值表填写。状态表中的值可以取 0，1，x。原语不能处理 z 值，传送给原语的 z 值被当作 x 值处理。

(6) 原语与其他模块同级，不能在其他模块内定义。

(7) 在原语中不能实例化其他模块或者其他原语；但可以在其他模块内部实例化原语，实例化方法与门级原语的实例化方法相同。

2. 组合逻辑电路的原语描述

UDP 的类型有两种：表示组合逻辑的 UDP，表示时序逻辑的 UDP。

例如，采用用户定义原语的方式设计 4 选 1 的多路选择器(MUX)。

4 选 1 的多路选择器为组合逻辑电路，其真值表如图 3-10 所示。根据真值表列出状态表，对 4 选 1 的多路选择器进行自定义原语设计，符号？表示可能是 0、1 或 x。代码如下。

```
primitive udp_mux4_to_1 (output out,input i0,i1, i2, i3, s1, s0 );
table
    // i0   i1   i2   i3   s1   s0 : out;
       0    ?    ?    ?    0    0 : 0 ;   // s1s0 = 00 时,输出为 i0,与其他输入无关
       1    ?    ?    ?    0    0 : 1 ;
       ?    0    ?    ?    0    1 : 0 ;   // s1s0 = 01 时,输出为 i1 ,与其他输入无关
       ?    1    ?    ?    0    1 : 1 ;
       ?    ?    0    ?    1    0 : 0 ;   // s1s0 = 10 时,输出为 i2,与其他输入无关
       ?    ?    1    ?    1    0 : 1 ;
       ?    ?    ?    0    1    1 : 0 ;   // s1s0 = 11 时,输出为 i3,与其他输入无关
       ?    ?    ?    1    1    1 : 1 ;
endtable
endprimitive
```

3. 时序逻辑电路的原语描述

例如，采用用户定义原语的方式设计 D 触发器(时钟下降沿触发)。

当时钟输入端 clock 由高电平向低电平跳变时，数据 d 进行锁存，即输出端下一状态值为 d；当时钟输入端 clock 由高电平变化到不定状态时，输出端 q 保持不变；当时钟输入端 clock 发生正跳变(由 0 跳变到任意状态？或者由不定状态 x 跳变到 1)，输出端 q 保持不

变；当时钟输入端 clock 保持某个值不变时，输出端 q 保持不变。

```
primitive udp_edge_d (output reg q,input d,clock);
initial q = 0;
table
    // d   clock    :   q   : q + ;     //时钟下降沿触发的 D 触发器状态表
       1   (10)     :   ?   : 1;        //在 clock 的下降沿将输入值 1 锁存到 q 中
       0   (10)     :   ?   : 0;        //在 clock 的下降沿将输入值 0 锁存到 q 中
       ?   (1x)     :   ?   : - ;       //clock 变化到不定状态时,q 保持不变
       ?   (0?)     :   ?   : - ;       //忽略 clock 的正跳变
       ?   (x1)     :   ?   : - ;       //忽略 clock 的正跳变
    (??)    ?       :   ?   : - ;       //当 clock 为某个值不变化时忽略 d 的变化
endtable
endprimitive
```

其中，符号(10)表示从逻辑 1 到逻辑 0 的负跳变沿；符号(1x)表示从逻辑 1 到不确定状态的跳变；符号(0?)表示从逻辑 0 到 0、1、x 的跳变，隐含正跳变沿；符号(??)表示信号值从 0、1 或者 x 到 0、1 或者 x 的任意跳变。符号(-)表示输出不改变。

4. 原语的实例化

用户定义原语的实例化与实例化 Verilog 内置基本门元件相同。

例如，采用实例化原语定义的 4 选 1 多路选择器(MUX)的方法实现 16 选 1 的多路选择器。

如图 3-15 所示，利用 5 个 4 选 1 多路选择器构造一个 16 选 1 的多路选择器。MUX0，MUX1，MUX2，MUX3，MUX4 输入分别为 i0，i1，i2，…，i15，MUX4 的控制信号为 s3，s2，其他 MUX 的控制信号都为 s1，s0。

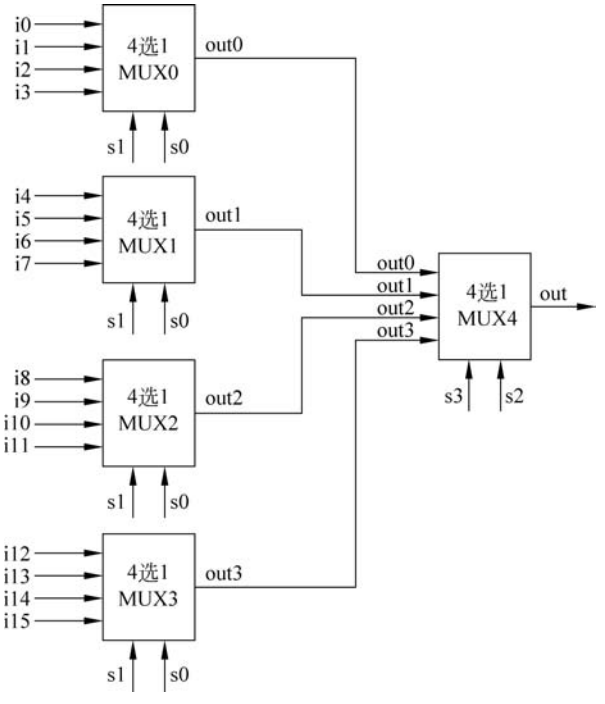


图 3-15　16 选 1 的多路选择器

```
// 4 选 1 多路选择器的 UDP
primitive udp_mux4_to_1 (output out,input i0,i1, i2, i3, s1, s0 );
…
endprimitive
module mux16_to_1 (out,
i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15,s3, s2, s1, s0);
output out;
    input i0, i1, i2, i3, i4, i5, i6, i7, i8, i9, i10, i11, i12, i13, i14, i15,s3, s2, s1, s0;
wire out0, out1, out2, out3;
udp_mux4_to_1 mux0 (out0, i0,i1, i2, i3, s1, s0);          //实例化 UDP
udp_mux4_to_1 mux1 (out1, i4, i5, i6, i7, s1, s0);
udp_mux4_to_1 mux2 (out2, i8, i9, i10, i11, s1, s0);
udp_mux4_to_1 mux3 (out3, i12, i13, i14, i15, s1, s0);
udp_mux4_to_1 mux4 (out , out0, out1, out2, out3, s3, s2);
endmodule
```

3.5 混合设计描述

复杂数字逻辑电路和系统设计过程,往往是多种设计模型的混合。例如,利用已有的全加器模块(FA_behavior)设计一个 4 位串行进位加法器(Four_bit_FA),在顶层模块(Four_bit_FA) 中采用结构描述方式对底层进行实例化,对底层模块(FA_behavior)可采用结构描述、数据流描述或行为级描述。在模块内部还可以将结构描述方式、数据流描述方式、行为描述方式自由混合。这也显示了 Verilog 语言功能的强大。它允许我们在详细的结构层次上建立系统的一部分模型,系统的其他部分则建立在较为抽象的层次上。

在接下来的例子中,将实现一个时钟驱动 4 位二进制显示的 Verilog 程序。这个功能相当于计数器连接寄存器数据显示。时钟输入实现计数器计数,每次计数输出值作为寄存器的地址,将地址中对应数据输出。如图 3-16 所示是寄存器中 4 位二进制地址的存储内容。这个程序中包含结构描述和行为描述的语句,并且采用行为描述的输出作为结构描述的输入,也就是行为描述驱动结构描述。

eSeg AB \ CD	00	01	11	10
00	1	0	1	1
01	0	0	1	0
11	0	0	1	1
10	1	1	1	1

图 3-16 寄存器数据存储内容

图 3-16 对应的输出函数为:

$$eSeg=\overline{B}\overline{D}+CD+A\overline{B}+AC$$

```
//采用行为描述驱动结构描述
module Behavior2Structure (eSeg, clock);
input clock;
output eSeg;
reg [3:0] count;                          //count[3]～count[0]分别对应 A,B,C,D
wire p1, p2, p3, p4;

initial                                   //行为描述
    count = 0;
always @ (posedge clock)
```

```
begin
    if (count == 15)
        count <= 0;
    else
        count <= count + 1;
end

and g1 (p1, ~count[2], ~count[0]);        //结构描述
and g2 (p2, count[3], ~count[2]);
and g3 (p3, count[1], count[0]);
and g4 (p4, count[3], count[1]);
or g5 (eSeg, p1, p2, p3, p4);
endmodule
```

下面一个例子中继续使用图 3-16 的寄存器内容,去除计数器的部分,将 A,B,C,D 直接作为输入,实现 eSeg 的数据输出。但是与之前例子不同的是,这里采用门级结构化描述的输出作为 always 块的输入。当 p1,p2,p3,p4 中任意一个发生变化时,行为语句都会计算它们的“或”的结果,并存储在 eSeg 中。

```
//采用结构描述驱动行为描述
module Structure2Behavior (eSeg, A, B, C, D);
input A, B, C, D;
output eSeg ;
reg eSeg ;
wire p1, p2, p3, p4;

and g1 (p1, ~B, ~D);                    //结构描述
and g2 (p2, A, ~B);
and g3 (p3, C, D);
and g4 (p4, A, C);

always @ (p1 or p2 or p3 or p4)         //行为描述
        eSeg = p1 + p2 + p3 + p4;
endmodule
```

3.6 用 Verilog HDL 建模实现自顶向下设计实例

在这一节中,将通过两个实例来介绍自顶向下层次化设计的流程。

实例一:通过数据流描述实现一个 2 位二进制的数据比较器,再通过结构化描述的方式,来实现一个 4 位二进制的数据比较器。

2 位比较器是指对两个 2 位二进制值 A 和 B 进行大小比较,其卡诺图如图 3-17 所示。为了用数据流描述实现这个功能,首先通过如下布尔方程组来写出输出表达式。其中,A1 和 A0 是 A 的高位和低位,同理,B1 和 B0 是 B 的二进制高位和低位。

$A\_It\_B=\overline{A1}B1+\overline{A1}\ \overline{A0}B0+\overline{A0}B1B0$

$A\_gt\_B=A1\overline{B1}+A0\overline{B1}\ \overline{B0}+A1A0\overline{B0}$

$A\_eq\_B=\overline{A1}\ \overline{A1}\ \overline{B1}\ \overline{B0}+A1\ A0\overline{B1}B0+A1A0B1B0+A1\overline{A0}B1\overline{B0}$

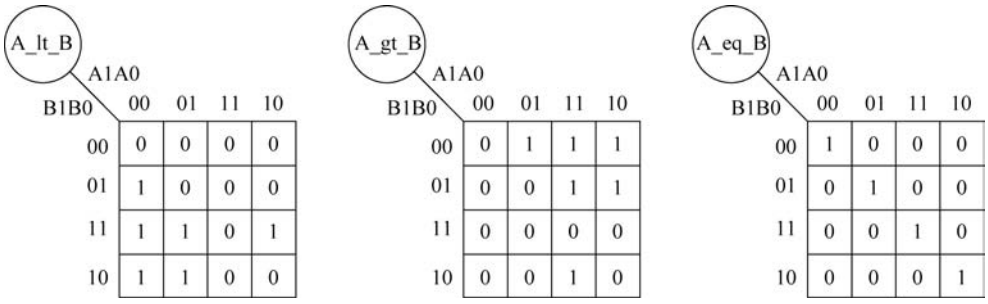


图 3-17 2 位比较器卡诺图

依据上述布尔方程写出的 2 位数据比较器模型如下。

```
module comp_2 (A_gt_B, A_lt_B, A_eq_B, A0, A1, B0, B1);
output A_gt_B, A_lt_B, A_eq_B;
input A0, A1, B0, B1;
assign A_lt_B = ~A1 & B1 | ~A1 & ~A0 & B0 | ~A0 & B1 & B0;
assign A_gt_B = A1 & ~B1 | A0 & ~B1 & ~B0 | A1 & A0 & ~B0;
assign A_eq_B = ~A1 & ~A0 & ~B1 & ~B0 | ~A1 & A0 & ~B1 & B0 | A1 & A0 & B1 & B0 | A1 & ~A0
& B1 & ~B0;
endmodule
```

4 位比较器如图 3-18(a)所示的框图符号表示。比较器通过比较 4 位二进制数来判断它们的相对大小。由于输出的布尔方程不易写出,因而可通过两个 2 位比较器的输出和附加逻辑的连接得到相应输出。连接 2 位比较器的逻辑依据规则是:高位的严格不等能够决定 4 位数的相对大小;如果高位相等,可逐位比较低位,由低位的大小决定输出。如图 3-18(b)所示的层次化结构实现了 4 位比较器。其 Verilog 源代码如下。

```
module comp_4 (
output A_gt_B, A_lt_B, A_eq_B,
input A0, A1, A2, A3, B0, B1, B2, B3);
wire w1, w0, A_gt_B_M1, A_lt_B_M1, A_eq_B_M1, A_gt_B_M0, A_lt_B_M0, A_eq_B_M0;
comp_2 M1 (A_gt_B_M1, A_lt_B_M1, A_eq_B_M1, A2, A3, B2, B3);
comp_2 M0 (A_gt_B_M0, A_lt_B_M0, A_eq_B_M0, A0, A1, B0, B1);
or (A_gt_B, A_gt_B_M1, w1);
and (w1, A_eq_B_M1, A_gt_B_M0);
and (A_eq_B, A_eq_B_M1, A_eq_B_M0);
or (A_lt_B, A_lt_B_M1, w0);
and (w0, A_eq_B_M1, A_lt_B_M0);
endmodule
```

仿真测试给出了 4 位比较器的仿真结果(如图 3-19 所示)。

上述例子中,仅包含两个层次的模块调用,下面的例子将包含更多的层次。

实例二:设计一个 16 位行波进位(ripple-carry)加法器。

本例中的层次化设计图如图 3-20 所示。首先,最顶层模块定义为 Add_rca_16,它可由 4 个 4 位行波进位加法器 Add_rca_4 级联而成。每个 Add_rca_4 模块产生的进位从最低位开始逐次传递至下一级的进位输入端。每个 Add_rca_4 又可以视为 4 个全加器 Add_full

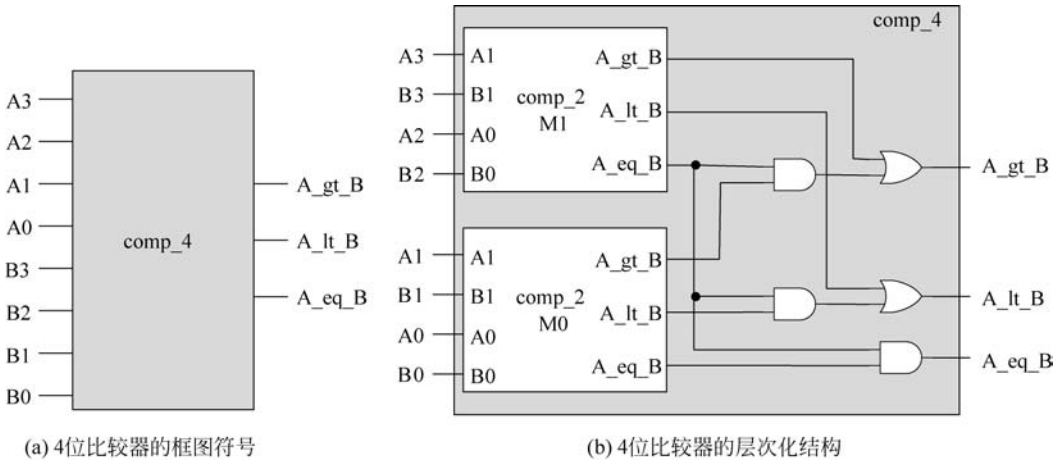


图 3-18　一个 4 位串行加法器的结构图

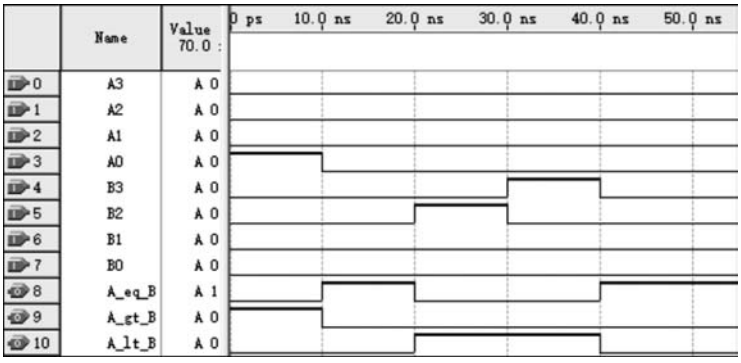


图 3-19　4 位比较器的仿真结果

的级联。每个全加器可分解为半加器 Add_half 和或门的组成。每个半加器再可以分解为门电路结构。

一个 Add_rca_16 的完整描述如下。

```
module Add_rca_16 (c_out, sum, a, b, c_in);      //顶层实体,16 位行波进位加法器
output c_out;
output [15:0] sum;
input[15:0] a, b;
input c_in;
wire c_in4, c_in8, c_in12;
Add_rca_4 M1 (c_in4, sum[3:0], a[3:0], b[3:0], c_in);
Add_rca_4 M2 (c_in8, sum[7:4], a[7:4], b[7:4], c_in4);
Add_rca_4 M3 (c_in12, sum[11:8], a[11:8], b[11:8], c_in8);
Add_rca_4 M4 (c_out, sum[15:12], a[15:12], b[15:12], c_in12);
endmodule

module Add_rca_4 (c_out, sum, a, b, c_in);      //4 位行波进位加法器
output c_out;
output [3:0] sum;
input [3:0] a, b;
```

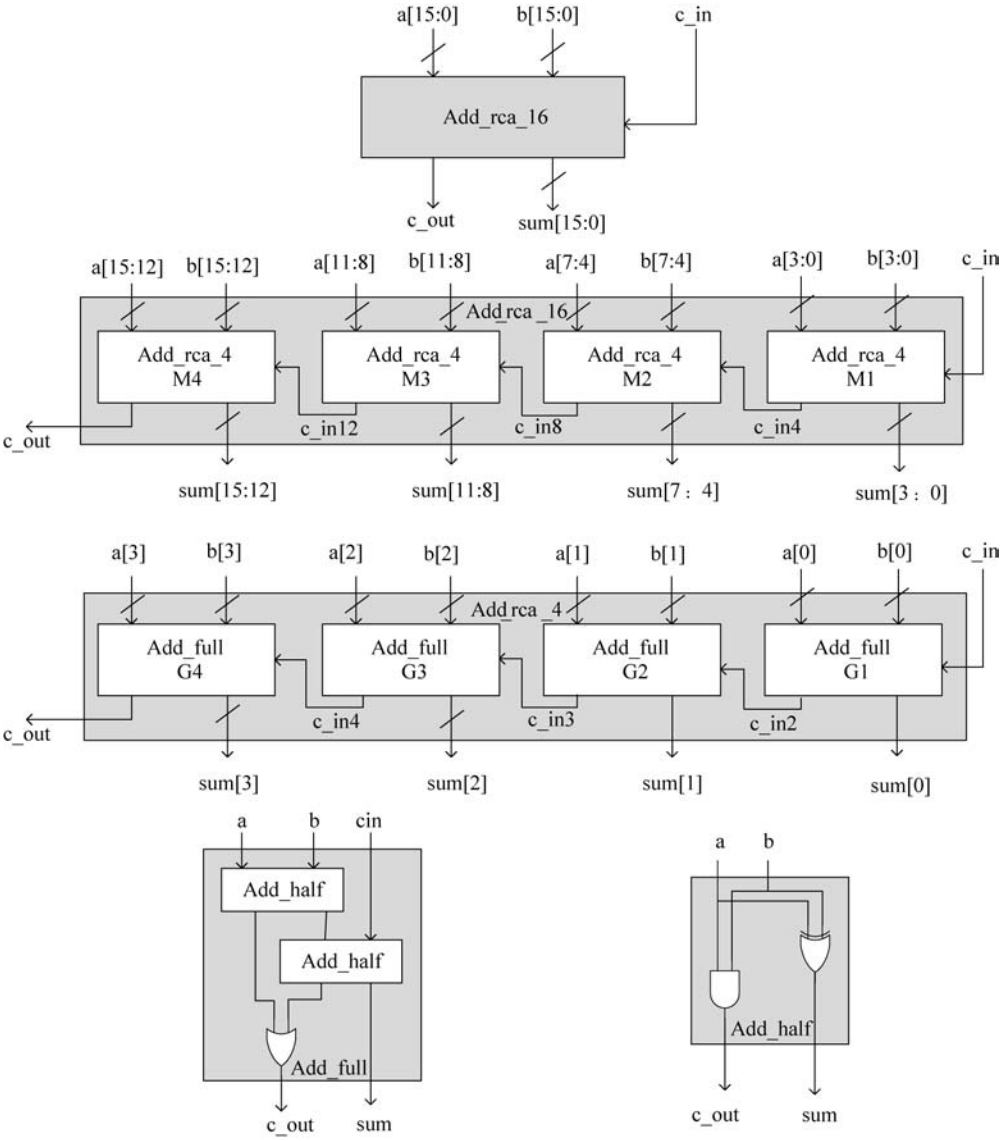


图 3-20 16 位行波进位加法器的层次化设计结构图

```
input c_in;
wire c_in2, c_in3, c_in4;
Add_full M1 (c_in2, sum[0], a[0], b[0], c_in);
Add_full M2 (c_in3, sum[1], a[1], b[1], c_in2);
Add_full M3 (c_in4, sum[2], a[2], b[2], c_in3);
Add_full M4 (c_out, sum[3], a[3], b[3], c_in4);
endmodule

module Add_full (c_out, sum, a, b, c_in);          //全加器
output c_out, sum;
input a, b, c_in;
wire w1, w2, w3;
Add_half M1 (w2, w1, a, b);
Add_half M2 (w3, sum, c_in, w1);
```

```
or M3(c_out, w2, w3);
endmodule

module Add_half(c_out, sum, a, b);      //半加器
output c_out, sum;
input a, b;
xor M1(sum, a, b);
and M2(c_out, a, b);
endmodule
```

Add_rca_16 的层次化模型通过采用模块内嵌套模块的方式说明了 Verilog 是如何支持自顶向下的结构化设计的。图 3-21 说明了 Add_rca_16 的设计层次。顶层功能单元为 Add_rca_16 的封装模块,它包含其他较低复杂度的功能单元的例化及其他模块。最底层是由基本门原语构成的。构成一个设计的所有模块可以放在一个文件中,也可以放在同一个工程的多个文本文件中,当这些文件被一起编译时,就能完全描述高层模块的功能。只要单个模块的描述是以独立文件形式存在的,多个源代码文件模块如何交叉分布都没有关系。

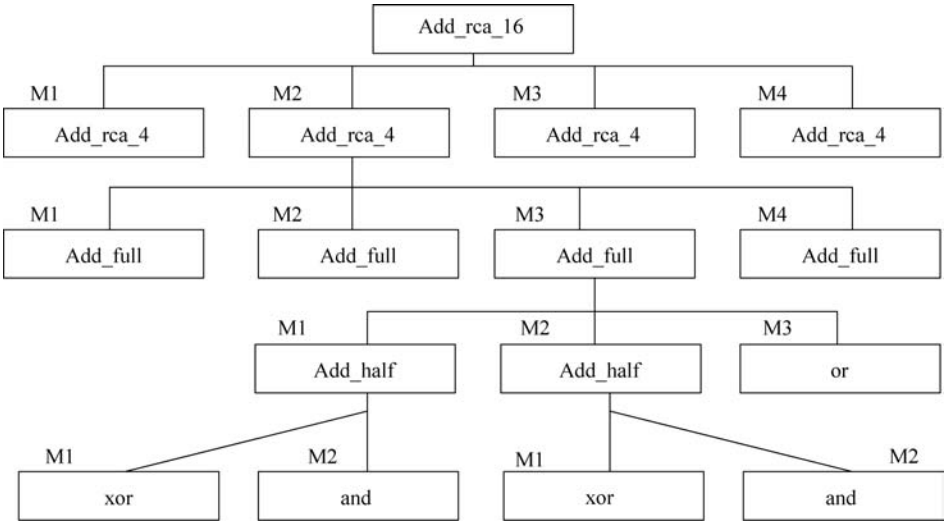


图 3-21　16 位行波进位加法器的设计层次

最终的仿真结果如图 3-22 所示。

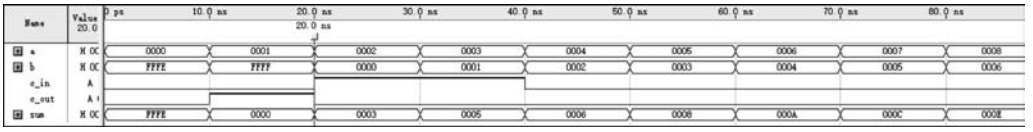


图 3-22　Add_rca_16 的仿真结果

3.7　小结

本章从描述方式与抽象层次两个角度介绍了 Verilog HDL 的建模方式。

系统设计时若基于抽象层次进行建模,则采用的抽象级别越高,设计越容易,程序代码

越简单，但耗用器件资源更多。系统级建模太抽象，有时无法综合成具体的物理电路。基于门级建模的硬件模型不仅可以仿真，而且可综合，系统速度快，但门级建模要求根据逻辑功能画出逻辑电路图，对于复杂的数字系统很难做到，一般适合小型设计；算法级和 RTL 级建模级别适中，代码不是很复杂，且一般容易综合成具体的物理电路。通常用 RTL 级建模来完成逻辑功能，尽量避免用门级建模，除非对系统速度要求比较高的场合才采用门级建模方式。

从建模的描述方式来看，结构化建模描述方式与电路结构一一对应，建模前必须设计好详细电路，比较适合逻辑门级和晶体管开关级的抽象层次；行为描述方式与硬件的物理实现无关，用于模拟器件的行为，以检验它是否执行正确的功能。所以，行为描述通常用于抽象层次建模方式中系统级建模、算法级建模及 RTL 级建模。数据流描述方式介于二者之间，可以根据信号在电路中的传输情况进行描述，也可以根据该系统实现的逻辑表达式直接进行描述，适合对组合逻辑电路进行建模。

对一个复杂电路系统进行设计，通常对顶层设计采用结构描述方式，对底层模块可采用数据流、行为级或两者的结合。

习题 3

1. 分别用结构化描述方式、行为描述方式和数据流描述方法设计如图 3-23 所示电路。

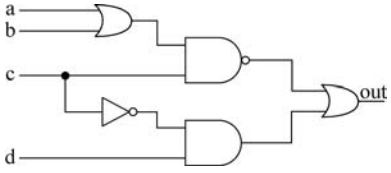


图 3-23 电路图

2. 请利用本章中关于全加器的编写思路，分别从算法层级和 RTL 层级编写 4 位减法器的 Verilog HDL 源程序。

3. 根据表 3-7 中提供的 JK 触发器的逻辑功能，编写 JK 触发器的 Verilog HDL 源程序。

要求：(1)采用行为建模描述方式编写。

(2) 采用用户定义原语(UDP)建模方式编写。

表 3-7 JK 触发器逻辑功能

J	K	Q^n	Q^{n+1}	功能
0	0	0	0	$Q^{n+1}=Q^n$
0	0	1	1	保持
0	1	0	0	$Q^{n+1}=0$
0	1	1	0	置 0
1	0	0	1	$Q^{n+1}=1$
1	0	1	1	置 1
1	1	0	1	$Q^{n+1}=\sim Q^n$
1	1	1	0	翻转

4. 编写一个功能模块 FU,使其具有 5 个端口：输出端口 clk,宽度为 1 位；输入端口 data1 和 data2,宽度为 8 位；输出端口 dout1,宽度为 8 位；输出端口 dout2,宽度为 4 位。完成模块的定义和端口声明,内部功能描述不需要编写。再编写一个顶层模块 top,调用 FU 模块。端口映射先采用名称关联实现,再使用顺序连接方式端口映射。

5. 编写一个 74138 功能译码器程序,再以这个模块和基本门电路为基础,实现一个多数表决器。多数表决器的功能是：有三位裁判,当举重选手完成比赛时,有多数裁判认定成功,则结果为成功；否则失败。