# 第5章 Chapter 5

# 数字集成电路前端设计 EDA

**本章学习目标**

- 理解 EDA 工具在数字前端设计中的核心作用。
- 掌握 UVM 验证方法学的工程实现。
- 精通 Verilog 可综合编码规范。
- 实践 VCS 仿真全流程技术栈。
- 实施 Design Compiler 综合优化策略。
- 解析数字仿真事件队列原理。

## 5.1 数字集成电路前端设计流程

数字集成电路设计流程遵循严格的层次化工程范式(图 5.1),其核心开发阶段可划分为前端逻辑设计与后端物理实现两大技术域。传统上以逻辑综合作为关键分水岭——前端设计聚焦寄存器传输级抽象至门级网表的转换过程,涵盖系统架构定义、功能验证、时序约束优化等工艺无关设计活动;后端设计面向物理设计规则与制造参数,执行布局布线、时钟树综合、设计规则检查等工艺强相关优化。本章重点解析前端设计领

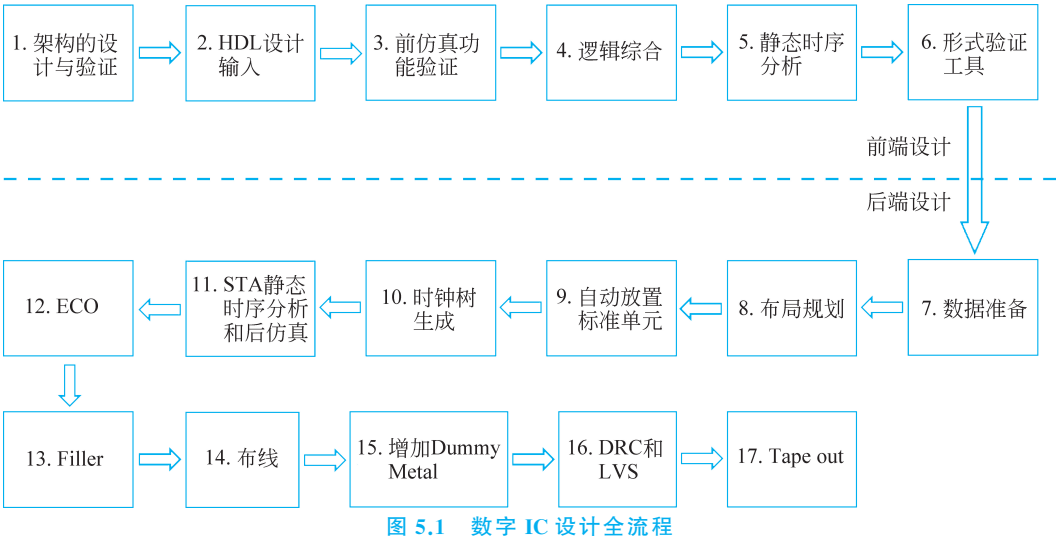


图 5.1 数字 IC 设计全流程

域的下列关键技术矩阵：基于 SystemVerilog/UVM 的验证方法学构建可重用测试平台，通过形式化验证以确保 RTL 代码与规格说明的一致性；采用约束驱动综合实现时序路径的拓扑优化与功耗预算分配；依托静态时序分析在签核阶段验证建立/保持时间约束的全局满足。值得注意的是，现代先进工艺节点下的前端设计已突破传统界限，需通过物理感知综合预判互连寄生效应，并与后端设计工具形成时序/功耗/面积的闭环优化迭代，这种设计阶段前移的协同策略显著降低了纳米级芯片的开发风险并缩短了迭代周期。

### 5.1.1 前端设计步骤

前端设计的各步骤简介如下。

**1. 架构的设计与验证**

芯片架构设计与验证环节应当基于市场调研结论与客户需求，由产品经理或系统架构师主导完成芯片功能框架构建及系统级参数定义。此阶段需明确芯片核心功能架构、性能指标参数及外部接口规范，为芯片整体能效比与功耗特性奠定技术基线。核心目标在于通过建立高抽象层算法模型，形成软硬件协同验证基础，同时依托架构级建模分析为后续 RTL 级实现提供全局约束与设计指引。具体实施流程包括系统功能解耦与规范细化：首先，架构师需对系统功能进行模块化分解，形成具备量化参数的设计规格文档，涵盖物理实现约束（工艺制程节点选择、裸片面积预算）、性能指标（运行速率、功耗阈值）及功能规范（模块功能定义、接口协议标准）等关键要素；其次，基于系统建模语言完成模块级行为描述与跨域仿真，验证架构方案的可行性。该阶段通常采用 MATLAB 算法验证、C++ /C 系统建模以及 SystemC/SystemVerilog 硬件功能描述等工具链的组合，构建可执行参考模型以实现架构原型的功能闭环验证。

**2. HDL 设计输入**

HDL 设计输入环节承担着将系统级架构转化为可综合电路实现的重要职能，该阶段需采用硬件描述语言对电路功能进行代码化描述与实现，完成自系统级功能描述至 Verilog HDL 代码实现的技术转化。工程师需依托专业文本编辑器工具链（如 Vim）完成 HDL 代码的规范化编写，在确保语法合规性与功能逻辑正确性的基础上，同步考量逻辑综合约束条件与功耗优化指标等工程化要素，通过多维度协同设计实现代码质量与物理实现可行性的双重验证。

**3. 前仿真功能验证**

前仿真功能验证阶段作为数字电路设计流程中的关键质量保障环节，其核心目标在于从功能层面验证 RTL 代码实现的正确性，确保电路行为完全符合设计规格书（Specification）所定义的功能约束。该过程通过在零延迟理想化假设条件下，依托 VCS、NC-Verilog、Modelsim 等商用仿真工具构建动态仿真环境，结合 SystemVerilog 断言验证与 SystemC 事务级建模技术，并基于通用验证方法学（UVM）构建分层验证平台。通过部署海量定向测试用例与随机激励组合，系统性检测由设计人员编码疏漏或架构规划缺陷引发的功能性异常。验证有效性主要依托功能覆盖率指标进行量化评估，包括但不限于代码覆盖率、分支覆盖率及断言覆盖率等维度。需要特别指出的是，前仿真作为典型的动态验证方法，受限于仿真时长与测试向量的组合爆炸问题，存在场景覆盖完备性

不足的固有局限性，需通过形式验证等补充手段实现验证闭环。

**4. 逻辑综合**

逻辑综合阶段承担着将 HDL 代码转化为特定工艺库约束下的门级网表的技术转化过程。该过程严格依赖半导体代工厂提供的工艺参数标准单元库作为综合约束基础，不同工艺节点的标准单元在驱动强度、传输延迟等时序参数以及单元占用面积等物理参数上呈现显著差异，这些关键参数将直接决定综合后的电路的时序收敛特性和物理实现密度。工程师需通过配置综合策略文件，在满足时序约束、功耗预算与面积指标的多维优化框架下，完成寄存器传输级描述到工艺相关门级电路的结构映射与逻辑优化。

**5. 静态时序分析**

静态时序分析作为集成电路物理实现流程中的核心验证环节，其核心目标在于独立于输入激励信号特征，通过数学建模方法确保所有时序路径的传输延迟均满足设计规范要求。该分析方法基于工艺库提供的标准单元时序参数与互连寄生模型，系统化校验电路在指定工作条件下建立时间(Setup Time)与保持时间(Hold Time)的约束合规性。自逻辑综合阶段起，设计流程中每个关键节点(如布局规划、时钟树综合等)均需执行多轮静态时序分析，通过迭代优化实现时序收敛目标。业内的主流工具链以 Synopsys PrimeTime 为代表，其在布图规划前阶段采用工艺库线负载模型进行互连延迟预估，待物理设计完成后，通过寄生参数反标实现纳米级精度的延迟计算。需要着重指出的是，尽管前端设计阶段可通过 RTL 代码调整快速修复时序违例，但由此引发的功能逻辑变更需重新执行全量仿真验证流程，其验证周期可能长达数个月，因此实际工程实践中通常将时序优化重心后置于物理实现阶段，以此平衡设计效率与质量管控的双重要求。

**6. 形式验证工具**

形式验证工具作为集成电路设计流程中功能等价性校验的核心环节，通常被定义为逻辑等价性检查，其核心使命在于通过数学形式化方法验证逻辑综合生成的门级网表与原始 HDL 源代码之间的功能一致性，确保工艺映射过程未引入非预期的电路功能偏差。该技术体系依托 Formality、Cadence Conformal、Mentor Questa Formal 等 EDA 工具链，凭借其具备高效性、验证路径全覆盖性及自动化程度高等显著优势，可系统化消除人工检查疏漏的风险。对于中小规模集成电路设计，基于测试向量的逻辑仿真仍可作为补充验证手段；而在 VLSI 场景下，形式验证因其具备多项式时间复杂度级别的验证效率，已成为保障 RTL 代码与综合后网表功能等价性的工业级标准方法。通过集成此类工具，设计团队能够在避免传统仿真耗时数个月的全向量覆盖测试的前提下，高效完成设计迭代周期内的功能保真度验证，为后续物理实现与流片环节构筑可靠的质量基线。

## 5.1.2 搭建 UVM 测试系统启动方法

芯片验证是贯穿集成电路研发全流程的关键环节，旨在通过系统化仿真手段确保设计在各阶段的功能正确性，规避流片风险。本节重点阐述 RTL 级功能验证：随着工艺演进与设计复杂度提升，验证工作量已占开发周期的 50%～80%，成为制约芯片研发效率的主要瓶颈。为应对快速上市需求，现代验证方法学强调系统化、可复用策略，包括受约束随机测试激励生成、功能覆盖率量化分析、事务级分层验证架构及可扩展的验证平台构建。验证工具链的选择直接影响效率。其中，通用验证方法学(Universal Verification

Methodology,UVM)基于 SystemVerilog 语言构建,提供标准化验证平台模板与完备源码库,支持覆盖率驱动的自动化验证流程。UVM 的核心价值体现为代码高度可复用性及平台跨项目移植性:通过模块化组件和预定义接口,工程师可快速调用基础验证单元并进行定向扩展;其平台架构虽然需根据具体设计调整参数配置,但核心层次结构具备通用性,初次搭建后可通过组件重构实现多项目复用,显著降低了验证环境的开发成本,并提升了迭代效率。图 5.2 展示了一个典型的 UVM 平台结构。

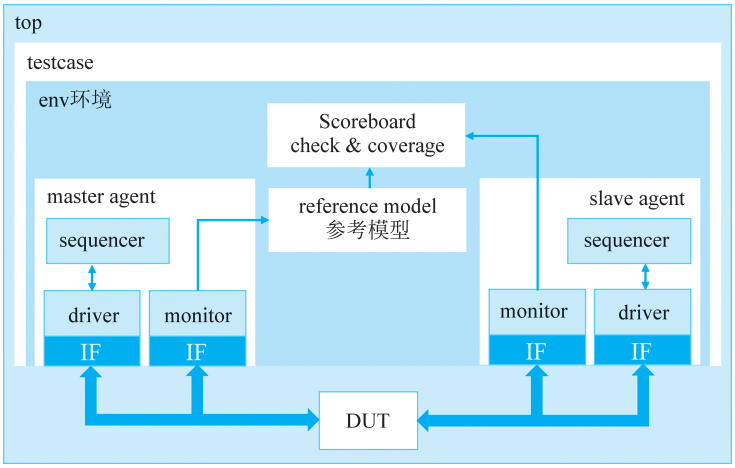


图 5.2 典型的 UVM 平台结构

### 1. Makefile 启动

以经典的 helloworld 工程为例,需按功能划分建立 5 个初始为空的核心目录:docs(存放项目文档与说明文件)、opt(存放编译输出及仿真结果)、sims(配置仿真启动脚本与配置文件)、src(管理 RTL 级设计源代码)及 tb(构建测试平台验证环境),以此实现工程资源的规范化分类管理。具体实施流程包含以下标准化操作步骤。

(1) 在 src 文件夹中创建 helloworld.sv 文件,注意扩展名,并写好文件的头,方便查阅和增加可读性。

【例 5.1】

```
1   //仿真专用模块,不可综合
2   module helloworld (
3      input key
4   );
5      always @ (key) begin                          //电平敏感,捕捉按键变化
6         if (key) begin                             //按键按下时打印
7            $display("Hello World! Time = %0t ns", $time);
8         end
9      end
10  endmodule
```

(2) 在 tb 目录下新建 TB.sv 文件,构建测试平台顶层架构。

(3) 在 sims 目录中新建 filelist.f 与 Makefile 文件,既支持新建空白配置文件,也可将既有预置脚本文件直接导入至该目录。

(4) 查阅 docs/README.md 文档以获取完整的参数配置说明,涵盖仿真库路径定

义、输出目录设置、仿真优化参数配置、运行参数调整及第三方库文件引用等关键信息。

(5) 进入 sims 目录并编辑 Makefile 文件。

【例 5.2】 典型配置示例：

```
#环境变量定义(需在外部或 Makefile 首部定义)
#默认目标(顺序执行)
all: compile vsim

#初始化库
lib:
    @echo "start compile for Questasim10.6c"
    vlib $(WORK)
    vmap work $(WORK)

#编译 RTL
vlog: lib
    vlog -incr -sv -work $(WORK) -f filelist.f -l \
$(OUTPUT)/compile.log

compile: vlog

#优化仿真模型
vopt:
    vsim $(VSIMOPT) -L $(WORK) -L $(LIB0) -L $(LIB1) \
-L $(LIB2) -work $(WORK) $(WORK).TB -o top

#执行仿真
sim: vopt
    vsim $(VSIMBATCH) -lib $(WORK) top -l $(OUTPUT)/simulate.log

#联合目标
vsim: sim

#清理
clean:
    rm -rf $(WORK) modelsim.ini * .wlf $(OUTPUT)/* .log
```

仿真流程启动前需完成全部前置配置工作，其标准执行流程遵循“编译→仿真→结果解析”三阶段范式，具体操作如下。在终端界面调用 make 指令触发自动化构建脚本，该命令将依次执行以下动作：首先基于 filelist.f 文件清单完成设计文件与验证组件的编译；其次启动仿真器加载测试向量并运行；最后生成波形文件及覆盖率报告至指定输出目录。

### 2. 脚本启动

在构建脚本驱动的仿真系统时，除基础设计文件(sv)、文件清单(f)与构建脚本(Makefile)外，需在 sim 目录中创建以下关键脚本文件以完善自动化流程 compile.do、run.do、sim.do 和 run.bat(Windows)或 run.sh(Linux)。

【例 5.3】

```
#编译阶段脚本 compile.do
#清理环境
```

```
quit -sim
if {[file exists work]} {
    vdel -all -lib work
}
file delete -force modelsim.ini
file mkdir ../opt

#初始化库
vlib work
vmap work work

#编译 RTL 文件
vlog -incr -sv -work work -f filelist.f -l ../opt/compile.log

#仿真阶段脚本 sim.do
#优化设计(启用全路径访问)
vopt +acc work.TB -o top_opt \
  -L work \
  -L "D:/mentor_libs/tech_lib" \
  -L "D:/mentor_libs/ip_lib"

#启动仿真并抑制数值警告
vsim -voptargs="+acc" work.top_opt -l ../opt/simulate.log
set NumericStdNoWarnings 1

#运行仿真至结束
run -all
quit

#run.do
onbreak {exit -code 1}

#编译阶段
if {[file exists work]} {vdel -all}
vlib work && vmap work work
do compile.do

#仿真阶段
do sim.do
run -all
exit -code 0

#compile.do
vlog -sv -f filelist.f -l $::env(LOG_DIR)/compile.log

#sim.do
vopt +acc work.tb -o opt_tb
vsim -voptargs="+acc" work.opt_tb

:: run.bat
@echo off
set "QUESTA_BIN=D:\Mentor\questasim\win64\vsim.exe"
set "LOG_DIR=logs"
```

```
mkdir %LOG_DIR% 2>nul
call "%QUESTA_BIN%" -c -do "do run.do" -l %LOG_DIR%\sim.log
exit /b %errorlevel%
```

### 3. Linux 系统环境搭建测试平台

在 Linux 系统环境下构建测试平台时，需通过 VCS 工具对 helloworld 程序进行功能验证与编译，以确认工具链的基础运行能力。随后，应在工作区内按标准规范创建源码、仿真及日志目录结构，并通过终端执行 code workspace/指令加载工程至 Visual Studio Code 集成开发环境，从而为后续的代码编写、调试分析及版本控制提供统一的操作界面。应通过环境变量定义工程根目录路径，确保多终端操作时文件引用的全局一致性。

【例 5.4】

```
#切换到用户主目录下的项目目录(示例路径,需替换实际路径)
project_dir="/home/your_username/project" #← 修改为您的实际用户名
mkdir -p "$project_dir" && cd "$project_dir" || exit 1

#打印当前工作目录
echo -e "\n\033[34mCurrent directory:\033[0m"
pwd

#创建标准化目录结构(忽略已存在目录)
echo -e "\n\033[32mCreating directory structure...\033[0m"
mkdir -p {tb,src,sims,opt,docs}

#验证目录结构
echo -e "\n\033[32mDirectory contents:\033[0m"
ls -al

#尝试启动 VSCode(若未安装会提示)
if command -v code &> /dev/null; then
    echo -e "\n\033[36mLaunching VSCode...\033[0m"
    code .
else
    echo -e "\n\033[31mError: VSCode not found. Install \
or add to PATH.\033[0m"
    exit 1
fi

#Makefile 脚本
#VCS 编译目标(添加-debug_acc+all 以支持 DVE 调试,-fsdb 生成波形)
vcs:
    mkdir -p ../04_outputs && \
    vcs -sverilog +v2k -timescale=1ns/1ps -full64 \
        -debug_acc+all -kdb -fsdb -lca \
        -LDFLAGS "-Wl,--no-as-needed" \
        -f filelist.f \
        -l ../04_outputs/compile.log

#仿真目标(依赖 VCS 编译结果)
sim: vcs
```

```
    ./simv -l ../04_outputs/sim.log +fsdb+autoflush

#清理所有生成文件和日志(强制删除避免报错)
clean:
    rm -rf AN.DB DVEfiles csrc simv * .simv * .daidir ucli.key
    rm -f * .vpd * .fsdb novas.* verdiLog
    rm -f ../04_outputs/* .log
    @echo "Clean completed."

.PHONY: vcs sim clean

#执行流程验证
#完整编译+仿真流程
make vcs && make sim

#快速清理(查看删除文件列表)
make clean
```

## 5.2 VCS 软件工具

### 5.2.1 层次化事件队列

数字芯片验证需特别注意不同 EDA 工具的差异，不同工具及版本对 IEEE 1364 标准的实现存在差异，这要求通过多工具交叉验证和版本控制确保设计可靠性。Verilog 仿真器的层次化事件队列机制包含 Active、Inactive、Nonblocking 等多个区域，该机制严格规定了阻塞赋值(立即生效，引发顺序依赖)与非阻塞赋值(RHS-LHS 分阶段更新，保持时序一致性)的执行时序差异。RTL 编码阶段必须遵循“时序逻辑用非阻塞，组合逻辑用阻塞”的设计原则，并通过 VCS 的-debug_acc+all 选项配合波形比对进行跨平台调试，以规避因仿真器事件调度差异导致的潜在设计风险。

【例 5.5】

```
always @ (posedge clk) begin
    q1 = d;
    q2 = q1;
    q = q2;
end
```

从硬件综合的视角分析，采用非阻塞赋值模式的代码结构在综合阶段通常仅映射为单级寄存器结构。该特性与仿真器分层事件队列(Hierarchical Event Queue)的时序调度机制存在直接对应关系：在非阻塞赋值区域(Nonblocking Assignment Region)中，右值表达式(RHS)的计算结果需经历事件队列的多级缓冲与同步，最终在时钟有效边沿触发后完成左值(LHS)的更新，这种时序特性直接映射为硬件实现中的寄存器锁存行为。

【例 5.6】

```
  //修正后的同步时序逻辑代码
always @(posedge clk) begin
```

```
    q1 <= d;                                    //非阻塞赋值确保并行更新
    q2 <= q1;                                   //前级寄存器的值传递到下一级
    q  <= q2;                                   //形成三级流水线结构
end
```

采用非阻塞赋值编码范式不仅能够确保仿真模型与设计意图的一致性，还能有效保障综合网表的功能完整性。由此表明，通过构建包含代码语义分析、综合电路映射验证以及事件队列动态追踪的三维分析框架，可系统化解析阻塞与非阻塞赋值在时序敏感场景下的行为特征差异，进而深度掌握 Verilog 语言规范的核心设计哲学。要特别注意的是，在跨并行执行的 always 过程块中对同一目标变量实施重复赋值操作（含非阻塞赋值类型）时，由于 Verilog 标准未定义过程块执行时序，将导致仿真器在 Active Region 内产生非确定性执行路径。这种语义不确定性在硬件实现层面将映射为潜在的时序竞争风险，可能引发寄存器建立/保持时间违例等电路级故障，这从本质上揭示了建立多维度的设计验证方法论的必要性。

## 5.2.2 VCS 仿真基础

Synopsys VCS 仿真工具支持多语言混合设计流程，包含 Verilog、SystemVerilog 设计以及通过 C/C++ PLI 接口的扩展功能，其处理机制通过语言转换层将硬件描述代码编译为中间 C 语言格式，经二次编译生成目标文件并动态链接 PLI 库至 simv 可执行文件。使用该工具须具备数字集成电路设计理论基础、Linux/UNIX 操作系统操作能力及文本编辑工具应用经验。仿真流程分为编译与执行两个阶段：首先需组织 RTL 设计文件与测试激励文件，通过命令行参数（如－l 指定日志存储、＋v2k 启用标准兼容模式、-debug_all 生成调试数据）完成代码编译；随后执行 simv 启动仿真进程，通过 $display 函数输出验证结果，辅以-gui 参数调用波形分析界面进行时序验证。针对多模块复杂工程，推荐采用-f 参数配合工程文件（f 格式）实现批量编译，避免人工输入错误。此外，可通过构建 Makefile 自动化流程，其语法结构以目标依赖关系为核心，采用 Tab 缩进定义编译规则，支持反斜杠换行实现长指令格式化编排，有效提升工程管理效率。

【例 5.7】 VCS 仿真流程自动化 Makefile：

```
#功能:支持多模块工程编译、仿真执行、波形调试及工程清理
#编译器参数定义
VCS_OPTS = -full64 \                            #启用 64 位编译
           +v2k \                               #支持 Verilog-2001 标准
           -debug_all \                         #生成全量调试信息
           -timescale=1ns/1ps \                 #定义仿真时间单位
           -LDFLAGS -Wl,--no-as-needed          #解决动态库链接问题

#工程文件管理(推荐使用 f 文件组织设计)
FILE_LIST  = file_list.f                        #存放 RTL/Testbench 文件路径
SIMV       = simv                               #生成的可执行文件
LOG_DIR    = ./logs                             #日志文件目录
WAVE_DB    = wave.fsdb                          #波形数据库文件

#默认目标:显示帮助信息
help:
```

```
    @echo "---------------- VCS 仿真流程帮助 ----------------"
    @echo "make compile  : 编译 RTL 代码并生成仿真程序"
    @echo "make sim      : 执行仿真并生成波形(自动编译)"
    @echo "make debug    : 启动 DVE 波形调试界面"
    @echo "make clean    : 清理所有生成文件"
    @echo "-------------------------------------------------"

#编译目标:生成仿真可执行文件
compile:
    mkdir -p $(LOG_DIR)
    vcs $(VCS_OPTS) \
        -f $(FILE_LIST) \              #读取文件列表
        -l $(LOG_DIR)/compile.log \
        -P ${VERDI_HOME}/pli.tab \     #链接 PLI 接口库(示例路径)
        -top tb_top                    #指定测试平台顶层模块

#仿真执行:运行仿真并记录波形
sim: compile
    ./$(SIMV) \
        +fsdb+autoflush \              #实时刷新波形文件
        -l $(LOG_DIR)/sim.log \
        +vcs+flush+log \               #强制刷新日志
        +define+DUMP_WAVE              #宏定义控制波形生成

#调试目标:启动 DVE 查看波形(需先运行 sim)
debug:
    dve \
        -vpd $(WAVE_DB) \              #加载波形数据库
        -session session.vpd &         #保存调试会话

#工程清理:删除所有生成文件
clean:
    rm -rf $(SIMV) csrc * .daidir * .vdb * .fsdb
    rm -rf $(LOG_DIR)/* .log ucli.key
    @echo "[INFO] 所有生成文件已清理"

#声明伪目标(防止与同名文件冲突)
.PHONY: help compile sim debug clean

#执行示例:
#make compile                          #仅编译设计
#make sim                              #编译并运行仿真(自动创建日志目录)
#make debug                            #必须先在另一个终端运行 sim 生成波形
#make clean                            #清理工程
```

### 5.2.3 VCS 调试

调试作为系统工程方法,其核心目标在于系统性地识别并消除计算机程序或电子硬件中的逻辑缺陷,以实现功能与设计规范的高度一致性。在复杂系统架构中,由于子系统间存在强耦合关系,因此调试难度呈指数级增长,局部模块的修正可能通过数据路径或控制流引发级联性异常。针对 VCS 仿真环境的调试实践,主要采用 3 类技术路径:基