第3章 数字集成电路设计描述与仿真

CHAPTER 3

目前集成电路大多都是SoC的设计，对SoC的设计大多采用自顶向下的设计方法，本章对设计方法中的各个层次进行简单介绍，同时介绍描述电路的几种常用方法。电路设计的正确与否是通过仿真完成的，早期的电路设计中，是通过在面包板或者电路板上对设计的电路进行调试来验证其功能，但随着电路规模的增大，再采用此方法显然是非常不现实的。幸运的是，随着EDA技术的发展，目前的集成电路设计可以采用各种软件在设计的不同阶段进行功能、时序等方面的仿真，从而快速及时地发现设计中出现的问题，对设计进行修改。本章后半部分即对仿真的一般概念、涉及的常用EDA软件进行介绍。最后还对业界广为采用的系统验证方法——UVM进行简单介绍。

3.1 数字集成电路的设计描述

工艺的进步使得数字电路的设计进入数字系统级的设计，一个集成电路芯片上集成的电路规模也变得十分庞大，要保证制造的芯片能正确工作，最初的设计是很关键的。在数字系统集成电路设计中，需要完成两方面的任务：①根据所要设计的电子系统硬件功能和行为描述出相应的电路结构；②对得到的电路进行仿真，以验证所设计电路是否确实满足指标要求。

3.1.1 数字集成电路的层次化设计及描述域

有效的设计方法是电路与系统设计成功的关键，而无论是自下向上的设计方法还是自顶向下的设计方法，均采用了层次化的设计思想，使得设计能力有了很大的提升。目前自顶向下的设计方法被广泛采用，这种设计方法其实是根据设计的抽象层次来划分的，也就是将一个复杂系统依次分解为复杂性较低的设计层次，使得最终实现系统的设计层次复杂性足够低。

1. 层次化设计方法定义

把完整的硬件设计划分为一些不同的设计层次，允许多个设计者同时设计一个硬件系统中的不同层次模块，其中每个设计者负责自己所承担的部分。层次化设计中对于每一层一般都可以在结构域或行为域中进行。

2. 集成电路硬件设计层次

图3.1给出了典型的自顶向下设计流程与抽象层次的关系。

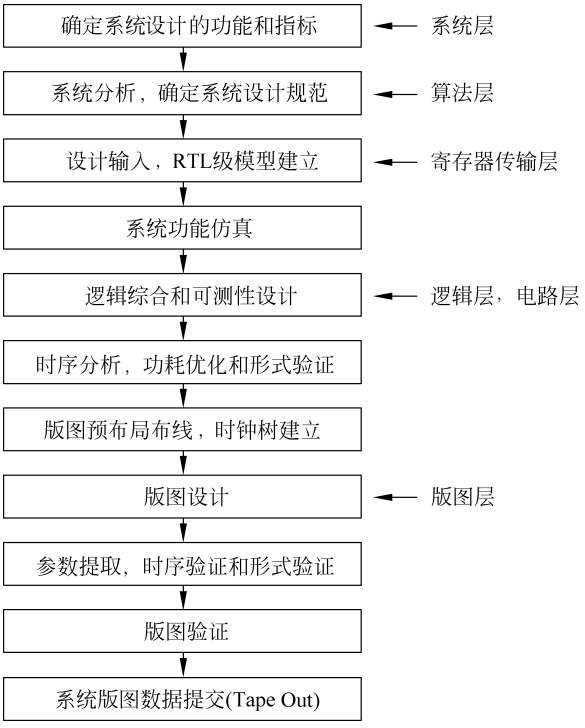


图 3.1　自顶向下设计流程与抽象层次关系图

（1）系统层：是整个设计过程的第一步，也是最关键的一步。主要进行电路功能的定义、各种电学参数（如工作频率、功耗、工作温度等）的确定。这一层的设计好坏直接决定了整个集成电路性能的好坏、价格的高低、市场的占有率，更决定了后续设计阶段的难易程度及效率。

（2）算法层：主要进行算法设计及描述。首先根据系统的功能要求，制定可以实现此功能的不同算法，分析和比较这些算法的优缺点，选定一种最适合的；其次，根据所选定的算法对整个系统进行功能划分及各个功能模块之间的数据流与控制流连接；此外，时钟方案等对系统性能起关键作用的部分也在算法层实现。由此可见，算法层的设计对于整个系统的性能起着至关重要的作用。

（3）寄存器传输层：RTL，主要将算法层用代码的形式进行抽象描述。在进行 RTL 层描述时，必须严格按照设计所使用的综合工具要求的规范进行，并尽可能地考虑时延、面积、测试等问题。

（4）逻辑层：将系统再进一步细化，转换为用普通的逻辑门来实现。对于当前的设计方法来讲，这一层次主要借助于综合工具来完成。

（5）电路层：将逻辑层中的门电路用具体的晶体管、电容、电阻等基本电子元器件来表示，并将之间的互连关系呈现出来。

（6）版图层：将各种电子元件用物理的方式呈现出来，根据所选工艺将这些元件转换成不同的几何图形，并相互连接。版图层的实现方式是系统最终的呈现方式，也是整个设计中最低的层次，并且仅仅是结构描述。

3. 集成电路描述域

层次化设计将整个设计划分为不同的抽象层次，而这些抽象层次又可以根据设计的表现形式不同分为几个不同的描述域，如表 3.1 所示。

表 3.1 集成电路设计的抽象层次与描述域关系表

抽象层次	行为域	结构域	物理域
系统层	设计的功能和指标	处理器、存储器等	
算法层	算法	硬件模块框图	
寄存器传输层	传输方程、状态机等	寄存器、ALU 等	宏单元
逻辑层	布尔逻辑函数	门级电路	标准单元库
电路层	网络方程	晶体管级电路	工艺层
版图层			版图

行为域主要关注系统的功能实现，对系统的输入/输出关系进行描述；结构域关注系统中每一抽象层次的实现方式，包含具体的逻辑和电路结构；物理域则更加关注集成电路最终的呈现方式，以物理特性表征。

3.1.2 集成电路设计的描述方式

设计描述是将所设计的电路以某种形式表达出来。整个集成电路的设计过程中主要包括系统设计、算法设计、逻辑设计、电路设计、版图设计等，这些设计过程所需要进行的设计描述有功能描述、逻辑描述、电路描述和版图描述等。这些描述均可以采用图形方式或者文字方式进行。

1. 图形描述方式

图形描述方式可以直观地描述一个设计的整体层次关系、主要输入/输出端口、关键信号的输入/输出关系等。根据设计过程中对应的层次不同，采用的具体图形方式也不一样。如采用方框图和原理图描述电路结构，而采用状态图、时序波形图描述电路的功能等。

图形描述直观易懂，在数字系统集成电路设计中是一种重要的设计手段。但在具体设计时，要受到特定的工艺库或者逻辑宏单元的限制，而且在电路规模很大时，用图形化描述方式也不方便易读，通用性和可移植性均差一些。

2. 文字描述方式

文字描述可以描述电路的结构，也可以描述电路的行为，特别适合描述复杂行为。采用的方式有自然语言描述、网表、硬件语言描述等，其中硬件描述语言（Hardware Description Language，HDL）采用最多。

HDL 语言主要有 VHDL 和 Verilog HDL 两种，但因为 Verilog HDL 的语法结构与 C 语言类似，编程风格简洁、高效，更容易被设计者掌握，所以 Verilog HDL（具体会在第 6 章详细介绍）成为业界目前最常用的设计语言。

用 Verilog HDL 描述电路行为通常有两种描述方式：①算法式——通过定义硬件的输入激励/输出响应描述硬件的行为，与硬件实现无关；②数据流式——采用与硬件物理实现相一致的数据流动方式描述硬件行为。一般认为，硬件行为的算法式描述是硬件的芯片级实现，数据流式硬件行为描述是硬件的寄存器级实现。（这里的实现是指硬件描述语言的实现，而非物理实现。）

【例 3.1】 设计一个半加器电路。

解：(1) 使用自然语言进行描述。

① 端口定义：A、B 为电路的输入端，Sum、Cout 为电路的输出端。

② 电路行为描述：电路实现两个 1 位二进制数 A 和 B 的相加，产生和 Sum，并向高位进位 Cout。

(2) 使用框图进行描述，如图 3.2 所示。

(3) 使用逻辑电路图进行描述，如图 3.3 所示。

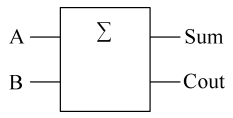


图 3.2　半加器的框图描述

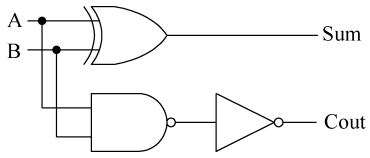


图 3.3　半加器的逻辑图

(4) 使用波形图进行描述，如图 3.4 所示。

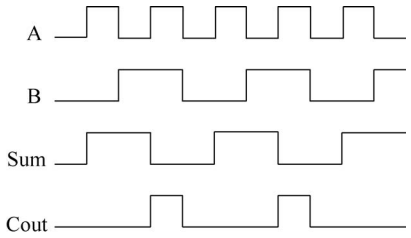


图 3.4　半加器的输出波形图

(5) 使用真值表进行描述，如表 3.2 所示。

表 3.2　半加器的真值表

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(6) 使用 Verilog HDL 进行描述。

```
module ha(A,B,Sum,Cout);                    //半加器模块名"ha"
input A,B;
output Sum,Cout;
assign{Cout, Sum} = A + B;
endmodule
```

(7) 使用版图描述，如图 3.5 所示。

3. 两种描述方式的应用特性

无论是文字描述方式还是图形描述方式，都有其优缺点，但这两种描述方式均可在上述提到的两种描述域——行为域和结构域中对电路进行描述。

在进行集成电路设计时，可以采用不同的描述方式，一般选择原则是：①文字方式适合描述行为，特别是复杂行为；②图形方式适合描述器件的内部互连关系，即描述结构；③在

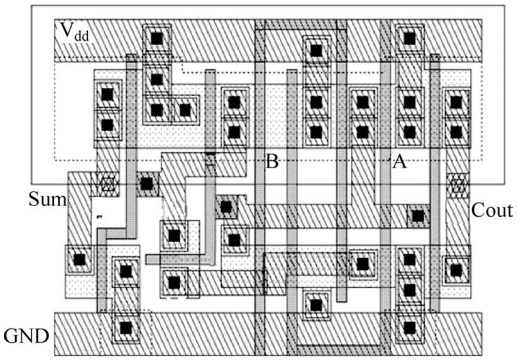


图 3.5　半加器的版图

进行大规模系统设计时，对于不同的设计层次，采用的描述方式往往不同，因此在整个设计过程中，文字描述方式和图形描述方式通常要交叉使用。

3.2　集成电路逻辑仿真与时序分析

整个数字系统集成电路设计中，不仅需要根据电子系统硬件的功能和行为描述出相应的电路结构，而且还需要对所设计的电路进行仿真，以验证其是否满足设计指标要求。随着设计规模的增大，验证在整个产品开发中也越来越重要，已经占到整个开发时间的一半以上，有些复杂的设计甚至达到了 80%。此外，电子产品的高速、低功耗特点也对产品的时序问题带来了严峻的考验，在制造前对设计进行良好的时序分析，也是保证产品成功的关键。

3.2.1　集成电路设计验证

随着集成电路设计规模的扩大，单个芯片上包含的晶体管数越来越多，如华为麒麟 9000 芯片集成了 153 亿个晶体管，苹果的 M2 Ultra 芯片集成的管子数更是多达 1340 亿个。因此，设计验证的工作量已经超过设计本身，设计验证也因此成为设计中一个至关重要的步骤。

现在集成电路设计采用的是层次化的设计方法，从系统层到版图层是一个从抽象到具体的过程，下一层是在上一层次工作的基础上进行的，设计验证同样如此，验证当前对象的行为是为了查看是否与上一层次的设计一致。例如，验证设计的逻辑功能是否符合设计最初设定的设计规范；在原理图设计和 HDL 设计后进行的功能原理仿真工作，主要是针对所设计硬件电路逻辑功能的验证；对逻辑层的仿真验证是为了确保综合后生成的网表行为与 RTL 级模型一致等。

此外，随着工艺的进步，验证设计结果的时间是否符合原始设计的要求也越来越严格。这是因为深亚微米级以下工艺条件下，为了生产出更小尺寸的芯片，提高集成度，芯片内部连线的平均长度会增加，而连线本身会引起电容、电阻、电感等寄生参数效应。随着器件尺寸的缩小以及对电路高速低功耗的要求，连线的这些寄生效应成为影响整个设计性能（如速度、功耗和可靠性等）的关键因素。而对连线的延时分析跟连线的布局有很大关系，因此逻辑综合后的仿真和版图布局布线后的仿真工作，除了进一步验证所设计电路逻辑功能的完

备性外，更重要的是对所设计电路的时序完备性进行验证。这与逻辑综合后对网表的仿真有区别，后者主要验证的是逻辑器件的门延迟对设计的影响。

对一个设计进行验证，往往做不到对这个设计的完整功能进行验证，因此有必要在验证时首先制定一个详细的验证标准，包括要验证的电路功能、电路本身要达到的设计要求、关键的输入/输出一致关系等。

验证常用的方法主要有 3 种：①仿真（或称模拟）过程（Simulation/Emulation）。从电路描述（文字描述/图形描述）中提取模型；再将外部激励信号/数据施加于此模型；观察该模型的响应，判断电路系统是否实现预期功能。②规则检查（Design Rule Checking）。分析电路设计结果中各种数据是否符合设计规则（如 ERC、DRC 等）。③形式验证（Formal Verification）。分为两个方向：等价性验证和模型验证，但无论哪个方向都基于严密的数字逻辑理论体系，都是用理论证明方法来验证设计结果的正确性。

1. 仿真（或称模拟）过程

这里的仿真（或称模拟）（Simulation/Emulation）主要是指常用的软件仿真。它是验证中最常用的方法，设计的层次不同，仿真的层次也不同，例如系统层仿真、电路层仿真、寄存器传输层仿真等。

软件仿真利用仿真工具（如 ModelSim、NC-Verilog、VCS 等）对设计的行为进行模拟。一般来讲，所要验证的对象是设计本身（大多数情况下是 Verilog 代码），但随着设计规模的扩大、功能复杂性的增强，纯粹通过编写代码对设计进行验证的效率会很低，于是就出现了“验证方法学”。验证方法学是可以快速完整地建立验证环境，并且对不同设计可以重复使用的方法学，能显著提高验证效率。

关于验证方法学的研究早在 2000 年就开始进行，并有相关的产品。2003 年，Synopsys 公司公布了可重用验证方法学库（Reference Verification Methodology，RVM），这个方法学采用了 Synopsys 公司的 Vera 语言。2006 年，Mentor 公司公布了高级验证方法学（Advanced Verification Methodology，AVM）。这个方法学主要采用了 OSCI SystemC 的事务抽象层方法学（Transaction Level Modeling，TLM）标准，它是用 SystemVerilog 和 SystemC 两种语言实现的。2006 年，Synopsys 公司推出了验证方法学手册（Verification Methodology Manual，VMM），这是 RVM 从 Vera 语言过渡到 SystemVerilog 的方法学。2007 年，Cadence 公司推出通用的可重用验证方法学（Universal Reusable Methodology，URM），主要是 eRM 从 E 语言过渡到 SystemVerilog 的方法学，同时加入了 TLM 接口、工厂模式替换、配置机制、策励类等。2008 年，Cadence 公司和 Mentor 公司共同推出了 OVM（Open Verification Methodology）。2010 年，Accellera 采用 OVM 作为基础，推出了 UVM（Universal Verification Methodology），同时引入 VMM 的 callbacks 等概念。作为业界方法学的一个统一雏形。2010 年，Synopsys 公司推出 VMM 1.2，基本上沿用了 OVM 的 TLM 通信机制，并采用 TLM 2.0（OSCI 最新的标准），采用 OVM 提出的 implicit phase，并且将验证流程继续细化，推出工厂模式替换机制，建立类层次（建立 parent 关系）。并且在此基础上，提出了 vmm_timeline 的概念，方便各个 phase 之间实现跳转，增加 phase 或删除 phase，增加了 rtl_config 等概念。2014 年，Accellera 又推出了 UVM 1.2 版本。目前，对 SoC 的验证普遍利用 UVM 验证平台进行。

2. 规则检查

规则检查（Design Rule Checking）主要检查电路设计结果中各种设计参数是否符合设

计规则。主要包含 DRC 和 ERC 两种。

DRC 是针对版图进行的，主要检查版图中各个掩膜层图形或不同的掩膜层图形之间的几何尺寸是否符合所选生产工艺的设计规则要求，这些设计规则主要包括版图几何图形的宽度、间距及层与层之间的相对位置(间隔和套准)等。DRC 的目的是确保设计的版图能在特定的集成电路制造工艺下流片成功，并且具有较高的成品率。由于不同的集成电路工艺具有与之对应的设计规则，因此设计规则检查与集成电路的工艺有关。由于这个验证步骤的重要性，DRC 是版图验证的必做步骤。ERC 则是检查版图是否有违反电学规则的错误，如短路、开路和悬空的节点，以及与工艺有关的错误，如无效器件、不适当的注入类型、不适当的衬底偏置、不适当的电源、地连接和孤立的电节点等。完成 ERC 检查后，按照电位的不同来标记电节点和元器件，并且产生图示输出。

进行规则检查的主要工具有 Cadence 公司的 Diva 和 Dracula(Cadence 新版本中的工具名称为 Assura)，Synopsys 的 Hercules 以及 Mentor 公司的 Calibre 工具等。Diva 工具是一个集成在版图编辑器(常用的是 Cadence 公司的 Virtuoso 工具)内的交互式验证工具，嵌入在 Cadence 的主体框架中，用来寻找并改正违反设计规则的错误，包含检查物理设计、电学设计和进行电路图与版图的一致性检查等功能。Diva 属于在线验证工具，在版图的设计过程中可以按照需要随时对版图进行设计规则的检查，方便及时发现错误并纠正。Dracula 工具是一种离线式的验证工具，需要先将版图转换为 GDS 文件后才能进行，但它适用于从小单元到大规模的集成电路，而 Diva 一般用于小规模的电路设计中。Calibre 工具是目前业界用得最多的深亚微米集成电路设计规则检查工具，其具有先进的分层次处理功能，支持平坦化和层次化的验证，大大提高了超大规模集成电路的验证速率。此外，Calibre 工具还具有独特的验证结果视图环境(Results Viewing Environment，RVE)界面，将验证的结果反标到版图编辑器中，准确快捷，一目了然。

3. 形式验证

形式验证(Formal Verification)是基于理论分析将待验证电路的功能描述与参考设计进行对比，以判断是否达到了设计要求，因此形式验证不需要获得测试激励。此外，形式验证可以对待验证电路的所有可能情况进行分析验证，而利用测试激励对大规模电路进行验证时只是尽可能多地考虑所有的工作情况，因此形式验证能克服仿真验证不全面的缺点。此外，形式验证可以进行从系统级到门级的验证，而且验证时间短，有利于尽早、尽快地发现和改正电路设计中的错误，缩短设计周期。

形式验证分为两种类型：等价性验证和模型验证。顾名思义，等价性验证是进行待验证电路与参考电路的一致性对比，它是目前在集成电路设计中经常用到的。整个电路的设计是分层进行的，在层和层的转化时都需要进行一系列的设计步骤，例如综合、布局布线、可测试性设计、时钟树的插入等，这些操作无疑都会对设计带来一些改变，利用等价性验证即可全面地检验变换前后的电路功能是否一致。等价性验证的原理是：首先建立被比较的两种电路的模型，之后依据两个模型之间的关系，自动确定被比较的两个设计的等价性，而不需要用户的输入，这使得在因为某种原因只对设计局部进行修改而不做功能性改变时，可利用等价性验证，避免进行长时间仿真。

综上所述，可以看出规则验证属于物理范畴，而仿真过程和形式验证都是基于电路的功能进行的，两者存在很多不同之处，如表 3.3 所示。

表 3.3　仿真与形式验证的区别

比较项目	仿　　真	形 式 验 证
激励源	需要激励源，且仿真效率受此影响	只对电路描述本身进行分析，不需外加激励
实现路径	通过信号在电路元件之间动态传播实现	通过静态逻辑推理实现
错误分析	不能直接指出电路是否有错误和错误位置，需用户自己分析仿真结果	直接给出“正确”或“错误”结论

虽然形式验证在大规模电路验证方面存在很多优势，但最初的参考电路功能还是需要通过仿真来验证，即仿真是一切验证的基础，因此本书主要介绍仿真的一些基本知识，而对于形式验证，后面只给出一个简单的例子加以说明。

3.2.2　集成电路设计验证中的逻辑仿真

早期的集成电路设计规模不大，对电路的验证主要通过硬件方式进行，即通过在电路板上构建验证系统进行调试验证。而随着集成电路设计规模的不断扩大，如果还采用类似的硬件仿真方式，肯定费时费力，甚至行不通。因此，目前集成电路设计领域中仿真的基本原理是在集成电路制造出来之前，利用计算机软件工具构造硬件模型，给定输入激励，模拟确定电路响应，验证硬件设计正确性的过程，如图 3.6 所示。

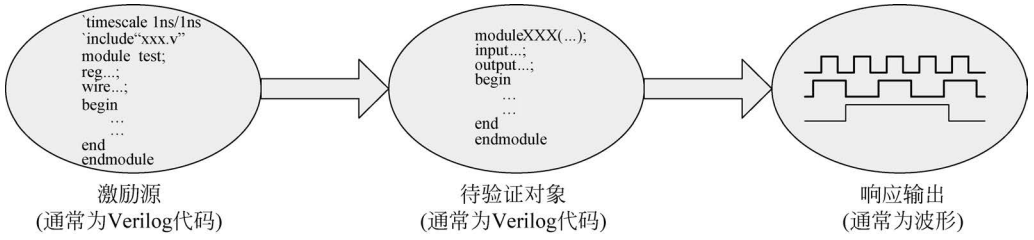


图 3.6　仿真路径图

对数字逻辑系统的仿真，一般称为“逻辑仿真”。图 3.6 表述的即是数字逻辑系统的仿真路径图。目前的数字逻辑系统设计一般都用 Verilog HDL 语言描述，对其仿真时所用的激励源(也称 Testbench)一般也是 Verilog HDL 代码，对系统进行仿真时，利用软件将激励源施加到待验证对象上，结合不同设计层次的电路模型输出响应，最后对响应进行分析，来验证系统设计是否在功能和时序等方面符合预期的目标。

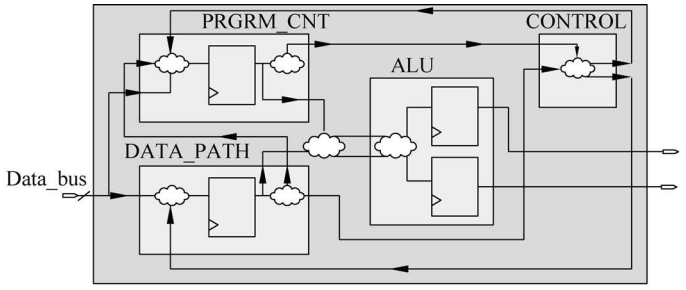
系统的设计遵循层次化的设计思想，同样，逻辑仿真根据不同的设计层次或仿真时器件的规模类型，按照由高级到低级又可分为不同的层次或级别，如功能块级仿真、逻辑门级仿真、开关级仿真等，如图 3.7 所示。

1. 功能块级仿真

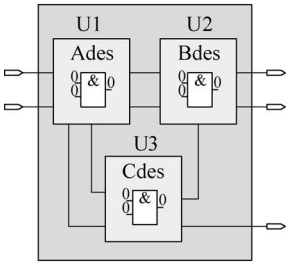
功能块级仿真中把寄存器、运算/控制器件和总线等作为基本单元，主要检查数据在各个寄存器中的传输情况，因此也常常被称作“寄存器传输级仿真”。此时系统的行为通过在寄存器间的数据流来表征，并分析设计是否符合预期要求。

功能块级仿真的抽象级别高，分为基于事件的仿真、基于时钟周期的仿真和基于对象转换的仿真。

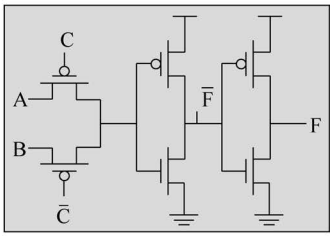
(1) 基于事件的仿真是把每次输入激励信号的变化都作为一个触发事件，根据这个触发事件来计算电路的运行结果。因为每一个信号变化就产生一个触发，因此在一个时钟周



(a) 功能块级示例



(b) 逻辑门级示例



(c) 开关级示例

图 3.7　系统不同层次级示例

期内可以有多次的触发存在。基于事件的仿真不仅包含电路的功能模型，也包含时序模型，因此仿真的精度高。

（2）基于时钟周期的仿真，是不论信号在整个时钟周期如何变化，对电路进行仿真时仅仅采集时钟上升沿或者下降沿到来时的信号值，也即在一个时钟周期内仅对电路进行一次计算处理。基于时钟的仿真加快了电路的仿真速度，但却降低了仿真的精度，且只适用于同步电路。

（3）基于对象转换的仿真与基于事件的仿真不同，它不需要在输入端口加入激励源，而是将数据包、图形等对象作为仿真激励施加于待测对象中。基于对象转换的仿真提高了测试激励的抽象层次，因此提高了仿真效率，适用于大规模的集成电路。

2. 逻辑门级仿真

逻辑门级仿真中把各种逻辑门、触发器和计数器等作为基本单元，主要检查逻辑设计的正确性和硬件有无竞争冒险等问题。此时，连接系统电路中这些基本单元的信号线用逻辑值 0，1，x 表示，通过在电路的输入端口中加载激励，在输出端观察响应来分析判断设计的功能和时序是否正确。

逻辑门仿真不仅可以用以验证电路的功能，而且更重要的是在仿真时，逻辑门单元的延迟信息会被调用，以此来验证电路的时序。对设计进行综合后所产生的门级网表进行仿真即属于逻辑门仿真。

3. 开关级仿真

开关级仿真中把每个晶体管看作一个独立的开关，此时整个系统电路看作由若干晶体管组成的结构，模拟硬件中信号强度对逻辑设计的影响。开关级仿真的精度最高，仿真时的计算量也最大。

开关级仿真中电路看作由晶体管和节点所组成的，根据电路的连接关系验证电路的逻辑功能，并且计算每个节点的逻辑值及其延迟时间。开关级仿真比逻辑门级仿真的时序更精确。

3.2.3　集成电路设计中的时序分析

目前集成电路设计中对性能的要求越来越高，相应地对于时序的要求也越来越高。因此目前时序分析已经独立于功能仿真，成为设计流程中一个不可缺少的非常重要的环节。

对电路进行时序仿真最传统的方法是对逻辑综合后的网表或者布局布线后的版图中所产生的网表施加相应的测试激励，模拟电路的工作环境，对电路进行再次仿真，即所谓的“动态时序仿真”。动态时序仿真需要对电路施加特定的测试激励来验证设计，是将功能仿真和时序仿真同时进行，因此动态时序仿真对于电路的时序分析结果较精确。原理上这种方法是没有问题的，但随着集成电路设计规模的增大，创建用以验证电路的时序测试矢量和功能测试矢量工作量都很巨大，因此采用传统的动态时序仿真方法会非常耗时。而且因为测试矢量未必对所有的时序路径都敏感，因此基本也不可能开发出完备的测试矢量既找出所有潜在的故障，又同时找出时序上的关键路径。以上原因促进了时序分析技术向静态时序分析技术的发展。

静态时序分析(Static Timing Analysis，STA)是基于路径对设计进行时序分析，其在整个设计流程中的位置如图 3.8 所示。在设计进行综合得出网表后，其根据电路网表的拓扑将设计分成若干路径的集合，通过计算每一条路径上的延迟(如建立时间、保持时间等)对电路的时序进行判断，以衡量电路的性能。由此可以看出，静态时序分析是不需要外加测试激

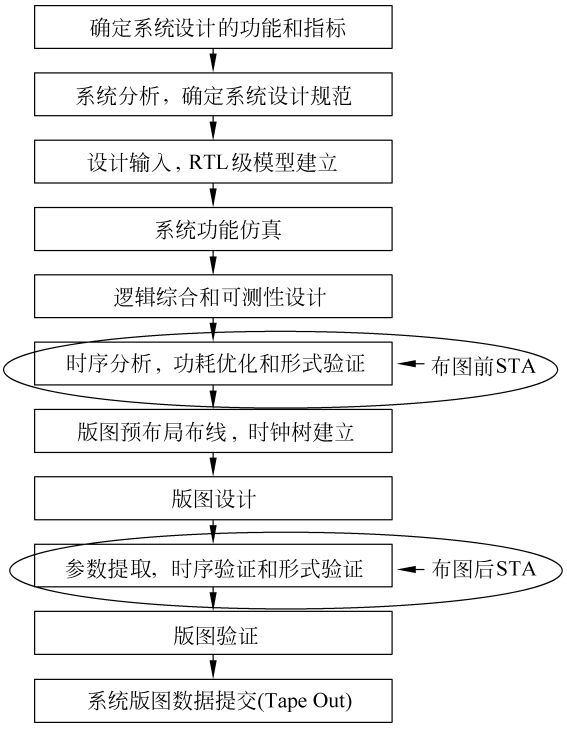


图 3.8　静态时序分析在设计流程中的位置

励的，因此运行速度快，占用的内存也小。同时，静态时序分析可以对电路中的所有路径都计算得出相应的时序信息，也即是穷尽所有的路径，理论上可以达到测试路径的100%覆盖率，因此比动态时序仿真的完备性强。但静态时序分析也存在着缺点，主要有：①不能同时进行功能仿真；②仅适用于同步时序电路。因此静态时序仿真和功能仿真验证是相辅相成的，为保证设计的正确性，同时进行这两方面的验证是必不可少的。常用的静态时序分析工具是Synopsys公司的PrimeTime工具。

1. 静态时序分析的基本原理

静态时序分析是基于路径的，路径的构成要素主要是组成电路的所有时序单元、门级单元等，因此STA主要是针对门级网表进行的。

静态时序分析是将整个设计分解为不同时序路径的集合，每条路径都有一个起点和一个终点。时序路径的起点只能是设计的基本输入端口或内部时序单元，如寄存器、锁存器的时钟输入端；时序路径的终点则只能是内部时序单元的数据输入端或设计的基本输出端。图3.9中用箭头标出了4条时序路径，分别代表了以下4类。

(1) 路径1：基本输入端口到时序单元的数据输入端。

(2) 路径2：内部时序单元时钟输入端到下一个内部时序单元数据输入端口。

(3) 路径3：内部时序单元的时钟输入端到基本输出端口。

(4) 路径4：基本输入端口到基本输出端口。

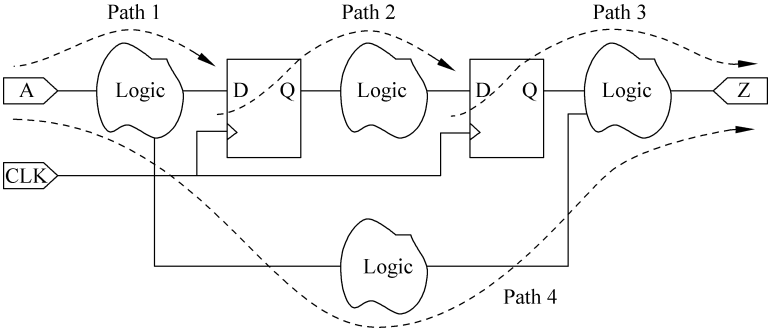


图3.9 静态时序分析过程中定义的时序路径

静态时序分析的一个重要目的是发现使芯片时序失效和对芯片性能起决定作用的电路关键路径，保证以上所有的路径都满足内部时序单元对建立时间和保持时间的要求。它采用穷尽分析方法，提取出整个电路存在的所有时序路径，计算信号在这些路径上的传播延时，检查信号的建立和保持时间是否满足时序要求，通过对最大路径延时和最小路径延时的分析，找出违背时序约束的错误。在工作过程中，静态时序分析的内容包含以下3个步骤。

(1) 按照前面所述的路径分类，把设计分成不同的时序路径集合。如图3.10所示给出了一个时序路径分割的示例，其把12条时序路径分成了3种不同的路径组合。其中Clock Group 1对应前述的路径1(基本输入端口到时序单元的数据输入端)，Clock Group 2对应前述的路径2(内部时序单元时钟输入端到下一个内部时序单元数据输入端)，Default Group对应前述的路径3(内部时序单元的时钟输入端到基本输出端口)。

(2) 计算每条路径的延时信息，为后续的路径延时检查做准备。

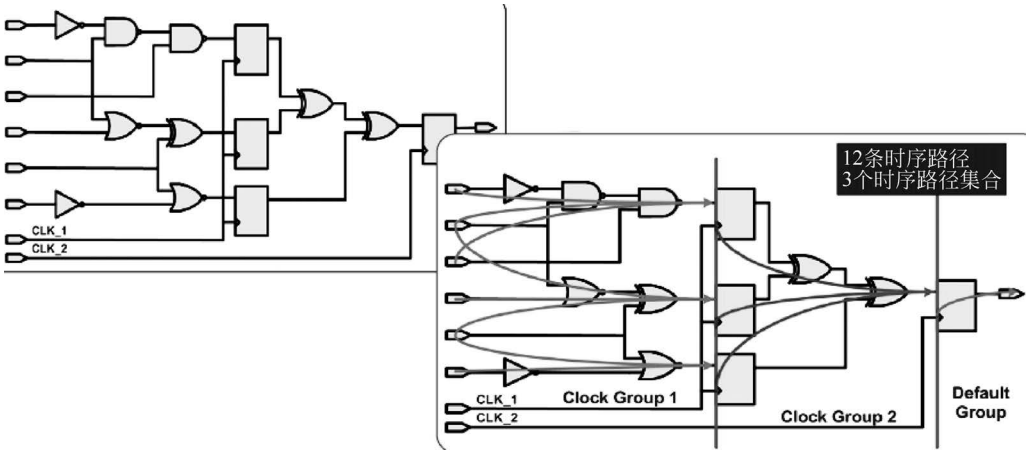


图 3.10　时序路径分割示例

(3) 检查所有路径的延时，分析时序约束是否可以满足设计要求。静态时序分析所要做的主要检查包括以下内容：建立时间和保持时间检查、门控时钟检查、数据恢复和数据移除检查、时钟脉冲宽度检查。

这些检查大致可以分为 3 类：对时序单元的检查、对时钟的检查、对组合逻辑的检查。在这些检查中，大部分都比较易于理解，下面先介绍几个基本概念，再着重分析一些基本的时序路径约束检查。

信号的到达时间(Arrival Time，AT)：表示实际计算得到的信号到达逻辑电路中某一时序路径终点的绝对时间之和。它等于信号到达某条路径起点的时间加上信号在该条路径上的逻辑单元间传递延时的总和。

要求到达时间(Required Arrival Time，RAT)：表示电路正常工作的时序约束要求信号到达逻辑电路某一时序路径终点处的绝对时间。

时间余量(Slack)：表示在逻辑电路的某一时序路径终点处，要求到达时间与实际到达时间之间的差，Slack 的值表示该信号到达得太早或太晚。

① 寄存器的建立和保持时间检查。

STA 对寄存器建立时间检查的目的是确保数据在时钟的有效沿之前到来，如图 3.11 所示，数据到达时间不能太晚，它必须满足：数据到来的最晚时间小于等于时钟有效沿最早到来的时间减去寄存器固有的建立时间。根据上面这个条件，可以得到时序路径时间余量的计算公式：

Slack＝RAT－AT＝(时钟有效沿最早到来的时间－寄存器固有的建立时间)－数据到达的最大延迟时间

STA 对寄存器保持时间的检查，其目的是保证数据在时钟的有效沿后能够稳定并保持足够长的时间以使时钟能够正确地采样到数据。如图 3.11 所示，这主要是保证数据不会到达太早，数据到来的最早时间大于等于时钟有效沿最迟到来时间加上寄存器固有的保持时间。由上面这个条件可以得到保持时间余量的计算公式：

Slack＝AT－RAT＝数据到达的最早时间－(时钟有效沿到达的最晚时间＋寄存器固有的保持时间)

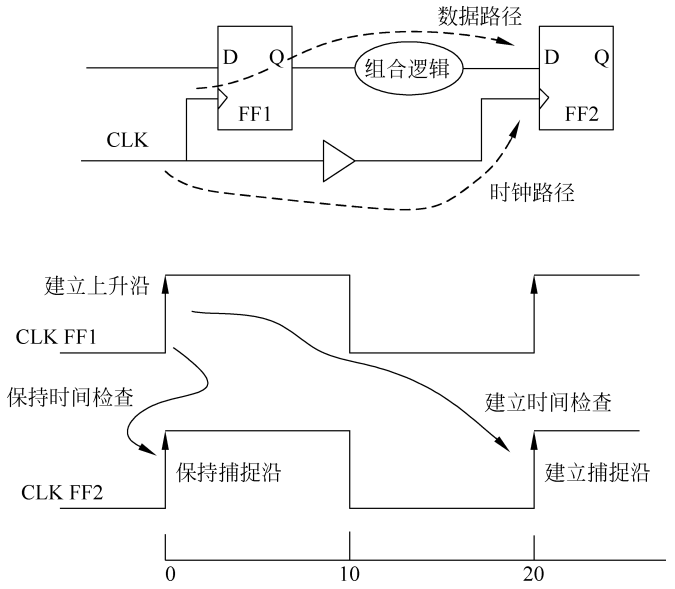


图 3.11　建立保持时间检查示意图

如果两个 Slack 中任何一个为负，就说明建立时间或保持时间不满足设计要求，要对约束条件进行改进：如果出现了建立时间不满足的问题，可以加快数据传送或是延迟时钟到来；如果出现保持时间不满足的问题，则需要加快时钟或是延迟数据。

② 同步时序电路周期检查。

同步时序电路中的各种操作都要受到时钟信号的控制，设计者既要保证电路工作频率尽可能高，又要保证电路在特定情况下工作可靠。图 3.11 也代表一个基本时序电路，参数如下：

t_{cq}，t_{cqmin} 分别是寄存器 FF1 和 FF2 最大和最小传输延时；

t_{su}，t_{hold} 分别是寄存器 FF1 和 FF2 的建立时间和保持时间；

t_{com}，t_{commin} 分别是组合逻辑的最大和最小延时。

在理想的情况下，时钟相位没有偏移，为了保证电路的正常工作，必须保证数据在一个时钟沿触发，经过 FF1 和组合逻辑的延时，在下一个时钟触发沿前到达 FF2，并且保证 FF2 有足够的建立时间。对时钟周期的约束公式为

$$T > t_{cq} + t_{com} + t_{su}$$

同时，FF2 寄存器采样数据还要求数据有足够的保持时间，也就是 FF2 要求的保持时间要小于 FF1 和组合逻辑的最小延时，约束公式为

$$t_{hold} < t_{cqmin} + t_{commin}$$

③ 门控时钟检查。

图 3.12 是静态时序分析支持的时序路径类型示意图，有门控时钟设计。对于门控电路的控制引脚，一般认为它有两种状态：使能和关断。在使能的时候，允许有效的时钟脉冲完整通过控制门；在关断的时候，不让时钟脉冲通过。

那么可以得到静态时序分析在做门控电路的建立和保持时间检查时的一个基本原则：绝不允许控制的 EN 信号的变化引起时钟有效脉冲的变化，图 3.13 表示一个门控电路的时序检查情况。

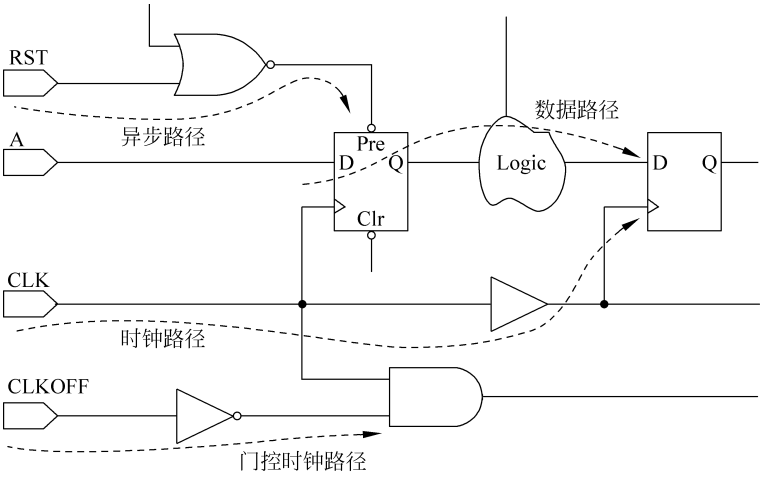


图 3.12　门控时钟检查示意图

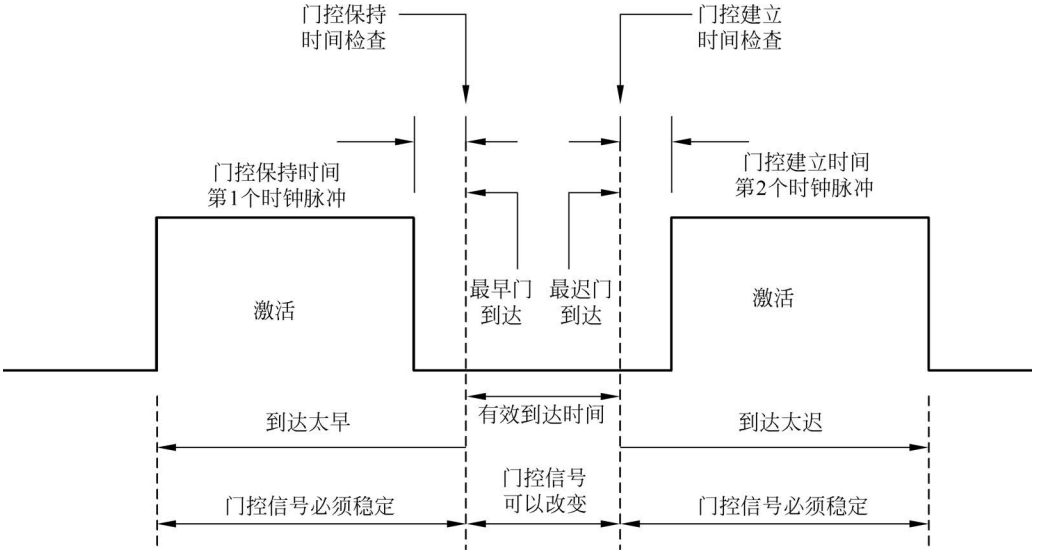


图 3.13　门控电路的时序检查要求

在做门控电路“建立时间”检查时，它检查的是有效时钟脉冲的前沿，在做门控电路“保持时间”检查时，它检查的是有效时钟脉冲的后沿，所以可得到关于门控电路时序检查所要求满足的两个方程式为

门控电路的建立时间余量(Slack)＝RAT－AT＝(有效时钟脉冲前沿到达的最早时间－门电路固有的建立时间)－门控信号到达的最迟时间

门控电路的保持时间余量(Slack)＝RAT－AT＝门控信号到达的最早时间－(有效时钟脉冲后沿到达的最晚时间＋门电路固有的保持时间)

2. 静态时序分析的特点

静态时序分析非常适合于同步设计，如流水线的处理器结构和数据通路类的逻辑电路。

同步时序逻辑电路的特点是电路主要由存储单元(时钟边沿触发的寄存器或电平触发的锁存器)和组合逻辑单元组成。相比于动态分析,静态时序分析在准确验证时序上具有多种优势。动态时序分析是一种与输入模式相关的技术。由于信号传输的路径和它的延时与电路状态相关,为了验证电路的所有路径,设计的输入模式必须逐一穷举。因此,在固定输入模式的前提下,设计的时序验证也只能局限在某几条路径上。考虑到动态仿真所花的时间和设计规模上的限制,动态时序分析仅仅适合小规模的设计。相反,由于静态时序分析不依赖于电路的状态,从而避免了动态分析的弊端。当然,动态时序分析还广泛应用于一些复杂的电路中,比如锁相环(Phase Locked Loop,PLL)、时钟发生器等。

STA 之所以能实现如此强大的功能,其主要原因是该技术与电路输入模式无关,无须以穷举的方式验证设计中所有的路径,从而实现对大规模集成电路的时序验证。另外,由于其能容易地实现对抽象层次的验证,因此在对整个芯片的时序分析中,该技术非常有效。正如前文提到的,静态时序分析的目的之一是找到设计中的关键路径,该路径的信号时序决定着芯片的工作频率。可想而知,这条路径的延迟时间(即从输入端到输出端的延时)必定是芯片所有路径中最大的,由于时钟的最小脉冲宽度必须大于最大的延时,因此这些关键路径决定了设计的工作频率。

STA 的另一个特点是可识别的时序故障数要比动态仿真多得多,如前面所说的,包括建立/保持和恢复/移除检查(包括反向建立/保持)、最小和最大跳变、时钟脉冲宽度和时钟畸变、门级时钟的瞬时脉冲检测、总线竞争与总线悬浮错误、不受约束的逻辑通道。另外,一些静态时序工具还能计算经过导通晶体管、传输门和双向锁存的延时,并能自动对关键路径、约束性冲突、异步时钟域和某些瓶颈逻辑进行识别与分类。

3.2.4 逻辑仿真与时序分析不足

对于一些工艺要求不高的小规模电路来讲,经过逻辑仿真后如没有问题,基本可以保证设计出的电路也是正确的。但对于功能复杂、规模较大的集成电路设计来讲,仅仅靠逻辑仿真来验证远远是不够的。这主要是因为门级逻辑仿真存在以下一些显著的缺点:①门级逻辑仿真对于验证电路时序的准确性在很大程度上依赖于仿真激励的完备性,这是最重要的一点。为了得到较高的仿真覆盖率,门级逻辑仿真需要大量的仿真激励,为了达到 100%的仿真覆盖率,在建立仿真激励文件时需要考虑到涉及的所有可能工作情况,对于复杂的设计来讲,这显然是不切合实际的,合理而充分地选择测试激励是一个比较复杂的问题。②基于事件驱动的门级逻辑仿真需要耗费大量的工作时间。电路中一个信号线的值发生变化就称为一个“事件”,相应的“事件”发生时,与此有关的门电路或功能电路的值都会发正变化。而设计时往往需要对设计进行多次的修改才能确定下来,每次修改后都需要进行仿真,因此设计人员消耗在仿真上的时间往往是设计时间的 2～3 倍;③由于深亚微米工艺的影响,通常需要在不同设计阶段进行多次针对网表的仿真,进而使得工作时间开销也变得难以接受。针对这些问题,目前的设计过程中采用了验证平台的方法,这将在后续章节进行详细介绍。

此外,静态时序分析虽然能够进行完备的电路时序分析,但也存在一些自身的弱点:①静态时序分析无法验证电路功能的准确性。这一点必须由逻辑仿真工作来完成。②静态时序分析一般只能有效地验证同步时序电路的准确性。如果设计中包含有异步电路的时序验证,则必须通过门级逻辑仿真来保证时序的准确性。③静态时序分析和门级逻辑仿真从

不同侧重角度来验证电路，以保证电路时序与功能正确，它们是相辅相成的。

3.3　仿真建模与仿真流程

系统电路的仿真验证是基于计算机进行的，将激励施加于待测电路，实际是将激励施加于电路的模型中观察其输出的状态及系统时序的过程，以此来衡量和评估实际设计的真实工作情况和真实性能。由此可以看出，建立正确的电路仿真模型是很重要的。

3.3.1　数字系统仿真模型的建立

仿真模型不是实际的电路元件，而是电路在计算机内部的表示形式，用来表示电路结构或行为。仿真模型越接近实际电路，模型就越复杂，相应的消耗的仿真验证时间也会越长，但得到的仿真验证的结果会越精确。选取什么样的仿真模型，取决于仿真验证的目的和对仿真的精度要求。

仿真模型一般有以下 4 种：功能模型、延迟模型、功率模型和时序模型。

功能模型用于仿真数字逻辑单元的功能。功能模型用功能和行为来描述，不关心其内部结构和组成。每一类功能块的功能是固定的，其输入和输出之间的关系用布尔逻辑关系表示，计算机在进行仿真时，将此布尔逻辑关系翻译成功能计算子程序，由模拟器调用。此外，功能块中包含多个输入/输出，每个输入/输出的负载又往往是不同的，因此延迟也不同，这在功能模型中也要包含。除了描述逻辑功能的布尔逻辑关系和延迟时间外，功能模型中还有用于综合的时序约束，如时钟脉冲、建立时间、保持时间等。仿真验证过程中如发现违反这些约束条件的情况，则必须将错误信息反馈给设计者。

延迟模型用于仿真数字逻辑单元的延迟。每个信号在通过门电路时都会产生延迟，延迟时间的计算关系到逻辑仿真验证的正确性。如果延迟模型准确，则通过分析计算得到的延迟时间能精确反映电路的实际工作情况，从而判断电路的时序是否可以满足设计的要求，在固定时钟周期内是否能保证完成所需的操作。因为延迟时间跟电路的负载、信号的电平转换时间等有关，因此根据不同的情况，延迟模型又可以分为以下几种。

(1) 零延迟模型。所有的门电路延迟时间都为 0。这显然是不可能的，但在仅需要验证组合逻辑电路功能的情况下，此种延迟模型会采用。

(2) 标准延迟模型。给每个元件设定一个标准的延迟值，这个值是工艺厂经过模拟验证而给出的。但这种模型往往不考虑电路中的寄生元件情况，因此与实际工作情况也有差别，不过对于大多数电路来讲已经足够准确。

(3) 上升/下降延迟模型。考虑信号的上升时间和下降时间对门电路延迟的影响而建立的，可以更准确地反映电路的实际工作情况。

(4) 模糊延迟模型。给每个元件设置一定范围的延迟时间，即给出延迟的最大值和最小值。这种模型一般适用于小规模的功能电路。

功率模型用于仿真数字逻辑单元的功耗。功率模型描述每个门电路的功耗，必须包含：①静态和动态功耗；②I/O 端口和内部节点的开关状态；③I/O 端口和内部节点的状态；④运行方式(如测试方式)；⑤运行、电压和温度等条件以及电容负载和输入瞬变时间。

时序模型：用于仿真数字逻辑单元之间的延迟。在此模型中不仅包含门电路的延迟，

还包含了电路连线的延迟。连线的延迟与连线的长度、宽度等信息有关，因此在进行版图后仿真时，时序模型运用得较多。

仿真模型建立中需要注意：

(1) 仿真单元的模型随描述层次的不同而不同。例如，在门级电路中，元件是各种基本门和触发器；而在功能块级，是一个组合电路模块。

(2) 不同层次信号值的规定也不同。例如，门级和功能级电路中信号一般用 0 或者 1 表示，而高层次描述中信号为一个位串。

(3) 信号的延迟时间也有不同的模型。例如，在门级和功能块级，延迟时间泛指元件的延迟时间；而在高层次描述中的延迟，一般指信号赋值时的延迟。

3.3.2 数字系统仿真流程

前面提到，仿真是利用软件对电路的不同层次开展的验证工作。如前所述，在对电路进行仿真时，首先必须建立一个针对电路的仿真模型，这个模型的合适与否不仅关系到能否全面正确对设计进行评判，也关系到仿真的速度。一般工艺库里都包含有相应的电路模型供使用。其次是建立合适的激励，其一般是由电路正常工作时可能存在的不同输入情况组合在一起形成，激励包含的输入情况越全面，对电路的仿真也越接近于实际情况，但随着电路规模的增大，仅依赖于传统的写激励文件的方式显然是不现实的，因此目前对系统的验证多采用验证平台的方法，这将在后续详细介绍。最后就是将激励施加于仿真模型，然后观察输出，进而判断设计正确与否。仿真流程图如图 3.14 所示。

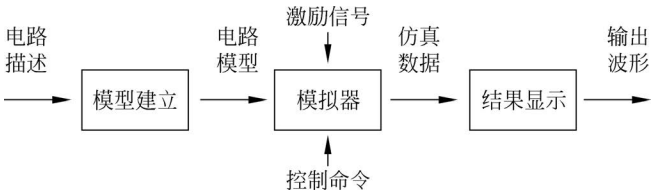


图 3.14 仿真流程图

仿真过程中一般以时钟为基准来观察输入/输出变化，仿真时一般均从 0 时刻开始，时钟可以自定义单位，也可以采用默认的单位。电路仿真中信号线的值发生改变就称为 1 个“事件”，事件的构成有 3 要素：信号名、信号值和事件发生时刻。如 A 信号在时钟 15ns 处为 1 值，则事件为(*A*，1，15ns)。在整个仿真过程中，输出是随着输入发生变化的，因此，输入激励的变化就相当于事件的触发条件。在每一个仿真时刻都可能会有事件触发，仿真软件会根据触发内容对电路的输出重新求值，设计者通过观察每一个事件的结果来判断电路在此触发情况下是否达到了设计要求。一个事件结束后会因为输出的变化而触发下一级电路的一个新的事件，所以事件之间是按照一种设计时的约定次序发生的，即形成一个事件链。在每一个时刻会存在多个并行的事件，对于这些并行的事件，仿真软件也是进行并行处理的。处理完当前的并行事件后就会将其从事件队列中删除，转而去执行接下来的事件。

事件处理的示意图如图 3.15 所示。

在计算机上实现仿真时，对于每一种逻辑器件，有一个或几个模型函数，每当器件输入端信号发生变化时，就调用这些函数，用于计算器件的输出；如果输出有变化，则事件发生，

此时，根据输入信号的变化时刻，建立事件队列；在某一个仿真时刻，可能会存在多个事件，只有当这一时刻对应的所有事件都处理结束后，才会将仿真时刻往前继续。

采用事件处理的最大优点是：只对那些输入有变化的电路器件进行计算处理。下面举一个简单例子来说明事件处理的仿真过程。

如图 3.16(a)所示电路是一个简单的一位比较器电路，逻辑分析很容易。但因为实际电路中每个门电路的输入到输出都存在延时，且连线之间也存在延时，因此电路实际工作时的分析比逻辑分析要复杂。现在假设每个反相器的门延时为 1ns，每个二输入与门的门延时为 2ns，同或门的门延时为 2ns，忽略连线延时。假设初始条件为 $A=0$，$B=0$，$g_1=1$，$g_2=1$，$g_3=0$，$g_4=0$，$g_5=1$。

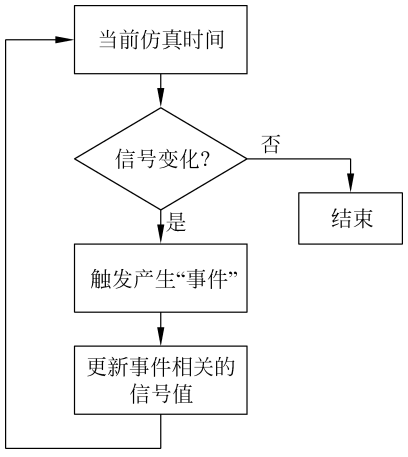
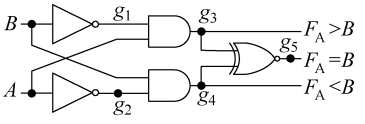
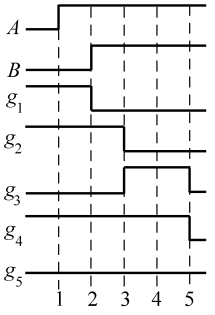


图 3.15　事件处理示意图



(a) 电路图



(b) 仿真波形图

图 3.16　事件处理举例

产生一个事件必须是相关的门有信号发生变化，因此整个仿真过程(列出了 5ns)中每个时刻可能发生的事件如表 3.4 所示。图 3.16(b)是对应的仿真波形图。

表 3.4　事件处理示例中的事件列表

仿真时刻							
0	$(A,0)$	$(B,0)$	$(g_1,1)$	$(g_2,1)$	$(g_3,0)$	$(g_4,1)$	$(g_5,0)$
1	$(A,1)$		$(g_1,1)$	$(g_2,1)$			
2		$(B,1)$	$(g_1,0)$		$(g_3,0)$	$(g_4,1)$	$(g_5,0)$
3				$(g_2,0)$	$(g_3,1)$		
4							
5					$(g_3,0)$	$(g_4,0)$	

从波形图中可以看出，因为门电路的门延时，使得 g_3、g_4 信号在时刻 3 和 4 时同时取 1 值，这显然是不符合电路设计要求的，因此采用这样的一个电路设计其实是不成功的，但电路本身的逻辑却没有问题。此时就需要对电路的时序进行调整以符合电路设计要求。

3.4 常用集成电路逻辑仿真工具介绍

集成电路的逻辑仿真工具有很多，常用的有 Mentor 公司的 ModelSim、Synopsys 公司的 VCS、Altera 公司的 Quartus Ⅱ、Cadence 公司的逻辑仿真工具 Verilog-XL、NC-Verilog 等。此外，常用的静态仿真工具是 Synopsys 公司的 Prime Time，形式验证工具是 Synopsys 公司的 Formality。本节将对以上工具进行简单介绍。

3.4.1 ModelSim 工具

Mentor 公司的 ModelSim 是业界最优秀的 HDL 语言仿真软件之一，它能提供友好的仿真环境，是业界唯一的单内核支持 VHDL 和 Verilog 混合仿真的仿真器。它采用直接优化的编译技术、Tcl/Tk 技术和单一内核仿真技术，编译仿真速度快，编译的代码与平台无关，便于保护 IP 核，个性化的图形界面和用户接口，为用户加快调错提供强有力的手段，是 FPGA/ASIC 设计的首选仿真软件。

其主要特点有：①RTL 和门级优化，本地编译结构，编译仿真速度快，跨平台、跨版本仿真；②单内核 VHDL 和 Verilog 混合仿真；③具有源代码模板和助手，可进行项目管理；④集成性能分析、波形比较、代码覆盖、数据流 ChaseX、Signal Spy、虚拟对象（Virtual Object）、Memory 窗口、Assertion 窗口、源码窗口显示信号值、信号条件断点等众多调试功能；⑤具有 C 和 Tcl/Tk 接口，可进行 C 调试；⑥对 SystemC 直接支持，和 HDL 任意混合；⑦支持 SystemVerilog 的设计功能；⑧具有对系统级描述语言的全面支持，包括 SystemVerilog、SystemC、PSL；⑨具有 ASIC Sign-off 功能；⑩可以单独或同时运行行为级、RTL 级和门级（gate-level）的代码。

ModelSim 有几种不同的版本：SE、PE、LE 和 OEM，其中 SE 是最高级的版本，而集成在 Actel、Atmel、Altera、Xilinx 以及 Lattice 等 FPGA 厂商设计工具中的均是其 OEM 版本。

SE 版和 OEM 版在功能和性能方面有较大差别，比如大家都关心的仿真速度问题，以 Xilinx 公司提供的 OEM 版本 ModelSim XE 为例，对于代码少于 40 000 行的设计，ModelSim SE 比 ModelSim XE 要快 10 倍；对于代码超过 40 000 行的设计，ModelSim SE 要比 ModelSim XE 快近 40 倍。

ModelSim SE 支持 Windows、UNIX 和 Linux 平台，提供全面、完善以及高性能的验证功能，全面支持业界广泛的标准。

3.4.2 VCS 工具

VCS 的全称是 Verilog Compile Simulator，是 Synopsys 公司强有力的电路仿真工具。目前，业界领先的设计人员在从事先进的设计工作时大多选择 Synopsys VCS 作为其时序功能验证工具。实际上，绝大多数的 32nm 及以下工艺的设计均采用 VCS 进行验证。全球顶尖的 20 家半导体公司也大多采用 VCS 作为其主要验证解决方案，VCS 可提供高性能仿真引擎、约束条件解算器引擎、Native Testbench（NTB）支持、SystemVerilog 支持、验证规划、覆盖率分析和收敛以及完整的调试环境。

VCS是编译型Verilog模拟器，它完全支持OVI标准的Verilog HDL语言、PLI和SDF。具有极高的模拟性能，其出色的内存管理能力足以支持千万门级的ASIC设计，而其模拟精度也完全满足深亚微米ASIC Sign-off的要求。VCS结合节拍式算法和事件驱动算法，具有高性能、大规模和高精度的特点，适用于从行为级、RTL到Sign-off等各个阶段的仿真。VCS和Scirocco也支持混合语言仿真，都集成了Virsim图形用户界面，它提供了对模拟结果的交互和后处理分析。

VCS可提供业内领先的性能和容量，同时支持一整套先进的调试、缺陷查找、覆盖率、验证规划和断言技术。其调试技术可以理解验证方法学，并提供对随机约束的调试。VCS的多核技术可在多台多核机器上并行运行设计、测试平台、断言和调试功能，将验证速度提高2倍，缩短验证时间。VCS的分区编译(Partition Compile)流程仅重新编译被修改的代码，缩短用户的迭代编译周期多达10倍。VCS还提供一整套全面诊断工具，包括仿真内存消耗和仿真时间解析、交互式约束调试、智能记录等，帮助用户快速分析问题。VCS支持原生的低功耗仿真和UPF格式，在既有完整的调试手段和高性能仿真的基础之上，可提供创新的电压感知验证技术，定位现代低功耗设计中的缺陷。VCS具有内置调试和可视化环境，支持所有流行设计和验证语言，包括Verilog、VHDL、SystemVerilog、OpenVera、SystemC以及VMM、OVM和UVM等方法学，可帮助用户进行优质的设计。

VCS运行时，首先需要输入编写好的Verilog代码，再将Verilog源文件进行编译，然后生成可执行的模拟文件，也可生成VCD或者VCD+记录文件。接着运行这个可执行的文件，可以进行调试与分析，或者直接查看生成的VCD或者VCD+记录文件。

VCS的运行方式有两种：一种是交互模式(Interactive Mode)，另一种是批处理模式(Batch Mode)。两种方式各有优缺点，具体用在不同的情况下。在测试小模块或者底层模块，情况不太复杂而又需要很详细信息的时候，为显示更为直观，可以采用交互模式。当进行复杂测试而关注于整体性能，仅查看所需的信号仿真结果不必查看每个信号的结果时，可以采用批处理模式。

3.4.3　Quartus Ⅱ工具

Quartus Ⅱ是Altera公司的综合性PLD/FPGA开发软件，内嵌自有的综合器以及仿真器，可以完成从设计输入到硬件配置的完整PLD设计流程。Quartus Ⅱ应用开发工具提供完整的多平台设计环境，它可以轻易满足特定设计的需要，是可编程片上系统设计的综合性环境。Quartus Ⅱ可在个人计算机或UNIX/Linux工作站下使用，大大简便了整个设计过程。

Quartus Ⅱ可以在Windows、UNIX和Linux上使用，除了可以使用Tcl脚本完成设计流程外，还提供了完善的用户图形界面设计方式。具有运行速度快、界面统一、功能集中、易学易用等特点。

Quartus Ⅱ支持Altera的IP核，包含LPM/MegaFunction宏功能模块库，使用户可以充分利用成熟的模块，简化了设计的复杂性，加快了设计速度。对第三方EDA工具的良好支持也使用户可以在设计流程的各个阶段使用熟悉的第三方EDA工具。

此外，Quartus Ⅱ通过和DSP Builder工具与MATLAB/Simulink相结合，可以方便地实现各种DSP应用系统；支持Altera的片上可编程系统(SoPC)开发，集系统级设计、嵌入

式软件开发、可编程逻辑设计于一体，是一种综合性的开发平台。

Altera Quartus Ⅱ作为一种可编程逻辑的设计环境，由于其强大的设计能力和直观易用的接口，越来越受到数字系统设计者的欢迎。

3.4.4 Cadence 公司逻辑仿真工具

Cadence 公司的逻辑仿真工具主要有 Verilog-XL 和 NC-Verilog 两种。两种仿真工具都是基于事件算法的仿真器，即只有电路状态发生变化时才进行处理，只模拟那些可能引起电路状态改变的元件，仿真器响应输入引脚上的事件，并将值在电路中向前传播。

Verilog-XL 是由开创 Verilog HDL 语言的 GDA(Gateway Design Automation)公司的 PhilMoorby 在 1985 年推出的第 3 代商用仿真器，并获得了巨大的成功，由此也使得 Verilog HDL 迅速得到推广应用，因此 Verilog-XL 仿真器是与 Verilog HDL 同时开发的。1989 年，Cadene 公司收购了 GDA 公司，使得 Verilog-XL 成为 Gadence 公司的逻辑仿真工具。Verilog-XL 是一个交互式的仿真器，仿真时先读入 Verilog 描述，进行语义语法检查，处理编译指导；然后在内存中将设计编译为中间格式，将所有的模块和示例组装成层次结构，源代码中的每个元件都被重新表示并能在产生的数据结构中找到；接着再决定仿真的时间精度，在内存中构造一个事件队列的时间数据结构；最后读入、调度并根据时间执行每一个语句。因此 Verilog-XL 仿真器是命令解释仿真器，其中采用了多种加速算法，对每种抽象级描述都能很好地仿真。

NC Verilog 是 Cadence 公司在 Verilog-XL 基础上推出的仿真工具，采用全编译技术，仿真速度、处理能力、编辑能力等都得到很大提升。它把 Verilog 代码编译成 Verilog 程序的定制仿真器，即把 Verilog 代码转换成一个 C 程序，然后再把该 C 程序编译成仿真器，因此它的启动比 Verilog-XL 稍微慢一些，但这样生成的编译仿真器运行得要比 Verilog-XL 的解释仿真器快很多。NC Verilog 的执行有 3 步：①ncvlog(编译)。即编译 Verilog 源文件，按照编译检查语义及语法，产生中间数据；②ncelab(产生可执行代码)。按照设计指示构造设计的数据结构，产生可执行代码，除非对优化进行限制，否则源代码中的元件可能被优化丢失，产生中间数据；③ncsim(对可执行代码进行仿真)。启动仿真核，调入设计的数据结构，构造事件序列，调度并执行事件的机器码。上述 3 个步骤也可以采用基于脚本的单步模式进行。

3.4.5 Prime Time 工具

Prime Time 是 Synopsys 的静态时序分析软件，常被用来分析大规模、同步、数字电路。Prime Time 适用于门级的电路设计，可以和 Synopsys 公司的其他 EDA 软件非常好地结合在一起使用。

作为专门的静态时序分析工具，Prime Time 可以为一个设计提供以下的时序分析和设计检查：①建立和保持时间的检查；②时钟脉冲宽度的检查；③时钟门的检查等。

Prime Time 具有以下特点：①Prime Time 是可以独立运行的软件，它不需要逻辑综合过程中所必需的各种数据结构，而且对内存的要求相对比较低。②Prime Time 特别适用于规模较大的 SoC 的设计。

在数字集成电路设计的流程中，版图前、全局布线之后以及版图后，都可以使用 Prime

Time 进行静态时序分析。它的分析原理是：首先，把整个芯片按照时钟分成许多时序路径(Timing Path)，然后对每条时序路径进行计算和分析。时序路径是指设计中一个点(开始点)到另一个点(结束点)的序列，开始点一般是时钟端口、输入端口或寄存器、锁存器的数据输入引脚等，结束点一般是时钟端口、输出端口或寄存器、锁存器的数据输出引脚等。

3.5　系统验证

前面主要介绍的是集成电路设计中最基本的逻辑仿真知识。一般的功能电路采用这些仿真技术即可以达到确认设计正确与否的目的，但随着微电子技术的迅速发展和系统芯片的出现，集成电路的规模日益庞大，复杂度日益增加，使用传统的方法已经难以完成系统级芯片的设计验证工作，这正是所谓的“验证危机”。在这样的大趋势下，电子设计和验证工具正迅速发生巨大的变革。原先基于 RTL 级的设计和验证方法必须向系统级的设计和验证方法学转变，从而导致了验证语言的出现和标准化。本节先对当前出现的系统级设计和验证语言进行综述，接着简单评述当前的验证方法，说明 UVM 验证方法学是大势所趋。本节并不会介绍如何用 UVM 搭建验证平台，只会结合代码论述 System Verilog 如何搭建验证平台。UVM 正是用 System Verilog 所写，通过学习 System Verilog 搭建的验证平台，为进一步学习 UVM 打下坚实的基础。

3.5.1　验证方法学和验证语言

通过前面的学习，知道了随着集成电路设计规模的日益扩大，现在的集成电路大多是 SoC 的设计，SoC 的特点是：更多、更复杂的功能集成和综合。内部包含存储器、处理器、模拟模块、接口模块和高速、高频输入/输出及软件模块等功能模块或 IP 核。从这些特点中可以看出，SoC 的功能非常复杂，而且因为其内部包含了 IP 核，这些 IP 核往往是设计者从其他设计公司那里购买的，因此对于 IP 核的内部结构不会了解得非常详细。相应地，SoC 的测试也变得越来越复杂。

衡量一个设计的验证效率，通常有两个标准：代码覆盖率和功能覆盖率。代码覆盖率是指验证过程中所能遍历的 RTL 代码执行比例。代码覆盖率包括语句覆盖率、条件覆盖率、状态机覆盖率、信号转换覆盖率等。语句覆盖率指验证时 RTL 语句被覆盖的比例；条件覆盖率指代码中的条件判断语句被验证覆盖的比例；状态机覆盖率是指验证过程中有限状态机的状态变换情况与所有可能的状态变化情况比率；信号转换率是指输入/输出信号 0 状态和 1 状态的切换率。目前大多数逻辑验证工具(如 ModelSim、VCS 等)都可以实现代码覆盖率的数据收集，通过此数据可以掌握代码的执行情况。但即使代码执行率为 100%，也不能保证所有语句的功能都被验证，因此产生了功能覆盖率。功能覆盖率是由验证工程师在对功能规范理解的基础上，人工构筑验证模型、仿真结束后通过分析报告查看没有被覆盖的功能，接着再有针对性地添加测试矢量以对未覆盖的功能进行测试，因此功能覆盖率实际上更能正确反映验证的全面性。

在目前的系统验证中，采用代码覆盖率和功能覆盖率相结合的衡量标准。这是因为虽然功能覆盖率能正确反映验证的全面性，但其是基于人工构建的验证模型进行的，模型的建立又是基于验证工程师对设计所具备的功能规范理解上进行的，每个人对设计规范的理解

不同，因此建立的验证模型也不一样，这样就可能会造成虽然功能覆盖率接近百分百，但因为模型的不全面造成代码的覆盖率并没有达到百分百，因此有必要采取功能覆盖率和代码覆盖率相结合的衡量标准。

进行系统设计时，往往需要多次重复地对设计进行修改，如果每次修改后都采用原先电路设计时的验证方法，则花费的时间会很长，因此如何缩短验证时间，是系统验证的关键问题。目前系统验证一般称为验证方法学，主要是指验证平台的建立和测试方案的制定，而且为了降低验证时间，这个平台和方案必须具有复用性。验证方法学经历了 RVM、AVM、VMM、URM、OVM 和 UVM 等。UVM 是 2011 年由 Accellera 推出的，并得到 Cadence 公司、Synopsys 公司和 Mentor 公司的支持，在包含 OVM 功能的基础上加入了寄存器解决方案，同时也吸取了 VMM 中的一些优点，代表了验证方法学的发展方向，成为目前最流行的系统验证方法学之一。

验证是基于设计的，目前的系统设计多采用 Verilog HDL 语言，相应的验证语言也要基于 Verilog 语言才有普适性。目前有两种常用的基于 Verilog 的验证语言。

一种是基于软件领域的语言和方法，如 C/C++、Java、UML 等。但目前受限于工具，这一类语言主要运用于算法级建模，而在验证时须把这些模型的输出与待验证设计的输出相比，也就是需要将基于 C/C++等语言的模型集成到验证平台中。SystemC 的出现为这一问题的解决提供了良好的语言平台，它提供一组硬件的基本元件，这些元件可以扩充，以便在更高的层次上支持硬件。在 SystemC 2.0 版本之前，有些人认为 SystemC 是侧重于模拟，但是在 SystemC 2.0 之后，这些说法也不准确了。因为现在的 SystemC 2.x 已经能够支持所有系统级的要求。SystemC 填补了传统的 HDL 和基于 C/C++的软件开发方法之间的鸿沟；它包含 C++类库和一个模拟内核，这个内核用来产生行为级和寄存器级的模型；有领先的 EDA 厂家管理和支持，并与商用的综合工具相结合；它支持通用的软件和硬件开发环境。但其最大的优势也是其最大的劣势，在 C++中，用户需要自己管理内存，指针的问题会花费很多精力，并且还存在内存泄漏的问题。基于这些问题，现在 System Verilog 成为业界使用越来越广泛的硬件描述和验证语言。

System Verilog 是基于 Verilog 的，并对其进行了扩展，集中了 Verilog 和 C/C++的优点。自 2002 年开始，经过多次修改。不仅具有面向对象、事务性建模的优点，而且还为验证提供了如约束随机激励产生、功能覆盖率统计分析等功能。此外，因为 System Verilog 是对 IEEE 1364 2001 Verilog 的扩展，因此便于 Verilog 代码设计者和验证者快速上手。与 SystemC 相比，System Verilog 提供 DPI 接口，可以把 C++的函数导入 System Verilog 代码中，而且其内部提供内存管理机制，用户不用担心内存泄漏问题。无论是否是算法类设计，System Verilog 都能完成。System Verilog 得到了 Synopsys、Cadence 和 Mentor 三大 EDA 公司的一致推广和支持，目前被业界很多设计公司应用于设计中，成为目前设计和验证工程师的首选语言。

3.5.2 UVM 简介

UVM 是目前使用最普遍的验证方法学，可以实现对小规模设计、大规模设计及数字系统进行有效且完整的验证。

UVM 提供一个最佳的实现覆盖率驱动验证（Coverage-Driven Verification，CDV）的框架。

CDV 包括自动生成测试矢量、自主检查测试平台、覆盖率指标统计，从而显著缩短验证所花费的时间。CDV 的目的是：允许多次验证以估计验证效率和时间开销；确保使用预先设定的目标，可以完成整体验证；接收前面验证结果中的错误信息，并进行实时分析以简化调试。

CDV 流程与传统直接验证流程是不同的。CDV 中，验证者首先使用一个有组织的规划过程设置验证目标。然后创建一个智能验证平台产生合适激励，并将其发送到 DUT。覆盖率监测器被添加到测试环境来监测验证进程和识别没有被执行的功能，以及检查被添加到 DUT 中的那些不希望实现的功能。覆盖模型和验证平台已经建立好后模拟启动，验证然后加以实现。

通过使用 CDV，人们可以通过改变验证平台参数或者随机种子对设计进行彻底验证，或者在顶层加入测试约束以协调仿真，以便尽早达到验证目的。排队技术允许人们识别当前测试和随机种子对验证目的的贡献度，以便从一个测试队列中删除多余的测试部分。图 3.17 是一个基本的 UVM 验证流程图。

UVM 中一个最重要的原则是建立和利用一个可重用的验证组件。UVM 具有所有验证的特征和能力：①UVM 代码以 OVM 库为基础，仅在被证明可以使用的 OVM 代码顶层单元上进行一些改动；②是开源的；③适用于大量的商业仿真软件。

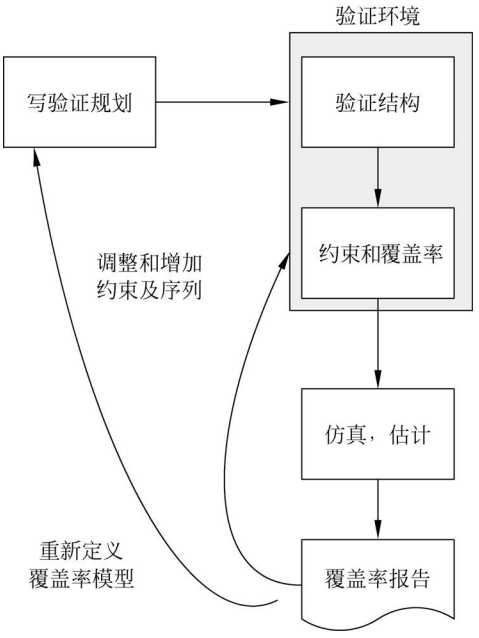


图 3.17　UVM 验证流程图

在技术层面，UVM 提供一个普通的使用模型的面向对象的验证组件，确保所有的验证部件可以在其内部操作，而不管源代码是用什么语言设计的。UVM 的特点如下。

（1）数据设计。通过方法和库代码提供清楚的验证环境分割，使其成为一组特定的数据项和组件。此外，UVM 提供许多内置功能简化活动，如文本打印和图形查看对象、分层次设置和获取对象数据值，以及自动化常用操作，如复制、比较和封装对象等。同时，这使得工程师们的关注对象包含哪些内容以及它们如何工作，而不是普通的关注代码。

（2）激励产生。提供类和框架，使得可以产生用于模型级和系统级的激励时序数据流。使用者能利用现有的环境状态随机验证数据，包括 DUT 状态、接口或者前面的生成数据。UVM 提供内部激励生成功能，它可以自定义包括用户自定义分层处理和产生事务流。

（3）建立和运行测试平台。为一个包含多个不同协议、接口和处理器的 SoC 建立一个完整的测试平台越来越困难，UVM 基本类提供自动化和 UVM 使用帮助。一个定义明确的流程允许建立一个层次化的可重用环境。一个普通接口使得用户可以自定义运行时间和测试平台拓扑而不用修改原始的测试平台。

（4）覆盖率模型设计和检查策略。可以实现验证部件的功能覆盖率检查、物理检查和时间、协议等检查。

UVM平台由可重用的UVM通用验证组件(Universal Verification Component,UVC)组成。一个UVM通用验证组件是为接口协议、一个设计的子模块或者一个软件验证而设置的封装好的、可以重用并可配置的验证环境。每个UVC遵循统一的结构,内部包含为特定协议或设计而设置的一系列用于发送激励、检查及收集覆盖率信息的元素。

UVC标准界面结构包括以下7个元素。

(1) 数据项。

数据项表示传输到DUT的激励,例如网络数据包、总线事务和指令。数据项的域和属性来自于数据项的定义。例如,以太网协议标准为一个以太网数据包定义了有效值和属性。在典型的验证中,产生很多数据项并被传递到DUT。利用System Verilog约束可随机产生大量有意义的测试数据同时最大化覆盖率。

(2) 驱动器。

驱动器是一个一直在运作的实体,把仿真逻辑的数据发送给DUT。一个典型的驱动器通过时序产生器重复产生数据项,并通过采样驱动数据项给DUT的信号。例如,驱动器为一些时钟周期执行写操作控制读写信号、地址总线、数据总线。

(3) 序列产生器。

一个序列产生器是激励产生器,并将激励传给驱动器。默认为一个序列产生器行为与采样激励产生器行为类似,同时可以根据驱动器的请求产生相应的随机应答数据。这些默认行为允许设计者为了控制随机值的分布而增加约束给数据项类。不同于随机化事务队列的生成器,序列产生器包含很多重要的内建特征。主要有以下几个:对每一个数据项的产生针对当前DUT的状态产生响应;获取用户自定义的数据项顺序,形成更有用的激励向量;在可重用情况下具有时间建模能力;对于同一个方案支持断言和程序化约束;系统级同步和控制多个接口。

(4) 监视器。

监视器被动采样DUT信号但不能驱动DUT信号。监视器收集覆盖率信息并用于检查。即使可重用驱动器和序列产生器驱动总线事务,也不能用于收集覆盖率用于检查。监视器执行以下功能:①收集数据项,一个监视器从总线上提取信号信息并传输这些能被其他组件或验证者利用的信息(注意,这些行为可能会被收集器部件执行);②提取事件,监视器可以提取信息的有效性、数据的结构,将一个事务的有效性通知给其他组件;③一个监视器还能捕获对其他组件或验证者有效的信息状态,执行检查和收集覆盖率,验证DUT的输出与协议定义符合,以及选择性地打印跟踪信息。

(5) 收集器。

当驱动激励时,UVM强制进行事务级(序列产生器)和信号级(驱动器)的分离。收集器在监视路径上也进行相应的分离。收集器也是被动的,它为了收集位和字节而遵循特定的协议并形成一个事务。有个端口用于传输收集到的事务给监视器,在这个端口处覆盖率和检查都被执行。此组件不是必需的。

(6) 代理。

序列产生器、驱动器、监视器和收集器都可以单独被重用,但是需要环境整合器来识别它们的名称、角色、配置,以及各自的连接关系。为了降低验证者的工作量,UVM建议环境创建者建立一个抽象的容器——代理。代理可以仿真和验证DUT。代理将驱动器、序列产

生器、监视器和收集器封装在一起。UVC 可以包含一个或多个代理。代理可以是主动的，也可以是被动的。主动模式时代理将激励传给 DUT，被动模式时代理用来监视 DUT 的行为。

（7）环境。

环境是 UVC 的最顶层，它包含一个或多个代理，也可以包含其他组件，如总线监视器。环境内的可配置属性使得用户可以根据重用性自定义其拓扑结构和行为。例如，当验证环境重用为系统验证时，主动模式代理可以变为被动模式代理。

图 3.18 是一个典型的 UVC 接口，代表了一个可重用的验证环境。注意，UVM 里的所有 UVC 包含一个环境级监视器。这些总线级监视器执行检查和功能覆盖率收集，并不是必须和一个单独的代理联系。一个代理监视器能利用通过全局监视器收集到的数据和事务。

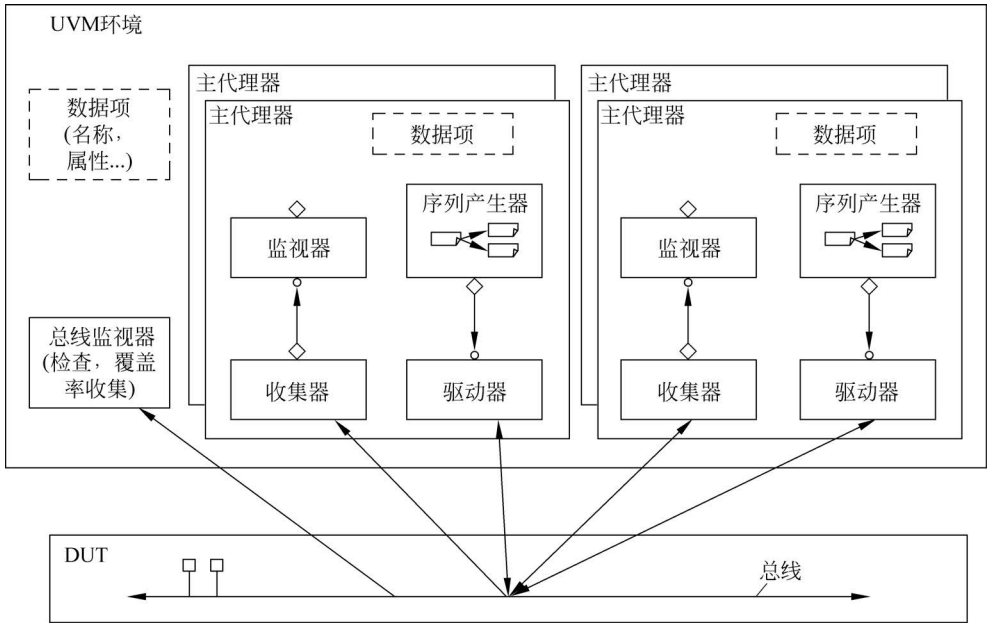


图 3.18　典型的 UVC 接口

3.5.3　基于 System Verilog 的 UVM 类库

UVM 是一个库，在这个库中几乎所有的内容都使用类来实现，验证平台中所有的组件都应该派生自 UVM 中的类。基于 System Verilog 的 UVM 类库为快速搭建基于可重用的验证组件和验证环境提供了基本的结构模块。类库包括基本类、公用程序及宏，组件可以层次化地进行封装和实例化，并且受可扩展的一系列 phases 来初始化、运行和完成每一个测试。这些 phases 在基本类中定义，但不能延伸为特定的项目使用，其层次结构如图 3.19 所示。

（1）uvm_object 是 UVM 中最基本的类，其他的类都来自于它的扩展。它的主要功能是完成创建、复制、封装/拆分、比较、打印和记录等方面的工作。

（2）uvm_report_object 为报告提供了一个接口。通过这个接口，组件在仿真过程中用不同的严重级别发出不同的信息。验证工作者可以通过配置将不同的信息写入不同的文件，或者把所有的信息写入同一个文件。

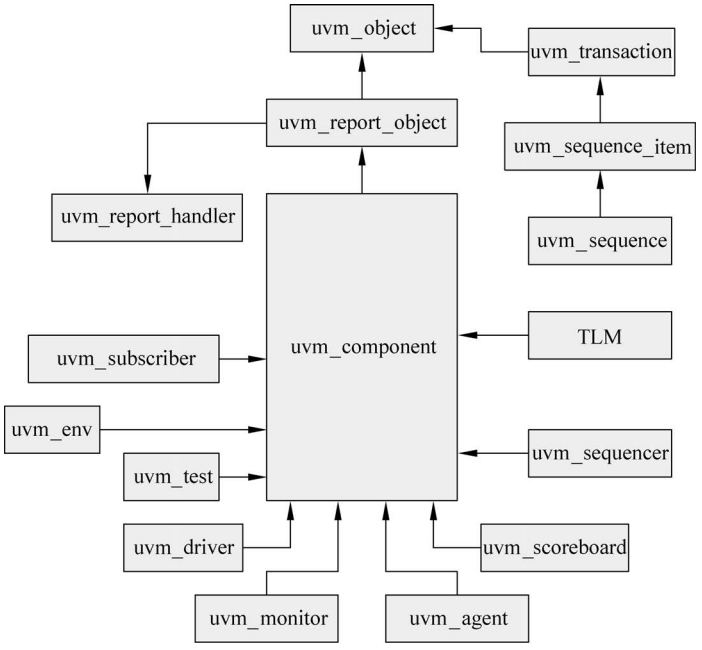


图 3.19 UVM 类的层次结构图

(3) uvm_transaction 是直接被 DUT 处理的数据项。是从 uvm_sequence_item 派生的,使得验证过程中的事务处理可以使用 UVM 中强大的 sequence 机制。transaction 指示在验证环境中哪些字段和方法需要被封装起来以用于通信,驱动器从序列产生器中得到事务处理信息,并且将其转换成端口上的信号。

(4) uvm_sequence_item。从图 3.19 中可以看出,虽然 UVM 中有一个 uvm_transaction 类,但是在 UVM 中,不能从 uvm_transaction 派生一个事务处理,而要从 uvm_sequence_item 派生。事实上,uvm_sequence_item 是从 uvm_transaction 派生而来的,因此,uvm_sequence_item 相比 uvm_transaction 添加了很多额外的字段,从 uvm_sequence_item 直接派生,就可以使用这些新增加的字段。

(5) uvm_sequence。所有的 sequence 要从 uvm_sequence 派生。sequence 就是 sequence_item 的组合。sequence 直接与序列产生器打交道,当驱动器向序列产生器索要数据时,序列产生器会检查是否有 sequence 要发送数据。当发现有 sequence_item 等待发送时,会把此 sequence_item 交给驱动器。

(6) uvm_report_handler 是 uvm_report_object 类的一个子类,用来实现消息的报告。uvm_report_handler 存储了对消息的显示和处理的一些配置信息,它对消息的处理进行决策,并对消息进行一些格式化、过滤等。最终消息将被 uvm_report_handler 送到 uvm_report_server。

(7) uvm_component 也是 UVM 的基类,除了具有 uvm_object 的性质外,还提供如指定父参数以形成层次化结构、内建 phase、报告等功能。uvm_component 是准静态的,在仿真前就必须创建好。

(8) uvm_env 是一个顶层的组件集合,为验证环境的顶层。包含构建一个完整验证环境的所有其他组件,并提供所有组件的阶段控制功能。一个环境类可以作为另一个环境类

的子环境使用。如图 3.20 所示为在 uvm_env 中实例化各类 agent、Reference Model、Scoreboard 等具体功能的组件。同时，uvm_env 又可以作为更顶层的一个组件。例如，当原先用于模块及验证的 UVM 平台需要升级为芯片顶层级的验证平台时，原来模块级的 env 可作为芯片级平台的 env 中的一个实例化的组件。

```
class test_env extends uvm_env;
    spi_master_agent     spi_agent_ist;
    ctr_agent            ctr_agent_ist;
    pw_agent             pw_agent_ist;
    dout_agent           dout_agent_ist;
    scoreboard           sb_ist;
    reference_model      rm_ist;
```

图 3.20　uvm_env 举例

(9) uvm_test 类并没有额外的功能添加，使用此类定义测试的优点是测试例的选择可以在命令行选项完成。在验证过程中，有各种不同的 test case 用于测试不同的功能。如某些 test case 用于测试复位功能，某些 test case 用于测试是否能正常上电，某些 test case 用于测试是否能正常下电等。UVM 中这些 test case 都派生自 uvm_test 类。

(10) uvm_driver 通过向序列产生器获取 sequence_item 信息，并将其传输给 DUT 端口上。uvm_driver 能够将在 UVM 中以 transaction 形式存在的激励信息转换为 DUT 端口能够接收的形式，进而将激励信息驱动到 DUT 中。

(11) uvm_monitor 的功能正好与 uvm_driver 的功能相反，uvm_monitor 用于接收 DUT 端口的信息，并将其转换为能够在 UVM 中流通的 transaction 信息。uvm_monior 类似于一个监视器，会不停地监测 DUT 输入/输出端口的信息。其中，监测到的 DUT 输入信息最终会输送到 reference model，监测到的 DUT 输出信息最终会输送到 scoreboard。

(12) uvm_scoreboard 为 UVM 中的比较器。作比较的数据有两个来源：一是来自于 DUT 的输出端口，也即需要检验正确与否的数据；二是来自于 reference model，该数据为 DUT 的理论输出值。当 scoreboard 的数据对比出错时，一般认为 DUT 的输出有问题。

(13) uvm_agent。一般以不同协议写进 DUT 的激励信息都有一个特定的 sequencer、driver 以及 monitor，如图 3.21 所示。当所验证的 DUT 需要该类激励信息时，需要在 env 中实例化这几个组件。为了增强 UVM 平台的可移植性及扩展型，通常将该组激励信息的 sequencer、driver 以及 monitor 封装打包在一个代理里面，而在 env 中实例化相应的代理。这样如果后续的项目当中不需要这类激励信息，只需取消相应的代理的实例化即可。

(14) uvm_sequencer 是 UVM 中用于产生激励的组件，uvm_sequencer 的参数即为相应的 transaction 类型，如图 3.22 所示。

```
class ctr_agent extends uvm_agent;
    ctr_driver       ctr_drv;
    ctr_monitor      ctr_mon;
    ctr_sequencer    ctr_sqr;
```

图 3.21　uvm_agent 举例

```
class ctr_sequencer extends uvm_sequencer #(ctr_trans);

    `uvm_component_utils(ctr_sequencer)

    function new(string name = "ctr_sequencer",uvm_component parent);
        super.new("ctr_sequencer",parent);
    endfunction

endclass
```

图 3.22　uvm_sequencer 举例

(15) TLM(Transaction Level Modeling)即事务级建模，为 UVM 中数据通信的一种方式。TLM 中有几种常用的操作：①Put 操作。通信的主动方 A 将一个 transaction 发送给被动方 B。②Get 操作。通信的主动方 A 向被动方 B 索取一个 transaction。③Transport 操作。通信的主动方 A 先向 B 发送一个请求，再向 B 索要一个 transaction。通信过程中，通信主动发起方 A 端的端口称为 PORT，被动方 B 端的端口称为 EXPORT。

3.5.4 UVM 举例

本节主要以 3 路抢答器电路为例，给出 UVM 的基本代码。抢答器的基本功能为：满足 3 名抢答者参加比赛的需要，3 名抢答者分别控制着 3 个按钮（即有 3 个输入端）；某个抢答者按下按钮后，抢答器的输出端显示其对应的号码（即有 1 个输出端）；结束 1 次抢答后，抢答器恢复初始态（即需要 reset 等控制信号）。

抢答器的 Verilog HDL 功能代码本节不提供，读者可以自行设计或者参考其他资料。

（1）env。在其内部包含 4 个 agent，其中 ctr_agent 是 rtl 的输入控制信号，resp1_agent、resp3_agent、resp3_agent 分别为 3 名抢答者的输入端。此外，还包含 1 个 rtl 输出检查端 dout_monitor、1 个数据比较端 resp_scoreboard、1 个 reference modelresp_rm。env 部分代码如下。

```
class responder_env extends uvm_env;
    ctr_agent           ctr_ugent_ist;
    resp1_agent          resp1_agent_ist;
    resp2_agent          resp2_agent_ist;
    resp3_agent          resp3_agent_ist;
    dout_monitor          data_mon_ist;
    resp_scoreboard       resp_sb;
    resp_rm               resp_rm_ist;

    `uvm_component_utils(_env)

    …
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        ctr_agent_ist = ctr_agent::type_id::create("ctr_agent_ist",this);
        resp1_agent_ist = resp1_agent::type_id::create("resp1_agent_ist",this);
        resp2_agent_ist = resp2_agent::type_id::create("resp2_agent_ist",this);
        resp3_agent_ist = resp3_agent::type_id::create("resp3_agent_ist",this);
        dout_mon_ist = dout_monitor::type_id::create("dout_mon_ist",this);
        resp_rm_ist = resp_rm::type_id::create("resp_rm_ist",this);
        resp_sb = resp_scoreboard::type_id::create("resp_sb",this);
      …
    endfunction
    …
endclass
```

（2）monitor。此处仅给出监测 DUT 输出结果的 monitor。monitor 部分代码如下。

```
class dout_monitor extends uvm_monitor;
    dout_trans  mon_tr;
    uvm_component_utils(dout_monitor)

    extern function new(string name = " ",uvm_component parent);
    extern virtual function void build_phase(uvm_phase phase);
    extern virtual task main_phase(uvm_phase phase);
endclass

function dout_monitor::new(string name = " ",uvm_component parent);
```

```
  super.new(name,parent);
endfunction

function void dout_monitor::build_phase(uvm_phase phase);
  super.build_phase(phase);
    dout_ap = new("dout_ap",this);
    …
endfunction

task dout_monitor::main_phase(uvm_phase phase);
  super.main_phase(phase);
  …
  dout_ap.write(mon_tr);
endtask
```

(3) reference model。用于验证抢答器是否正确的参考模型。reference model 部分代码如下。

```
class resp_rm extends uvm_component;
  `uvm_component_utils(resp_rm)
  uvm_analysis_port#(dout_trans) rm_scb;

  function new(string name,uvm_component partent);
  super.new(name,parent);
  …
  endfunction: new

  function void build_phase(uvm_phase phase);
  super. build-phase(phase);
  rm_scb = new("rm_scb",this);
  …
  endfunction

  task main_phase(uvm-phase phase);
    …
    rm_scb.write(dout_tr);
  endtask

endclass
```

(4) scoreboard。用于 DUT 和 reference model 之间的对比输出。scoreboard 部分代码如下。

```
`uvm_analysis_imp_decl(_rmdout)
`uvm_analysis_imp_decl(_mondout)
class resp_scoreboard extends uvm_scoreboard;

      dout_trans rm_dout_q[$];
      dout_trans dout_tr;

      uvm_analysis_imp_mondout#(dout_trans,resp_scoreboard) mondout_ imp;
      uvm_analysis_imp_rmdout#(dout_trans,resp_scoreboard) rmdout_imp;

      `uvm_component_utils(resp_scoreboard);
```

```
    function new(string name = "resp_scoreboard",uvm_component parent);
        super·new(name,parent);
        mondout_imp = new("mondout_imp",this);
        rmdout_imp = new("rmdout_imp",this);
    endfunction

    function void write_mondout(input dout_trans mon_tr);
       compare_data_data(mon_tr);
    endfunction

    function void write_rmdout(input dout_trans dout_tr);
        rm_dout_q.push_back(dout_tr);
    endfunction

    function void compare_data_data(input dout_trans dout_tr);
        rm_tr =  rm_dout_q.pop_front();
        …
    endfunction
endclass
```

(5) agent。仅给出一个 agent1 的代码,其余 agent 类似。agent1 部分代码如下。

```
class respl_agent extends uvm_agent;
    respl_driver     resp1_drv;
    respl_sequencer   resp1_sqr;
    respl_monitor     resp1_mon

uvm_analysis_port#(resp1_trans) ap;

`uvm_component_utils(resp1_agent)

function  new(string name = "resp1_agent",uvm_component parent);
    super.new(name,parent);
    ap = new("ap",this);
endfunction

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
        resp1_drv = resp1_driver::type_id::create("resp1_drv",this);
        resp1_sqr = resp1_sequencer::type_id::create("resp1_sqr",this);
endfunction

function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    resp1_drv.seq_item_port.connect(resp1_sqr.seq_item_export);
    this.op = resp1_mon.resp1_ap;
endfunctian
endclass
```

(6) driver。与 agent1 对应的 driver,其余 driver 类似。driver1 部分代码如下。

```
class resp1_drier extends uvm_driver #(resp1_trans);
    virtual respllif    resp1_intf;
    resp1_trans          resp1_tr;
```

```
    `uvm_component_utils_begin(resp1_driver)
    `uvm_field_object(resp1_tr,UVM_ALL_ON);
    `uvm_component_utils_end

    function new(string name = "resp1_driver",uvm_component parent);
        super.new(name,parent);

    endfunction
    …
    task main_phase(uvm_phase phase);
      super.main_phase(phase);
        while(1)begin
           seq_item_port.get_next_item(resp1_tr);
           send_ data(resp1_tr);
           seq_item_port.item_done();

          end
    endtask

    task send_data(resp1_trans resp1_tr);
      resp1_intf.req <= resp_tr.req;
      …
    endtask
endclass
```

习　　题

1. 简述集成电路设计有哪些描述域,有哪些描述方式。
2. 集成电路设计的层次一般有哪些？各个层次的内容是什么？
3. 集成电路设计验证一般要完成什么工作？
4. 仿真与形式验证有什么不同？
5. 什么是逻辑仿真？
6. 静态时序分析起什么作用？其优点是什么？
7. 静态时序分析的时序路径分类有哪些？
8. 简述建立事件和保持事件的检查工作。
9. 数字集成电路仿真模型通常有哪几种类型？
10. 简述仿真中的事件处理过程。
11. 常用的逻辑仿真工具有哪些？
12. 什么是验证方法学？
13. UVM 验证平台主要由哪些部分构成？每部分的主要作用是什么？

参 考 文 献

[1] Bhatnagar H. 高级 ASIC 芯片设计[M]. 张文俊,译. 北京：清华大学出版社,2007.
[2] 边计年,薛宏熙,苏明,等. 数字系统设计自动化[M]. 2 版. 北京：清华大学出版社,2006.

[3] 金西. 数字集成电路设计[M]. 合肥：中国科学技术大学出版社，2013.
[4] 唐杉，徐强，王莉薇. 数字 IC 设计方法、技巧与实践[M]. 北京：机械工业出版社，2006.
[5] 周润德. 数字集成电路——电路、系统与设计[M]. 北京：电子工业出版社，2011.
[6] Brunvand E. 数字 VLSI 芯片设计——使用 Cadence 和 Synopsys CAD 工具[M]. 周润德，译. 北京：电子工业出版社，2009.
[7] 潘中良. 数字电路的仿真与验证[M]. 北京：国防工业出版社，2006.
[8] Universal Verification Methodology (UVM) 1.0 User's Guide, Accellera.