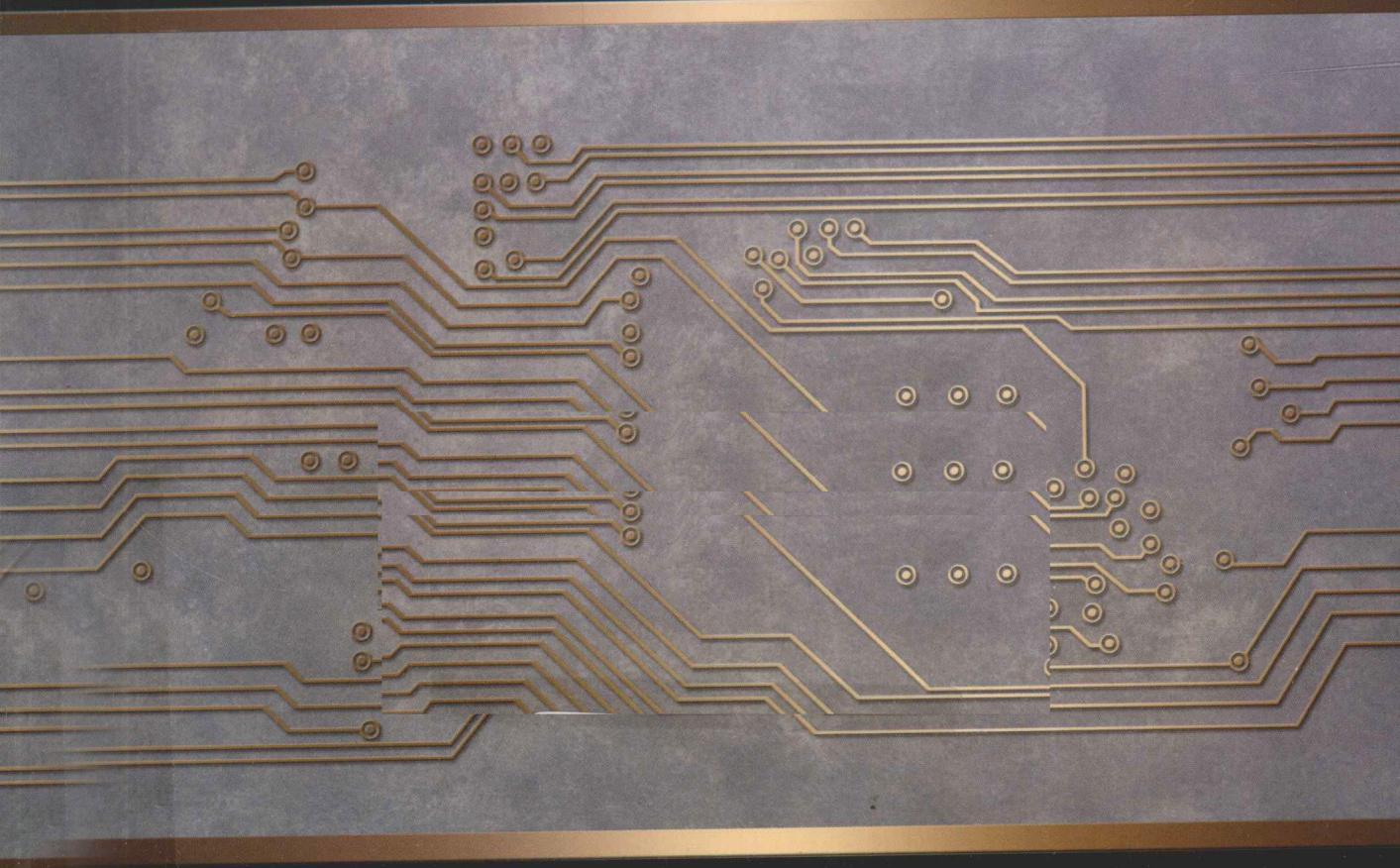
新编电气与电子信息类规划教材

Verilog HDL

数字系统设计及实践

刘睿强　童贞理　尹洪剑　编著



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

新编电气与电子信息类规划教材

Verilog HDL
数字系统设计及实践

刘睿强　童贞理　尹洪剑　编著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内 容 简 介

本书介绍硬件描述语言 Verilog HDL 及电路设计方法。共 11 章，主要内容包括：Verilog HDL 层次化设计、Verilog HDL 基本语法、Verilog HDL 行为描述、组合逻辑建模、时序逻辑建模、行为级仿真模型建模、各层次 Verilog HDL 描述形式与电路建模、任务和函数、编译预处理、Verilog HDL 设计与综合中的陷阱、异步设计与同步设计的时序分析。本书配套实验，提供电子课件和习题参考答案。

本书可作为高等学校电子信息类相关课程的教材，也可供相关工程技术人员学习参考。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容
版权所有 • 侵权必究

图书在版编目(CIP)数据

Verilog HDL 数字系统设计及实践/刘睿强，童贞理，尹洪剑编著. —北京：电子工业出版社，2011.1
(新编电气与电子信息类规划教材)
ISBN 978-7-121-12021-3

I. ①V… II. ①刘…②童…③尹… III. ①硬件描述语言，Verilog—程序设计—高等学校—教材 IV. ①TP312

中国版本图书馆 CIP 数据核字(2011)第 200272 号

策划编辑：王羽佳
责任编辑：王羽佳　　特约编辑：王　崧
印　　刷：北京市铁成印刷厂
装　　订：
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×1092　1/16　印张：14　字数：358.4 千字
印　　次：2011 年 1 月第 1 次印刷
印　　数：4000 册　　定价：29.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010) 88258888。

前　言

随着超大规模集成电路(VLSI)技术和计算机辅助设计(CAD)技术的发展，越来越多的系统设计开始基于现场可编程门阵列(FPGA)。采用 FPGA 器件可以将原来的电路板级产品集成为芯片级产品，从而降低功耗，提高系统的可靠性。今天，FPGA 正在以惊人的速度发展。一个芯片可以包含数百万个门，而且越来越多的 FPGA 内可以嵌入各种档次的 CPU，出现了 SOPC 系统，它代表着嵌入式系统发展的新方向。芯片设计工作的承担者正由传统的专业芯片设计机构向个人转变，显然，谁能早一步掌握这门技术，谁就能在激烈的竞争中处于更加有利的位置。而 Verilog HDL 硬件描述语言正是掌握这门技术的必备基础之一，Verilog HDL 硬件描述语言是一种以文本形式来描述数字系统硬件结构和行为的语言，是目前世界上最流行的一种硬件描述语言，用它可以表示逻辑电路图、逻辑表达式，还可以表示数字逻辑系统所完成的逻辑功能。

本书共分 11 章，内容分别为：第 1 章 Verilog HDL 层次化设计；第 2 章 Verilog HDL 基本语法；第 3 章 Verilog HDL 行为描述；第 4 章 组合逻辑建模；第 5 章 时序逻辑建模；第 6 章 行为级仿真模型建模；第 7 章 各层次 Verilog HDL 描述形式与电路建模；第 8 章 任务和函数；第 9 章 编译预处理；第 10 章 Verilog HDL 设计与综合中的陷阱；第 11 章 异步设计与同步设计的时序分析。

通过学习本书，读者可以掌握硬件描述语言 Verilog HDL 的语法结构，掌握组合逻辑和时序逻辑电路的设计方法，掌握 Modelsim 和 Debussy 仿真调试工具的应用及 Synplify 综合工具的应用，了解 task 和 function 等的高级语法，深入理解“自顶向下”的设计思想，具备设计可综合 RTL 的能力。本课程配套的实验提供了创建可综合 RTL 代码的实践基础，其中涵盖了设计流程的方方面面。实验着重于让学生写出能最优地推断出高性能可靠电路的代码。

本书可作为高等学校电子信息工程技术、应用电子技术、通信工程等专业学生的教材，同时可供相关工程技术人员学习参考。本课程的学习应在修完电路分析、数字电子技术、模拟电子技术等课程之后进行。本书内容丰富，图文并茂，文字简练流畅，注重实际操作，阐述方式新颖，具有较强的实用性和可操作性。

本书提供配套实验、电子课件和习题解答，请登录华信教育资源网 http://www.hxedu.com.cn 注册下载。

本书第 1~9 章由刘睿强主笔，第 10 章由童贞理主笔，第 11 章由尹洪剑主笔，参与本书编写的人员还有王荣辉、张林生、毛朝庆、许磊、彭华、冯静等。企业专家王挺等提出了诸多修改意见，电子工业出版社的工作人员为本书的出版付出了辛勤的劳动。所有这些，都是本书得以顺利出版的保障，在此一并向他们表示衷心的感谢！

当然，现代 EDA 技术飞速发展，相应的教学内容和教学方法也在不断改进，其中一定还有许多问题值得深入探讨。由于编者水平有限，时间仓促，书中难免有缺点和不足之处，恳请读者批评指正！

作　者

目　录

试读结束，需要全本PDF请购买 www.ertongbook.com

第 1 章　Verilog HDL 层次化设计

【知识目标】

(1) 了解 Verilog HDL 设计中的模块的概念;

(2) 了解层次化设计的概念;

(3) 了解 Testbench 的概念。

【技能目标】

(1) 能够描述一个完整的简单模块;

(2) 能够通过模块实例化完成层次化的设计。

【重点难点】

(1) 模块实例化的理解;

(2) Testbench 的概念。

【参考学时】

4 学时。

本章将从一个简单的 Verilog HDL 设计实例开始，从 Verilog HDL 层次化设计方法出发，展开学习 Verilog HDL 语言的旅程。

1.1　一个简单的例子——4 位全加器的设计

首先来看一个简单的 Verilog HDL 设计的例子。

【例 1.1】 利用 Verilog HDL 语言和层次化设计方法来设计一个 4 位全加器电路。

```
// example_1_1: full adder
// 4 位全加器由 4 个 1 位全加器构成。
module fadder_4
    (
        i_A,
        i_B,
        i_Cin,
        o_S,
        o_Cout
    );
    input  [3:0] i_A, i_B;          // 输入端口 i_A, i_B
    input  i_Cin;                   // 输入端口 i_Cin
    output [3:0] o_S;               // 输出端口 o_S
    output  o_Cout;                 // 输出端口 o_Cout
    wire    Cout_1, Cout_2, Cout_3; // wire 型数据 Cout_1, Cout_2, Cout_3
    // 实例化 4 个 1 位全加器
    fadder_1 u_fadder_1_1
```

```
    (
        .i_A(i_A[0]),
        .i_B(i_B[0]),
        .i_Cin(i_Cin),
        .o_S(o_S[0]),
        .o_Cout(Cout_1)
    );
    fadder_1 u_fadder_1_2
    (
        .i_A(i_A[1]),
        .i_B(i_B[1]),
        .i_Cin(Cout_1),
        .o_S(o_S[1]),
        .o_Cout(Cout_2)
    );
    fadder_1 u_fadder_1_3
    (
        .i_A(i_A[2]),
        .i_B(i_B[2]),
        .i_Cin(Cout_2),
        .o_S(o_S[2]),
        .o_Cout(Cout_3)
    );
    fadder_1 u_fadder_1_4
    (
        .i_A(i_A[3]),
        .i_B(i_B[3]),
        .i_Cin(Cout_3),
        .o_S(o_S[3]),
        .o_Cout(o_Cout)
    );
endmodule
// 定义1个1位全加器
module fadder_1
    (
        i_A,
        i_B,
        i_Cin,
        o_S,
        o_Cout
    );
    input  i_A, i_B;          //输入端口 i_A, i_B
    input  i_Cin;             //输入端口 i_Cin
    output o_S, o_Cout;       //输出端口 o_S, o_Cout
    // 计算结果值: o_S = i_A⊕i_B⊕i_Cin
    assign  o_S = i_A ^ i_B ^ i_Cin;
    // 计算进位值: o_Cout = (i_A⊕i_B)i_Cin + (i_A)(i_B)
    assign  o_Cout = (i_A ^ i_B) & i_Cin | i_A & i_B;

endmodule
```

该例描述了 1 个 4 位二进制全加器，模块名为 fadder_4。而此四位全加器是由 4 个 1 位全加器串联而成的。例子的后半部分描述的是 1 位全加器子模块，模块名为 fadder_1。fadder_1 中所表示的逻辑为 o_S = i_A⊕i_B⊕i_Cin，o_Cout = (i_A⊕i_B)i_Cin + (i_A)(i_B)，其中 o_S 为全加器的和，o_Cout 为全加器的进位。

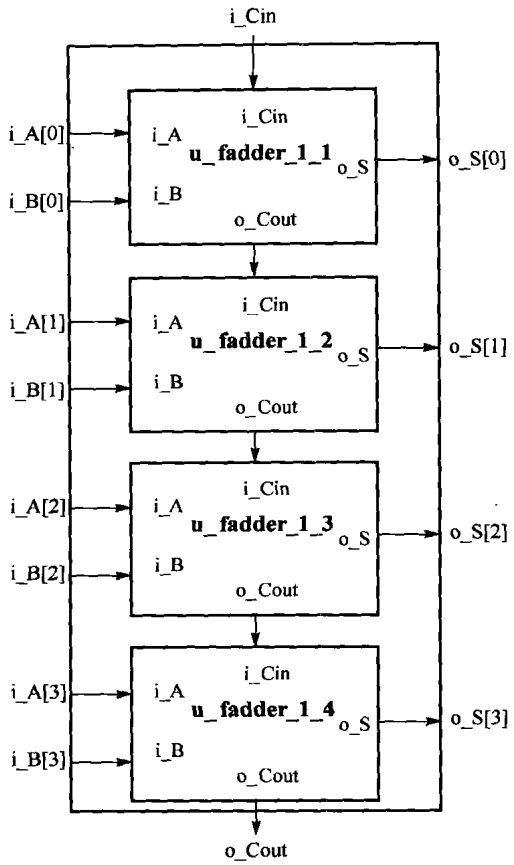


图 1.1　4 位二进制全加器结构图

提示：注释符号

*在 Verilog HDL 语言中，可以使用符号//进行单行注释，也可以使用/*和*/进行多行注释。Verilog HDL 的注释方式与 C++和 Java 语言的注释方式相同。*

利用 Verilog HDL 进行层次化设计，其描述的电路结构与真实的电路结构很相近，因此通常可以直接画出电路的结构图。用结构图来表示这个四位全加器，如图 1.1 所示。

1.2　模块和端口

由例 1.1 可以看出，“模块”是 Verilog HDL 设计中的一个基本组成单元，一个设计是由一个或多个模块组成的。一个模块的代码主要由下面几个部分构成：模块名定义、端口描述和内部功能逻辑描述。一个模块通常就是一个电路单元器件，如图 1.2 所示。

```
module exap
    (A,B,C);
    input A,B;
   output C;
    ...
    //逻辑功能描述
endmodule
```

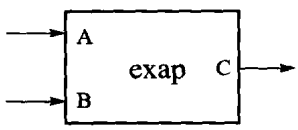


图 1.2　模块和端口

图 1.2 中的代码定义了一个名为 exap 的电路器件。代码中用关键字 module 定义了模块的名字，然后用括号列出了该模块的端口。在模块名定义的后面，分别用 input 和 output 关键字指定端口的方向。端口定义完成后，给出描述该模块功能的代码，最后用关键字 endmodule 来结束该模块的描述。上述代码描述的电路，实际上对应于实际硬件中的一个功能模块，该模块有 2 个输入端口 A 和 B，以及 1 个输出端口 C。通过对该模块的端口进行连线，可以将这个模块与其他模块连接在一起，形成功能更复杂的电路。

1.2.1 模块定义

定义模块要使用关键字 module 和 endmodule，其语法格式为

```
module 模块名 (端口声明列表);
端口定义
...
endmodule
```

模块名在一个设计中必须是唯一的，用以区别其他模块。该模块的描述必须全部写在这两个关键字之间，不允许模块中嵌套定义其他模块。

模块中的逻辑功能描述主要由 5 部分组成：变量声明、数据流描述语句、门级实例化描述语句、行为描述语句及任务与函数。上述各个部分的描述可以以任意顺序出现，但要注意的是，虽然变量的声明可以出现在任何位置，但必须在该变量使用之前进行声明。关于描述模块功能的语句的介绍，将在本书后面的章节中陆续给出。

1.2.2 端口定义

在 Verilog HDL 中定义端口有两种风格：普通风格和 ANSI C 风格。

用普通风格定义模块的端口，首先在模块名后把所有的输入/输出端口列举出来(如果一个模块和外部没有任何连接关系，则可以没有端口列表，直接打空括号即可)，如

```
module    模块名 (端口名 1，端口名 2，...);
```

接下来需要对输入/输出端口进行定义，如

```
input     [位宽-1: 0]  端口名 1，端口名 2;
output    [位宽-1: 0]  端口名 3;
inout     [位宽-1: 0]  端口名 4;
```

定义端口时，使用关键字 input、output 和 inout 来分别指定该端口的方向为输入、输出或双向。之后是用中括号指定位宽的可选语句，再后是端口名。进行端口定义的端口必须首先在端口声明列表中出现，否则将被视为语法错误。端口定义的一行可以定义多个输入/输出方向和位宽均相同的端口，多个端口的端口名用逗号隔开。

用普通风格定义端口时需要对端口分别进行声明和定义，使用起来较为麻烦，因此 Verilog HDL 语言还提供了另一种称为 ANSI C 风格的端口定义方式。ANSI C 风格的端口定义允许将端口的声明和定义合写在一起，并且都出现在模块名之后的括号中，其格式如下：

```
module  模块名
    (   input     [位宽-1: 0]  端口名 1，端口名 2;
        output    [位宽-1: 0]  端口名 3;
        inout     [位宽-1: 0]  端口名 4;
    );
```

可以看出，利用 ANSI C 风格，可以一次性地完成模块名和端口的定义，使得代码更为紧凑，减少了出错的概率，因此推荐使用这种风格进行端口定义。本书中给出的所有例子都采用 ANSI C 风格来定义端口。

【例 1.2】 利用 ANSI C 风格来定义例 1.1 中加法器模块的端口。

```
// example_1_2: full adder
```

试读结束，需要全本PDF请购买 www.ertongbook.com

```
// 利用 ANSI C 风格进行全加器模块的端口定义
module fadder_4
    (   input  [3:0] i_A, i_B,        // 输入端口 i_A, i_B
        input   i_Cin,                // 输入端口 i_Cin
        output [3:0] o_S,             // 输出端口 o_S
        output  o_Cout                // 输出端口 o_Cout
    );
// 全加器功能描述代码，与例 1.1 相同
// ...
endmodule
// 定义一个 1 位全加器
module fadder_1
    (   input  i_A, i_B,              //输入端口 i_A, i_B
        input  i_Cin,                 //输入端口 i_Cin
        output o_S, o_Cout            //输出端口 o_S, o_Cout
    );
// 一位加法器功能描述代码，与例 1.1 相同
// ...
endmodule
```

在定义端口时，各个端口的定义顺序没有任何限制，可先定义输出端口，再定义输入端口。在用普通风格进行端口定义时，端口声明列表和端口定义的排列顺序也可以不同。

1.2.3　模块实例化

在例 1.1 中提到了模块的实例化。模块定义中是不允许嵌套定义模块的，模块之间的相互调用只能通过实例化来实现。

定义好的模块可以视为一个模板，使用该模板可以创建一个对应的实际对象。当一个模块被调用时，Verilog HDL 语言可以根据模板创建一个唯一的模块对象，每个对象都有自己的名字、参数、端口连接关系等。使用定义好的模板创建对象的过程称为实例化(Instantiation)，创建的对象称为实例(Instance)。每个实例必须有唯一的名字。图1.3 所示为对一位加法器进行多次实例化来构建四位加法器的示意图。

通过多次实例化相同的模块，实际上在电路中设计了 4 个相同的 1 位加法器，只是它们在电路中的名字和连接关系各不相同。

对已定义好的模块进行实例化引用的语法格式如下：

```
模块名　实例名 (端口连接关系列表);
```

在实例化时，可以用两种方式书写端口连接关系列表。

第一种方式是命名端口连接方式，其语法格式为

```
模块名　实例名 (.端口名(连接线 1), .端口名 2(连接线 2), ...);
```

用命名端口的方式进行连接，每个连接关系用一个点开头，然后是需要进行连接的模块的端口名，端口名后面在括号中指定该端口需要连接到当前层次模块中的哪个信号。例 1.1 中实例化一位加法器模块的方式使用的就是命名端口连接，例如

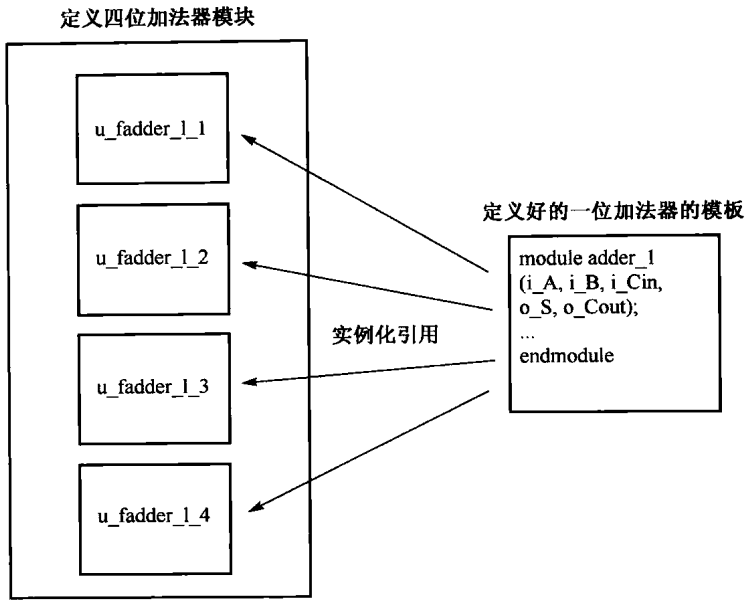


图 1.3 模块实例化示意图

```
fadder_1 u_fadder_1_1
(   .i_A(i_A[0]),
    .i_B(i_B[0]),
    .i_Cin(i_Cin),
    .o_S(o_S[0]),
    .o_Cout(Cout_1)
);
```

其中，i_A，i_B，i_Cin，o_S，o_Cout 都是在模块 fadder_1 中定义过的端口，i_A[0]是连接到端口 i_A 的信号。因此，用命名端口的方式书写端口列表，其实就是将端口名和需要连接到的信号名成对地写在一起。由于在端口连接列表中明确指定了端口的连接关系，因此各个端口在连接列表中的顺序可以随意交换，而不影响实际的连接结果。例如，上面对模块 fadder_1 的实例化也可以这样书写：

```
fadder_1 u_fadder_1_1
(   .i_B(i_B[0]),
    .i_Cin(i_Cin),
    .o_S(o_S[0]),
    .o_Cout(Cout_1),
    .i_A(i_A[0])
);
```

即将端口 i_A 的连接关系写在最后面。这样写的效果与之前写法的效果相同，端口的连接关系并未改变。

若需要某个端口不连接，则在连接列表中不列出该端口即可。例如，我们不希望输出端口 o_S 在实例化时连接到任何信号，则可以这样书写：

```
fadder_1 u_fadder_1_1
(   .i_B(i_B[0]),
    .i_Cin(i_Cin),
```

```
    .o_Cout(Cout_1),
    .i_A(i_A[0])
);
```

但这种在端口连接列表中忽略某个端口连接关系的写法通常会在仿真工具编译时报警，因此，一种更好的方式是这样书写：

```
fadder_1 u_fadder_1_1
(   .i_B(i_B[0]),
    .i_Cin(i_Cin),
    .o_S(),                                    // 端口 o_S 悬空
    .o_Cout(Cout_1),
    .i_A(i_A[0])
);
```

即在端口连接关系列表中写出 o_S 端口，但是不指定它所连接的信号，而是打一个空括号。在实际使用 Verilog HDL 进行设计时，应坚持使用这样的方式来指定不进行连接的端口。当他人检查这段代码时，可以得到明确的信息，即设计者是有意不对端口 o_S 进行连接的，而不是在实例化模块时忘记了写该端口。

第二种进行实例化端口连接的书写方式是顺序端口连接方式，其语法形式为

```
模块名　实例名(连接线名 1，连接线名 2，...);
```

用顺序端口连接方式来指定连接关系时，不需要给出模块的端口名，只需要按一定的顺序列出需要连接到的信号名即可。Verilog HDL 语言将根据端口在模块声明列表中的声明顺序，把信号和模块端口连接起来。排列在连接关系列表第一位的信号，将连接到模块端口声明列表中排列第一位的端口。例如，用顺序连接方式实例化例 1.1 中的一位加法器模块：

```
fadder_1 add_1 (i_A[0], i_B[0], i_Cin, o_S[0], o_Cout);
// ...
// 定义一个 1 位全加器
module fadder_1
    (   i_A,
        i_B,
        i_Cin,
        o_S,
        o_Cout
    )
// ...
```

Verilog HDL 语句将按照模块 fadder_1 定义中端口的声明顺序，把端口连接列表中的各个信号与模块连接起来。因此，对于上述代码，i_A[0]信号将连接到 i_A 端口，i_B[0]信号将连接到 i_B 端口，依次类推。若利用 ANSI C 风格定义模块的端口，情况也类似，如

```
fadder_1 add_1 (i_A[0], i_B[0], i_Cin, o_S[0], o_Cout);
// ...
// 定义一个 1 位全加器
module fadder_1
    (   input  i_A, i_B,                       //输入端口 i_A, i_B
        input  i_Cin,                          //输入端口 i_Cin
```

```
        output o_S, o_Cout                    //输出端口 o_S, o_Cout
    );
// ...
```

使用顺序端口连接方式进行实例化时，不能随意改变端口连接列表中信号的排列顺序，否则会导致错误的连接关系，比如，若写成

```
// 错误的一位加法器连接关系
fadder_1 add_1 (i_A[0], i_Cin, i_B[0], o_S[0], o_Cout);
// ...
// 定义一个 1 位全加器
module fadder_1
    (   i_A,
        i_B,
        i_Cin,
        o_S,
        o_Cout
    )
// ...
```

则这时加法器的连接关系就被改变了。i_A[0]依然与 i_A 端口相连，但是由于调换了 i_Cin 和 i_B[0]信号在端口连接列表中的顺序，这时 i_B 端口和 i_Cin 信号连接在一起，而 i_Cin 端口则和 i_B[0]信号连接在一起，造成了连接错误。因此，使用顺序端口连接方式进行实例化时，需要十分小心地安排信号在端口连接列表中的顺序，以免造成连接错误。在使用 Verilog HDL 时，应坚持使用命名端口的方式进行实例化和端口连接，以减少出现设计错误的概率。

提示：顺序端口连接参考的是端口声明顺序而非定义顺序

用顺序端口连接方式进行实例化时，端口的连接关系参考的是模块定义中端口的声明顺序，而非定义顺序，例如

```
fadder_1 add_1 (i_A[0], i_B[0], i_Cin, o_S[0], o_Cout);
// ...
// 定义一个 1 位全加器
module fadder_1
    (   // 端口声明
        i_A,
        i_B,
        i_Cin,
        o_S,
        o_Cout
);
// 端口定义
    output o_S, o_Cout;
    input  i_A, i_B;
    input  i_Cin;
// ...
```

虽然输出端口 o_S 在端口定义时排在最前面，但是信号 i_A[0]还是连接在 i_A 端口，因为 i_A 端口出现在端口声明列表的第一位。

在利用顺序端口连接方式进行实例化时，若希望某个端口不做连接，则可在端口连接列表中留出其位置，但不指定任何要连接的信号，如

```
fadder_1 add_1 (i_A[0], i_B[0], i_Cin, , o_Cout);
```

上述实例化代码同样使得输出端口 o_S 不连接到任何信号。需要特别注意的是，要在顺序端口连接方式中使某个端口悬空，不能像命名端口连接方式那样直接在连接列表中忽略该端口，而要在连接列表中预留该端口的位置，但不指定任何连接信号。例如，下面的实例化代码产生错误的连接关系：

```
// 错误的一位加法器连接关系
fadder_1 add_1 (i_A[0], i_B[0], i_Cin, o_Cout);
```

该代码与上面正确的实例化代码相比，在 o_Cout 信号前少一个逗号。这时按照端口连接列表和模块端口声明的排列顺序，o_S 端口将连接到 o_Cout 信号，而模块的最后一个端口将悬空。

注意：信号连接类型

模块端口和与之连接的信号的数据类型必须遵循如下规定：

1. 输入端口在模块内部必须为 wire 型数据，在模块外部可以连接 wire 或 reg 型数据。
2. 输出端口在模块内部可以为 wire 或 reg 型数据，在模块外部必须连接到 wire 型数据。
3. 连接的两个端口位宽可以不同，但其仿真结果可能因 Verilog HDL 仿真器而异，通常会有警告。

1.3　层次化设计思想

在讨论了模块及其实例化的方法后，下面进一步了解层次化设计的概念。层次化设计是 Verilog HDL 设计描述的一种风格，而模块实例化则是其具体的实现方式。通过图1.1，可以理解模块实例化的实质是利用层次化关系来描述电路结构。

前面已经提到，模块是 Verilog HDL 设计中的一个基本功能单元。狭义来看，模块与模块之间的关系，也就体现了设计层次的概念。比如，若模块 A 与模块 B 是一个并行连接的关系，那么 A 与 B 处于设计中的同一层次。再如，若模块 A 与模块 B 是包含或被包含的关系，那么 A 与 B 则分别是设计中的上层或下层。

数字电路设计中，一种重要也最基本的设计方法是自顶向下(Top-down)的设计。所谓自顶向下，是指从整个系统设计的顶层开始，往下一层将系统划分为若干子模块，然后再将每一个子模块又向下一层划分为若干子模块。通过这种方式将整个系统逐次向下分解，一个顶层设计最后可以细分为若干较小的基本功能块，直到不能继续分解为止，如图 1.4 所示。自顶向下的设计可以允许多个设计者同时设计一个系统中的不同模块，而且在不同的层次上都可以对设计进行仿真验证。

以例 1.1 中的全加器设计为例来简单分析其设计层次。模块 fadder_4 是设计的顶层，4 位全加器向下划分成为 4 个较小的 1 位全加器子模块。模块 fadder_1 是设计的底层，实现了 1 位全加器的功能。顶层模块通过实例化调用 4 个 1 位全加器，将其串联在一起构成最终的四位全加器电路。从图 1.1 中，可以清楚地看到全加器的层次化结构。

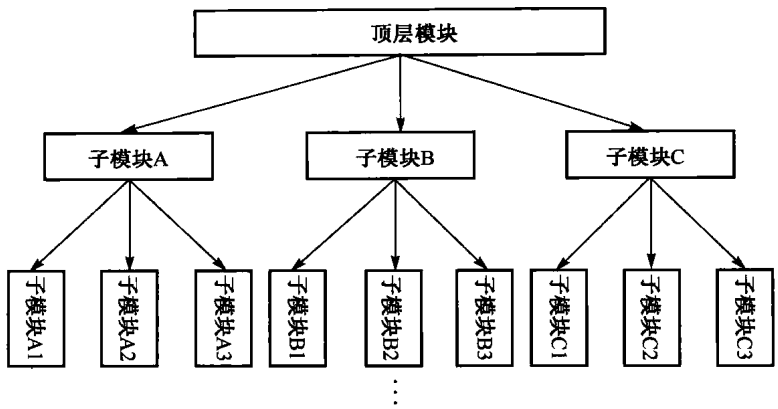


图 1.4 自顶向下的设计

另一种常见的设计方法是自底向上(Bottom-up)的设计，它与自顶向下的设计相反，首先对现有的底层功能模块进行分析和设计，然后使用这些模块来搭建上一层的功能更丰富的模块，直至完成顶层模块的设计。

提示：Verilog HDL 并行编程的思想

在 Verilog HDL 设计中，各个模块总是并行地出现的。由于 Verilog HDL 描述的是电路的结构，而各个模块最后也可以对应到实际电路图中不同位置的功能模块；因此在进行仿真运行时，Verilog HDL 语言中的各个模块的运行是并行的，即在同一仿真时间各个模块同时完成计算。这种并行的运行方式是 Verilog HDL 与其他高级编程运行(如 C 语言)的最本质的区别。Verilog HDL 初学者最容易犯的错误，就是在编写 Verilog HDL 代码时还抱有 C 语言顺序编程的思想，最后导致写出的 Verilog HDL 代码的仿真结果和预想的不同。因此，初学者需要在学习 Verilog HDL 时多参考各种电路模型的实例，并且亲自利用仿真工具对实例进行仿真，通过观察正确的仿真结果和波形图来慢慢体会并行编程的思想。

1.4 Testbench 的概念

除了利用层次化设计思想设计电路的功能模块外，通常一个完整的 Verilog 系统设计还应该包括测试模块。必须对设计进行全面的测试以验证其功能正确与否，以便在进行芯片生产前及时发现问题并进行修改。在设计数字电路系统时，通常将测试模块和功能模块分开设计，其中测试模块也称测试台(Testbench)。Testbench 同样可以用 Verilog 来描述，这使得系统测试更为容易。

Testbench 是通过对设计部分施加激励，然后检查其输出正确与否来完成其验证功能的。下面以例 1.1 的设计为测试目标，来设计一个简单的 Testbench。

【例1.3】为例 1.1 中的全加器设计 Testbench，以验证其功能。

```
// example_1_3: 一个简单的 Testbench
module tb_fadder ();
    reg  [3:0] A, B;
    reg  Cin;
    wire [3:0] S;
    wire Cout;
```

试读结束，需要全本PDF请购买 www.ertongbook.com