文档中心	文档编号	版本	密级
		1.0	*内部公开*
	资源类别：HDL语言		共41页

Verilog HDL入门教程

(仅供内部使用)

拟制：		中研基础	日期：	2004.8.3
批准：		中研基础	日期：	
批准：			日期：	yyyy/mm/dd

修订记录

日期	修订版本	描述	作者
2004.8.3	1.00	初稿完成	

目 录

Verilog HDL 入门教程

关键词：

摘　要：本文主要介绍了Verilog HDL 语言的一些基本知识，目的是使初学者能够迅速掌握HDL设计方法，初步了解并掌握Verilog HDL语言的基本要素，能够读懂简单的设计代码并能够进行一些简单设计的Verilog HDL建模。

缩略语清单： *对本文所用缩略语进行说明，要求提供每个缩略语的英文全名和中文解释。*

参考资料清单： *请在表格中罗列本文档所引用的有关参考文献名称、作者、标题、编号、发布日期和出版单位等基本信息。*

参考资料清单					
名称	作者	编号	发布日期	查阅地点或渠道	出版单位*（若不为本公司发布的文献，请填写此列）*
Quisck Reference for Verilog HDL	AMBIT Design System			苏文彪	
Verilog HDL 硬件描述语言	J.Bhasker 著 徐振林 等译		2000.7	图书馆	机械工业出版社

1 前言

当前业界的硬件描述语言中主要有VHDL 和Verilog HDL。公司根据本身ASIC设计现有的特点、现状，主推Verilog HDL 语言，逐渐淡化VHDL语言，从而统一公司的ASIC/FPGA设计平台，简化流程。

为使新员工在上岗培训中能迅速掌握ASIC/FPGA 设计的基本技能，中研基础部ASIC设计中心开发了一系列的培训教材。该套HDL语言培训系列包括如下教程：

《Verilog HDL 入门教程》

《Verilog HDL 代码书写规范》

《Verilog 基本电路设计指导书》

《TestBench 编码技术》

系列教材完成得较匆忙，本身尚有许多不完善的地方，同时，可能还需要其他知识方面的培训但没有形成培训教材，希望大家在培训过程中，多提宝贵意见，以便我们对它进行修改和完善。

2 HDL设计方法学简介

2.1 数字电路设计方法

当前的数字电路设计从层次上分可分成以下几个层次：

1. 算法级设计：利用高级语言如C语言及其他一些系统分析工具（如MATLAB）对设计从系统的算法级方式进行描述。算法级不需要包含时序信息。

2. RTL级设计：用数据流在寄存器间传输的模式来对设计进行描述。

3. 门级：用逻辑级的与、或、非门等门级之间的连接对设计进行描述。

4. 开关级：用晶体管和寄存器及他们之间的连线关系来对设计进行描述。

算法级是高级的建模，一般对特大型设计或有较复杂的算法时使用，特别是通讯方面的一些系统，通过算法级的建模来保证设计的系统性能。在算法级通过后，再把算法级用RTL级进行描述。门级一般对小型设计可适合。开关级一般是在版图级进行。

2.2 硬件描述语言

在传统的设计方法中，当设计工程师设计一个新的硬件、一个新的数字电路或一个数字逻辑系统时，他或许在CAE 工作站上做设计，为了能在CAE工作站做设计，设计者必须为设计画一张线路图，通常地，线路图是由表示信号的线和表示基本设计单元的符号连在一起组成线路图，符号取自设计者用于构造线路图的零件库。若设计者是用标准逻辑器件（如74系列等）做板极设计线路图，那么在线路图中，符号取自标准逻辑零件符号库；若设计是进行ASIC设计，则这些符号取自ASIC库的可用的专用宏单元。这就是传统的原理图设计方法。

对线路图的逻辑优化，设计者或许利用一些EDA工具或者人工地进行逻辑的布尔函数逻辑优化。为了能够对设计进行验证，设计者必须通过搭个硬件平台（如电路板），对设计进行验证。

随着电子设计技术的飞速发展，设计的集成度、复杂度越来越高，传统的设计方法已满足不了设计的要求，因此要求能够借助当今先进的EDA工具，使用一种描述语言，对数字电路和数字逻辑系统能够进行形式化的描述，这就是硬件描述语言。

硬件描述语言HDL（Hardware Description Language ）是一种用形式化方法来描述数字电路和数字逻辑系统的语言。数字逻辑电路设计者可利用这种语言来描述自己的设计思想，然后利用EDA工具进行仿真，再自动综合到门级电路，最后用ASIC或FPGA实现其功能。举个例子，在传统的设计方法中，对2输入的与门，我们可能需到标准器件库中调个74系列的器件出来，但在硬件描述语言中，“& ”就是一个与门的形式描述，“C = A & B”就是一个2输入与门的描述。而“and ”就是一个与门器件。

硬件描述语言发展至今已有二十多年历史，当今业界的标准中（IEEE标准）主要有VHDL和Verilog HDL 这两种硬件描述语言。

2.3 设计方法学

当前的ASIC设计有多种设计方法，但一般地采用自顶向下的设计方法。

随着技术的发展，一个芯片上往往集成了几十万到几百万个器件，传统的自底向上的设计方法已不太现实。因此，一个设计往往从系统级设计开始，把系统划分成几个大的基本的功能模块，每个功能模块再按一定的规则分成下一个层次的基本单元，如此一直划分下去。自顶向下的设计方法可用下面的树状结构表示：

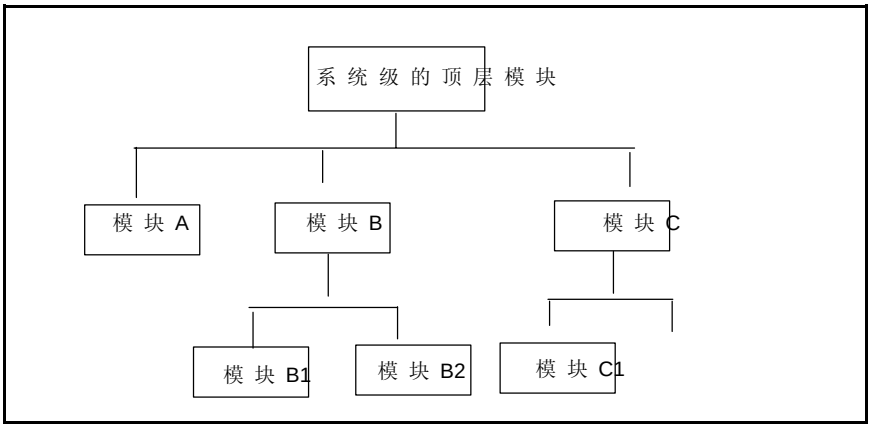

图1 TOP-DOWN 设计思想

通过自顶向下的设计方法，可实现设计的结构化，使一个复杂的系统设计可由多个设计者分工合作；还可以实现层次化的管理。

2.4 Verilog HDL简介

Verilog HDL 是一种硬件描述语言，用于从算法级、RTL级、门级到开关级的多种抽象设计层次的数字系统建模。被建模的数字系统对象的复杂性可介于简单的门级和完整的电子数字系统之间。数字系统可按层次描述。

2.4.1 历史

Verilog HDL 语言最初是于1983 年由Gateway Design Automation 公司为其模拟器产品开发的硬件建模语言。那时它只是一种专用语言。由于他们的模拟、仿真器产品的广泛使用，Verilog HDL 作为一种便于使用且实用的语言逐渐为众多设计者所接受。在一次努力增加语言普及性的活动中，Verilog HDL 语言于1990 年被推向公众领域。Open Verilog International（ＯＶＩ）是促进Verilog 发展的国际性组织。1992 年，OVI 决定致力于推广Verilog OVI 标准成为IEEE 标准。这一努力最后获得成功，Verilog 语言于1995 年成为IEEE 标准，称为IEEE Std1364－1995 。完整的标准在Verilog 硬件描述语言参考手册中有详细描述。

2.4.2 能力

对初学者，可先大致了解一下Verilog HDL所提供的能力，掌握Verilog HDL 语言的核心子集就可以了。

1. 概述

Verilog HDL 语言具有下述描述能力：设计的行为特性、设计的数据流特性、设计的结构组成以及包含响应监控和设计验证方面的时延和波形产生机制。所有这些都使用同一种建模语言。此外，Verilog HDL 语言提供了编程语言接口，通过该接口可以在模拟、验证期间从设计外部访问设计，包括模拟的具体控制和运行。

Verilog HDL 语言不仅定义了语法，而且对每个语法结构都定义了清晰的模拟、仿真语义。因此，用这种语言编写的模型能够使用Verilog 仿真器进行验证。语言从C 编程语言中继承了多种操作符和结构。Verilog HDL 提供了扩展的建模能力，其中许多扩展最初很难理解。但是，Verilog

HDL 语言的核心子集非常易于学习和使用，这对大多数建模应用来说已经足够。当然,完整的硬件描述语言足以对从最复杂的芯片到完整的电子系统进行描述。

2. 主要功能list

- 基本逻辑门，例如and 、or 和nand 等都内置在语言中。
- 开关级基本结构模型，例如pmos 和nmos 等也被内置在语言中。
- 可采用三种不同方式或混合方式对设计建模。这些方式包括：行为描述方式—使用过程化结构建模；数据流方式—使用连续赋值语句方式建模；结构化方式—使用门和模块实例语句描述建模。
- Verilog HDL 中有两类数据类型：线网数据类型和寄存器数据类型。线网类型表示构件间的物理连线，而寄存器类型表示抽象的数据存储元件。
- 能够描述层次设计，可使用模块实例结构描述任何层次。
- 设计的规模可以是任意的；语言不对设计的规模（大小）施加任何限制。
- Verilog HDL 不再是某些公司的专有语言而是I E E E 标准。
- 人和机器都可阅读Verilog 语言，因此它可作为E D A 的工具和设计者之间的交互语言。
- 设计能够在多个层次上加以描述，从开关级、门级、寄存器传送级（RT L ）到算法级。
- 能够使用内置开关级原语在开关级对设计完整建模。
- 同一语言可用于生成模拟激励和指定测试的验证约束条件，例如输入值的指定。
- Verilog HDL 能够监控模拟验证的执行，即模拟验证执行过程中设计的值能够被监控和显示。这些值也能够用于与期望值比较，在不匹配的情况下，打印报告消息。
- 在行为级描述中，Verilog HDL 不仅能够在RT L 级上进行设计描述，而且能够在体系结构级描述及其算法级行为上进行设计描述。
- 能够使用门和模块实例化语句在结构级进行结构描述。
- 对高级编程语言结构，例如条件语句、情况语句和循环语句，语言中都可以使用。

下图显示了Verilog HDL 的混合方式建模能力，即在一个设计中每个模块均可以在不同设计层次上建模。

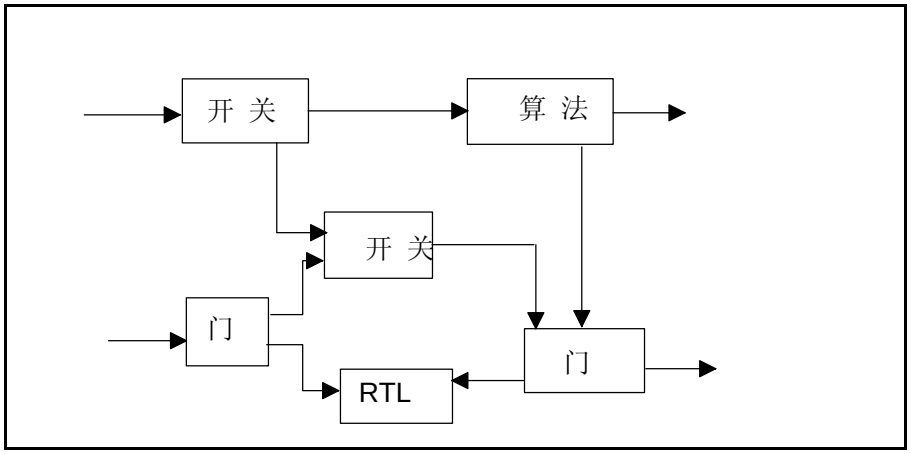

图2 混合设计层次建模示意图

3 Verilog HDL 建模概述

在数字电路设计中，数字电路可简单归纳为两种要素：线和器件。线是器件管脚之间的物理连线；器件也可简单归纳为组合逻辑器件（如与或非门等）和时序逻辑器件（如寄存器、锁存器、RAM等）。一个数字系统（硬件）就是多个器件通过一定的连线关系组合在一块的。因此，Verilog HDL的建模实际上就是如何使用HDL语言对数字电路的两种基本要素的特性及相互之间的关系进行描述的过程。

下面通过一些实例，以便对Verilog HDL 的设计建模有个大概的印象。

3.1 模块

模块（module）是Verilog 的基本描述单位，用于描述某个设计的功能或结构及与其他模块通信的外部端口。

模块在概念上可等同一个器件就如我们调用通用器件（与门、三态门等）或通用宏单元（计数器、ALU、CPU）等，因此，一个模块可在另一个模块中调用。

一个电路设计可由多个模块组合而成，因此一个模块的设计只是一个系统设计中的某个层次设计，模块设计可采用多种建模方式。

3.1.1 简单事例

下面先介绍几个简单的Verilog HDL程序。

例[1] 加法器

```
module addr (a, b, cin, count, sum);
    input [2:0] a;
    input [2:0] b;
    input cin;
    output count;
    output [2:0] sum;

    assign  {count,sum} = a +b + cin;
endmodule
```

该例描述一个3位加法器，从例子可看出整个模块是以module 开始，endmodule 结束。

例[2] 比较器

```
module compare （equal，a，b）;
    input    [1:0]  a,b;   // declare  the  input  signal ;
    output   equare ;      // declare  the output signal;

    assign   equare = (a == b) ? 1:0 ;
    / * if  a = b , output  1, otherwise   0；*/
endmodule
```

该例描述一个比较器， 从上可看到，/* .... */ 和 // ... 表示注释部分。注释只是为了方便设计者读懂代码，对编译并不起作用。

例[3] 三态驱动器

```
module mytri (din, d_en, d_out);
    input din;
    input d_en;
    output d_out;

    // -- Enter your statements here -- //
    assign   d_out = d_en ? din :'bz;
endmodule

module trist (din, d_en, d_out);
    input din;
    input d_en;
    output d_out;

// -- statements here -- //
    mytri  u_mytri(din,d_en,d_out);
endmodule
```

该例描述了一个三态驱动器。其中三态驱动门在模块 mytri 中描述，而在模块trist 中调用了模块mytri 。模块mytri 对trist 而言相当于一个已存在的器件，在trist 模块中对该器件进行实例化，实例化名 u_mytri 。

3.1.2 模块的结构

通过上面的实例可看出，一个设计是由一个个模块（module）构成的。一个模块的设计如下：

1、模块内容是嵌在module 和endmodule两个语句之间。每个模块实现特定的功能，模块可进行层次的嵌套，因此可以将大型的数字电路设计分割成大小不一的小模块来实现特定的功能，最后通过由顶层模块调用子模块来实现整体功能，这就是Top-Down的设计思想，如 3.3.1的例[3]。

2、模块包括接口描述部分和逻辑功能描述部分。这可以把模块与器件相类比。

模块的端口定义部分：

如上例： module addr (a, b, cin, count, sum); 其中module 是模块的保留字，addr 是模块的名字，相当于器件名。（）内是该模块的端口声明，定义了该模块的管脚名，是该模块与其他模块通讯的外部接口，相当于器件的pin 。

模块的内容，包括I/O说明，内部信号、调用模块等的声明语句和功能定义语句。

I/O说明语句如： input [2:0] a; input [2:0] b; input cin; output count; 其中的 input 、output、inout 是保留字，定义了管脚信号的流向，[n:0]表示该信号的位宽（总线或单根信号线）。

逻辑功能描述部分如： assign d_out = d_en ? din :'bz;

mytri u_mytri(din,d_en,d_out);

功能描述用来产生各种逻辑（主要是组合逻辑和时序逻辑，可用多种方法进行描述，具体的用法下面章节有介绍），还可用来实例化一个器件，该器件可以是厂家的器件库也可以是我们自己用HDL设计的模块（相当于在原理图输入时调用一个库元件）。在逻辑功能描述中，主要用到assign 和always 两个语句。

3、对每个模块都要进行端口定义，并说明输入、输出口，然后对模块的功能进行逻辑描述，当然，对测试模块，可以没有输入输出口。

4、Verilog HDL 的书写格式自由，一行可以写几个语句，也可以一个语句分几行写。具体由代码书写规范约束。

5、除endmodule 语句外，每个语句后面需有分号表示该语句结束。

3.1.3 模块语法

1. 一个模块的基本语法如下：

一个模块的基本语法如下：

```
module   module_name (port1, port2, ......) ;
// D e c l a r a t i o n s :
    input, output, inout,
    reg, wire, parameter,
    function, task, . . .
//S t a t e m e n t s :
    Initial statement
    Always statement
    Module instantiation
    Gate instantiation
    Continuous assignment
endmodule
```

模块的结构需按上面的顺序进行，声明区用来对信号方向、信号数据类型、函数、任务、参数等进行描述。语句区用来对功能进行描述如：器件调用 （Module instantiation ）等。

2. 书写语法建议

一个模块用一个文件；

模块名与文件名要同名；

一行一句语句。

信号方向按输入、输出、双向顺序描述。

设计模块时可尽量考虑采用参数化，提高设计的重用。

以上是初学者的建议，具体的或深入的方面可看相关的文档。下面的有关语法建议类似。

3.2 时延

信号在电路中传输会有传播延时等，如线延时、器件延时。时延就是对延时特性的HDL描述。举例如下：

assign # 2　B = A;

表示 B信号在2个时间单位后得到A信号的值。如下图：

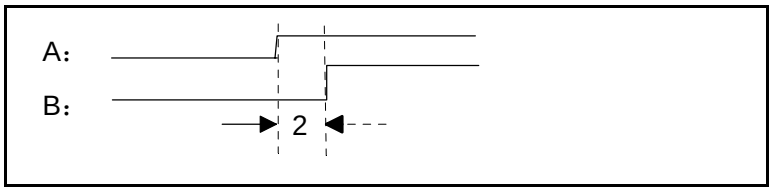

图3

在Verilog HDL中，所有时延都必须根据时间单位进行定义，定义方式为在文件头添加如下语句：

`timescale　1ns /100ps

其中’timescale 是Verilog HDL 提供的预编译处理命令， 1ns 表示时间单位是1ns ， 100ps表示时间精度是100ps。根据该命令，编译工具才可以认知 #2　为2ns。

在Verilog HDL 的IEEE标准中没有规定时间单位的缺省值，由各模拟工具确定。因此，在写代码时必须确定。

3.3 三种建模方式

在HDL的建模中，主要有结构化描述方式、数据流描述方式和行为描述方式，下面分别举例说明三者之间的区别。

3.3.1 结构化描述方式

结构化的建模方式就是通过对电路结构的描述来建模，即通过对器件的调用（HDL概念称为例化），并使用线网来连接各器件的描述方式。这里的器件包括Verilog HDL的内置门如与门and ，异或门xor等，也可以是用户的一个设计。结构化的描述方式反映了一个设计的层次结构。

例[1]：一位全加器

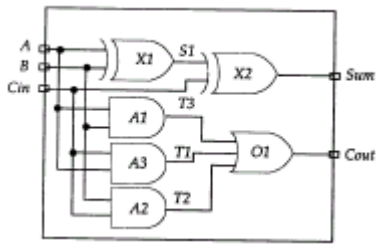

图4 一位全加器的结构图

代码：

```
module FA_struct (A, B, Cin, Sum, Count);
        input A;
        input B;
        input Cin;
```

```
    output Sum;
    output Count;
    wire S1, T1, T2, T3;

    // -- statements -- //
    xor x1 (S1, A, B);
    xor x2 (Sum, S1, Cin);

    and A1 (T3, A, B );
    and A2 (T2, B, Cin);
    and A3 (T1, A, Cin);

    or O1 (Cout, T1, T2, T3 );
endmodule
```

该实例显示了一个全加器由两个异或门、三个与门、一个或门构成。S1、T1、T2、T3则是门与门之间的连线。代码显示了用纯结构的建模方式，其中xor 、and、or 是Verilog HDL 内置的门器件。以 xor x1 (S1, A, B) 该例化语句为例：

xor 表明调用一个内置的异或门，器件名称xor，代码实例化名x1（类似原理图输入方式）。

括号内的S1，A，B 表明该器件管脚的实际连接线（信号）的名称，其中 A、B是输入，S1是输出。其他同。

例[2]：两位的全加器

两位的全加器可通过调用两个一位的全加器来实现。该设计的设计层次示意图和结构图如下：

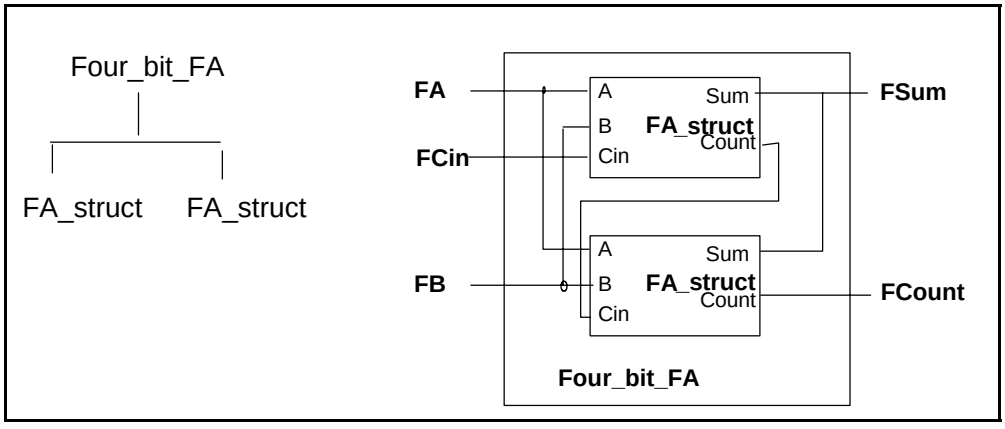

图5 两位全加器的结构示意图

代码：

```
module  Four_bit_FA  (FA, FB, FCin, FSum, FCout ) ;
    parameter  SIZE = 2;
    input [SIZE:1] FA;
```

```
input [SIZE:1]  FB;
input  FCin;

output [SIZE:1] FSum;
output FCout;

wire FTemp;

FA_struct   FA1(
            .A (FA[1]),
            .B (FB[1]),
           .Cin (FCin) ,
           .Sum (FSum[1]),
           .Cout (Ftemp)
 );
FA_struct    FA2(
            .A (FA[2]),
            .B (FB[2]),
            .Cin (FTemp) ,
           .Sum (FSum[2]),
           .Cout (FCount )
 );

endmodule
```

该实例用结构化建模方式进行一个两位的全加器的设计，顶层模块Four_bit_FA 调用了两个一位的全加器 FA_struct 。在这里，以前的设计模块FA_struct 对顶层而言是一个现成的器件，顶层模块只要进行例化就可以了。

注意这里的例化中，端口映射（管脚的连线）采用名字关联，如 .A （FA[2]），其中.A 表示调用器件的管脚A，括号中的信号表示接到该管脚A的电路中的具体信号。

wire 保留字表明信号Ftemp 是属线网类型（下面有具体描述）。

另外，在设计中，尽量考虑参数化的问题。

器件的端口映射必须采用名字关联。

3.3.2 数据流描述方式

数据流的建模方式就是通过对数据流在设计中的具体行为的描述的来建模。最基本的机制就是用连续赋值语句。在连续赋值语句中，某个值被赋给某个线网变量（信号），语法如下：

assign [delay] net_name = expression;

如：assign #2   A =   B;

在数据流描述方式中，还必须借助于HDL提供的一些运算符，如按位逻辑运算符 ：逻辑与（&)，逻辑或（|）等。

以上面的全加器为例，可用如下的建模方式：

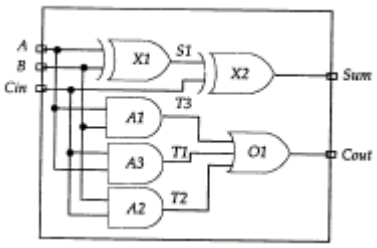

图6 一位全加器的结构图

```
`timescale  1ns/100ps
module  FA_flow(A,B,Cin,Sum,Count)
    input A,B,Cin;
    output  Sum, Count;

    wire  S1,T1,T2,T3;

    assign   # 2 S1 =  A  ^ B;
    assign   # 2   Sum =  S1 ^ Cin;
    assign  #2  T3 =  A & B;
    assign  #2   T1 =  A & Cin;
    assign #2   T2  =  B & Cin ;
endmodule
```

注意在各assign 语句之间，是并行执行的，即各语句的执行与语句之间的顺序无关。如上，当A有个变化时，S1、T3、T1 将同时变化，S1的变化又会造成Sum的变化。

3.3.3 行为描述方式

行为方式的建模是指采用对信号行为级的描述（不是结构级的描述）的方法来建模。在表示方面，类似数据流的建模方式，但一般是把用initial 块语句或always 块语句描述的归为行为建模方式。行为建模方式通常需要借助一些行为级的运算符如加法运算符（+），减法运算符（-）等。

以下举个例子，对initial 和always 语句的具体应用可看相关章节的介绍，这里，先对行为建模方式有个概念。

例[1] 一位全加器的行为建模

```
module FA_behav1(A, B, Cin, Sum, Cout );
    input A,B,Cin;
    output Sum,Cout;
```

```
    reg  Sum, Cout;
    reg  T1,T2,T3;

    always@ ( A or B or Cin )
    begin
        Sum = (A ^ B) ^ Cin ;
        T1 = A & Cin;
        T2 = B & Cin ;
        T3 = A & B;
        Cout = (T1| T2) | T3;
    end
endmodule
```

需要先建立以下概念：

1、只有寄存器类型的信号才可以在always和initial 语句中进行赋值，类型定义通过reg语句实现。

2、always 语句是一直重复执行，由敏感表（always 语句括号内的变量）中的变量触发。

3、always 语句从0 时刻开始。

4、在begin 和end 之间的语句是顺序执行，属于串行语句。

例[2]：一位全加器的行为建模

```
module FA_behav2(A, B, Cin, Sum, Cout );
    input A,B,Cin;
    output Sum,Cout;

    reg  Sum, Cout;

    always@ ( A or B or Cin )
    begin
        {Count ， Sum} = A  +  B + Cin ;
    end
endmodule
```

在例2中，采用更加高级（更趋于行为级）描述方式，即直接采用 “+”来描述加法。{Count，Sum}表示对位数的扩展，因为两个1bit 相加，和有两位，低位放在Sum 变量中，进位放在Count 中。

3.3.4 混合设计描述

在实际的设计中，往往是多种设计模型的混合。一般地，对顶层设计，采用结构描述方式，对低层模块，可采用数据流、行为级或两者的结合。如上面的两bit 全加器，对顶层模块（Four_bit_FA)采用结构描述方式对低层进行例化，对低层模块（FA）可采用结构描述、数据流描述或行为级描述。

4 Verilog HDL 基本语法

本节介绍Verilog HDL 语言的一些基本要素，包括标识符、注释、格式、数字值集合、两种数据类型、运算符和表达式和一些基本的语句如IF语句等。

4.1 标识符

4.1.1 定义

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

Count

COUNT //与Count 不同。

R56_68

FIVE$

4.1.2 关键词

Verilog HDL 定义了一系列保留字，叫做关键词，附录A 列出了语言中的所有保留字。注意只有小写的关键词才是保留字。例如，标识符always (这是个关键词)与标识符ALWAYS(非关键词)是不同的。

4.1.3 书写规范建议

以下是一些书写规范的要求，可参考公司的《Verilog 代码书写规范》。

1、用有意义的有效的名字如 Sum 、CPU_addr等。

2、用下划线区分词。

3、采用一些前缀或后缀，如

时钟采用Clk 前缀：Clk_50，Clk_CPU；

低电平采用_n 后缀：Enable_n；

4、统一一定的缩写 如全局复位信号 Rst。

5、同一信号在不同层次保持一致性，如同一时钟信号必须在各模块保持一致。

6、自定义的标识符不能与保留字同名。

7、参数采用大写，如SIZE。

4.2 注释

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

/* statement1，

statement2，

.. ...

statementn */

以上n个语句都是注释语句。

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

4.3 格式

Verilog HDL 是区分大小写的，即大小写不同的标识符是不同的。

另外Verilog HDL的书写格式是自由的，即一条语句可多行书写；一行可写多个语句。

白空（新行、制表符、空格）没有特殊意义。

如 input A；input B；与

input A；

input B；

是一样的。

书写规范建议：

一个语句一行。

采用空四格的table 键进行缩进。

4.4 数字值集合

本小节介绍Verilog HDL的值的集合和常量（ 整型、实型、字符型）和变量等。

4.4.1 值集合

Verilog HDL中规定了四种基本的值类型：

0：逻辑0或“假”；

1：逻辑1或“真”；

X：未知值；

Z：高阻。

注意这四种值的解释都内置于语言中。如一个为z 的值总是意味着高阻抗，一个为0 的值通常是指逻辑0 。

在门的输入或一个表达式中的为“z ”的值通常解释成“x ”。

此外，x 值和z 值都是不分大小写的，也就是说，值0x1z 与值0X1Z 相同。

Verilog HDL 中的常量是由以上这四类基本值组成的。

4.4.2 常量

Verilog HDL 中有三种常量：

整型、实型、字符串型。

下划线符号（_）可以随意用在整数或实数中，它们就数量本身没有意义。它们能用来提高易读性；唯一的限制是下划线符号不能用作为首字符。

下面主要介绍整型和字符串型。

1. 整型

整型数可以按如下两种方式书写：

1) 简单的十进制数格式

2) 基数格式

A. 简单的十进制格式

这种形式的整数定义为带有一个可选的“+”（一元）或“－”（一元）操作符的数字序列。下面是这种简易十进制形式整数的例子。

32　　十进制数32

－15　　十进制数－15

B. 基数表示法

这种形式的整数格式为：

[size] 'base value

size 定义以位计的常量的位长；base 为o 或O（表示八进制），b 或B（表示二进制），d 或D（表示十进制），h 或H （表示十六进制）之一；value 是基于base 的值的数字序列。值x 和z 以及十六进制中的a 到f 不区分大小写。

下面是一些具体实例：

5 'O37	5 位八进制数（二进制 11111 ）
4 'D2	4 位十进制数 （二进制0011）
4 'B1x_01	4 位二进制数
7 'Hx	7位x(扩展的x), 即xxxxxxx
4 'hZ	4 位z(扩展的z) , 即zzzz
4 'd-4	非法：数值不能为负
8 'h 2A	在位长和字符之间，以及基数和数值之间允许出现空格
3' b 001	非法：`和基数b 之间不允许出现空格
(2+3)'b10	非法：位长不能够为表达式

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 ）。

基数格式计数形式的数通常为无符号数。这种形式的整型数的长度定义是可选的。如果没有定义一个整数型的长度，数的长度为相应值中定义的位数。下面是两个例子：

'o 721	9 位八进制数
'h AF	8 位十六进制数

如果定义的长度比为常量指定的长度长，通常在左边填0 补位。但是如果数最左边一位为x 或z ，就相应地用x 或z 在左边补位。例如：

10'b10 左边添0 占位, 0000000010

10'bx0x1 左边添x 占位, x x x x x x 0 x 1

如果长度定义得更小，那么最左边的位相应地被截断。例如：

3 ' b1001 _ 0011 与3'b011 相等

5'H0FFF 与5'H1F 相等

2. 字符串型

字符串是双引号内的字符序列。字符串不能分成多行书写。例如：

"INTERNAL ERROR"

" REACHED－>HERE "

用8 位ASCII 值表示的字符可看作是无符号整数。因此字符串是8 位ASCII 值的序列。为存储字符串“INTERNAL ERROR ”，变量需要8 * 1 4 位。

r e g [1: 8*14] Message;

. . .

Message = "INTERNAL ERROR"

4.5 数据类型

Verilog HDL 主要包括两种数据类型

线网类型(net type) 和 寄存器类型（reg type）。

4.5.1 线网类型

1. wire 和 tri 定义

线网类型主要有wire 和tri 两种。线网类型用于对结构化器件之间的物理连线的建模。如器件的管脚，内部器件如与门的输出等。以上面的加法器为例，输入信号A，B是由外部器件所驱动，异或门X1的输出S1是与异或门X2输入脚相连的物理连接线，它由异或门X1所驱动。

由于线网类型代表的是物理连接线，因此它不存贮逻辑值。必须由器件所驱动。通常由assign 进行赋值。如 assign A = B ^ C;

当一个wire 类型的信号没有被驱动时，缺省值为Z（高阻）。

信号没有定义数据类型时，缺省为 wire 类型。

如上面一位全加器的端口信号 A，B，SUM等，没有定义类型，故缺省为wire 线网类型。

2. 两者区别

tri 主要用于定义三态的线网。

4.5.2 寄存器类型

1. 定义

reg 是最常用的寄存器类型，寄存器类型通常用于对存储单元的描述，如D型触发器、ROM等。存储器类型的信号当在某种触发机制下分配了一个值，在分配下一个值之时保留原值。但必须注意的是，reg 类型的变量，不一定是存储单元，如在always 语句中进行描述的必须用reg 类型的变量。

reg 类型定义语法如下：

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

msb 和lsb 定义了范围，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]   Sat;        // S a t 为4 位寄存器。
reg  Cnt;                //1 位寄存器。
reg [1:32]   Kisp, Pisp, Lisp ;
```

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，如：

```
reg   A ;
.....
A = -1;
....
```

则A的二进制为 1111，在运算中，A总按 无符号数15 来看待。

2. 寄存器类型的存储单元建模举例

用寄存器类型来构建两位的D触发器如下：

```
reg  [1：0]  Dout ;
.....
always@(posedge Clk)
    Dout    <=   Din;
....
```

用寄存器数组类型来建立存储器的模型，如对2个8位的RAM建模如下：

```
reg   [7：0]  Mem[0：1] ;
```

对存储单元的赋值必须一个个第赋值，如上2个8位的RAM的赋值必须用两条赋值语句：

```
.....
Mem[0]  =  ’h 55;
Mem[1]  =  ’haa;
....
```

3. 书写规范建议

对数组类型，请按降序方式，如[7：0] ；

4.6 运算符和表达式

4.6.1 算术运算符

在常用的算术运算符主要是：

加法（二元运算符）：“+”；

减法 （二元运算符）： “-”；

乘法（二元运算符）：“*”；

在算术运算符的使用中，注意如下两个问题：

1. 算术操作结果的位数长度

算术表达式结果的长度由最长的操作数决定。在赋值语句下，算术操作结果的长度由操作符左端目标长度决定。考虑如下实例：

```
reg [3:0] Arc, Bar, Crt;
reg [5:0] Frx;
. . .
Arc = Bar + Crt;
Frx = Bar + Crt;
```

第一个加的结果长度由Bar ，Crt 和A rc 长度决定，长度为4 位。

第二个加法操作的长度同样由Frx 的长度决定（Frx 、Bat 和Crt 中的最长长度），长度为6 位。

在第一个赋值中，加法操作的溢出部分被丢弃；而在第二个赋值中，任何溢出的位存储在结果位Frx [4]中。

在较大的表达式中，中间结果的长度如何确定？在Verilog HDL 中定义了如下规则：表达式中的所有中间结果应取最大操作数的长度（赋值时，此规则也包括左端目标）。考虑另一个实例：

```
wire [4:1] Box, Drt;
wire [5:1] Cfg;
wire [6:1] Peg;
wire [8:1] Adt;
. . .
assign Adt = (Box + Cfg) + (Drt + Peg) ;
```

表达式右端的操作数最长为6 ，但是将左端包含在内时，最大长度为8 。所以所有的加操作使用8 位进行。例如：Box 和Cfg 相加的结果长度为8 位。

2. 有符号数和无符号数

在设计中，请先按无符号数进行。

4.6.2 关系运算符

关系运算符有：

?>（大于）

?<（小于）

?>=（不小于）

?<=（不大于）

= = （逻辑相等）

！= （逻辑不等）

关系操作符的结果为真（1 ）或假（0 ）。如果操作数中有一位为X 或Z ，那么结果为X 。例：

23 > 45

结果为假（0 ），而：

52 < 8'hxFF

结果为x 。

如果操作数长度不同，长度较短的操作数在最重要的位方向（左方）添0 补齐。例如：

'b1000 > = 'b01110

等价于：

'b01000 > = 'b01110

结果为假（0 ）。

在逻辑相等与不等的比较中，只要一个操作数含有x 或z，比较结果为未知 （x），如：

假定：

Data = 'b11x0;

Addr = 'b11x0;

那么：

Data = = Addr 比较结果不定，也就是说值为x 。

4.6.3 逻辑运算符

逻辑运算符有：

&& (逻辑与)

|| (逻辑或)

！(逻辑非)

用法为：（表达式1） 逻辑运算符 （表达式2） ....

这些运算符在逻辑值0（假） 或1（真） 上操作。逻辑运算的结果为0 或1 。例如, 假定：

Crd = 'b0; //0 为假

Dgs = 'b1; //1 为真

那么：

Crd && Dgs 结果为0 (假)

Crd || Dgs 结果为1 (真)

！D g s 结果为0 (假)

逻辑与（&&)的真值表如下：

表1 逻辑与真值表

&&	0（假）	1（真）	X/Z（不定）
0(假）	0	0	x
1（真）	0	1	x
X/Z（不定）	x	x	x

逻辑或的真值表如下：

表2 逻辑或真值表

||	0（假）	1（真）	x/z（不定）
0	0	1	x
1	1	1	1
x/z （不定）	x	1	x

4.6.4 按位逻辑运算符

按位运算符有：

?~（一元非）：（相当于非门运算）

?&（二元与）：（相当于与门运算）

?|（二元或）： （相当于或门运算）

?^（二元异或）：（相当于异或门运算）

?~^, ^~（二元异或非即同或）：（相当于同或门运算）

这些操作符在输入操作数的对应位上按位操作，并产生向量结果。下表显示对于不同按位逻辑运算符按位操作的结果：

& 与	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

| 或	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

^ 异或	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

^~异或非	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

~ 非	0	1	x	z
	1	0	x	x

图7 按位逻辑运算符真值表

例如，假定,

A = 'b0110;

B = 'b0100;

那么：

A | B 结果为0 1 1 0

A & B 结果为0 1 0 0

如果操作数长度不相等, 长度较小的操作数在最左侧添0 补位。例如,

'b0110 ^ 'b10000

与如下式的操作相同:

'b00110 ^ 'b10000

结果为' b 1 0 11 0 。

4.6.5 条件运算符

条件操作符根据条件表达式的值选择表达式，形式如下：

cond_expr ? expr1 : expr2

如果cond_expr 为真(即值为1)，选择expr1 ；如果cond_expr 为假(值为0)，选择expr2 。如果cond_expr 为x 或z ，结果将是按以下逻辑expr1 和expr2 按位操作的值： 0 与0 得0 ， 1 与1 得1 ，其余情况为x 。

如下所示:

wire [2:0] Student = Marks > 18 ? Grade_A : Grade_C;

计算表达式Marks > 18; 如果真, Grade_A 赋值为Student; 如果Marks < =18, Grade_C 赋值为Student 。

4.6.6 连接运算符

连接操作是将小表达式合并形成大表达式的操作。形式如下：

{expr1, expr2, . . .， exprN}

实例如下所示：

wire [7:0] Dbus;

assign Dbus [7:4] = {Dbus [0], Dbus [1], Dbus[2], Dbus[3] } ;

/ /以反转的顺序将低端4 位赋给高端4 位。

assign Dbus = {Dbus [3:0], Dbus [7 : 4] } ;

/ /高4 位与低4 位交换。

由于非定长常数的长度未知, 不允许连接非定长常数。例如, 下列式子非法：

{Dbus,5} / /不允许连接操作非定长常数。

4.7 条件语句

if 语句的语法如下：

if(condition_1)

procedural_statement_1

```
{else if(condition_2)
        procedural_statement_2}
{else
        procedural_statement_3}
```

如果对condition_1 求值的结果为个非零值，那么procedural_statement_1 被执，如果condition_1 的值为0 、x 或z ，那么procedural_statement_1 不执行。如果存在一个else 分支，那么这个分支被执行。以下是一个例子。

```
if(Sum < 60)
begin
        Grade = C;
        Total_C = Total _c + 1;
end
else if(Sum < 75)
begin
        Grade = B;
        Total_B = Total_B + 1;
end
else
begin
        Grade = A;
        Total_A = Total_A + 1;
end
```

注意条件表达式必须总是被括起来，如果使用if - if - else 格式，那么可能会有二义性，如下例所示：

```
if(C l k)
    if(R e s e t)
        Q = 0;
    else
        Q = D;
```

问题是最后一个else 属于哪一个if? 它是属于第一个if 的条件(Clk)还是属于第二个if的条件(Reset)? 这在Verilog HDL 中已通过将else 与最近的没有else 的if 相关联来解决。在这个例子中，else 与内层if 语句相关联。

以下是另一些if 语句的例子。

```
if(Sum < 100)
    Sum = Sum + 10;
if(Nickel_In)
    Deposit = 5;
```

```
elseif (Dime_In)
    Deposit = 10;
else if(Quarter_In)
    Deposit = 25;
else
    Deposit = ERROR;
```

书写建议：

1、条件表达式需用括号括起来。

2、若为if - if 语句，请使用块语句 begin --- end ：

```
if(C l k)
begin
        if(R e s e t)
                Q = 0;
        else
                Q = D;
end
```

以上两点建议是为了使代码更加清晰，防止出错。

3、对if 语句，除非在时序逻辑中，if 语句需要有else 语句。若没有缺省语句，设计将产生一个锁存器，锁存器在ASIC设计中有诸多的弊端（可看同步设计技术所介绍）。如下一例：

```
if （T）
    Q = D;
```

没有else 语句，当T为1（真）时，D 被赋值给Q，当T为0（假）时，因为没有else 语句，电路保持 Q 以前的值，这就形成一个锁存器。

4.8 case 语句

case 语句是一个多路条件分支形式，其语法如下：


```
case(case_expr)
    case_item_expr{ ,case_item_expr} :procedural_statement
. . .
. . .
    [default:procedural_statement]
endcase
```

case 语句首先对条件表达式case_expr 求值，然后依次对各分支项求值并进行比较，第一个与条件表达式值相匹配的分支中的语句被执行。可以在1 个分支中定义多个分支项；这些值不需要互斥。缺省分支覆盖所有没有被分支表达式覆盖的其他分支。

例：

```
case (HEX)
        4'b0001 :       LED = 7'b1111001;       // 1
```

```
        4'b0010 :       LED = 7'b0100100;       // 2
        4'b0011 :       LED = 7'b0110000;       // 3
        4'b0100 :       LED = 7'b0011001;       // 4
        4'b0101 :       LED = 7'b0010010;       // 5
        4'b0110 :       LED = 7'b0000010;       // 6
        4'b0111 :       LED = 7'b1111000;       // 7
        4'b1000 :       LED = 7'b0000000;       // 8
        4'b1001 :       LED = 7'b0010000;       // 9
        4'b1010 :       LED = 7'b0001000;       // A
        4'b1011 :       LED = 7'b0000011;       // B
        4'b1100 :       LED = 7'b1000110;       // C
        4'b1101 :       LED = 7'b0100001;       // D
        4'b1110 :       LED = 7'b0000110;       // E
        4'b1111 :       LED = 7'b0001110;       // F
        default : LED = 7'b1000000;       // 0
    endcase
```

书写建议：

case 的缺省项必须写，防止产生锁存器。

5 结构建模

在3.3.1中，我们已简单介绍了结构化的描述方式，本章节再总结一下。

5.1 模块定义结构

我们已经了解到，一个设计实际上是由一个个module 组成的。一个模块module 的结构如下：

```
module   module_name (port_list) ;
    Declarations_and_Statements
endmodule
```

在结构建模中，描述语句主要是实例化语句，包括对Verilog HDL 内置门如与门（and）异或门（xor）等的例化，如3.3.1节中全加器的xor 门的调用；及对其他器件的调用，这里的器件包括FPGA厂家提供的一些宏单元以及设计者已经有的设计。

在实际应用中，实例化语句主用指后者，对内置门建议不采纳，而用数据流或行为级方式对基本门电路的描述。

端口队列port_list 列出了该模块通过哪些端口与外部模块通信。

5.2 模块端口

模块的端口可以是输入端口、输出端口或双向端口。缺省的端口类型为线网类型（即wire 类型）。输出或输入输出端口能够被重新声明为reg 型。无论是在线网说明还是寄存器说明中，线网或寄存器必须与端口说明中指定的长度相同。下面是一些端口说明实例。

```
module Micro (PC, Instr, NextAddr );
    / / 端口说明
    input [3:1] PC;
    output [1:8] Instr;
    inout [16:1] NextAddr;

    / /重新说明端口类型：
    wire [16:1] NextAddr;     // 该说明是可选的，因为缺省的就是wire类型，但如果指定了，就必须与它的端口说明保持相同长度，这里定义线的位宽16，是总线。
    reg [1:8] Instr;    / /Instr 已被重新说明为reg 类型，因此它能在always 语句或在initial 语句中赋值。
. . .
endmodule
```

5.3 实例化语句

1. 例化语法

一个模块能够在另外一个模块中被引用，这样就建立了描述的层次。模块实例化语句形式如下：

```
module_name instance_name(port_associations) ;
```

信号端口可以通过位置或名称关联；但是关联方式不能够混合使用。端口关联形式如下：

```
port_expr              / /通过位置。
.PortName (port_expr) / /通过名称。
```

例[1]：

```
....
module  and  （C，A，B）；
    input  A，B；
    output   C；
...
    and A1 (T3, A, B );  //实例化时采用位置关联，T3对应输出端口C，A对应A，B对应B。
    and  A2（                  //实例化时采用名字关联，.C是and 器件的端口，其与信号T3相连
            .C（T3），
            .A（A），
            .B（B）
    ）；
```

....

port_expr 可以是以下的任何类型：

1) 标识符（reg 或net ）如 .C（T3），T3为wire型标识符。

2) 位选择，如 .C（D[0]），C端口接到D信号的第0bit 位。

3) 部分选择，如 .Bus （Din[5：4]）。

4) 上述类型的合并，如 .Addr（{ A1，A2[1：0]}。

5) 表达式（只适用于输入端口），如 .A （wire Zire = 0 ）。

建议：在例化的端口映射中请采用名字关联，这样，当被调用的模块管脚改变时不易出错。

2. 悬空端口的处理

在我们的实例化中，可能有些管脚没用到，可在映射中采用空白处理，如：

```
DFF d1 (
    .Q(QS),
   .Qbar ( ),
   .Data (D ) ,
   .Preset ( ),    // 该管脚悬空
    .Clock (CK)
); //名称对应方式。
```

对输入管脚悬空的，则该管脚输入为高阻 Z，输出管脚被悬空的，该输出管脚废弃不用。

3. 不同端口长度的处理

当端口和局部端口表达式的长度不同时，端口通过无符号数的右对齐或截断方式进行匹配。例如：

```
module    Child (Pba, Ppy) ;
input [5:0]  Pba;
output [2:0] Ppy;
. . .
endmodule

module   Top;
    wire [1:2] Bdl;
    wire [2:6] M p r;
    Child C1 (Bdl, Mpr) ;
endmodule
```

在对Child 模块的实例中，Bdl[2]连接到Pba[0]，Bdl[1] 连接到Pba[1]，余下的输入端口Pba[5]、Pba[4]和Pba[3]悬空，因此为高阻态z 。与之相似，Mpr[6]连接到Ppy[0]，Mpr[5]连接到Ppy[1]，Mpr[4] 连接到Ppy[2]。参见下图：

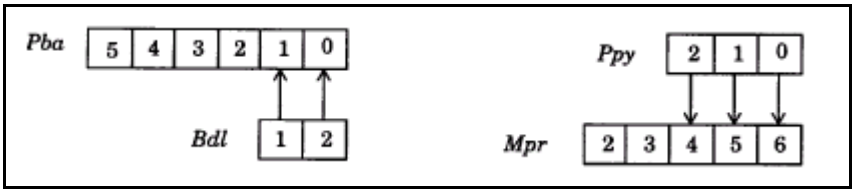

图8 端口匹配

5.4 结构化建模具体实例

对一个数字系统的设计，我们采用的是自顶向下的设计方式。可把系统划分成几个功能模块，每个功能模块再划分成下一层的子模块。每个模块的设计对应一个module ，一个module 设计成一个verilog HDL 程序文件。因此，对一个系统的顶层模块，我们采用结构化的设计，即顶层模块分别调用了各个功能模块。下面以一个实例（一个频率计数器系统）说明如何用HDL进行系统设计。

在该系统中，我们划分成如下三个部分：2输入与门模块，LED显示模块，4位计数器模块。系统的层次描述如下：

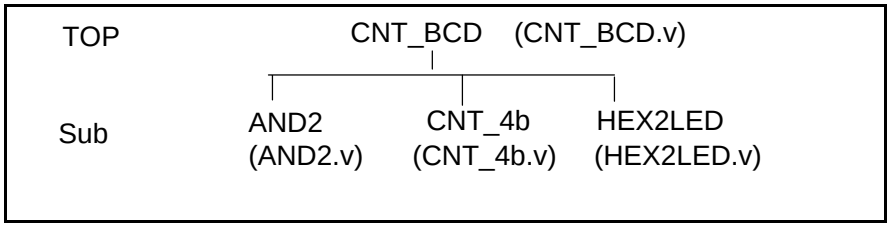

图9 系统层次描述

顶层模块CNT_BCD，文件名CNT_BCD.v，该模块调用了低层模块 AND2、CNT_4b和HEX2LED 。

系统的电路结构图如下：

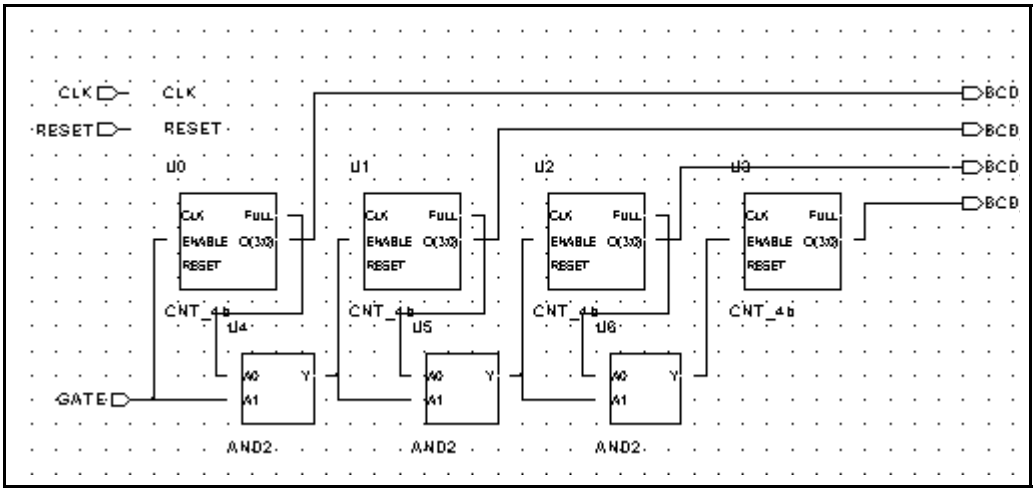

图10 系统电路框图

顶层模块CNT_BCD对应的设计文件 CNT_BCD.v 内容为：

```
module CNT_BCD (BCD_A,BCD_B,BCD_C,BCD_D,CLK,GATE,RESET) ;
    // ------------ Port declarations --------- //
    input CLK;
    input GATE;
    input RESET;
    output [3:0] BCD_A;
    output [3:0] BCD_B;
    output [3:0] BCD_C;
    output [3:0] BCD_D;

    wire CLK;
    wire GATE;
    wire RESET;
    wire [3:0] BCD_A;
    wire [3:0] BCD_B;
    wire [3:0] BCD_C;
    wire [3:0] BCD_D;

    // ------------ Signal declarations -------- //
    wire NET104;
    wire NET116;
    wire NET124;
    wire NET132;
    wire NET80;
    wire NET92;
    // -------- Component instantiations -------//
    CNT_4b U0(
            .CLK(CLK),
            .ENABLE(GATE),
            .FULL(NET80),
            .Q(BCD_A),
            .RESET(RESET)
    );

    CNT_4b U1(
            .CLK(CLK),
            .ENABLE(NET116),
```

```
        .FULL(NET92),
        .Q(BCD_B),
        .RESET(RESET)
    );

    CNT_4b U2(
        .CLK(CLK),
        .ENABLE(NET124),
        .FULL(NET104),
        .Q(BCD_C),
        .RESET(RESET)
    );

    CNT_4b U3(
        .CLK(CLK),
        .ENABLE(NET132),
        .Q(BCD_D),
        .RESET(RESET)
    );

AND2 U4(
    .A0(NET80),
    .A1(GATE),
    .Y(NET116)
    );
AND2 U5(
    .A0(NET92),
    .A1(NET116),
    .Y(NET124)
    );
AND2 U6(
    .A0(NET104),
    .A1(NET124),
    .Y(NET132)
    );

endmodule
```

注意：这里的AND2是为了举例说明，在实际设计中，对门级不要重新设计成一个模块，同时对涉及保留字的（不管大小写）相类似的标识符最好不用。

6 数据流建模

在3.3.2 节中，我们已经初步了解到数据流描述方式，本节对数据流的建模方式进一步讨论，主要讲述连续赋值语句、阻塞赋值语句、非阻塞赋值语句，并针对一个系统设计 频率计数器 的实例进行讲解。

6.1 连续赋值语句

数据流的描述是采用连续赋值语句(assign)语句来实现的。语法如下：

assign net_type = 表达式；

连续赋值语句用于组合逻辑的建模。 等式左边是wire 类型的变量。等式右边可以是常量、由运算符如逻辑运算符、算术运算符参与的表达。如下几个实例：

```
wire [3:0]  Z, Preset, Clear;          //线网说明
assign  Z =  Preset & Clear;           //连续赋值语句


wire  Cout, C i n ;
wire [3:0]  Sum, A, B;
. . .
assign {Cout, Sum} = A + B + Cin;
assign  Mux  =  (S = = 3)? D : 'bz;
```

注意如下几个方面：

1、连续赋值语句的执行是：只要右边表达式任一个变量有变化，表达式立即被计算，计算的结果立即赋给左边信号。

2、连续赋值语句之间是并行语句，因此与位置顺序无关。

6.2 阻塞赋值语句

“=”用于阻塞的赋值，凡是在组合逻辑（如在assign 语句中）赋值的请用阻塞赋值。更深的知识以后再讲。

6.3 数据流建模具体实例

以上面的 频率计数器为例，其中的AND2模块我们用数据流来建模。

AND2模块对应文件AND2.v 的内容如下：

```
module AND2 (A0, A1, Y);
    input A0;
    input A1;
    output Y;
```

```
    wire A0;
    wire A1;
    wire Y;

    // add your code here
    assign Y = A0 & A1;
endmodule
```

7 行为建模

在3.3.3节中，我们已经对行为描述方式有个概念，这里对行为建模进一步的描述，并通过 一个系统设计 频率计数器 加以巩固。

7.1 简介

行为建模方式是通过对设计的行为的描述来实现对设计建模，一般是指用过程赋值语句（initial 语句和always 语句 ）来设计的称为行为建模。

7.2 顺序语句块

语句块块提供将两条或更多条语句组合成语法结构上相当于一条语句的机制。这里主要讲 Verilog HDL 的 顺序语句块(begin . . . end) ： 语句块中的语句按给定次序顺序执行。

顺序语句块中的语句按顺序方式执行。每条语句中的时延值与其前面的语句执行的模拟时间相关。一旦顺序语句块执行结束，跟随顺序语句块过程的下一条语句继续执行。顺序语句块的语法如下：

```
begin
    [ :block_id{declarations} ]
    procedural_statement ( s )
end
```

例如：

```
/ / 产生波形:
begin
    #2 Stream = 1;
    #5 Stream = 0;
    #3 Stream = 1;
    #4 Stream = 0;
    #2 Stream = 1;
    #5 Stream = 0;
end
```

假定顺序语句块在第1 0 个时间单位开始执行。两个时间单位后第1 条语句执行，即第12 个时间单位。此执行完成后，下1 条语句在第17 个时间单位执行(延迟5 个时间单位)。然后下1 条语句在第2 0 个时间单位执行，以此类推。该顺序语句块执行过程中产生的波形如图：

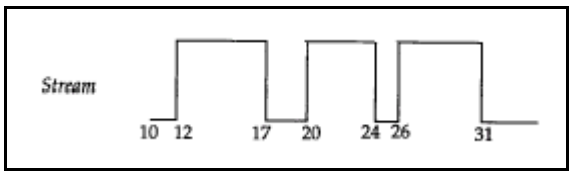

图11 顺序语句块中积累延时

7.3 过程赋值语句

Verilog HDL 中提供两种过程赋值语句 initial 和 always 语句，用这两种语句来实现行为的建模。这两种语句之间的执行是并行的，即语句的执行与位置顺序无关。这两种语句通常与语句块（begin ....end）相结合，则语句块中的执行是按顺序执行的。

1. initial 语句

initial 语句只执行一次，即在设计被开始模拟执行时开始（0时刻）。通常只用在对设计进行仿真的测试文件中，用于对一些信号进行初始化和产生特定的信号波形。

语法如下：（大家只要先有个概念就可以）

```
initial
        [timing_control] procedural_statement
procedural_statement 是下列语句之一：
        procedural_assignment (blocking or non-blocking ) / /阻塞或非阻塞性过程赋值语句/ /
        procedural_continuous_assignment
        conditional_statement
        case_statement
        loop_statement
        wait_statement
        disable_statement
        event_trigger
        task_enable (user or system)
```

事例如上产生一个信号波形：

```
initial
begin
    #2 Stream = 1;
    #5 Stream = 0;
    #3 Stream = 1;
    #4 Stream = 0;
    #2 Stream = 1;
    #5 Stream = 0;
end
```

2. always 语句

always 语句与initial 语句相反，是被重复执行，执行机制是通过对一个称为敏感变量表的事件驱动来实现的，下面会具体讲到。always 语句可实现组合逻辑或时序逻辑的建模。

例[1]：

```
initial
    Clk  = 0 ;

always
    #5  Clk  =  ~Clk;
```

因为always 语句是重复执行的，因此，Clk 是初始值为0 的，周期为10 的方波。

例[2] D 触发器

```
always @ ( posedge  Clk  or  posedge Rst )
begin
    if  Rst
            Q <=  ‘ b 0;
    else
            Q <= D;
```

上面括号内的内容称为敏感变量，即整个always 语句当敏感变量有变化时被执行，否则不执行。因此，当Rst 为1 时，Q被复位，在时钟上升沿时，D被采样到Q。 有@ 的用来描述一个时序器件。

例[3] 2 选一的分配器

```
always @( sel ， a ， b）
    C =  sel ? a ： b;
```

这里的sel ， a，b 同样称为敏感变量，当三者之一有变化时，always 被执行，当sel 为 1 ，C被赋值为a ，否则为b 。描述的是一个组合逻辑 mux 器件。

注意以下几点：

1、对组合逻辑的always 语句，敏感变量必须写全，敏感变量是指等式右边出现的所有标识符如上的a，b和条件表达式中出现的所以标识符 如上例3的sel。

2、对组合逻辑器件的赋值采用阻塞赋值 “=”

3、时序逻辑器件的赋值语句采用非阻塞赋值 “<=”，如上的 Q 〈= D；

7.4 行为建模具体实例

以上面的 频率计数器 为例，其中的 HEX2LED 和 CNT_4b 模块采用行为建模。

CNT_4b 模块对应的文件 CNT_4b.v 的内容如下：

```
module CNT_4b (CLK, ENABLE, RESET, FULL, Q);
    input CLK;
    input ENABLE;
    input RESET;
    output FULL;
    output [3:0] Q;

    wire CLK;
    wire ENABLE;
    wire RESET;
    wire FULL;
    wire [3:0] Q;

    // add your declarations here
    reg [3:0] Qint;

    always @(posedge RESET or posedge CLK)
    begin
        if (RESET)
            Qint = 4'b0000;
        else if (ENABLE)
        begin
            if (Qint == 9)
                Qint = 4'b0000;
            else
                Qint = Qint + 4'b1;
        end
    end
    assign Q = Qint;
    assign FULL = (Qint == 9) ? 1'b1 : 1'b0;

endmodule
```

该模块实现一个模10 的计数器。

HEX2LED 模块对应的文件HEX2LED.v 的内容为：

```
module HEX2LED (HEX, LED);
```

```
input [3:0] HEX;
output [6:0] LED;

wire [3:0] HEX;
reg [6:0] LED;

// add your declarations here
always @(HEX)
 begin
        case (HEX)
                4'b0001 :       LED = 7'b1111001;       // 1
                4'b0010 :       LED = 7'b0100100;       // 2
                4'b0011 :       LED = 7'b0110000;       // 3
                4'b0100 :       LED = 7'b0011001;       // 4
                4'b0101 :       LED = 7'b0010010;       // 5
                4'b0110 :       LED = 7'b0000010;       // 6
                4'b0111 :       LED = 7'b1111000;       // 7
                4'b1000 :       LED = 7'b0000000;       // 8
                4'b1001 :       LED = 7'b0010000;       // 9
                4'b1010 :       LED = 7'b0001000;       // A
                4'b1011 :       LED = 7'b0000011;       // B
                4'b1100 :       LED = 7'b1000110;       // C
                4'b1101 :       LED = 7'b0100001;       // D
                4'b1110 :       LED = 7'b0000110;       // E
                4'b1111 :       LED = 7'b0001110;       // F
                default : LED = 7'b1000000;       // 0
        endcase
end

endmodule
```

该模块实现模10 计数器的值到 7段码的译码。

至此，整个频率计数器的系统设计由4个模块（4个文件）我们已设计完毕。这就是HDL 的自顶向下的设计方式和HDL的多种建模方式的应用。

8 其他方面

以上的介绍仅为入门级，为了进行更多的设计，请大家在入门之后继续看一些书，主要了解一下常用的编译指令 `define 、`include ， 了解任务task的使用方法和状态机的设计。

9 习题

1、数字电路设计有那几种层次，可否分别举个例子？

2、能否回忆一下在学校中用原理图进行设计的方法或在数字电路课中的进行数字电路设计的方法？能否用HDL语言简单描述一下4 位宽的加法器？

3、当前两种硬件描述语言是什么？

4、以CPU为例，能否画出Top-Down的树状图？

5、假设一D触发器组的器件 Reg8，输入信号 Din，输出信号Qout，位宽8位，时钟信号Clk，异步复位信号Rst，用于实现对8位数据总线的寄存，请描述出module 语句，并画出电路的示意图，D触发器图标如下：

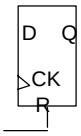

图12 D触发器symbol

6、HDL有哪几种建模方式？

7、结构化建模方式主要的语句是什么？.A（A）的两个A具体含义是什么？

8、数据流的建模方式采用什么语句？语法是什么？

9、行为建模方式采用什么语句？信号的数据类型必须是什么？

10、能简单介绍一下串行语句和并行语句的概念吗？

11、Verilog HDL中规定了哪四种基本值类型？

12、写出下面整型数值的二进制表示： 5 'O37、4 'D2、8 'h 2A、7'Hx、5'H7F

13、有哪几种主要的数据类型？可否说明它的简单用法？

14、initial 语句和always语句的区别是什么？可否用语句产生一个周期20ns 的方波，初始值为1；产生一个复位信号Rst ，0到40ns 为1，之后保持为0。

10 附录A Verilog 保留字

always	and	assign	begin	buf	buf if0	bufif1	case	casex	casez	cmos
deassign	default	defparam		disable	edge	else	end	endcase		endmodule
endfunction		endprimitive		endspecify		endtable		endtask		event
for	force	forever		fork	function	highz0		highz1	if	ifnone
initial	inout	input	integer	join	large	macrmodule		medium		module
nand	negedge		nmos	nor	not	notif0		notif1	or	output
parameter	pmos	posedge		primitive		pull0	pull1	pullup	pulldown	
rcmos	real	realtime		reg	release	repeat	rnmos	rpmos	rtran	rtranif0

rtranif1	scalared		small	specify	specparam		strong0	strong1	supply0	supply1
table	task	time	trantranif0		tranif1	tri	tri0	tri1	triand	trior
trireg	vectored		wait	wand	weak0	weak1	while	wire	wor	xnor	xor