查看所有版本

Vivado Design Suite 用户指南

逻辑仿真

UG900 (v2024.1) 2024 年 5 月 30 日

本文档为英语文档的翻译版本，若译文与英语原文存在歧义、差异、不一致或冲突，概以英语文档为准。译文可能并未反映最新英语版本的内容，故仅供参考，请参阅最新版本的英语文档获取最新信息。

AMD 自适应计算矢志不渝地为员工、客户与合作伙伴打造有归属感的包容性环境。为此，我们正从产品和相关宣传资料中删除非包容性语言。我们已发起内部倡议，以删除任何排斥性语言或者可能固化历史偏见的语言，包括我们的软件和 IP 中嵌入的术语。虽然在此期间，您仍可能在我们的旧产品中发现非包容性语言，但请确信，我们正致力于践行革新使命以期与不断演变的行业标准保持一致。如需了解更多信息，请参阅此链接。

目录

第 1 章

概述

按设计进程浏览内容

AMD 自适应计算文档按一组标准设计进程进行组织，以便帮助您查找当前开发任务相关的内容。您可以在设计中心页面上访问 AMD Versal™ 自适应 SoC 设计进程。您还可以使用设计流程助手来更深入了解设计流程，并找到特定于预期设计需求的内容。本文档涵盖了以下设计进程：

- 硬件、IP 和平台开发：为硬件平台创建 PL IP 块、创建 PL 内核、功能仿真以及评估 AMD Vivado™ 时序收敛、资源使用情况和功耗收敛。还涉及为系统集成开发硬件平台。本文档中适用于此设计进程的主题包括：
  - 第 3 章：使用第三方仿真器进行仿真
  - 第 4 章：使用 Vivado 仿真器进行仿真
  - 附录 F: Vivado IDE 中的 SystemC 支持

逻辑仿真概述

仿真是在软件环境内对真实的设计行为进行仿真的进程。仿真有助于通过注入激励并观察设计输出来验证设计的功能。

本章提供了仿真进程以及 AMD Vivado™ Design Suite 中的仿真选项的概述。

仿真进程包括：

- 为仿真创建测试激励文件、设置库并指定仿真设置
- 生成网表（如果执行综合后或实现后仿真）
- 使用 Vivado 仿真器或第三方仿真器运行仿真。如需了解有关受支持的仿真器的更多信息，请参阅 受支持的仿真器。

受支持的仿真器

Vivado Design Suite 中的支持的仿真器如下所示：

表 1：**受支持的仿真器**

仿真器	版本	是否与 Vivado 集成设计环境集成
AMD Vivado™ 仿真器	2024.1	与 Vivado 集成设计环境集成，其中每次仿真启动都会显示为 Vivado IDE 中的一个窗口框架。
Siemens EDA Questa Advanced Simulator	2023.3 (SE)	是
Siemens EDA ModelSim 仿真器	2023.3 (DE)	是
Synopsys Verilog Compiler Simulator (VCS)	U-2023.03-SP2	是
Aldec Rivera-PRO 仿真器	2023.04	是
Aldec Active-HDL	14.0	是
Cadence Xcelium Parallel Simulator	23.09.001	是

如需了解有关受支持的第三方仿真器版本的信息，请参阅《Vivado Design Suite 用户指南：版本说明、安装和许可》(UG973)。

如需了解有关 Vivado IDE 和 Vivado Design Suite 流程的更多信息，请参阅：

- 《Vivado Design Suite 用户指南：使用 Vivado IDE》(UG893)
- 《Vivado Design Suite 用户指南：设计流程概述》(UG892)

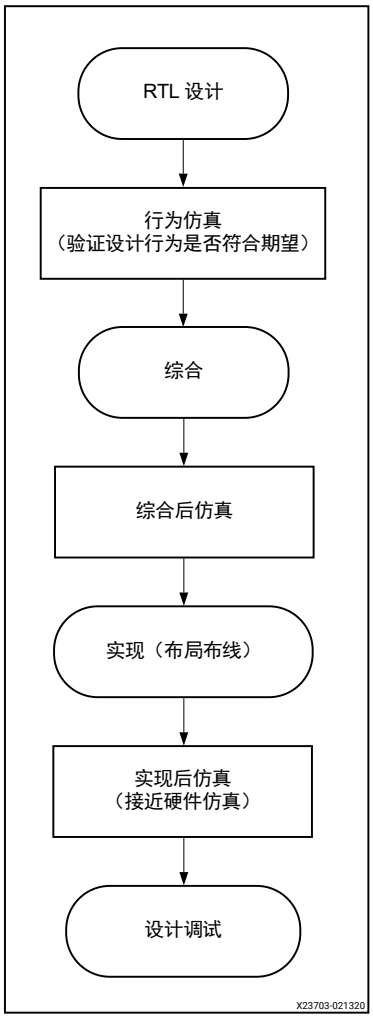
仿真流程

可在设计流程中的多个时间点应用仿真。它是设计输入后的首个步骤之一，也是实现后的最后步骤之一，在验证设计的最终功能和性能过程中执行。

仿真是迭代进程，通常会重复执行直至设计功能和时序要求都得到满足为止。

下图显示了典型设计的仿真流程：

图 1：**仿真流程**



寄存器传输级的行为仿真

寄存器传输级 (RTL) 行为仿真可包括：

- RTL 代码
- 例化的 UNISIM 库组件
- 例化的 UNIMACRO 组件
- UNISIM 门级模型（适用于 Vivado Logic Analyzer）
- SECUREIP 库

RTL 级仿真允许您先对自己的设计进行仿真和验证，而后再由综合或实现工具执行任何转换。您可将自己的设计作为模块或实体、块、器件或者系统来进行验证。

通常，执行 RTL 仿真是为了验证代码语法和确认代码功能符合期望。在此步骤中，设计主要以 RTL 来描述，故此无需时序信息。

除非设计包含已例化的器件库组件，否则 RTL 仿真并非局限于特定架构。为支持例化，AMD 提供了 UNISIM 库。

通过行为 RTL 来验证设计即可尽早修复设计问题并节省设计周期。

将初始设计创建局限于行为代码即可得到：

- 更多可读代码
- 更快且更简单的仿真
- 代码可移植性（移植到其他器件系列的能力）
- 代码复用（在未来设计中使用相同代码的能力）

综合后仿真

您可对已综合的网表进行仿真，验证已综合的设计是否满足功能要求并且行为与期望相符。在此仿真时间点，您可以采用估算的时序数值来执行时序仿真，但这并非典型操作。

功能仿真网表是分层折叠网表，它扩展至原语模块和实体级别，其层级的最低层次由原语和宏原语组成。

这些原语包含在 UNISIMS_VER 库（对应 Verilog）和 UNISIM 库（对应 VHDL）内。

相关信息

UNISIM 库

实现后仿真

您可在实现后执行功能仿真或时序仿真。时序仿真是最接近实际将设计下载到器件的仿真。它允许您确保已实现的设计满足功能要求和时序要求，并且在器件内的行为与期望的行为相符。

重要提示！执行完整的时序仿真可确保已完成的设计不含任何以其他方式可能无法发现的缺陷，例如：

- 由于下列原因导致的综合后和实现后功能更改：
  - 造成不匹配问题的综合属性或约束（例如，`full_case` 和 `parallel_case`）
  - 赛灵思设计约束 (XDC) 文件中应用的 UNISIM 属性
  - 仿真期间由不同仿真器对语言进行的解释
- 双端口 RAM 冲突
- 缺失时序约束或者错误应用时序约束
- 异步路径的操作
  - 由于最优化技巧而引发的功能问题

注释：对于 Versal 器件，仅限互连结构逻辑 (PL) 才支持综合后和实现后仿真，对于含硬核块 (NoC/AIE/PS) 的设计，则不予支持。仅限使用硬核块的设计才支持行为仿真。

语言和加密支持

Vivado 仿真器支持：

- VHDL，请参阅《IEEE 标准 VHDL 语言参考手册》(IEEE-STD-1076-1993) 和 VHDL-2008 的部分内容。
- Verilog，请参阅《IEEE 标准 Verilog 硬件描述语言》(IEEE-STD-1364-2001)。
- SystemVerilog，请参阅《适用于 SystemVerilog 的 IEEE 标准 - 统一硬件设计、规范和验证语言》(IEEE-STD-1800-2009)。
- IEEE P1735 加密，请参阅《推荐的电子产品设计 IP 加密与管理实践》(IEEE-STD-P1735)。

第 2 章

准备仿真

本章描述了您在 AMD Vivado™ 集成设计环境 (IDE) 内仿真 AMD 器件时需要的组件。

执行仿真前完成如下设置：

- 创建测试激励文件以反映您要运行的仿真操作。
- 在 Vivado IDE 中设置安装位置（如不使用 Vivado 仿真器）。
- 编译您的库（如不使用 Vivado 仿真器）。
- 选择并声明您需要使用的库。
- 指定仿真设置，例如，目标仿真器、仿真顶层模块名称、顶层模块（受测设计）、显示仿真集，并定义编译、细化、仿真、网表和高级选项。
- 生成网表（如果执行综合后或实现后仿真）。

使用测试激励文件和激励

测试激励文件是专为仿真器编写的硬件描述语言 (HDL) 代码，用于：

- 对设计进行例化和初始化。
- 生成激励并应用于设计。
- 监控设计输出结果并检查功能正确性（可选）。

您还可设置测试激励文件以显示仿真并输出到文件、波形或显示屏。测试激励文件可采用简单结构，并且可按顺序将激励应用于特定输入。

测试激励文件也可以采用复杂结构并包含：

- 子例程调用
- 从外部文件读入的激励
- 条件激励
- 其他更复杂的结构

测试激励文件相较于交互仿真的优势在于它能：

- 在整个设计进程中允许执行可重复的仿真
- 提供测试条件的文档记录

以下项目符号项是创建有效的测试激励文件的建议。

- 始终在 Verilog 测试激励文件内指定 `timescale。例如：

```
`timescale 1ns/1ps
```

- 在仿真时间零时，将所有输入都初始化到测试激励文件内的设计，这样即可按已知的值正确开始仿真。
- 在 100 ns 后应用激励数据以便考量在功能仿真和时序仿真内所使用的默认全局置位/复位 (GSR) 脉冲。
- 在释放全局置位/复位 (GSR) 前，开始时钟源。

如需了解有关测试激励文件的更多信息，请参阅《编写高效的测试激励文件》(XAPP199)。

提示：创建测试激励文件时，请谨记，在综合后和实现后时序仿真中会自动发生 GSR 脉冲。这样即可在仿真的前 100 ns 内使所有寄存器保持处于复位状态。

相关信息

使用全局复位和三态

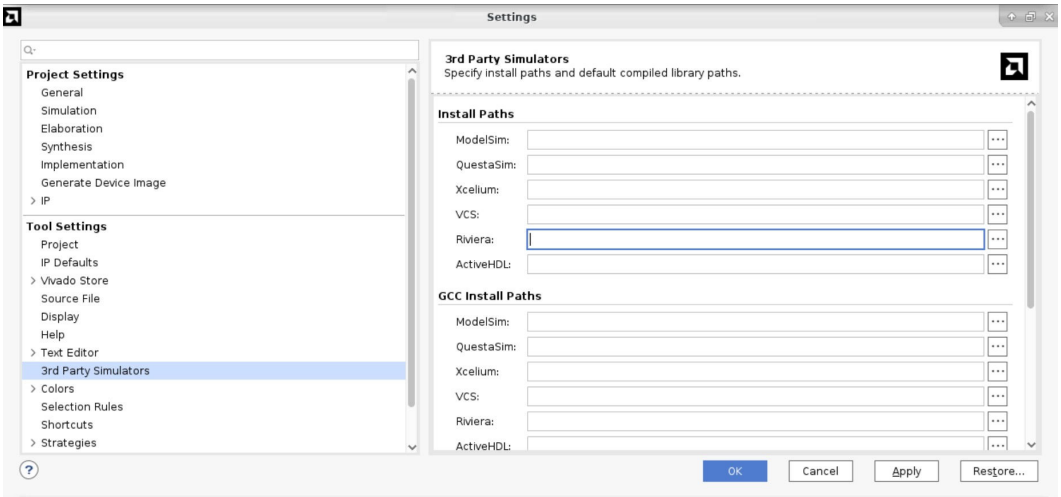
指向仿真器安装位置

要定义安装路径，请执行以下操作：

1. 选择“Tools”→“Settings”→“Tool Settings”→“Third-party Simulators”（工具 > 设置 > 工具设置 > 第三方仿真器）。
2. 在“Settings”（设置）对话框的“third-party simulators”（第三方仿真器）选项卡中，选中“Install Paths”（安装路径）下的仿真器（如下图所示），并浏览至该安装路径。
3. 选中“Default Compiled Library Paths”（默认已编译的库路径）并浏览至相关已编译的库路径。您可稍后再设置库路径。如需了解有关如何为仿真器编译库的更多信息，请参阅 编译仿真库。

   注释：在执行 Vivado IDE 安装的过程中，会安装 Vivado 仿真器。因此，您无需为 Vivado 仿真器设置安装位置。

图 2：**Vivado 仿真器安装**



编译仿真库

重要提示！使用 Vivado 仿真器时，无需编译仿真库。但使用第三方仿真器时，必须编译这些库。

Vivado Design Suite 以一组文件和库的形式提供了仿真模型。您的仿真工具必须先编译这些文件，然后再进行设计仿真。仿真库包含器件以及 IP 行为和时序模型。编译后的库可供多个设计工程使用。

编译进程期间，Vivado 会创建默认初始化文件，供仿真器用于引用已编译的库。`compile_simlib` 命令会在库编译期间指定的库输出目录中创建此文件。默认初始化文件包含控制变量用于指定引用库路径、优化、编译器和仿真器设置。如果在此路径中未找到正确的初始化文件，则无法在包含 AMD 原语的设计上运行仿真。

初始化文件名称因所用仿真器而异，如下所示：

- Questa Advanced Simulator/ModelSim：`modelsim.ini`
- Xcelium：`cds.lib`
- VCS：`synopsys_sim.setup`
- Riviera/Active-HDL：`library.cfg`

如需了解有关已编译的仿真器专用库文件的更多信息，请参阅第三方仿真工具文档。

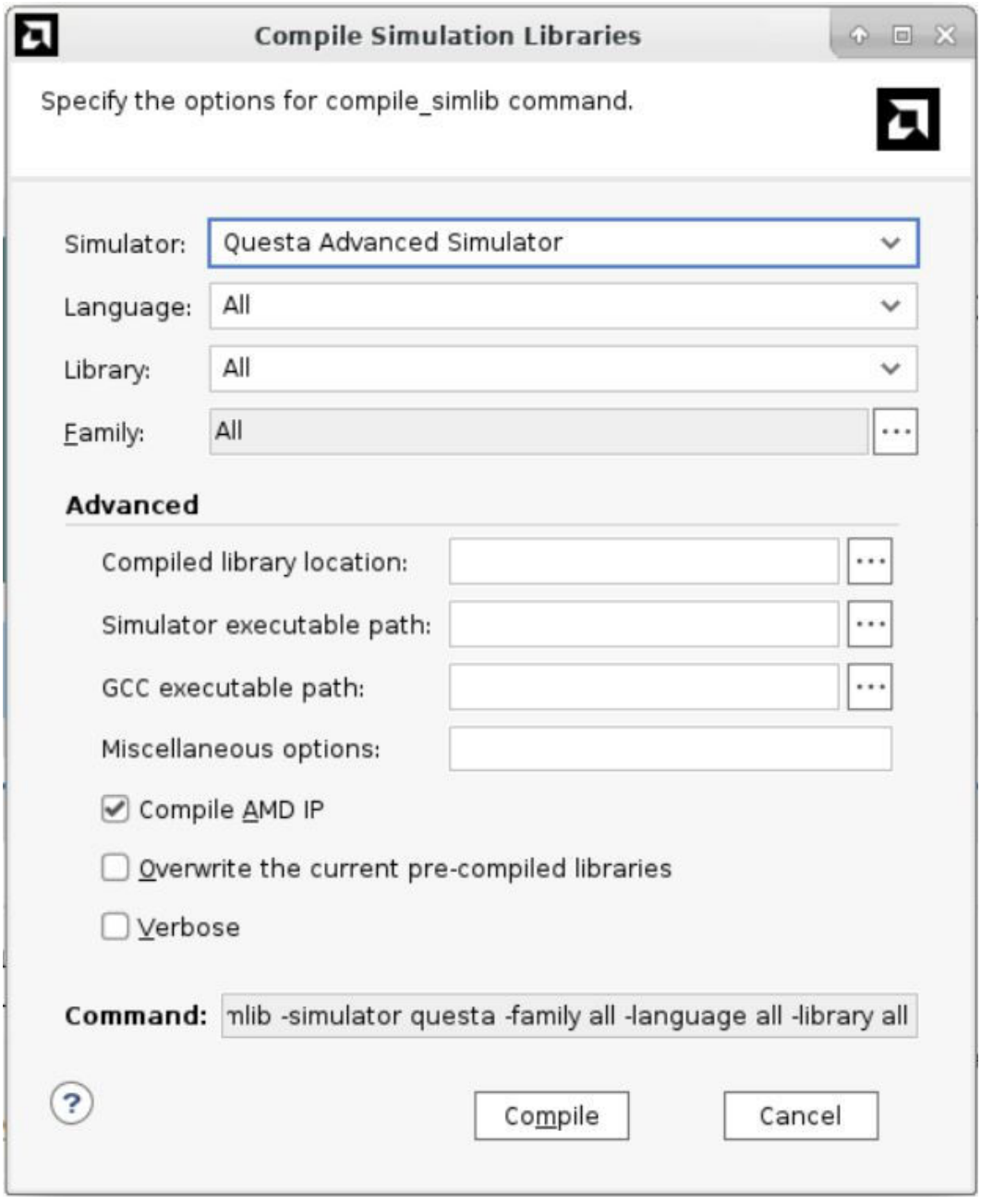
重要提示！库的编译通常是一次性操作，前提是使用相同版本的工具。但如果 Vivado 工具或仿真器版本有变，就需要重新编译这些库。

您可使用 Vivado IDE 或者使用 Tcl 命令来编译库，如以下各节所述。

使用 Vivado IDE 编译仿真库

选择“Tools”→“Compile Simulation Libraries”（工具 > 编译仿真库）打开如下图所示对话框。

图 3： **“Compile Simulation Libraries”对话框**



设置以下选项：

- “Simulator”（仿真器）：从仿真器下拉菜单中选择仿真器。
- “Language”（语言）：为指定语言编译库。如果不指定该选项，则语言设置为对应以上所选仿真器的语言。对于多语言仿真器，将编译 Verilog 和 VHDL 库。
- “Library”（库）：指定要编译的仿真库。默认情况下，`compile_simlib` 命令会编译所有仿真库。
- “Family”（系列）：将所选库编译到指定器件系列。默认生成所有器件系列。

- “Compiled library location”（已编译的库位置）：指定用于保存已编译的库结果的目录路径。默认情况下，在非工程模式下，库保存在当前工作目录内，在工程模式下，库保存在 `<project>/<project>.cache/compile_simlib` 目录内。如需了解有关工程模式和非工程模式的更多信息，请参阅《Vivado Design Suite 用户指南：设计流程概述》(UG892)。

  提示：由于 Vivado 仿真器具有预编译的库，因此无需识别库位置。

- “Simulator executable path”（仿真器可执行文件路径）：指定用于从中查找仿真器可执行文件的目录。如果在 `$PATH` 或 `%PATH%` 环境变量内未指定目标仿真器，或者要覆盖来自 `$PATH` 或 `%PATH%` 环境变量的路径，那么该选项是必需的。
- “GCC executable path”（GCC 可执行文件路径）：指定用于从中查找 GCC 安装文件的目录。如果未按 GCC 路径设置 所述完成 GCC 路径设置，那么该选项是必需的。如果不使用 SystemC IP，则忽略该选项。
- “Miscellaneous Options”（其他选项）：为 `compile_simlib` Tcl 命令指定其他选项。
- “Compile AMD IP”（编译 AMD IP）：为 AMD IP 启用或禁用编译仿真库。
- “Overwrite current pre-compiled libraries”（覆盖当前预编译的库）：覆盖当前预编译的库。
- “Verbose”（详细）：暂时覆盖所有消息限制，并返回来自该命令的所有消息。
- “Command”（命令）：显示您在对话框中输入的选项的等效 Tcl 命令。

  提示：您可使用“Command”命令的值在 Tcl/非工程模式下生成仿真库。

使用 Tcl 命令编译仿真库

或者，您也可以使用 `compile_simlib` Tcl 命令来编译仿真库。欲知详情，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835) 中的 `compile_simlib`，或者输入 `compile_simlib -help`。

以下提供了每个第三方仿真器的命令示例：

- Questa Advanced Simulator：为当前目录中的所有库和所有产品系列生成适用于 Questa 的仿真库。

```
compile_simlib -language all -simulator questa -library all -family all
```

- ModelSim：在 `/a/b/c` 中为 ModelSim 生成仿真库，其中 ModelSim 可执行路径为 `<simulator_installation_path>`。

```
compile_simlib -language all -dir {/a/b/c} -simulator modelsim -
simulator_exec_path
{<simulator_installation_path>} -library all -family all
```

- VCS：在 `/a/b/c` 中为 UNISIM 库生成适用于 VCS 的仿真库，以供 Verilog 语言使用。

```
compile_simlib -language verilog -dir {/a/b/c} -simulator vcs -library
unisim
-family all
```

- Xcelium：在 /a/b/c 中为 UNISIM 库生成适用于 Xcelium 的仿真库，以供 Verilog 语言使用。

```
compile_simlib -language verilog -dir {/a/b/c} -simulator xcelium -
library unisim
-family all
```

更改 compile_simlib 默认设置

config_compile_simlib Tcl 命令支持您配置第三方仿真器选项以供 compile_simlib 命令使用。

Tcl 命令

```
config_compile_simlib [-cfgopt <arg>] [-simulator <arg>] [-reset] [-quiet]
[-verbose]
```

其中：

- -cfgopt <arg>：表示配置选项，格式为 <simulator>:<language>:<library>:<options>。
- -simulator：表示仿真器名称，您所用配置即来自该仿真器。
- -reset：允许您为指定仿真器复位所有先前配置。
- -quiet：执行命令，但不在 Tcl 控制台中显示任何信息。
- -verbose：执行命令，并向 Tcl 控制台输出所有命令。

例如，要更改用于编译 UNISIM VHDL 库的选项，请输入：

```
config_compile_simlib -cfgopt {modelsim.vhdl.unisim:-source -93}
```

重要提示！ compile_simlib 命令用于编译 AMD 原语和 AMD Vivado IP 的仿真模型。生成 AMD Vivado IP 时，其核作为输出文件来交付；因此这些 IP 核均包含在使用 compile_simlib 创建的预编译库中。

在新输出目录中使用 XILINX_PATH 来编译已安装补丁的 IP 存储库

假定已安装补丁的 IP 存储库位于以下位置：

```
'/test/patched_ip_repo/data/ip/xilinx'
```

假定已安装补丁的 IP 存储库位于以下位置：

要对默认已安装的 IP 存储库和新输出目录中 XILINX_PATH 所指向的存储库进行编译，请将 XILINX_PATH 环境 (env) 变量指向已安装补丁的 IP 存储库，并运行 compile_simlib。compile_simlib 会处理来自默认已安装的存储库和来自 XILINX_PATH 所设置的存储库的 IP 库源文件。

```
% setenv XILINX_PATH /test/patched_ip_repo % compile_simlib -simulator
<simulator> -directory <new_clibs_dir>
```

在现有输出目录中使用 XILINX_VIVADO 来编译已安装补丁的 IP 存储库

假定已安装补丁的 IP 存储库位于以下位置：

```
'/test/patched_ip_repo/data/ip/xilinx'
```

如果要对现有输出目录中 XILINX_PATH 所指向的存储库进行编译，但该输出目录中的库先前已针对安装的默认 IP 存储库进行了编译，那么请将 XILINX_PATH 的 env 变量设置为指向该已安装补丁的 IP 存储库，然后运行 compile_simlib。compile_simlib 会对来自现有输出目录中 XILINX_PATH 所设置的存储库的 IP 库源文件进行处理。

```
% setenv XILINX_PATH /test/patched_ip_repo % compile_simlib -simulator
<simulator> -directory <existing_clibs_dir>
```

使用 AMD 仿真库

您可将 AMD 仿真库搭配支持 VHDL-93 和 Verilog-2001 语言标准的任意仿真器一起使用。某些延迟和建模信息已构建到库中；这是正确执行 AMD 硬件器件仿真所必需的前提条件。

对设计中的组件进行例化时，仿真器必须引用描述组件功能的库才能确保正确完成仿真。AMD 库根据模型功能分为多个类别。

下表列出了 AMD 提供的仿真库：

表 2：**仿真库**

库名称	描述	VHDL 库名称	Verilog 库名称
UNISIM	AMD 原语的功能仿真。	UNISIM	UNISIMS_VER
UNIMACRO	AMD 宏的功能仿真。	UNIMACRO	UNIMACRO_VER
UNIFAST	快速仿真库。	UNIFAST	UNIFAST_VER
SIMPRIM	AMD 原语的时序仿真。	不适用	SIMPRIMS_VER [1]
SECUREIP	用于 AMD 器件功能特性（例如，PCIe IP、千兆位收发器等）的功能仿真和时序仿真的仿真库。 您可在以下位置的 SECUREIP 下找到 IP 列表： <Vivado_Install_Dir>/data/secureip	SECUREIP	SECUREIP
XPM	AMD 原语的功能仿真	XPM	XPM [2]

注释：

1. SIMPRIMS_VER 是逻辑库名称，Verilog SIMPRIM 物理库映射到此逻辑库名称。
2. 支持将 XPM 作为预编译 IP 来使用。因此，您无需向工程添加源文件。对于第三方仿真器，Vivado 工具会映射到以 compile_simlib 生成的预编译 IP。

重要提示！您必须根据仿真点指定不同仿真库。在实现前网表和实现后网表中有不同的门级单元。

下表列出了每个仿真点所需的仿真库。

表 3：**仿真点和相关库**

仿真点	UNISIM	UNIFAST	UNIMACRO	SECUREIP	SIMPRIM（仅限 Verilog）	SDF
1.寄存器传输级 (RTL)（行为）	支持	支持	支持	支持	不适用	不支持
2.综合后仿真（功能）	支持	支持	不适用	支持	不适用	不适用
3.综合后仿真（时序）	不适用	不适用	不适用	支持	支持	支持
4.实现后仿真（功能）	支持	支持	不适用	支持	不适用	不适用
5.实现后仿真（时序）	不适用	不适用	不适用	支持	支持	支持

重要提示！Vivado 仿真器使用预编译的仿真器件库。安装完库的更新后，预编译的库会自动更新。

注释：Verilog SIMPRIMS_VER 使用的源代码与 UNISIM 相同，但添加了适用于时序注解的指定块。SIMPRIMS_VER 是逻辑库名称，Verilog SIMPRIM 物理库映射到此逻辑库名称。

下表中列出了库的位置。

表 4：**仿真库位置**

库	HDL 类型	位置
UNISIM	Verilog	<Vivado_Install_Dir>/data/verilog/src/unisims
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unisims
UNIFAST	Verilog	<Vivado_Install_Dir>/data/verilog/src/unifast
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unifast
UNIMACRO	Verilog	<Vivado_Install_Dir>/data/verilog/src/unimacro
	VHDL	<Vivado_Install_Dir>/data/vhdl/src/unimacro
SECUREIP	Verilog	<Vivado_Install_Dir>/data/secureip/

如需了解有关这些库的更多详细信息，请参阅后文各小节。

UNISIM 库

功能仿真使用 UNISIM 库，并且包含器件原语或低级别构建块的描述。

重要提示！默认情况下，compile_simlib 命令会为 IP 目录中的所有 IP 编译静态仿真文件。

加密的组件文件

下表列出了 UNISIM 库组件文件，这些文件支持您在设计中包含 IP 时调用预编译的加密库文件。在库搜索路径中包含所需路径。

表 5：**组件文件**

组件文件	描述
<Vivado_Install_Dir>/data/verilog/src/unisim_retarget_comp.vp	加密的 Verilog 文件
<Vivado_Install_Dir>/data/vhdl/src/unisims/unisim_retarget_VCOMP.vhdp	加密的 VHDL 文件

重要提示！Verilog 模块名称和文件名均为大写字母。例如，模块 BUFG 名为 BUFG.v，模块 IBUF 名为 IBUF.v。请确保 UNISIM 原语例化遵循大写命名约定。

VHDL UNISIM 库

VHDL UNISIM 库拆分为以下几个文件，用于为 AMD 器件系列指定原语：

- 组件声明 (unisim_VCOMP.vhd)
- 封装文件 (unisim_VPKG.vhd)

要使用这些原语，请将以下两行内容置于每个文件开头：

```
library UNISIM;
use UNISIM.VCOMPONENTS.all;
```

重要提示！您还必须编译该库，并将其映射到仿真器。方法取决于仿真器。

注释：对于 Vivado 仿真器，库编译和映射是集成功能，无需额外的用户编译或映射。

注释：从 AMD Versal™ 自适应 SoC 开始，AMD 仅为新原语交付 Verilog/SystemVerilog 模型。这意味着对于仅限 VHDL 的设计，需要采用混合语言环境，正如先前 IP 和 XPM 所需的环境一样。如需了解更多信息，请参阅 AR76496。

Verilog UNISIM 库

在 Verilog 中，在独立 HDL 文件内指定各库模块。这样即可允许 `-y` 库规范开关为所有组件搜索指定的目录，并自动扩展该库。

使用模块前，无法在 HDL 文件中指定 Verilog UNISIM 库。要使用库模块，请使用全大写字母来指定模块名称。

以下示例显示了例化的模块名称以及与该模块关联的文件名：

- 模块 BUFG 名为 `BUFG.v`
- 模块 IBUF 名为 `IBUF.v`

Verilog 区分大小写。请确保 UNISIM 原语例化遵循大写命名约定。

如果使用预编译库，请使用正确的仿真器命令行开关来指向预编译的库。Vivado 仿真器示例如下所示：

```
-L unisims_ver
```

其中：

`-L` 是库规范选项。

UNIMACRO 库

UNIMACRO 库在功能仿真期间使用，包含选定器件原语的宏描述。

重要提示！只要包含《Vivado Design Suite 7 系列 FPGA 和 Zynq 7000 SoC 库指南》(UG953) 中列出的器件宏，就必须指定 UNIMACRO 库。

VHDL UNIMACRO 库

要使用这些原语，请将以下两行内容置于每个文件开头：

```
library UNIMACRO;
use UNIMACRO.Vcomponents.all;
```

Verilog UNIMACRO 库

在 Verilog 中，在独立 HDL 文件内指定各库模块。这样即可允许 `-y` 库规范开关为所有组件搜索指定的目录，并自动扩展该库。

不同于 VHDL 中的要求，使用模块之前，无需在 HDL 文件中指定 Verilog UNIMACRO 库。要使用库模块，请使用全大写字母来指定模块名称。您还必须编译和映射该库；所使用的方法取决于您选择的仿真器。

重要提示！Verilog 模块名称和文件名均为大写字母。例如，模块 BUFG 名为 `BUFG.v`。请确保 UNIMACRO 原语例化遵循大写命名约定。

SIMPRIM 库

SIMPRIM 库可用于对综合或实现后生成的时序仿真网表进行仿真。

重要提示！时序仿真仅在 Verilog 中受支持；不存在 VHDL 版本的 SIMPRIM 库。

提示：如果您是 VHDL 用户，您可运行综合后和实现后功能仿真（在此情况下无需标准延迟格式 (SDF) 注解，仿真网表使用 UNISIM 库）。您可使用 write_vhdl Tcl 命令来创建网表。如需了解使用信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)。

以下提供了为 Vivado 仿真器指定库的示例：

```
-L SIMPRIMS_VER
```

其中：

- `-L` 是库规范选项。
- `SIMPRIMS_VER` 是逻辑库名称，Verilog SIMPRIM 已映射到此逻辑库名称。

SECUREIP 仿真库

SECUREIP 库可用于复杂的器件组件（例如，GT）的功能仿真和时序仿真。

注释：在 Vivado 仿真器中对 Secure IP 块给予完整支持，无需额外设置。

AMD 利用 IEEE 标准《推荐的电子产品设计 IP 加密与管理实践》(IEEE-STD-P1735) 中指定的加密方法。库编译进程会自动处理加密。

注释：请参阅仿真器文档，了解可搭配您的仿真器一起用于指定库的命令行开关的相关信息。

下表列出了各项特殊注意事项，您必须与仿真器供应商根据这些注意事项共同协商安排，才能使用这些库。

表 6：**使用 SECUREIP 库的特殊注意事项**

仿真器名称	供应商	要求
Siemens EDA ModelSim SE	Siemens	如果设计输入采用的是 VHDL，则需要混合语言许可证。请联系供应商获取更多信息。
Siemens EDA Questa Advanced Simulator		
VCS	Synopsys	
Xcelium	Cadence	
Active-HDL	Aldec	如果设计输入采用的是 VHDL，则需要混合语言许可证。
Riviera-PRO		

重要提示！如需了解有关受支持的第三方仿真器版本的信息，请参阅《Vivado Design Suite 用户指南：版本说明、安装和许可》(UG973)。

VHDL SECUREIP 库

UNISIM 库包含 VHDL SECUREIP 的封装文件。请将以下两行内容置于每个文件开头，以便仿真器绑定到实体：

```
Library UNISIM;
UNISIM.VCOMPONENTS.all;
```

Verilog SECUREIP 库

使用 Verilog 代码运行仿真时，必须为大部分仿真器引用 SECUREIP 库。

如果使用预编译库，请使用正确的指令来指向预编译的库。Vivado 仿真器示例如下所示：

```
-L SECUREIP
```

重要提示！您可在编译时使用 `-f` 开关来使用 Verilog SECUREIP 库。通过以下路径即可获取文件列表：`<Vivado_Install_Dir>/data/secureip/secureip_cell.list.f`。

UNIFAST 库

UNIFAST 库是可选库，可供您在 RTL 行为仿真期间用于缩短仿真运行时间。

重要提示！UNIFAST 库是可选库，可供您在功能仿真期间用于缩短仿真运行时间。仅限 7 系列器件才支持 UNIFAST 库。AMD UltraScale™ 以及后续器件架构不支持 UNIFAST 库，因为在 UNISIM 库中默认集成所有优化。UNIFAST 库无法用于仿真验收，因为库组件并不包含完整模型中可用的所有检查/功能特性。

建议：请将 UNIFAST 库用于设计的初始验证，然后使用 UNISIM 库运行完整验证。

通过支持仿真模式下的一小部分原语功能特性，即可改善仿真运行时。

注释：仿真模型仅检查不受支持的属性值。

MMCME2

为了缩短仿真运行时间，快速 MMCME2 仿真模型与完整模型相比存在如下变化：

1. 快速仿真模型仅提供基本时钟生成功能。不支持其他功能，如 DRP、精细相移、时钟停止和时钟级联。
2. 它假定输入时钟保持稳定，不存在频率和相位更改。当 LOCKED 信号断言 HIGH 有效后，输入时钟频率采样就会停止。
3. 输出时钟频率、相位、占空比和其他功能特性是根据输入时钟频率和参数设置直接计算所得的。

   注释：输出时钟频率并不是从输入到 VCO 时钟生成的。
4. 标准和快速 MMCME2 仿真模型 LOCKED 信号断言有效时间各不相同。
   - 标准模型 LOCKED 断言有效时间取决于 M 和 D 设置。对于较大的 M 值和 D 值，标准 MMCME2 仿真模型的锁定时间相对较长。
   - 在快速仿真模型中，LOCKED 断言有效时间会被缩短。

DSP48E1

为了缩短仿真运行时间，快速 DSP48E1 仿真模型与完整模型相比移除了如下功能特性。

- 模式检测
- 上溢/下溢
- DRP 接口支持

GTHE2_CHANNEL/GTHE2_COMMON

为了缩短仿真运行时间，快速 GTHE2 仿真模型与完整模型相比存在如下功能特性差异：

- GTH 链路必须保持同步，近端和远端链路伙伴之间不存在百万分率（Parts Per Million，PPM）差异。
- 对于硬件操作，通过 GTH 的时延并非周期精确。
- 您无法对 DRP 量产复位序列进行仿真。使用 UNIFAST 模型时请绕过该序列。

使用 Verilog UNIFAST 库

为了缩短仿真运行时间，快速 GTXE2 仿真模型与完整模型相比存在如下功能特性差异：

- GTX 链路必须保持同步，近端和远端链路伙伴之间不存在百万分率（Parts Per Million，PPM）差异。
- 对于硬件操作，通过 GTX 的时延并非周期精确。

方法 1：使用完整的 Verilog UNIFAST 库（建议）

方法 1 是建议方法，可供您用于对所有 UNIFAST 模型进行仿真。

在 Tcl 控制台中，以下 Tcl 命令可用于在 Vivado 工程环境内为 Vivado 仿真器、ModelSim 或 VCS 启用 UNIFAST 支持（快速仿真模型）：

```
set_property unifast true [current_fileset -simset]
```

如需了解有关组件文件的更多信息，请参阅 UNISIM 库。

如需了解更多信息，请参阅相应的第三方仿真用户指南。

方法 2：使用专用 UNIFAST 模块

建议高级用户使用，这类用户可使用此方法指定搭配 UNIFAST 模型执行仿真的模块。

要指定个别库组件，请改用 Verilog 配置语句。在 `config.v` 文件中指定：

- 顶层模块或配置的名称（例如：`config cfg_xilinx;`）
- 设计配置适用于的名称（例如：`design test bench;`）
- 未显式调出的单元或实例的库搜索顺序（例如：`default liblist unisims_ver unifast_ver;`）
- 将特定 CELL（单元）或 INSTANCE（实例）映射到特定库（例如，`instance testbench.inst.O1 use unifast_ver.MMCME2;`）

  注释：对于 ModelSim (vsim)，仅将 `-genblk` 添加到层级名称（例如：`instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.MMCME2; - VSIM`）。

`config.v` 示例

```
config cfg_xilinx;
design testbench;
default liblist unisims_ver unifast_ver;
//Use fast MMCM for all MMCM blocks in design
cell MMCME2 use unifast_ver.MMCME2;
//use fast dSO48E1for only this specific instance in the design
instance testbench.inst.O1 use unifast_ver.DSP48E1;
//If using ModelSim or Questa, add in the genblk to the name
(instance testbench.genblk1.inst.genblk1.O1 use unifast_ver.DSP48E1)
endconfig
```

使用 VHDL UNIFAST 库

VHDL UNIFAST 库的基本结构与 Verilog 相同，可搭配多种体系结构或库一起使用。您可在测试激励文件中包含该库。

以下示例使用向下钻取层级搭配 `for` 调用：

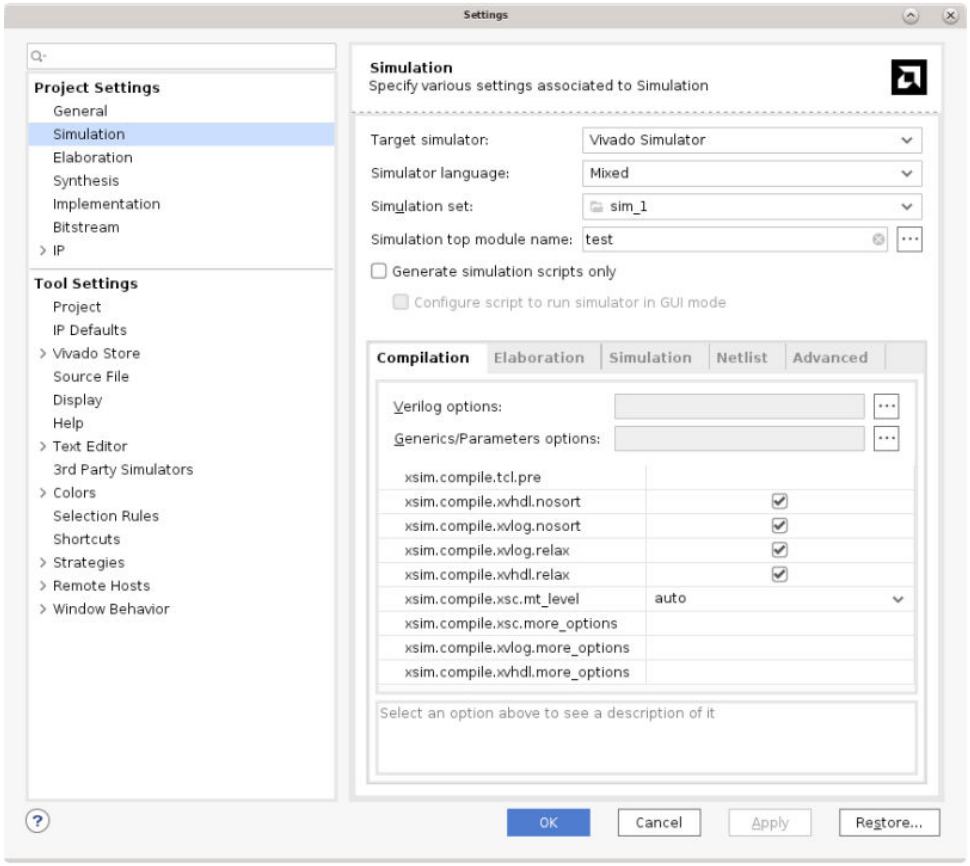
```
library unisim;
library unifast;
configuration cfg_xilinx of testbench
is for xilinx
.. for inst:netlist
. . . use entity work.netlist(inst);
.......for inst
.........for all:MMCME2
..........use entity unifast.MMCME2;
.........end for;
.......for O1 inst:DSP48E1;
.........use entity unifast.DSP48E1;
.......end for;
...end for;
..end for;
end for;
end cfg_xilinx;
```

注释：如果您要使用 VHDL UNIFAST 模型，请在细化期间使用配置来绑定 UNIFAST 库。

使用仿真设置

您可以使用仿真设置来指定目标仿真器、显示仿真集、仿真顶层模块名称、顶层模块（受测设计）、编译、细化、仿真和网表的选项卡式列表以及各项高级选项。在 Vivado IDE Flow Navigator 中，右键单击“Simulation”（仿真），然后选择“Simulation Settings”（仿真测试）以打开“Settings”（设置）对话框中的“Simulation Settings”，如下图所示。

图 4：**“Settings”对话框**



“Settings”对话框包含以下仿真设置：

- “Target simulator”（目标仿真器）：从仿真器下拉菜单中选择仿真器。默认仿真器是 AMD Vivado™ 仿真器。但也支持许多第三方仿真器。
- “Simulator language”（仿真器语言）：选择仿真器语言模式。设计中各种 IP 使用的仿真模型因 IP 支持的语言而异。
- “Simulation set”（仿真集）：选择仿真命令默认使用的仿真集。

  重要提示！先前定义的仿真集的编译和仿真设置并不会应用于新定义的仿真集。

- “Simulation top module name”（仿真顶层模块名称）：输入仿真期间要使用的备用顶层模块。
- “Generate simulation scripts only”（仅生成仿真脚本）：如果选中此项，则生成脚本。不调用仿真。
- “Configure script to run simulator in GUI mode”（将脚本配置为以 GUI 模式运行仿真器）：如果选中此项，则以仅限脚本模式启动仿真器 GUI。
- “Compiled library location”（已编译的库位置）：选择第三方仿真器时会显示该选项。仿真器会从该目录位置访问已编译的库。为便于执行此操作，您必须使用 `comb_simlib` 命令来显式定义库的编译路径。
- “Compilation”（编译）选项卡：该选项卡用于定义和管理编译器指令，这些指令作为属性存储在仿真文件集上，供 xvlog 和 xvhdl 实用工具用于编译 Verilog 和 VHDL 源文件以便进行仿真。

注释：xvlog 和 xvhdl 均为 Vivado 仿真器专用命令。适用的实用工具会根据目标仿真器而变。

- “Elaboration”（细化）选项卡：该选项卡用于定义和管理细化指令，这些指令作为属性存储在仿真文件集上，并供 xelab 实用工具用于细化和生成仿真快照。选择该表中的属性即可显示其描述并编辑其值。

  注释：xelab 是 Vivado 仿真器专用命令。适用的实用工具会根据目标仿真器而变。

- “Simulation”（仿真）选项卡：该选项卡用于定义和管理仿真指令，这些指令作为属性存储在仿真文件集上，供 xsim 应用用于对当前工程进行仿真。选择该表中的属性即可显示其描述并编辑其值。

- “Netlist”（网表）选项卡：该选项卡允许访问与 Verilog 网表的 SDF 注解以及 SDF 延迟所捕获的工艺角相关的网表配置选项。这些选项作为属性存储在仿真文件集上，在为仿真编写网表时使用。

- “Advanced”（高级）选项卡：该选项卡包含两个选项：

  - “Enable incremental compilation”（启用增量编译）：该选项用于在连续运行仿真期间启用增量编译并保留仿真文件。默认启用该选项。

  - “Include all design sources for simulation”（包含所有设计源文件用于仿真）：默认启用该选项。选中该选项可确保来自设计源文件的所有文件以及来自当前仿真集的文件都用于仿真。即使您更改设计源文件，启动行为仿真时所更改的内容也会一并更新。

  注意！仅限必要时才应更改“Advanced”选项卡中的设置。默认情况下选中“Include all design sources for simulation”复选框。取消选中此框可能产生意外结果。只要选中此复选框，仿真集就会包含非关联 (OOC) IP、IP integrator 文件和 DCP。

注释：如需了解有关“Compilation”、“Elaboration”、“Simulation”、“Netlist”和“Advanced”选项卡中的属性的详细信息，请参阅 附录 A: 编译、细化、仿真、网表和高级选项。

认识仿真器语言选项

大部分 AMD IP 只能为单一语言提供行为仿真模型，这样即可在您未获得相应语言的许可时，为锁定语言的仿真器有效禁用仿真。`simulator_language` 属性可确保 IP 为任意给定语言提供仿真模型。例如，如果您当前使用单一语言仿真器，可设置 `simulator_language` 属性来匹配仿真器语言。

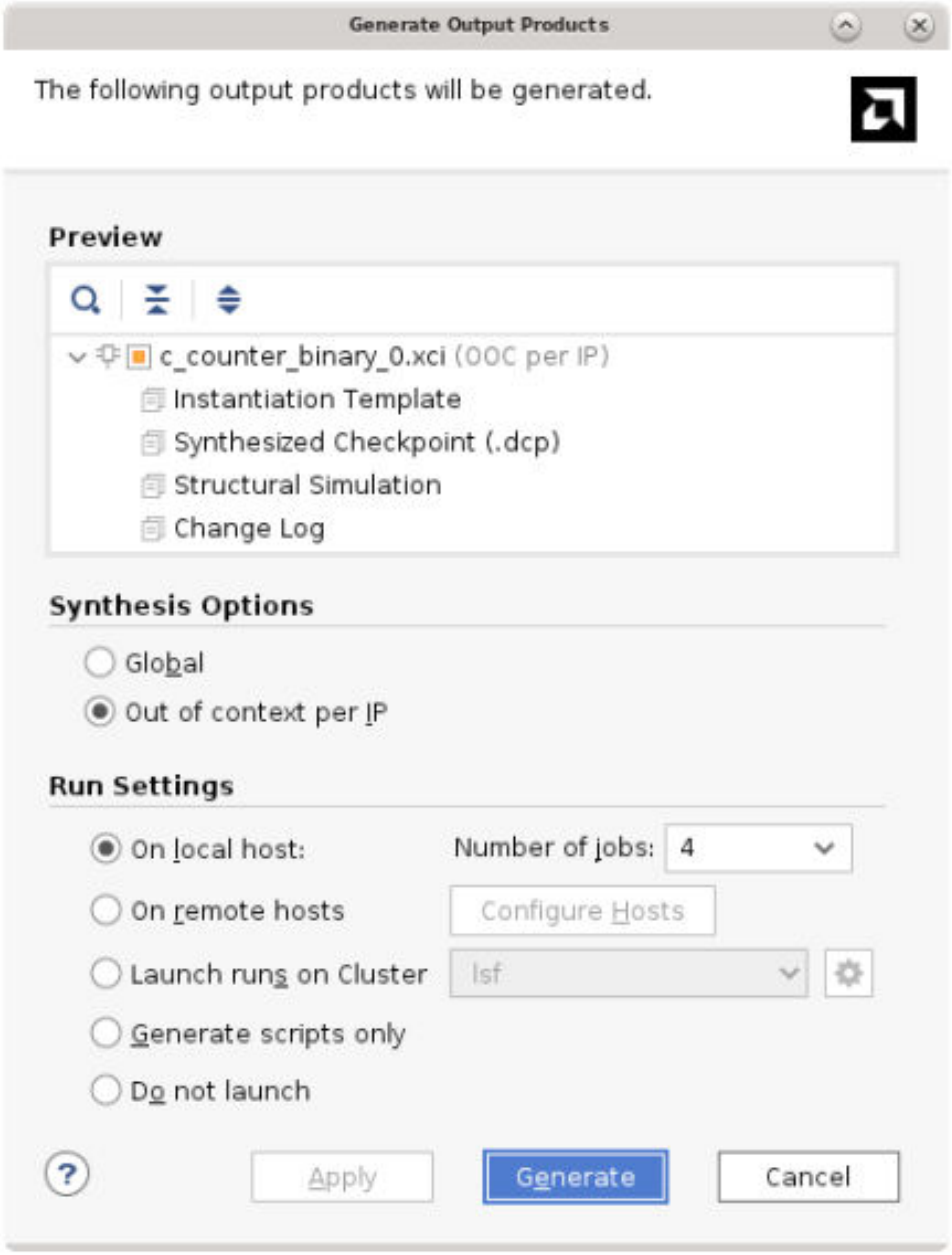
Vivado Design Suite 可通过使用 IP 的可用仿真文件按需生成特定语言的结构化仿真模型，从而确保仿真模型可用性。如果出现缺少行为模型或者不匹配许可仿真语言等情况，Vivado 工具会自动生成结构化仿真模型来启用仿真。否则，则使用 IP 的现有行为仿真模型。如果不存在任何综合或仿真文件，则不支持仿真。

注释：如果禁用生成的综合检查点 (.dcp)，`simulator_language` 属性就无法交付特定语言的仿真网表文件。

1. 在 Flow Navigator 中，单击“IP Catalog”（IP 目录）打开 IP 目录。
2. 右键单击相应的 IP 并从弹出菜单中选中“Customize IP”（自定义 IP）。
3. 在“Customize IP”对话框中，单击“OK”。

这样会打开“Generate Output Products”（生成输出文件）对话框，如下图所示。

图 5：“Generate Output Products”对话框



下表演示了 `simulator_language` 属性的功能。

表 7：**simulator_language 属性的功能**

IP 提供的仿真模型	simulator_language 值	使用的仿真模型
IP 可提供 VHDL 和 Verilog 行为模型	混用	行为模型 (`target_language`)
	Verilog	Verilog 行为模型
	VHDL	VHDL 行为模型

表 7：**simulator_language 属性的功能** (续)

IP 提供的仿真模型	simulator_language 值	使用的仿真模型
IP 仅提供 Verilog 行为模型	混用	Verilog 行为模型
	Verilog	Verilog 行为模型
	VHDL	从 DCP 生成的 VHDL 仿真网表
IP 仅提供 VHDL 行为模型	混用	VHDL 行为模型
	Verilog	从 DCP 生成的 Verilog 仿真网表
	VHDL	VHDL 行为模型
IP 不提供任何行为模型	混用、Verilog 和 VHDL	从 DCP 生成的网表 (`target_language`)

注释：

1. 行为仿真模型（如果可用）始终优先于结构化仿真模型。Vivado 工具会根据可用模型自动选择行为模型或结构化模型。自动选择无法覆盖。
2. 如果仿真 Tcl 可使用任一语言，请使用 `target_language` 属性：`set_property target_language VHDL [current_project]`
3. 当前尚不支持将 VHDL 设置为 AMD Versal™ 器件的目标语言。这会导致仿真中出错。

设置仿真运行时分辨率

在测试激励文件中使用 `` `timescale `` 设置仿真运行时分辨率。对 AMD 仿真模型使用较粗粒度的分辨率无法达成任何仿真器性能增益。（在 AMD 仿真模型中，大部分仿真时间用于增量周期，而增量周期不受仿真器分辨率影响。）

重要提示！使用 1 fs 的时间分辨率运行仿真。部分 AMD 原语组件（例如，GT）需要 1 fs 分辨率才能在功能仿真或时序仿真内正常工作。

如需了解有关“Settings”（设置）对话框中“Simulation Options”（仿真选项）的详细信息，请参阅 仿真选项。

重要提示！使用皮秒作为最小分辨率，原因是测试设备只能按最接近的皮秒分辨率来测量时间。

添加或创建仿真源文件

要将仿真源文件添加到 Vivado Design Suite 工程中，请执行以下操作：

1. 选择“File” → “Add Sources”（文件 > 添加源文件）或者单击 Flow Navigator 中的“Add Sources”（添加源文件）。

   这样会打开“Add Sources”Wizard（添加源文件向导）。

2. 选择“Add or Create Simulation Sources”（添加或创建仿真源文件），然后单击“Next”（下一步）。

   这样会打开“Add or Create Simulation Sources”对话框。选项为：

   - “Add Files”（添加文件）：调用文件浏览器以便您选择要添加到工程中的仿真源文件。
   - “Add Directories”（添加目录）：调用目录浏览器以便从所选目录添加所有仿真源文件。指定目录中含有效源文件扩展名的文件都将被添加到工程中。
   - “Create File”（创建文件）：调用“Create Source File”（创建源文件）对话框，以便您可在其中创建新的仿真源文件。如需了解有关工程源文件的更多信息，请参阅《Vivado Design Suite 用户指南：系统级设计输入》(UG895)。

- 此对话框侧边的按钮可供您执行以下操作：
  - “Remove”（移除）：从要添加的文件列表中移除所选源文件。
  - “Move Up”（上移）：将列表中的文件按顺序上移。
  - “Move Down”（下移）：将列表中的文件按顺序下移。
- 此向导中的复选框可提供下列选项：
  - “Scan and add RTL include files into project”（扫描 RTL include 文件并将其添加到工程中）：扫描已添加的 RTL 文件，并添加所有引用的 include 文件。
  - “Copy sources into project”（将源文件复制到工程中）：将原始的源文件复制到工程中，并在工程中使用此文件的本地复制版本。

    如果选择使用“Add Directories”命令来添加源文件的目录，那么将这些文件复制到工程本地时，会保留目录结构。
  - “Add sources from subdirectories”（从子目录添加源文件）：从“Add Directories”选项中指定的目录的子目录添加源文件。
  - “Include all design sources for simulation”（包含所有设计源文件用于仿真）：包含所有设计源文件用于仿真。

视频：如需获取该功能特性的演示，请观看 Vivado Design Suite QuickTake 视频：逻辑仿真。

处理仿真集

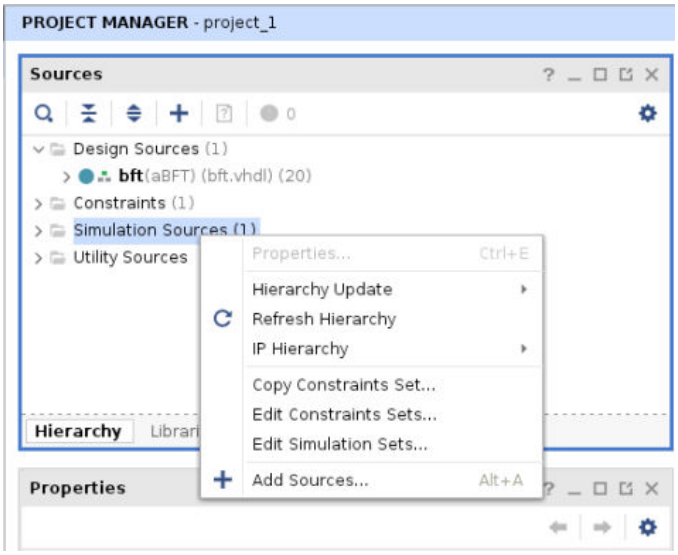
Vivado IDE 会将仿真源文件存储在“Sources”（源文件）窗口中的文件夹内显示的仿真集中，在本地工程目录内可远程引用或存储这些仿真集。

仿真集允许您定义不同的源文件，以供设计的不同阶段使用。例如，某一个测试激励文件可提供激励用于为细化的设计或设计模块执行行为仿真，另一个测试激励文件则可提供激励用于为实现的设计执行时序仿真。

向工程添加仿真源文件时，可以指定要使用的仿真源文件集。

要编辑仿真集，请执行以下操作：

1. 在“Sources”窗口弹出菜单中，右键单击“Simulation Sources”（仿真源文件），然后单击“Edit Simulation Sets”（编辑仿真集），如下图所示。



这样会打开“Add or Create Simulation Sources”Wizard（添加或创建仿真源文件向导）。

2. 在“Add or Create Simulation Sources”Wizard 中，选中 + “Add Sources”（添加源文件）。

   这样即可将工程关联的源文件添加到新创建的仿真集中。

3. 请按需添加其他文件。

所选仿真集用于活动的设计运行。

生成网表

要为已综合或已实现的设计运行仿真，请运行网表生成进程。网表生成 Tcl 命令可以提取已综合或已实现的设计数据库，并为整个设计写出单一网表。

当您使用 IDE 或 `launch_simulation` 命令启动仿真器时，Vivado Design Suite 会自动生成网表。

网表生成 Tcl 命令可以写入 SDF 和设计网表。Vivado Design Suite 提供了以下 Tcl 命令：

- `write_verilog`：Verilog 网表
- `write_vhdl`：VHDL 网表
- `write_sdf`：SDF 生成

提示：SDF 值只是设计进程早期（例如，综合期间）的估算值。随着设计进程的推进，数据库中可用信息增加时，时序数值的准确性也会提升。

生成功能网表

Vivado Design Suite 支持写出 Verilog 或 VHDL 结构网表用于进行功能仿真。此网表的目的是运行仿真（无时序）以检查结构网表的行为与期望的行为模型 (RTL) 仿真相匹配。

功能仿真网表是分层折叠网表，它扩展至原语模块或实体级别，其层级的最低层次由原语和宏原语组成。

这些原语包含在下列库中：

- UNISIMS_VER 仿真库，用于 Verilog 仿真
- UNISIMS 仿真库，用于 VHDL 仿真

在多数情况下，用于行为仿真的测试激励文件同样可用于执行更准确的仿真。

以下 Tcl 命令分别用于生成 Verilog 和 VHDL 功能仿真网表：

```
write_verilog -mode funcsim <Verilog_Netlist_Name.v>
write_vhdl -mode funcsim <VHDL_Netlist_Name.vhd>
```

生成时序网表

在 Vivado 工具完成最差情况下的布局布线延迟计算之后，您可以使用 Verilog 时序仿真来验证电路工作情况。

在多数情况下，用于功能仿真的测试激励文件同样可用于执行更准确的仿真。

将来自两次仿真的结果进行比较，验证您的设计的当前性能表现与最初指定要求是否相符。

生成时序仿真网表分两步：

1. 为设计生成仿真网表文件。
2. 生成 SDF 延迟文件并为其中所有时序延迟添加注解。

重要提示！ Vivado IDE 仅支持 Verilog 时序仿真。

提示： 如果您是 VHDL 用户，则可运行综合后和实现后功能仿真（在此情况下无需标准延迟格式 (SDF) 注解，仿真网表使用 UNISIM 库）。您可使用 write_vhdl Tcl 命令来创建网表。如需了解使用信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)。

用于生成时序仿真网表的 Tcl 语法如下所示：

```
write_verilog -mode timesim -sdf_anno true <Verilog_Netlist_Name>
```

使用 Versal CIPS VIP

AMD Versal™ 自适应 SoC Control, Interfaces, and Processing System (CIPS) Verification Intellectual Property (VIP) 支持对 Versal 自适应 SoC 应用执行功能仿真。其目标是通过模拟处理器系统 (PS)-PL 接口和 PS 逻辑的 OCM 存储器来支持对可编程逻辑 (PL) 进行功能验证。此 VIP 是以 SystemVerilog 模块数据包的形式提供的。VIP 操作是通过使用一连串 SystemVerilog 任务来控制的。在最新版本的 Vivado 中支持此操作。如需了解更多信息，请参阅《Versal 自适应 SoC CIPS Verification IP 数据手册》(DS996)。

第 3 章

使用第三方仿真器进行仿真

AMD Vivado™ Design Suite 支持使用第三方工具进行仿真。要使用第三方工具执行仿真，可直接在 Vivado 集成设计环境 (IDE) 内执行，或者也可以使用定制外部仿真环境来执行。

表 8：**受支持的第三方仿真器**

第三方仿真器	Red Hat 64 位 Linux	Windows 64 位
Siemens EDA ModelSim DE	支持	支持
Siemens EDA Questa Advanced Simulator	支持	支持
Cadence Xcelium Parallel Simulator	支持	不适用
Synopsys VCS	支持	不适用
Aldec Active HDL	不适用	支持
Aldec Riviera PRO	支持	支持

《Vivado Design Suite 用户指南：使用 Vivado IDE》(UG893) 描述了 Vivado IDE 的用法。

请先设置以下环境变量，然后在 Vivado IDE 内运行仿真。

表 9：**第三方仿真器的环境变量设置**

仿真器	Linux	Windows

Modelsim	`setenv MODEL_TECH <tool installation path>`
	`setenv LM_LICENSE_FILE <license file>`
	`setenv PATH ${MODEL_TECH}/bin:$PATH`
		`set MODEL_TECH=<tool installation path>`
		`set LM_LICENSE_FILE=<license file>`
		`set Path=%MODEL_TECH%\win32;%Path%`

Questa	`setenv MODEL_TECH <tool installation path>`
	`setenv LM_LICENSE_FILE <license file>`
	`setenv PATH ${MODEL_TECH}/bin:$PATH`
		`set MODEL_TECH=<tool installation path>`
		`set LM_LICENSE_FILE=<license file>`
		`set Path=%MODEL_TECH%\win32;%Path%`

Riviera	`In BASH`
	`source <install_path>/etc/setenv`
	`source <install_path>/etc/setgcc`
		`call <install_path>/etc/setenv.bat`
		`call <install_path>/etc/setgcc.bat`

表 9：**第三方仿真器的环境变量设置** (续)

仿真器	Linux	Windows

Active-HDL	不适用	`set ACTIVE_BIN=<tool installation path>`
		`set Path=%<Active_hdl install dir>%\BIN;%Path%`
		`set LM_LICENSE_FILE=<license file>`

Xcelium	`setenv CDS_INST_DIR <xcelium_install_dir>`
	`setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools/xcelium/lib:$LD_LIBRARY_PATH`
	`setenv PATH $CDS_INST_DIR/tools/xcelium/bin:$CDS_INST_DIR/tools/bin:$PATH`
	`setenv CDS_LICENSE_DIR <tool_license>`
		不适用

VCS	`setenv VCS_HOME <tool_install_path>`
	`setenv LM_LICENSE_FILE <license_file_path>`
	`setenv PATH ${VCS_HOME}/bin:${PATH}`
		不适用

注释：

1. 工具安装路径应添加到环境变量 `PATH` 中（与操作系统无关）。要为受支持的仿真器仿真基于 SystemC 的设计，请按 附录 F: Vivado IDE 中的 SystemC 支持 中所述提供所需的 g++ 版本安装路径。`LD_LIBRARY_PATH` 还应包含仿真器库路径。

如需了解有关第三方仿真器的更多信息，请参阅 指向第三方仿真器相关附加信息的链接 中提供的链接。

重要提示！仅限使用受支持的第三方仿真器版本。如需了解有关受支持的仿真器和操作系统的更多信息，请参阅《Vivado Design Suite 用户指南：版本说明、安装和许可》(UG973) 中的“兼容的第三方工具”表。

将第三方仿真器与 Vivado IDE 搭配使用来运行仿真

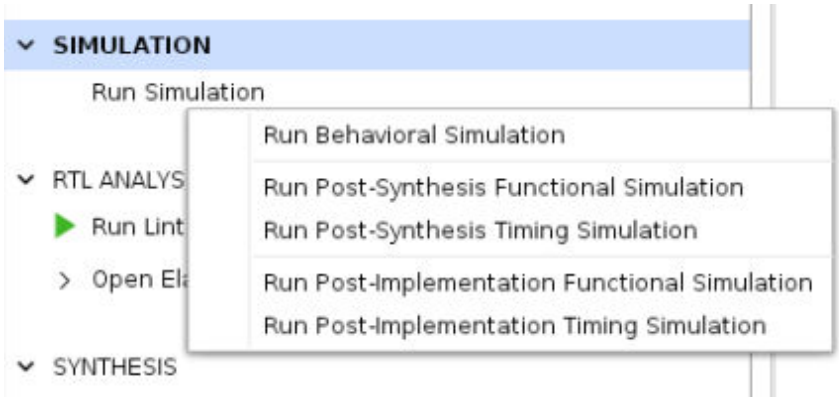
重要提示！运行第三方仿真前，请确认已编译的库位置（调用 `compile_simlib` 的路径或者您使用 `-directory` 选项指定的路径）。

在 Vivado IDE 中，您可以基于仿真设置来编译、细化和仿真设计，并在单独窗口中启动仿真器。

在设计综合前运行仿真时，仿真器会运行行为仿真。成功完成每个设计步骤（综合与实现）后，运行功能仿真或时序仿真的选项都会变为可用。您可从 Flow Navigator 或者输入 Tcl 命令来发起仿真运行。

在 Flow Navigator 中，单击“Run Simulation”（运行仿真）并选择要运行的仿真类型，如下图所示：

图 6：**仿真类型**



要使用对应的 Tcl 命令，请输入：`launch_simulation`。

提示：此命令提供了 `-scripts_only` 选项，可用于根据目标仿真器来编写 DO 或 SH 文件。使用 DO 或 SH 文件在 IDE 外部运行仿真。

注释：如果您当前在 Vivado 外部运行 VCS 仿真器，请确保使用 `-full64` 开关。否则，如果设计包含 AMD IP，仿真器就无法运行。

重要提示！请使用以下命令来运行的 32 位仿真器：`set_property 32bit 1 [current_fileset -simset]`

注释：AMD Verification IP (VIP) 使用 SystemVerilog 构造。如果您当前使用的任意 IP 会例化 VIP，请确保您的仿真器支持 SystemVerilog。

使用第三方工具运行时序仿真

提示：综合后时序仿真使用基于已综合的网表估算所得的时序延迟。实现后时序仿真使用的则是实际的时序延迟。

运行综合后和实现后时序仿真时，仿真器包括：

- 包含 SIMPRIMS 库组件的门级网表
- SECUREIP
- 标准延迟格式 (SDF) 文件

您在开始时定义整体设计功能。实现设计时，准确的时序信息即可供使用。

为创建网表和 SDF，Vivado Design Suite 会执行如下操作：

- 调用网表编写程序 `write_verilog` 搭配 `-mode timesim` 开关和 `write_sdf`（SDF 注解器）
- 将生成的网表发送到目标仿真器

您可使用“Simulation Settings”（仿真设置）来控制这些选项，如 使用仿真设置 中所述。

重要提示！仅限 Verilog 才支持综合后和实现后时序仿真。不支持 VHDL 时序仿真。如果您是 VHDL 用户，您可运行综合后和实现后功能仿真（在此情况下无需 SDF 注解，仿真网表使用 UNISIM 库）。您可使用 `write_vhdl` Tcl 命令来创建网表。如需了解使用信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)。

综合后时序仿真

成功运行综合后，“Run Simulation” → “Post-Synthesis Timing Simulation”（运行仿真 > 综合后时序仿真）选项就会变为可用。

选择综合后时序仿真后，就会生成时序网表和 SDF 文件。网表文件包含 `$sdf_annotate` 命令，因此即可选取生成的 SDF 文件。

实现后时序仿真

成功完成实现后操作时，“Run Simulation” → “Post-Implementation Timing Simulation”（运行仿真 > 实现后时序仿真）选项就会变为可用。

选择实现后时序仿真后，就会生成时序网表和 SDF 文件。网表文件包含 `$sdf_annotate` 命令，因此即可选取生成的 SDF 文件。

给 SDF 文件添加注解用于时序仿真

指定仿真设置时，已指定是否创建 SDF 文件，以及工艺角是设置为快速还是慢速。

提示：要查找 SDF 文件选项的设置，请在 Vivado IDE Flow Navigator 中，右键单击“Simulation”并选择“Simulation Settings”。在“Settings”（设置）对话框中，选中“Simulation”类别，然后单击“Netlist”（网表）选项卡。

基于指定的工艺角，SDF 文件会包含不同的 `min` 和 `max` 数值。

建议：要运行建立时间检查，请创建 SDF 文件并将 -process_corner 设为 slow，并使用来自该 SDF 文件的 max 列。

要运行保持时间检查，请创建 SDF 文件并将 -process_corner 设为 fast，并使用来自该 SDF 文件的 min 列。指定要使用的 SDF 延迟字段的方法取决于使用的仿真工具。请参阅特定仿真工具文档以获取有关如何设置该选项的信息。

要获取完整信息，请运行全部 4 项时序仿真，并指定如下设置：

1. 慢速角：SDFMIN 和 SDFMAX
2. 快速角：SDFMIN 和 SDFMAX

运行独立时序仿真

如果从 Vivado IDE 运行时序仿真，它会向仿真器添加时序仿真相关开关。如果运行独立时序仿真，那么请确保在细化期间将以下开关传递给仿真器：

对于 VCS：

```
+pulse_e/<number> and +pulse_r/<number>  +transport_int_delays
```

在细化期间（搭配 VCS）

对于 ModelSim/Questa Advanced Simulator：

```
+transport_int_delays +pulse_int_e/0 +pulse_int_r/0
```

在细化期间（搭配 vsim）

> **重要提示！**Vivado 仿真器模型使用互连延迟，因此，需要如下额外开关才能正确执行时序仿真：`-transport_int_delays -pulse_r 0 -pulse_int_r 0`。如需了解有关这些命令的描述，请参阅 表 15：xelab、xvhd 和 xvlog 命令选项。

转储 SAIF 用于功耗分析

“切换活动交换格式”(Switching Activity Interchange Format, SAIF) 是 ASCII 报告，有助于提取和存储由仿真器工具生成的切换活动信息。此切换活动可以反标注解回 AMD 功耗分析和最优化工具内，用于功耗测量和估算。

在 Questa Advanced Simulator/ModelSim 中转储 SAIF

Questa Advanced Simulator/ModelSim 使用显式功耗命令来转储 SAIF 文件，如下所示：

1. 请输入以下命令指定要转储的作用域或信号：

   ```
   power add <hdl_objects>
   ```

2. 按特定时间运行仿真（或者 `run -all`）。
3. 请输入以下命令转储输出功耗报告：

   ```
   power report -all filename.saif
   ```

如需了解有关每一条命令的详细用法或信息，请参阅 ModelSim 文档。

DO 文件示例

```
power add tb/fpga/*
run 500us
power report -all -bsaif routed.saif
quit
```

在 VCS 中转储 SAIF

VCS 提供了多项 power 命令用于生成具有特定要求的 SAIF。

1. 输入以下命令即可指定要生成的信号和作用域：

   ```
   power <hdl_objects>
   ```

2. 启用 SAIF 转储。您可使用仿真器工作空间内的命令行：

   ```
   power -enable
   ```

3. 针对特定时间运行仿真。

4. 输入以下命令即可禁用 power 转储并报告 SAIF：

```
power -disable
power -report filename.saif
```

如需了解有关每一条命令的详细用法或信息，请参阅 Synopsys VCS 文档。

如需了解有关切换活动交换格式 (SAIF) 的更多信息，请参阅 使用 Vivado 仿真器执行功耗分析。

转储 VCD

您可使用“Value Change Dump (VCD)”（值更改转储）文件来捕获仿真输出。Tcl 命令基于与转储值相关的 Verilog 系统任务。

在 Questa Advanced Simulator/ModelSim 中转储 VCD

Questa Advanced Simulator/ModelSim 使用显式 VCD 命令来转储 VCS 文件，如下所示：

1. 打开 VCD 文件：

```
vcd file my_vcdfile.vcd
```

2. 指定要转储的作用域或信号：

```
vcd add <hdl_objects>
```

3. 按指定的时间段运行仿真（或者 `run -all`）。

如需了解有关每一条命令的详细用法或信息，请参阅 ModelSim 文档。

DO 文件示例：

```
vcd file my_vcdfile.vcd
vcd add -r tb/fpga/*
run 500us
quit
```

在 VCS 中转储 VCD

在 VCS 中，您可以使用 `dumpvar` 命令生成 VCD 文件。请指定文件名和实例名称（默认是其完整层级）。

```
vcs +vcs+dumpvars+test.vcd
```

IP 仿真

在以下示例中，`accum_0.xci` 文件是您从 AMD Vivado™ IP 目录生成的 IP。请使用以下命令在 VCS 中仿真此 IP：

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>[current_project]
launch_simulation -noclean_dir -of_objects [get_files accum_0.xci]
```

在集成仿真运行期间使用自定义 DO 文件

如果您有一组特定的命令（自定义 DO 文件）要在运行仿真之前调用，请将这些命令添加到一个文件中，并使用相应的命令传递此文件，如下所示：

在 Questa Advanced Simulator 中

```
set_property -name {questa.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects [get_filesets sim_1]
```

在 Modelsim 中

```
set_property -name {modelsim.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects [get_filesets sim_1]
```

在 VCS 中

```
set_property -name {vcs.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>} -objects
[get_filesets sim_1]
```

在 Xcelium 中

```
set_property -name {xcelium.simulate.tcl.post} -value
{<AbsolutePathOfFileLocation>}
-objects [get_filesets sim_1]
```

ModelSim 和 Questa Advanced Simulator 的仿真步骤控制构造

下表概括了用于基于 .do 文件格式控制步骤执行的构造：

- 本机 .do 文件：这是默认 .do 文件格式。在此格式下，编译和细化 shell 脚本会调用 `source <tb>_compile/elaborate.do`。例如：

  ```
  source bft_tb_compile.do 2>&1 | tee -a compile.log
  ```

  仿真脚本会调用 `vsim -64 -c -do do {<tb>_simulate.do}`。例如：

  ```
  $bin_path/vsim -64 -c -do do {bft_tb_simulate.do} -l simulate.log
  ```

- 经典 .do 文件：经典 .do 文件格式不同于编译和细化 shell 脚本中的本机 .do 文件。仿真脚本并无更改。在编译和细化 shell 脚本中，它会调用 `vsim -c -do do {<tb>_compile/elaborate.do}`。例如：

  ```
  $bin_path/vsim -64 -c -do do {bft_tb_compile.do} -l compile.log
  ```

  要完成此操作，请在 Tcl 控制台命令上调用 `set_param project.writeNativeScriptForUnifiedSimulation 0` 将 `project.writeNativeScriptForUnifiedSimulation` 设置为 0。

此文件格式对于共享工程很有用，因为 Questa Advanced Simulator/ModelSim 实用工具的路径会硬编码到 shell 脚本内。

表 10：**仿真步骤控制构造参数**

参数	描述	默认
`project.writeNativeScriptForUnifiedSimulation`	仅使用仿真器命令编写纯 .do 文件（无 Tcl 或 Shell 构造）。	0 (false)
`simulator.quitOnSimulationComplete`	在仿真器完成 ModelSim/Questa Advanced Simulator 仿真时，退出仿真器。要禁用退出，请将此参数设为 false。	1 (true)
`simulator.modelsimNoQuitOnError`	默认情况下，ModelSim/Questa Advanced Simulator 仿真出现错误或中断时不会退出。要在遇到错误或中断时退出仿真，请将此参数设为 false。	1 (true)

解释

- `simulator.quitOnSimulationComplete`：默认情况下，生成的 `simulate.do` 包含 `quit -force`。当仿真在指定时间内完成时，仿真器就会退出。如果您不希望仿真器退出，请通过调用 `set_peram simulator.quitOnSimulationComplete 0` 将 `simulator.quitOnSimulationComplete` 设置为 0。
- `simulator.modelsimNoQuitOnError`：默认情况下，仿真器遇到错误或中断时并不会退出。如果希望退出仿真器，请设置以下参数：

  ```
  set_param simulator.modelsimNoQuitOnError 0
  ```

  这样会在 `<tb>_simulate.do` 中添加以下 2 行。

  ```
  onbreak {quit -f} onerror {quit -f}
  ```

在批处理模式下运行第三方仿真器

Vivado Design Suite 支持使用批处理或脚本仿真来进行第三方验证。收集完设计文件并生成脚本以支持目标仿真器后，您可检验脚本并将其整合到自己的验证环境内。AMD 建议您使用 `export_simulation` 脚本作为仿真流程起点，而不是构建定制 API 来生成脚本。如需了解有关导出仿真脚本的更多信息，请参阅 导出仿真文件和脚本。

请确保先为仿真器正确设置环境，而后再运行脚本。如需了解有关仿真器配置的更多信息，请参阅 使用仿真设置。如需了解有关运行批处理模式或脚本模式的详细信息，请参阅您的特定仿真器的对应用户指南。

第 4 章

使用 Vivado 仿真器进行仿真

Vivado 仿真器是硬件描述语言 (HDL) 事件驱动型仿真器，支持为 VHDL、Verilog、SystemVerilog (SV) 及混合 VHDL/Verilog 或 VHDL/SV 设计提供功能仿真和时序仿真。

Vivado 仿真器支持下列功能特性：

- 源代码调试（步进、断点、当前值显示）
- 用于时序仿真的 SDF 注解
- VCD 转储
- SAIF 转储，用于功耗分析和最优化
- 针对硬核 IP 块（例如，串行收发器和 PCIe®）的原生支持
- 多线程编译
- 混合语言（VHDL、Verilog 或 SystemVerilog 设计构造）
- 单击式仿真重新编译和重新启动
- 单击式编译和仿真
- 内置 AMD 仿真库支持
- 实时波形更新

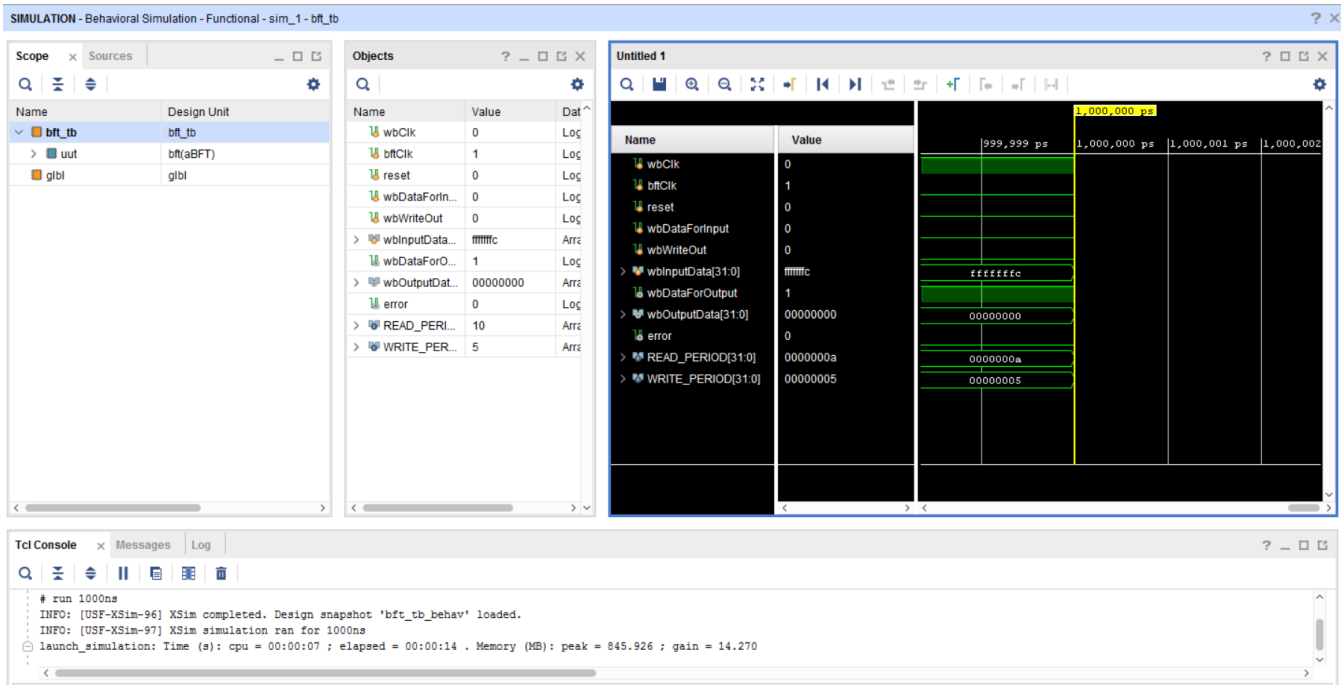
如需获取 Vivado 仿真运行方式的分步式演示，请参阅《Vivado Design Suite 教程：逻辑仿真》(UG937)。

运行 Vivado 仿真器

重要提示！如果您使用的是 Vivado 仿真器，请务必先为设计指定所有相应的工程设置，然后再运行仿真。如需了解受支持的第三方仿真器，请参阅 第 3 章：使用第三方仿真器进行仿真。

在 Flow Navigator 中单击“Run Simulation”（运行仿真）并选择仿真类型以调用 Vivado 仿真器工作空间，如下图所示。

图 7：**Vivado 仿真器工作空间**



主工具栏

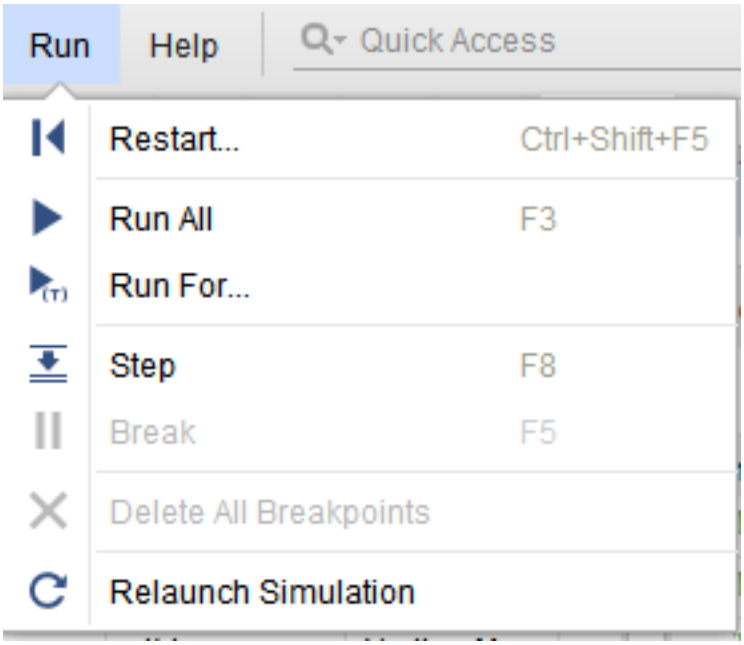
主工具栏可为 Vivado IDE 中最常用的命令提供单击式访问。只要您悬停在某个选项上，就会出现工具提示，以提供更多信息。

“Run”菜单

此菜单提供的选项与 Vivado IDE 相同，并在您运行仿真后添加了一个“Run”（运行）菜单。

仿真的“Run”菜单如下图所示。

图 8：**仿真的“Run”菜单选项**



Vivado 仿真器的“Run”菜单选项：

- “Restart”（重新启动）：允许您从时间 0 开始重新启动现有仿真。Tcl 命令：`restart`
- “Run All”（全部运行）：允许您运行未完成的仿真直至其完成。Tcl 命令：`run -all`
- “Run For”（运行持续时间）：允许您指定仿真运行的时间。Tcl 命令：`run <time>`

> **提示：**虽然您始终可以在 run 命令中指定时间单位（例如，`run 100 ns`），但也可以省略时间单位。如果省略时间单位，Vivado 仿真器会假定采用 TIME_UNIT Tcl 属性的时间单位。要查看 TIME_UNIT 属性，请使用 Tcl 命令 `get_property time_unit [current_sim]`。要更改 TIME_UNIT 属性，请使用 Tcl 命令 `set_property time_unit <unit> [current_sim]`，其中 <unit> 为下列选项之一：fs、ps、ns、us、ms 和 s。

- “Step”（步进）：运行仿真直至下一行 HDL 源代码为止。
- “Break”（中断）：允许您暂停运行中的仿真。
- “Delete All Breakpoints”（删除所有断点）：删除所有断点。
- “Relaunch Simulation”（重新启动仿真）：重新编译仿真文件并重新启动仿真。

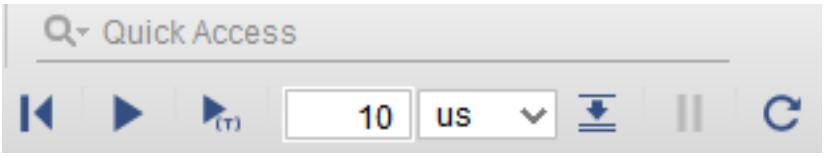
相关信息

设计更改（重新启动）后重新运行仿真

仿真工具栏

运行 Vivado 仿真器时，会在主工具栏右侧打开仿真专用工具栏（如下图所示）。

图 9：**仿真工具栏**



这里提供了便于您使用的各种按钮，这些按钮与上文 “Run”菜单 中所标记的按钮相同，但不含“Delete All Breakpoints”（删除所有断点）选项。

仿真工具栏按钮描述

悬停在工具栏按钮上即可出现工具提示描述。

- “Restart”（重新启动）：将仿真时间复位为 0。
- “Run all”（全部运行）：运行仿真直至完成所有事件或者直至 HDL 语句表明仿真应停止。
- “Run For”（运行持续时间）：按指定的时间段运行。
- “Step”（步进）：运行仿真直至出现下一条 HDL 语句为止。
- “Break”（中断）：暂停当前仿真。
- “Relaunch”（重新启动）：重新编译仿真源文件，并重新启动仿真（例如，执行代码更改后）。

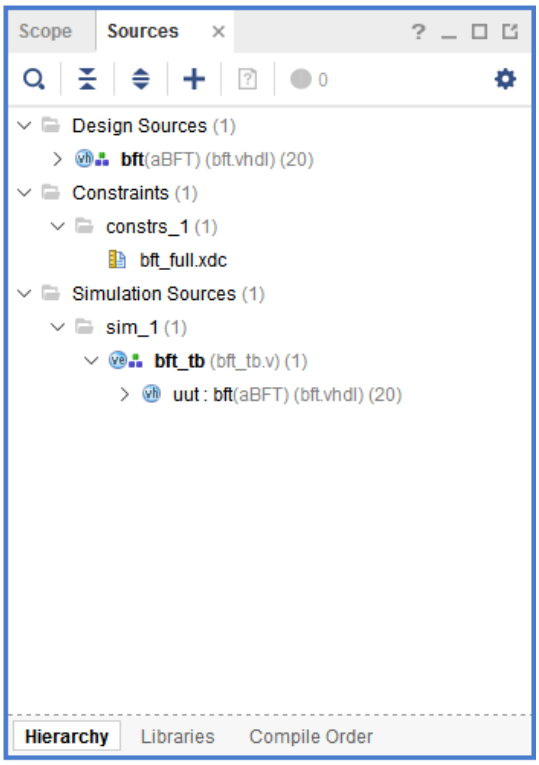
相关信息

设计更改（重新启动）后重新运行仿真

“Sources”窗口

“Sources”（源文件）窗口以分层树状结构显示了仿真源文件，其中包含用于显示“Hierarchy”（层级）、“IP Sources”（IP 源文件）、“Libraries”（库）和“Compile Order”（编译顺序）的各视图，如下图所示。

图 10：**“Sources”窗口**



只需将鼠标悬停在各“Sources”按钮上，工具提示即可显示各按钮的描述信息。这些按钮支持您对文件进行检验、展开、折叠、添加、打开、筛选和滚动操作。

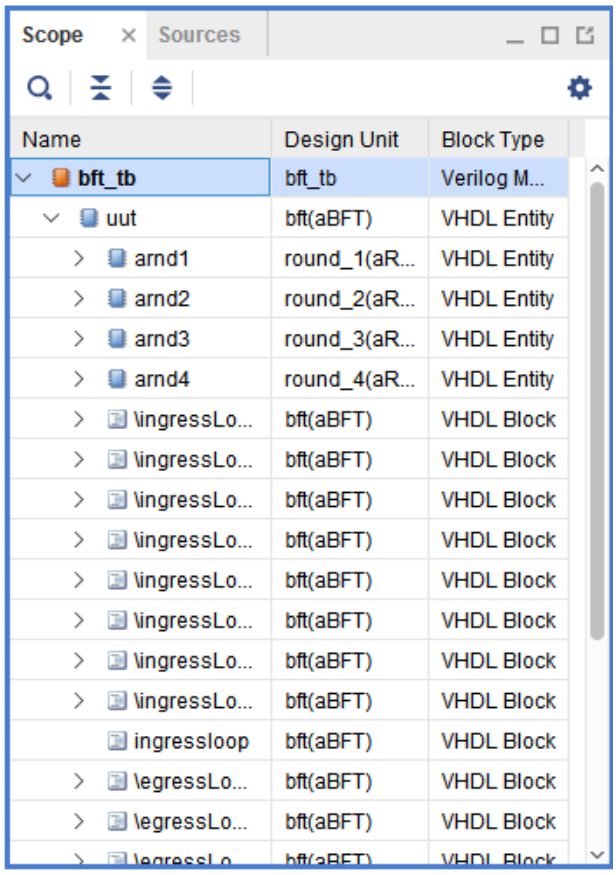
您也可以右键单击源对象并选择“Open File”（打开文件）或“Add Sources”（添加源文件）选项来打开或添加源文件。

“Scope”窗口

作用域是 HDL 设计的分层分区。例化设计单元或定义进程、块、包或子程序时，就会创建作用域。

在下图所示“Scope”（作用域）窗口中，您可看到设计层级。在“Scope”窗口中选中作用域时，会在“Objects”（对象）窗口中显示来自该作用域的所有 HDL 对象。您可在“Objects”窗口中选择 HDL 对象，并将其添加到波形查看器中。

图 11：**“Scope”窗口**



筛选作用域

- 单击作用域子菜单上的“Settings”（设置）选项即可在显示或隐藏（选中或取消选中）对应作用域类型之间进行切换。

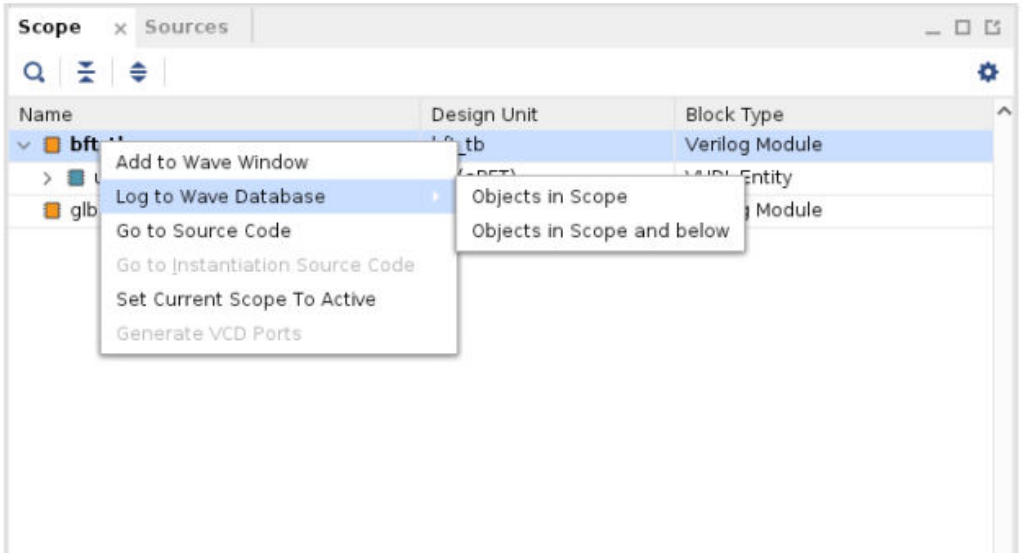
> **提示：**使用“Setting”选项隐藏作用域时，该作用域内的所有作用域（无论任何类型）也会一并隐藏。例如，在前图中，单击“Verilog Module”（Verilog 模块）按钮会隐藏所有 Verilog 模块时，不仅会隐藏 `bft_tb` 作用域，还会隐藏 `uut`（即使 `uut` 是 VHDL 实体作用域也是如此）。

- 要将显示的作用域限制为包含指定字符串的作用域，请单击“Search”（搜索）按钮 并在文本框中输入该字符串。

“Objects”窗口中显示的对象会根据当前作用域而变（或者经筛选后显示）。选择当前作用域即可更改“Objects”窗口中的对象。

右键单击作用域时，会显示包含下列选项的菜单（如下图所示）：

图 12： **“Scope”窗口选项**



- “Add to Wave Window”（添加到波形窗口）：将所选作用域的所有可查看的 HDL 对象添加到波形配置中。

  提示：大位宽的 HDL 对象可能减慢波形查看器的显示速度。在发出“Add to Wave Window”命令之前，您可在波形配置上设置显示限制来过滤掉此类对象。要设置显示限制，请使用 Tcl 命令 `set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]`。

  “Add to Wave Window”命令可能会添加另一组 HDL 对象，这些对象不同于“Objects”窗口中的对象。除了所选作用域内直接定义的对象外，在“Scope”窗口中选择作用域时，“Objects”窗口还可能会显示来自外围作用域的 HDL 对象。相反，“Add to Wave Window”命令仅添加来自所选作用域的对象。

  或者，您可将“Objects”窗口中的项拖放到“Wave”（波形）窗口中的“Name”（名称）列内。

  重要提示！ “Wave”窗口显示的对象值会从添加该对象的仿真时间开始随时间而变。

  提示：要显示插入时间之前的对象值，必须重新启动仿真。为避免因未能捕获值的变化而不得不重新启动仿真，请在仿真运行开始时发出 `log_wave -r /` Tcl 命令，这样即可捕获设计中可显示的所有 HDL 对象的值变化。如需了解更多信息，请参阅 使用 log_wave Tcl 命令。

  对波形配置执行的更改（包括创建波形配置或添加 HDL 对象）仅在您保存 WCFG 文件后才会变为永久生效。

- “Go To Source Code”（转至源代码）：在定义所选作用域时打开源代码。
- “Go To Instantiation Source Code”（转至例化源代码）：对于 Verilog 模块和 VHDL 实体实例，该选项用于在例化所选实例时打开源代码。
- “Set Current Scope to Active”（将当前作用域设为活动作用域）：将当前作用域设为选定的作用域。选定的作用域会变为活动的仿真作用域（即，`get_property active_scope [current_sim]`）。处于活动状态的仿真作用域是 HDL 进程作用域，在此作用域内仿真当前处于暂停状态。使用 Vivado 仿真器时，如果在设置中禁用“follow active scope”（遵循活动作用域），那么该仿真器会记住最后一个 current_scope 选择，即使继续执行仿真也是如此。命中断点时，current_scope 仍会指向设为活动作用域的最后一个作用域。
- “Log to Wave Database”（记录到波形数据库内）：您可记录以下任意对象：
  - 当前作用域的对象。
  - 当前作用域及其下层的所有作用域的对象。

提示：默认情况下，Vivado 仿真器禁止记录大型 HDL 对象。要更改所记录的对象的大小限制，请使用 `set_property trace_limit <size> [current_sim]` Tcl 命令，其中 `<size>` 是 HDL 对象中标量元素的数量。

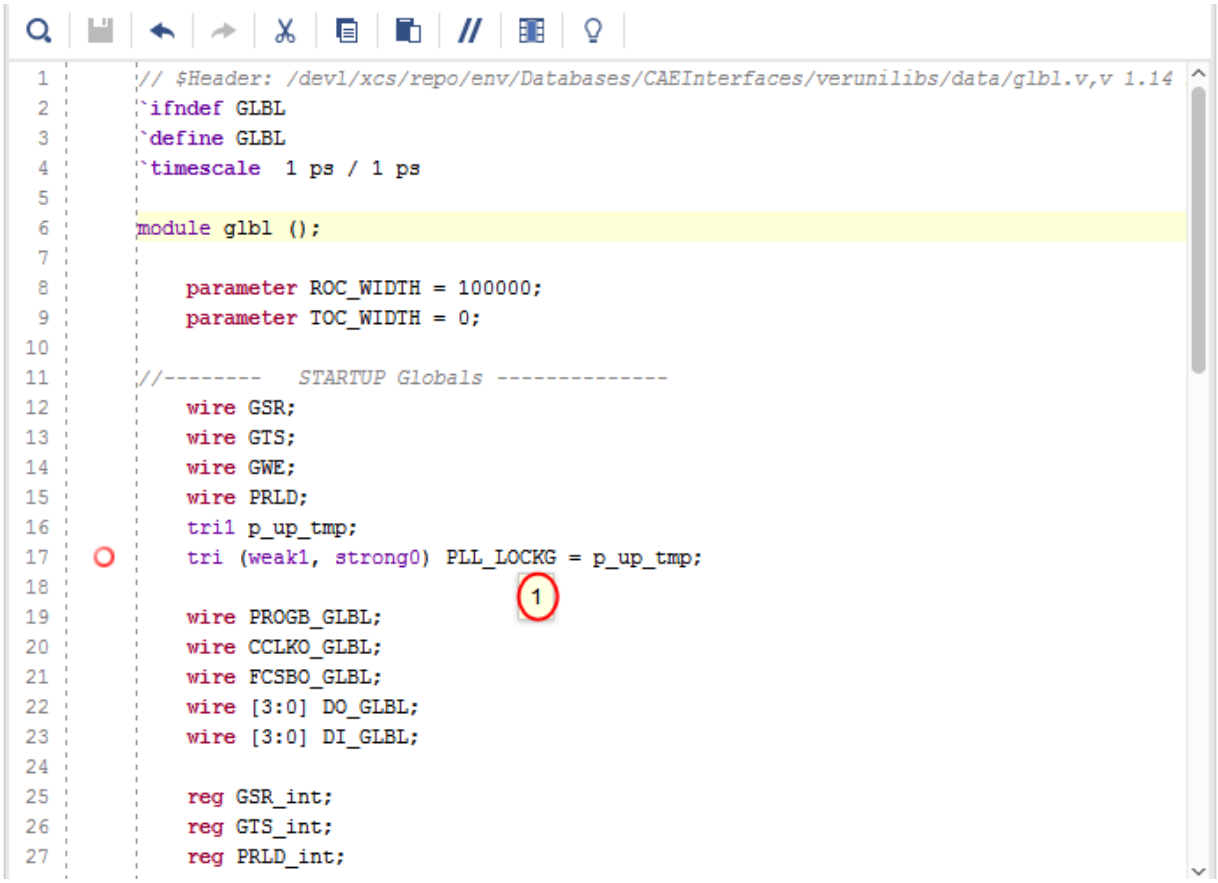
在源代码文本编辑器中，您可悬停在代码中的标识上以获取其值，如 “Scope” 窗口 中所示。

重要提示！要正常使用该功能，请确保您的作用域与“Scope”窗口中所选源代码已关联。

提示：由于顶层模块不执行例化，因此选中顶层模块时，前图所示“Go to Instantiation Source Code”右键单击选项会变为灰显。

提示：`log_wave` 可用于记录当前作用域或下层作用域的对象。仿真后，您可在波形上添加任意对象，并可看到从时间 0 开始直至当前仿真为止的绘图。

图 13：**已显示标识值的源代码**



其他作用域和源代码选项

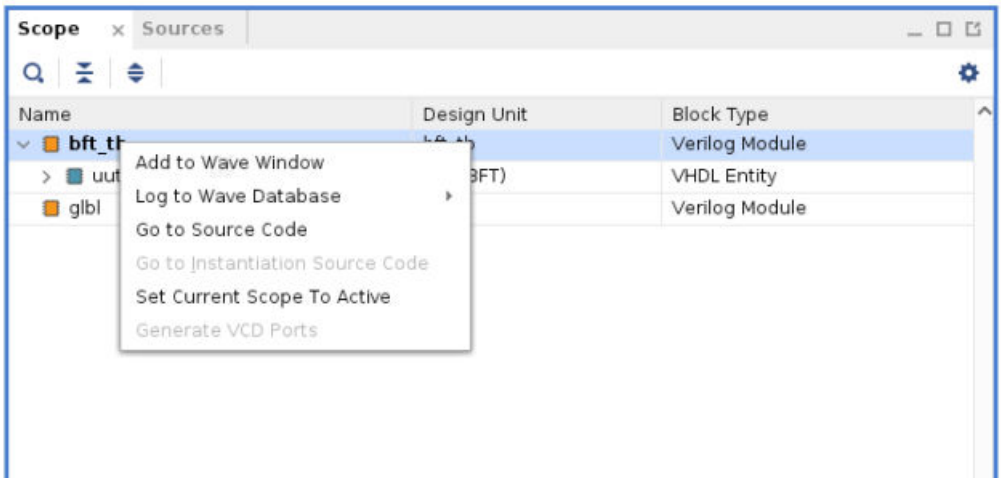
在“Scope”窗口或“Sources”窗口中，选中“Show Search”（显示搜索）按钮 时会显示搜索字段。

您也可以在 Tcl 控制台中输入以下命令来浏览 HDL 设计，效果与使用“Scope”窗口和“Objects”窗口相同：

```
get_scopes
current_scope
report_scopes
report_values
```

提示：要访问源文件以进行编辑，可以在“Scope”窗口或“Objects”窗口中选中“Go to Source Code”来打开这些文件，如 “Scope”窗口 中所示。

图 14： **“Scope”窗口的上下文菜单**

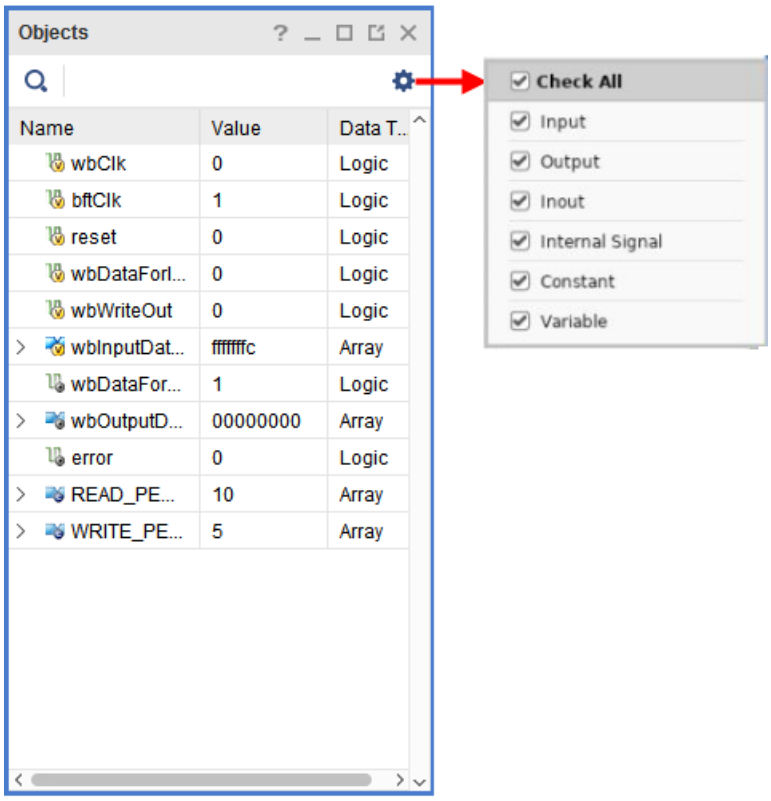


提示：完成编辑源代码并保存文件后，可以单击“Relaunch”（重新启动）按钮 来重新编译并重新启动仿真，而无需关闭再重新打开仿真。

“Objects”窗口

“Objects”（对象）窗口显示了与“Scope”（作用域）窗口中所选作用域关联的 HDL 仿真对象，如下图所示。

图 15： **“Objects”窗口**



HDL 对象旁的图标显示了每个对象的类型或端口模式。此视图列出了仿真对象的“Name”（名称）、“Value”（值）和“Data Type”（数据类型）。

您可在 Tcl 控制台中输入以下命令，获取对象的当前值。

```
get_value <hdl_object>
```

提示：要限制显示的矢量位数，请使用 `set_property array_display_limit <bits> [current_sim]` 命令，其中 <bits> 表示要显示的位数。

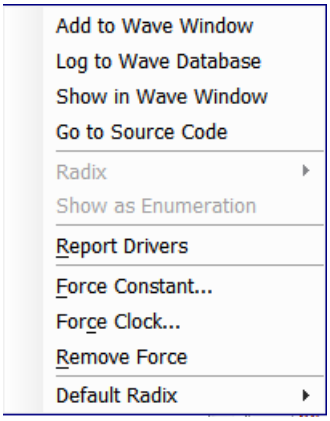
“Objects”窗口顶部可用的选项如下所示。单击“Settings”（设置）即可查看“Objects”窗口中所选对象。请使用该选项来筛选或限制“Objects”窗口的内容。

- “Search”（搜索）：您可使用“Search”选项来搜索对象名称。
- “Settings”（设置）：“Settings”选项允许您显示或隐藏“Objects”窗口中的各 HDL 对象。

对象上下文菜单

右键单击“Objects”窗口中的对象时，会显示上下文菜单，如 “Objects”窗口 中所示。上下文菜单中的选项如下所述。

图 16： **“Objects”窗口中的上下文菜单**



- “Add to Wave Window”（添加到波形窗口）：将所选对象添加到波形配置中。或者您也可以将这些对象从“Objects”窗口拖放到波形窗口的“Name”（名称）列中。
- “Log to Wave Database”（记录到波形数据库内）：将所选对象的事件记录到波形数据库 (WDB) 内，以便稍后在波形窗口中查看。

  提示：默认情况下，Vivado 仿真器禁止记录大型 HDL 对象。要更改所记录的对象的大小限制，请使用 `set_property trace_limit <size> [current_sim]` Tcl 命令，其中 `<size>` 是 HDL 对象中标量元素的数量。

- “Show in Wave Window”（显示在波形窗口中）：在波形窗口中高亮显示所选对象。
- “Default Radix”（默认基数）：在“Objects”窗口和文本编辑器中为所有对象设置默认基数。默认基数为十六进制。您可通过上下文菜单更改该选项。

  Tcl 命令：

  ```
  set_property radix <new radix> [current_sim]
  ```

  其中，`<new radix>` 为以下任意值：

  - bin
  - unsigned（适用于无符号十进制）
  - hex
  - dec（适用于有符号十进制）
  - ascii
  - oct
  - smag（适用于有符号量级）

  注释：如需更改个别信号的基数，请使用上下文菜单中的基数选项。

- “Radix”（基数）：在“Objects”窗口以及源代码窗口中显示所选对象的值时，该选项可用于选择数字格式。

  您可按如下方式更改个别对象的基数：

  1. 在“Objects”窗口中右键单击对象。
  2. 在上下文菜单中，选择要使用的“Radix”和格式：

- 默认
- 二进制
- 十六进制
- 八进制
- ASCII
- 无符号十进制
- 有符号十进制
- 有符号量级

提示：如果您在“Objects”窗口中更改基数，此更改不会反映在波形窗口中。

- “Show as Enumeration”（显示为枚举）：选择该选项可使用枚举标签来显示 SystemVerilog 枚举信号或变量的值。

  注释：仅针对 SystemVerilog 枚举才会启用此菜单项。如不选中此项，则会根据对象所设置的基数，以数值方式显示枚举对象的所有值。如选中此项，那么枚举声明已定义标签的值就会显示标签文本，而所有其他值则显示数值。
- “Report Drivers”（报告驱动程序）：在 Tcl 控制台中显示 HDL 进程的报告，这些进程用于向所选对象赋值。
- “Go To Source Code”（转至源代码）：在定义所选对象时打开源代码。
- “Force Constant”（强制设为常量）：强制将所选对象设为常量值。
- “Force Clock”（强制设为时钟）：强制将所选对象设为振荡值。
- “Remove Force”（移除强制）：移除所选对象上的任意强制设置。

提示：如果您观察到某些 HDL 对象并未显示在“Waveform Viewer”（波形查看器）中，这是因为 Vivado 仿真器不支持对某些 HDL 对象进行波形追踪，例如，Verilog 和本地变量中的指定事件。

相关信息

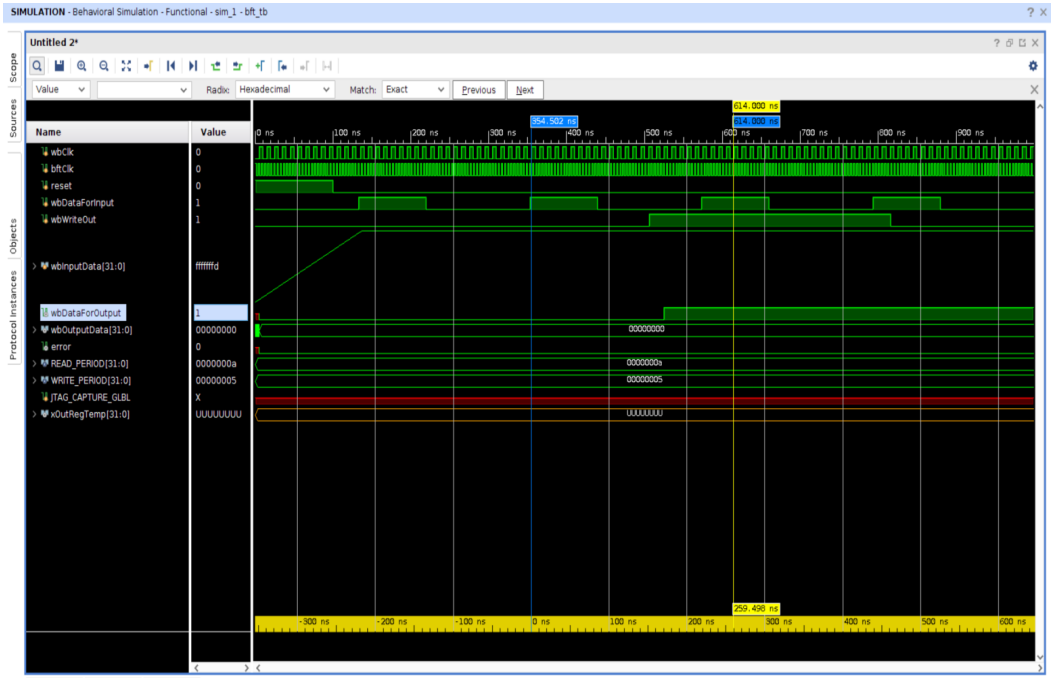
使用 Force 命令

“Wave”窗口

调用仿真器时，默认情况下它会打开“Wave”（波形）窗口。“Wave”窗口会显示由来自仿真的顶层模块的可追踪 HDL 对象组成的新波形配置，如 “Wave”窗口 中所示。

提示：关闭和重新打开工程时，必须重新运行仿真才能查看波形窗口。但如果在仿真处于活动状态时意外关闭默认波形窗口，则可从主菜单中选择“Window” → “Waveform”（窗口 > 波形）来将其复原。

图 17：**“Wave”窗口**



要将个别 HDL 对象或者一组对象添加到波形窗口中：请在“Objects”（对象）窗口中右键单击一个或多个对象，然后从上下文菜单中选择“Add to Wave Window”（添加到波形窗口）选项，如 “Objects”窗口 中所示。

要使用 Tcl 命令添加对象，请输入：`add_wave <HDL_objects>`。

使用 `add_wave` 命令可以指定到 HDL 对象的完整路径或相对路径。

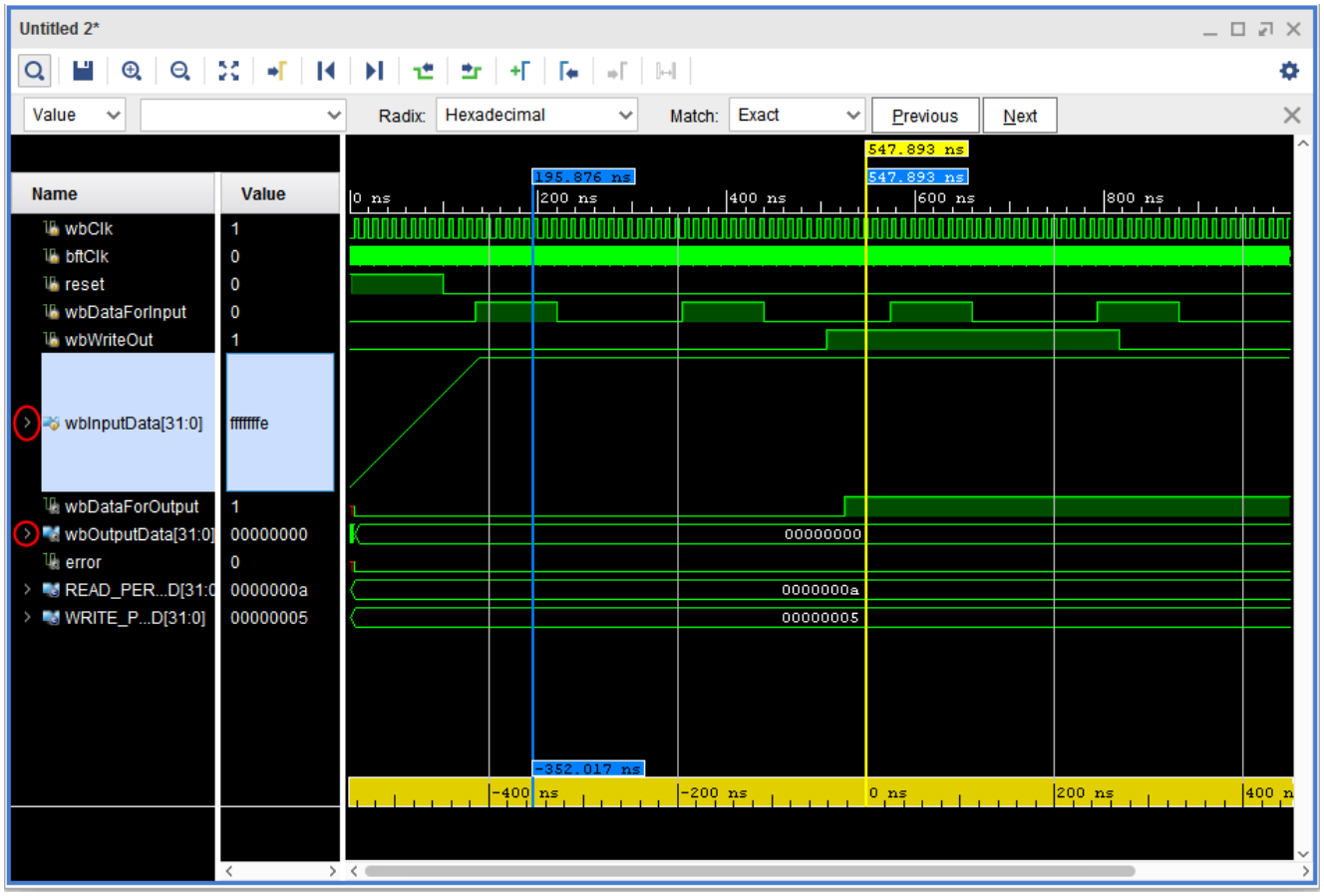
例如，如果当前作用域为 `/bft_tb/uut`，那么到 `uut` 下的复位寄存器的完整路径是：`/bft_tb/uut/reset`；相对路径则是 `reset`。

提示： `add_wave` 命令可接受 HDL 作用域和 HDL 对象。将 `add_wave` 与作用域搭配使用等效于使用“Scope”（作用域）窗口中的“Add To Wave Window”命令。大位宽的 HDL 对象可能减慢波形查看器的显示速度。在发出“Add to Wave Window”命令之前，您可在波形配置上设置显示限制来过滤掉此类对象。要设置显示限制，请使用 Tcl 命令 `set_property DISPLAY_LIMIT <maximum bit width> [current_wave_config]`。

波形对象

Vivado IDE 的“Wave”（波形）窗口是各种 Vivado Design Suite 工具中常用的窗口。波形配置中的波形对象示例如下图所示。

图 18：**波形中的 HDL 对象**



“Wave”窗口显示了 HDL 对象、其值、其波形，以及用于组织这些 HDL 对象的项目，例如，分组、分频器和虚拟总线。

这些 HDL 对象和组织项目统称为波形配置。“Wave”窗口的显示了用于时序测量的其他项目，包括，游标、标记和时间刻度标尺。

仿真期间，Vivado IDE 会在“Wave”窗口内跟踪 HDL 对象值发生的更改，您可以使用波形配置来检验仿真结果。

设计层级和仿真波形并不包含在波形配置内，而是存储在单独的波形数据库文件 (WDB) 内。

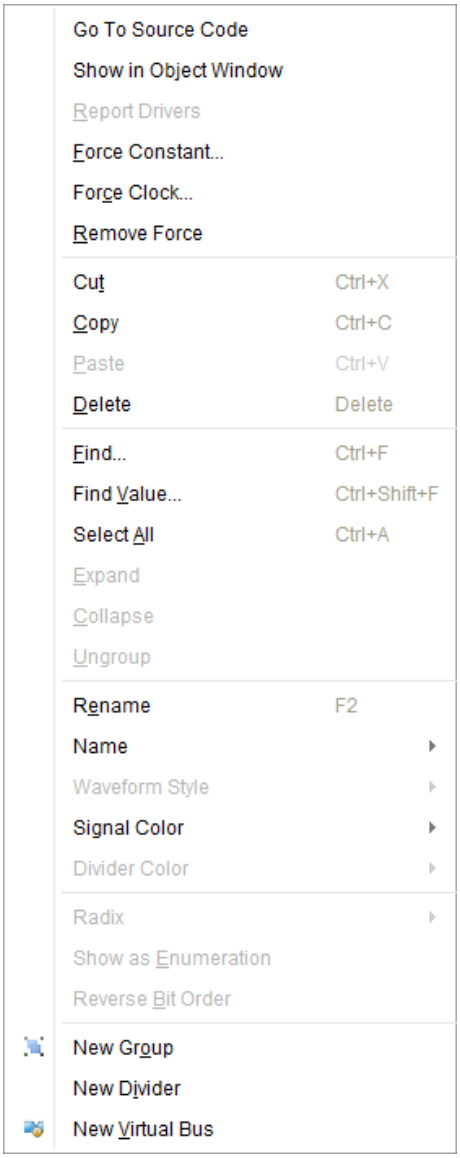
“Wave”窗口的上下文菜单

右键单击“Wave”窗口中的某个对象时，会显示其上下文菜单，如下图所示。如需了解有关波形中 HDL 对象的更多信息，请参阅 认识波形配置中的 HDL 对象。上下文菜单中的选项如下所述：

- “Go To Source Code”（转至源代码）：在定义所选设计波形对象时打开源代码。
- “Show in Object Window”（在对象窗口中显示）：在“Objects”（对象）窗口中显示设计波形对象的 HDL 对象。
- “Report Drivers”（报告驱动程序）：在 Tcl 控制台中显示 HDL 进程的报告，这些进程用于向所选波形对象赋值。
- “Force Constant”（强制设为常量）：强制将所选对象设为常量值。

- “Force Clock”（强制设为时钟）：强制将所选对象设为振荡值。
- “Remove Force”（移除强制）：移除所选对象上的任意强制设置。
- “Find”（查找）：在“Waveform”窗口中打开“Find Toolbar”（查找工具栏），按名称搜索波形对象。
- “Find Value”（查找值）：在“Waveform”窗口中打开“Find Toolbar”，以搜索波形值。
- “Select All”（全选）：在“Waveform”窗口中搜索所有波形对象。
- “Expand”（展开）：显示所选波形对象的子对象。
- “Collapse”（折叠）：隐藏所选波形对象的子对象。
- “Ungroup”（取消分组）：将所选分组或虚拟总线解包。
- “Rename”（重命名）：更改所选波形对象的显示名称。
- “Name”（名称）：将所选波形对象的显示名称更改为显示完整的分层名称（长名称）、简单的信号或总线名称（短名称）或者自定义名称。
- “Waveform Style”（波形样式）：将所选设计波形对象的波形更改为数字格式或模拟格式。
- “Signal Color”（信号颜色）：设置所选设计波形对象的波形颜色。
- “Divider Color”（分频器颜色）：设置所选分频器的各条状颜色。
- “Radix”（基数）：设置基数，以便在其中显示所选设计波形对象的值。
- “Show as Enumeration”（显示为枚举）：尽可能将所选择 SystemVerilog 枚举波形对象的值显示为枚举器标签以代替数字。
- “Reverse Bit Order”（反转位顺序）：将所选阵列波形对象显示的值的位顺序进行反转。
- “New Group”（新建组合）：将所选波形对象打包到类似文件夹的组合波形对象中。
- “New Divider”（新建分频器）：在“Waveform”窗口的波形对象列表中，创建水平分隔符。
- “New Virtual Bus”（新建虚拟总线）：创建新的逻辑矢量波形对象，该对象由所选设计波形对象的位组成。
- “Cut”（剪切）：允许您在“Waveform”窗口中剪切信号。
- “Copy”（复制）：允许您在“Waveform”窗口中复制信号。
- “Paste”（粘贴）：允许您在“Waveform”窗口中粘贴信号。
- “Delete”（删除）：允许您在“Waveform”窗口中删除信号。

图 19：**波形对象窗口的上下文菜单**



如需了解有关使用“Waveform”窗口的更多信息，请参阅 第 5 章：使用 Vivado 仿真器对仿真波形进行分析。

保存波形配置

新的波形配置并不会自动保存到硬盘上。请选择“File” → “Simulation Waveform” → “Save Configuration As”（文件 > 仿真波形 > 将配置另存为）并提供文件名以保存 WCFG 文件。

要将波形配置保存到 WCFG 文件，请输入 Tcl 命令 `save_wave_config <filename.wcfg>`。

指定的命令实参会为该 WCFG 文件命名并保存。

重要提示！缩放设置不随波形配置一起保存。

相关信息

使用模拟波形
更改 SystemVerilog 枚举的格式
波形组织
波形对象命名样式
使用 Force 命令
在波形配置中搜索值
信号与对象分组
反转总线位顺序

创建和使用多个波形配置

在仿真会话中，您可创建和使用多个波形配置，每个配置都单独显示在其“Wave”（波形）窗口中。如果同时显示多个“Wave”窗口，那么最近创建或者最近使用的窗口将作为活动窗口。所谓活动窗口需满足下列条件：该“Wave”窗口当前可见，并且窗口外部的命令也会应用于该窗口。例如：“HDL Objects” → “Add to Wave Window”（HDL 对象 > 添加到波形窗口）。

您可单击窗口标题以将不同“Wave”窗口设置为活动窗口。

相关信息

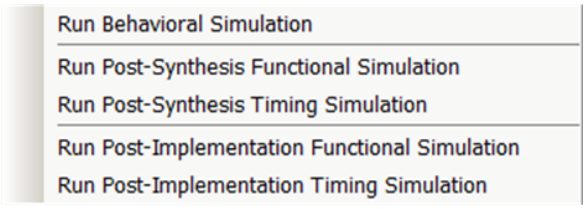
区分多轮仿真运行
创建新的波形配置

运行功能仿真和时序仿真

在 Vivado Design Suite 中创建完工程后，您即可立即运行行为仿真。成功运行综合和/或实现后，您可在自己的设计上运行功能仿真和时序仿真。要运行仿真，请在 Flow Navigator 中选中“Run Simulation”（运行仿真），并从下图所示弹出菜单中选择相应的选项，

提示：可用的弹出菜单选项取决于设计开发阶段。例如，如果您已运行综合，但尚未运行实现，那么弹出菜单中的实现选项将灰显。

图 20：**仿真运行选项**



运行功能仿真

综合后功能仿真

成功完成综合运行后，您可查看“Run Simulation” → “Post-Synthesis Functional Simulation”（运行仿真 > 综合后功能仿真）选项，如前图所示。

综合后，常规逻辑设计已综合到器件专用原语内。执行综合后功能仿真可确保没有任何综合最优化会对设计的功能产生影响。选择综合后功能仿真之后，即可生成功能网表，并使用 UNISIM 库执行仿真。

实现后功能仿真

完成实现运行后，前图所示“Run Simulation” → “Post-Implementation Functional Simulation”（运行仿真 > 实现后功能仿真）选项即可变为可用。

实现后，设计即已在硬件中完成布局布线。在此阶段，功能验证可用于确定实现期间是否有任何物理最优化对设计功能产生影响。

选择实现后功能仿真之后，即可生成功能网表，并使用 UNISIM 库执行仿真。

运行时序仿真

提示：综合后时序仿真使用器件模型估算的时序延迟，不包含互连延迟。实现后时序仿真使用的则是实际的时序延迟。

运行综合后和实现后时序仿真时，仿真器工具包括：

- 包含 SIMPRIMS 库组件的门级网表
- `SECUREIP`
- 标准延迟格式 (SDF) 文件

开始时，已定义设计的总体功能。实现设计时，准确的时序信息即可供使用。

为创建网表和 SDF，Vivado Design Suite 会执行如下操作：

- 调用网表编写程序 `write_verilog` 搭配 `-mode timesim` 开关和 `write_sdf`（SDF 注解器）
- 将生成的网表发送到目标仿真器

您可使用“Simulation Settings”（仿真设置）来控制这些选项，如 使用仿真设置 中所述。

重要提示！仅限 Verilog 才支持综合后和实现后时序仿真。不支持 VHDL 时序仿真。如果您是 VHDL 用户，您可运行综合后和实现后功能仿真（在此情况下无需 SDF 注解，仿真网表使用 UNISIM 库）。您可使用 write_vhdl Tcl 命令来创建网表。如需了解使用信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)。

重要提示！Vivado 仿真器模型使用互连延迟，因此，需要如下额外开关才能正确执行时序仿真：`-transport_int_delays -pulse_r 0 -pulse_int_r 0`

综合后时序仿真

成功完成综合运行后，前图所示“Run Simulation” → “Post-Synthesis Timing Simulation”（运行仿真 > 综合后时序仿真）选项即可变为可用。

完成综合后，通用逻辑设计已综合到器件专用原语中，估算的组件延迟即可供使用。您可执行综合后时序仿真来查看潜在的时序关键路径，而后再专注于实现。选择综合后时序仿真后，就会生成时序网表和 SDF 文件中估算的延迟。网表文件包含 `$sdf_annotate` 命令，因此仿真工具包含生成的 SDF 文件。

实现后时序仿真

完成实现运行后，前图所示“Run Simulation” → “Post-Implementation Timing Simulation”（运行仿真 > 实现后时序仿真）选项即可变为可用。

实现后，设计即已实现并已在硬件中完成布线。在此阶段，时序仿真有助于使用准确的时序延迟来判定设计功能是否按指定速度来工作。此仿真可用于检测未约束的路径或异步路径时序错误，例如，复位上的此类错误。选择实现后时序仿真后，就会生成时序网表和 SDF 文件。网表文件包含 `$sdf_annotate` 命令，因此即可选取生成的 SDF 文件。

指定仿真设置时，已指定是否创建 SDF 文件，以及工艺角是设置为快速还是慢速。

提示：要查找 SDF 文件的可选设置，请在 Vivado IDE Flow Navigator 中，右键单击“Simulation”并选择“Simulation Settings”。在“Settings”（设置）对话框中，选中“Simulation”类别，然后单击“Netlist”（网表）选项卡。

基于指定的工艺角，SDF 文件会包含不同的 `min` 和 `max` 数值。

运行两次不同的仿真以检查建立时间违例和保持时间违例。

要运行建立时间检查，请创建 SDF 文件并将 -process_corner 设为 slow，并使用来自该 SDF 文件的 max 列。

要运行保持时间检查，请创建 SDF 文件并将 -process_corner 设为 fast，并使用来自该 SDF 文件的 min 列。指定要使用的 SDF 延迟字段的方法取决于使用的仿真工具。请参阅特定仿真工具文档以获取有关如何设置该选项的信息。

要获取完整信息，请运行全部 4 项时序仿真，并指定如下设置：

- 慢速角：SDFMIN 和 SDFMAX
- 快速角：SDFMIN 和 SDFMAX

保存仿真结果

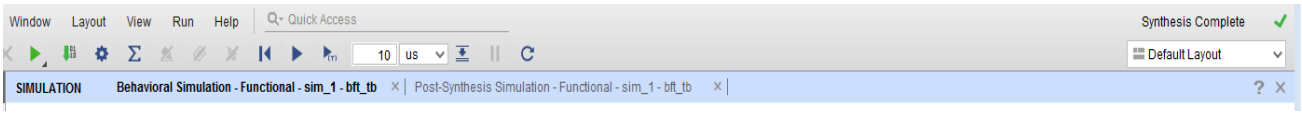
Vivado 仿真器会将追踪的对象（VHDL 信号、Verilog reg 或 wire）的仿真结果保存到 `<project>.sim/<simset>` 目录的波形数据库 (WDB) 文件 (`<filename>.wdb`) 中。

如果您将这些对象添加到“Wave”（波形）窗口中并运行仿真，那么完整设计的设计层级以及已添加的对象的转换操作都会自动保存到 WDB 文件中。您也可以使用 `log_wave` 命令将“Wave”窗口中不显示的对象添加到波形数据库中。如需了解有关 `log_wave` 命令的信息，请参阅 使用 log_wave Tcl 命令。

区分多轮仿真运行

如果针对某个设计运行多轮仿真，那么 Vivado 仿真器会在工作空间顶部显示命名的选项卡，并高亮窗口内当前的仿真类型，如下图所示。

图 21：**处于活动状态的仿真类型**



关闭仿真

要关闭仿真，请在 Vivado IDE 中执行以下操作：

选中“File > Exit”（文件 > 退出），或者单击工程窗口右上角的“X”。

注意！如有多个仿真在运行，请单击蓝色标题栏上的“X”关闭所有仿真。要关闭单个仿真，请单击蓝色标题栏下的小型灰色或白色选项卡上的“X”。

要从 Tcl 控制台关闭仿真，请输入：

```
close_sim
```

此 Tcl 命令会首先检查是否存在未保存的波形配置。如有，则此命令会发出错误。请关闭或保存未保存的波形配置，然后再发出 `close_sim` 命令或者向 Tcl 命令添加 `-force` 选项。

注释：我们始终建议先使用 `close_sim` 命令来完全关闭仿真，然后再使用 `close_project` 命令关闭当前工程。

添加仿真启动脚本文件

您可将定制 Tcl 命令以批处理文件方式添加到工程内，以便随仿真运行这些命令。仿真开始后就会运行这些命令。下列步骤描述了该过程的示例。

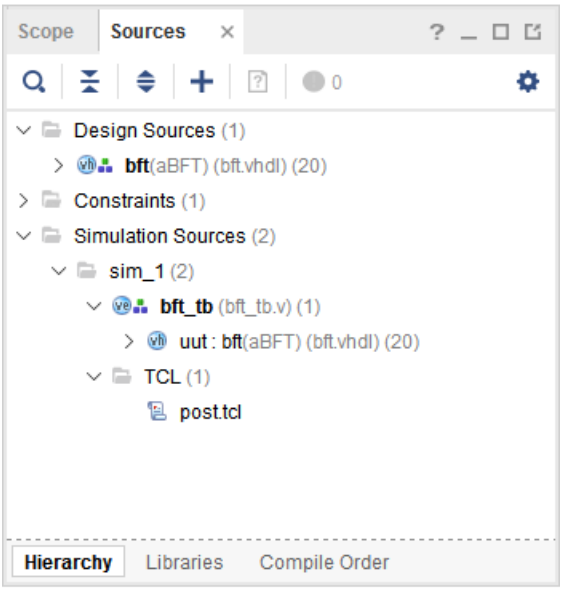
1. 创建 Tcl 脚本，其中包含您要添加到仿真源文件内的仿真命令。例如，如果您的仿真运行 1,000 ns，您希望延长其运行时间，请创建包含如下内容的文件：

   ```
   run 5us
   ```

   或者，如果您要监控非顶层信号（因为默认情况下，仅将顶层信号添加到波形中），您可将这些信号添加到 `post.tcl` 脚本中。例如：

   ```
   add_wave/top/I1/<signalName>
   ```

2. 将此文件命名为 `post.tcl` 并保存。
3. 使用 Flow Navigator 中的“Add Sources”（添加源文件）选项来调用“Add Sources”Wizard（添加源文件向导），并选择“Add or Create Simulation Sources”（添加或创建仿真源文件）。
4. 将 `post.tcl` 文件作为仿真源文件添加到 Vivado Design Suite 工程中。`post.tcl` 文件会显示在“Simulation Sources”文件夹下，如下图所示。



5. 在“Simulation”（仿真）工具栏中，单击“Relaunch”（重新启动）按钮 。

   这样就会再次运行仿真，并在原始指定时间基础上额外增加您在 `post.tcl` 文件中指定的时间。Vivado 会在调用 `post.tcl` 文件的所有命令后，自动使用 source 命令运行该文件。

查看仿真消息

Vivado IDE 包含消息区域，您可在其中查看参考消息、警告消息和错误消息。如下图所示，来自 Vivado 仿真器的部分消息包含问题描述和建议的解决方案。

要在 Tcl 控制台中查看相同的详细信息，请输入：

```
help -message {message_number}
```

图 22：**仿真器消息描述和解决方案信息**

以下提供了此类命令的示例：

```
help -message {simulator 43-3120}
```

管理消息输出

如果您的 HDL 设计生成大量消息（例如，通过 `$display` Verilog 系统任务或 `report` VHDL 语句），那么您可限制发送给 Tcl 控制台和 log 日志文件的文本输出量。这样可以节省计算机存储器和磁盘空间。要完成此操作，请使用 `-maxlogsize` 命令行选项：

1. 在 Flow Navigator 中，右键单击“SIMULATION”（仿真）并选中“Simulation Settings”（仿真设置）。
2. 在“Settings”（设置）对话框中，在 `xsim.simulate.xsim.more_options` 旁添加 `-maxlogsize <size>`，其中 `<size>` 是最大文本输出量（以兆字节 (MB) 为单位）。

使用 launch_simulation 命令

`launch_simulation` 命令允许您以脚本模式运行任意受支持的仿真器。

launch_simulation 语法如下所示：

```
launch_simulation [-step <arg>] [-simset <arg>] [-mode <arg>] [-type <arg>]
[-scripts_only] [-gui] [-exec] [-of_objects <args>] [-absolute_path][-
install_path <arg>] [-noclean_dir] [-quiet] [-verbose][-gcc_install_path
<arg>][-exec]
```

下表描述了 launch_simulation 的选项。

表 11：**launch_simulation 选项**

选项	描述
[-step]	启动仿真步骤。值：all、compile、elaborate 和 simulate。默认值：all（启动所有步骤）。
[-simset]	仿真文件集的名称。
[-mode]	仿真模式。值：behavioral、post-synthesis 和 post-implementation；默认值：behavioral。
[-type]	网表类型。值：functional 和 timing。仅当模式设置为“post-synthesis”（综合后）或“post-implementation”（实现后）时，此项才适用。
[-scripts_only]	仅生成脚本。
[-gui]	调用仿真器 GUI（适用于 -scripts_only 模式）。
[-exec]	为以 -step 开关指定的步骤执行现有脚本。
[-of_objects]	为此对象生成编译顺序文件（仅适用于 -scripts_only 选项）。
[-absolute_path]	使所有文件路径都变为相对于引用目录的绝对路径。
[-install_path]	自定义安装目录路径。
[-noclean_dir]	不得移除仿真运行目录文件。
[-quiet]	忽略命令错误。
[-verbose]	在命令执行期间暂挂消息限制。
[-gcc_install_path]	指定表示对应 g++/gcc 可执行文件的 GNU 编译器安装目录路径。
[-exec]	为以 -step 开关指定的步骤执行现有脚本。

示例

- 使用 Vivado 仿真器运行行为仿真。

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
launch_simulation
```

- 使用 Questa Advanced Simulator 生成脚本用于行为仿真。

```
create_project project_1 project_1 -part xc7vx485tffg1157-1
add_files -norecurse tmp.v
add_files -fileset sim_1 -norecurse testbench.v
import_files -force -norecurse
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
```

```
set_property target_simulator Questa [current_project]
set_property compxlib.questa_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -scripts_only
```

- 使用 Synopsys VCS 启动综合后功能仿真。

```
set_property target_simulator VCS [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location>
[current_project]
launch_simulation -mode post-synthesis -type functional
```

- 使用 Synopsys VCS 运行实现后时序仿真。

```
set_property target_simulator vcs [current_project]
set_property compxlib.vcs_compiled_library_dir
<compiled_library_location> [current_project]
launch_simulation -mode post-implementation -type timing
```

注释： `launch_simulation` 命令是 Vivado IDE 中原生集成的命令，可用于在 GUI 或 Tcl 模式下运行仿真或生成脚本。您还可使用 `export_simulation` 命令来仅生成脚本。如需了解更多详情，请参阅 导出仿真文件和脚本。

在非预编译模式下运行设计

默认情况下，`launch_simulation` 命令会在细化期间编译设计文件并绑定预编译的 IP 模型或仿真模型。这些预编译的 IP 模型或仿真模型既可以随设计一起编译，也可以在本地运行目录中编译，对于后者，将在 .do/prj/sh 文件中添加这些 IP 模型或仿真模型的相关源文件用于编译。要在非预编译模式下运行设计，请取消选中“Project Settings” → “IP” → “Precompiled IP simulation libraries”（工程设置 > IP > 预编译的 IP 仿真库）复选框，或者在 Tcl 控制台中将以下属性设为 false。

```
set_property sim.use_ip_compiled_libs false [current_project]
```

注释： 在非预编译模式下运行设计时，不会使用先前使用 `compile_simlib` 编译的 IP 模型或仿真模型的预编译版本。

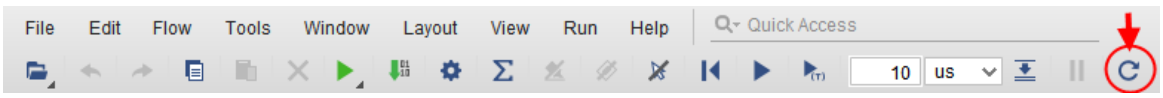
设计更改（重新启动）后重新运行仿真

使用 Vivado 仿真器调试 HDL 设计时，您可能会发现自己的 HDL 设计需要纠错。

您可使用以下步骤来修改自己的设计并重新运行仿真：

1. 使用 Vivado 代码编辑器或其他文本编辑器来更新并保存任何必要的源代码更改。
2. 使用 Vivado IDE 工具栏上的“Relaunch Simulation”（重新启动仿真）按钮来重新编译并重新启动仿真，如下图所示。您也可以选择使用 `relaunch_sim` Tcl 命令来重新编译和重新启动仿真。

图 23： **“Relaunch Simulation”按钮**



3. 如果修改后的设计编译失败，则会出现一个错误框，其中会显示失败原因。Vivado IDE 会继续在禁用状态下显示先前运行的仿真结果。返回步骤 1 纠正错误，然后重新运行仿真。

设计成功完成重新编译后，就会再次启动仿真。

重要提示！除了编译错误外，重新启动也可能因其他原因而失败，例如，文件系统错误。如果“Simulation”（仿真）工具栏上的“Run”（运行）按钮在重新启动之后变为灰显，表示仿真处于已禁用状态，请检查 Tcl 控制台的内容，查找可能导致重新启动无法成功完成的错误。

注意！您也可以使用 Flow Navigator 中的“Run Simulation”（运行仿真）或者使用 `launch_simulation` Tcl 命令来重新启动仿真。但使用这些选项可能完全关闭仿真，丢弃波形更改和仿真设置（例如，基数自定义）。

注释：仅当使用 `launch_simulation` 成功运行一次 Vivado 仿真器后，“Relaunch Simulation”按钮  才会变为活动。如果在批处理/脚本模式下运行仿真，“Relaunch Simulation”按钮将为灰显。

使用保存的仿真器用户界面设置

默认情况下，在您使用 Vivado 仿真器的用户界面控制和 Tcl 命令时，Vivado 仿真器会将配置更改保存到仿真的工作目录下的文件中。保存的设置包括：

- “Scopes”（作用域）窗口和“Objects”（对象）窗口的筛选按钮和列宽状态。
- 仿真的 Tcl 属性，包括阵列显示限制、默认基数、运行命令的默认时间单位和追踪限制。
- 您在“Objects”窗口中的 HDL 对象上设置的基数和“Show as Enumeration”（显示为枚举）状态。

关闭仿真后，在您重新打开和运行 Vivado 仿真器时，Vivado 仿真器会复原您的设置。

重要提示！关闭 Vivado 的“Simulation Settings”（仿真设置）中的“Clean Up Simulation Files”（清除仿真文件）复选框，以免重新启动仿真时擦除设置文件。

提示：要还原至默认设置，请删除设置文件。您可在位于以下位置的 Vivado 工程目录中找到设置文件：`<project>.sim/<simset>/<simtype>/xsim.dir/<snapshot>/xsimSettings.ini`。例如，BFT 设计示例的默认行为仿真运行的设置文件应驻留在 `bft.sim/sim_1/behav/xsim.dir/bft_tb_behav/xsimSettings.ini`。或者，在“Simulation Settings”中开启“Clean Up Simulation Files”复选框。

默认设置

AMD Vivado™ 工程 Tcl 对象支持多个属性，以便您提供默认设置用于已清除的仿真或新创建的仿真。这些仿真尚不含设置文件。以下列表显示了工程的默认设置属性：

- XSIM.ARRAY_DISPLAY_LIMIT
- XSIM.RADIX
- XSIM.TIME_UNIT
- XSIM.TRACE_LIMIT

您可使用 `report_property [current_project]` Tcl 命令查看这些属性的当前值，并使用 `set_property <property name> <property value> [current_project]` Tcl 命令设置这些属性值。例如，要将阵列显示限制设为 16，请使用以下命令。

```
set_property xsim.array_display_limit 16 [current_project]
```

启动新仿真或已清除的仿真时，仿真 Tcl 对象会继承您的工程属性。您可以使用以下 Tcl 命令验证该对象：

```
report_property [current_sim]
```

重要提示！工程属性仅适用于已清除的仿真或新创建的仿真。针对特定运行类型和仿真设置（例如，`sim_1/behav`）的仿真，完成运行后，该仿真会保留一份单独的设置副本，以供所有后续启动使用。对工程属性执行的更改对于该仿真不再生效。仅当该仿真已清除或者删除设置文件后，工程设置才会再次生效。

第 5 章

使用 Vivado 仿真器对仿真波形进行分析

在 AMD Vivado™ 仿真器中，您可以使用波形来分析设计和调试代码。仿真器会在工作空间的其他区域内填充设计信号数据，例如在“Objects”（对象）窗口和“Scope”（作用域）窗口内。

通常，在测试激励文件中设置仿真的位置就是您定义要仿真的 HDL 对象的位置。如需了解有关测试激励文件的更多信息，请参阅《编写高效的测试激励文件》(XAPP199)。

启动 Vivado 仿真器时，会显示波形配置和顶层 HDL 对象。Vivado 仿真器会在工作空间的其他区域内填充设计数据，例如在“Scope”窗口和“Objects”窗口内。随后，您可添加其他 HDL 对象或者运行仿真。请参阅以下 使用波形配置和窗口。

使用波形配置和窗口

Vivado 仿真器允许自定义波形显示。当前显示状态称为“波形配置”。此配置可保存以供将来在 WCFG 文件中使用。

波形配置可取名或者可设为 `untitled`。名称显示在波形配置窗口的标题栏上。如果波形配置从未保存到文件，则设为“untitled”（无标题）。

创建新的波形配置

创建新的波形配置用于显示波形，如下所示：

1. 选中“File” → “Simulation Waveform” → “New Configuration”（文件 > 仿真波形 > 新建配置）。

   这样会打开新的“Wave”（波形）窗口，并显示新的无标题波形配置。Tcl 命令： `create_wave_config <waveform_name>`。

2. 使用 认识波形配置中的 HDL 对象 中所列示的步骤将 HDL 对象添加到波形配置中。

如需了解有关创建新的波形配置的更多信息，请参阅 第 4 章：使用 Vivado 仿真器进行仿真。另请参阅 创建和使用多个波形配置，获取有关多个波形的信息。

打开 WCFG 文件

打开 WCFG 文件以搭配仿真使用，如下所示：

1. 选中“File > Simulation Waveform > Open Configuration”（文件 > 仿真波形 > 打开配置）。

   这样会打开“Open Waveform Configuration”（打开波形配置）对话框。

2. 找到并选择 WCFG 文件。

   注释：打开 WCFG 文件时，如果其中包含对 HDL 对象的引用，但这些对象在静态仿真 HDL 设计层级内不存在，那么 Vivado 仿真器会忽略这些 HDL 对象，并在加载的波形配置中将其省略。

这样会打开“Wave”窗口，并显示仿真器为列示的 WCFG 文件的波形对象查找的波形数据。

Tcl 命令：`open_wave_config <waveform_name>`

保存波形配置

完成编辑后，要将波形配置保存到 WCFG 文件，请选中“File” → “Simulation Waveform” → “Save Configuration As”（文件 > 仿真波形 > 将配置另存为），并为波形配置输入名称。

Tcl 命令：`save_wave_config <waveform_name>`

打开先前保存的仿真运行

有三种方法可使用 Vivado Design Suite 打开先前保存的仿真：交互式方法和编程式方法。

独立模式

您可在 Vivado 外使用以下命令打开 WDB 文件：

```
xsim <name>.wdb -gui
```

提示：您可将 `-view <WCFG file>` 添加到 `xsim` 命令来同时打开 WCFG 文件和 WDB 文件。

交互式方法

- 如果加载 Vivado Design Suite 工程，请单击“Flow” → “Open Static Simulation”（流程 > 打开静态仿真）并选择包含来自先前运行仿真的波形的 WDB 文件。

提示：静态仿真是一种 Vivado 仿真器模式，在此模式下，仿真器会在其窗口中显示来自 WDB 文件的数据，以替代原先显示的来自运行中的仿真的数据。

- 或者，在 Tcl 控制台中运行：`open_wave_database <name>.wdb`。

编程方法

创建包含如下内容的 Tcl 文件（例如，`design.tcl`）：

```
current_fileset
open_wave_database <name>.wdb
```

然后按如下方式运行：

```
vivado -source design.tcl
```

重要提示！Vivado 仿真器可打开在任意受支持的操作系统上创建的 WDB 文件。它还能打开在 Vivado Design Suite 2014.3 版本和更高版本中创建的 WDB 文件。Vivado 仿真器无法打开在低于 2014.3 版本的 Vivado Design Suite 中创建的 WDB 文件。

运行仿真并在“Wave”窗口中显示 HDL 对象时，运行中的仿真会生成波形数据库 (WDB) 文件，其中包含所显示的 HDL 对象的波形活动。此 WDB 文件还会存储有关仿真设计中所有 HDL 作用域和对象的信息。在此模式下，您无法使用监控仿真的命令（例如，run 命令），因为没有底层的“活动”仿真模型可供控制。

但您可以查看静态仿真内的波形和 HDL 设计层级。

认识波形配置中的 HDL 对象

向波形配置添加 HDL 对象时，波形查看器会创建 HDL 对象的波形对象 (wave object)。波形对象链接到关联的 HDL 对象，但两者有别。

您可从同一个 HDL 对象创建多个波形对象，并单独设置每个波形对象的显示属性。

例如，您可以为名为 `myBus` 的 HDL 对象设置一个显示十六进制值的波形对象，并为 `myBus` 设置另一个显示十进制值的波形对象。

还有其他种类的波形对象可供显示在波形配置中，例如，分频器、分组和虚拟总线。

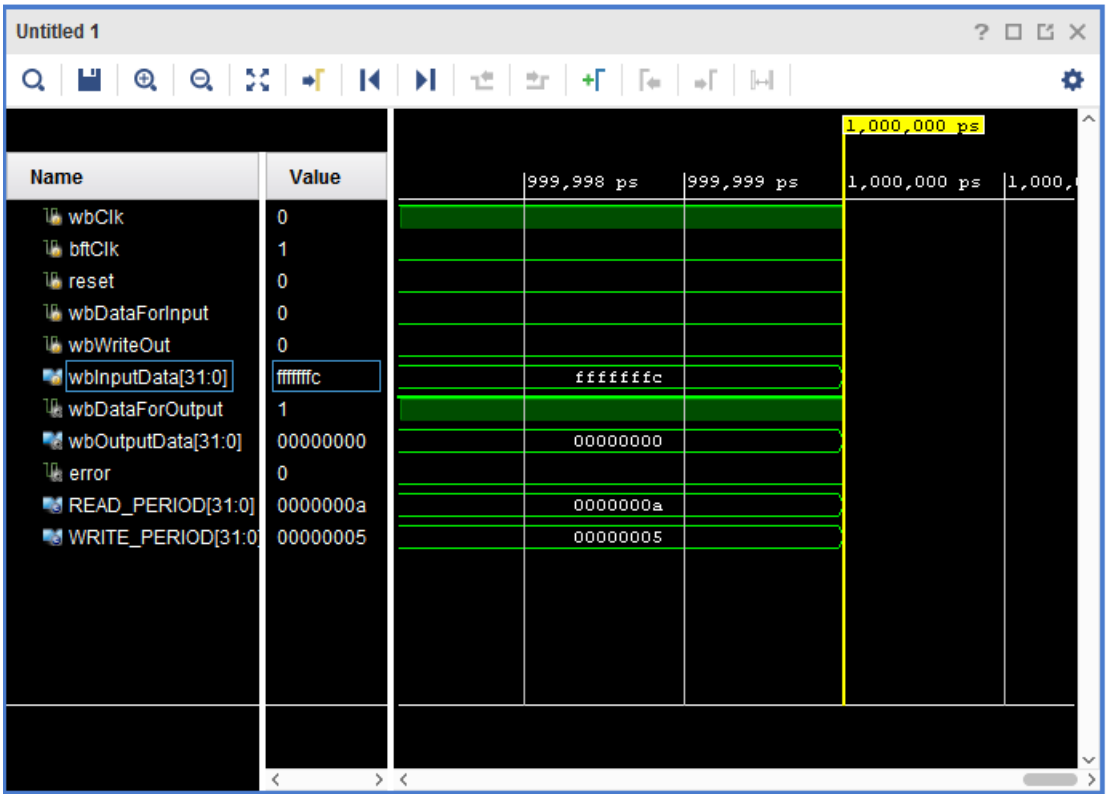
从 HDL 对象创建的波形对象称为设计波形对象。这些对象均以对应图标来显示。对于设计波形对象，此图标用于指示该对象属于标量  或复合对象  ，例如，Verilog 矢量或 VHDL 记录。

提示：要在“Objects”窗口中查看设计波形对象的 HDL 对象，请右键单击设计波形对象的名称，然后选择“Show in Object Window”（在对象窗口中显示）。

下图显示波形配置窗口中 HDL 对象示例。设计对象会显示“Name”（名称）和“Value”（值）。

- “Name”：默认情况下，显示 HDL 对象的短名称：仅显示名称本身，不含对象的分层路径。您可更改“Name”以显示含完整分层路径的长名称或者为其分配自定义名称。
- “Value”：显示位于波形窗口的主光标所指示的时间处的对象的值。您可单独更改该值的格式或基数，此更改不影响链接到相同 HDL 对象的其他设计波形对象的格式以及“Objects”窗口和源代码窗口中显示的值的格式。

图 24：**波形 HDL 对象**



“Scope”（作用域）窗口提供了将所选作用域的所有可查看 HDL 对象都添加到波形窗口的能力。如需了解有关使用“Scope”窗口的信息，请参阅 “Scope”窗口。

关于基数

了解总线上的数据类型至关重要，而为了有效使用数字和模拟波形选项，您需要识别基数设置与数据类型之间的关系。

重要提示！要更改基数设置，请在您要看到此更改的窗口中执行更改。在“Objects”（对象）窗口中对任一项的基数进行的更改并不会应用于“Wave”（波形）窗口或 Tcl 控制台中的值。例如，项 wbOutputData[31:0] 在“Objects”窗口中可设为“Signed Decimal”（有符号十进制），但在“Wave”窗口中它仍设为“Binary”（二进制）。

更改默认基数

对于您未显式设置其基数的波形对象，默认波形基数用于控制所有这些对象的值的数字格式。波形基数默认为“Hexadecimal”（十六进制）。

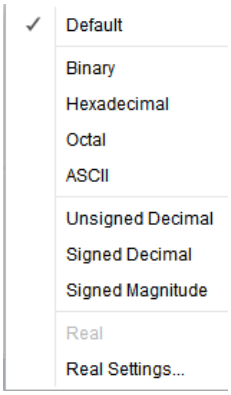
要更改默认波形基数，请执行以下操作：

1. 在“Waveform”（波形）窗口中，单击“Settings”（设置）按钮 打开“Waveform Settings”（波形设置）。
2. 在“General”页面上，单击“Default Radix”（默认基数）下拉菜单。
3. 在下拉列表中，选择基数。

更改个别对象的基数

要在“Wave”（波形）窗口中更改波形对象的基数，请执行以下操作：

1. 右键单击波形对象名称。
2. 选中“Radix”（基数），然后从下拉菜单中选择所需格式：
   - 默认
   - 二进制
   - 十六进制
   - 八进制
   - ASCII
   - 无符号十进制
   - 有符号十进制
   - 有符号量级
   - 实数



注释：如需获取有关实数和实数设置的用法描述，请参阅 使用模拟波形。

3. 在 Tcl 控制台中，要更改显示的值的数字格式，请输入以下 Tcl 命令：

```
set_property radix <radix> <hdl_object>
```

其中，`<radix>` 是以下值之一：`bin`、`unsigned`、`hex`、`dec`、`ascii` 或 `oct`，而 `<wave_object>` 则是 `add_wave` 命令返回的对象。

提示：如果您在“Wave”窗口中更改基数，此更改将不会反映在“Objects”（对象）窗口中。

自定义波形

使用模拟波形

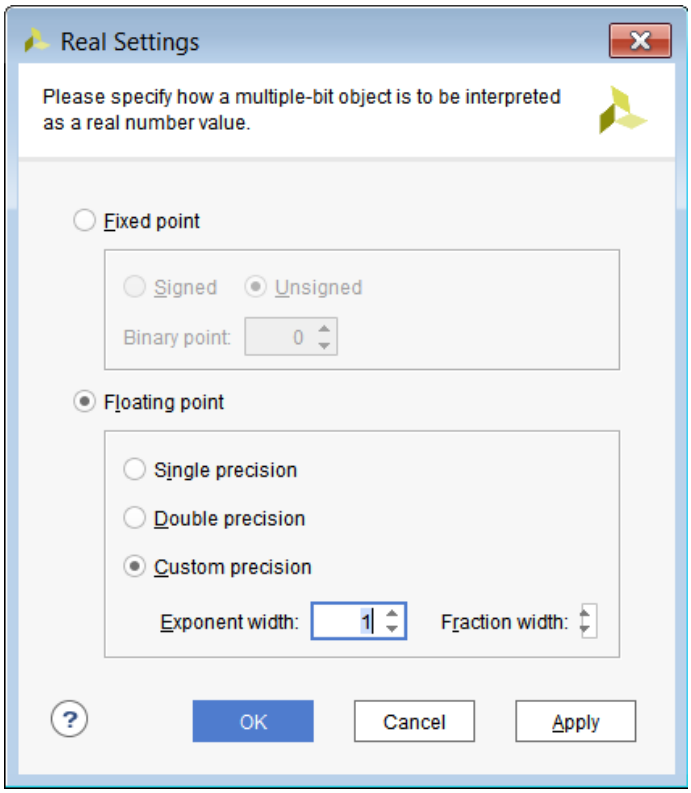
使用基数和模拟波形

总线值解读为数值，由总线波形对象上的基数设置来确定，如下所示：

- 使用二进制、八进制、十六进制、ASCII 和无符号十进制基数会导致将总线值解读为无符号整数。
- 如果总线中的任意位既非 0 也非 1，那么整个总线值会被解读为 0。
- 有符号十进制和有符号量级基数则会导致将总线值解读为有符号整数。
- 基于“Real Settings”（实数设置）对话框的设置，实数基数会导致将总线值解读为定点或浮点实数。

要将波形对象设置为实数基数，请执行以下操作：

1. 在波形配置窗口中，选择 HDL 对象，然后右键单击以打开弹出菜单。
2. 选择“Radix”→“Real Settings”（基数 > 实数设置）以打开“Real Settings”对话框，如下图所示。



您可将波形基数设置为“Real”（实数）以将对象的值显示为实数。选择此基数前，必须选择相应设置来指示波形查看器如何解读这些值的各个位。

“Real Setting”对话框选项如下：

- “Fixed Point”（定点）：指定将所选波形的波形对象的各个位解读为定点、有符号或无符号实数。

- “Binary Point”（二进制点）：指定解读为二进制点右侧的位数。如果“Binary Point”大于波形对象的位宽，那么波形对象值无法解读为定点，当波形对象以数字波形样式显示时，所有值都显示为 <Bad Radix>。显示为模拟时，所有值都解读为 0（零）。
- “Floating Point”（浮点）：指定所选总线波形对象的位应解读为 IEEE 浮点实数。

  注释：仅支持单精度和双精度（以及值设置为单精度和双精度的定制精度）。

  其他值则会导致出现 `<Bad Radix>` 值，与“Fixed Point”相同。指数宽度和小数宽度相加必须等于波形对象的位宽，否则会得到 `<Bad Radix>` 值。

提示：如果行索引分隔线不可见，可在 使用“Waveform Settings”对话框 中将其开启，使其可见。

将波形显示为模拟波形

重要提示！以模拟波形形式查看 HDL 总线对象时，要生成期望的波形，请选择与 HDL 对象中的数据性质相匹配的基数。例如：

- 如果总线上编码的数据是 2 的补码有符号整数，则必须选择有符号基数。
- 如果数据是采用 IEEE 格式编码的浮点，则必须选择实数基数。

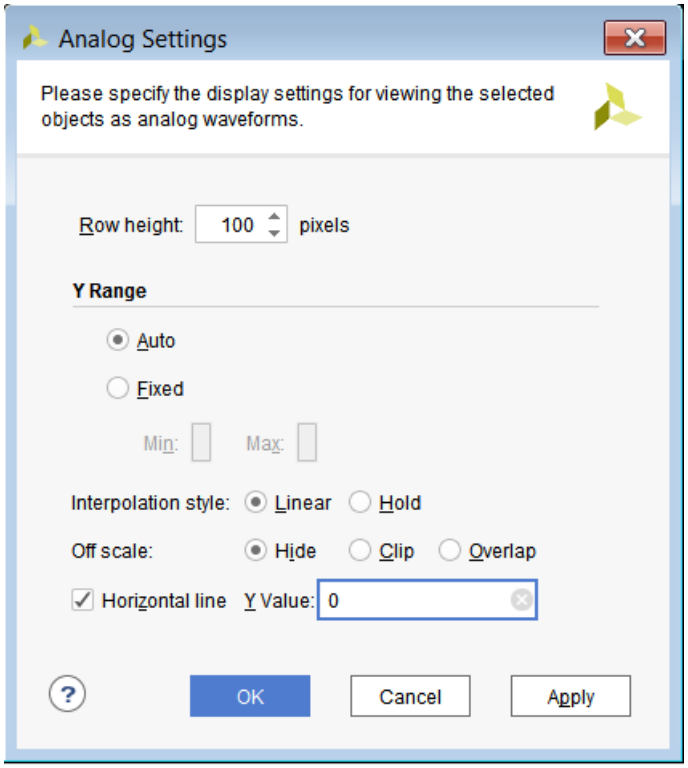
自定义模拟波形外观

要自定义显示的模拟波形，请右键单击波形配置窗口中的“Name”（名称）列中的 HDL 对象，并从下拉菜单中选择“Waveform Style”（波形样式）。这样会显示弹出菜单，其中会显示下列选项：

- “Analog”（模拟）：将波形设置为“Analog”。
- “Digital”（数字）：将波形对象设置为“Digital”。
- “Analog Settings”（模拟设置）：打开“Analog Settings”对话框（如下图所示），其中提供了各种模拟波形显示选项。

“Wave”窗口只能为位宽不超过 64 位的总线显示模拟波形。

图 25：**“Analog Settings”对话框**



“Analog Settings”对话框选项描述

- “Row Height”（行高）：指定所选波形对象的高度（以像素为单位）。更改行高不会影响垂直方向显示或隐藏的波形，而只是对波形高度进行伸缩。

  在“模拟波形”与“数字波形”之间进行切换时，行高会设置为对应波形的相应默认值（针对数字波形为 20，针对模拟波形为 100）。

  > **提示：**如果行索引分隔线不可见，请在“Waveform Settings”中启用对应复选框以将其开启。如需了解有关如何更改选项设置的信息，请参阅 使用“Waveform Settings”对话框。您也可通过将行索引分隔线拖至波形名称左侧或下方来更改行高。

- “Y Range”（Y 范围）：指定波形区域内显示的数字值范围。
  - “Auto”（自动）：指定只要发现窗口内可见时间范围超出当前范围，此范围就持续扩展。
  - “Fixed”（固定）：指定时间范围保持固定间隔不变。
  - “Min”（最小值）：指定波形区域底部显示的值。
  - “Max”（最大值）：指定波形区域顶部显示的值。

  这两个值均可指定为浮点值，但如果波形对象基数为整数，则这些值将被截位至整数。

- “Interpolation Style”（内插样式）：指定用于连接数据点的线的绘制方式。
  - “Linear”（线性）：指定 2 个数据点之间为直线。
  - “Hold”（保持）：指定在 2 个数据点之间，从左侧点绘制 1 条水平线到右侧点的 X 坐标，然后绘制另一条线以将前一条线连接到右侧数据点，构成 L 形。

- “Off Scale”（超标度）：指定超出波形区域的 Y 范围的波形值的绘制方式。
  - “Hide”（隐藏）：指定不显示超出范围的值，例如，达到波形区域上限或下限的波形将消失，直至值重新恢复到范围内为止。
  - “Clip”（剪切）：指定更改超出范围的值，使其位于波形区域顶部或底部。这样达到波形区域的上限或下限的波形就会沿边界呈现为水平线，直至值重新恢复到范围内为止。
  - “Overlap”（重叠）：指定在波形值达到波形窗口本身上限的任意位置都需绘制波形，即使其值在波形区域边界外并与其他波形重叠。
- “Horizontal Line”（水平线）：指定是否在给定值绘制水平规则。如果开启此复选框，则会在指定 Y 值的垂直位置绘制 1 条水平网格线，前提是该值在波形的 Y 范围内。

  就像“Min”和“Max”一样，Y 值接受浮点值，但如果所选波形对象的基数为整数，则会被截位为整数。

波形对象命名样式

本工具提供了相应的选项用于重命名对象、查看对象名称和更改名称显示。

重命名对象

您可在波形配置中重命名任何波形对象，例如，设计波形对象、分频器、组和虚拟总线。

1. 在“Name”（名称）列中指定对象名称。
2. 右键单击并从弹出菜单中选择“Rename”（重命名）。

   这样会打开“Rename”对话框。
3. 在“Rename”对话框中输入新名称，然后单击“OK”（确定）。

   注释：更改波形配置中设计波形对象的名称不影响底层 HDL 对象的名称。

更改对象显示名称

您可显示每个设计波形对象的完整分层名称（长名称）、简单的信号或总线名称（短名称）或者定制名称。对象名称会显示在波形配置的“Name”（名称）列中。如果此名称处于隐藏状态，请执行以下操作：

1. 展开“Name”列直至您看到完整名称。
2. 在“Name”列中，使用滚动条查看名称。

要更改显示名称，请执行以下操作：

1. 选中一个或多个信号或总线名称。使用 Shift + 单击或 Ctrl + 单击选择多个信号名称。
2. 右键单击并从下拉菜单中选择“Name”。这样会显示弹出菜单，其中会显示下列选项：
   - 选中“Long”（长名称）即可显示设计对象的完整分层名称。
   - 选中“Short”（短名称）仅显示信号或总线的名称。
   - 选中“Custom”（定制名称）即可在重命名对象时，显示给予该对象的定制名称。请参阅 更改对象显示名称。

提示：重命名波形对象会将名称显示模式更改为“Custom”。要复原原始名称显示模式，请将显示模式更改为“Long”或“Short”，如上所述。长名称或短名称仅对设计波形对象有意义。其他波形对象（分频器、分组和虚拟总线）默认显示其“Custom”名称，对于其“Long”名称和“Short”名称显示 ID 字符串。

反转总线位顺序

您可在波形配置中反转总线位顺序，这样所显示的总线值即可在下列两种位顺序之间进行切换：MSB 优先（大字节序）或 LSB 优先（小字节序）。

要反转位顺序，请执行以下操作：

1. 选择总线。
2. 右键单击并选中“Reverse Bit Order”（反转位顺序）。

   这样即可反转总线位顺序。“Reverse Bit Order”命令会标记为显示这是当前行为。

重要提示！ “Reverse Bit Order”命令仅对总线上显示的值进行操作。展开总线波形对象时，该命令不会反转总线下出现的总线元素列表。

提示：总线的长名称和短名称上显示的索引范围表示总线元素的位顺序。例如，对总线 `bus[0:7]` 应用“Reverse Bit Order”后，总线会显示 `bus[7:0]`。

更改 SystemVerilog 枚举的格式

SystemVerilog 枚举是具有数字值的 HDL 对象，通过为其定义文本标签来表示特定的值。例如，枚举可定义 LABEL1 以表示值 1，定义 LABEL2 来表示值 5。上下文菜单中的“Show As Enumeration”（显示为枚举）选项允许您指定是使用给定标签还是以数字方式来显示枚举值。在前述示例中，如果开启“Show As Enumeration”，那么值 5 显示为 `LABEL2`。如果关闭该选项，那么值 5 的显示方式与为枚举设置的任意基数中显示的方式相同，如“Radix”（基数）菜单中所示。

要使用标签显示枚举，请执行以下操作：

1. 选中枚举
2. 右键单击并勾选“Display As Enumeration”（显示为枚举）

要以数字方式显示枚举，请执行以下操作：

1. 选中枚举
2. 右键单击并取消勾选“Display As Enumeration”

   注释：未定义标签的枚举值始终以数字方式显示，与“Display As Enumeration”设置无关。仅限 SystemVerilog 枚举对象才能启用“Display As Enumeration”选项。

控制波形显示

您可使用以下方法控制波形显示：

- 调整“Wave”（波形）窗口的“Name”（名称）、“Value”（值）和“Waveform”（波形）列之间句柄的大小
- 结合使用鼠标滚轮来执行滚动操作
- “Wave”窗口侧边栏中的缩放功能按钮
- 结合使用鼠标滚轮来执行缩放操作
- Vivado IDE Y 轴缩放手势

- Vivado 仿真器 X 轴缩放手势。如需了解有关使用鼠标进行平移和缩放的更多信息，请参阅《Vivado Design Suite 用户指南：使用 Vivado IDE》(UG893)。

  注释：与其他 Vivado Design Suite 图形窗口相反，在“Wave”窗口中放大适用于 X（时间）轴，与 Y 轴无关。因此，用于指定窗口缩放的时间范围的“Zoom Range X”（缩放范围 X 轴）手势取代了其他 Vivado Design Suite 窗口的“Zoom to Area”（缩放到区域）手势。

提示：除了波形对象和标记外，还会额外保存 WCFG 文件记录波形窗口设置。波形窗口设置包括名称和值列宽度、缩放级别、滚动位置、分组和总线的扩展状态以及主光标的位置。

使用列大小调整句柄

要更改“Name”（名称）列或“Value”（值）列的宽度，请将鼠标置于列右侧的垂直条上，直至光标改变形状，然后将鼠标向左或向右拖动以便按需缩小或扩大列宽度。

注释：如果“Value”列宽度已达其最小值，那么您可能需要首先拓宽“Value”列，才能拓宽“Name”列。

使用鼠标滚轮滚动

单击波形窗口内部即可使用鼠标滚轮进行向上和向下滚动。您也可以搭配 Shift 键使用鼠标滚轮向左和向右滚动波形。

使用缩放功能特性按钮

在“Wave”窗口中包含各种缩放功能，例如，“Zoom in”（放大）、“Zoom Out”（缩小）和“Zoom Fit”（缩放适应），这些功能支持您按需缩放波形配置。

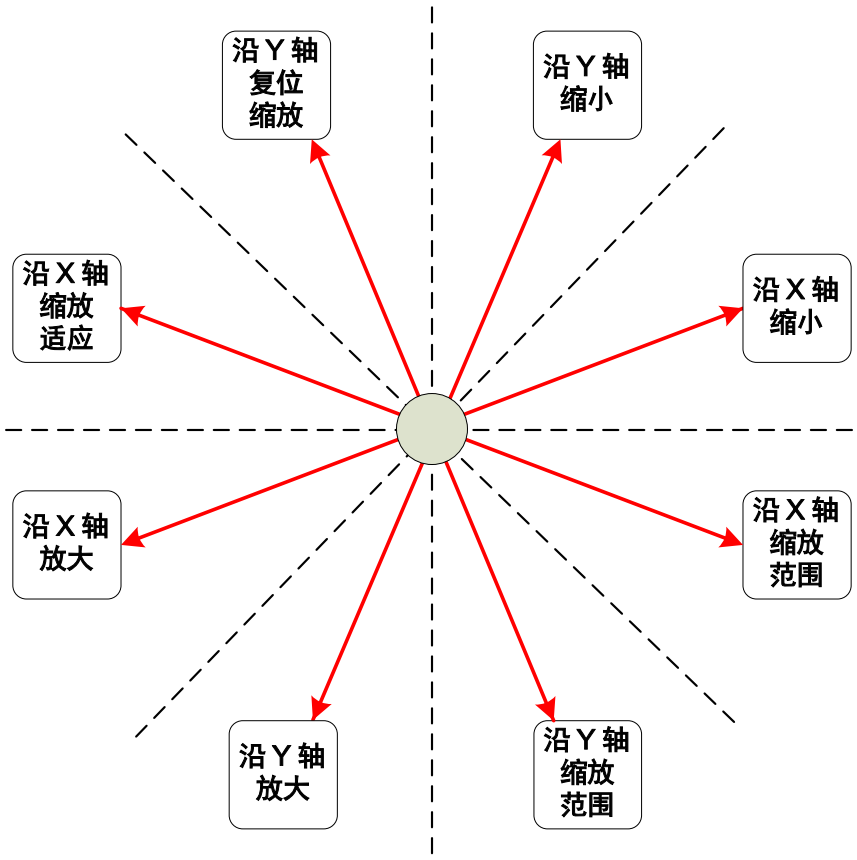
使用鼠标滚轮缩放

单击波形区域内部并使用鼠标滚轮与 Ctrl 键搭配进行缩放，模拟示波器上的旋钮操作。

模拟波形的 Y 轴缩放手势

除了支持在 X 维进行缩放的缩放手势外，在模拟波形上还可使用额外的缩放手势，如下图所示。

图 26：**模拟缩放选项**



要调用缩放手势，请按住鼠标左键并按图中所示方向拖动，其中起始鼠标位置即图中的中心点。

其他缩放手势如下所示：

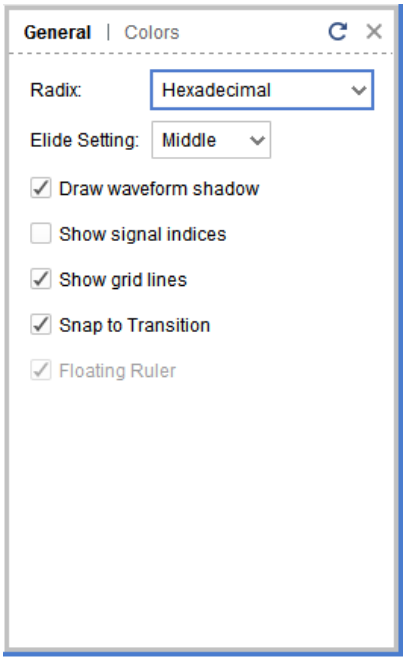
- Y 轴缩小：沿 Y 维度按 2 次幂缩小，幅度由释放鼠标按键的位置与起点的距离决定。执行缩放时，起始鼠标位置的 Y 值保持静止。
- Y 轴缩放范围：绘制垂直幕布，以指定松开鼠标时要显示的 Y 范围。
- Y 轴放大：沿 Y 维度按 2 次幂放大，幅度由释放鼠标按键的位置与起点的距离决定。执行缩放时，起始鼠标位置的 Y 值保持静止。
- Y 轴缩放复位：将 Y 范围复位至波形窗口中当前显示的值对应的 Y 范围，并将 Y 范围模式设置为“Auto”（自动）。

Y 维度的所有缩放手势都可设置 Y 范围模拟设置。“Reset Zoom Y”可将“Y Range”设置为“Auto”，其他手势可将“Y Range”设置为“Fixed”（固定）。

使用“Waveform Settings”对话框

单击“Settings”（设置）按钮 即可打开“Waveform Settings”（波形设置），如下图所示。

图 27：**Waveform Settings**



在“General”（常规）选项卡上，您可配置下列“Waveform Settings”：

- “Radix”（基数）：设置用于新创建的设计波形对象的数字格式。
- “Elide Setting”（省略设置）：控制信号名称的截断，此类名称对于“Wave”（波形）窗口而言可能过长。
  - “Left”（左）表示截断长名称的左端。
  - “Right”（右）表示截断长名称的右端。
  - “Middle”（中间）表示保留左端和右端，省略长名称的中间部分。
- “Draw Waveform Shadow”（绘制波形阴影）：创建波形的阴影表示法。
- “Show signal indices”（显示信号索引）：在每个波形对象名称左侧显示行号。您可拖动各条线，将行号分割以更改波形对象的高度。
- “Show grid lines”（显示网格线）：显示含网格线的波形窗口。
- “Snap to Transition”（对齐到转换位置）：选中此项时，会导致拖动的光标和标记沉降至靠近鼠标光标的波形转换位置。如需了解更多信息，请参阅 使用光标。
- “Floating Ruler”（浮动标尺）：只要有辅助光标可见或者选中标记，即显示浮动标尺。如需了解更多信息，请参阅 使用浮动标尺。

> **提示：**如果在“Settings”对话框中“Floating Ruler”选项显示为禁用（未勾选）状态，请使用“Wave”窗口上的 Shift+单击使辅助光标可见。此操作会导致在“Settings”对话框中启用“Floating Ruler”选项。

- 在“Colors”（颜色）选项卡中，您可以设置波形内各项的颜色。

更改显示的时间刻度

右键单击以上标尺以显示时间刻度菜单。此菜单允许您选择希望在时间刻度上显示的时间值。

时间刻度菜单上的选项如下：

- Auto（自动）：时间刻度自动选择适合波形窗口的缩放级别的时间单位。
- Default（默认值）：显示对应于编译 HDL 设计时所确定的仿真精度的时间单位。
- Samples（样本数）：以离散样本数形式（代替几分之一秒）显示时间（对于 HDL 仿真不可用）。
- User（用户）：用户定义的时间单位（对于 HDL 仿真不可用）。
- fs：以飞秒 (femtosecond) 为单位显示时间单位。
- ps：以皮秒 (picosecond) 为单位显示时间单位。
- ns：以纳秒 (nanosecond) 为单位显示时间单位。
- us：以微秒 (microsecond) 为单位显示时间单位。
- ms：以毫秒 (millisecond) 为单位显示时间单位。
- s：以秒 (second) 为单位显示时间单位。

波形组织

以下各小节描述了允许您在波形内组织信息的各选项。

信号与对象分组

“Group”（组）是可扩展且可折叠的容器，用于组织相关波形对象组合。组本身不显示波形数据，但可展开以显示其中内容，也可折叠以隐藏内容。您可添加、更改和移除组。

要添加组，请执行以下操作：

1. 在“Wave”（波形）中，选中一个或多个波形对象以添加到组中。

   注释：每个组均可包含分频器、虚拟总线和其他组。

2. 右键单击并从上下文菜单中选中“New Group”（新建分组）。

   这样即可将包含选定波形对象的组添加到波形配置中。

   在 Tcl 控制台中，输入 `add_wave_group` 以添加新的组。

每个组均以“Group”按钮  来表示。您可通过拖放信号或总线名称来将其他 HDL 对象移至该组中。

保存波形配置文件时，就会保存新的组及其嵌套的波形对象。

您可按如下方式移动或移除组：

- 通过拖放组名来将组移至“Name”（名称）列中的其他位置。
- 通过高亮组、右键单击并从弹出菜单中选择“Ungroup”（取消分组）来移除组。这样会将原先位于该组中的波形对象置于波形配置中的顶层层级内。

组也可以重命名。请参阅 更改对象显示名称。

注意！Delete 键可用于从波形配置中移除选定的组及其嵌套的波形对象。

使用分频器

分频器可以在 HDL 对象之间创建可视化分隔符，使某些信号或对象更便于查看。您可以将分频器添加到自己的波形配置中，以创建 HDL 对象的可视化分隔符，如下所示：

1. 在“Wave”（波形）窗口的“Name”（名称）列中，单击信号在其下方添加分频器。
2. 右键单击并选择“New Divider”（新建分频器）。

   这样当您保存波形配置文件时，就会同时保存新的分频器。

   Tcl 命令：`add_wave_divider`

您可按如下方式移动或删除分频器：

- 要将分频器移至波形中的其他位置，请拖放分频器名称。
- 要删除分频器，请将其高亮，然后按 Delete 键，或者右键单击并从上下文菜单中选择“Delete”（删除）。

分频器也可以重命名；请参阅 更改对象显示名称。

定义虚拟总线

您可将逻辑标量和矢量组合在一起添加到波形配置中，并可为该波形配置定义虚拟总线。

虚拟总线会显示总线波形，该波形值的构成方式为：将源自所添加的标量和阵列的对应值按虚拟总线下所示垂直顺序平铺成一维矢量来构成。

要添加虚拟总线，请执行以下操作：

1. 在波形配置中，选中 1 个或多个波形对象以添加到虚拟总线。
2. 右键单击并从弹出菜单中选择“New Virtual Bus”（新建虚拟总线）。

   虚拟总线会随“Virtual Bus”（虚拟总线）按钮  一起呈现。

   Tcl 命令：`add_wave_virtual_bus`

您可通过拖放信号或总线名称来将其他逻辑标量和阵列移至该虚拟总线中。

保存波形配置文件时，就会保存新的虚拟总线及其嵌套的项。您也可以通过拖放虚拟总线名称来将其移至波形中的其他位置。

您可重命名虚拟总线；欲知详情，请参阅 更改对象显示名称。

要移除虚拟总线并取消其内容分组，请高亮该虚拟总线，然后右键单击并从弹出菜单中选择“Ungroup”（取消组合）。

> **注意！** Delete 键可用于从波形配置中移除虚拟总线及总线内嵌套的 HDL 对象。

分析波形

以下各小节描述了有助于您分析波形内的数据的可用功能特性。

使用光标

光标是临时时间标记，可以频繁移动用于测量两个波形边沿之间的时间。

放置主光标和辅助光标

您只需在“Wave”（波形）窗口中左键单击即可放置主光标。

要放置辅助光标，请按住 Ctrl 键 + 单击，按住波形，向左或向后拖动。您可在光标顶部看到 1 个标志用于标记位置。或者，您也可以按住 Shift 键并单击波形中的任一点。

如果辅助光标尚未开启，那么此操作会将辅助光标设为主光标的当前位置，并将主光标置于鼠标单击的位置。

注释：要保留辅助光标的位置同时定位主光标，请按住 Shift 键然后单击。通过拖动方式放置辅助光标时，必须拖动一段最短距离，然后才会显示辅助光标。

移动光标

要移动光标，请悬停在光标上直至出现抓取符号，然后单击光标并将其拖至新位置。

在“Wave”窗口中拖动光标时，如果选中“Snap to Transition”（对齐到转换位置）波形设置（默认行为是选中此设置），那么您会看到一个空心圆或实心圆。

- 鼠标下的空心圆 ○ 表示您当前位于选定信号的波形中的转换位置之间。
- 鼠标下的实心圆 ● 表示光标已锁定在鼠标下或者标记上的波形转换位置处。

通过单击“Wave”窗口中没有任何光标、标记或浮动标尺的任意位置即可隐藏辅助光标。

查找波形上的下一个或上一个转换

“Waveform”窗口包含相应的按钮，可用于将主光标跳转到所选波形的上一个或下一个转换，或者从光标的当前位置跳转到主光标。

要将主光标移动到下一个或上一个波形转换，请执行以下操作：

1. 请单击波形中的波形对象名称，确保该波形对象处于活动状态。

   这样即可选中波形对象，此对象所显示的波形会显示比平常更粗的线条。

2. 单击波形工具栏的“Next Transition”（下一个转换）或“Previous Transition”（上一个转换）  按钮，或者使用键盘的向右或向左方向键来分别移至下一个转换或上一个转换。

提示：同时选中多个波形对象即可跳转到一组波形的最近的转换。

使用标记

如果要以永久性方式来对波形内的重大事件进行标记，请使用标记。标记支持您测量与所标记的事件相关的时序。

您可通过如下方式来添加、移动和删除标记：

- 您可在波形配置中主光标所在位置添加标记。
  1. 在“Waveform”（波形）窗口中，单击位于要添加标记的对应时间处或者转换位置处即可在此处放置主光标。
  2. 右键单击“Markers”→“Add Marker”（标记 > 添加标记）。

     这样即可在光标处放置标记，或者如果此光标位置处已存在标记，则会在稍偏移位置放置标记。该标记的时间会显示在行顶部。

     要创建新的波形标记，请使用 Tcl 命令：

     ```
     add_wave_marker <time> <timeunit> -name <name of the marker> -into
     <wcfg file>
     ```

- 您可通过拖放操作来将标记移至“Wave”窗口中其他位置。单击标记标签（位于标记或标记行顶部）并将其拖至相应的位置。
  - 在“Wave”窗口中拖动标记时，如果选中“Snap to Transition”（对齐到转换位置）选项（默认选中），那么您会看到一个空心圆或实心圆。
  - 实心圆 ● 表示您当前正悬停在选定信号的波形转换处或者正悬停在另一个标记上。
  - 如果悬停在标记上，则实心圆为白色。
  - 空心圆 ○ 表示标记已锁定在鼠标下或者另一标记上的波形转换位置处。

    松开鼠标以将标记拖到新的位置。
- 只需一条命令即可删除任一或全部标记。右键单击标记并执行以下任一操作：
  - 从弹出菜单中选中“Delete Marker”（删除标记）以删除单一标记。
  - 从弹出菜单中选中“Delete All Markers”（删除所有标记）以删除所有标记。

  您也可以使用 Delete 键来删除选定的标记。

如需了解命令使用方法，请参阅 Vivado Design Suite 帮助或《Vivado Design Suite Tcl 命令参考指南》(UG835)。

使用浮动标尺

浮动标尺使用时基来辅助时间测量，取代使用“Wave”（波形）窗口顶部的标准标尺上所示的绝对仿真时间。

您可显示（或隐藏）浮动标尺，也可以拖动浮动标尺以更改“Wave”窗口中的垂直位置。浮动标尺的时基（时间 0）属于辅助光标，或者如果没有辅助光标，则作为选定的标记来使用。

仅当辅助光标或标记存在时，浮动标尺才可见。

1. 请执行以下任一操作以显示或隐藏浮动标尺：
   - 放置辅助光标。
   - 选择标记。
2. 在“Waveform Settings”（波形设置）窗口中，启用（勾选）“Floating Ruler”（浮动标尺）选项。

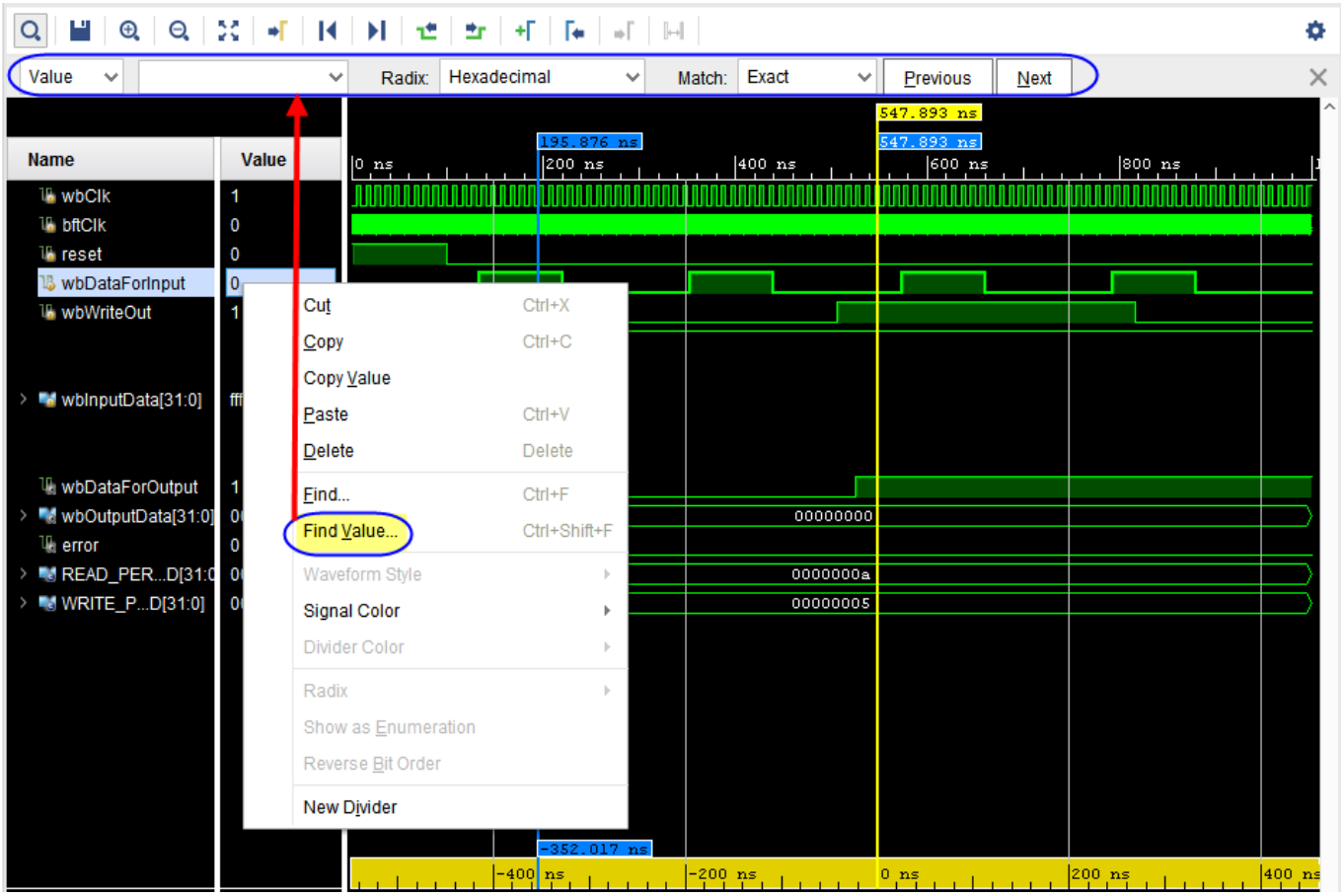
   首次执行此操作时，只需遵循该过程即可。每次放置辅助光标或选择标记时，都会显示浮动标尺。

   取消勾选/禁用“Floating Ruler”选项可隐藏浮动标尺。

在波形配置中搜索值

“Find”（查找）工具栏允许您在一个或多个波形中搜索指定值。您可以搜索精确值，例如，23*F，或者搜索匹配一组值的模式，例如“any value whose first two digits are 23 and whose fourth digit is F”（前两个位是 23 且第四个位是 F 的值）。

图 28： **“Find Value”（查找值）选项和“Find”（查找）工具栏**



重要提示！此搜索功能特性仅支持“logic”（逻辑）类型的标量和矢量 (1-D) 波形对象。逻辑类型包含二态或四态类型的 Verilog/SystemVerilog 以及 VHDL 的 bit 和 std_logic。

请遵循以下步骤执行搜索：

1. 在“Name”（名称）列中，选中一个或多个设计波形对象，即包含波形的波形对象。
2. 在“Name”列或“Value”（值）列中，右键单击选中的波形对象之一并选择“Find Value”（查找值）选项即可激活“Find”工具栏。
3. 在“Find”工具栏上，从“Radix”（基数）下拉列表中选择搜索值的基数。搜索功能支持以下基数：
   - 二进制
   - 十六进制
   - 八进制
   - 无符号十进制

- 有符号十进制

4. 在“Find”工具栏上的空白文本框中，根据您所选基数，输入由有效数位字符串组成的值模式。有效数位包括数值、VHDL MVL 9 字面值（U、X、0、1、Z、W、L、H 和 -）以及 Verilog 字面值（0、1、x 和 z）。

   注释：如果您输入的数位无效，那么该文本框会变为红色，并在工具栏右侧显示一条错误消息。有效的数值组合取决于基数。例如，如果您选择“Octal”（八进制）基数，那么数值为 0 到 7 之间的数值。十六进制的数位包括 0 到 9 和 A 到 F（或 a 到 f）。您可输入特殊数位“.”来指定匹配任意数位值。例如，八进制值模式 12.4 匹配波形中出现的 1234、1204 和 12X4。

5. 从“Match”（匹配）下拉列表中的以下选项选择匹配样式：

   - “Exact”（精确）：波形值所含的数位的数量与值模式中所含数位的数量必须完全相同，才能视作为匹配。例如，值模式 1234 与波形中出现的 1234 相匹配，但与 123 或 12345 不匹配。

     提示：采用“Exact”匹配样式时，您可以省略值模式中的前导零位。例如，要在波形中查找值 0023，可以指定值模式 0023 或者只需指定 23 即可。

   - “Beginning”（起始）：只要任意波形值的起始数位与值模式相匹配即可视作为匹配。例如，值模式 1234 与波形中出现的 1234 和 12345 相匹配，但与 1235 或 123 不匹配。该选项仅可用于下列基数：“Binary”（二进制）、“Octal”（八进制）和“Hexadecimal”（十六进制）。
   - “END”（末尾）：只要任意波形值的末尾数位与值模式相匹配即可视作为匹配。例如，值模式“1234”与波形中出现的 1234 和 91234 相匹配，但与 1235 或 234 不匹配。该选项仅可用于下列基数：“Binary”（二进制）、“Octal”（八进制）和“Hexadecimal”（十六进制）。
   - 单击“Next”（后一项）按钮或者按 Enter 键将主光标按正向移至最近的匹配，或者单击“Previous”（前一项）按钮将主光标按反向移至最近的匹配。选中多个波形对象后，光标会停止于任意选中波形对象的最近的匹配。

     提示：如果请求的方向中没有找到任何匹配，那么光标会保持静止，并在工具栏右侧显示一条“Value not found”（未找到值）消息。

分析 AXI 接口传输事务

如果您使用 Vivado IP integrator 将自己的设计编写为块设计，那么启动 Vivado 仿真器时，Vivado 会自动将 AMBA® AXI 接口从您的设计导入 Vivado 仿真器，并将其作为协议实例，以供在波形窗口中查看。将 AXI 接口的协议实例添加到波形窗口中后，它会为您显示仿真期间该接口上发生的数据传输事务。

认识协议实例

AXI 接口由一组标准逻辑信号组成，这些信号是由 Arm® 在 AMBA® AXI 和 ACE 协议规范以及 AMBA® 4 AXI4-Stream 协议规范中所定义的。这些信号按上述规范中所述方式来传达已编码为逻辑事件的数据传输事务。Vivado 仿真器使这些信号可供在波形查看器内直接查看。但难以单独直观显示从这些信号发生了哪些传输事务。

为了便于查看传输事务，Vivado 仿真器提供了一项功能特性，可以分析信号活动，并生成新信号以在传输事务级别汇总这些活动。此进程称为“protocol analysis”（协议分析）。对于每个 AXI 接口，Vivado 仿真器都会创建名为“protocol instance”（协议实例）的新设计对象，以表示 AXI 接口以及协议分析的输入和输出。协议实例通常与其输入信号驻留在相同作用域内。

使用 IP integrator 标记 AXI 接口以供在 Vivado 仿真器内查看

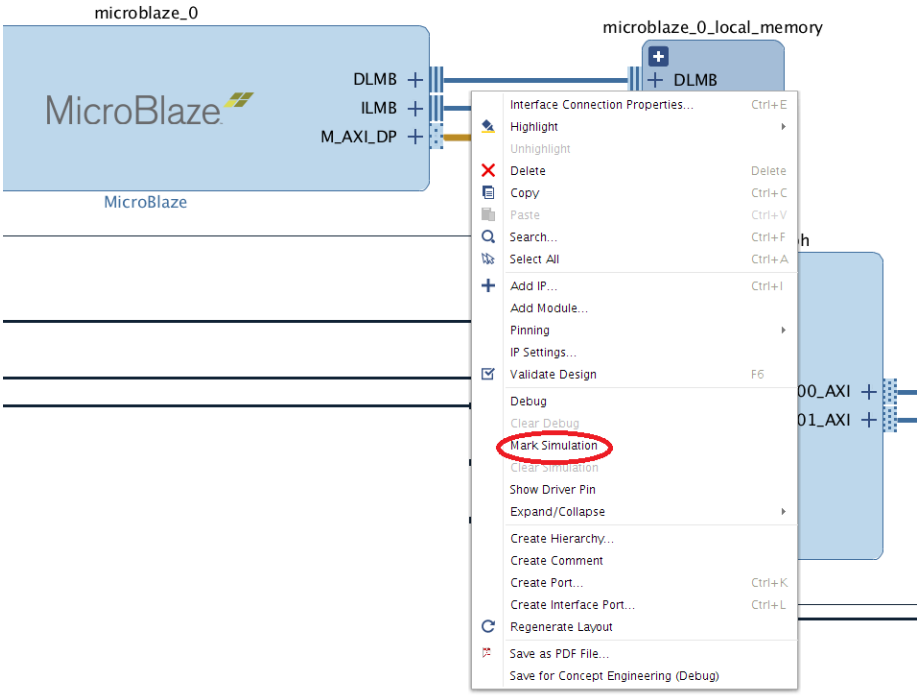
Vivado IP integrator 提供了相应的功能特性用于识别要在 Vivado 仿真器的波形查看器的块设计窗口内直接显示的 AXI 接口。请执行以下步骤以标记要在 Vivado 仿真器内查看的 AXI 接口：

1. 找到要查看的 AXI 接口。
2. 右键单击对应的信号线连接（下图所示的橙色线）。
3. 单击“Mark Simulation”（标记仿真）。

   注释： “Mark Simulation”只能应用于 AXI 接口。
4. 重复步骤 1-3 来标记其他接口。
5. 单击“Clear Simulation”（清除仿真）选项以清除已标记的 AXI 接口。

   注释：仅当已标记 AXI 接口时，“Clear Simulation”选项才适用。
6. 启动 Vivado 仿真器。按提示保存块设计。

   启动 Vivado 仿真器时，您标记的接口就会显示在“Wave”窗口中。如果 Vivado 工程已自定义为自动打开波形配置，那么已标记的接口如果在波形配置中不存在，就会被自动添加到此配置中。如果 Vivado 工程未自定义为打开波形配置，Vivado 仿真器会创建默认波形配置，其中会包含已标记的接口取代一般情况下的顶层 HDL 信号列表。

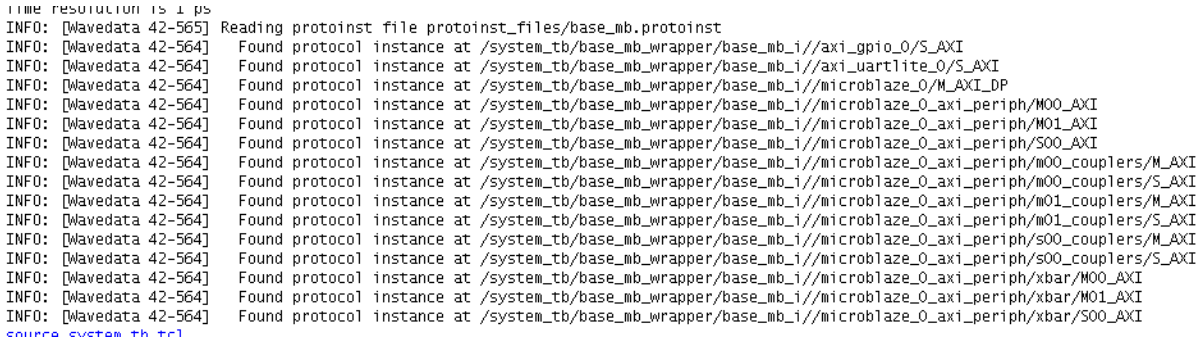


注意！根据 AXI Interconnect 的配置，其内部的部分 AXI 接口可能无法正确显示在波形查看器内。建议您仅标记位于其边界处的接口。

在 Vivado 仿真器中查找协议实例

当 Vivado 仿真器启动时，它会扫描设计及其输入文件以查找协议实例。扫描结果显示在 Tcl 控制台中仿真器输出顶部附近，如下图所示。您可将协议实例路径从 Tcl 控制台复制粘贴到 Tcl 命令中。

图 29：**Tcl 控制台中识别的协议实例**



在“Objects”窗口中查找协议实例

协议实例对象显示在“Objects”（对象）窗口中包含对应 AXI 接口信号的作用域内。要使用“Scope”（作用域）窗口查找协议实例，请执行以下步骤：

1. 在“Scope”窗口中，选择包含 AXI 接口信号的作用域。

   注释：作用域层级与您的块设计大致匹配。

2. 在块设计中找到块的 AXI 接口的协议实例。选中与块的实例名称相匹配的作用域名称。

要使用“Objects”窗口查找协议实例，请执行以下步骤：

1. 在“Objects”窗口中，滚动至列表底部。
2. 找到与块设计中的 AXI 接口的端口名称相匹配的协议实例名称。AXI 端口名称通常以 `_AXI` 结尾，常用 `M_AXI` 或 `S_AXI`。

> **提示：**协议实例的端口模式为“Internal Signal”（内部信号）。为加速在“Objects”窗口中搜索协议实例，您可隐藏除“Internal Signal”对象外的所有对象。单击齿轮按钮，取消选中“Select All”（全选）复选框，并选中“Internal Signal”复选框。要复原“Objects”窗口，请选中“Select All”复选框。

使用 Tcl 命令查找协议实例

协议实例对象包含 `proto_inst` 的 Tcl `type` 字段。您可使用 `get_objects` Tcl 命令在特定作用域内或特定作用域下查找协议实例。

使用以下命令查找设计中的所有协议实例：

```
get_objects /* -r -filter {type==proto_inst}
```

使用以下命令查找作用域中的所有协议实例：

```
get_objects <Design scope hierarchy>/* -filter {type==proto_inst}
```

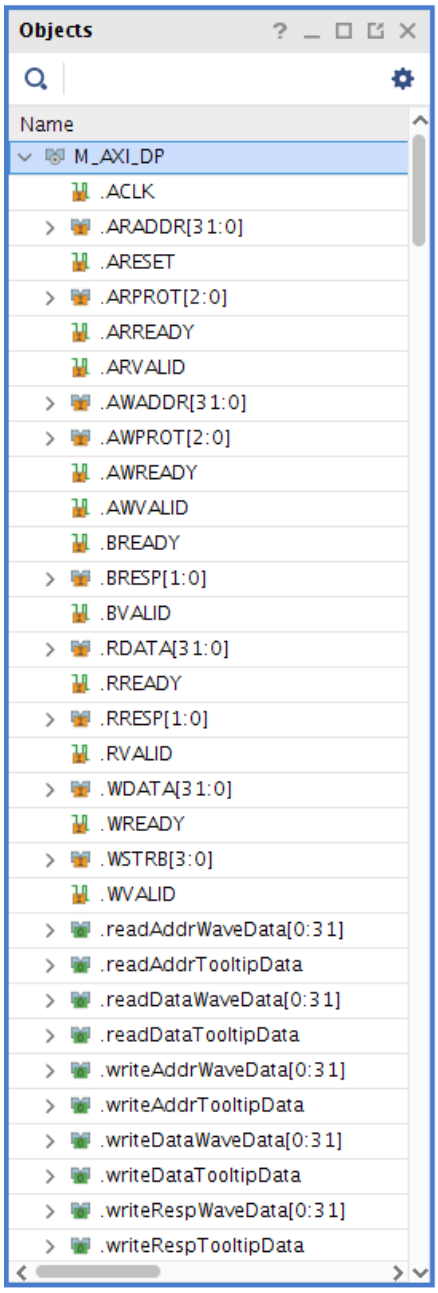
使用以下命令查找作用域中或作用域下的所有协议实例：

```
get_objects /system_tb/base_mb_wrapper/base_mb_i/* -r -filter {type==proto_inst}
```

“Objects”窗口中的协议实例

在“Objects”（对象）窗口中，协议实例显示为聚集设计对象，其名称与块设计中的 AXI 接口的端口名称相同。单击箭头扩展按钮即可查看协议实例的协议分析的输入和输出，如以下名为 `M_AXI_DP` 的协议实例图例所示。

图 30：**“Objects”窗口中的协议实例**

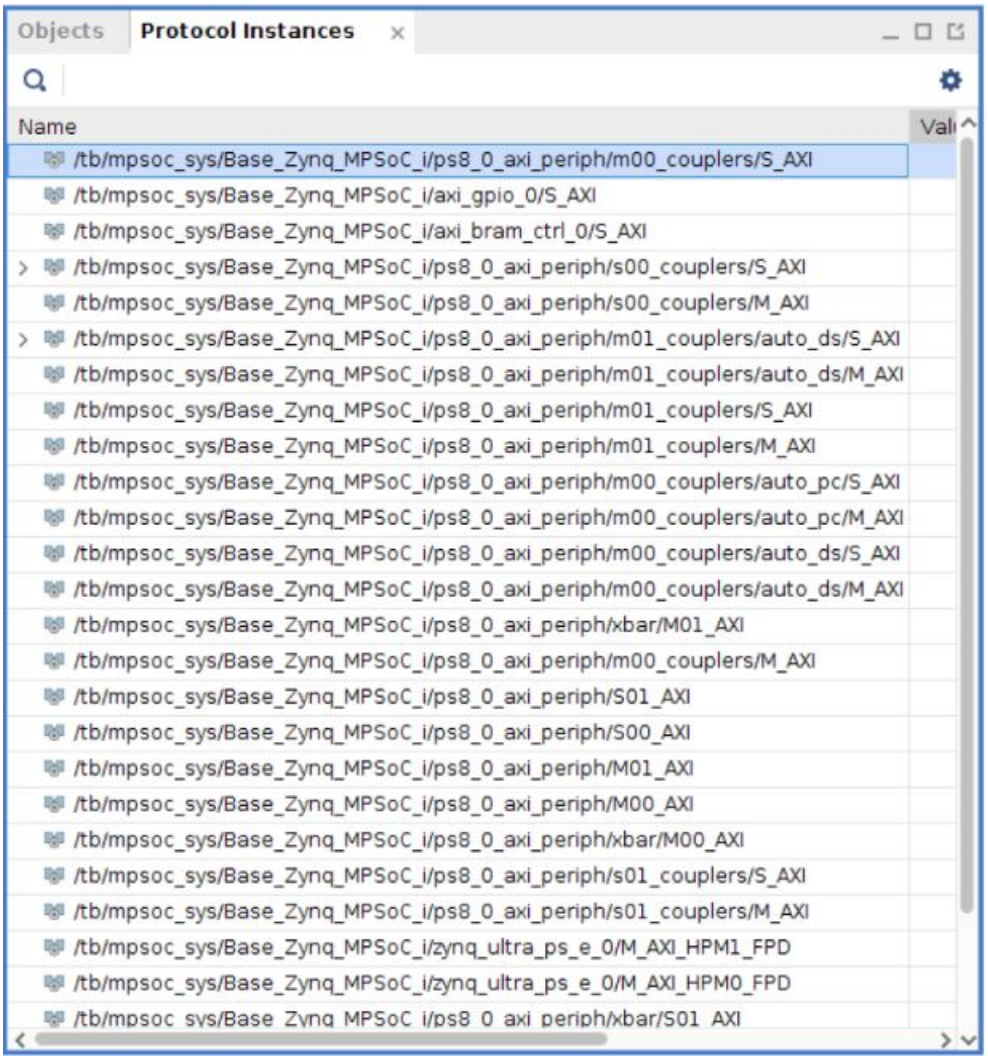


注释：为最优化计算机资源，协议实例不显示其子对象，直至首次将该协议实例添加到波形窗口中为止。

- 协议实例窗口：

协议实例窗口包含给定设计中显示的完整协议实例列表。它包含指向协议实例的绝对路径，以确保可根据路径来区分同名的实例。

图 31：**协议实例窗口**



- 协议实例输入：

  显示有橙色图标的协议实例输入信号是来自 HDL 设计的 AXI 信号的混叠。将鼠标悬停在输入上即可查看工具提示，其中提供了别名的完整路径以及实际（混叠的）信号的完整路径。您也可以在“Scope and Objects”（作用域和对象）窗口中从协议实例输入跳转到实际信号。执行以下步骤即可跳转到协议实例输入的实际信号：

  1. 要查看“Objects”窗口中的所有信号类型，请单击齿轮图标，然后选中“Check All”（全选）复选框。
  2. 右键单击协议实例输入。
  3. 选择“Go To Actual”（转至实际值）。

提示：如果要在波形配置 (WCFG) 文件中保存协议实例输入信号，请改为添加输入信号的实际信号。因为，仅在加载 WCFG 文件后才会创建协议实例输入和输出，波形配置中不包含保存的输入。在 WCFG 文件中无法保存协议实例输出。

- 协议实例输出：

  协议实例输出以绿色 O 图标来显示。这些是协议实例的特殊信号，在 HDL 设计中没有对应的信号。输出信号仅向波形查看器生成有意义的事件，用于显示传输事务。

> **注意！**一组协议实例输出信号及其内容可能随不同 Vivado 版本而变。建议使用 Tcl 生成脚本时，不要依赖于协议实例输出信号的特定行为来判断。

将协议实例添加到波形窗口

您可将设计中存在的任何协议实例添加到波形窗口中。将协议实例添加到波形窗口会导致 Vivado 仿真器在协议实例输入上运行协议分析，从仿真时间 0 开始且与已耗用的仿真时间多少无关。由于协议分析使用波形数据库 (WDB)，在波形数据库中始终追踪所有协议实例的输入，即使您并未请求追踪输入，或没有将协议实例添加到波形窗口中，也同样如此。

除了在 IP integrator 块设计中按 使用 IP integrator 标记 AXI 接口以供在 Vivado 仿真器内查看 小节所述标记 AXI 接口外，您可以使用“Objects”（对象）窗口或 Tcl 命令将协议实例添加到波形窗口中。

> **重要提示！**协议实例可能使用更多计算机资源，建议您仅添加当前所需的协议实例。您始终可以在后续仿真期间添加更多协议实例，而不会造成数据丢失。

执行以下步骤即可使用“Objects”窗口将协议实例添加到“Wave”窗口：

1. 要在“Objects”窗口中查找协议实例，请参阅 在“Objects”窗口中查找协议实例 小节中所述步骤。
2. 通过以下两种方式将协议实例添加到“Wave”窗口中：
   a. 右键单击协议实例，并选择“Add to Wave”（添加到波形）。
   b. 将协议实例拖放到“Wave”窗口的“Name”（名称）列。

执行以下步骤即可使用 Tcl 命令将协议实例添加到“Wave”窗口：

1. 要在“Objects”窗口中查找协议实例，请参阅 在 Vivado 仿真器中查找协议实例 小节中所述步骤。
2. 将协议实例路径复制到剪贴板：
   a. 如果您已在“Objects”窗口中找到协议实例，请左键单击将其选中，并复制协议实例路径。
   b. 如果您已在 Tcl 控制台中找到协议实例，请使用鼠标来选中协议实例路径并复制。
   c. 如果您已使用 `get_objects` Tcl 命令找到协议实例，请使用鼠标来选中 Tcl 控制台中的协议实例路径文本并复制。或者，您可按以下章节所述同时获取多个对象。
3. 输入 `add_wave` 并粘贴协议实例名称。

> **提示：**如果您的协议实例路径包含特殊字符，请使用双括号将路径括起。例如，`add_wave {{path}}`。

以编程方式使用 get_objects

按 使用 Tcl 命令查找协议实例 中所述使用 `get_objects` Tcl 命令时，此命令会以 Tcl 列表方式返回协议实例。您可将此列表存储在 Tcl 变量内：

```
set p [get_objects -r /* -filter {type==proto_inst}]
```

并将此列表与 `add_wave` Tcl 命令搭配使用来添加列表中的所有协议实例：

```
add_wave $p
```

或者使用内置 `lindex` 命令添加来自列表的特定协议实例，如以下示例所示，该示例用于添加来自列表的首个协议实例：

```
add_wave [lindex $p 0]
```

在“Wave”窗口中分析协议实例

本节描述了所有接口类型通用的波形功能特性。请参阅以下对应各协议的章节，以获取有关特定接口类型（AXI4 或 AXI4-Stream）的更多信息。

认识“Wave”窗口中的协议实例

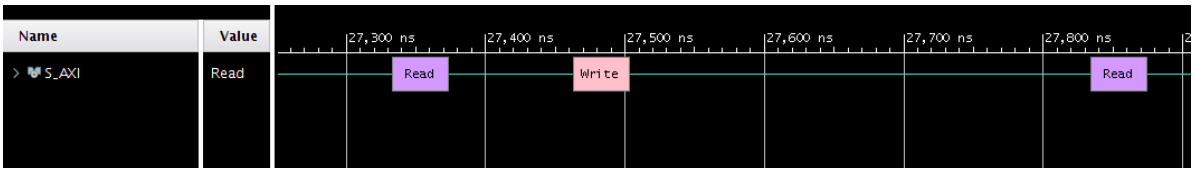
将协议类型添加到波形窗口中时，Vivado 仿真器会创建波形对象层级以表示协议实例。您无法更改此层级的结构。AXI 接口的类型用于确定该层级。

提示：您可能需要查看波形对象层级内不包含的协议实例输入信号。无法将信号添加到层级中时，可以将其添加到该层级之前或之后。

认识传输事务波形

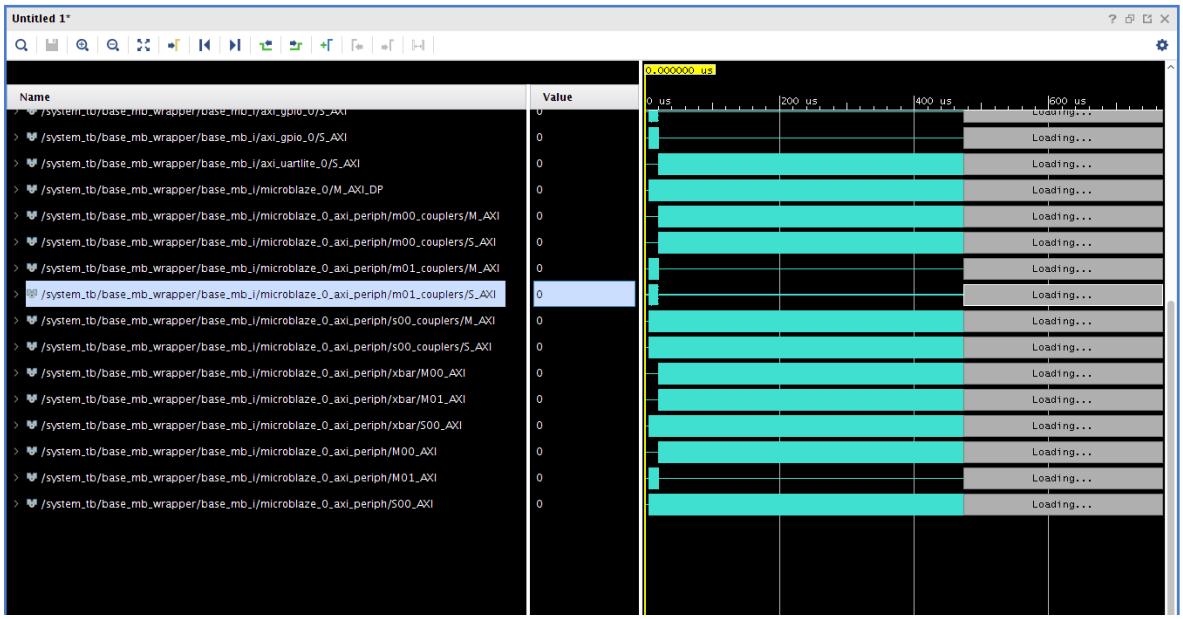
传输事务波形有别于其他类型的波形。传输事务波形显示了仿真的设计的某些方面保持活动和不活动状态的周期，并将其与显示信号值随时间而发生的更改进行对比。下图显示了传输事务波形的示例。细线条表示不活动状态的周期，而矩形则表示活动状态（通常称为传输事务条）的周期。下图示例显示了三个传输事务条。

图 32：**传输事务波形显示**



如下图所示，传输事务波形显示了一个灰色条，其中包含文本 `Loading`，与此同时在协议实例的输入上正在执行协议分析。随着协议分析的进行，灰色条逐渐缩小，以显示新处理的传输事务数据。

图 33：**显示未完成的协议分析的传输事务波形**

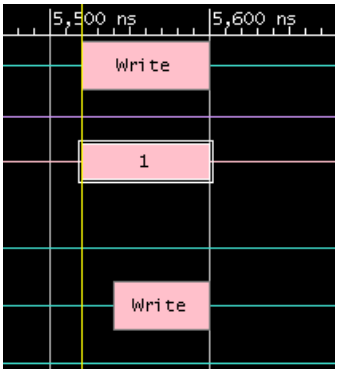


使用传输事务条

选择传输事务条

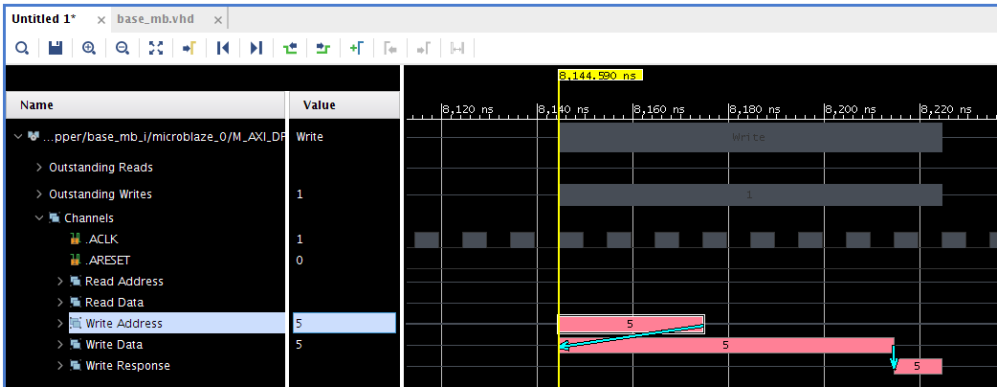
将光标置于传输事务条上并左键单击以选中高亮的传输事务条（含双边框），如下图所示：

图 34：**选中的传输事务条**



如果所选传输事务条是某一组相关传输事务条中的成员，则会显示一组箭头（称为关联）用于连接相关传输事务条。波形窗口中的其余对象会变暗，以便高亮显示该组传输事务条。下图显示了选定的传输事务条及其关联的传输事务条（以关联相连接）。

图 35：**含关联的传输事务条**



按 Esc 键即可清除所选传输事务条。

提示：要将主光标重新定位到传输事务波形内，请按住 Ctrl 键并左键单击所需光标时间。

使用关联导航传输事务

单击关联的一端时，所选项会移至位于关联另一端的传输事务条，波形窗口会滚动，使另一端显示在视图内。

注释：如果另一端已在视图内，波形窗口可能不发生滚动。

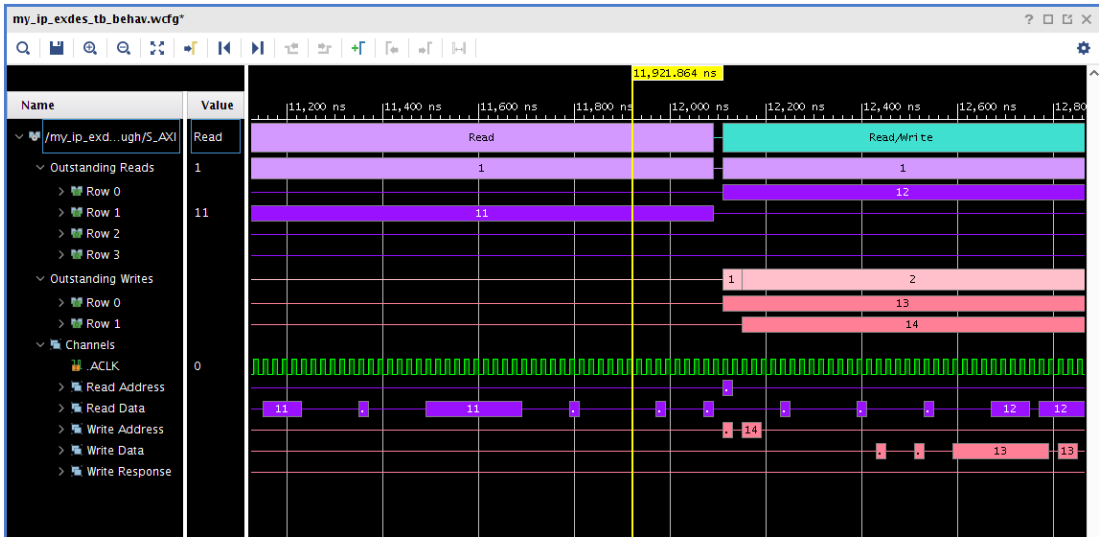
工具提示

使用鼠标悬停在传输事务条或关联上时，会根据协议实例接口的类型出现相应的工具提示，显示有关传输事务条或关联的额外信息。

分析 AXI 存储器映射 (AXI-MM) 接口

本节描述了 AXI4 协议实例专用的传输事务查看功能特性。AXI4 接口的协议实例显示在波形窗口中，所含波形对象层级如下图所示。

图 36：**AXI-MM 接口**



认识顶层汇总行

AXI4 协议实例的波形对象层级顶层是顶层汇总行。此传输事务波形基于下列规则显示了 AXI 接口的总体读写活动：

- 如有一项或多项 AXI 读取传输事务正在进行中，那么顶层汇总会显示紫色读取传输事务条。
- 如有一项或多项 AXI 写入传输事务正在进行中，那么顶层汇总会显示粉色写入传输事务条。
- 如有一项或多项 AXI 读写传输事务正在进行中，那么顶层汇总会显示青色读写传输事务条。

AXI 传输事务是抽象概念，不应与图形化传输事务条混淆。它是使用 AXI 信令执行的完整数据交换，包含地址、数据以及（可选）响应阶段。

提示：出于性能原因，波形查看器缩小后，不会以不同颜色显示传输事务条。而是改为以青色显示所有传输事务条。您需要放大才能区分读取传输事务与写入传输事务。

认识未完成的读取行和未完成的写入行

在 AXI4 协议实例的波形对象层级顶层下方有一组未完成的 AXI 读取传输事务和一组写入传输事务。如果主接口发起 `A*VALID` 或 `WVALID` 但最后数据阶段或（可选）响应阶段尚未完成，那么此类 AXI 传输事务称为未完成。未完成的读取行会显示当前未完成的 AXI 读取传输事务的计数，或者该行会显示不活动状态（以一条细线来显示）表示未完成的 AXI 读取传输事务数量为 0。同样，未完成的写入行会显示当前未完成的 AXI 写入传输事务的计数，或者该行会显示不活动状态以表示未完成的 AXI 写入传输事务数量为 0。

认识传输事务汇总行

在每个未完成的读取和写入行下方都有一组传输事务汇总行，标记为“Row <n>”，其中 <n> 是整数。传输事务汇总是表示单个 AXI 传输事务的传输事务条，该传输事务从 AXI 传输事务的首个阶段开始并截止于最后一个阶段。将传输事务汇总分配到带有特定编号的行并非旨在传递任何特殊含义，而是为了防止在同一行内有多个未完成的 AXI 传输事务发生重叠。

提示：传输事务汇总行的数量可随仿真进程而增加。出于性能原因，波形窗口仅在协议分析完成后才会更新该行。要在仿真期间查看这些行的最新状态，而不等待整个仿真完成，可以暂停仿真并允许“Loading”（正在加载）栏消失。

每个传输事务汇总都带有一个序号标记。第一个 AXI 传输事务的序号为 1，第二个 AXI 传输事务的序号为 2，以此类推。读写序号的递进彼此分离，并且与所有其他协议实例的 AXI 传输事务也彼此分离。例如，某个特定协议实例可具有序号为 16 的 AXI 读取传输事务，和另一个同样序号为 16 的 AXI 写入传输事务。

认识通道行

通道波形对象组默认处于折叠状态。展开该组时，可以看到 AXI 接口时钟和复位（如果存在）的逻辑信号，接口中存在的每个 AXI 通道各占一个传输事务行。

注释：并非全部 5 条通道都需存在于 AXI 接口内。对于只读接口，不含写入通道。对于只写接口，不含读取通道。采用写入通道的部分 AXI 接口可以省略响应通道，前提是 AXI 主接口不使用响应信息。

每个通道行都会显示一个传输事务条，用于汇总该 AXI 通道从 VALID 到 READY 的各握手，例外是相同 AXI 传输事务的多个连续数据节拍显示为单个传输事务条。为了将单个 AXI 传输事务的所有通道传输事务条以可视方式捆绑在一起，每个通道传输事务条都带有与对应传输事务汇总相同的序号标记。您可以展开通道行，以显示该通道的关键 AXI 信号。

提示：您可能需要查看波形对象层级内不包含的协议实例输入信号。无法将信号添加到层级中时，可以将其添加到该层级之前或之后。

提示：发生对应 AXI 信号事件后的一个时钟周期内会出现通道传输事务条。AXI 协议分析器将发生在正时钟沿上或之后的 AXI 信号事件视作为在下一个正时钟沿上生效。

当您将鼠标悬停在任何通道传输事务条、关联或传输事务汇总上时，都会出现工具提示，显示来自此 AXI 传输事务的地址阶段的参考性 AXI 地址通道信号的值。

注释：在工具提示中省略该接口中缺少的可选 AXI 地址通道信号。

选中通道传输事务条时，在与所选传输事务条参与相同 AXI 传输事务的所有通道传输事务条之间会显示关联。您可单击关联箭头尾部来关注该 AXI 传输事务从地址阶段到响应阶段的进展。关联链始终从地址阶段传输事务开始，即使数据阶段位于地址阶段之前也是如此。

错误条件

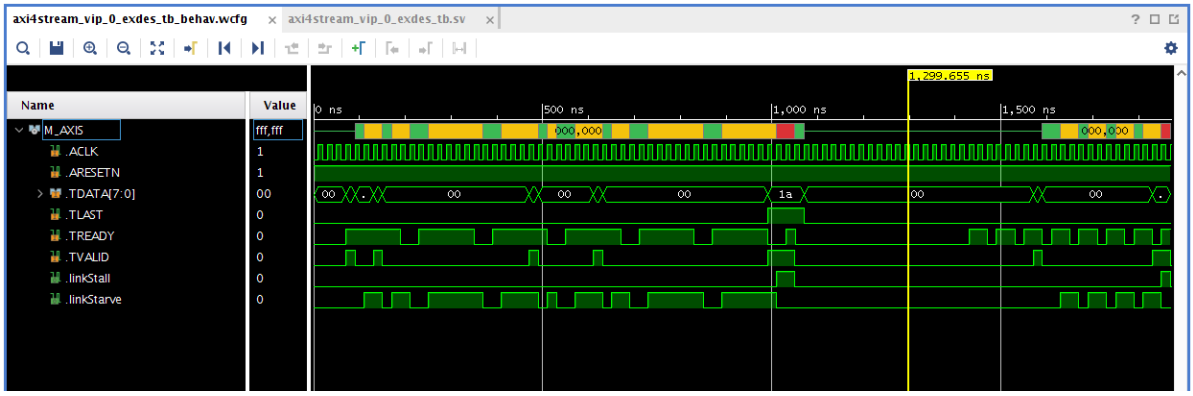
如果接口上存在握手错误，您可能会看到通道传输事务序号包含全部由 9 组成的一个字符串。该序号表示数据和/或响应阶段无法与地址和/或数据阶段匹配。常见原因是读/写 ID 标签不匹配，以及 AXI 各阶段进行过程中，协议分析器保持处于复位状态（ARESET 或 ARESETn 信号处于活动状态）。

注意！由于 AXI 互连的某些配置是专为性能（而非传输事务调试）而最优化的，因此 AXI Interconnect 内部的 AXI 接口遵循的一组复位信号可能与连接到该接口的复位信号不同，从而导致波形查看器中出现传输事务错误。如果您在接口上观测到传输事务错误，建议您改为监控互连外部的接口。

分析 AXI4-Stream (AXI-S) 接口

本节描述了 AXI4-Stream 协议实例专用的传输事务查看功能特性。AXI-S 接口的协议实例显示在波形窗口中，所含波形对象层级如下图所示。

图 37：**AXI-Stream (AXI-S) 接口**



AXI-S 接口的波形对象层级仅包含一个传输事务行，位于层级顶部。该行中的每个传输事务条都对应于一起完整的 AXI 传输事务。此传输事务上的文本包含来自 AXI 信号 TID 的串流标识符和来自 AXI 信号 `TDEST` 的粗略布线信息。

传输事务条包含颜色编码的条带，用于指示 AXI 传输事务的状态，如下表所述。

表 12：**AXI 传输事务状态**

颜色	状态	描述
绿	正常	数据正常串流。
黄色	匮乏	从接口正在等待来自主接口的数据。
红	停滞	主接口生成数据的速度比从接口耗用数据的速度更快。

在顶层传输事务行下，波形对象层级包含部分关键 AXI 信号以及 `stall` 和 `starve` 状态信号，以指示停滞状态和匮乏状态。状态信号提供的信息与传输事务行中彩色条带所提供的信息相同。

提示：发生对应 AXI 信号事件后的一个时钟周期内会出现通道传输事务条。AXI 协议分析器将发生在正时钟沿上或之后的 AXI 信号事件视作为在下一个正时钟沿上生效。

错误条件

握手错误会生成错误传输事务，其中所含文本全为 F。

第 6 章

使用 Vivado 仿真器调试设计

AMD Vivado™ Design Suite 仿真器支持您：

- 检验源代码
- 设置断点并运行仿真直至到达断点为止
- 单步跳过各代码节
- 强制将波形对象设为特定值

本章描述了调试方法并包含调试进程中实用的 Tcl 命令。此外还提供了有关使用第三方仿真器进行调试的流程描述。

源码级别调试

您可调试自己的 HDL 源代码，以跟踪设计中的意外行为。调试的完成方式是通过受控方式执行源代码来确定问题可能出自何处。可用调试策略有：

- 逐行单步执行代码：对于开发中任意点的任意设计，您都可使用 `step` 命令来逐行调试自己的 HDL 源代码，以验证设计是否按期望方式运行。在运行每一行代码后，再次运行 `step` 命令以继续分析。如需了解更多信息，请参阅单步执行仿真。
- 在特定 HDL 代码行上设置断点，运行仿真直至达到断点：在较大的设计中，在运行每一行 HDL 源代码后都停止可能较为繁琐。您可在 HDL 源代码中预先确定的任意点处设置断点，从测试激励文件开始处或者从设计内您所在的当前位置运行仿真，并在每个断点处停止。在停止后，您可使用 Step、Run All 或 Run For 命令来继续执行仿真。如需了解更多信息，请参阅以下 使用断点 章节。
- 设置条件。该工具会评估每一项条件并在满足条件时执行 Tcl 命令。使用 Tcl 命令：

  ```
  add_condition <condition> <instruction>
  ```

  如需了解更多信息，请参阅 添加条件。

单步执行仿真

您可使用 `step` 命令来验证设计是否按期望方式工作，此命令每次执行一行 HDL 源代码。

代码行会高亮显示，并显示一个箭头，指向当前执行的代码行。

您也可以在单步执行仿真时创建断点以便额外增加停止位置。如需了解有关在仿真器内调试策略的更多信息，请参阅以下 使用断点 章节。

1. 要单步执行仿真，请执行以下步骤：
   - 在当前运行时间内，选中“Run > Step”（运行 > 步进）或单击“Step”（步进）按钮  。

这样与顶层设计单元关联的 HDL 会作为新视图在“Wave”窗口中打开。

- 在开始时 (0 ns)，重新启动仿真。使用“Restart”（重新启动）命令将时间复位到测试激励文件开始时间。请参阅 第 4 章：使用 Vivado 仿真器进行仿真。

2. 在波形配置窗口中，右键单击波形或 HDL 选项卡，并选择“Tile Horizontally”（横向平铺）以便同时查看波形和 HDL 代码。
3. 重复“Step”操作，直至完成调试为止。

执行每一行代码时，您可看到箭头随之下移。如果仿真器正在另一个文件内执行代码行，那么会打开新文件，箭头会单步执行其中代码。在大多数仿真中，运行“Step”命令时一般都会打开多个文件。Tcl 控制台还会指出 step 命令已处理的 HDL 代码量。

使用断点

断点是源代码中用户判定的停止点，可供您用于调试设计。

提示：断点对于大型设计调试尤为实用，在此类设计中，使用 Step 命令（在每一行代码上都停止仿真）执行调试可能过于繁琐且耗时。

您可在 HDL 文件的可执行行内设置断点，这样即可连续运行代码直至仿真器遇到断点为止。

注释：您可在仅含可执行代码的行上设置断点。如果您将断点置于不可执行的代码行上，则不会添加此断点。

1. 运行仿真。
2. 转至您的源文件，单击感兴趣的源码行左侧的空心圆  。红点  表示确认断点已设置成功。

   完成该过程后，会在代码行旁打开仿真断点按钮。

   输入 Tcl 命令：`add_bp <file_name> <line_number>`

   此命令会在 `<file_name>` 的 `<line_number>` 处添加断点。如需了解命令使用方法，请参阅 Vivado Design Suite 帮助或《Vivado Design Suite Tcl 命令参考指南》(UG835)。

   打开 HDL 源文件。
3. 在 HDL 源文件中的可执行行上设置断点。
4. 重复步骤 1 和 2 直至设置完所有断点。
5. 使用“Run”（运行）选项来运行仿真：
   - 要从头开始运行，请使用“Run > Restart”（运行 > 重新启动）命令。
   - 使用“Run > Run All”（运行 > 全部运行）或“Run > Run For”（运行 > 运行对象）命令。

   这样仿真会运行直至到达断点，然后停止。

   HDL 源文件会显示箭头，指示断点停止点。
6. 重复步骤 4 继续执行仿真，逐个完成每个断点，直至您对结果满意为止。

   受控制的仿真会持续运行并在您的 HDL 源文件中所设的每个断点处停止。

   设计调试期间，您还可运行“Run > Step”（运行 > 步进）命令来推进仿真，以便对设计进行更详细的调试。

您可从 HDL 源代码中删除单个断点或所有断点。

要删除单个断点，请单击“Breakpoint”（断点）按钮  。

要移除所有断点，请选中“Run > Delete All Breakpoints”（运行 > 删除所有断点），或者单击“Delete All Breakpoints”（删除所有断点）按钮  。

要删除所有断点，请执行以下操作：

- 输入 Tcl 命令：`remove_bps -all`。

要获取指定断点对象列表的断点信息，请执行以下操作：

- 输入 Tcl 命令：`report_bps`

添加条件

要基于条件添加断点并输出诊断消息，请使用以下命令：

```
add_condition <condition> <message>
```

使用 Vivado IDE BFT 设计示例时，如果要在 `wbClk` 信号和 `reset` 均为高电平有效时停止，请在仿真启动时发出以下命令以在 reset 转至 1 且 wbClk 转至 1 时打印诊断消息并暂停仿真：

```
add_condition {reset == 1 && wbClk == 1} {puts "Reset went to high"; stop}
```

在 BFT 示例中，添加的条件导致在满足此条件时，仿真在 5 ns 处暂停并在控制台中打印 `"Reset went to high"`。仿真器会等待下一条 step 或 run 命令以恢复仿真。

-notrace 选项

正常情况下执行 `add_condition` 命令时，指定的 Tcl 命令还会回显到控制台、log 日志文件和 journal 日志文件内。`-notrace` 开关会导致静默执行这些命令，禁止在以上三处位置显示这些命令（但不禁止显示其输出）。

例如，如果执行以下命令示例：

```
puts 'Hello'
```

以上命令的正常行为是在控制台、log 日志文件和 journal 日志文件中发出以下输出：

```
# puts 'Hello'
Hello
```

执行 `-notrace` 开关时，仅生成以下输出：

```
Hello
```

暂停仿真

无论仿真运行多久，您都可以使用“Break”（中断）命令暂停仿真，此命令会使仿真会话保持处于打开状态。

要暂停运行中的仿真，请选择“Simulation > Break”（仿真 > 中断）或者单击“Break”按钮  。

仿真器会在下一个可执行 HDL 行上停止。仿真停止时所在的行会显示在文本编辑器中。

注释：此行为适用于使用 `-debug <kind>` 开关编译的设计。

您可使用“Run All”（全部运行）、“Run”（运行）或“Step”（步进）命令随时恢复仿真。如需了解更多信息，请参阅 单步执行仿真。

追踪仿真的执行

您可在 Tcl 控制台上为仿真运行期间遇到的每个源码行显示一条注释。这种连续显示遇到的行的方式称为行追踪。

要开启行追踪，请使用以下任一 Tcl 命令：

```
ltrace on
set_property line_tracing true [current_sim]
```

要关闭行追踪，请使用以下任一 Tcl 命令：

```
ltrace off
set_property line_tracing false [current_sim]
```

您可在 Tcl 控制台上为仿真运行期间遇到的每个进程显示一条注释。这种连续显示遇到的进程的方式称为进程追踪。

要开启进程追踪，请使用以下任一 Tcl 命令：

```
ptrace on
set_property process_tracing true [current_sim]
```

要关闭进程追踪，请使用以下任一 Tcl 命令：

```
ptrace off
set_property process_tracing false [current_sim]
```

强制将对象设为特定值

使用 Force 命令

Vivado 仿真器提供了交互机制，用于在指定时间或指定时间段将信号、连线或寄存器强制设为指定值。您也可将对象上的值设为过一段时间后就强制更改。

提示：“force”不仅是操作（即，改写信号上 HDL 定义的行为），也是 Tcl 一类对象，可保留在 Tcl 变量中。

您可对 HDL 信号使用 force 命令，使其按 HDL 设计中定义的方式来改写此信号的行为。例如，您可选择将信号的行为改为：

- 向 HDL 测试激励文件本身不驱动的测试激励文件信号提供激励
- 在调试期间临时纠正错误的值（以便允许您继续分析问题）

可用的 force 命令包括：

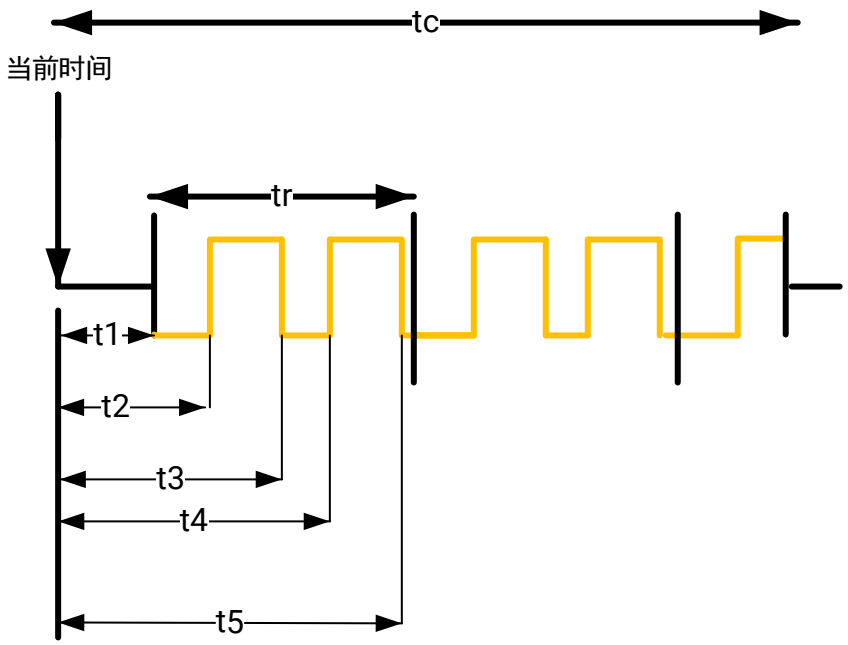
- “Force Constant”（强制设为常量）
- “Force Clock”（强制设为时钟）

- “Remove Force”（移除强制）

下图显示了给定以下命令时，应用 `add_force` 功能的方式：

```
add_force mySig {0 t1} {1 t2} {0 t3} {1 t4} {0 t5} -repeat_every tr -
cancel_after tc
```

图 38：**-add_force 功能图示**



您可在 Tcl 控制台中输入以下内容以获取有关命令的更多详细信息：

```
add_force -help
```

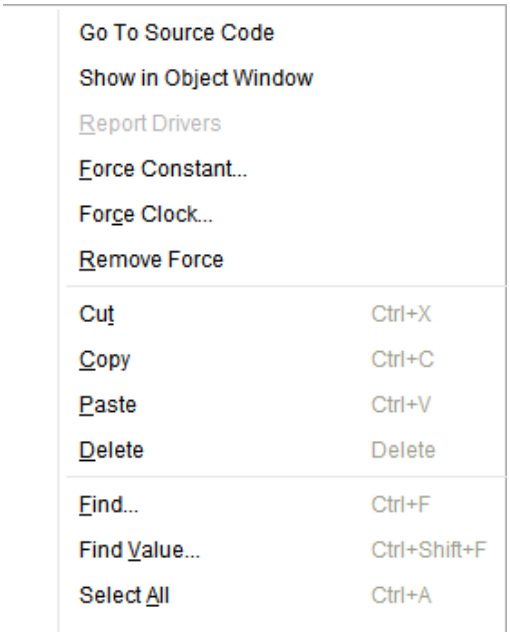
Force Constant

“Force Constant”（强制设为常量）选项允许您将信号固定为常量值，覆盖 HDL 代码中的赋值或者先前应用的其他常量或时钟强制设置。

“Force Constant”和“Force Clock”（强制设为时钟）均为“Objects”（对象）或波形窗口右键菜单（如下图所示）或文本编辑器（源代码）中的选项。

提示：双击“Objects”（对象）、“Sources”（源代码）或“Scope”（作用域）窗口中的某个项即可在文本编辑器中将其打开。如需了解有关文本编辑器的其他信息，请参阅《Vivado Design Suite 用户指南：使用 Vivado IDE》(UG893)。

图 39：**强制选项**

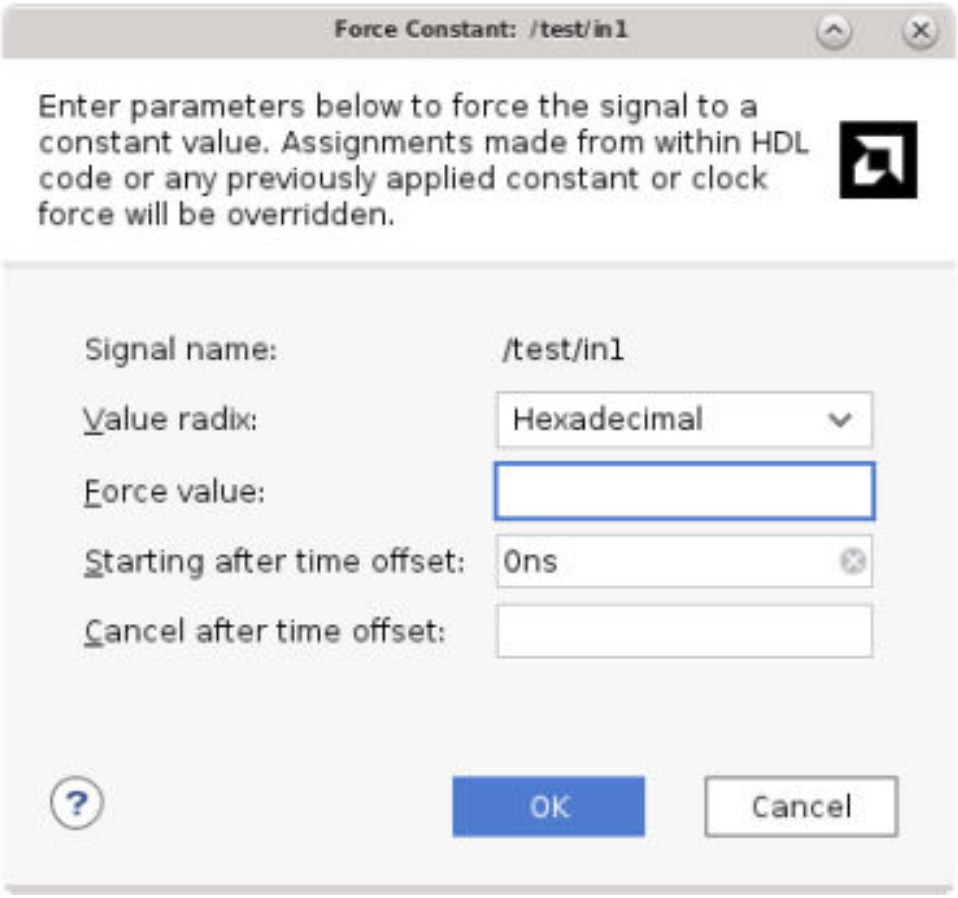


对于 Vivado 仿真器不支持强制设置的对象，禁用“Force”（强制）选项。对象类型或者 Vivado 仿真器的建模中针对这些对象的限制可能会导致不支持这些对象。

提示：要对禁用“Force”选项的模块或实体端口进行强制设置，请尝试对其上一个作用域层次的已连接的实际信号执行强制设置。使用 `add_force` Tcl 命令（例如，`add_force myObj 0`）可查看禁用这些选项的原因。

选中“Force Constant”选项时，会打开“Force Constant”对话框，以便您输入相关值，如下图所示。

图 40： **“Force Constant”对话框**



“Force Constant”选项描述如下：

- “Signal name”（信号名称）：显示默认信号名称，即所选对象的完整路径名称。
- “Value radix”（值基数）：显示所选信号的当前基数设置。您可选择任一受支持的基数类型：Binary（二进制）、Hexadecimal（十六进制）、Unsigned Decimal（无符号十进制）、Signed Decimal（有符号十进制）、Signed Magnitude（有符号量级）、Octal（八进制）和 ASCII。随后，GUI 将根据“Radix”（基数）设置来禁止输入相应的值。例如，如果选择 Binary，那么不允许输入除 0 和 1 以外的其他数字值。
- “Force value”（强制值）：使用定义的基数值来指定强制常量值。（如需了解有关基数的更多信息，请参阅 更改默认基数 和 使用模拟波形。）
- “Starting after time offset”（时间偏移后开始）：在指定时间过后开始。默认开始时间为 0。时间可采用字符串，例如，10 或 10 ns。输入不含单位的数字时，Vivado 仿真器会使用默认单位 (ns)。
- “Starting after time offset”（时间偏移后取消）：在指定时间过后取消。时间可采用字符串，例如，10 或 10 ns。如果输入不含单位的数字，则会使用默认仿真时间单位。

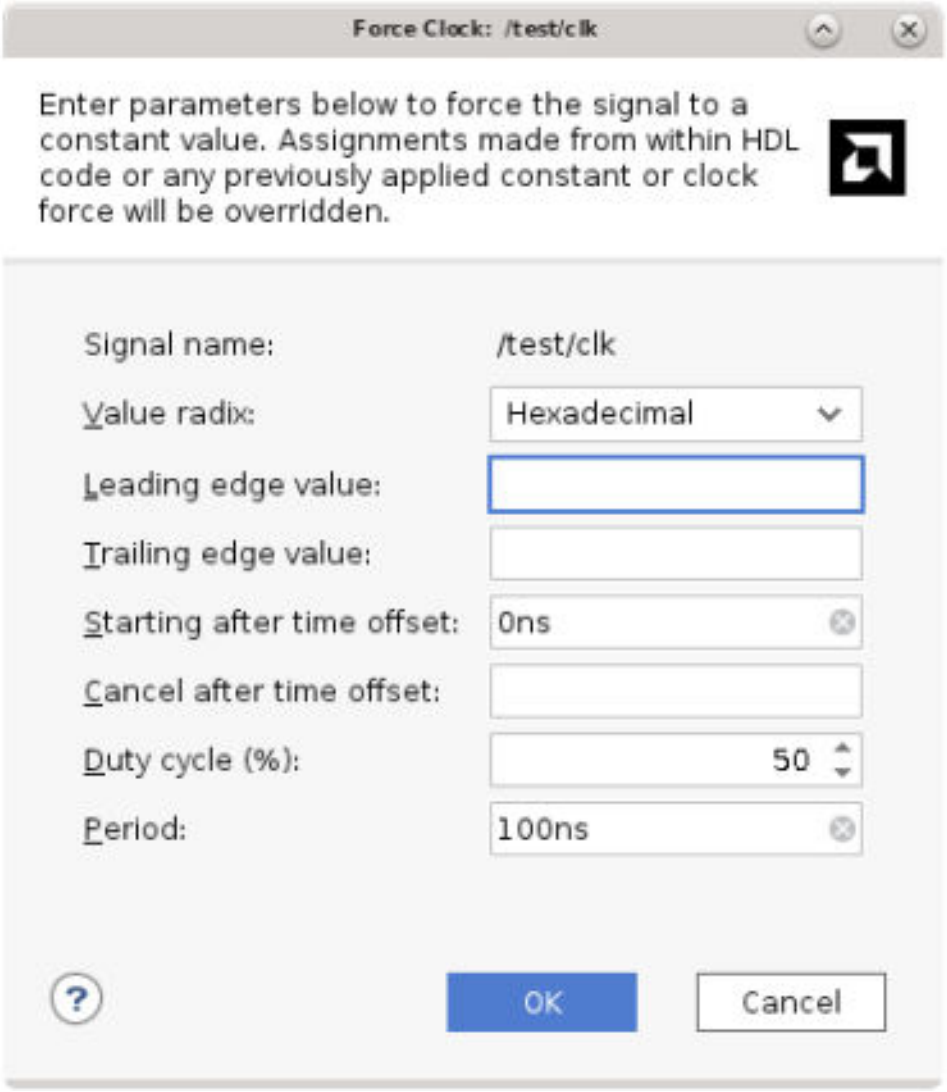
  Tcl 命令：

```
add_force /testbench/TENSOUT 1 200 -cancel_after 500
```

Force Clock

“Force Clock”（强制设为时钟）命令允许您为信号赋值，用于在指定时间长度范围内，以时钟信号的方式在两种状态之间切换指定速率。在“Objects”（对象）窗口菜单中选中“Force Clock”选项时，会打开“Force Clock”对话框，如下图所示。

图 41： **“Force Clock”对话框**



“Force Clock”对话框中的选项如下所示。

- “Signal name”（信号名称）：显示默认信号名称；即，“Objects”窗口或波形中所选项的完整路径名称。

  提示： “Force Clock”命令可应用于任意信号（不仅是时钟信号）以定义振荡值。

- “Value radix”（值基数）：显示所选信号的当前基数设置。从下拉菜单中选择显示的基数类型之一：“Binary”（二进制）、“Hexadecimal”（十六进制）、“Unsigned Decimal”（无符号十进制）、“Signed Decimal”（有符号十进制）、“Signed Magnitude”（有符号量级）、“Octal”（八进制）或“ASCII”。

- “Leading edge value”（前置边沿值）：指定首个时钟沿模式。前置边沿值使用“Value radix”字段中定义的基数。
- “Trailing edge value”（后置边沿值）：指定第二个时钟沿模式。后置边沿值使用“Value radix”字段中定义的基数。
- “Starting after time offset”（时间偏移后开始）：在当前仿真经过指定时间后开始执行 force 命令。默认开始时间为 0。时间可采用字符串，例如，10 或 10 ns。如果输入不含单位的数字，Vivado 仿真器会使用默认用户单位。
- “Cancel after time offset”（时间偏移后取消）：在当前仿真经过指定时间后取消执行 force 命令。时间可采用字符串，例如，10 或 10 ns。如果输入不含单位的数字，Vivado 仿真器会使用默认仿真时间单位。
- “Duty cycle (%)”（占空比）：指定时钟脉冲保持活动状态的时间占比。可接受的值范围为 0 至 100（默认值为 50%）。
- “Period”（周期）：指定时钟脉冲长度，定义为时间值。时间可采用字符串，例如，10 或 10 ns。

  注释：如需了解有关基数的更多信息，请参阅 更改默认基数 和 使用模拟波形。

Tcl 命令示例：

```
add_force /testbench/TENSOUT -radix bin {0} {1} -repeat_every 10ns -
cancel_after 3us
```

Remove Force

Remove Force

要从对象移除任意指定的 force，请使用以下 Tcl 命令：

```
remove_forces <force object>
remove_forces <HDL object>
```

在批处理模式下使用 Force

以下代码示例显示了如何使用 `add_force` 命令将信号强制设为指定值。其中提供了简单的 Verilog 电路。第一个示例显示了 `add_force` 命令的交互用法，第二个示例显示了脚本用法。

示例 1：添加 Force

以下代码片段是 Verilog 电路：

```
module bot(input in1, in2,output out1);
reg sel;
assign out1 = sel? in1: in2;
endmodule
module top;
reg in1, in2;
wire out1;
bot I1(in1, in2, out1);
initial
begin
    #10 in1 = 1'b1; in2 = 1'b0;
    #10 in1 = 1'b0; in2 = 1'b1;
end
initial
    $monitor("out1 = %b\n", out1);
endmodule
```

您可以调用以下命令来观察 `add_force` 的效果：

```
xelab -vlog tmp.v -debug all
xsim work.top
```

在命令提示符处输入：

```
add_force /top/I1/sel 1
run 10
add_force /top/I1/sel 0
run all
```

您可使用 `add_force` Tcl 命令来将信号、连线或寄存器强制设为指定值：

```
add_force [-radix <arg>] [-repeat_every <arg>] [-cancel_after <arg>] [-
quiet]
[-verbose] <hdl_object> <values>...
```

如需了解有关此命令及其他 Tcl 命令的更多信息，请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)。

示例 2：add_force 搭配 remove_forces 的脚本用法

以下 Verilog 文件示例 `top.v` 用于例化计数器。您可在以下命令示例中使用此文件。

```
module counter(input clk,reset,updown,output [4:0] out1);
reg [4:0] r1;
always@(posedge clk)
begin
    if(reset)
        r1 <= 0;
    else
        if(updown)
            r1 <= r1 + 1;
        else
            r1 <= r1 - 1;
end
assign out1 = r1;
endmodule
module top;
reg clk;
reg reset;
reg updown;
wire [4:0] out1;
counter I1(clk, reset, updown, out1);
initial
begin
    reset = 1;
    #20 reset = 0;
end
initial
begin
    updown = 1; clk = 0;
end
initial
    #500 $finish;
initial
    $monitor("out1 = %b\n", out1);
endmodule
```

命令示例

1. 使用以下命令创建名为 `add_force.tcl` 的文件：

```
create_project add_force -force
add_files top.v
set_property top top [get_filesets sim_1]
set_property -name xelab.more_options -value {-debug all} -objects
[get_filesets
sim_1]
set_property runtime {0} [get_filesets sim_1]
launch_simulation -simset sim_1 -mode behavioral
add_wave /top/*
```

2. 以 Tcl 模式调用 Vivado Design Suite 并使用 source 命令找到 `add_force.tcl` 文件。
3. 在 Tcl 控制台中，输入：

```
set force1 [add_force clk {0 1} {1 2} -repeat_every 3 -cancel_after 500]
set force2 [add_force updown {0 10} {1 20} -repeat_every 30]
run 100
```

   在“Wave”（波形）窗口中可以观察到 `out1` 值的递增和递减。您可在 Vivado IDE 中使用 `start_gui` 命令观察波形。

   在“Wave”窗口中观察 `updown` 信号的值。

4. 在 Tcl 控制台中，输入：

```
remove_forces $force2
run 100
```

   观察可见，仅有 `out1` 的值发生递增。

5. 在 Tcl 控制台中，输入：

```
remove_forces $force1
run 100
```

   观察可见，`out1` 的值并未更改，因为 `clk` 信号并未切换。

使用 Vivado 仿真器执行功耗分析

“切换活动交换格式”(Switching Activity Interchange Format, SAIF) 是 ASCII 报告，有助于提取和存储由仿真器工具生成的切换活动信息。此切换活动可以反标注解回 AMD 功耗分析与优化工具内，用于功耗测量和估算。

切换活动交换格式 (SAIF) 转储专为 AMD 功耗工具进行优化，可供 `report_power` Tcl 命令使用。Vivado 仿真器会将以下 HDL 类型写入 SAIF 文件。请参阅《Vivado Design Suite 用户指南：功耗分析与优化》(UG907) 中的 AMD 器件中的功耗以获取更多信息。

- Verilog：
  - 输入端口、输出端口和输入输出端口
  - 内部连线声明

- VHDL：
  - 类型为 `std_logic`、`std_ulogic` 和 `bit`（标量、矢量和阵列）的输入端口、输出端口和输入输出端口。

  注释：在 Vivado Design Suite 中并不会为时序仿真生成 VHDL 网表；因此，VHDL 源文件仅适用于 RTL 级代码，不适用于网表仿真。

对于 RTL 级仿真，仅生成块级端口，不生成内部信号。

如需了解有关使用第三方仿真工具进行功耗分析的信息，请参阅 第 3 章：使用第三方仿真器进行仿真 中的 转储 SAIF 用于功耗分析 和 在 VCS 中转储 SAIF。

生成 SAIF 转储

使用 `log_saif` 命令前，必须调用 `open_saif`。`log_saif` 命令不会返回任何对象或值。

1. 请使用 `-debug typical` 选项来编译您的 RTL 代码以启用 SAIF 转储：

```
xvlog -sv <fileName>.sv
xelab -debug typical top -s mysim
xsim mysim
```

2. 使用以下 Tcl 命令启动 SAIF 转储：

```
open_saif <saif_file_name>
```

3. 请输入以下任一 Tcl 命令添加要生成的作用域和信号：

```
log_saif [get_objects]
```

要以递归方式记录所有实例，请使用以下 Tcl 命令：

```
log_saif [get_objects -r *]
```

4. 运行仿真（使用任一 run 命令）。
5. 使用以下 Tcl 命令将仿真数据导入 SAIF 格式：

```
close_saif
```

SAIF Tcl 命令示例

您可记录 SAIF 用于：

- 作用域内的所有信号：`/tb: log_saif /tb/*`
- 作用域的所有端口：`/tb/UUT`
- 名称以 `a` 开头并以 `b` 结尾且中间包含数字的对象：

```
log_saif [get_objects -regexp {^a[0-9]+b$}]
```

- `current_scope` 和 `children_scope` 内的对象：

```
log_saif [get_objects -r *]
```

- `current_scope` 内的对象：

```
log_saif * or log_saif [get_objects]
```

- 如果仅用于作用域 `/tb/UUT` 的端口，请使用以下命令：

```
log_saif [get_objects -filter {type == in_port || type == out_port ||
type == inout_port || type == port } /tb/UUT/* ]
```

- 如果仅用于 `scope /tb/UUT` 的内部信号，请使用以下命令：

```
log_saif [get_objects -filter { type == signal } /tb/UUT/* ]
```

提示：此筛选适用于需要 HDL 对象的所有 Tcl 命令。

使用 Tcl 仿真批处理文件转储 SAIF

```
sim.tcl:
open_saif xsim_dump.saif
log_saif /tb/dut/*
run all
close_saif
quit
```

使用 report_drivers Tcl 命令

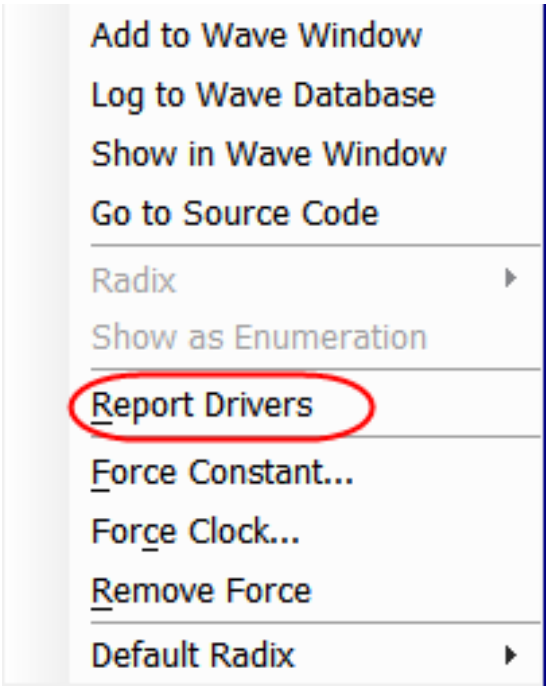
您可以使用 `report_drivers` Tcl 命令来判定哪个信号正在驱动 HDL 对象上的值。语法如下：

```
report_drivers <hdl_object>
```

该命令会将驱动程序（执行赋值操作的 HDL 语句）打印到 Tcl 控制台，并将当前驱动值置于提供给连线或信号类型的 HDL 对象的赋值的右侧。

您还可以从“Object”（对象）窗口或“Wave”（波形）窗口上下文菜单或文本编辑器调用 `report_drivers` 命令。要打开下图所示上下文菜单，请右键单击任意信号，然后单击“Report Drivers”（程序报告驱动）。结果会显示在 Tcl 控制台中。

图 42：**含“Report Drivers”命令选项的上下文菜单**



使用值更改转储功能

您可使用“Value Change Dump (VCD)”（值更改转储）文件来捕获仿真输出。Tcl 命令基于与转储值相关的 Verilog 系统任务。

对于 VCD 功能特性，下表中列出的 Tcl 命令会对这些 Verilog 系统任务进行建模。

表 13：**适用于 VCD 的 Tcl 命令**

Tcl 命令	描述
`open_vcd`	打开 VCD 文件，用于捕获仿真输出。此 Tcl 命令会对 `$dumpfile` Verilog 系统任务的行为进行建模。
`checkpoint_vcd`	对 `$dumpall` Verilog 系统任务的行为进行建模。
`start_vcd`	对 `$dumpon` Verilog 系统任务的行为进行建模。
`log_vcd`	为指定 HDL 对象记录 VCD。此命令会对 `$dumpvars` Verilog 系统任务的行为进行建模。
`flush_vcd`	对 `$dumpflush` Verilog 系统任务的行为进行建模。
`limit_vcd`	对 `$dumplimit` Verilog 系统任务的行为进行建模。
`stop_vcd`	对 `$dumpoff` Verilog 系统任务的行为进行建模。
`close_vcd`	关闭 VCD 生成。

请参阅《Vivado Design Suite Tcl 命令参考指南》(UG835)，或者在 Tcl 控制台中输入：

注释：这些命令对 RTL 设计和 SystemC 设计都适用。

```
<command> -help
```

示例：

```
open_vcd xsim_dump.vcd
log_vcd /tb/dut/*
log_vcd -include_systemc * /tb/dut/hier1/sc_object
log_vcd -ports * /tb/dut/HDL_signal
run all
close_vcd
quit
```

如需了解更多信息，请参阅 Verilog 语言支持例外。

您可使用 VCD 数据来确认仿真器输出，以调试仿真失败问题。

相关信息

Vivado 仿真器混合语言支持和语言例外

使用 log_wave Tcl 命令

`log_wave` 命令会记录仿真输出以便通过 Vivado 仿真器的波形查看器来查看指定 HDL 对象。不同于 `add_wave`，`log_wave` 命令不会将 HDL 对象添加到波形查看器（即波形配置）中。它仅启用将输出记录到 Vivado 仿真器波形数据库 (WDB) 的功能。

提示：要显示插入时间之前的对象值，必须重新启动仿真。为避免因未能捕获值的变化而不得不重新启动仿真，请在仿真运行开始时发出 log_wave -r / Tcl 命令，这样即可捕获设计中可显示的所有 HDL 对象的值变化。

语法：

```
log_wave [-recursive] [-r] [-quiet] [-verbose] <hdl_objects>...
```

log_wave Tcl 命令用法示例

记录波形输出的适用范围和相应的命令如下所述：

- 用于设计内的所有信号（不包括备选顶层模块的信号）：

  ```
  log_wave -r /
  ```

- 用于作用域 `/tb` 内的所有信号：

  ```
  log_wave /tb/*
  ```

- 用于名称以 a 开头并以 b 结尾且中间包含数字的对象：

  ```
  log_wave [get_objects -regexp {^a[0-9]+b$}]
  ```

- 用于当前作用域和所有子作用域内的所有对象（递归方式）：

```
log_wave -r *
```

- 用于暂时覆盖所有消息限制，并返回来自以下命令的所有消息：

```
log_wave -v
```

- 用于当前作用域的对象：

```
log_wave *
```

- 如果仅用于作用域 `/tb/UUT` 的端口，请使用以下命令：

```
log_wave [get_objects -filter {type == in_port || type == out_port ||
type ==
inout_port || type == port} /tb/UUT/*]
```

- 如果仅用于作用域 `/tb/UUT` 的内部信号，请使用以下命令：

```
log_wave [get_objects -filter {type == signal} /tb/UUT/*]
```

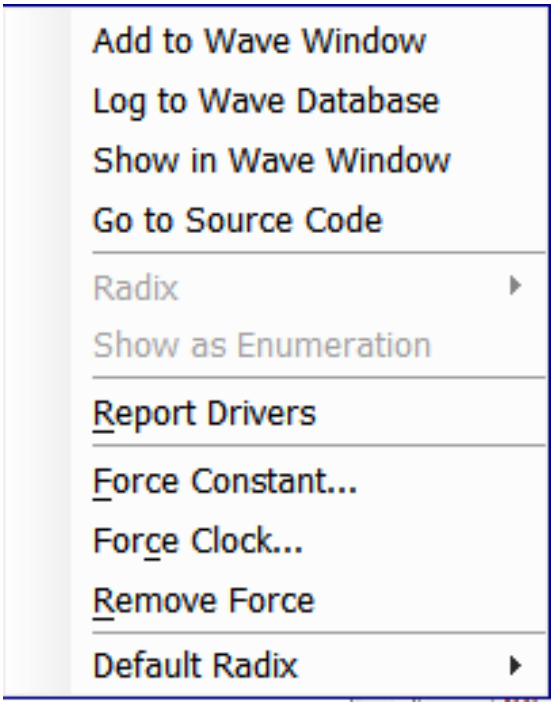
波形配置设置（包括信号顺序、名称样式、基数和颜色）按需保存到波形配置 (WCFG) 文件中。请参阅 第 5 章：使用 Vivado 仿真器对仿真波形进行分析。

在对象、波形和文本编辑器窗口中执行信号交叉探测

在 Vivado 仿真器中，您可对“Objects”（对象）窗口、“Wave”（波形）窗口和文本编辑器中存在的信号进行交叉探测。

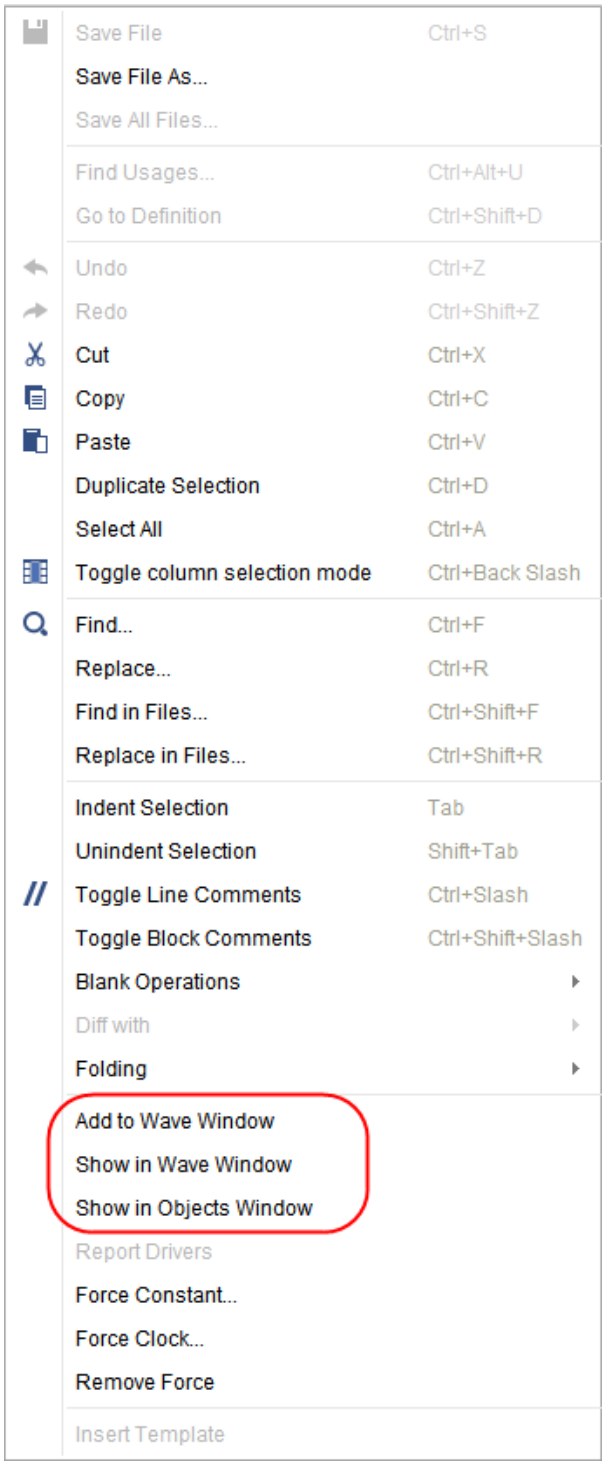
在“Objects”窗口中，您可检查“Wave”窗口中是否存在某个信号，反之亦然。右键单击此信号即可打开下图所示上下文菜单。单击“Show in Wave Window”（显示在波形窗口中），或者如果“Wave”窗口中尚未存在信号，则单击“Add to Wave Window”（添加到波形窗口）。

图 43：**“Objects”窗口的上下文菜单选项**



您还可从文本编辑器对信号进行交叉探测。右键单击信号即可打开如下图所示上下文菜单。选择“Add to Wave Window”、“Show in Waveform”或“Show in Objects”（在对象窗口中显示）。随后就会在“Wave”窗口或“Objects”窗口中高亮显示此信号。

图 44：**文本编辑器右键单击（上下文）菜单**



工具专用 init.tcl

执行仿真期间，Vivado 仿真器使用 source 命令查找存在于以下位置的 init 文件：

```
$HOME/.Xilinx/xsim/xsim_init.tcl
```

如果要跨多轮运行设置某个属性，那么此文件很有用。在此类情况下，您可将这些属性写入某个 Tcl 文件，Vivado 仿真器会在执行期间在时间 0' 之前使用 source 命令找到此 Tcl 文件。

子程序调用栈支持

现在，您可使用 `get_value/set_value` 选项来单步执行子程序调用并访问子程序内部的自动变量和静态变量。

当前，如果子程序位于调用栈顶层，那么您只能访问这些变量。

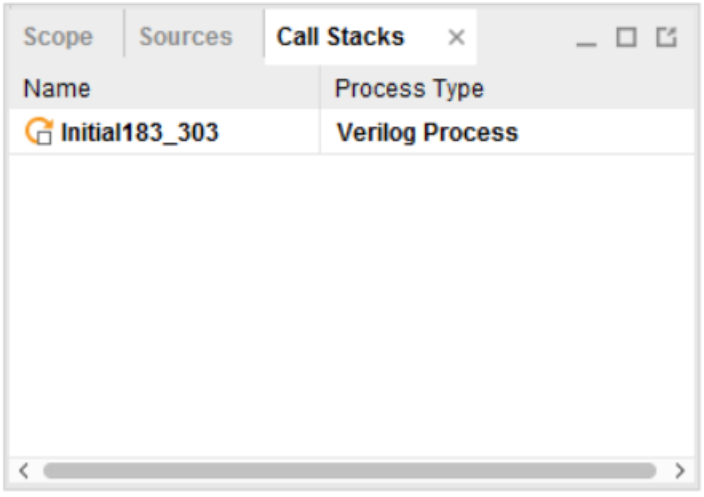
以下选项可用于支持访问位于调用栈的任意级别的变量。

“Call Stacks”窗口

如果设计中的 VHDL/Verilog 进程包含在某个子程序中，并且正在等待当前仿真时间，那么“Call Stacks”（调用栈）窗口可显示所有这些进程的 HDL 作用域。这与 `get_stacks` Tcl 命令类似。

默认情况下，如果当前进程内的仿真处于停止状态（包含在某个子程序内），就会在“Call Stacks”窗口中选中该进程。但您可以选择子程序内当前处于等待状态的任何其他进程。在“Call Stack”窗口上选中某个进程的效果与从“Scope”（作用域）窗口或者使用 `current_scope` Tcl 命令选中进程作用域的效果相同。在“Call Stacks”窗口上选中某个进程时，更新的进程会显示在“Scope”窗口、“Objects”（对象）窗口、“Stack Frames”（栈帧）窗口和“Locals”（本地）选项卡中。在“Call Stacks”窗口中会显示含有绝对路径的进程名称及选定进程的类型。

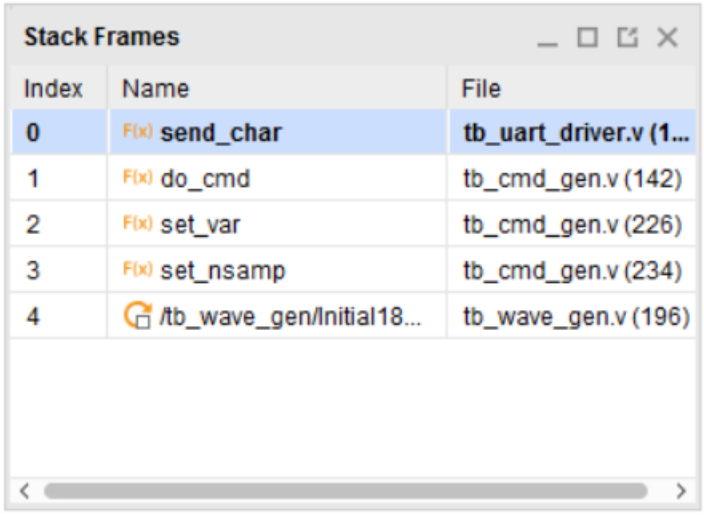
图 45：**“Call Stacks”窗口**



“Stack Frames”窗口

“Stack Frames”（栈帧）窗口会显示子程序内当前处于等待状态的 HDL 进程及其调用栈内的子程序。这与 `report_frames` 和 `current_frame` Tcl 命令类似。在“Stack Frames”窗口中，当前层级的最近的子程序显示在顶部，后接调用子程序。调用程序 HDL 进程显示在底部。您可选择其他帧作为当前帧，效果与 `current_frame -set <selected_frame_index>` Tcl 命令相同。“Objects”窗口中的“Locals”选项卡随所选子程序帧而变，它会显示所选子程序帧本地的静态变量和自动变量。在“Stack Frames”窗口中会显示所选 HDL 进程的帧数、子程序/进程名称、源文件和当前行。

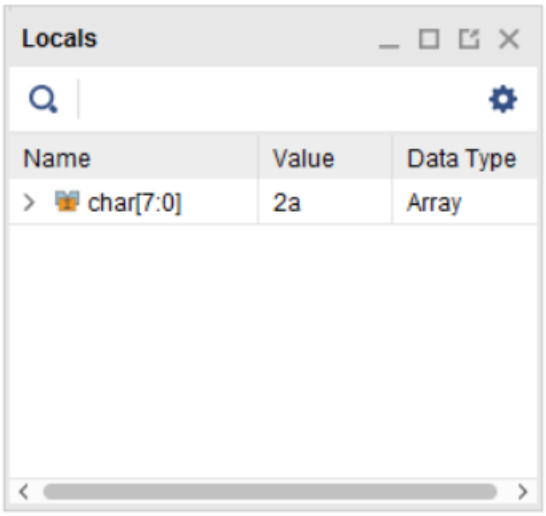
图 46：**“Stack Frames”窗口**



“Objects”窗口中的“Locals”选项卡

“Objects”窗口中的“Locals”选项卡会显示当前执行（或选中）的子程序本地的静态变量和自动变量的名称、值和类型。这与 `get_objects -local` Tcl 命令类似。此窗口随“Stack Frames”窗口中所选的帧而变。在“Locals”选项卡中会显示每个变量/实参的名称、值和类型。

图 47：**“Objects”窗口中的“Locals”选项卡**



调试动态类型

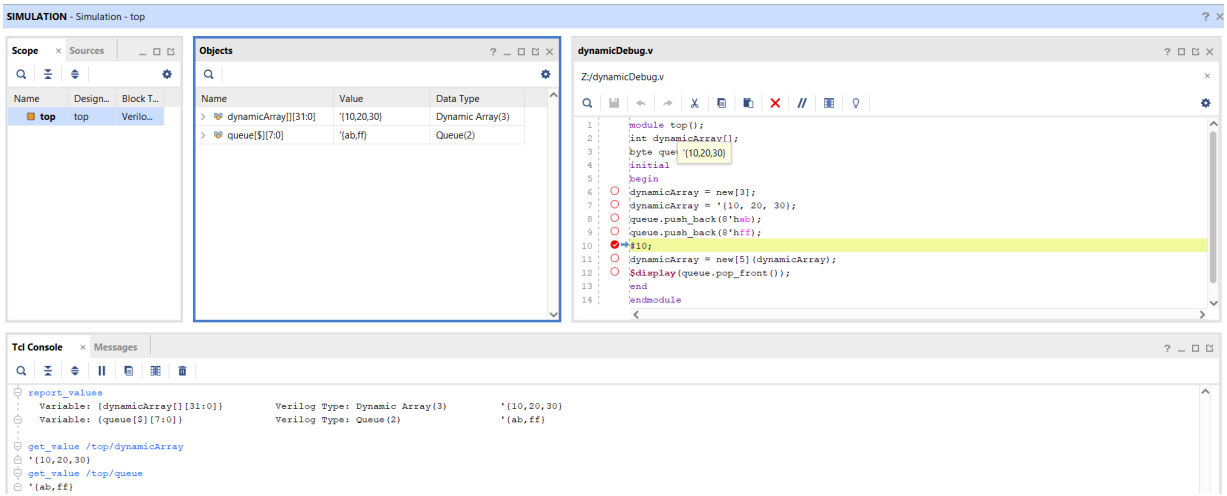
在 SystemVerilog 中存在 Class（类）、Dynamic Array（动态阵列）、Queue（队列）和 Associative Array（关联阵列）等动态类型。这些动态类型在 Vivado 仿真器中均受支持。Vivado 允许您探测动态类型变量。例如：

```
module top();
int dynamicArray[];
byte queue[$];
initial
begin
    dynamicArray = new[3];
    dynamicArray = '{10, 20, 30};
    queue.push_back(8'hab);
    queue.push_back(8'hff);
    #10;
    dynamicArray = new[5](dynamicArray);
    $display(queue.pop_front());
end
endmodule
```

您以使用以下窗口来探测动态类型变量，如下图所示：

- “Objects”窗口
- “Tcl Console”（Tcl 控制台）窗口，使用 `get_value` 和 `report_value` 命令即可。
- “Sources”窗口中的工具提示

图 48：**探测动态类型**



注释：不支持用于追踪波形 (`add_wave`) 或用于创建波形数据库 (`log_wave`) 的动态类型。

第 7 章

在 Vivado 仿真器中以批处理模式或脚本模式执行仿真

本章描述了命令行编译和仿真进程。

Vivado 支持集成仿真流程，其中该工具可以从 IDE 启动 Vivado 仿真器或第三方仿真器。但许多用户还想在其验证环境内以批处理模式或脚本模式来运行仿真，这可能包括系统级别仿真或高级验证（例如，UVM）。Vivado Design Suite 支持在 Vivado 仿真器中使用批处理仿真或脚本仿真。

本章描述了收集所需设计文件、为目标仿真器生成仿真脚本和在批处理模式下运行仿真的进程。您可为来自 Vivado IP integrator 的顶层 HDL 设计、分层模块、托管 IP 工程或块设计生成仿真脚本。在基于脚本的工程模式和非工程模式流程中均支持批处理仿真。

导出仿真文件和脚本

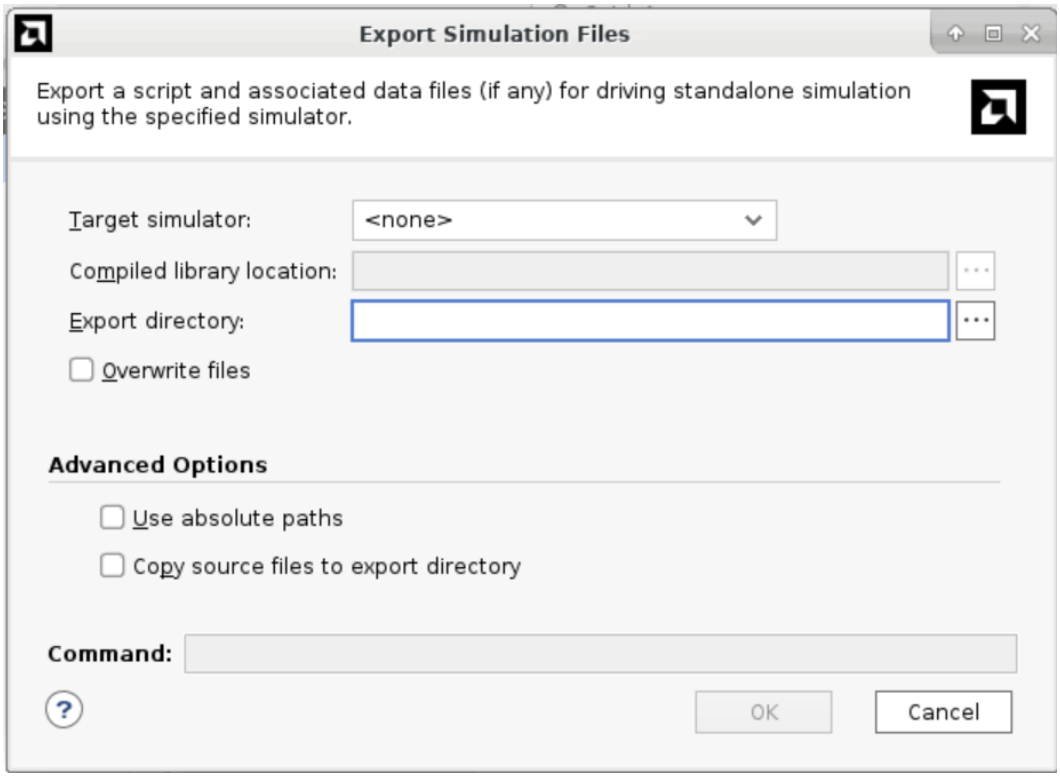
要从命令行运行仿真来执行行为仿真或时序仿真，请执行以下步骤：

1. 识别和解析设计文件。
2. 细化和生成设计的可执行仿真快照。
3. 使用可执行快照运行仿真。

Vivado Design Suite 提供“Export Simulation”（导出仿真）命令以支持您快速收集仿真所需设计文件，并为顶层 RTL 设计或子设计生成仿真脚本。`export_simulation` 命令会为所有受支持的第三方仿真器或您所选目标仿真器生成脚本。

在 Vivado IDE 内，使用“File”→“Export”→“Export Simulation”（文件 > 导出 > 导出仿真）命令打开“Export Simulation”对话框，如下图所示。

图 49：**“Export Simulation”对话框**



“Export Simulation”命令会为所有受支持的仿真器或者为指定目标仿真器写入仿真脚本文件。生成的脚本包含仿真器命令，用于设计的编译、细化和仿真。

“Export Simulation”对话框的功能特性包括：

- “Target simulator”（目标仿真器）：指定所有仿真器或者指定特定仿真器，以便为其生成命令行脚本。目标仿真器包括 Vivado 仿真器以及各种受支持的第三方仿真器。如需了解更多信息，请参阅 第 3 章：使用第三方仿真器进行仿真。

  注释：在 Windows 操作系统上，仅为 Windows 上运行的仿真器生成脚本。

  注释：在命令行模式下，等效的选项 -simulator 的值为“all”，表示为所有受支持的仿真器生成脚本。仅限命令行支持该选项，在 GUI 中则不予支持。

- “Compiled library location”（已编译的库位置）：要使用“Export Simulation”所生成的脚本来执行仿真，必须首先使用 compile_simlib Tcl 命令来编译您的仿真库。生成的脚本会自动包含来自已编译的库目录的目标仿真器所需的设置文件。如需了解更多信息，请参阅 编译仿真库。

  提示：建议每次运行“Export Simulation”时都提供指向“Compile library location”的路径。这样即可确保脚本始终指向正确的仿真库。

- “Export directory”（导出目录）：为“Export Simulation”所编写的脚本指定输出目录。默认情况下，仿真脚本会写入当前工程的本地目录。
- “Overwrite files”（覆盖文件）：覆盖 `export` 目录中已存在的同名文件。
- “Use absolute paths”（使用绝对路径）：默认情况下，生成的脚本中的源文件和目录路径会设为脚本中定义的引用目录的相对路径。使用此切换开关即可使脚本中的文件路径从相对路径变为绝对路径。
- “Copy source files to export directory”（将源文件复制到导出目录）：将设计文件复制到输出目录。这样会复制仿真源文件和生成的脚本，使整个仿真文件夹更便携。

- “Command”（命令）：该字段用于为 `export_simulation` 命令提供 Tcl 命令语法，您在“Export Simulation”对话框中指定各种选项和设置后就会运行该命令。
- “Help”（帮助）：如需获取“Export Simulation Files”（导出仿真文件）对话框内各种选项的详细信息，请单击此“Help”按钮。

“Export Simulation”（导出仿真）命令支持工程设计和非工程设计。它并不会从当前工程读取属性来查询 Verilog defines 和 includes 目录。而是改为通过“Export Simulation”命令从对话框或 `export_simulation` 命令选项获取指令。您必须指定适当的选项才能获取所需的结果。此外，您必须已完成为顶层设计内使用的所有 IP 和 BD 生成输出文件的操作。

重要提示！如果 IP 和 BD 输出文件不存在，`export_simulation` 命令就不会为其生成输出文件。而是改为返回错误并退出。

单击“Export Simulation”对话框上的“OK”（确定）时，此命令会获取对下列指定设计对象执行仿真所需的所有设计文件的仿真编译顺序，包括：顶层设计、分层模块、IP 核、来自 Vivado IP integrator 的块设计或具有多个 IP 的托管 IP 工程。使用目标仿真器的编译器命令和选项即可将所需设计文件的仿真编译顺序导出至 shell 脚本。

仿真脚本会写入“Export Simulation”对话框中指定的导出目录内的单独文件夹。对于每个指定的仿真器都会创建一个独立文件夹，并且会为仿真器写入 `compile`、`elaborate` 和 `simulate` 脚本。

“Export Simulation”命令生成的脚本使用 3 步进程，即分析/编译、细化和仿真，包括 Vivado 仿真器在内的诸多仿真器都适用此进程。但对于 ModelSim 和 Riviera 而言，生成的脚本使用该工具所需的 2 步进程：即编译和仿真。

提示：要在 Questa 仿真器内使用 2 步进程，可先从专为 ModelSim 生成的脚本开始操作，并按需对脚本进行更改。

此“Exporting Simulation”命令还会把任意已有数据文件从文件集或 IP 复制到指定导出目录。如果设计包含 Verilog 源文件，那么生成的脚本还会把 `glbl.v` 文件从 Vivado 软件安装路径复制到输出目录。

```
export_ip_user_files -no_script -force export_simulation -directory "C:/
Data/project_wave1" -simulator all
```

从对话框运行“Export Simulation”命令时，Vivado IDE 实际上会按顺序运行一连串命令来为导出的脚本定义基本目录（或位置）、导出 IP 用户文件，然后运行 `export_simulation` 命令。

Vivado IDE 命令会自动运行 `export_ip_user_files` 命令，这样可确保支持核容器与非核容器 IP 的仿真所需的所有文件以及块设计都可供使用。如需了解更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896) 中的相应内容。虽然使用“Export Simulation”对话框时会自动运行 `export_ip_user_files`，但您必须确保先手动运行该命令，然后才能运行 `export_simulation` 命令。

提示：Vivado IDE 自动运行 `export_ip_user_files` 时会指定 `-no_script` 选项。这是为了防止为顶层设计中使用的各 IP 和 BD 生成仿真脚本，否则可能显著增加命令的运行时间。但您可通过去除 `-no_script` 选项，或者在指定的对象 (`-of_objects`) 上运行 `export_ip_user_files`，来为各 IP 和 BD 生成这些仿真脚本。

`export_ip_user_files` 命令会为仿真与综合所需的 IP 和块设计设置用户文件环境。该命令会创建名为 `ip_user_files` 的文件夹，其中包含例化模板、搭配第三方综合工具使用的存根文件、封装文件、存储器初始化文件和仿真脚本。

`export_ip_user_files` 命令还会将工程内所有 IP 和块设计之间共享的静态仿真文件加以整合，并将其复制到 `ipstatic` 文件夹内。工程中的多个 IP 和 BD 之间共享的许多 IP 文件对于特定 IP 自定义并不会发生更改。这些静态文件会被复制到 `ipstatic` 目录中。针对仿真创建的脚本会根据需要引用此目录中的这些共享文件。IP 自定义专用的动态仿真文件会被复制到 IP 文件夹中。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896) 中的“认识 IP 用户文件”。

重要提示！ `export_simulation` 命令生成的脚本和文件指向 `ip_user_files` 目录中的文件。您必须先运行 `export_ip_user_files` 命令，然后再运行 `export_simulation`，否则可能发生仿真错误。

导出顶层设计

要为顶层 RTL 设计创建仿真脚本，请使用 `export_simulation` 并提供仿真文件集对象。在以下示例中，`sim_1` 是仿真文件集，导出仿真会为设计中的所有 RTL 实体、IP 和 BD 对象创建仿真脚本。

```
export_ip_user_files -no_script
export_simulation -of_objects [get_filesets sim_1] -directory C:/test_sim \
-simulator questa
```

从 AMD IP 目录和块设计导出 IP

要为 IP 或 Vivado IP integrator 块设计生成脚本，可在 IP 或块设计对象上直接运行以下命令：

```
export_ip_user_files -of_objects [get_ips blk_mem_gen_0] -no_script -force
export_simulation -simulator xcelium -directory ./export_script \
-of_objects [get_ips blk_mem_gen_0]
```

或者为设计中的所有 IP 和 BD 导出 ip_user_files：

```
export_ip_user_files -no_script -force
export_simulation -simulator xcelium -directory ./export_script
```

您也可以为块设计对象生成仿真脚本：

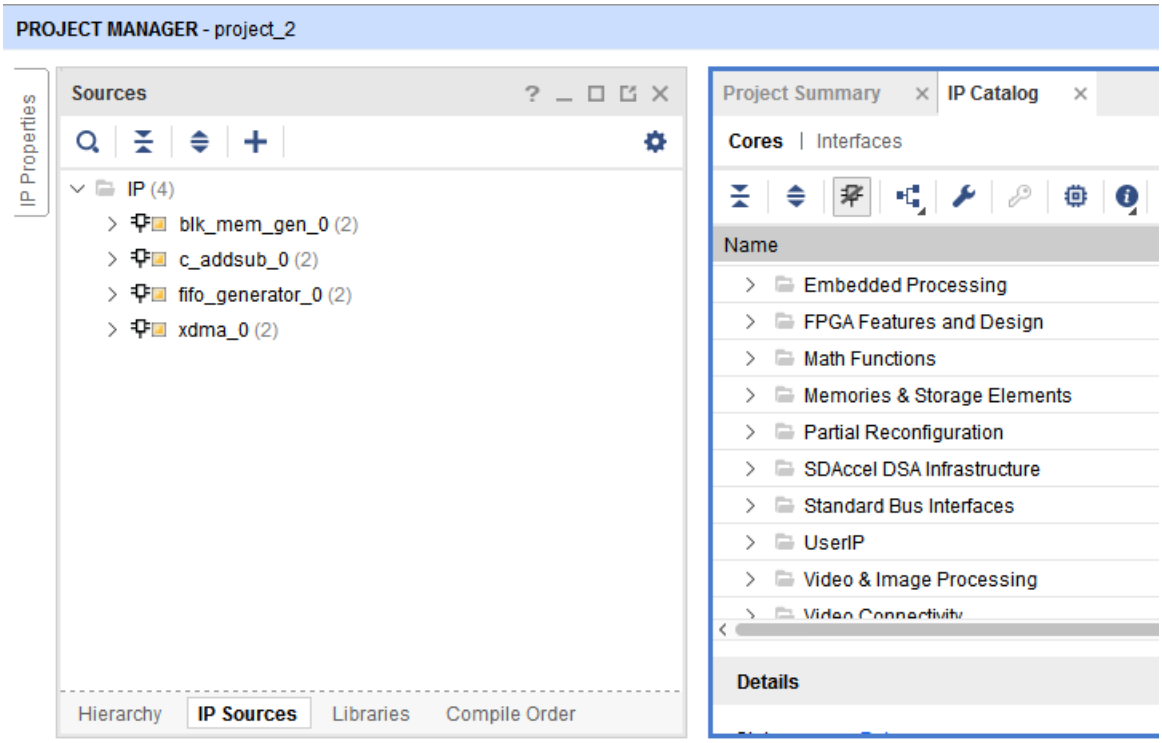
```
export_ip_user_files -of_objects [get_files base_microblaze_design.bd] \
-no_script -force
export_simulation -of_objects [get_files base_microblaze_design.bd] \
-directory ./sim_scripts
```

重要提示！ 您必须已完成为顶层设计内使用的所有 IP 和 BD 生成输出文件的操作。如果 IP 和 BD 输出文件不存在，`export_simulation` 命令就不会为其生成输出文件。而是改为返回错误并退出。

导出 Manage IP 工程

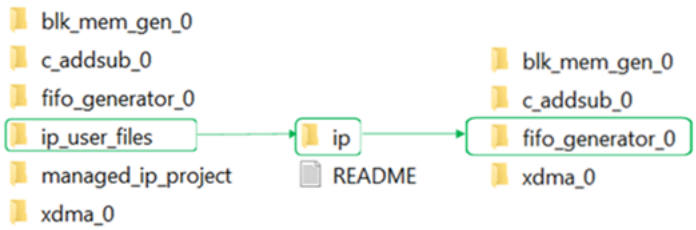
“Manage IP”（管理 IP）工程为用户提供了创建和管理自定义 IP 的集中存储库的功能。如需了解有关“Manage IP”工程的更多信息，请访问此链接以参阅《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896) 中的相应内容。为“Manage IP”工程生成 IP 输出文件时，Vivado 工具还会使用 `export_ip_user_files` 命令为每个 IP 生成仿真脚本，如前文所述。

图 50：**“Managed IP”工程**



如上所示“Managed IP”（托管 IP）工程特有 4 个不同的自定义 IP：`blk_mem_gen_0`、`c_addsub_0`、`fifo_generator_0` 和 `xdma_0`。对于此工程，Vivado Design Suite 会创建 `ip_user_files` 文件夹，如下图所示。

图 51：**托管 IP 目录结构**



`ip_user_files` 文件夹是由 `export_ip_user_files` 命令创建的，如前所述。在“Manage IP”工程上运行此命令时，它会以递归方式处理工程中的所有 IP，并生成 IP 综合与仿真所需的脚本和其他文件。`ip_user_files` 文件夹包含用于批处理仿真的脚本以及支持仿真所需的动态和静态 IP 文件。

您的目标仿真器或所有受支持的仿真器的仿真脚本都位于 `./sim_scripts` 文件夹内，如 导出 Manage IP 工程 中所示。您可以转至目标仿真器的文件夹，并将 `compile`、`elaborate` 和 `simulate` 脚本整合到仿真流程中。

Vivado 工具会将供设计内的多个 IP 和 BD 使用的所有共享仿真文件整合到名为 `./ipstatic` 的文件夹内。动态文件因 IP 自定义的规格而异，这些文件位于 `./ip` 文件夹内。

提示：除了导出“Manage IP”工程内的所有 IP 外，您还可使用 从 AMD IP 目录和块设计导出 IP 中所述步骤来为工程中的个别 IP 生成脚本。

在批处理模式下运行 Vivado 仿真器

为了在批处理模式或脚本模式下运行，Vivado 仿真器需依靠由 `export_simulation` 命令生成的文件所支持的 3 个进程。

- 解析设计文件 xvhdl 和 xvlog
- 细化和生成设计快照 xelab
- 设计快照 xsim 仿真

对于时序仿真，需要执行额外步骤和提供额外数据才能完整仿真，如下文所述：

- 生成时序网表
- 运行综合后和实现后仿真

解析设计文件 xvhdl 和 xvlog

`xvhdl` 命令和 `xvlog` 命令分别用于解析 VHDL 文件和 Verilog 文件。如需获取每个选项的描述，请参阅 表 15：xelab、xvhd 和 xvlog 命令选项。

xvhdl

`xvhdl` 命令是 VHDL 分析器（解析器）。

xvhdl 语法

```
xvhdl
[-encryptdumps]
[-f [-file] <filename>]
[-h [-help]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-nolog]
[-prj <filename>]
[-relax]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=<library_dir>]
[-incr]
[-2008]
[-93_mode]
[-nosignalhandlers]
```

此命令用于解析 VHDL 源文件，并将解析后的转储存储到硬盘上的 HDL 库中。

xvhdl 示例

```
xvhdl file1.vhd file2.vhd
xvhdl -work worklib file1.vhd file2.vhd
xvhdl -prj files.prj
```

xvlog

xvlog 命令是 Verilog 解析器。xvlog 命令用于解析 Verilog 源文件，并将解析后的转储存储到硬盘上的 HDL 库中。

xvlog 语法

```
xvlog
[-d [define] <name>[=<val>]]
[-encryptdumps]
[-f [-file] <filename>]
[-h [-help]]
[-i [include] <directory_name>]
[-initfile <init_filename>]
[-L [-lib] <library_name> [=<library_dir>]]
[-log <filename>]
[-nolog]
[-noname_unamed_generate]
[-relax]
[-prj <filename>]
[-sourcelibdir <sourcelib_dirname>]
[-sourcelibext <file_extension>]
[-sourcelibfile <filename>]
[-sv]
[-v [verbose] [0|1|2]]
[-version]
[-work <library_name> [=<library_dir>]
[-incr]
[-nosignalhandlers]
[-uvm_version arg]
```

xvlog 示例

```
xvlog file1.v file2.v
xvlog -work worklib file1.v file2.v
xvlog -prj files.prj
```

注释： xelab、xvlog 和 xvhdl 都不是 Tcl 命令。xvlog、xvhdl 和 xelab 均为 Vivado 无关的编译器可执行文件。因此，并没有对应的 Tcl 命令。

仿真启动与 Vivado 有关，因此通过 xsim Tcl 命令来完成。

为了在 Vivado 外使用仿真，专门提供了与 xsim 同名的可执行文件。xsim 可执行文件会在无工程模式下启动 Vivado，并执行 xsim Tcl 命令以启动仿真。因此，要在 Vivado IDE 中获取有关 xvlog、xvhdl 和 xelab 形式的帮助，请在这些命令前添加 exec。

示例：exec xvlog -help。

要获取有关 xsim 的帮助，请使用 xsim -help。

细化和生成设计快照 xelab

使用 Vivado 仿真器进行仿真分为两个阶段：

- 在第一阶段，仿真器编译器 xelab 会将您的 HDL 模型编译为快照，即以仿真器可执行的形式来表示该模型。

- 在第二阶段，仿真器会使用 `xsim` 命令加载并执行此快照以进行模型仿真。在非工程模式下，您可跳过第一阶段并重复第二阶段来复用此快照。

当仿真器创建快照时，它会基于模型中顶层模块的名称来给快照分配名称。但您可通过将快照名称指定为编译器的选项的方式来覆盖默认名称。在目录或 SIMSET 中快照名称必须唯一；复用快照名称（无论是默认名称还是自定义名称）都会导致覆盖先前构建的含此名称的快照。

重要提示！在同一目录或 SIMSET 内无法以相同快照名称运行两次仿真。

xelab

用于给定顶层单元的 `xelab` 命令可执行下列操作：

- 使用语言绑定规则或 `-L <library>` 命令行指定的 HDL 库加载子设计单元
- 执行设计的静态细化（设置参数、泛型、使生成语言生效等）
- 生成可执行代码
- 将生成的可执行代码与仿真内核库相链接，以创建可执行仿真快照

随后，可使用产生的可执行仿真快照名称作为 `xsim` 命令的选项，搭配其他选项来影响 HDL 仿真。

提示：`xelab` 可隐式调用解析命令 `xvlog` 和 `xvhdl`。您可使用 `xelab -prj` 选项整合解析步骤。如需了解有关工程文件的更多信息，请参阅 工程文件 (.prj) 语法。

注释：`xelab`、`xvlog` 和 `xvhdl` 不是 Tcl 命令，而是独立于 Vivado 的编译器可执行文件。因此，并没有对应的 Tcl 命令。

xelab 命令语法选项

在以下代码块中提供了每个选项的描述。

```
xelab [-d [define] <name>[=<val>] [-debug <kind>] [-f [-file] <filename>] [-
generic_top <value>] [-h [-help] [-i [include] <directory_name>] [-initfile
<init_filename>] [-log <filename>] [-L [-lib] <library_name>
[=<library_dir>] [-maxdesigndepth arg] [-mindelay] [-typdelay] [-
maxarraysize arg] [-maxdelay] [-mt arg] [-nolog] [-noname_unnamed_generate]
[-notimingchecks] [-nosdfinterconnectdelays] [-nospecify] [-O arg] [-
override_timeunit] [-override_timeprecision] [-prj <filename>] [-pulse_e
arg] [-pulse_r arg] [-pulse_int_e arg] [-pulse_int_r arg] [-pulse_e_style
arg] [-r [-run]] [-R [-runall]] [-rangecheck] [-relax] [-s [-snapshot] arg]
[-sdfnowarn] [-sdfnoerror] [-sdfroot <root_path>] [-sdfmin arg] [-sdftyp
arg] [-sdfmax arg] [-sourcelibdir <sourcelib_dirname>] [-sourcelibext
<file_extension>] [-sourcelibfile <filename>] [-stats] [-timescale] [-
timeprecision_vhdl arg] [-transport_int_delays] [-v [verbose] [0|1|2]] [-
version] [-sv_root arg] [-sv_lib arg] [-sv_liblist arg] [-dpiheader arg] [-
driver_display_limit arg] [-dpi_absolute] [-incr] [-93_mode] [-
nosignalhandlers] [-dpi_stacksize arg] [-transform_timing_checkers] [-a[ --
standalone] [-ignore_assertions] [-ignore_coverage] [-cov_db_dir arg] [-
cov_db_name arg] [-uvm_version arg] [-report_assertion_pass] [-
dup_entity_as_module] [-cc_celldefines] [-cc_libs] [-cc_type arg] [-
cov_db_dir arg] [-cov_db_name arg] [-ignore_localparam_override] [-sc_lib
arg] [-sc_root arg]
```

xelab 示例

```
xelab work.top1 work.top2 -s cpusim
xelab lib1.top1 lib2.top2 -s fftsim
xelab work.top1 work.top2 -prj files.prj -s pciesim
xelab lib1.top1 lib2.top2 -prj files.prj -s ethernetsim
```

Verilog 搜索顺序

`xelab` 命令使用以下搜索顺序来搜索和绑定例化的 Verilog 设计单元：

1. 由 Verilog 代码中的 `'uselib` 指定的库。例如：

   ```
   module
   full_adder(c_in, c_out, a, b, sum)
   input c_in,a,b;
   output c_out,sum;
   wire carry1,carry2,sum1;
   `uselib lib = adder_lib
   half_adder adder1(.a(a),.b(b),.c(carry1),.s(sum1));
   half_adder adder1(.a(sum1),.b(c_in),.c(carry2),.s(sum));
   c_out = carry1 | carry2;
   endmodule
   ```

2. 命令行上使用 `-lib|-L` 选项指定的库。
3. 父设计单元的库。
4. `work` 库。

Verilog 例化单元

当 Verilog 设计例化组件时，`xelab` 命令会将组件名称作为 Verilog 单元来处理，并按用户指定的顺序在用户指定的统一逻辑库列表中搜索 Verilog 模块。

- 如果找到该模块，`xelab` 会绑定此单元，并停止搜索。
- 如果区分大小写搜索不成功，那么 `xelab` 会执行不区分大小写的搜索，按统一逻辑库的顺序在用户指定的列表中搜索构造为扩展标识符的 VHDL 设计单元名称、选中第一个匹配的名称，然后停止搜索。
- 如果 `xelab` 为任一库找到唯一绑定，那么它会选中该名称并停止搜索。

  注释：对于混合语言设计，如果所使用的端口名称与 Verilog 模块例化的 VHDL 实体存在命名关联，那么此类端口名称始终作为不区分大小写来处理。此外，无法使用 `defparam` 语句来修改 VHDL 泛型。如需了解更多信息，请参阅 使用混合语言仿真。

  重要提示！不支持将整个 VHDL 记录对象连接到 Verilog 对象。

VHDL 例化单元

当 VHDL 设计例化组件时，`xelab` 命令会将组件名称作为 VHDL 单元来处理，并在 `work` 逻辑库中搜索此单元。

- 如果找到 VHDL 单元，`xelab` 命令会将其绑定并停止搜索。
- 如果 `xelab` 未找到 VHDL 单元，那么它会将大小写保留不变的组件名称作为 Verilog 模块名称来处理，并继续在用户指定的统一逻辑库列表中按指定顺序执行区分大小写的搜索。此命令会选中首个匹配的名称，然后停止搜索。

- 如果区分大小写搜索不成功，那么 `xelab` 会在用户指定的统一逻辑库中按指定顺序对 Verilog 模块执行不区分大小写的搜索。如果为任一库找到唯一绑定，则停止搜索。

`uselib Verilog 指令

支持 Verilog `uselib 指令，该指令用于设置库的搜索顺序。

`uselib 语法

```
<uselib compiler directive> ::= `uselib [<Verilog-XL uselib directives>|
<lib
directive>]
<Verilog-XL uselib directives> :== dir = <library_directory> | file =
<library_file>
| libext = <file_extension>
<lib directive>::= <library reference> {<library reference>}
<library reference> ::= lib = <logical library name>
```

`uselib 库语义

`uselib lib 指令无法搭配 Verilog-XL `uselib 指令一起使用。例如，下列代码违规：

```
`uselib dir=./ file=f.v lib=newlib
```

在一条 `uselib 指令内可指定多个库。

指定库的顺序决定了搜索顺序。例如：

```
`uselib lib=mylib lib=yourlib
```

指定搜索已例化的模块的操作首先在 `mylib` 内执行，而后在 `yourlib` 内执行。

正如 `uselib dir、`uselib file 和 `uselib libext 指令，在任一给定的解析和分析调用中，`uselib lib 指令跨 HDL 文件保持有效，就像解析调用保持有效一样。除非遇到另一条 `uselib 指令，否则 HDL 源代码中的 `uselib（包括任何 Verilog XL `uselib）指令保持有效。不含任何实参的 `uselib 会移除当前处于活动状态的任意 `uselib <lib|file|dir|libext> 的效果。

以下模块搜索机制用于通过 Verific Verilog 细化算法来解析已例化的模块或 UDP：

- 首先，按当前处于活动状态的 `uselib lib（如有）的逻辑库的排序列表来搜索已例化的模块。
- 如未找到任何结果，请按在 `xelab` 命令行中搜索库的相同方式，在库的有序列表内搜索已例化的模块。
- 如未找到任何结果，请在父模块的库中搜索已例化的模块。例如，如果库 `work` 的模块 A 例化了库 `mylib` 的模块 B，而 B 例化了模块 C，那么请在模块 B 的 `/mylib` 库（C 的父级库）中搜索模块 C。
- 如未找到任何结果，请在 `work` 库中搜索已例化的模块，此库为下列库之一：
  - HDL 源文件编译到的库
  - 显式设置为 `work` 库的库
  - 默认 work 库名称为 `work`

`uselib 示例

表 14：**`uselib 示例**

已编译到名为 adder_lib 的逻辑库中的 half_adder.v 文件	已编译到名为 Work 的逻辑库中的 full_adder.v 文件

`module half_adder(a,b,c,s);`
`input a,b;`
`output c,s;`
`s = a ^ b;`
`c = a & b;`
`endmodule`
	`module`
	`full_adder(c_in, c_out, a, b, sum)`
	`input c_in,a,b;`
	`output c_out,sum;`
	`wire carry1,carry2,sum1;`
	`` `uselib lib = adder_lib ``
	`half_adder`
	`adder1(.a(a),.b(b),. c(carry1),.s(sum1)); half_adder`
	`adder1(.a(sum1),.b(c_in),.c (carry2),.s(sum)); c_out =`
	`carry1 | carry2; endmodule`

xelab、xvhdl 和 xvlog xsim 命令选项

下表列出了 `xelab`、`xvhdl` 和 `xvlog` xsim 命令的命令选项。

表 15：**xelab、xvhd 和 xvlog 命令选项**

命令选项	描述	适用命令
`-d [define] <name>[=<val>]`	定义 Verilog 宏。针对每个 Verilog 宏使用 `-d|--define`。宏的格式为 `<name>[=<val>]`，其中，`<name>` 是宏名称，`<value>` 是宏的值（可选）。	xelab 解析设计文件 xvhdl 和 xvlog
`-debug <kind>`	开启指定调试功能的前提下执行编译。`<kind>` 选项包括： · `typical`：最常用的功能，包括：`line` 和 `wave`。 · `line`：HDL 断点。 · `wave`：波形生成、条件执行、强制值。 · `xlibs`：可查看 AMD 预编译库。该选项仅在命令行上可用。 · `off`：关闭所有调试功能（默认）。 · `all`：使用所有调试选项。	xelab
`-encryptdumps`	对当前正在编译的设计单元的已解析转储进行加密。	解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-f [-file] <filename>`	从指定文件读取其他选项。	xelab xsim 可执行文件选项 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-generic_top <value>`	以指定值覆盖顶层设计单元的泛型或参数。例如，`-generic_top "P1=10"`	xelab

表 15：**xelab、xvhd 和 xvlog 命令选项** (续)

命令选项	描述	适用命令
`-h [-help]`	打印此帮助消息。	xelab xsim 可执行文件选项 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-i [include] <directory_name>`	为使用 Verilog `` `include `` 包含的文件指定要搜索的目录。针对每个指定的搜索目录使用 `-i|--include`。	xelab 解析设计文件 xvhdl 和 xvlog
`-initfile <init_filename>`	用户定义的仿真器初始化文件，用于添加到或者覆盖默认 `xsim.ini` 文件所提供的设置。	xelab 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-L [-lib] <library_name> [=<library_dir>]`	为已例化的非 VHDL 设计单元指定搜索库。例如，Verilog 设计单元。 针对每个搜索库使用 `-L|--lib`。该实参的格式为 `<name>[=<dir>]`，其中，`<name>` 是库的逻辑名称，`<library_dir>` 是库的物理目录（可选）。	xelab 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-log <filename>`	指定 log 日志文件名。默认值：`<xvlog|xvhdl|xelab|xsim>.log`。	xelab xsim 可执行文件选项 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-maxarraysize <arg>`	将 VHDL 阵列大小最大值设为 2**n（默认值：n = 28，即 2**28）。	xelab
`-maxdelay`	以最大延迟来编译 Verilog 设计单元。	xelab
`-maxdesigndepth <arg>`	覆盖细化器允许的设计层级最大深度（默认值：5000）。	xelab
`-maxlogsize <arg>`	设置 log 日志文件可达到的最大大小 (MB)。默认设置为无限制。	xsim 可执行文件选项
`-mindelay`	以最小延迟来编译 Verilog 设计单元。	xelab
`-mt <arg>`	指定可并行运行的子编译作业数量。可能的值包括 `auto`、`off` 或大于 1 的 `integer`。 如果指定 `auto`，那么 `xelab` 会基于主机上的 CPU 数量来选择并行作业数量。（默认值：`auto`。） 高级用法：要进一步控制 `-mt` 选项，可按如下方式设置 Tcl 属性： `set_property XELAB.MT_LEVEL off|N [get_filesets sim_1]`	xelab
`-nolog`	禁止生成 log 日志文件。	xelab xsim 可执行文件语法 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-noieeewarnings`	禁用来自 VHDL IEEE 函数的警告。	xelab

表 15：**xelab、xvhd 和 xvlog 命令选项** (续)

命令选项	描述	适用命令
-noname_unnamed_generate	不给未命名的生成块生成名称。	xelab 解析设计文件 xvhdl 和 xvlog
-notimingchecks	忽略 Verilog 指定块中的时序检查构造。	xelab
-nosdfinterconnectdelays	忽略 SDF 端口和 SDF 中的互连延迟构造。	xelab
-nospecify	忽略 Verilog 路径延迟和时序检查。	xelab
-O <arg>	启用或禁用优化。 • -O 0 = 禁用优化 • -O 1 = 启用基本优化 • -O 2 = 启用最常用的优化（默认） • -O 3 = 启用高级优化 **注释：**值越低，编译速度越快，但代价是仿真越慢：值越高，编译越慢，但仿真运行速度更快。	xelab
-override_timeunit	使用 -timescale 选项中指定的时间单位覆盖所有 Verilog 模块的时间单位。	xelab
-override_timeprecision	使用 -timescale 选项中指定的时间精度覆盖所有 Verilog 模块的时间精度。	xelab
-pulse_e <arg>	路径脉冲错误限制占路径延迟的百分比值。允许的值为 0 到 100（默认值为 100）。	xelab
-pulse_r <arg>	路径脉冲拒绝限制占路径延迟的百分比值。允许的值为 0 到 100（默认值为 100）。	xelab
-pulse_int_e arg	互连脉冲拒绝限制占延迟的百分比值。允许的值为 0 到 100（默认值为 100）。	xelab
-pulse_int_r <arg>	互连脉冲拒绝限制占延迟的百分比值。允许的值为 0 到 100（默认值为 100）。	xelab
-pulse_e_style <arg>	指定出现脉冲比模块路径延迟短的错误时，何时应对错误进行处理。选项包括： • ondetect：检测到违例后立即报告错误。 • onevent：在模块路径延迟过后报告错误。 默认值：onevent	xelab
-prj <filename>	指定包含一个或多个 vhdl|verilog <work lib> <HDL file name> 条目的 Vivado 仿真器工程文件。	xelab 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
-r [-run]	以命令行交互模式来运行生成的可执行快照。	xelab
-rangecheck	为 VHDL 启用运行时值范围检查。	xelab
-R [-runall]	运行生成的可执行快照直至仿真结束。	xelab xsim 可执行文件语法

表 15：**xelab、xvhd 和 xvlog 命令选项** (续)

命令选项	描述	适用命令
`-relax`	放宽严格的语言规则。	xelab 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-s [-snapshot] <arg>`	指定输出仿真快照的名称。默认值为 `<worklib>.<unit>`；例如：`work.top`。其他单元名称使用 # 来串联；例如：`work.t1#work.t2`。	xelab
`-sdfnowarn`	不发出 SDF 警告。	xelab
`-sdfnoerror`	将 SDF 文件中发现的错误作为警告来处理。	xelab
`-sdfmin <arg>`	`<root>=<file>` SDF 以最小延迟来给位于 `<root>` 的 `<file>` 添加注解。	xelab
`-sdftyp <arg>`	`<root>=<file>` SDF 以典型延迟来给位于 `<root>` 的 `<file>` 添加注解。	xelab
`-sdfmax <arg>`	`<root>=<file>` SDF 以最大延迟来给位于 `<root>` 的 `<file>` 添加注解。	xelab
`-sdfroot <root_path>`	应用 SDF 注解的默认设计层级。	xelab
`-sourcelibdir <sourcelib_dirname>`	未编译模块的 Verilog 源文件所在的目录。 为每个源目录使用 `-sourcelibdir <sourcelib_dirname>`。	xelab 解析设计文件 xvhdl 和 xvlog
`-sourcelibext <file_extension>`	未编译模块的 Verilog 源文件的文件扩展名。 为源文件扩展名使用 `-sourcelibext <file extension>`。	xelab 解析设计文件 xvhdl 和 xvlog
`-sourcelibfile <filename>`	未编译模块的 Verilog 源文件的文件名。	xelab 解析设计文件 xvhdl 和 xvlog
`-stat`	打印工具 CPU 和存储器使用情况以及设计统计数据。	xelab
`-sv`	以 SystemVerilog 模式编译输入文件。	解析设计文件 xvhdl 和 xvlog
`-timescale`	为 Verilog 模块指定默认时间刻度。默认值：`1ns/1ps`。	xelab
`-timeprecision_vhdl <arg>`	为 VHDL 设计指定时间精度。默认值：`1ps`。	xelab
`-transport_int_delays`	为互连延迟使用传输模型。	xelab
`-typdelay`	以典型延迟来编译 Verilog 设计单元（默认）。	xelab
`-v [verbose] [0|1|2]`	指定打印消息的详细程度。默认值为 0。	xelab 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-version`	在屏幕上打印编译器版本。	xelab xsim 可执行文件选项 解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog

表 15：**xelab、xvhd 和 xvlog 命令选项** (续)

命令选项	描述	适用命令
`-work <library_name> [=<library_dir>]`	指定工作库。该实参的格式为：`<name>[=<dir>]`，其中： · `<name>` 是库的逻辑名称。 · `<library_dir>` 是库的物理目录（可选）。	解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog
`-sv_root <arg>`	要在其中查找 DPI 库的根目录。 默认值：`<current_directory/xsim.dir/xsc>`	xelab
`--sc_lib arg`	SystemC 函数共享库的名称；(.dll/.so)，不含文件扩展名	xelab
`--sc_root <arg>`	要在其中查找 SystemC 库的根目录。默认值：`<current_directory>/xsim.dir/work/xsc`	xelab
`-sv_lib <arg>`	DPI 导入的函数共享库的名称；(.dll/.so)，不含文件扩展名。	xelab
`-sv_liblist <arg>`	指向 DPI 共享库的启动文件。	xelab
`-dpiheader <arg>`	导出和导入函数的头文件名。	xelab
`-driver_display_limit <arg>`	为具有最大大小的信号启用驱动程序调试（默认值：n = 65536）。	xelab
`-dpi_absolute`	在 Linux 上为 DPI 库使用绝对路径代替 `LD_LIBRARY_PATH`，此类库的格式为 `lib<libname>.so`。	xelab
`-incr`	在仿真中启用增量分析/细化。	解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog xelab
`-93_mode`	以纯 93 模式编译 VHDL。	解析设计文件 xvhdl 和 xvlog xelab
`-2008`	以 2008 模式编译 VHDL。	解析设计文件 xvhdl 和 xvlog
`-nosignalhandlers`	不允许编译器捕获防病毒软件、防火墙信号。	解析设计文件 xvhdl 和 xvlog 解析设计文件 xvhdl 和 xvlog xelab
`-dpi_stacksize <arg>`	用户为 DPI 任务定义的栈大小。	xelab
`-transform_timing_checkers`	将时序检查器变换为 Verilog 进程。	xelab
`-a`	生成独立非交互仿真可执行文件，用于执行 run-all。 始终搭配 `-R` 一起使用。 要在不含任何调试功能的情况下加速运行仿真，请将 `-standalone` 与 `-R` 搭配使用。它会单独调用仿真，而不调用 Vivado IDE。该选项能节省许可证加载时间。	xelab
`-ignore_assertions`	忽略 SystemVerilog 并发断言有效。	xelab
`-ignore_coverage`	忽略 SystemVerilog 功能覆盖。	xelab
`-cov_db_dir <arg>`	功能覆盖数据库转储目录。覆盖数据存在于 `<arg>/xsim.covdb/<cov_db_name>` 目录下。默认设为 `./`。	xelab
`-cov_db_name <arg>`	功能覆盖数据库名称。覆盖数据存在于 `<cov_db_dir>/xsim.covdb/<arg>` 目录下。默认值为快照名称。	xelab

表 15：**xelab、xvhd 和 xvlog 命令选项** (续)

命令选项	描述	适用命令
`-uvm_version <arg>`	指定 UVM 版本（默认值 1.2）。	解析设计文件 xvhdl 和 xvlogxelab
`-report_assertion_pass`	报告 SystemVerilog 并发断言成功，即使不存在任何成功操作块也是如此。	xelab
`-dup_entity_as_module`	在混合语言设计的 Verilog 层级内启用对分层引用的支持。 **注意！**这可能导致编译速度显著下降。	xelab
`-cc_celldefines`	指定是否需要为已设置单元定义属性的库/模块捕获代码覆盖信息。默认设为 OFF（关）。	xelab
`-cc_libs`	指定是否需要为指定的所有库捕获代码覆盖信息。默认设为 OFF（关）。	xelab
`-cc_type arg`	指定用于生成代码覆盖统计数据 -bcesfxt 的选项。支持：(s)Statement Coverage（语句覆盖）、(b)Branch Coverage（分支覆盖）和 (c)Condition Coverage（条件覆盖）。	xelab
`-ignore_localparam_override`	忽略 localparam override	xelab

设计快照 xsim 仿真

`xsim` 命令会加载仿真快照以应用批处理模式仿真，或者提供工作空间 (GUI) 和/或基于 Tcl 的交互式仿真环境。

xsim 可执行文件语法

此命令语法如下：

```
xsim <options> <snapshot>
```

其中：

- `xsim` 是命令。
- `<options>` 是下表中指定的选项。
- `<snapshot>` 是仿真快照。

xsim 可执行文件选项

表 16：**xsim 可执行文件命令选项**

xsim 选项	描述
`-f [-file] <filename>`	从文件加载命令行选项。
`-g [-gui]`	使用交互式工作空间运行。
`-h [-help]`	在屏幕上打印帮助消息。
`-log <filename>`	指定 log 日志文件名。

表 16：**xsim 可执行文件命令选项** (续)

xsim 选项	描述
-maxdeltaid arg (=-1)	指定最大增量数值。如果该数值超过最大同时仿真循环数，则报告错误。
-maxlogsize arg (=-1)	设置 log 日志文件可达到的最大大小 (MB)。默认设置为无限制。
-ieeewarnings	启用来自 VHDL IEEE 函数的警告。
-nolog	禁止生成 log 日志文件。
-nosignalhandlers	禁用在仿真中安装操作系统级别的信号处理程序。出于性能原因，对于某些可能会生成致命的操作系统级运行时错误的状况（例如，整数除以 0），仿真器并不会进行显式检查。仿真器会改为通过安装信号处理程序来捕获这些错误并生成报告。 禁用信号处理程序后，仿真器即可在存在此类安全软件的前提下运行，但操作系统级别的致命错误可能导致仿真突然崩溃，且几乎不提供任何有关故障性质的指示信息。 **注意！**仅限在您的安全软件阻止仿真器成功运行时，才能使用该选项。
-onfinish <quit|stop>	指定仿真结束时的行为。
-onerror <quit|stop>	指定出现仿真运行时错误时的行为。
-R [-runall]	运行仿真直至结束（例如，do 'run all;quit'）。
-stats	退出时显示存储器和 CPU 统计数据。
-testplusarg <arg>	指定 plusargs 以供 $test$plusargs 和 $value$plusargs 系统功能使用。
-t [-tclbatch] <filename>	指定 Tcl 文件，以供在批处理模式下执行。
-tp	启用如下功能：在屏幕上打印当前执行的进程的分层名称。
-tl	启用如下功能：在屏幕上打印当前执行的语句的文件名和行号。
-wdb <filename.wdb>	指定波形数据库输出文件。
-version	在屏幕上打印编译器版本。
-view <wavefile.wcfg>	打开波形配置文件。将此开关与 -gui 开关搭配使用。
-protoinst	指定 .protoinst 文件用于执行协议分析。
-sv_seed	SystemVerilog 约束随机种子。
-cov_db_dir	功能覆盖数据库转储目录。覆盖数据存在于 <arg>/xsim.covdb/<cov_db_name> 目录下。默认是 ./ 或者继承 xelab 中所设的值。
-cov_db_name	功能覆盖数据库名称。覆盖数据存在于 <cov_db_dir>/xsim.covdb/<arg> 目录下。默认是快照名称或者继承 xelab 中所设的值。
-downgrade_error2info	将 HDL 消息的严重性级别从 Error（错误）降级至 Info（参考）。
-downgrade_error2warning	将 HDL 消息的严重性级别从 Error（错误）降级至 Warning（警告）。
-downgrade_fatal2info	将 HDL 消息的严重性级别从 Fatal（致命）降级至 Info（参考）。
-downgrade_fatal2warning	将 HDL 消息的严重性级别从 Fatal（致命）降级至 Warning（警告）。
-downgrade_severity	将 HDL 消息的严重性级别降级。选项如下： · error2warning · error2info · fatal2warning · fatal2info
-ignore_assertions	忽略 SystemVerilog 并发断言有效。
-ignore_coverage	忽略 SystemVerilog 功能覆盖。
-ignore_feature	忽略特定 HDL 特性或构造的影响。选项如下： · assertion · coverage

表 16：**xsim 可执行文件命令选项** (续)

xsim 选项	描述
-tempDir	指定临时目录名称。
-autoloadwcfg	加载已保存的波形配置文件。
-clear_child_on_exit	退出仿真时清除子进程（仅针对 Linux）
-disable_multi_driver_resolution_scheduling	在调度进程期间还原到最初的多驱动程序解决办法流程
-xsimdir	xsim.dir 目录的位置。默认值为“.”（当前运行路径）

提示：在批处理文件或脚本中运行 xelab、xsc、xsim、xvhdl、xcrg 或 xvlog 命令时，可能还需要将 XILINX_VIVADO 环境变量定义为指向 Vivado Design Suite 的安装层级。要设置 XILINX_VIVADO 变量，请将以下某一命令添加到您的脚本或批处理文件中：

- Windows 上：set XILINX_VIVADO=<vivado_install_area>/Vivado/<version>
- Linux 上：setenv XILINX_VIVADO vivado_install_area>/Vivado/<version>
- 其中 <version> 是您当前使用的 Vivado 工具的版本：2014.3、2014.4、2015.1 等

代码覆盖率支持

代码覆盖率是衡量测试激励文件是否有效执行 RTL 代码的指标。启用代码覆盖率后，仿真器就会自动抽取该指标。AMD Vivado™ 仿真器当前支持 4 种类型的代码覆盖率：行、分支、条件和翻转。当您为上述任一代码覆盖率类型启用代码覆盖率指标时，该工具就会自动生成代码覆盖数据库。AMD Vivado™ 仿真器提供了名为赛灵思覆盖率报告生成器 (Xilinx Coverage Report Generator, XCRG) 的独立可执行文件用于查看设计的覆盖率，此文件可读取覆盖率数据库以生成覆盖率报告。

本仿真器支持的代码覆盖率功能包括：

- 行/语句覆盖率，包括语句的精确执行计数
- 对应 if-else、if-elseif-else、switch case 和三元运算符的分支覆盖率
- 检测并高亮代码覆盖率报告中缺失的 else 和缺失的 default
- 条件覆盖率，包括对条件进行检查并求值得到 TRUE/FALSE 的精确次数
- packed/unpacked reg、bit、logic、wire 数据类型
- int、shorting、integer、byte 数据类型和非动态结构体成员
- 使用 XCRG 生成代码覆盖率 HTML 报告
- 设计的代码覆盖率的仪表板视图
- 整个设计的文件、模块和分层实例列表
- 文件专用的代码覆盖率视图、模块和实例专用的视图
- 将使用 XCRG 运行多次所得的行/语句、分支、条件和翻转覆盖率加以合并

注释：当前，Vivado 仿真器仅支持将这些功能特性应用于 SystemVerilog 和 Verilog 代码。尚未支持 VHDL。

表 17：**XCRG 命令选项和描述**

XCRG 选项	描述
-db_name arg	xsim.covdb 内的数据库名称。如不指定，则使用目录中存在的所有数据库。
-dir arg	指向 xsim.covdb 数据库目录所在位置的路径。默认设为 ./xsim.covdb。
-file arg	该选项所指定的文件中包含了覆盖数据库要复原到的位置。
-h	打印帮助消息并退出。
-help	打印帮助消息并退出。
-merge_db_name arg	合并的数据库的名称。默认值为 xcrg_mdb。
-merge_dir arg	合并的数据库保存到的目录。默认设为 ./xsim.covdb。
-nolog	禁止生成 log 日志文件。
-report_dir arg	覆盖数据库和报告保存到的目录。该选项为必需选项。
-report_format arg	指定期望的覆盖报告格式，值包括：HTML、text 或 all。默认设置为 HTML。
-log arg	指定保存的 log 日志的文件名。默认设为 xcrg.log。
-version	打印 XCRG 的版本并退出。
-cov_db_name <arg>	指定用于保存代码覆盖数据库的数据库名称（快照名称）。代码覆盖数据库可从以下位置复原：<cov_db_dir value>/xsim.codeCov/<cov_db_name value>。
-cov_db_dir <arg>	指定保存代码覆盖信息数据库的目录。代码覆盖数据库可从以下位置复原：<cov_db_dir value>/xsim.codeCov/<cov_db_name value>。默认值为 ./xsim.CodeCov/。
-cc_fullfile	在代码覆盖报告中显示完整文件。默认情况下，对于超过 50000 行的文件，该选项处于关闭状态，仅显示文件的模块内容。
-report_dir <arg>	保存代码覆盖 HTML 报告的目录。默认值为 xcrg_code_cov_report。
-cc_instancescount <arg>	指定代码覆盖报告中显示的实例的最大数量。默认值为 100。

XCRG 示例

```
# Functional Coverage with one DUT and one TB generating html and text
reports xelab -svlog DUT1.v -svlog TB1.v -cov_db_dir ./fRun1 -cov_db_name
DB1 -R xcrg -dir ./fRun1/ -db_name DB1 -report_dir ./fReport1 -
report_format html firefox ./fReport1/dashboard.html & xcrg -dir ./fRun1/ -
db_name DB1 -report_dir ./fReport1 -report_format text gvim ./fReport1/
xcrg_report.txt # Merging Functional Coverage runs of one DUT and 2 TBs
generating merged html report (using TB from previous example) xelab -svlog
DUT1.v -svlog TB2.v -cov_db_dir ./fRun2 -cov_db_name DB2 -R xcrg -dir ./
fRun1 -db_name DB1 -dir ./fRun2 -db_name DB2 -merge_dir ./fMerge1 -
merge_db_name mDB1 -report_dir ./mfReport1 firefox ./mfReport1/
dashboard.html & # Code Coverage run and html report generation xelab -
svlog ccDUT.v -svlog ccTB.v -cc_type sbct -cov_db_name DB1 -cov_db_dir ./
cRun1 -R xcrg -cov_db_name DB1 -cov_db_dir ./cRun1 -cc_report ./cReport1
firefox ./cReport1/dashboard.html & # Other examples xcrg -h xcrg -file /
path/to/file xcrg -file /path/to/file -db_name work.top xcrg -dir /
path/to/abc xcrg -dir ./abc -report_dir def -report_format html xcrg -
dir ./abc -db_name work.top -report_dir def -report_format html xcrg -dir /
path/to/abc -db_name work.top -report_dir def -report_format text xcrg -
merge_dir m xcrg -merge_db_name xyz -report_dir def xcrg -report_format
html -nolog xcrg -report_format html -log xcrgOutput.log xcrg -cov_db_name
a1 -cov_db_dir ./ xcrg -report_dir abc -cov_db_name work.testbench -
cov_db_dir ./xsim.codeCov/
```

在独立模式下运行 Vivado 仿真器的示例

在独立模式下运行 Vivado 仿真器时，可以执行命令来：

- 分析设计文件
- 细化设计和创建快照
- 打开 Vivado 仿真器工作空间和波形配置文件并运行仿真

步骤 1：分析设计文件

首先按类型分析 HDL 源文件，如下表所示。每条命令均可包含多个文件。

表 18：**用于设计文件分析的文件类型和关联命令**

文件类型	命令
Verilog	`xvlog <VerilogFileName(s)>`
SystemVerilog	`xvlog -sv <SystemVerlilogFileName(s)>`
VHDL	`xvhdl <VhdlFileName(s)>`

步骤 2：细化和创建快照

完成分析后，使用 `xelab` 命令细化设计并创建快照用于仿真：

```
xelab <topDesignUnitName> -debug typical
```

重要提示！您可使用 `xelab` 提供多个顶层设计单元名称。要将 Vivado 仿真器工作空间用于 `launch_simulation` 期间的类似用途，必须将调试级别设为 `typical`。

步骤 3：运行仿真

成功完成 `xelab` 阶段后，Vivado 仿真器会创建快照用于运行仿真。

要调用 Vivado 仿真器工作空间，请使用以下命令：

```
xsim <SnapShotName> -gui
```

要打开波形配置文件，请执行以下操作：

```
xsim <SnapShotName> -view <wcfg FileName> -gui
```

您可使用多个 `-view` 标志打开多个 `wcfg` 文件。例如：

```
xsim <SnapShotName> -view <wcfg FileName> -view <wcfg FileName>
```

工程文件 (.prj) 语法

注释：此处讨论的工程文件是基于 Vivado 仿真器文本的工程文件。它不同于 Vivado Design Suite 创建的工程文件 (`.xpr`)。

要使用工程文件解析设计文件，请创建名为 `<proj_name>.prj` 的文本文件，并在工程文件内使用如下所示语法。

```
verilog <work_library> <file_names>... [-d <macro>]...[-i
<include_path>]...
vhdl <work_library> <file_name>
sv <work_library> <file_name>
vhdl2008 <work_library> <file_name>
```

其中：

`<work_library>`：表示给定行上的 HDL 文件将编译到其中的库。

`<file_names>`：表示 Verilog 源文件。每行可指定多个 Verilog 文件。

`<file_name>`：表示 VHDL 源文件；每行仅限指定一个 VHDL 文件。

1. 对于 Verilog 或 SystemVerilog：`[-d <macro>]` 提供了定义一个或多个宏的选项。
2. 对于 Verilog 或 SystemVerilog：`[-i <include_path>]` 提供了定义一个或多个 `<include_path>` 目录的选项。

预定义的宏

`XILINX_SIMULATOR` 是 Verilog 预定义宏。该宏的值为 1。预定义的宏可以执行工具特有的功能，或者识别要在设计流程中使用的工具。用法示例如下：

```
`ifdef VCS
     // VCS specific code
`endif
`ifdef INCA
     // NCSIM specific code
`endif
`ifdef MODEL_TECH
     // MODELSIM specific code
`endif
`ifdef XILINX_ISIM
     // ISE Simulator (ISim) specific code
`endif
`ifdef XILINX_SIMULATOR
     // Vivado Simulator (XSim) specific code
`endif
`ifdef _VCP
//Aldec specific code
`endif
```

库映射文件 (xsim.ini)

HDL 编译程序 xvhdl、xvlog 和 xelab 使用 xsim.ini 配置文件来查找 VHDL 和 Verilog 逻辑库的定义和物理位置。

编译器会尝试按如下顺序从这些位置中读取 xsim.ini：

1. 当前工作目录中的 xsim.ini。
2. 通过 -initfile 开关指定的用户文件。如未指定 -initfile，那么程序会在当前工作目录中搜索 xsim.ini。
3. <Vivado_Install_Dir>/data/xsim

xsim.ini 文件语法如下：

```
<logical_library1> = <physical_dir_path1>
<logical_library2> = <physical_dir_path2>
```

xsim.ini 文件示例如下：

```
std=<Vivado_Install_Area>/xsim/vhdl/std
ieee=<Vivado_Install_Area>/xsim/vhdl/ieee
vl=<Vivado_Install_Area>/xsim/vhdl/vl
ieee_proposed=$RDI_DATADIR/xsim/vhdl/ieee_proposed
synopsys=<Vivado_Install_Area>/xsim/vhdl/synopsys
uvm=<Vivado_Install_Area>/xsim/system_verilog/uvm
unisim=<Vivado_Install_Area>/xsim/vhdl/unisim
unimacro=<Vivado_Install_Area>/xsim/vhdl/unimacro
unifast=<Vivado_Install_Area>/xsim/vhdl/unifast
simprims_ver=<Vivado_Install_Area>/xsim/verilog/simprims_ver
unisims_ver=<Vivado_Install_Area>/xsim/verilog/unisims_ver
unimacro_ver=<Vivado_Install_Area>/xsim/verilog/unimacro_ver
unifast_ver=<Vivado_Install_Area>/xsim/verilog/unifast_ver
secureip=<Vivado_Install_Area>/xsim/verilog/secureip
work=./work
```

xsim.ini 文件具有以下功能特性和限制：

- 在 xsim.ini 文件内最多仅限包含一条库路径。
- 如不存在对应于物理路径的目录，那么在编译器首先尝试写入该路径时，xvhd 或 xvlog 会创建此目录。
- 您可从环境变量方面来描述物理路径。此环境变量必须以 $ 字符开头。
- 逻辑库的默认物理目录是 xsim/<language>/<logical_library_name>，逻辑库名称示例如下：

  ```
  <Vivado_Install_Area>/xsim/vhdl/unisim
  ```

- 文件注释必须以 -- 开头。

  注释：从 2018.2 版本起，AMD 提供了两个 init 文件，分别名为 xsim.ini 和 xsim_legacy.ini。xsim_legacy.ini 文件类似于旧版本的 xsim.ini。其中包含 UNISIM 库的映射，而新的 xsim.ini 文件则包含 UNISIM 库的所有文件的映射以及预编译的 IP 映射。

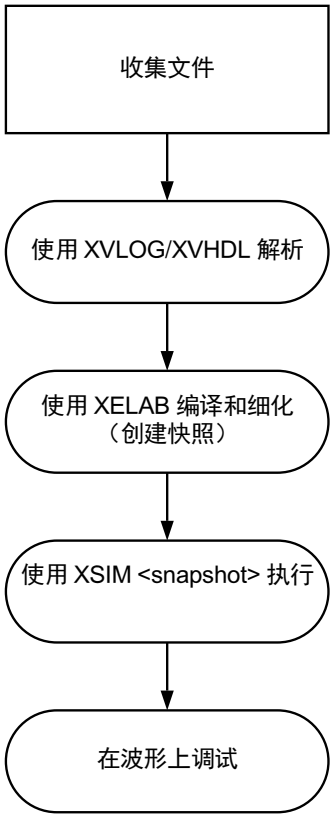
运行仿真模式

您可从命令行运行任意仿真模式。以下各小节演示并描述了从命令行运行时的仿真模式。

行为仿真

下图显示了行为仿真进程：

图 52：**行为仿真进程**

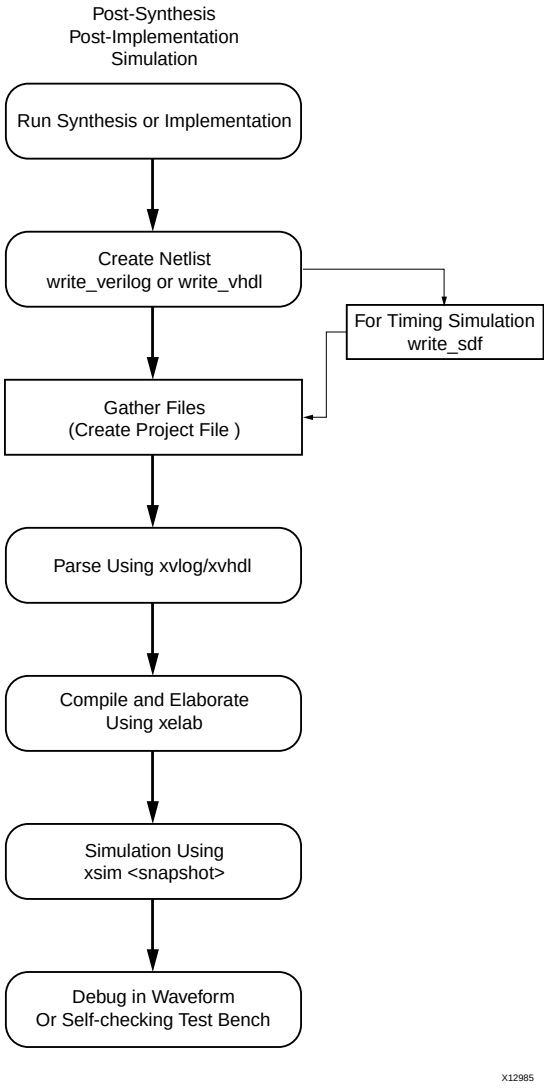


要从 Vivado Design Suite 内运行行为仿真，请使用 Tcl 命令：`launch_simulation -mode behavioral`。

运行综合后和实现后仿真

在综合后和实现后阶段，您可运行功能仿真或 Verilog 时序仿真。下图显示了综合后和实现后仿真进程：

图 53：**综合后和实现后仿真**



以下是从命令行运行综合后功能仿真的示例：

```
synth_design -top top -part xc7k70tfbg676-2
open_run synth_1 -name netlist_1
write_verilog -mode funcsim test_synth.v
launch_simulation -mode post-synthesis
```

提示：运行综合后或实现后时序仿真时，必须先运行 `write_verilog` 命令，然后再运行 `write_sdf` 命令，并且需要提供相应的注解命令用于细化和仿真。

使用 Tcl 命令和脚本

您可在 Tcl 控制台上单独运行各条 Tcl 命令，或者也可以将命令批量导入 Tcl 脚本以供运行仿真。

使用 -tclbatch 文件

您可在 Tcl 文件中输入仿真命令，并使用以下命令来引用此 Tcl 文件：`-tclbatch <filename>`

`-tclbatch` 选项可用于在文件内包含命令，并在开始仿真时执行这些命令。例如，您可在名为 `run.tcl` 的文件内包含：

```
run 20ns
```

```
current_time quit
```

然后启动仿真，如下所示：

```
xsim <snapshot> -tclbatch run.tcl
```

您可设置用于表示仿真命令的变量，以便快速运行常用的仿真命令。

从 Tcl 控制台启动 Vivado 仿真器

以下示例演示的是如何使用各种 Tcl 命令来创建工程、读入源文件、启动 Vivado 仿真器、执行布局布线、写出 SDF 文件和重新启动仿真。

```
Vivado -mode Tcl
Vivado% create_project prj1
Vivado% read_verilog dut.v
Vivado% synth_design -top dut
Vivado% launch_simulation -simset sim_1 -mode post-synthesis -type
functional
Vivado% place_design
Vivado% route_design
Vivado% write_verilog -mode timesim -sdf_anno true -sdf_file postRoute.sdf
postRoute_netlist.v
Vivado% write_sdf postRoute.sdf
Vivado% launch_simulation -simset sim_1 -mode post-implementation -type
timing
Vivado% close_project
```

export_simulation

为目标仿真器导出仿真脚本文件。生成的脚本包含仿真器命令，用于设计的编译、细化和仿真。

此命令会检索指定对象的仿真编译顺序，并在 shell 脚本中通过目标仿真器的编译器命令和默认选项来导出此信息。指定的对象可以是仿真文件集或 IP。如果要在 Vivado IDE 外运行仿真，则可使用 `export_simulation` 代替 `launch_simulation -scripts_only` 来生成脚本文件。

```
export_simulation [-simulator <arg>] [-of_objects <arg>]
[-step <arg>] [-ip_user_files_dir <arg>] [-
ipstatic_source_dir <arg>]
[-lib_map_path <arg>] [-script_name <arg>]
[-directory <arg>] [-runtime <arg>] [-define <arg>]
[-generic <arg>] [-include <arg>] [-use_ip_compiled_libs]
[-absolute_path] [-export_source_files]
[-generate_hier_access] [-single_step] [-exec] [-force] [-quiet]
[-verbose][-gcc_install_path <arg>] [-more_options <arg>]
```

用法

表 19：**export_simulation 选项**

名称	描述
`[-simulator]`	表示将为其创建仿真脚本的仿真器。允许的值包括：all、xsim、modelsim、questa、vcs、xcelium、riviera 和 activehdl。 默认值：all
`[-of_objects]`	为指定对象导出仿真脚本。 默认值：None
`[-step]`	表示需为其生成仿真脚本的步骤名称。 默认值：None
`[-ip_user_files_dir]`	表示指向导出的 IP/BD 用户文件的目录路径（适用于静态文件、动态文件和数据文件）。对于托管 IP 工程，默认路径为 ip_user_files。 默认值：空
`[-ipstatic_source_dir]`	指向导出的 IP/BD 静态文件的目录路径。 默认值：空
`[-lib_map_path]`	表示预编译的仿真库目录路径。如不指定，则遵循生成的脚本报头中的指示信息来手动提供仿真库映射信息。 默认值：空
`[-gcc_install_path]`	表示对应 g++/gcc 可执行文件的 GNU 编译器安装目录路径。如不指定，则该工具会尝试基于仿真器安装路径设置来查找该路径。 默认值：空
`[-script_name]`	输出 shell 脚本文件名。如不指定，则以默认名称创建文件。 默认值：`top_module.sh`
`[-directory]`	表示将在其中生成仿真脚本的目录。 默认值：`export_sim`
`[-runtime]`	仿真运行时间。 默认值：完整仿真运行的时间或者直至发生逻辑中断或完成条件为止的时间
`[-define]`	从以此开关指定的列表中读取 Verilog define 内容。 默认值：空
`[-generic]`	从以此开关指定的列表中读取 VHDL generic 内容。 默认值：空
`[-include]`	从以此开关指定的列表中读取 include 目录路径。 默认值：空
`[-more_options]`	将指定选项传递给仿真器工具。 默认值：空

表 19：**export_simulation 选项** (续)

名称	描述
-single_step	写出单步脚本。仅限 Xcelium 支持该选项。
[-use_ip_compiled_libs]	在编译期间，引用预编译的 IP 静态库。此开关也可搭配 -ip_user_files_dir 和 -ipstatic_source_dir 开关一起使用，以便使用预编译的 IP 库来生成脚本。
[-absolute_path]	使所有文件路径都变为相对于引用目录的绝对路径。
[-export_source_files]	将 IP/BD 设计文件复制到输出目录。
[-generate_hier_access]	抽取路径，用于分层访问仿真。
-exec	执行生成的脚本。
[-force]	覆盖先前文件。
[-quiet]	忽略命令错误。
[-verbose]	在命令执行期间暂挂消息限制。

描述

为目标仿真器导出仿真脚本文件（请参阅以下受支持的仿真器列表）。生成的脚本包含仿真器命令，用于设计的编译、细化和仿真。

此命令会检索指定对象的仿真编译顺序，并在 shell 脚本中通过目标仿真器的编译器命令和默认选项来导出此信息。指定的对象可以是仿真文件集、IP 或 BD（块设计）。

如不指定对象，那么此命令会为活动状态的仿真 top 生成脚本。包含 Verilog define 语句的文件所在的任意 Verilog include 目录或文件路径都会被添加到编译器命令行上。

默认情况下，编译器命令行中的设计源文件和 include 目录路径都会被设置为所生成的脚本中设置的 ref_dir 变量的相对路径。要将这些路径设为绝对路径，请指定 -absolute_path 开关。

此命令还会把任意已有数据文件从文件集或 IP 复制到输出目录。如果设计包含 Verilog 源文件，那么生成的脚本还会把 glbl.v 文件从软件安装路径复制到输出目录。

目标仿真器的仿真脚本中的编译器命令内使用的默认 .do 文件会写入输出目录。

注释：为了以生成的脚本执行仿真，必须首先使用 compile_simlib Tcl 命令对仿真库进行编译。生成此脚本时，必须指定已编译的库目录路径。生成的脚本会自动包含来自已编译的库目录的目标仿真器的设置文件。

受支持的仿真器

- Vivado 仿真器 (xsim)
- ModelSim 仿真器 (modelsim)
- Questa Advanced Simulator (questa)
- Verilog Compiler Simulator (vcs)
- Riviera-PRO 仿真器 (riviera)
- Active-HDL 仿真器 (activehdl)
- Cadence Xcelium Parallel Simulator (xcelium)

实参

- `-of_objects`：（可选）指定需为其生成仿真脚本文件的目标对象。此目标对象可以是仿真文件集 (simset) 或 IP。如不指定该选项，那么此命令会为当前仿真文件集生成文件。
- `-step`：（可选）指定需为其生成仿真脚本的步骤名称。有效值包括：compile、elaborate 和 simulate。如不指定，那么该脚本会执行适用的所有步骤。
- `-ip_user_files_dir`：（可选）为生成的 IP/BD 源文件指定指向已解压的设计文件的目录路径。对于托管 IP 工程，默认路径为 ip_user_files。如不指定，则会从 `IP.USER_FILES_DIR` 工程属性所设置的默认目录路径中选取解压的设计文件。
- `-ipstatic_source_dir`：（可选）指定指向导出的 IP/BD 生成的静态源文件的目录路径。对于托管 IP 工程，默认路径为 ip_user_files/ipstatic。如不指定该选项，则会从 `SIM.IPSTATIC_SOURCE_DIR` 工程属性所设置的默认目录路径中选取解压的设计文件。
- `-lib_map_path`：（可选）为选定的仿真器指定指向 AMD 预编译的仿真库的路径。仿真库是使用 `compile_simlib` 编译的。请参阅生成的脚本中的报头部分以获取更多信息。如不指定此开关，那么生成的脚本将不会引用预编译的仿真库，静态 IP 文件将执行本地编译。
- `-gcc_install_path`：（可选）为目标仿真器指定指向 GNU 可执行文件的目录路径。对于包含 SystemC、C++ 或 C 语言源文件的设计，该选项是必需的。如果不指定该选项，此工具会尝试基于仿真器安装路径设置来计算指向 GNU 可执行文件的路径。
- `-script_name`：（可选）指定生成的脚本名称。默认名称为 `<simulation_top>.sh`。如果指定 `-of_objects` 开关，那么此脚本的默认语法如下所示：

```
-of_objects [current_fileset -simset] .sh
-of_objects [get_ips ] .sh
-of_objects [get_files .xci] .sh
-of_objects [get_files .bd] .sh
```

- `-absolute_path`：（可选）指定该选项即可使源文件和 include 目录路径变为绝对路径。默认情况下，所有路径都设为以 `-directory` 开关指定的输出目录的相对路径。
- `-force`：（可选）覆盖同名的现有脚本文件。如果脚本文件已存在，那么该工具会返回错误，除非指定了 `-force` 实参。
- `-directory`：（必需）指定用于导出脚本文件的目录路径。
- `-simulator`：（必需）指定仿真脚本的目标仿真器名称。有效的仿真器名称包括 `xsim`、`modelsim`、`questa`、`vcs`（或 `vcs_mx`）、`xcelium`、`riviera` 和 `activehdl`。
- `-quiet`：（可选）以静默方式执行命令，忽略所有命令行错误，不返回任何消息。此命令还会返回 `TCL_OK`，忽略执行期间遇到的所有错误。
- `-verbose`：（可选）暂时覆盖所有消息限制，并返回来自该命令的所有消息。
- `-generate_hier_access`：（可选）抽取路径，用于分层访问仿真。
- `-runtime`：（可选）指定仿真运行时。如不指定，那么仿真将保持运行直至发现逻辑中断或完成声明为止。或者，可使用“<simulator>.simulate.runtime”文件集属性指定运行时
- `-define`：（可选）指定设计中使用的 Verilog define 的列表。或者，可使用“verilog_defines”文件集属性来指定。
- `-generic`：（可选）指定设计中使用的 VHDL generic 的列表。或者，可使用“vhdl_generic”文件集属性来指定。

- `-include`：（可选）指定设计中 verilog include 文件的 include 目录路径列表。或者，可使用“include_dirs”文件集属性来指定。
- `-export_source_files`：（可选）指定该选项即可将 IP 设计文件复制到名为 `srcs` 的子目录中生成的脚本目录。生成的脚本会引用来自该 `srcs` 目录的设计文件。

export_ip_user_files

从工程生成和导出 IP/IP integrator 用户文件。可将作用域限定为处理一个或多个 IP。

语法

```
export_ip_user_files [-of_objects <arg>] [-ip_user_files_dir <arg>]
                     [-ipstatic_source_dir <arg>] [-lib_map_path <arg>]
                     [-no_script] [-sync] [-reset] [-force] [-quiet]
                     [-verbose]
```

返回：已导出的文件列表。

用法

表 20：**export_ip_user_files**

名称	描述
`[-of_objects]`	IP、IP integrator 或文件集。 默认值：None
`[-ip_user_files_dir]`	指向仿真基本目录的目录路径（适用于静态文件、动态文件、封装文件、网表文件、脚本文件和 MEM 文件）。 默认值：None
`[-ipstatic_source_dir]`	指向静态 IP 文件的目录路径。 默认值：None
`[-lib_map_path]`	表示已编译的仿真库目录路径。 默认值：空
`[-no_script]`	不导出仿真脚本。 默认值：1
`[-sync]`	删除 IP/IP integrator 动态文件和仿真脚本文件。
`[-reset]`	删除所有 IP/IP integrator 静态文件、动态文件和仿真脚本文件。
`[-force]`	覆盖文件。
`[-quiet]`	忽略命令错误。
`[-verbose]`	在命令执行期间暂挂消息限制。

描述

导出 IP 用户文件存储库，其中包含静态文件、动态文件、网表文件、Verilog/VHDL 存根文件，和存储器初始化文件。

实参

- `-of_objects`：（可选）指定需为其导出 IP 静态文件和动态文件的目标对象。

- `-ip_user_files_dir`：（可选）指向 IP 用户文件基本目录的目录路径（适用于动态文件和其他 IP 非静态文件）。默认情况下，如果不指定此开关，此命令会使用 `IP.USER_FILES_DIR` 工程属性值所指定的路径。
- `-ipstatic_source_dir`：（可选）指向静态 IP 文件的目录路径。默认情况下，如果不指定此开关，此命令会使用 `SIM.IPSTATIC_SOURCE_DIR` 工程属性值所指定的路径。

  注释：如果指定 `-ip_user_files_dir` 开关，那么默认情况下会在名为 `ipstatic` 的子目录下导出 IP 静态文件。如果指定此开关时包含 `-ipstatic_source_dir`，那么会在 `-ipstatic_source_dir` 开关指定的路径中导出 IP 静态文件。

示例

以下命令用于把 `char_fifo` IP 动态文件导出到 `<project>/<project>.ip_user_files/ip/char_fifo` 目录，并把 `char_fifo` IP 静态文件导出到 `<project>/<project>.ip_user_files/ipstatic` 目录：

```
% export_ip_user_files -of_objects [get_ips char_fifo]
```

AMD

附录 A

编译、细化、仿真、网表和高级选项

在 Vivado IDE Flow Navigator 中，您可右键单击“Simulation”（仿真），并选择“Simulation Settings”（仿真设置）以在“Settings”（设置）对话框中打开仿真设置。在仿真设置中，您可设置各种编译、细化、仿真、网表和高级选项。

编译器选项

“Compilation”（编译）选项卡用于定义和管理编译器指令，这些指令作为属性存储在仿真文件集上，供 xvlog 和 xvhdl 实用工具用于编译 Verilog 和 VHDL 源文件以便进行仿真。

Vivado 仿真器编译选项

表 21：**Vivado 仿真器编译选项**

选项	描述
Verilog 选项	浏览并设置 Verilog include 路径和定义宏
泛型/参数选项	指定或浏览并设置泛型/参数值
xsim.compile.tcl.pre	此 Tcl 文件包含一组命令，应在启动编译前调用这组命令
xsim.compile.xvlog.nosort	在编译期间不对 Verilog 文件进行排序
xsim.compile.xvhdl.nosort	在编译期间不对 VHDL 文件进行排序
xsim.compile.xvlog.relax	放宽严格的 Verilog 和 SystemVerilog 语言检查规则
xsim.compile.xvhdl.relax	放宽严格的 VHDL 语言检查规则
xsim.compile.xsc.mt_level	指定要并行运行的子编译作业数量
xsim.compile.xvlog.more_options	更多 XVLOG 编译选项
xsim.compile.xvhdl.more_options	更多 XVHDL 编译选项
xsim.compile.xsc.more_options	更多 XSC 编译选项

Questa Advanced Simulator 编译选项

表 22：**Questa Advanced Simulator 编译选项**

选项	描述
Verilog 选项	浏览并设置 Verilog include 路径和定义宏
泛型/参数选项	指定或浏览并设置泛型/参数值
questasim.compile.tcl.pre	此 Tcl 文件包含一组命令，应在启动编译前调用这组命令

表 22：**Questa Advanced Simulator 编译选项** (续)

选项	描述
questasim.compile.vhdl_syntax	指定 VHDL 语法
questasim.compile.use_explicit_decl	记录所有信号
questasim.compile.load_glbl	加载 GLBL 模块
questasim.compile.vlog.more_options	更多 VLOG 编译选项
questasim.compile.vcom.more_options	更多 VCOM 编译选项
questasim.compile.sccom.cores	指定要并行运行的处理器核的数量
questasim.compile.sccom.more_options	更多 SCCOM 编译选项

ModelSim 仿真器编译选项

表 23：**ModelSim 编译选项**

选项	描述
Verilog 选项	浏览并设置 Verilog include 路径和定义宏
泛型/参数选项	指定或浏览并设置泛型/参数值
modelsim.compile.tcl.pre	此 Tcl 文件包含一组命令，应在启动编译前调用这组命令
modelsim.compile.vhdl_syntax	指定 VHDL 语法
modelsim.compile.use_explicit_decl	记录所有信号
modelsim.compile.load_glbl	加载 GLBL 模块
modelsim.compile.vlog.more_options	更多 VLOG 编译选项
modelsim.compile.vcom.more_options	更多 VCOM 编译选项

VCS 仿真器编译选项

表 24：**VCS 仿真器编译选项**

选项	描述
Verilog 选项	浏览并设置 Verilog include 路径和定义宏
泛型/参数选项	指定或浏览并设置泛型/参数值
vcs.compile.tcl.pre	此 Tcl 文件包含一组命令，应在启动编译前调用这组命令
vcs.compile.load_glbl	加载 GLBL 模块
vcs.compile.vhdlan.more_options	更多 VHDLAN 编译选项
vcs.compile.vlogan.more_options	额外 VLOGAN 编译选项
vcs.compile.syscan.more_options	更多 SYSCAN 编译选项
vcs.compile.g++.more_options	更多 G++ 编译选项
vcs.compile.gcc.more_options	更多 GCC 编译选项

Xcelium Simulator 编译选项

表 25：**Xcelium 编译选项**

选项	描述
Verilog 选项	浏览并设置 Verilog include 路径和定义宏
泛型/参数选项	指定或浏览并设置泛型/参数值
xcelium.compile.tcl.pre	此 Tcl 文件包含一组命令，应在启动编译前调用这组命令
xcelium.compile.v93	启用 VHDL-93 功能特性
xcelium.compile.relax	启用宽松的 VHDL 解读
xcelium.compile.load_glbl	加载 GLBL 模块
xcelium.compile.xmvhdl.more_options	更多 XMVHDL 编译选项
xcelium.compile.xmvlog.more_options	更多 XMVLOG 编译选项
xcelium.compile.xmsc.more_options	更多 XMSC 编译选项
xcelium.compile.g++.more_options	更多 G++ 编译选项
xcelium.compile.gcc.more_options	更多 GCC 编译选项

细化选项

“Elaboration”（细化）选项卡用于定义和管理细化指令，这些指令作为属性存储在仿真文件集上，并供 xelab 实用工具用于细化和生成仿真快照。选择该表中的属性即可显示其描述并编辑其值。

Vivado 仿真器细化选项

表 26：**Vivado 仿真器细化选项**

选项	描述
xsim.elaborate.snapshot	指定仿真快照名称
xsim.elaborate.debug_level	选择仿真调试可视化级别。默认为“typical”
xsim.elaborate.relax	放宽严格的 HDL 语言检查规则
xsim.elaborate.mt_level	指定要并行运行的子编译作业数量
xsim.elaborate.load_glbl	加载 GLBL 模块
xsim.elaborate.rangecheck	为 VHDL 启用运行时值范围检查
xsim.elaborate.sdf_delay	指定要读取的 sdf 时序延迟类型以供在时序仿真内使用
xsim.elaborate.xelab.more_options	更多 XELAB 细化选项
xsim.elaborate.xsc.more_options	更多 XSC 选项，可在细化期间使用
xsim.elaborate.coverage.name	指定覆盖数据库名称
xsim.elaborate.coverage.dir	指定覆盖数据库目录名称
xsim.elaborate.coverage.type	指定覆盖类型（行分支条件或全部）
xsim.elaborate.coverage.library	跟踪 std/unisims/retarget 库
xsim.elaborate.coverage.celldefine	跟踪含 celldefine 属性的模块

表 26：**Vivado 仿真器细化选项** (续)

选项	描述
xsim.elaborate.link.sysc	指定要绑定的 SystemC 库
xsim.elaborate.link.c	指定要绑定的 C/C++ 库

Questa Advanced Simulator 细化选项

表 27：**Questa Advanced Simulator 细化选项**

选项	描述
questasim.elaborate.acc	启用对仿真对象的访问，默认情况下可能会对这些仿真对象进行最优化（默认值：npr）
questasim.elaborate.vopt.more_options	更多 VOPT 细化选项
questasim.elaborate.sccom.more_options	更多 sccom 选项，可在细化期间使用
questasim.elaborate.link.sysc	指定要绑定的 SystemC 库
questasim.elaborate.link.c	指定要绑定的 C/C++ 库

ModelSim 仿真器细化选项

表 28：**ModelSim 细化选项**

选项	描述
modelsim.elaborate.acc	启用对仿真对象的访问，默认情况下可能会对这些仿真对象进行最优化
modelsim.elaborate.vopt.more_options	更多 VOPT 细化选项

VCS 仿真器细化选项

表 29：**VCS 细化选项**

选项	描述
vcs.elaborate.debug_pp	启用后处理调试访问
vcs.elaborate.vcs.more_options	更多 VCS 细化选项
vcs.elaborate.link.sysc	指定要绑定的 SystemC 库
vcs.elaborate.link.c	指定要绑定的 C/C++ 库

Xcelium Simulator 细化选项

表 30：**Xcelium 细化选项**

选项	描述
xcelium.elaborate.update	写入前检查此单元是否处于最新状态
xcelium.elaborate.xmelab.more_options	更多 `xmelab` 细化选项
xcelium.elaborate.link.sysc	指定要绑定的 SystemC 库

表 30：**Xcelium 细化选项** (续)

选项	描述
xcelium.elaborate.link.c	指定要绑定的 C/C++ 库

仿真选项

“Simulation”（仿真）选项卡用于定义和管理仿真指令，这些指令作为属性存储在仿真文件集上，供 xsim 应用用于对当前工程进行仿真。选择该表中的属性即可显示其描述并编辑其值。

Vivado 仿真器仿真选项

表 31：**Vivado 仿真器仿真选项**

选项	描述
xsim.simulate.runtime	为 Vivado 仿真器指定仿真运行时间。输入空白即可仅加载仿真快照并等待用户输入。
xsim.simulate.tcl.post	此 Tcl 文件包含一组您需在仿真末尾调用的命令。
xsim.simulate.log_all_signals	记录所有对象信号
xsim.simulate.wdb	指定仿真波形数据库文件
xsim.simulate.saif	指定 SAIF 文件名
xsim.simulate.saif_scope	指定要执行功耗估算的设计层级实例名称。
xsim.simulate.saif_all_signals	为受测设计记录所有对象信号，用于 SAIF 文件生成
xsim.simulate.xsim.more_options	更多 Vivado 仿真器仿真选项
xsim.simulate.custom_tcl	指定定制 tcl 文件的名称，此文件将在仿真期间作为源文件替代 Vivado 生成的常规 Tcl 文件
xsim.simulate.add_positional	向 XSIM 添加定位参数，用于传递命令行实参
xsim.simulate.no_quit	不退出仿真

Questa Advanced Simulator 仿真选项

表 32：**Questa Advanced Simulator 仿真选项**

选项	描述
questasim.simulate.runtime	指定仿真运行时
questasim.simulate.tcl.post	此 Tcl 文件包含一组您需在仿真末尾调用的命令。
questasim.simulate.log_all_signals	记录所有信号
questasim.simulate.custom_do	指定自定义 `do` 文件的名称
questasim.simulate.custom_udo	指定自定义用户 `do` 文件的名称
questa.simulate.ieee_warning	禁止 IEEE 警告
questasim.simulate.sdf_delay	指定 `sdf` 注解的延迟类型
questasim.simulate.saif	指定 `SAIF` 文件
questasim.simulate.saif_scope	指定要执行功耗估算的设计层级实例名称

表 32：**Questa Advanced Simulator 仿真选项** (续)

选项	描述
questasim.simulate.sc_async_update	为 SystemC 启用异步请求更新
questasim.simulate.vsim.more_options	更多 VSIM 仿真选项
questa.simulate.custom_wave_do	自定义 wave.do 文件的名称，此文件用于替代常规 Vivado 生成的 wave.do 文件

ModelSim 仿真器仿真选项

表 33：**ModelSim 仿真选项**

选项	描述
modelsim.simulate.runtime	指定仿真运行时。
modelsim.simulate.tcl.post	此 Tcl 文件包含一组您需在仿真末尾调用的命令。
modelsim.simulate.log_all_signals	记录所有信号。
modelsim.simulate.custom_do	指定自定义 `do` 文件的名称。
modelsim.simulate.custom_udo	指定自定义用户 `do` 文件的名称。
modelsim.simulate.sdf_delay	指定 `sdf` 注解的延迟类型。
modelsim.simulate.ieee_warning	禁止 IEEE 警告。
modelsim.simulate.saif	指定 `SAIF` 文件。
modelsim.simulate.saif_scope	指定要执行功耗估算的设计层级实例名称。
modelsim.simulate.vsim.more_options	更多 VSIM 仿真选项。
modelsim.simulate.custom_wave_do	自定义 wave.do 文件的名称，此文件用于替代常规 Vivado 生成的 wave.do 文件。

VCS 仿真器仿真选项

表 34：**VCS 仿真选项**

选项	描述
vcs.simulate.runtime	指定仿真运行时
vcs.simulate.tcl.post	此 Tcl 文件包含一组您需在仿真末尾调用的命令。
vcs.simulate.log_all_signals	记录所有信号
vcs.simulate.saif	SAIF 文件名
vcs.simulate.saif_scope	指定要执行功耗估算的设计层级实例名称
vcs.simulate.vcs.more_options	更多 VCS 仿真选项

Xcelium Simulator 仿真选项

表 35：**Xcelium Simulator 仿真选项**

选项	描述
xcelium.simulate.tcl.post	此 Tcl 文件包含一组您需在仿真末尾调用的命令
xcelium.simulate.runtime	指定仿真运行时

表 35：**Xcelium Simulator 仿真选项** (续)

选项	描述
xcelium.simulate.log_all_signals	记录所有信号
xcelium.simulate.update	写入前检查此单元是否处于最新状态
xcelium.simulate.ieee_warnings	禁止 IEEE 警告
xcelium.simulate.saif_scope	SAIF 文件名
xcelium.simulate.saif	指定要执行功耗估算的设计层级实例名称
xcelium.simulate.xmsim.more_options	更多 XMSIM 仿真选项

网表选项

“Netlist”（网表）选项卡允许访问与 Verilog 网表的 SDF 注解以及 SDF 延迟所捕获的工艺角相关的网表配置选项。这些选项作为属性存储在仿真文件集上，在为仿真编写网表时使用。

Vivado 仿真器网表选项

表 36：**Vivado 仿真器网表选项**

选项	描述
`-sdf_anno`	有复选框可用于选择 `-sdf_anno` 选项。默认启用该选项
`-process_corner`	您可指定的 `-process_corner` 选项包括：fast 和 slow

注释：所有第三方仿真器（Questa Advanced Simulator、ModelSim 仿真器、VCS 和 Xcelium Simulator）的网表选项都与 Vivado 仿真器的网表选项类似。

高级仿真选项

“Advanced”（高级）选项卡包含两个选项。

- “Enable incremental compilation”（启用增量编译）：该选项用于在连续运行期间启用增量编译并保留仿真文件。
- “Include all design sources for simulation”（包含所有设计源文件用于仿真）选项：默认启用该选项。选中该选项可确保来自设计源文件的所有文件以及来自当前仿真集的文件都用于仿真。即使您更改设计源文件，启动行为仿真时所更改的内容也会一并更新。

> **重要提示！**这是高级用户功能。取消选中此框可能产生意外结果。默认情况下选中“Include all design sources for simulation”复选框。只要选中此复选框，仿真集就会包含非关联 (OOC) IP、IP integrator 文件和 DCP。

取消选中此框支持您灵活选择仅包含仿真所需的文件，但是，如上所述，您可能会遇到意外的结果。

注释：所有仿真器的高级仿真选项都相同。

附录 B

Vivado 仿真器中的 SystemVerilog 支持

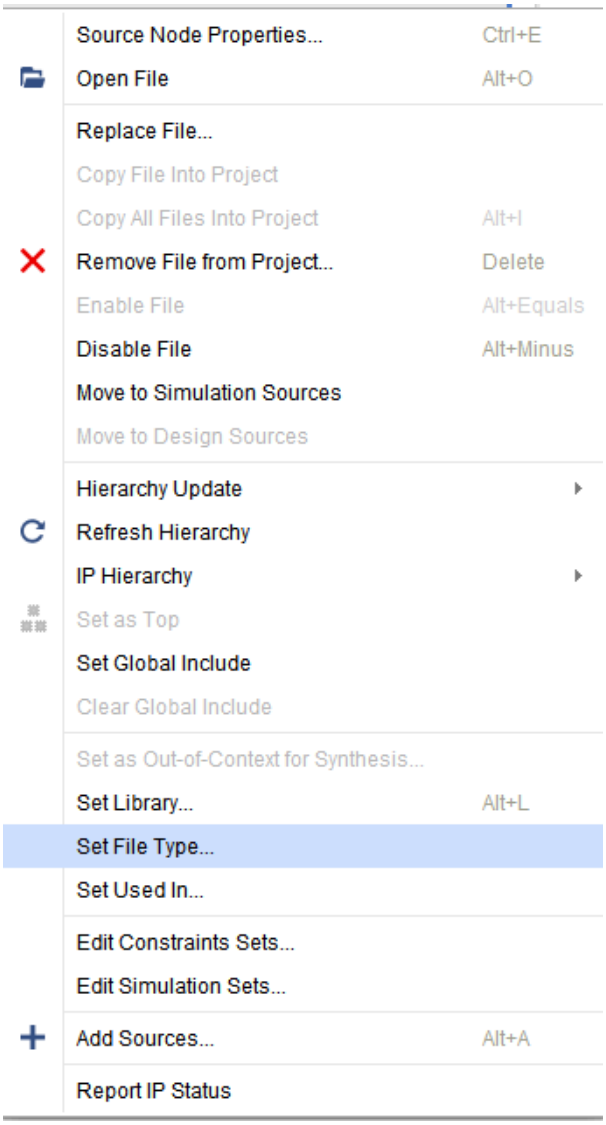
Vivado 仿真器支持 SystemVerilog 子集。下表中列出了 SystemVerilog 的可综合集合。表 38：受支持的动态类型构造中列出了支持的测试激励文件功能特性。

将 SystemVerilog 用于特定文件

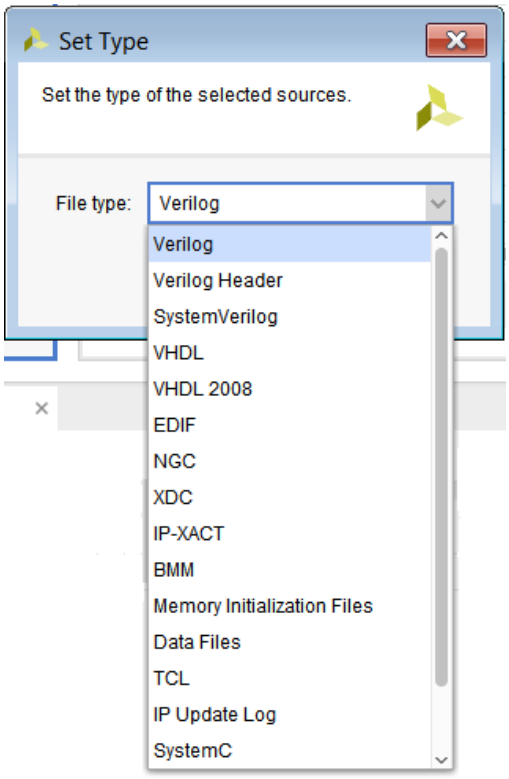
默认情况下，Vivado IDE 会使用 Verilog 2001 语法编译 `.v` 文件，并使用 SystemVerilog 语法编译 `.sv` 文件。

要将 SystemVerilog 用于 Vivado IDE 中的特定 `.v` 文件，请执行以下操作：

1. 右键单击此文件并选择“Set file type”（设置文件类型），如下图所示。



2. 在下图所示“Set Type”（设置类型）对话框中，将文件类型从 Verilog 更改为“SystemVerilog”，然后单击“OK”（确定）。



或者，您可在 Tcl 控制台中使用以下命令：

```
set_property file_type SystemVerilog [get_files <filename>.v]
```

在独立模式或 prj 模式下运行 SystemVerilog

独立模式

对于 `xvlog` 现已引入了新的 `-sv` 标志，因此，如果您要读取任何 SystemVerilog 文件，可使用以下命令：

```
    xvlog -sv <Design file list>
    xvlog -sv -work <LibraryName> <Design File List>
    xvlog -sv -f <FileName> [Where FileName contain path of test cases]
```

prj 模式

如果要在基于 `prj` 的流程中运行 Vivado 仿真器，请使用 `sv` 作为文件类型，就像使用 `verilog` 或 `vhdl` 一样。

```
xvlog -prj <prj File>
xelab -prj <prj File> <topModuleName> <other options>
```

其中，`prj` 文件中的条目如下所示：

```
verilog     library1 <FileName>
sv      library1 <FileName> [File parsed in SystemVerilog mode]
vhdl     library2 <FileName>
sv      library3 <FileName> [File parsed in SystemVerilog mode]
```

表 37：**SystemVerilog 1800-2012 的可综合集合**

主构造	辅构造	LRM 部分	状态
数据类型		6	
	单类型与聚合类型	6.4	受支持
	信号线和变量	6.5	受支持
	变量声明	6.8	受支持
	矢量声明	6.9	受支持
	二态（双值）和四态（四值）数据类型	6.11.2	受支持
	有符号和无符号整数类型	6.11.3	受支持
	real、shortreal 和 realtime 数据类型	6.12	受支持
	用户定义的类型	6.18	受支持
	枚举	6.19	受支持
	将新数据类型定义为枚举类型	6.19.1	受支持
	枚举类型范围	6.19.2	受支持
	类型检查	6.19.3	受支持
	数字表达式中的枚举类型	6.19.4	受支持
	枚举类型方法	6.19.5	受支持
	类型参数	6.20.3	受支持
	Const 常量	6.20.6	受支持
	type 类型运算符	6.23	受支持
	cast 强制转换运算符	6.24.1	受支持
	`$cast` 动态强制转换	6.24.2	受支持
	比特流强制转换	6.24.3	受支持
聚合数据类型		7	
	结构	7.2	受支持
	打包/解包结构	7.2.1	受支持
	给结构赋值	7.2.2	受支持
	联合体	7.3	受支持
	打包/解包联合体	7.3.1	受支持
	带标签的联合体	7.3.2	不支持
	打包阵列	7.4.1	受支持
	解包阵列	7.4.2	受支持
	阵列运算	7.4.3	受支持
	多维阵列	7.4.5	受支持
	阵列索引和分片	7.4.6	受支持
	阵列赋值	7.6	受支持
	阵列作为子例程的实参	7.7	受支持
	阵列查询函数	7.11	受支持
	阵列操纵方法	7.12	受支持
进程		9	
	组合逻辑 `always_comb` 过程	9.2.2	受支持

表 37：**SystemVerilog 1800-2012 的可综合集合** (续)

主构造	辅构造	LRM 部分	状态
	默示 `always_comb` 敏感度	9.2.2.1	受支持
	锁存逻辑 `always_latch` 过程	9.2.2.3	受支持
	顺序逻辑 `always_ff` 过程	9.2.2.4	受支持
	顺序块	9.3.1	受支持
	并行块	9.3.2	受支持
	过程性时序控制	9.4	受支持
	条件事件控制	9.4.2.3	受支持
	顺序事件	9.4.2.4	不支持
赋值语句		10	
	连续赋值语句	10.3.2	受支持
	变量声明赋值（变量初始化）	10.5	受支持
	类赋值上下文	10.8	受支持
	阵列赋值模式	10.9.1	受支持
	结构赋值模式	10.9.2	受支持
	解包阵列串联	10.10	受支持
	信号线失真	10.11	受支持
运算符和表达式		11	
	常量表达式	11.2.1	受支持
	聚合表达式	11.2.2	受支持
	含实数操作数的运算符	11.3.1	受支持
	逻辑（四态）和位（二态）类型运算	11.3.4	受支持
	表达式内赋值	11.3.6	受支持
	赋值运算符	11.4.1	受支持
	增量运算符和减量运算符	11.4.2	受支持
	含无符号类型和有符号类型的算术表达式	11.4.3.1	受支持
	通配符等同性运算符	11.4.6	受支持
	串联运算符	11.4.12	受支持
	设置成员资格运算符	11.4.13	受支持
	`stream_expressions` 串联	11.4.14.1	受支持
	泛型串流重新排序	11.4.14.2	受支持
	串流串联作为赋值目标（解包）	11.4.14.3	受支持
	串流动态大小的数据	11.4.14.4	受支持
过程性编程语句		12	
	`unique-if`、`unique0-if` 和 `priority-if`	12.4.2	受支持
	由 `unique-if`、`unique0-if` 和 `priority-if` 构造生成的违例报告	12.4.2.1	受支持
	If 语句违例报告和多进程	12.4.2.2	受支持
	`unique-case`、`unique0-case` 和 `priority-case`	12.5.3	受支持

表 37：**SystemVerilog 1800-2012 的可综合集合** (续)

主构造	辅构造	LRM 部分	状态
	由 `unique-case`、`unique0-case` 和 `priority-case` 构造生成的违例报告	12.5.3.1	受支持
	Case 语句违例报告和多进程	12.5.3.2	受支持
	设置成员资格 case 语句	12.5.4	受支持
	模式匹配和条件语句	12.6	不支持
	循环语句	12.7	受支持
	跳转语句	12.8	受支持
任务		13.3	
	静态和自动任务	13.3.1	受支持
	任务存储器使用情况和并发激活	13.3.2	受支持
功能		13.4	
	返回值和无效函数	13.4.1	受支持
	静态和自动函数	13.4.2	受支持
	常量函数	13.4.3	受支持
	由函数调用生成的后台进程	13.4.4	受支持
子例程调用和实参传递		13.5	
	按值传递	13.5.1	受支持
	按引用传递	13.5.2	受支持
	默认实参值	13.5.3	受支持
	按名称绑定实参	13.5.4	受支持
	可选实参列表	13.5.5	受支持
	导入和导出函数	13.6	受支持
	任务和函数名称	13.7	受支持
实用工具系统任务和系统函数（仅限可综合集合）		20	受支持
I/O 系统任务和系统函数（仅限可综合集合）		21	受支持
编译器指令		22	受支持
模块和层级		23	
	默认端口值	23.2.2.4	受支持
	顶层模块和 `$root`	23.3.1	受支持
	模块例化语法	23.3.2	受支持
	嵌套模块	23.4	受支持
	外部模块	23.5	受支持
	分层名称	23.6	受支持
	成员选择和分层名称	23.7	受支持
	向上名称引用	23.8	受支持
	覆盖模块参数	23.10	受支持
	将辅助代码绑定到作用域或实例	23.11	不支持
接口		25	

表 37：**SystemVerilog 1800-2012 的可综合集合** (续)

主构造	辅构造	LRM 部分	状态
	接口语法	25.3	受支持
	嵌套接口	25.3	受支持
	接口中的端口	25.4	受支持
	指定端口捆绑示例	25.5.1	受支持
	连接端口捆绑示例	25.5.2	受支持
	将端口捆绑连接到泛型接口的示例	25.5.3	受支持
	Modport 表达式	25.5.4	受支持
	时钟块和 modport	25.5.5	受支持
	接口和 specify 块	25.6	受支持
	在接口中使用任务的示例	25.7.1	受支持
	在 modport 中使用任务的示例	25.7.2	受支持
	导出任务和函数的示例	25.7.3	受支持
	多项任务导出的示例	25.7.4	受支持
	参数化接口	25.8	受支持
	虚拟接口	25.9	受支持
数据包		26	
	数据包声明	26.2	受支持
	引用数据包内的数据	26.3	受支持
	在模块头文件中使用数据包	26.4	受支持
	从数据包导出已导入的名称	26.6	受支持
	std 内置数据包	26.7	受支持
生成构造		27	受支持

测试激励文件功能特性

在 Vivado 仿真器中，添加了对于部分常用测试激励文件功能特性的支持，如下表所示：

表 38：**受支持的动态类型构造**

主构造	辅构造	LRM 部分	状态
字符串数据类型		6.16	受支持
	字符串运算符（IEEE 1800-2012 的表 6-9）	6.16	受支持
	Len()	6.16.1	受支持
	Putc()	6.16.2	受支持
	Getc()	6.16.3	受支持
	Toupper()	6.16.4	受支持
	Tolower()	6.16.5	受支持
	Compare	6.16.6	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	Icompare()	6.16.7	受支持
	Substr()	6.16.8	受支持
	Atoi()、atohex()、atooct() 和 atobin()	6.16.9	受支持
	Atoreal()	6.16.10	受支持
	Itoa()	6.16.11	受支持
	Hextoa()	6.16.12	受支持
	Octtoa()	6.16.13	受支持
	Bintoa()	6.16.14	受支持
	Realtoa()	6.16.15	受支持
动态阵列		7.5	受支持
	动态阵列新增	7.5.1	受支持
	大小	7.5.2	受支持
	删除	7.5.3	受支持
关联阵列		7.8	受支持
	通配符索引	7.8.1	受支持
	字符串索引	7.8.2	受支持
	类索引	7.8.3	受支持
	整型索引	7.8.4	受支持
	其他用户定义的类型	7.8.5	受支持
	访问无效的索引	7.8.6	受支持
	关联阵列方法	7.9	受支持
	Num() 和 Size()	7.9.1	受支持
	Delete()	7.9.2	受支持
	Exists()	7.9.3	受支持
	First()	7.9.4	受支持
	Last()	7.9.5	受支持
	Next()	7.9.6	受支持
	Prev()	7.9.7	受支持
	遍历方法的实参	7.9.8	受支持
	关联阵列赋值	7.9.9	受支持
	关联阵列实参	7.9.10	受支持
	关联阵列字面值	7.9.11	受支持
队列		7.10	受支持
	队列运算符	7.10.1	受支持
	队列方法	7.10.2	受支持
	Size()	7.10.2.1	受支持
	Insert()	7.10.2.2	受支持
	Delete()	7.10.2.3	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	Pop_front()	7.10.2.4	受支持
	Pop_back()	7.10.2.5	受支持
	Push_front()	7.10.2.6	受支持
	Push_back()	7.10.2.7	受支持
	队列元素的持久引用	7.10.3	受支持
	使用赋值和解包的阵列串联来更新队列	7.10.4	受支持
	受限队列	7.10.5	受支持
类		8	受支持
	通用类	8.1	受支持
	概述	8.2	受支持
	语法	8.3	受支持
	对象（类实例）	8.4	受支持
	对象属性和对象参数数据	8.5	受支持
	对象方法	8.6	受支持
	构造函数	8.7	受支持
	静态类属性	8.8	受支持
	静态方法	8.9	受支持
	this 关键字	8.10	受支持
	赋值、重命名和复制	8.11	受支持
	继承和子类	8.12	受支持
	被覆盖的成员	8.13	受支持
	super 关键字	8.14	受支持
	强制类型转换	8.15	受支持
	链式构造函数	8.16	受支持
	数据隐藏和封装	8.17	受支持
	常量类属性	8.18	受支持
	虚拟方法	8.19	受支持
	抽象类和纯虚拟方法	8.20	受支持
	多态性：动态方法查找	8.21	受支持
	类作用域解析运算符 ::	8.22	受支持
	块外声明	8.23	受支持
	参数化的类	8.24	受支持
	参数化的类的类解析运算符	8.24.1	受支持
	Typedef 类	8.25	受支持
	接口类	8.26	受支持
	接口类的多重继承	8.26.6	受支持
	存储器管理	8.27	受支持
	类和结构	8.28	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
进程		9	受支持
	并行进程 - Join_Any 分叉和 Join_None 分叉	9.3	受支持
	等待分叉	9.6.1	受支持
	禁用分叉	9.6.3	受支持
	高精度进程控制	9.7	受支持
时钟设置块		14	受支持
	通用	14.1	受支持
	概述	14.2	受支持
	时钟设置块声明	14.3	受支持
	输入和输出偏差	14.4	受支持
	分层表达式	14.5	不支持
	多个时钟设置块内的信号	14.6	受支持
	时钟设置块作用域和生存期	14.7	受支持
	多个时钟设置块的示例	14.8	受支持
	接口和时钟设置块	14.9	受支持
	时钟设置块事件	14.10	受支持
	周期延迟	14.11	受支持
	默认时钟设置	14.12	受支持
	输入采样	14.13	受支持
	全局时钟设置	14.14	不支持
	同步事件	14.15	受支持
	同步驱动	14.16	受支持
	驱动和非阻塞赋值	14.16.1	受支持
	驱动时钟设置输出信号	14.16.2	受支持
信号量		15.3	受支持
	信号量方法 new()	15.3.1	受支持
	信号量方法 put()	15.3.2	受支持
	信号量方法 get()	15.3.3	受支持
	信号量方法 try_get()	15.3.4	受支持
邮箱		15.4	受支持
	邮箱方法 new()	15.4.1	受支持
	邮箱方法 num()	15.4.2	受支持
	邮箱方法 put()	15.4.3	受支持
	邮箱方法 try_put()	15.4.4	受支持
	邮箱方法 get()	15.4.5	受支持
	邮箱方法 try_get()	15.4.6	受支持
	邮箱方法 peek()	15.4.7	受支持
	邮箱方法 try_peek()	15.4.8	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	参数化邮箱	15.4.9	受支持
命名的事件		15.5	受支持
	触发事件	15.5.1	受支持
	等待事件	15.5.2	受支持
	持久触发	15.5.3	不支持
	事件序列	15.5.4	不支持
	对命名的事件变量执行的操作	15.5.5	受支持
	合并事件	15.5.5.1	受支持
	回收事件	15.5.5.2	受支持
	事件比较	15.5.5.3	受支持
断言有效		16	受支持
	通用	16.1	受支持
	概述	16.2	受支持
	断言有效	16.2	受支持
	假定	16.2	受支持
	涵盖	16.2	不支持
	限制	16.2	不支持
	即时断言有效	16.3	受支持
	延迟断言有效	16.4	不支持
	并发断言有效概述	16.5	受支持
	采样	16.5.1	受支持
	断言有效时钟	16.5.2	受支持
	布尔表达式	16.6	受支持
	顺序	16.7	受支持
	声明顺序	16.8	受支持
	顺序声明中的有型形参	16.8.1	受支持
	顺序声明中的局部变量形参	16.8.2	受支持
	顺序操作	16.9	受支持
	运算符优先	16.9.1	受支持
	顺序重复	16.9.2	受支持
	采样值函数	16.9.3	受支持
	全局时钟设置过去和未来采样值函数	16.9.4	不支持
	AND 运算	16.9.5	受支持
	相交（含长度限制的 AND）	16.9.6	受支持
	OR 运算	16.9.7	受支持
	First_match 运算	16.9.8	受支持
	顺序条件	16.9.9	受支持
	序列中包含的另一序列	16.9.10	受支持
	根据较简单的序列来构成序列	16.9.11	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	局部变量	16.10	受支持
	在匹配的序列上调用子例程	16.11	受支持
	Declaring 属性	16.12	受支持
	Sequence 属性	16.12.1	受支持
	Negation 属性	16.12.2	受支持
	Disjunction 属性	16.12.3	受支持
	Conjunction 属性	16.12.4	受支持
	If-else 属性	16.12.5	受支持
	Implication	16.12.6	受支持
	Implies 属性和 iff 属性	16.12.7	受支持
	属性例化	16.12.8	受支持
	Followed-by 属性	16.12.9	不支持
	Next time 属性	16.12.10	不支持
	Always 属性	16.12.11	不支持
	Until 属性	16.12.12	不支持
	Eventually 属性	16.12.13	不支持
	Abort 属性	16.12.14	不支持
	弱运算符和强运算符	16.12.15	不支持
	大小写	16.12.16	不支持
	递归属性	16.12.17	不支持
	属性声明中的有型形参	16.12.18	受支持
	属性声明中的局部变量形参	16.12.19	受支持
	属性示例	16.12.20	受支持
	有限长度与无限长度行为对比	16.12.21	受支持
	非简并	16.12.22	受支持
	多时钟支持	16.13	不支持
	并发断言有效	16.14	受支持
	断言有效声明	16.14.1	受支持
	假定声明	16.14.2	受支持
	涵盖声明	16.14.3	不支持
	限制声明	16.14.4	不支持
	在过程性代码外使用并发断言有效声明	16.14.5	受支持
	在过程性代码内嵌入并发断言有效	16.14.6	不支持
	推断值函数	16.14.7	不支持
	非空洞求值	16.14.8	不支持
	禁用 iff 解析	16.15	受支持
	时钟解析	16.16	受支持
	适用于多时钟序列和属性的语义前导时钟	16.16.1	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	Expect 语句	16.17	不支持
	时钟设置块和并发断言有效	16.18	受支持
随机约束		18	受支持
	概念和用法	18.3	受支持
	随机变量	18.4	受支持
	Rand 修饰符	18.4.1	受支持
	Randc 修饰符	18.4.2	受支持
	约束块	18.5	受支持
	外部约束块	18.5.1	受支持
	约束继承	18.5.2	受支持
	设置成员资格	18.5.3	受支持
	分布	18.5.4	受支持
	Implication	18.5.6	受支持
	If-else 约束	18.5.7	受支持
	迭代约束	18.5.8	受支持
	foreach 迭代约束	18.5.8.1	受支持
	阵列缩减迭代约束	18.5.8.2	受支持
	全局约束	18.5.9	受支持
	可变排序	18.5.10	受支持
	静态约束块	18.5.11	受支持
	约束中的函数	18.5.12	受支持
	约束保护	18.5.13	受支持
	软核约束	18.5.14	受支持
	随机化方法	18.6.1	受支持
	Pre_randomize 和 post_randomize	18.6.2	受支持
	随机化方法的行为	18.6.3	受支持
	内联约束	18.7	受支持
	局部作用域解析	18.7.1	受支持
	禁用含 rand_mode 的随机变量	18.8	受支持
	控制含 constraint_mode 的约束	18.9	受支持
	动态约束修改	18.10	受支持
	内联随机变量控制	18.11	受支持
	内联约束检查器	18.11.1	受支持
	作用域变量 std::randomize 的随机化	18.12	受支持
	项作用域变量 std::randomize 添加约束	18.12.1	受支持
	随机数字系统函数和方法	18.13	受支持
	$urandom	18.13.1	受支持
	$urandom_range	18.13.2	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	srandom	18.13.3	受支持
	Get_randstate	18.13.4	受支持
	Set_randstate	18.13.5	受支持
	随机稳定性	18.14	受支持
	手动植入随机化	18.15	受支持
	Randcase	18.16	受支持
	Randsequence	18.17	不支持
编程		24	受支持
	编程构造	24.3	受支持
	调度编程构造内代码的语义	24.3.1	受支持
	编程端口连接	24.3.2	受支持
	消除测试激励文件竞赛	24.4	受支持
	周期/事件模式内的阻塞任务	24.5	受支持
	匿名程序	24.6	不支持
	程序控制任务	24.7	受支持
功能覆盖		19	受支持
	通用	19.1	受支持
	概述	19.2	受支持
	定义覆盖模型：covergroup	19.3	受支持
	在类中使用 covergroup	19.4	受支持
	定义覆盖点	19.5	受支持
	指定值的分箱	19.5.1	受支持
	含 covergroup 表达式的覆盖点分箱	19.5.1.1	受支持
	设置 covergroup 表达式的覆盖点分箱	19.5.1.2	不支持
	指定转换的分箱	19.5.2	受支持
	对应覆盖点的自动分箱创建	19.5.3	受支持
	覆盖点分箱的通配符规范	19.5.4	受支持
	排除覆盖点值或转换	19.5.5	受支持
	指定违规的覆盖点值或转换	19.5.6	受支持
	值解析	19.5.7	受支持
	定义交叉覆盖	19.6	受支持
	定义交叉覆盖分箱	19.6.1	受支持
	用户定义的交叉覆盖和选择表达式示例	19.6.1.1	受支持
	含 covergroup 表达式的交叉分箱	19.6.1.2	受支持
	交叉分箱自动定义的类型	19.6.1.3	受支持
	交叉分箱设置表达式	19.6.1.4	受支持
	排除叉乘	19.6.2	受支持
	指定违规的叉乘	19.6.3	受支持

表 38：**受支持的动态类型构造** (续)

主构造	辅构造	LRM 部分	状态
	指定覆盖选项	19.7	受支持
	Covergroup 类型选项	19.7.1	受支持
	预定义的 coverage 方法	19.8	受支持
	覆盖内置采样方法	19.8.1	受支持
	预定义的覆盖系统任务和系统函数	19.9	受支持
	选项和 type_option 成员的组织	19.10	受支持

注释：不支持区分下列动态类型：队列、动态阵列、关联阵列和类等。因此，等待动态类型更新的块可能无法正常工作。例如：

```
module top();
int c[$];
event e1;
initial
begin
    c[0] = 10;
    for(int i = 0; i <= 10; i++)
    begin
        c = {i, c};
        -> e1;
        #5;
    end
end
always@(*) $display($time, " trying to read sensitivity on dynamic type :
%d", c[0]);
// this won't work as sensitivity on dynamic type is not supported
always @(e1) $display($time, " coming from event sensitivity : %d",
c[0]); // this we
can do as WA
always_comb if(c.size() > 0) $display($time, " Coming from size
sensitivity : %d",
c[0]); // sensitivity on size works
```

通用验证方法论支持

AMD Vivado™ 集成设计环境支持在 Vivado 仿真器 (XSim) 内使用通用验证方法论 (UVM)。Vivado 提供了预编译的 UVM v1.2 库。如果您当前通过 Vivado 运行设计，则无需进行任何设置。但如果您当前运行独立 Vivado 仿真器，您需要将 `-L uvm` 传递给 `xvlog` 和 `xelab` 命令。

默认情况下，Vivado 仿真器支持 UVM v1.2。如果您要使用 UVM v1.1，则需要将 `-uvm_version 1.1` 传递给 `xvlog` 和 `xelab` 命令。如果您通过 Vivado 集成设计环境来使用 UVM，则请设置以下属性：

```
set_property -name {xsim.compile.xvlog.more_options} -value {-uvm_version
1.1} -objects [get_filesets sim_1] set_property -name
{xsim.elaborate.xelab.more_options} -value {-uvm_version 1.1} -objects
[get_filesets sim_1]
```

您也可以在 Vivado 集成设计环境 (IDE) 中使用仿真设置下的“Compilation and Elaboration”（编译和细化）选项卡来设置这些属性。如需了解更多信息，请参阅 使用仿真设置。

AMD

附录 D

Vivado 仿真器中的 VHDL 2008 支持

简介

AMD Vivado™ 仿真器支持 VHDL 2008 (IEEE 1076-2008) 的子集。在 VHDL 2008 (IEEE1076-2008) 的受支持的功能特性中提供了完整列表。

编译和仿真

Vivado 仿真器可执行文件 `xvhdl` 用于将 VHDL 设计单元转换为解析器转储文件 (`.vdb`)。默认情况下，Vivado 仿真器使用混合 93 与 2008 标准 (STD) 和 IEEE 封装来允许自由混用 93 和 2008 功能特性。如果您想强制仅使用 VHDL-93 标准 (STD) 和 IEEE 封装，请向 xvhdl 传递 `-93_mode`。要仅限使用 VHDL 2008 模式来编译文件，需要向 xvhdl 传递 `-2008` 开关。

例如，要以 VHDL-2008 编译称为 top.vhdl 的设计，可使用如下命令行：

```
xvhdl -2008 -work mywork top.vhdl
```

Vivado 仿真器可执行文件 `xelab` 用于细化设计，并为仿真生成可执行镜像。

xelab 可执行以下任一操作：

- 对 `xvhdl` 生成的解析器转储文件执行细化
- 直接使用 vhdl 源文件。

无需开关即可对 xvhdl 所生成的解析器转储文件执行细化。您可将 `-vhdl2008` 传递给 xelab 以便直接使用 vhdl 源文件。

示例 1：

```
xelab top -s mysim; xsim mysim -R
```

示例 2：

```
xelab -vhdl2008 top.vhdl top -s mysim; xsim mysim -R
```

您无需在命令行中为 `xvhdl` 和 `xelab` 指定 VHDL 文件，可改为使用 .prj 文件。如有两个文件用于设计，分别名为 `top.vhdl`（2008 模式）和 `bot.vhdl`（93 模式），则可按如下所示方式创建名为 `example.prj` 的工程文件：

```
vhdl xil_defaultlib bot.vhdl
```

```
vhdl2008 xil_defaultlib top.vhdl
```

在工程文件内，每一行均以文件的语言类型开头，后接库名称（如，`xil_defaultlib`）以及一个或多个文件名（含一个空格分隔符）。对于 VHDL 93，请使用 `vhdl` 作为语言类型。对于 VHDL 2008，则改为使用 `vhdl2008`。

.prj 文件可按如下示例所示方式来使用：

```
xelab -prj example.prj xil_defaultlib.top -s mysim; xsim mysim -R
```

或者，要混用 VHDL 93 和 VHDL 2008 设计单元，请单独编译文件，并将相应的语言模式指定为 `xvhdl`。然后，在设计顶层执行细化。例如，如果在 `bot.vhdl` 文件内有一个名为 bot 的 VHDL 93 模块，在文件 `top.vhdl` 内有一个名为 top 的 VHDL-2008 模块，可按如下示例所示方式对这两个文件进行编译：

```
xvhdl bot.vhdl
xvhdl -2008 top.vhdl
xelab -debug typical top -s mysim
```

xelab 生成可执行文件后，您即可照常运行仿真。

示例 1：

```
xsim mysim -gui
```

示例 2：

```
xsim mysim -R
```

定点包和浮点包

Vivado 仿真器使用的定点包和浮点包是 VHDL-2008 中引入的增强型新 IEEE 标准包。如果您当前使用的是 VHDL-93 标准定点包或浮点包，这些包在 Vivado 综合中可能有效。但您必须编辑自己的 VHDL 源文件才能执行仿真。

例如，如果您在 Vivado 综合中为定点包使用以下语法：

```
library ieee;
use ieee.fixed_pkg.all;
```

在 VHDL-2008 中请将此更改为如下语法，以供在 Vivado 仿真器内使用：

```
library ieee_proposed;
use ieee_proposed.fixed_pkg.all;
```

如需了解有关 Vivado 综合中的定点包和浮点包的更多信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的“定点支持”。

类似更改也适用于浮点包。

受支持的功能特性

以下是受支持的 VHDL 2008 (IEEE1076-2008) 功能特性：

- 匹配关系运算符。
- 最大运算符和最小运算符。
- 移位运算符（rol、ror、sll、srl、sla 和 sra）。
- 一元逻辑缩位运算符。
- 混用阵列与标量逻辑运算符。
- If-else-if 和 case generate。
- 顺序赋值。
- Case? 语句。
- Select? 语句。
- 未约束的元素类型。
- boolean_vector 和 integer_vector 阵列类型。
- 读取输出端口。
- 端口映射中的表达式。
- 进程（所有）语句。
- 引用泛型列表中的泛型。
- 实体中的泛型类型。
- 为函数返回值放宽返回规则。
- 给全局静态表达式和局部静态表达式增加扩展。
- 范围边界内的静态范围和整数表达式。
- 块注释。
- 上下文声明。
- 聚合中的阵列分片。
- 受保护的类型。
- 用于信号的外部名称。
- 物理类型中的 MOD 和 REF 运算符。

注释：如需了解有关这些功能特性的详细信息，请参阅《Vivado Design Suite 用户指南：综合》(UG901) 中的“受支持的 VHDL-2008 功能特性”部分。

AMD

附录 E

Vivado 仿真器中的直接编程接口 (DPI)

简介

您可使用 SystemVerilog 直接编程接口 (DPI) 将 C 语言代码绑定到 SystemVerilog 代码。SystemVerilog 代码可使用 DPI 来调用 C 语言函数，该函数则可回调 SystemVerilog 任务或函数。AMD Vivado™ 仿真器支持使用所有构造作为 DPI 任务/函数，如下所述。

编译 C 语言代码

我们提供了一个新的编译器可执行 `xsc` 文件，用于将 C 语言代码转换为对象代码文件，并将多个对象代码文件链接到单个共享库内（Windows 上的 `.a`，Linux 上的 `.so`）。此 xsc 编译器可从 `<Vivado installation>/bin` 目录中获取。您可以使用 `-sv_lib` 将包含 C 语言代码的共享库传递到 Vivado 仿真器/细化器可执行文件。此 xsc 编译器的工作方式与 C 语言编译器（如 gcc）相同。此 xsc 编译器会执行以下操作：

- 调用 LLVM clang 编译器，将 C 语言代码转换为对象代码
- 调用 GNU 连接器，基于对应 C 语言文件的一个或多个对象文件来创建共享库（Windows 上的 `.a`，Linux 上的 `.so`）

此 xsc 编译器生成的共享库会使用 xelab 中一个或多个新添加的开关与 Vivado 仿真器内核相链接，如下所示。这样由 xelab 创建的仿真快照就可以将已编译的 C 语言代码与已编译的 SystemVerilog 代码相连，并在 C 语言与 SystemVerilog 之间进行有效通信。

xsc 编译器

xsc 编译器可帮助您从一个或多个 C 语言文件创建共享库（Windows 上的 `.a` 或 Linux 上的 `.so`）。xelab 可用于将 xsc 生成的共享库绑定到设计的其余部分。您可使用以下进程创建共享库：

- 单步进程：不使用 `-compile` 或 `-shared/shared_systemc/static` 开关，直接将所有 C 语言文件传递到 `xsc`。
- 双步进程：

```
xsc -compile <C files>
xsc --shared or -shared_systemc or -static <object files>
```

用法

```
xsc [options] <files...>
```

Switches（开关）

您可为开关使用双连字符 (--) 或单连字符 (-)。

表 39：**XSC 编译器开关**

开关	描述
-compile [c]	仅从 C 语言源文件生成对象文件。不运行链接阶段。
-f [-file] <arg>	从指定文件读取其他选项。
-h [-help]	打印此帮助消息。
-i [-input_file] <arg>	用于编译或链接的输入文件（每个开关对应一个文件）列表。
-mt <arg> (=auto)	指定可并行运行的子编译作业数量。选项包括： · auto：自动 · n：其中 n 是大于 1 的整数 · off：关闭多线程 默认值：auto
-o [-output] <arg>	指定输出共享库的名称。仅适用于 --shared、--shared_systemc 和 --exe 选项。默认共享库是 <current_directory>/xsim.dir/work/xsc/dpi.so。
-work <arg>	指定用于放置输出（对象文件）的工作目录。 默认值：<current_directory>/xsim.dir/xsc
-v [-verbose] <arg>	指定打印消息的详细程度。 允许的值包括：0 和 1 默认值：0
-gcc_compile_options <arg>	向编译器提供一个额外选项。您可使用多个 -gcc_compile_options 开关。
-gcc_link_options <arg>	向连接器提供一个额外选项。您可使用多个 -gcc_link_options 开关。
-shared	仅运行链接阶段，以从对象文件生成共享库 (.so)。
-gcc_path	打印内部使用的 C 语言编译器路径。
-lib <arg>	指定读取的逻辑库目录。默认设为 <current_directory>/xsim.dir/xs。
-cppversion <arg>	设置 CPP 版本。当前支持 CPP 11 和 14。默认值为 11。
--shared_systemc	仅运行链接阶段，以从对象文件生成共享库 (.dll) 供 SystemC 使用。
--static	仅运行链接阶段，以从对象文件生成静态库 (.a) 供 SystemC 使用。
--exe	为独立 SystemC 创建可执行文件。
--version	打印当前使用的 Vivado 仿真器 xsc 的版本。
--debug	调试 SystemC 模块。该选项仅在搭配 -exe 选项使用时才有效，否则会被忽略。
--print_gcc_version	打印内部使用的 C 语言编译器版本。

示例

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
xsc -compile function1.c function2.c -work abc
xsc -shared abc/function1.lnx64.o abc/function2.lnx64.o -work abc
```

注释：默认情况下，Linux 使用 `LD_LIBRARY_PATH` 搜索 DPI 库。因此，如果库名以 `lib*` 开头，请向 Linux 上的 xelab 提供 `-dpi_absolute` 标志。

注释：您可向编译器使用 `-additional_option` 来传递额外开关。

- 示例：

  ```
  xsc t1.c --additional_option "-I<path>"
  ```

- 传递多条路径的示例：

  ```
  xsc t1.c --additional_option "-I<path>" --additional_option "-I<path>"
  ```

使用 xelab 将已编译的 C 语言代码绑定到 SystemVerilog

可供 xelab 用于将已编译的 C 语言代码绑定到 SystemVerilog 的 DPI 相关开关如下所示：

表 40：**适用于 xelab 的 DPI 相关开关**

开关	描述
`-sv_root arg`	应搜索与 DPI 共享库相关的根目录。（默认值：`<current_directory>/xsim.dir/xsc`）
`-sv_lib arg`	DPI 共享库名称，不含文件扩展名，用于定义 SystemVerilog 中导入的 C 语言函数。
`-sv_liblist arg`	指向 DPI 共享库的启动文件。
`-dpiheader arg`	生成 DPI C 语言头文件（包含导入和导出的函数的 C 语言声明）。
-dpi_absolute	在 Linux 上为 DPI 库使用绝对路径代替 `LD_LIBRARY_PATH`，此类库的格式为 `lib<libname>.so`。
-dpi_stacksize arg	用户为 DPI 任务定义的栈大小。

如需了解有关 `r-sv_liblist arg` 的更多信息，请参阅《适用于 SystemVerilog 的 IEEE 标准 - 统一硬件设计、规范和验证语言》附录 J.4.1，第 1228 页。

C 语言和 SystemVerilog 边界上允许的数据类型

SystemVerilog 的 IEEE 标准仅允许在 C 语言和 SystemVerilog 边界上使用一小部分 C 语言和 SystemVerilog 数据类型。以下提供了：

1. 有关 Vivado 仿真器中受支持的数据类型的详细信息。
2. C 语言与 SystemVerilog 数据类型之间的映射描述。

受支持的数据类型

下表描述了 C 语言和 SystemVerilog 边界上允许的数据类型，以及 SystemVerilog 与 C 语言之间的双向数据类型映射。

表 41：**C-SystemVerilog 边界上允许的数据类型**

SystemVerilog	C	受支持	注释
byte	char	支持	无
shortint	short int	支持	无
int	int	支持	无
longint	long long	支持	无
real	double	支持	无
shortreal	float	支持	无
chandle	void *	支持	无
string	const char*	支持	无
bit	unsigned char	支持	sv_0 和 sv_1
在 C 语言侧，可通过 svdpi.h 来使用			
logic 和 reg	unsigned char	支持	sv_0、sv_1、sv_z 和 sv_x:
bit 阵列（打包）	svBitVecVal	支持	在 svdpi.h 中已定义
logic/reg 的阵列（打包）	svLogicVecVal	支持	在 svdpi.h 中已定义
enum	底层 enum 类型	支持	无
已打包的 struct（结构体）和 union（联合体）	作为阵列来传递	支持	无
已解包的 bit 和 logic 阵列	作为阵列来传递	支持	C 语言可以调用 SystemVerilog
已解包的 struct	作为 struct 来传递	支持	无
已解包的 union	作为 struct 来传递	不支持	无
打开的阵列	svOpenArrayHandle	支持	无

要生成 C 语言头文件以提供有关 SystemVerilog 数据类型如何映射到 C 语言数据类型的详细信息：请将 `-dpiheader <file name>` 参数传递给 `xelab`。如需了解有关数据类型映射的其他详细信息，请参阅 SystemVerilog 的 IEEE 标准。

适用于用户定义的类型的映射

枚举

您可根据 `enum` 的基本类型，定义枚举类型 (`enum`) 以便转换为等效的 SystemVerilog `svLogicVecVal` 类型或 `svBitVecVal` 类型。对于枚举阵列，会创建等效的 SystemVerilog 阵列。

示例

- SystemVerilog 类型：：

```
typedef enum reg [3:0] { a = 0, b = 1, c} eType;
eType e;
eType e1[4:3];
typedef enum bit { a = 0, b = 1} eTypeBit;
eTypeBit  e3;
eTypeBit  e4[3:1] ;
```

- C 语言类型：

```
svLogicVecVal e[SV_PACKED_DATA_NELEMS(4)];
svLogicVecVal e1[2][SV_PACKED_DATA_NELEMS(4)];
svBit e3;
svBit e4[3];
```

提示：C 语言实参类型取决于 `enum` 的基本类型和方向。

打包结构体/联合体

使用打包结构体或联合体类型时，会在 DPI C 语言侧创建等效的 SystemVerilog 类型 `svLogicVecVal` 或 `svBitVecVal`。

示例

- SystemVerilog 类型：

```
typedef struct packed {
        int i;
        bit b;
        reg [3:0]r;
        logic [2:0] [4:8][9:1] l;
    } sType;
    sType c_obj;
   sType [3:2] c_obj1[5];
```

- C 语言类型：

```
svLogicVecVal c_obj[SV_PACKED_DATA_NELEMS(172)];
svLogicVecVal c_obj1[5][SV_PACKED_DATA_NELEMS(344)];
```

打包和解包的阵列均表示为 `svLogicVecVal` 阵列或 `svBitVecVal` 阵列。

解包结构体

在 C 语言侧创建等效的解包类型，其中所有成员都会转换为等效的 C 语言表示法。

示例

- SystemVerilog 类型：

```
typedef struct {
        int i;
        bit b;
        reg r[3:0];
        logic [2:0] l[4:8][9:1];
    } sType;
```

- C 语言类型：

```
typedef struct {
    int i;
    svBit b;
    svLogic r[4];
    svLogicVecVal l[5][9][SV_PACKED_DATA_NELEMS(3)];
} sType;
```

svdpi.h 函数支持

在以下目录中提供了 `svdpi.h` 头文件：`<vivado installation>/data/xsim/include`。

支持以下 `svdpi.h` 函数：

```
svBit svGetBitselBit(const svBitVecVal* s, int i);
svLogic svGetBitselLogic(const svLogicVecVal* s, int i);
void svPutBitselBit(svBitVecVal* d, int i, svBit s);
void svPutBitselLogic(svLogicVecVal* d, int i, svLogic s);
void svGetPartselBit(svBitVecVal* d, const svBitVecVal* s, int i, int w);
void svGetPartselLogic(svLogicVecVal* d, const svLogicVecVal* s, int i, int
w);
void svPutPartselBit(svBitVecVal* d, const svBitVecVal s, int i, int w);
void svPutPartselLogic(svLogicVecVal* d, const svLogicVecVal s, int i, int
w);
const char* svDpiVersion();
    svScope svGetScope();
    svScope svSetScope(const svScope scope);
    const char* svGetNameFromScope(const svScope);
    int svPutUserData(const svScope scope, void*userKey, void* userData);
    void* svGetUserData(const svScope scope, void* userKey);
```

DPI 中的开放阵列

在 SystemVerilog 中声明导入函数时，可以指定形参作为开放阵列。将阵列形参的某些维度指定为空白（开放），即可允许传递不同大小的实参，这有助于得到更为通用的 C 语言代码。在 C 语言侧，开放阵列表示为 `SVOpenArrayHandle`。将此句柄传递给所提供的函数即可查询开放阵列的信息。例如，开放维度的大小以及对实际数据的访问权限。

声明

在 SystemVerilog 代码中，开放阵列只能出现在导入函数/任务声明中。使维度保持开放时，您必须指定开放阵列，空白维度的大小根据实参来确定。

示例

SystemVerilog 函数声明：

```
import "DPI-C" function int myFunction1(input bit[] v);
import "DPI-C" function void myFunction2(input int v1[], input int v2[],
output int
v3[]);
```

在 C 语言侧，仅限该句柄和提供的 API 才能访问开放阵列：

```
int myFunction1(const SVOpenArrayHandle v);
void myFunction2(const SVOpenArrayHandle v1, const SVOpenArrayHandle v2,
const
SVOpenArrayHandle v3);
```

svdpi.h 支持

在 svdpi.h 中支持下列开放阵列相关的函数：

```
int svLeft(const svOpenArrayHandle h, int d);
int svRight(const svOpenArrayHandle h, int d);
int svLow(const svOpenArrayHandle h, int d);
int svHigh(const svOpenArrayHandle h, int d);
int svIncrement(const svOpenArrayHandle h, int d);
int svSize(const svOpenArrayHandle h, int d);
int svDimensions(const svOpenArrayHandle h);
void *svGetArrayPtr(const svOpenArrayHandle);
int svSizeOfArray(const svOpenArrayHandle);
void *svGetArrElemPtr(const svOpenArrayHandle, int indx1, ...);
void *svGetArrElemPtr1(const svOpenArrayHandle, int indx1);
void *svGetArrElemPtr2(const svOpenArrayHandle, int indx1, int indx2);
void *svGetArrElemPtr3(const svOpenArrayHandle, int indx1, int indx2,
int indx3);
void svPutBitArrElemVecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, ...);
void svPutBitArrElem1VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1);
void svPutBitArrElem2VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, int indx2);
void svPutBitArrElem3VecVal(const svOpenArrayHandle d, const svBitVecVal* s,
int indx1, int indx2, int indx3);
void svPutLogicArrElemVecVal(const svOpenArrayHandle d, const svLogicVecVal*
s, int indx1, ...);
void svPutLogicArrElem1VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1);
void svPutLogicArrElem2VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1, int indx2);
void svPutLogicArrElem3VecVal(const svOpenArrayHandle d, const
svLogicVecVal*
s, int indx1, int indx2, int indx3);
void svGetBitArrElemVecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, ...);
void svGetBitArrElem1VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1);
void svGetBitArrElem2VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2);
void svGetBitArrElem3VecVal(svBitVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2, int indx3);
```

```
void svGetLogicArrElemVecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, ...);
void svGetLogicArrElem1VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int
indx1);
void svGetLogicArrElem2VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2);
void svGetLogicArrElem3VecVal(svLogicVecVal* d, const svOpenArrayHandle s,
int indx1, int indx2, int indx3);
svBit svGetBitArrElem(const svOpenArrayHandle s, int indx1, ...);
svBit svGetBitArrElem1(const svOpenArrayHandle s, int indx1);
svBit svGetBitArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svBit svGetBitArrElem3(const svOpenArrayHandle s, int indx1, int indx2, int
indx3);
svLogic svGetLogicArrElem(const svOpenArrayHandle s, int indx1, ...);
svLogic svGetLogicArrElem1(const svOpenArrayHandle s, int indx1);
svLogic svGetLogicArrElem2(const svOpenArrayHandle s, int indx1, int indx2);
svLogic svGetLogicArrElem3(const svOpenArrayHandle s, int indx1, int indx2,
int
indx3);
void svPutLogicArrElem(const svOpenArrayHandle d, svLogic value, int
indx1, ...);
void svPutLogicArrElem1(const svOpenArrayHandle d, svLogic value, int
indx1);
void svPutLogicArrElem2(const svOpenArrayHandle d, svLogic value, int
indx1, int
indx2);
void svPutLogicArrElem3(const svOpenArrayHandle d, svLogic value, int indx1,
int indx2, int indx3);
void svPutBitArrElem(const svOpenArrayHandle d, svBit value, int
indx1, ...);
void svPutBitArrElem1(const svOpenArrayHandle d, svBit value, int indx1);
void svPutBitArrElem2(const svOpenArrayHandle d, svBit value, int indx1,
int indx2);
void svPutBitArrElem3(const svOpenArrayHandle d, svBit value, int indx1,
int indx2, int indx3);
```

使用示例 - SystemVerilog 代码

```
module m();
import "DPI-C" function void myFunction1(input int v[]);
int arr[4];
int dynArr[];
initial begin
arr = '{4, 5, 6, 7};
myFunction1(arr);
dynArr = new[6];
dynArr = '{8, 9, 10, 11, 12, 13};
myFunction1(dynArr);
end
endmodule
C code:
#include "svdpi.h"
void myFunction1(const svOpenArrayHandle v)
{
int l1 = svLow(v, 1);
int h1 = svHigh(v, 1);
for(int i = l1; i<= h1; i++) {
    printf("\t%d", *((char*)svGetArrElemPtr1(v, i)));
}
printf("\n");
                }
```

示例

注释：以下所有示例都会打印 `PASSED` 表示运行成功。

示例包括：

- 使用 -sv_lib、-sv_liblist 和 -sv_root 的导入示例：函数导入示例，用于演示使用 `-sv_lib`、`-sv_liblist` 和 `-sv_root` 选项的不同方法。
- 含输出的函数：具有输出实参的函数。
- 简单的导入导出流程（演示 xelab -dpiheader 流程）：显示简单的导入 > 导出流程（演示 `xelab -dpiheader <filename>` 流程）。

使用 -sv_lib、-sv_liblist 和 -sv_root 的导入示例

代码

假定存在：

- 两个文件，每个文件包含一个 C 语言函数
- 一个 SystemVerilog 文件，它使用下列函数：
  - function1.c
  - function2.c
  - file.sv

function1.c

```
#include "svdpi.h"
DPI_DLLESPEC
int myFunction1()
{
    return 5;
}
```

function2.c

```
#include <svdpi.h>
DPI_DLLESPEC
int myFunction2()
{
    return 10;
}
```

file.sv

```
module m();
import "DPI-C" pure function int myFunction1 ();
import "DPI-C" pure function int myFunction2 ();
integer i, j;
initial
begin
#1;
```

```
  i = myFunction1();
  j = myFunction2();
  $display(i, j);
  if( i == 5 && j == 10)
    $display("PASSED");
  else
    $display("FAILED");
end
endmodule
```

用法

将 C 语言文件编译和链接到 Vivado 仿真器内的方法如下所述。

单步骤流程（最简单的流程）

```
xsc function1.c function2.c
xelab -svlog file.sv -sv_lib dpi
```

流程描述：

xsc 编译器会编译并链接 C 语言代码以创建共享库 `xsim.dir/xsc/dpi.so`，xelab 则通过 `-sv_lib` 开关来引用此共享库。

双步骤流程

```
xsc -compile function1.c function2.c -work abc
xsc -shared/-shared_systemc abc/function1.lnx64.o abc/function2.lnx64.o -
work abc
xelab -svlog file.sv -sv_root abc -sv_lib dpi -R
```

流程描述：

- 将两个 C 语言文件编译到工作目录 `abc` 中的对应对象代码内。
- 将这两个文件链接在一起，以创建共享库 `dpi.so`。
- 确保通过 `-sv_root` 开关从工作库 `abc` 中提取此库。

> **提示：** `-sv_root` 会指定查找通过 `-sv_lib` 开关指定的共享库的位置。在 Linux 上，如果未指定 `-sv_root` 并且 DPI 库名称带有前缀 `lib` 和后缀 `.so`，那么请为共享库位置使用 LD_LIBRARY_PATH 环境变量。

双步骤流程（与上述流程相同但附加多个选项）

```
xsc -compile function1.c function2.c -work "abc" -v 1
xsc -shared/-shared_systemc "abc/function1.lnx64.o" "abc/function2.lnx64.o"
-work "abc" -o final -v 1
xelab -svlog file.sv -sv_root "abc" -sv_lib final -R
```

流程描述：

如果您要自行进行编译和链接，可以使用 `-verbose` 开关来查看用于调用编译器的路径和选项。随后，您可根据自己的需求对此路径和选项进行调整。在以上示例中，创建了一个专用共享库 `final`。此示例还演示了文件路径中空格的用法。

含输出的函数

代码

file.sv

```
/*- - - -*/
package pack1;
import "DPI-C" function int myFunction1(input int v, output int o);
import "DPI-C" function void myFunction2 (input int v1, input int v2,
output int o);
endpackage
/*-- ---*/
module m();
int i, j;
int o1 ,o2, o3;
initial
begin
#1;
j = 10;
o3 =pack1:: myFunction1(j, o1);//should be 10/2 = 5
pack1::myFunction2(j, 2+3, o2); // 5 += 10 + 2+3
$display(o1, o2);
if( o1 == 5 && o2 == 15)
$display("PASSED");
else
$display("FAILED");
end
endmodule
```

function.c

```
#include "svdpi.h"
DPI_DLLESPEC
int myFunction1(int j, int* o)
{
*o = j /2;
return 0;
}
DPI_DLLESPEC
void myFunction2(int i, int j, int* o)
{
*o = i+j;
return;
}
```

run.ksh

```
xsc function.c
xelab -vlog file.sv -sv -sv_lib dpi -R
```

简单的导入导出流程（演示 xelab -dpiheader 流程）

在此流程中：

1. 运行 xelab 搭配 `-dpiheader` 开关来创建头文件 `file.h`。

2. 随后，`file.c` 中的代码就会包含 xelab 生成的头文件 (`file.h`)，此文件列在末尾。
3. 像之前一样，在 `file.c` 和 `test.sv` 中编译代码以生成仿真可执行文件。

file.c

```
#include "file.h"
/* NOTE: This file is generated by xelab -dpiheader <filename> flow */
int cfunc (int a, int b) {
//Call the function exported from SV.
return c_exported_func (a,b);
}
```

test.sv

```
module m();
export "DPI-C" c_exported_func = function func;
import "DPI-C" pure function int cfunc (input int a ,b);
/*This function can be called from both SV or C side. */
function int func(input int x, y);
begin
func = x + y;
end
endfunction
int z;
initial
begin
#5;
z = cfunc(2, 3);
if(z == 5)
$display("PASSED");
else
$display("FAILED");
end
endmodule
```

run.ksh

```
xelab -dpiheader file.h -svlog test.sv
xsc file.c
xelab -svlog test.sv -sv_lib dpi -R
file.h
/**********************************************************************/
/* ____ ____ */
/* / /\/ / */
/* /___/ \ / */
/* \ \ \/ */
/* \ \ Copyright (c) 2003-2013 Xilinx, Inc. */
/* / / All Right Reserved. */
/* /---/ /\ */
/* \ \ / \ */
/* \___\/\___\ */
/**********************************************************************/
/* NOTE: DO NOT EDIT. AUTOMATICALLY GENERATED FILE. CHANGES WILL BE LOST. */
#ifndef DPI_H
#define DPI_H
#ifdef __cplusplus
#define DPI_LINKER_DECL extern "C"
#else
#define DPI_LINKER_DECL
```

```
#endif
#include "svdpi.h"
/* Exported (from SV) function */
DPI_LINKER_DECL DPI_DLLISPEC
int c_exported_func(
int x, int y);
/* Imported (by SV) function */
DPI_LINKER_DECL DPI_DLLESPEC
int cfunc(
int a, int b);
#endif
```

Vivado Design Suite 随附的 DPI 示例

Vivado Design Suite 随附了 2 个示例，可帮助您了解如何在 Vivado 仿真器内使用 DPI。您可在安装目录中找到这些示例：`<vivado installation dir>/examples/xsim/systemverilog/dpi`。每个示例都包含可帮助您快速入门的 `README` 文件。这些示例包括：

- `simple_import`：简单导入纯函数
- `simple_export`：简单导出纯函数

提示：当仅根据函数的输入值来计算其返回值时，该函数称为“pure function”（纯函数）。

附录 F

Vivado IDE 中的 SystemC 支持

AMD Vivado™ Design Suite 以一组文件和库的形式提供了仿真模型。仿真库包含器件以及 IP 行为和时序模型。编译后的库可供多个设计工程使用。执行设计仿真前，您必须先编译这些文件，然后再通过名为 `compile_simlib` 的实用工具来为目标仿真器编译仿真模型。该实用工具可从 Vivado IDE 调用，或者也可以从 Tcl 控制台执行。

对于 SystemC 仿真验证，在 C/C++/SystemC 内提供仿真模型。Vivado Design Suite 提供了两组仿真模型：

- 受保护的模型
- 不受保护的模型

注释：使用 Vivado 仿真器时，无需编译仿真库。一般情况下，库必须使用新软件版本来编译或重新编译，以更新仿真模型和支持新版本的仿真器和 GCC。

选择仿真模型类型

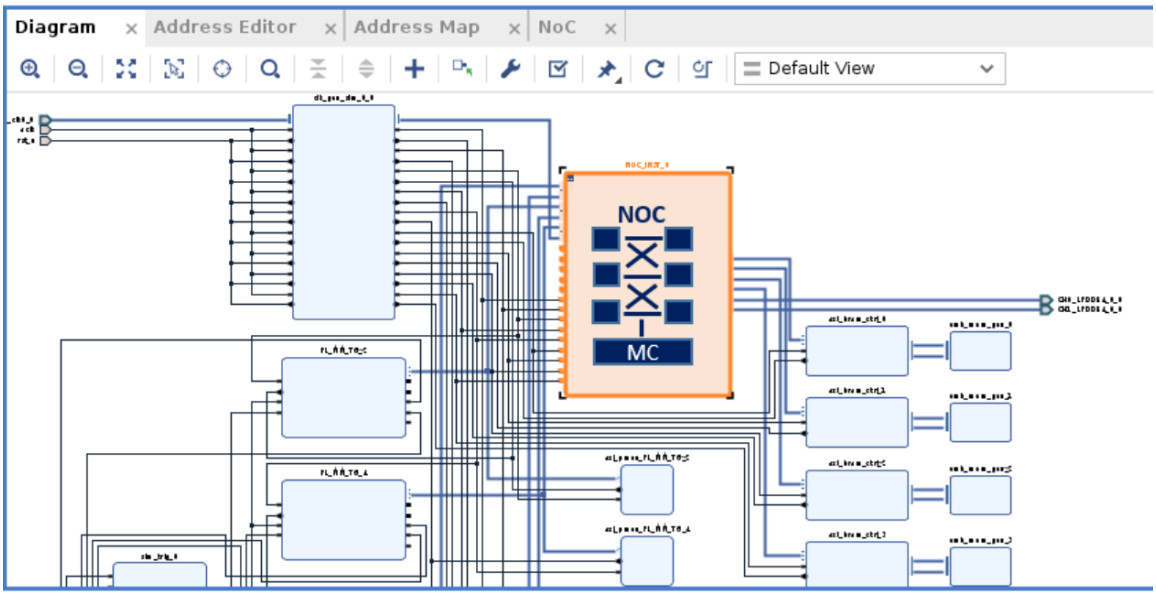
为缩短仿真运行时，AMD 提供了传输事务级仿真模型 (tlm) 供某些 IP 使用，如 Control, Interfaces and Processing System、SmartConnect、NoC 和 AI 引擎。您可使用工程属性 (PREFERRED_SIM_MODEL) 或 IP 属性 (SELECTED_SIM_MODEL) 来为自己的 IP 选择任一受支持的仿真模型。受支持的仿真模型属性如下：

- ALLOWED_SIM_MODELS：这是只读属性。它描述了可供特定 IP 使用的不同仿真模型类型，如 `rtl`、`tlm`、`tlm_dpi` 和 `dpi`。
- SELECTED_SIM_MODEL：这是 IP 级设置，允许您从 ALLOWED_SIM_MODELS 中选择并设置任一仿真模型。
- PREFFERED_SIM_MODEL：这是工程级设置，允许您为工程设置默认仿真模型。此项是工程中存在的所有 IP 的通用设置。

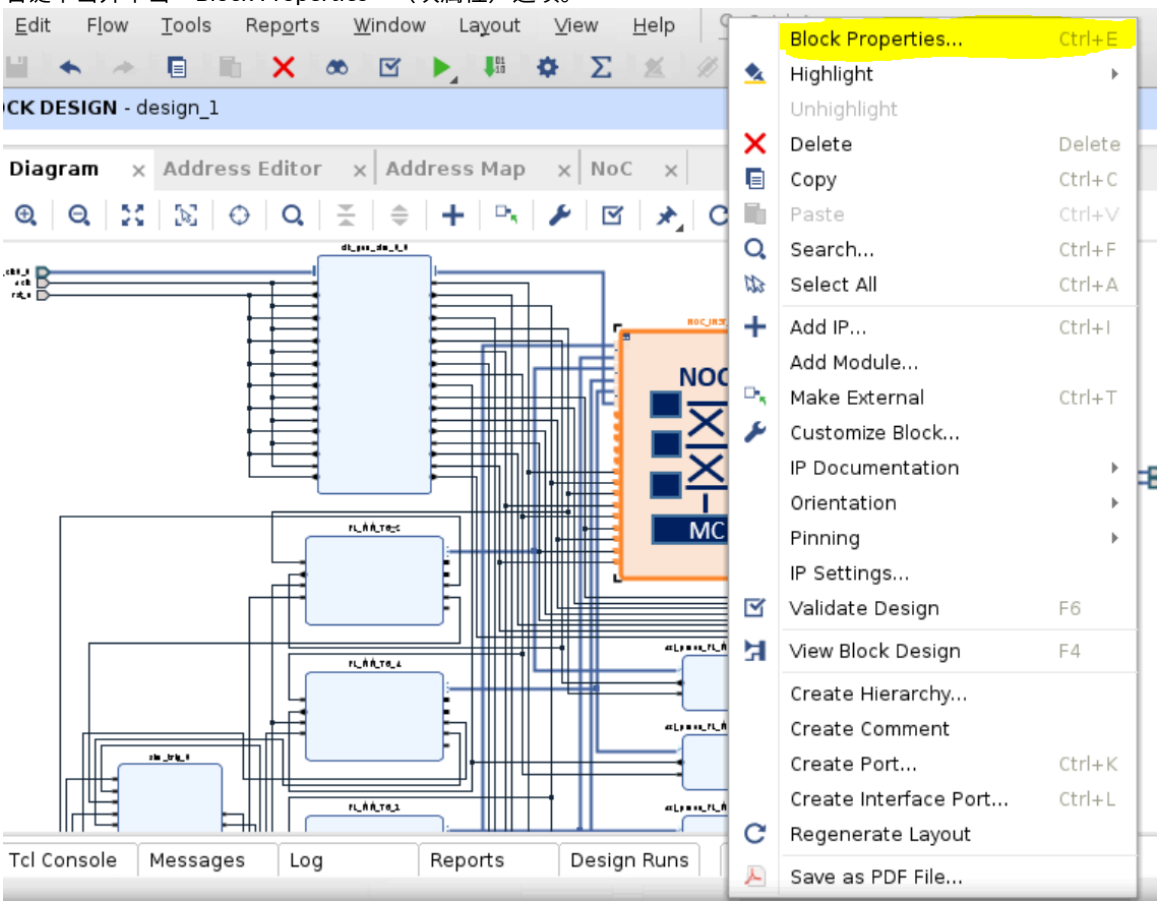
使用 SELECTED_SIM_MODEL IP 属性

执行以下步骤，使用 SELECTED_SIM_MODEL 更改您的 IP 的仿真模型：

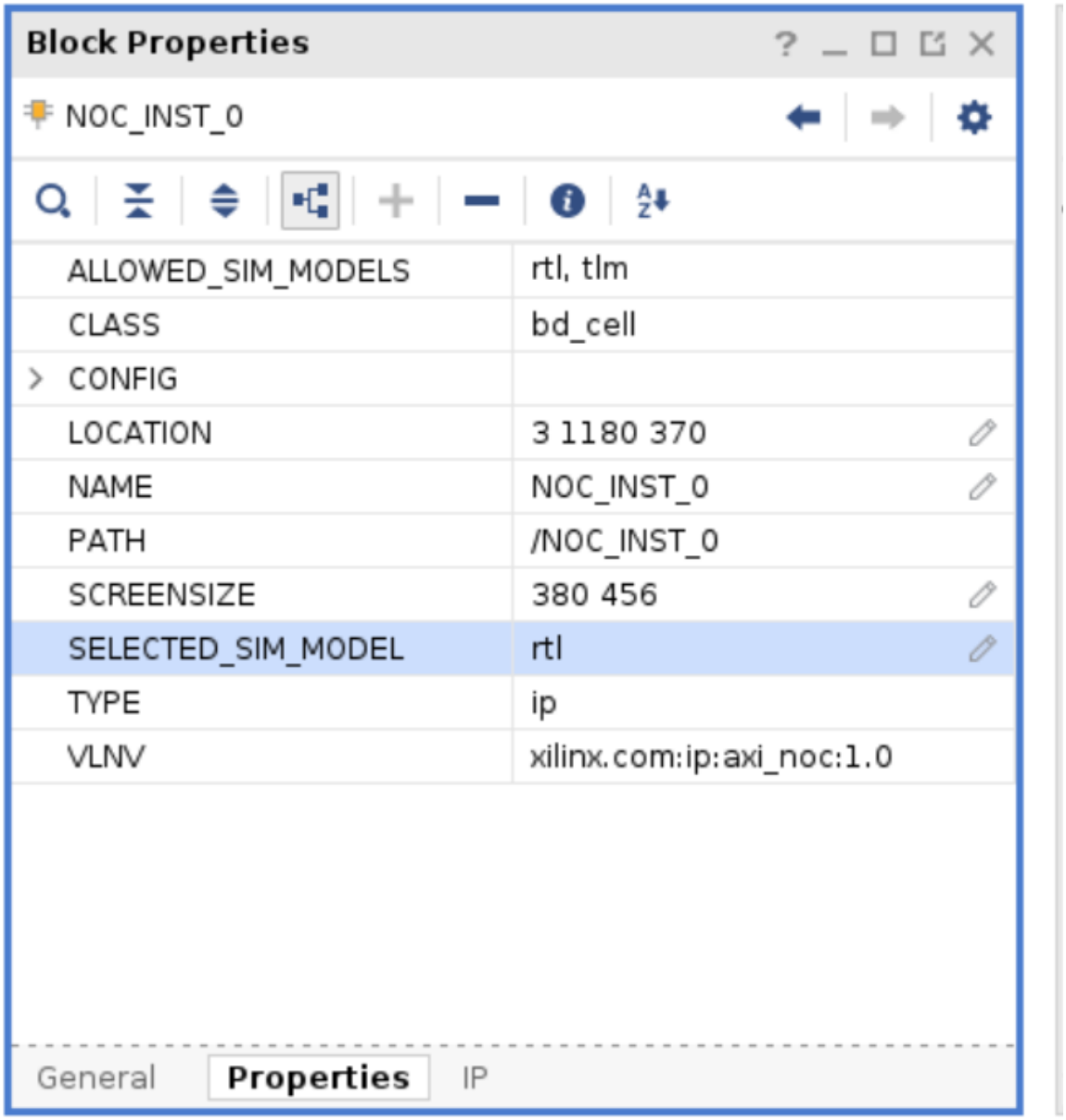
1. 在 Flow Navigator 中，单击“Open Block Design”（打开块设计）以打开块设计。
2. 从块设计中选择所需 IP。



3. 右键单击并单击“Block Properties”（块属性）选项。



4. 从 IP 的“Block Properties”窗口更改“SELECTED_SIM_MODEL”选项。例如，从 rtl 更改为 tlm。



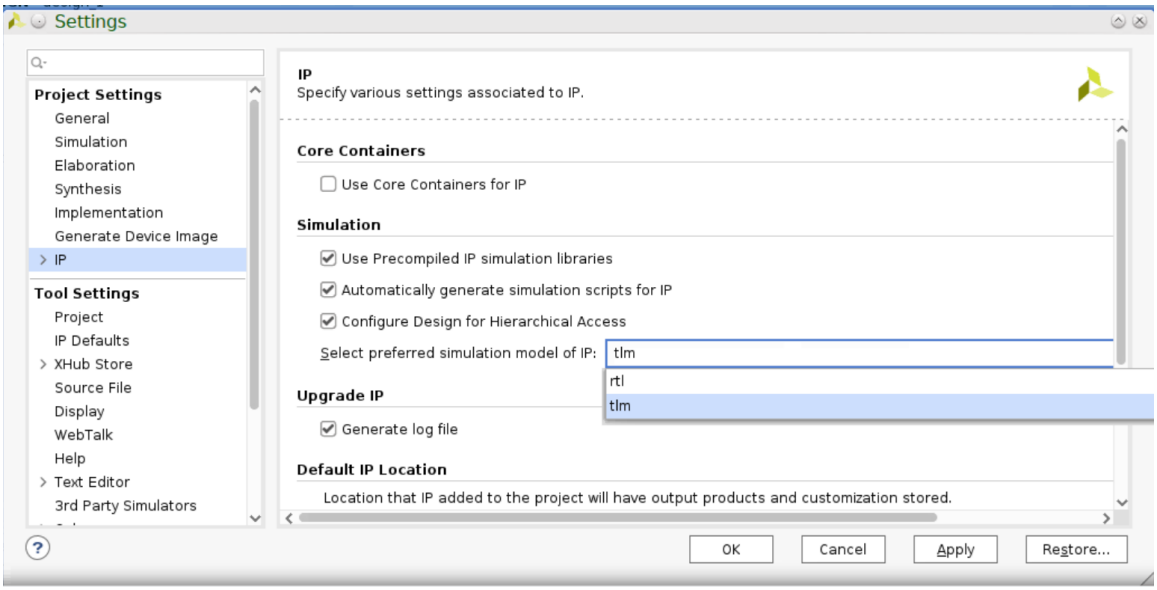
以下 Tcl 命令等效于更改 SELECTED_SIM_MODEL：

```
set_property SELECTED_SIM_MODEL tlm [get_bd_cells /NOC_INST_0]
```

使用 PREFERRED_SIM_MODEL 工程属性

执行以下步骤，使用 PREFERRED_SIM_MODEL 更改您的 IP 的仿真模型：

1. 单击“Settings”（设置）对话框中的“IP”选项。
2. 从“Select preferred simulation model of IP”（选择首选 IP 仿真模型）下拉菜单中选择“tlm”。



以下 Tcl 命令等效于更改 PREFERRED_SIM_MODEL：

```
set_property preferred_sim_model tlm [current_project]
```

注释：将 PREFFERED_SIM_MODEL 设置为 tlm 会将所有 IP 的 SELECTED_SIM_MODEL 都设置为 tlm（不支持 tlm 的 IP 除外）。

受保护的模型

受保护的模型以专为相应仿真器而构建的共享库的形式进行预编译和释放。此共享库封装为 AMD Vivado™ 安装的一部分，这些模型在细化期间基于设计配置来加以绑定。在 Vivado 安装中会一并交付以下两个受保护的模型：

- AI 引擎
- 片上网络 (NoC)

这些模型均以共享库的形式存在于以下安装路径中：

```
<Vivado-install-path>/data/simmodels/<simulator>/<simulator_version>/
<os_type>/<gcc_version>/systemc/protected
```

- Vivado 仿真器：`<Vivado-install-path>/data/simmodels/xsim/2024.1/lnx64/9.3.0/systemc/protected`
- Xcelium Simulator：`<Vivado-install-path>/data/simmodels/xcelium/23.09.001/lnx64/9.3.0/systemc/protected`
- Questa Advanced Simulator：`<Vivado-install-path>/data/simmodels/questa/2023.3/lnx64/7.4.0/systemc/protected`
- VCS 仿真器：`<Vivado-install-path>/data/simmodels/vcs/U-2023.03-SP2/lnx64/9.2.0/systemc/protected/`

- Riviera：`<Vivado-install-path>/data/simmodels/riviera/2023.04/lnx64/9.3.0/systemc/protected/`

注释：对于综合后仿真，不支持 Versal 模型（例如，AI 引擎和 NoC）。

不受保护的模型

不受保护的模型在安装中会作为源代码加以释放。您需使用 `compile_simlib` 实用工具来为目标仿真器编译模型。对于 AMD Vivado™ 仿真器，在编译其他库的 `<Vivado-install-path>/data/xsim` 标准文件夹内对这些不受保护的模型进行预编译。对于第三方仿真器，必须使用 `compile_simlib` 编译这些模型。在 Vivado 安装中会一并交付以下不受保护的模型：

- `aie_xtlm`
- `axi_tg_sc`
- `axis_dwidth_converter_sc`
- `axis_switch_sc`
- `common_cpp`
- `common_rpc`
- `debug_tcp_server`
- `emu_perf_common`
- `noc_sc`
- `pl_fileio`
- `remote_port_c`
- `remote_port_sc`
- `rwd_tlmmodel`
- `sim_ddr`
- `sim_qdma_cpp`
- `sim_qdma_sc`
- `sim_xdma_cpp`
- `sim_xdma_sc`
- `tlm_ext`
- `xtlm`
- `xtlm_ap_ctrl`
- `xtlm_ipc`
- `xtlm_simple_interconnect`
- `xtlm_trace_model`

这些仿真模型源文件所在安装路径如下所示：

```
<Vivado-install-path>/data/systemc/
```

使用 Vivado 进行 SystemC 仿真

运行 SystemC 仿真设计需要：

- 创建设计源文件
- 使用 `compile_simlib` 编译仿真模型
- 指定所需工具/设计设置

c/c++/SystemC 源文件可使用 GCC 来进行编译。每个仿真器支持的 GCC 版本都各不相同。如果设计包含 AMD 提供的 SystemC 模型，那么所使用的 GCC 版本应为受支持的版本。如果 GCC 版本发生更改，就需要重新编译设计。

SystemC 仿真支持使用的仿真器

以下是 AMD Vivado™ Design Suite 中的 SystemC 仿真支持使用的仿真器：

表 42：**SystemC 仿真支持使用的仿真器**

仿真器	版本	兼容的 GCC 版本	SystemC 编译器
AMD Vivado™ 仿真器	2024.1	9.3.0	XSC
Siemens EDA Questa Advanced Simulator	2023.3	7.4.0	SCCOM
Cadence Xcelium Parallel Simulator	23.09.001	9.3.0	XMSC
VCS	U-2023.03-SP2	9.2.0	SYSCAN
Riviera	2023.04	9.3.0	CCOMP

第三方工具的仿真器设置

表 43：**第三方工具的仿真器设置**

仿真器	Linux

Questa	`setenv MODEL_TECH <tool installation path>`
	`setenv LM_LICENSE_FILE <license file>`
	`setenv PATH ${MODEL_TECH}/bin:$PATH`

Xcelium	`setenv CDS_INST_DIR <xcelium_install_dir>`
	`setenv LD_LIBRARY_PATH $CDS_INST_DIR/tools/xcelium/lib:$LD_LIBRARY_PATH`
	`setenv PATH $CDS_INST_DIR/tools/xcelium/bin:$CDS_INST_DIR/tools/bin:$PATH`
	`setenv CDS_LICENSE_DIR <tool_license>`

表 43：**第三方工具的仿真器设置** (续)

仿真器	Linux

VCS	`setenv VCS_HOME <tool_install_path>`
	`setenv SYSTEMC_HOME $VCS_HOME/linux64/lib`
	`setenv LM_LICENSE_FILE <license file>`
	`setenv VG_GNU_PACKAGE /tools/installs/synopsys/vg_gnu/2020.12/linux`
	`setenv PATH ${VCS_HOME}/bin:${PATH}`
	`source $VG_GNU_PACKAGE/source_me[.sh|.csh]`

Riviera	`source <tool_install_dir>/etc/setenv`
	`source <tool_install_dir>etc/setgcc`
	`export ALDEC_LICENSE_FILE=<tool_license>`

注释：默认情况下，GCC 路径是根据 Questa 和 Xcelium 的工具安装位置自动确定的。

GCC 路径设置

下表描述了 `compile_simlib` 和 `launch_simulation` 的 GCC 可执行路径设置：

表 44：**GCC 路径设置**

命令	设置
`compile_simlib`	• 使用 `-gcc_exec_path` 开关指定 GCC 编译器安装路径。 • 否则，设置环境变量 `GCC_SIM_EXE_PATH <gcc_install_dir>`。
`launch_simulation`	• 使用 `-gcc_install_path` 开关指定 GCC 编译器安装路径。 • 否则，使用 `set_property simulator.<name>_gcc_install_dir <gcc_path> [current_project]` 设置属性。 • 否则，设置环境变量 `GCC_SIM_EXE_PATH <gcc_install_dir>`。

注释：如未找到这些建议设置，Vivado 会从 PATH 环境变量提取安装路径。并且，始终建议使用工具原生的 SystemC 编译器。

AMD

附录 G

适用于子设计的自动测试激励文件生成

从 2021.2 起，在 Vivado 仿真器 (XSim) 内引入了全新的方法论，以便为任意语言的子设计单元创建真实的功能测试激励文件。此方法论当前适用于 Verilog/VHDL/SystemVerilog 以及这些语言的混合设计。为使用此方法论，引入了 2 条全新 Tcl 命令。在后续章节中解释了如何将此新方法论应用于真实的设计。

generate_vcd_ports

`generate_vcd_port` 命令会打开 VCD 文件句柄，用于写入给定实例的端口活动。`create_testbench` Tcl 命令会读取此 VCD 文件，以读取端口活动用于写入激励源。该实例可通过如下方式选择：在 Vivado 仿真器 IDE 中的作用域窗口内选择，或者在从 Tcl 控制台执行此命令时通过指定指向该实例的分层路径来选择。该命令会创建 `dummports.vcd` 文件，为选定的实例作用域运行仿真时会填充此文件。

注释： Vivado 仿真器必须处于活动状态才能为所选实例生成此文件。

如果从 Vivado IDE 运行此命令，那么会在仿真运行目录中创建 `dumpports.vcd` 文件。如果从 Vivado 仿真器独立 GUI 运行此命令，则在当前目录的 `vcd2tb` 子目录内创建 `dumpports.vcd` 文件。`generate_vcd_port` 选项如下：

- -scope <arg>（必需）：指定实例作用域的层级名称。
- -quiet（可选）：以静默方式执行命令，不返回来自该命令的任何消息。此命令还会返回 `TCL_OK`，忽略执行期间遇到的所有错误。

  注释： 启动该命令时，会返回命令行上遇到的任何错误。仅捕获该命令内部发生的错误。
- -verbose（可选）：暂时覆盖所有消息限制，并返回来自该命令的所有消息。

  注释： 可使用 `set_msg_config` 命令定义消息限制。

以下命令示例会为 buf 类型的模块的 `/top/DUT/fifo/buf_1` 实例创建 VCD 文件、记录 2000 ns 内的波形活动，并关闭 VCD 文件句柄：

```
generate_vcd_ports {/top/DUT/fifo/buf_1}
run 2000ns
close_vcd -ports
```

create_testbench

为设计单元实例创建测试激励文件。此命令会为限定作用域的分层实例创建基于功能系统 Verilog 的测试激励文件。该测试激励文件包含选定实例的端口/信号规范、参数声明、仿真矢量包含文件以及模块例化作为受测设计 (DUT)。此命令允许您将测试激励文件添加到现有或新的仿真文件集，以便您可从中启动仿真。

注释：生成的测试激励文件与仿真器无关。

表 45：**create_testbench 命令选项**

选项	描述
`-name <arg>`	用于指定测试激励文件模块名称。默认名称为 test bench。
`-add_to_simset <arg>`	指定测试激励文件需添加到的仿真文件集名称。如果不指定此开关，那么该命令会将测试激励文件添加到当前活动的仿真文件集。
`-set_as_top`	在测试激励文件添加到的仿真文件集顶层设置生成的测试激励文件模块。
`-mode <arg>`	指定仿真模式。允许的值包括：behavioral（行为）、post-synthesis（综合后）或 post-implementation（实现后）。默认值为 behavioral。
`-type <arg>`	指定仿真类型。允许的值包括：functional（功能）或 timing（时序）（不适用于行为模式）。
`-force`	覆盖现有测试激励文件。
`-quiet`	以静默方式执行命令，不返回来自该命令的任何消息。此命令还会返回 TCL_OK，忽略执行期间遇到的所有错误。 **注释：**启动该命令时，会返回命令行上遇到的任何错误。仅捕获该命令内部发生的错误。
`-verbose`	暂时覆盖所有消息限制，并返回来自该命令的所有消息。 **注释：**可使用 `set_msg_config` 命令定义消息限制。

注释：

1. 由于按各不同标志所解释的方式来设置默认值，因此所有实参均为可选。

以下命令示例用于为 fifo 模块创建测试激励文件，并将其添加到 `sub_design_fifo` 仿真文件集中：

```
create_testbench -name fifo -add_to_simset sub_design_fifo
```

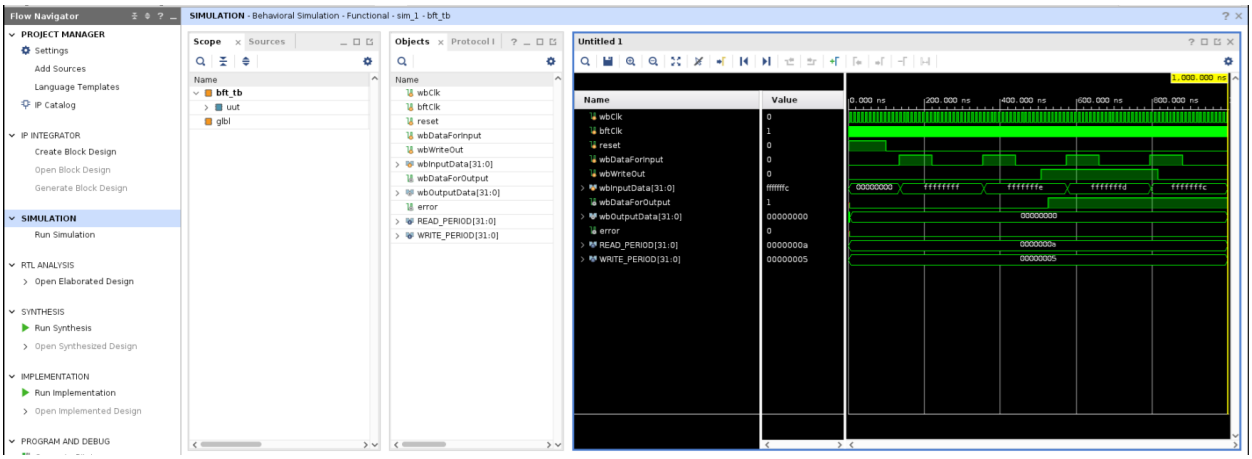
以下命令示例用于为类型为 buf 的模块的 `/top/DUT/fifo/buf_1` 实例生成 VCD 文件、记录此 VCD 文件内 2000 ns 内的波形活动、创建含名为 tb 的模块的测试激励文件、将此测试激励文件添加到 `test_buffer` 仿真文件集，并将 tb 设为此文件集中的顶层模块：

```
generate_vcd_ports {/top/DUT/fifo/buf_1}
run 2000ns
close_vcd -ports
create_testbench -name tb -add_to_simset test_buffer -set_as_top
```

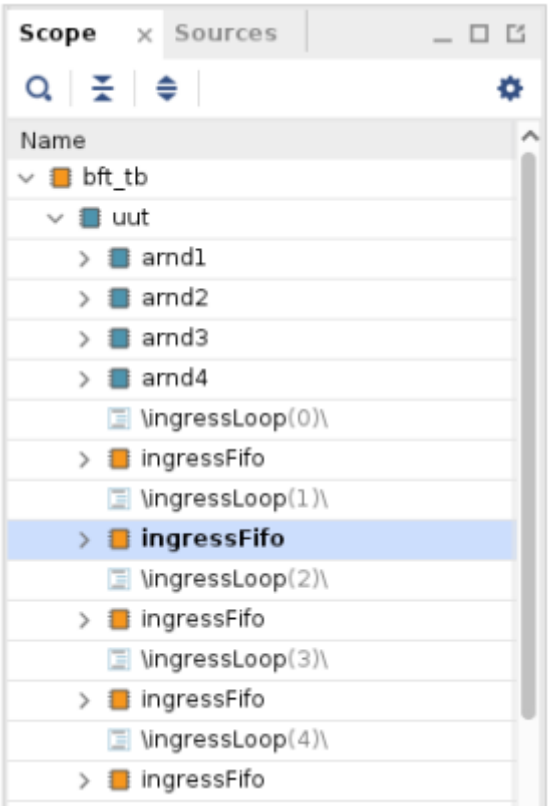
对设计示例使用测试激励文件自动生成

为便于演示，此处使用 Vivado IDE 随附的 BFT 设计示例。

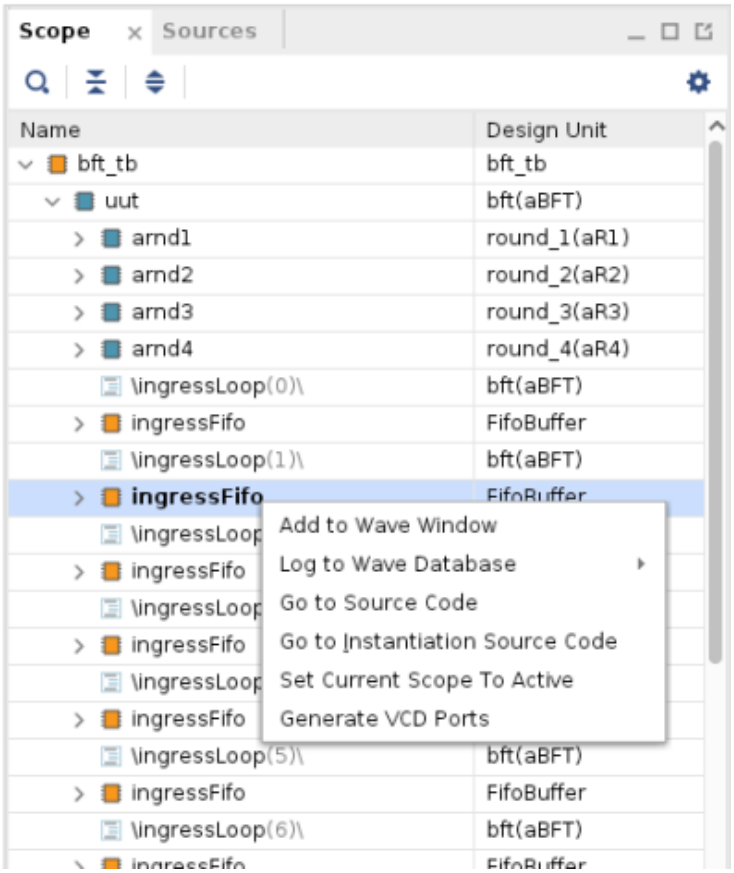
1. 在 Vivado IDE 中打开 BFT 设计示例。
2. 调用 `launch_simulation`，选择 Vivado 作为仿真器。这样您将看到如下波形所示端口。



3. 选择要生成测试激励文件的目标作用域，如下图所示：



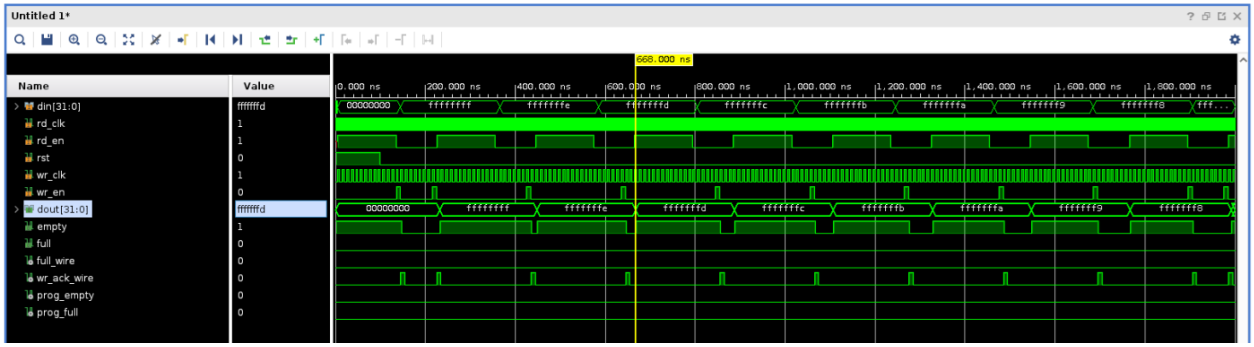
4. 右键单击选定的作用域，然后选择“Generate VCD Port”（生成 VCD 端口）。



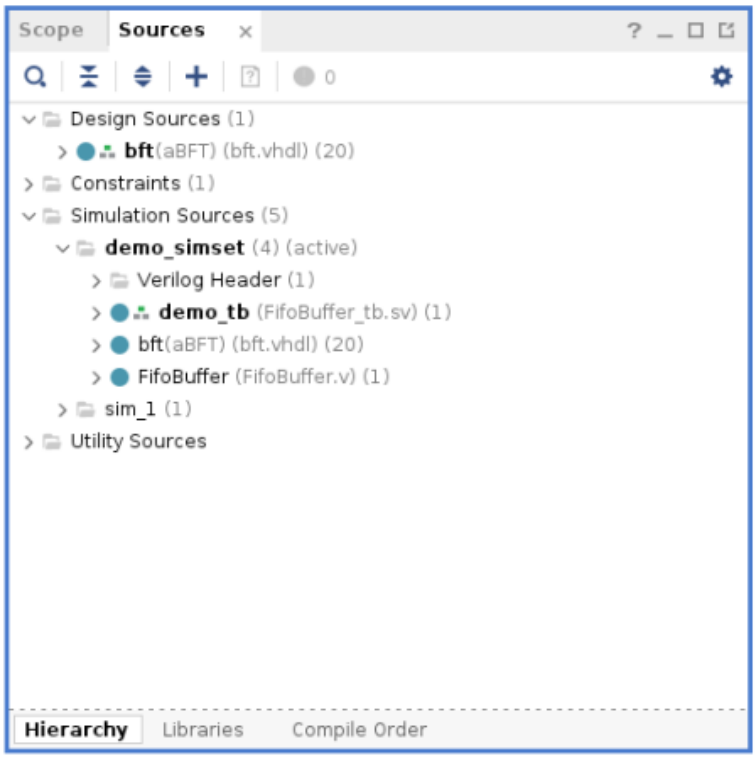
5. 删除波形上的所有现有信号，然后为选定的作用域选择“Add to Wave Window”（添加到波形窗口）。

   注释：步骤 5 用于演示所生成的测试激励文件正确驱动设计单元的过程。

6. 在 Tcl 控制台上使用 `restart`、`run 2000 ns` 和 `close_vcd -ports` 命令来转储信号活动。这样即可记录波形上时间从 0 到 2000 ns 的信号，如下图所示：



7. 在 Tcl 控制台上使用 `create_testbench -name demo_tb -add_to_simset demo_simset -set_as_top` 命令来生成测试激励文件。这样即可以模块名称 `demo_tb` 来创建测试激励文件，并以此测试激励文件作为顶层模块来创建 `demo_simset`。



8. 使用 `launch_simulation` 命令搭配新生成的测试激励文件来运行仿真。
9. 将此波形的输入/输出与您的原始设计的波形进行比较；输入/输出是相同的。

这就是您为子设计创建测试激励文件并搭配任意标准仿真器来独立使用生成的测试激励文件的方法。

附录 H

处理特殊情况

使用全局复位和三态

AMD 器件具有专用布线和电路，可连接到器件中的每个寄存器。

全局置位和复位信号线

配置期间，会将专用全局置位/复位 (GSR) 信号断言有效。完成器件配置时，此 GSR 信号会断言无效。所有触发器和锁存器都会接收到此复位，并根据寄存器的定义方式进行置位或复位。

虽然您可在配置后访问 GSR 信号线，但请避免使用 GSR 电路代替手动复位。这是因为，FPGA 器件为高扇出信号（例如，系统复位）提供高速主干布线。此主干布线比专用 GSR 电路更快，并且相较于负责传输 GSR 信号的专用全局布线更易于分析。

在综合后和实现后仿真中，前 100 ns 内 GSR 信号会自动断言有效，以对配置后发生的复位进行仿真。

在综合前功能仿真中可以选择提供 GSR 脉冲，但如果设计具有局部复位用于复位所有寄存器，那么此脉冲并非必需。

提示：创建测试激励文件时，请谨记，在综合后和实现后仿真中会自动发生 GSR 脉冲。这样即可在仿真的前 100 ns 内使所有寄存器保持处于复位状态。

注释：如果设计使用 ICAP 原语，届时 GSR 会持续 1.281 us。

全局三态信号线

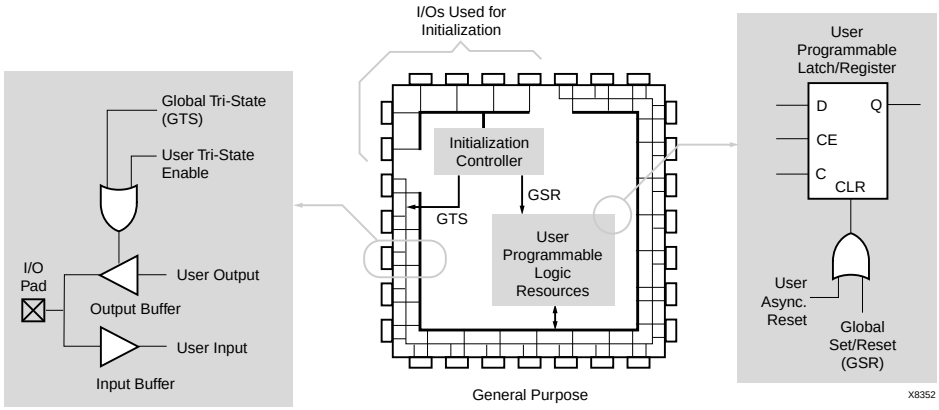
除了专用全局 GSR 外，输出缓冲器还会在含专用全局三态 (GTS) 信号线的配置模式期间设为高阻抗状态。所有通用输出在正常操作期间无论是常规输出、三态输出还是双向输出，都会受此影响。这样可以确保输出不会因为 FPGA 已配置而错误驱动其他器件。

在仿真中，通常不会驱动 GTS 信号。驱动 GTS 的电路在综合后和实现后仿真中可用，并且还可选择为综合前功能仿真添加该电路，但默认 GTS 脉冲宽度设为 0（零）。

使用全局三态及全局置位和复位信号

下图显示的是全局三态 (GTS) 及全局置位/复位 (GSR) 信号在 FPGA 中的使用方式。

图 54：**内置 FPGA 初始化电路图**



Verilog 中的全局置位和复位以及全局三态信号

GSR 和 GTS 信号是在以下模块中定义的：`<Vivado_Install_Dir>/data/verilog/src/glbl.v`。

大多数情况下，无需在测试激励文件中定义 GSR 和 GTS。

`glbl.v` 文件会声明全局 GSR 和 GTS 信号，并自动将 GSR 脉冲 100 ns。

VHDL 中的全局置位和复位以及全局三态信号

GSR 和 GTS 信号是在以下文件中定义的：`<Vivado_Install_Dir>/data/vhdl/src/unisims/primitive/GLBL_VHD.vhd`。

要使用 `GLBL_VHD` 组件，您必须将其例化到测试激励文件内。

`GLBL_VHD` 组件会声明全局 GSR 和 GTS 信号，并自动将 GSR 脉冲 100 ns。

以下代码片段所示示例中演示了如何在测试激励文件中例化 `GLBL_VHD` 组件并将“Reset on Configuration (ROC)”（配置时复位）的断言脉冲宽度更改为 90 ns：

```
GLBL_VHD inst:GLBL_VHD generic map (ROC_WIDTH => 90000);
```

增量周期和争用状况

本用户指南描述了基于事件的仿真器。基于事件的仿真器可以在给定仿真时间处理多起事件。在处理这些事件的同时，仿真器无法推进仿真时间。此事件处理时间通常被称为增量周期。在给定仿真时间步骤中可能存在多个增量周期。

仅当当前仿真时间没有任何其他传输事务需要处理时，仿真时间才会推进。因此，根据时间步骤内调度事件的时间，仿真器可能出现意外结果。以下 VHDL 编码示例显示了可能发生意外结果的方式。

含意外结果的 VHDL 编码示例

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;
clk_b_prcs : process (clk_b)
begin
  if (clk_b'event and clk_b='1') then
    result1 <= result;
  end if;
end process;
```

在此示例中，有两个同步进程：

- `clk_prcs`
- `clk_b_prcs`

仿真器先执行 `clk_b <= clk` 赋值，然后再推进仿真时间。因此，应在两个时钟沿发生的事件改为仅在一个时钟沿发生，导致出现争用状况。

对于此类情况，在仿真器中引入因果关系的建议方法包括：

- 不得同时更改时钟和数据。在每个输出插入延迟。
- 使用相同的时钟。
- 使用临时信号强制增量延迟，如下示例所示：

```
clk_b <= clk;
clk_prcs : process (clk)
begin
  if (clk'event and clk='1') then
    result <= data;
  end if;
end process;
result_temp <= result;
clk_b_prcs : process (clk_b)
begin
if (clk_b'event and clk_b='1') then
    result1 <= result_temp;
  end if;
end process;
```

大部分基于事件的仿真器都能显示增量周期。调试仿真问题时请务必对此加以充分利用。

使用 ASYNC_REG 约束

ASYNC_REG 约束：

- 识别设计中的异步寄存器
- 禁用这些寄存器的 X 传输

ASYNC_REG 约束可通过以下方式附加到前端设计中的寄存器：

- 在 HDL 代码中使用属性
- 在赛灵思设计约束 (XDC) 中使用约束

ASYNC_REG 附加到的寄存器会在时序仿真期间保留先前的值，并且不会向仿真输出 X。请谨慎使用；也可能在此过程中已通过时钟设置来输入新的值。

ASYNC_REG 约束仅适用于 CLB 以及输入输出块 (IOB) 寄存器和锁存器。欲知详情，请参阅《Vivado Design Suite 属性参考指南》(UG912) 中的 ASYNC_REG。

如果无法避免在异步数据中进行时钟设置，请仅对 IOB 或 CLB 寄存器执行时钟设置。在异步信号中对 RAM、移位寄存器 LUT (SRL) 或其他同步元素进行时钟设置无法得到确定性的结果；因此应避免使用。AMD 强烈建议您首先在寄存器、锁存器或 FIFO 中正确同步任何异步信号，然后再写入 RAM、移位寄存器 LUT (SRL) 或任何其他同步元素。如需了解更多信息，请参阅《Vivado Design Suite 用户指南：使用约束》(UG903)。

为同步元件禁用 X 传输

如果时序仿真期间发生时序违例，那么锁存器、寄存器、RAM 或其他同步元件的默认行为是向仿真器输出 X。发生此操作的原因是实际输出值未知。寄存器的输出可能：

- 保留其原始值
- 更新至新的值
- 进入亚稳态，在此情况下，直至同步元件完成时钟设置并经过一段时间之后，方能得到确定的值

由于无法确定该值，无法保证仿真结果准确，因此元件会输出 X 表示未知值。X 输出会保留至下一个时钟周期，在此时钟周期内如果未再次发生违例，下一个时钟设置的值就会更新此输出。

X 输出的存在会对仿真产生显著影响。例如，某个寄存器生成的 X 可能会传输至后续时钟周期的其他寄存器。这可能导致大部分受测设计变为未知。

要纠正生成的 X，请执行以下操作：

- 在同步路径上，分析路径并修复与此路径或其他路径关联的所有时序问题，以确保工作电路正常。
- 在异步路径上，如果无法以其他方式避免时序违例，请在时序违例期间使用 ASYNC_REG 属性禁用同步元件上的 X 传输。

禁用 X 传输后，在寄存器输出时会保留先前的值。在实际硅片中，寄存器可能已更改为“新的”值。禁用 X 传输产生的仿真结果可能与硅片行为不匹配。

注意！使用该选项时请谨慎实践。仅当您无法以任何其他方式避免时序违例时，才能使用该选项。

仿真配置接口

本节描述了以下配置接口的仿真：

- JTAG 仿真
- SelectMAP 仿真

JTAG 仿真

仿真支持 JTAG 端口与部分 JTAG 操作命令的交互。JTAG 接口（包括与扫描链对接的接口）不完全受支持。要对此接口进行仿真，请执行以下操作：

1. 例化 BSCANE2 组件，并将其连接到设计。
2. 将 JTAG_SIME2 组件例化到测试激励文件（而不是设计）中。

这样即可得到：

- 与外部 JTAG 信号（例如，TDI、TDO 和 TCK）对接的接口
- 到 BSCAN 组件的通信通道

组件之间的通信在 VPKG VHDL 包文件内部或者 glbl Verilog 全局模块内部发生。与此对应，特定 JTAG_SIME2 组件与设计或特定 BSCANE2 符号之间不需要隐式连接。

可从测试激励文件内的特定 JTAG_SIME2 组件驱动和查看激励，以了解 JTAG/BSCAN 函数的运算。在 AMD Vivado™ Design Suite 模板和特定器件库指南内都提供了这两个组件的例化模板。

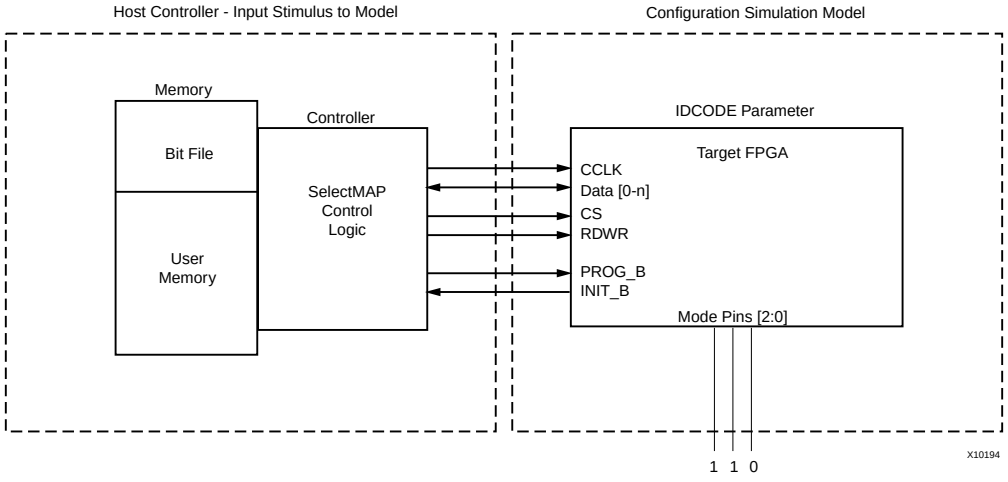
SelectMAP 仿真

含例化模板的配置仿真模型（SIM_CONFIGE2 和 SIM_CONFIGE3）允许对受支持的配置接口进行仿真，最终这样即可显示转至 HIGH（高电平）的 DONE 管脚。该模型演示了受支持的器件对受支持的配置接口上的激励进行响应的方式。

这些模型会处理控制信号活动以及比特文件下载。内部寄存器设置（例如，CRC 和 IDCODE）以及状态寄存器也包含在内。您可对进入器件的 Sync Word（同步字）进行监控，也可对同步字处理过程中的启动顺序进行监控。下图显示了系统从硬件映射到仿真环境的方式。

在配置用户指南中专为每个器件概括了配置进程。这些指南包含有关配置顺序以及配置接口的信息。

图 55：**模型交互的模块框图**



系统级别描述

配置模型允许在硬件可用之前对配置接口控制逻辑进行测试。它会仿真整个器件，在系统级别的适用范围包括：

- 适用于使用处理器来控制配置逻辑的应用，目的是确保连线、控制信号处理以及数据输入对齐都正确无误。
- 适用于使用 CS（SelectMAP 芯片选择）或 CLK 信号来控制数据加载进程的应用，目的是确保数据对齐正确无误。
- 适用于需要执行 SelectMAP `ABORT` 或 `Readback` 的系统。

`config_test_bench.zip` 文件具有测试激励文件样本，用于对运行 SelectMAP 逻辑的处理器进行仿真。这些测试激励文件具有控制逻辑用于对控制 SelectMAP 接口的处理器进行仿真，并包含诸如完整配置、ABORT 以及对 IDCODE 和状态寄存器执行 Readback（回读）等功能特性。

要获取与该模型关联的 ZIP 文件，请参阅 AMD 答复记录 53632。

所仿真的主机系统必须具有相应的方法用于文件交付和控制信号管理。这些控制系统应按器件配置用户指南中指定的方式来进行设置。

该配置模型还演示了在将 BIT 文件载入器件时，配置过程中器件内部所发生的状况。

BIT 文件下载期间，该模型会处理每条命令并更改寄存器设置以反映硬件更改。

您可在 CRC 寄存器主动累积 CRC 值的过程中，对其进行监控。该模型还演示了器件经历不同配置阶段的过程中，所设置的状态寄存器位。

模型调试

每个配置模型均提供了一个正确配置示例。如果您遇到器件烧录问题，可利用此示例来辅助完成调试过程。

您可使用 Vivado Device Programmer（Vivado 器件烧录器）工具通过 JTAG 来读取状态寄存器。该寄存器包含有关器件当前状态的信息，是非常实用的调试资源。如果您的开发板上遇到问题，那么首先要采用的调试步骤之一就是读取 Vivado Device Programmer 中的状态寄存器。

读取状态寄存器后，您可将其映射到仿真以确定器件的配置阶段。

例如，数据加载进程成功完成后，`GHIGH` 位会置于 HIGH（高位），如果该位并未置位，则表示数据加载操作并未完成。您还可以监控启动顺序中释放的 `GTS`、`GWE` 和 `DONE` 信号，这些信号均在 BitGen 中置位。

配置模型还允许错误注入。如果数据加载因出现任何问题而暂停并重新启动，那么处于活动状态的 CRC 逻辑会检测所有问题。它还会检测 BIT 文件中手动插入的比特翻转，并按照与器件相同的方式来处理该错误。

功能特性支持

每个器件专用的配置用户指南中均概述了针对每个配置接口所支持的交互方法。下表显示了配置用户指南中提到的各项受支持的功能特性。

`SIM_CONFIGE2 and SIM_CONFIGE3` 模型：

- 不支持回读配置数据。
- 不存储所提供的配置数据，但是会计算 CRC 值。
- 只能对特定寄存器执行回读，目的是为了确保向器件提供有效的命令序列和信号处理。
- 它并非旨在允许生成回读数据文件。

表 46：**模型支持的 Slave SelectMAP 功能特性**

Slave SelectMAP 功能特性	受支持
主动模式	不支持

表 46：**模型支持的 Slave SelectMAP 功能特性** (续)

Slave SelectMAP 功能特性	受支持
菊链 - 从动并行菊链	不支持
SelectMAP 数据加载	支持
SelectMAP 连续数据加载	支持
SelectMAP 非连续数据加载	支持
SelectMAP ABORT	支持
SelectMAP 重新配置	不支持
SelectMAP 数据排序	支持
重配置和多重启动	不支持
配置 CRC - 配置期间执行 CRC 检查	支持
配置 CRC - 配置后 CRC	不支持

为仿真禁用块 RAM 冲突检查

AMD 块 RAM 存储器是真正的双端口 RAM，其中两个端口均可随时访问任意存储器位置。请确保同时执行读取和写入时，不会对相同地址空间进行寻址。这会导致块 RAM 地址冲突。这些属于有效的冲突，因为从读取端口读取的数据是无效的。

在硬件中，读取的值可能是旧数据、新数据或者新旧数据组合。

在仿真中，这是通过输出 X 来完成建模的，因为读取的值未知。如需了解有关块 RAM 冲突的更多信息，请参阅器件的用户指南。

在某些应用中，无法避免此类冲突，也无法通过设计来绕过此状况。在这类情况下，块 RAM 可配置为不查找这些违例。这是通过块 RAM 原语中的泛型 (VHDL) 或参数 (Verilog) SIM_COLLISION_CHECK 字符串来控制的。

下表所示的字符串选项可供您搭配 SIM_COLLISION_CHECK 用于控制发生冲突情况下的仿真行为。

表 47：**SIM_COLLISION_CHECK 字符串**

字符串	写入冲突消息	在输出中写入 X
ALL	支持	支持
WARNING_ONLY	支持	否。仅在发生冲突时才适用。后续读取相同地址空间可能会在输出上生成 X。
GENERATE_X_ONLY	不支持	支持
无	不支持	否。仅在发生冲突时才适用。后续读取相同地址空间可能会在输出上生成 X。

在实例级别应用 SIM_COLLISION_CHECK，这样您即可更改每个块 RAM 实例的设置。

转储切换活动交换格式文件用于功耗分析

- Vivado 仿真器：使用 Vivado 仿真器执行功耗分析
- 第 3 章：使用第三方仿真器进行仿真 中的 转储 SAIF 用于功耗分析 和 在 VCS 中转储 SAIF

跳过编译或仿真

跳过编译

您可在现有快照上运行仿真，并跳过设计编译（或重新编译），方法是在仿真文件集上设置 SKIP_COMPILATION 属性：

```
set_property SKIP_COMPILATION 1 [get_filesets sim_1]
```

注释：设置该属性后，最后一次编译后对设计文件执行的任意更改都不会反映在仿真中。

跳过仿真

要通过细化和编译仿真快照而不运行仿真的方式来对设计 HDL 文件执行语义检查，可以在仿真文件集上设置 SKIP_SIMULATION 属性：

```
set_property SKIP_SIMULATION true [get_filesets sim_1]
```

重要提示！如果选择使用以上任一属性，请在仿真设置中禁用“Clean up simulation files”（清除仿真文件）复选框，或者如果您当前在批处理/Tcl 模式下运行，请调用 launch_simulation 并搭配 -noclean_dir。

AMD

附录 I

Vivado 仿真器 Tcl 命令中的值规则

本附录包含适用于 `add_force` Tcl 命令和 `set_value` Tcl 命令的值规则。

字符串值解读

值字符串的解读是根据 HDL 对象的声明类型和 `-radix` 命令行选项来确定的。`-radix` 始终覆盖 HDL 对象类型确定的默认基数。

- 对于类型为 `logic` 的 HDL 对象，值是 `logic` 类型的一维阵列，或者值是指定基数的数字字符串。
  - 如果字符串指定的位数少于该类型期望的位数，那么该字符串将采用隐式零位扩展（而非符号位扩展），以匹配该类型的长度。
  - 如果该字符串指定的位数多于该类型期望的位数，那么 MSB 侧的额外的位必须为零，否则，该命令会生成大小不匹配错误。

  例如，值 3F 指定 8 位（每个十六进制数字 4 位），含基数十六进制和 1 个 6 位 `logic` 阵列，等同于二进制 0011 1111。但由于 3 的上 2 个位为 0（零），该值可赋值给 HDL 对象。相较之下，值 7F 会生成错误，因为上 2 个位不为零。
  - 标量（非阵列或记录）`logic` HDL 对象的隐式长度为 1 位。
  - 对于声明为 `a [left:right]` (Verilog) 或 `a(left TO/DOWNTO right)` 的 `logic` 阵列，最左侧的值位（位于扩展/截断位之后）赋值给 `a[left]`，最右侧的值位赋值给 `a[right]`。

Vivado Design Suite 仿真逻辑

逻辑并非 HDL 中定义的概念，而是 AMD Vivado™ 仿真器引入的启发式概念。

- Verilog 对象如果属于隐式 Verilog bit 类型（包含 wire 和 reg 对象以及整数和时间），则会被视为 `logic` 类型。
- VHDL 对象如果符合下列条件，则会被视为 `logic` 类型：对象类型为 bit、`std_logic` 或所含枚举器是 `std_logic` 的子集且至少包含 0 和 1 的任意枚举类型，或者对象类型是前述任意类型的一维阵列。
- 对于属于 VHDL 枚举类型的 HDL 对象，值可设为任一枚举器字面值，如果枚举器是字符字面值，则不含单引号。忽略基数。
- 对于整型类型的 VHDL 对象，值可设为该类型范围内的有符号十进制整数。忽略基数。
- 对于 VHDL 和 Verilog 浮点类型，值可设为浮点值。忽略基数。
- 对于所有其他类型的 HDL 对象，Tcl 命令集不支持设置值。

附录 J

Vivado 仿真器混合语言支持和语言例外

Vivado 集成设计环境 (IDE) 支持下列语言：

- VHDL，请参阅《IEEE 标准 VHDL 语言参考手册》(IEEE-STD-1076-1993)
- Verilog，请参阅《IEEE 标准 Verilog 硬件描述语言》(IEEE-STD-1364-2001)
- SystemVerilog 可综合子集。请参阅《适用于 SystemVerilog 的 IEEE 标准 - 统一硬件设计、规范和验证语言》(IEEE-STD-1800-2009)
- IEEE P1735 加密，请参阅《推荐的电子产品设计 IP 加密与管理实践》(IEEE-STD-P1735)

本附录列出了 Vivado 仿真器中应用的混合语言以及针对 Verilog、SystemVerilog 和 VHDL 支持的例外。

使用混合语言仿真

Vivado 仿真器支持混合语言工程文件及混合语言仿真。这样您即可在 VHDL 设计中包含 Verilog/SystemVerilog (SV) 模块，反之亦然。

仿真中混用语言的限制

- VHDL 设计可以例化 Verilog/SystemVerilog (SV) 模块，Verilog/SV 设计可以例化 VHDL 组件。基于组件例化的默认绑定用于将 Verilog/SV 模块绑定到 VHDL 组件。任何其他种类的 VHDL 和 Verilog 混用（例如，VHDL 进程调用 Verilog 函数）均不予支持。
- 在 Verilog/SV 模块边界上允许少量 VHDL 类型、泛型和端口。同样，在 VHDL 组件边界上允许少量 Verilog/SV 类型、参数和端口。请参阅 表 49：受支持的 VHDL 和 Verilog 数据类型。

  重要提示！不支持将整个 VHDL 记录对象连接到单个 Verilog 对象；但可将受支持类型的 VHDL 记录元素连接到兼容的 Verilog 端口。

- Verilog/SV 分层引用无法引用 VHDL 单元，VHDL 扩展名称或选定名称也无法引用 Verilog/SV 单元。

混合语言仿真中的关键步骤

1. （可选）在混合语言工程的设计库中为 VHDL 组件或 Verilog/SV 模块指定搜索顺序。
2. `xelab -L` 可用于指定混合语言工程的设计库中的 VHDL 组件或 Verilog/SV 模块的绑定顺序。

   注释：`-L` 指定的库搜索顺序也可用于将 Verilog 模块绑定到其他 Verilog 模块。

混合语言绑定与搜索

例化 Verilog/SV 模块中的 VHDL 组件或者 VHDL 架构中的 Verilog/SV 模块时，`xelab` 命令会：

- 首先搜索与例化的设计单元相同的语言单位。
- 如果未找到相同语言单位，`xelab` 就会在 `-L` 选项指定的库中搜索跨语言设计。

搜索顺序与 `xelab` 命令行上显示的库的顺序相同。

注释：使用 Vivado IDE 时，会自动指定库搜索顺序。无需也无法进行用户干预。

相关信息

Verilog 搜索顺序

例化混合语言组件

在混合语言设计中，您可以在 VHDL 架构内例化 Verilog/SV 模块，或者也可以在 Verilog/SV 模块内例化 VHDL 组件，如后续各小节中所述。

为确保您正确匹配端口类型，请复查 端口映射和受支持的端口类型。

在 VHDL 设计单元内例化 Verilog 模块

1. 声明同名的 VHDL 组件，在某些情况下，声明与您要例化的 Verilog 模块相同名称的 VHDL 组件。例如：

```
COMPONENT MY_VHDL_UNIT PORT (
    Q : out STD_ULOGIC;
    D : in  STD_ULOGIC;
    C : in  STD_ULOGIC );
END COMPONENT;
```

2. 使用命名关联或位置关联来例化 Verilog 模块。例如：

```
UUT : MY_VHDL_UNIT PORT MAP(
    Q => O,
    D => I,
    C => CLK);
```

在 Verilog/SV 设计单元内例化 VHDL 组件

要在 Verilog/SV 设计单元内例化 VHDL 组件，请按在 Verilog/SV 模块中例化此 VHDL 组件的相同方式对其进行例化。

例如：

```
module testbench ;
wire in, clk;
wire out;
FD FD1(
  .Q(Q_OUT),
  .C(CLK);
  .D(A);
);
```

端口映射和受支持的端口类型

下表列出了受支持的端口类型。

表 48：**受支持的端口类型**

VHDL[1]	Verilog/SV[2]
IN	INPUT
OUT	OUTPUT
INOUT	INOUT

注释：

1. 不支持 VHDL 的缓冲器和链接端口。
2. 在 Verilog 中不支持连接到双通开关。在混用的设计边界上不允许存在未命名的 Verilog 端口。

下表显示了针对混用语言的设计边界上的端口受支持的 VHDL 和 Verilog 数据类型。

表 49：**受支持的 VHDL 和 Verilog 数据类型**

VHDL 端口	Verilog 端口
`bit`	信号线
`std_logic`	信号线
`bit_vector`	矢量信号线
`signed`	矢量信号线
`unsigned`	矢量信号线
`std_ulogic_vector`	矢量信号线
`std_logic_vector`	矢量信号线

注释：在混用语言的边界上，支持类型为 `reg` 的 Verilog 输出端口。在边界上，`reg` 输出端口作为输出信号线（连线）端口来处理。混用语言边界上存在的任意其他类型的端口均视为错误。

注释：Vivado 仿真器支持将记录元素用作为混合域中例化的 Verilog 模块的端口映射中的实际元素。表 49：受支持的 VHDL 和 Verilog 数据类型 中列明支持作为 VHDL 端口的数据类型也都同样支持用作为记录元素。

表 50：**受支持的 SV 和 VHDL 数据类型**

SV 数据类型	VHDL 数据类型
Int	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
byte	
	`bit_vector`
	`std_logic_Vector`
	`std_ulogic_vector`
	`signed`
	`unsigned`
shortint	

表 50：**受支持的 SV 和 VHDL 数据类型** (续)

SV 数据类型	VHDL 数据类型
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
longint	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
integer	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
bit(1D) 的矢量	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
logic(1D) 的矢量	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
reg(1D) 的矢量	
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned
logic/bit	
	bit
	std_logic
	std_ulogic

表 50：**受支持的 SV 和 VHDL 数据类型** (续)

SV 数据类型	VHDL 数据类型
	bit_vector
	std_logic_Vector
	std_ulogic_vector
	signed
	unsigned

注释：支持用于例化具有真实端口的 Verilog 模块的 VHDL 实体。

泛型（参数）映射

Vivado 仿真器支持以下 VHDL 泛型类型（及其 Verilog/SV 等效类型）：

- integer
- 实数
- 字符串
- 布尔值

  注释：混用语言边界上存在的任意其他泛型类型均视为错误。

VHDL 值与 Verilog 值之间的映射

下表列出了 Verilog 状态到 `std_logic` 与 `bit` 之间的映射。

表 51：**映射到 std_logic 和 bit 的 Verilog 状态**

Verilog	std_logic	bit
Z	Z	0
0	0	0
1	1	1
X	X	0

注释：忽略 Verilog 强度。不存在与 VHDL 中的强度的对应映射。

下表列出了 VHDL 类型 `bit` 到 Verilog 状态的映射。

表 52：**VHDL bit 到 Verilog 状态的映射**

bit	Verilog
0	0
1	1

下表列出了 VHDL 类型 `std_logic` 到 Verilog 状态的映射。

表 53：**VHDL std_logic 到 Verilog 状态的映射**

std_logic	Verilog
U	X
X	X
0	0
1	1
Z	Z
W	X
L	0
H	1
-	X

由于 Verilog 区分大小写，因此组件声明中使用的命名关联和本地端口名都必须与对应 Verilog 端口名大小写相匹配。

VHDL 语言支持例外

某些语言构造不受 Vivado 仿真器支持。下表列出了 VHDL 语言支持例外。

表 54：**VHDL 语言支持例外**

受支持的 VHDL 构造	例外
abstract_literal	不支持将浮点值表达为基数字面值 (based literal)。
alias_declaration	总体上不支持非对象别名；以下情况下尤其如此： · 别名的别名 · 不含 subtype_indication 的别名声明 · 别名声明的签名 · 运算符符号表达为 alias_designator · 运算符符号的别名 · 字符字面值表达为别名指示符
alias_designator	Operator_symbol 表达为 alias_designator Character_literal 表达为 alias_designator
association_element	在关联元素内对实际值进行分片时可接受全局和局部静态范围。
attribute_name	不支持前缀后接符号。
binding_indication	不支持不使用 entity_aspect 的 Binding_indication。
bit_string_literal	不支持空的 bit_string_literal (" ")。
block_statement	不支持 Guard_expression。例如，保护块、保护信号、保护目标和保护赋值均不受支持。
choice	不支持在 case 语句中使用聚合 (aggregate) 作为选项。
concurrent_assertion_statement	不支持推迟 (postponed)。
concurrent_signal_assignment_statement	不支持推迟 (postponed)。

表 54：**VHDL 语言支持例外** (续)

受支持的 VHDL 构造	例外
concurrent_statement	不支持包含 wait 语句的并发过程调用。
conditional_signal_assignment	不支持选项中受保护的关键字，因为不支持受保护的信号赋值。
configuration_declaration	不支持在配置中针对生成索引使用非局部静态。
entity_class	不支持使用字面值、单位、文件和分组作为实体类。
entity_class_entry	不支持用于分组模板的可选 <>。
file_logical_name	虽然允许 file_logical_name 采用求值结果为字符串值的任意非规范表达式，但仅接受字符串字面值和标识符作为文件名。
function_call	在 function_call 中的指定参数关联内不支持不同形式的分片、索引和选择。
instantiated_unit	不支持直接配置例化。
mode	不完全支持链接和缓冲器端口。
options	不支持保护对象。
primary	如果使用 primary，则在其中扩展分配器。
procedure_call	在 procedure_call 中的指定参数关联内不支持不同形式的分片、索引和选择。
process_statement	不支持推迟进程。
selected_signal_assignment	不支持选项中包含 guarded 关键字，因为不支持受保护的信号赋值。
signal_declaration	不支持 signal_kind。signal_kind 用于声明保护信号，此类信号不受支持。
subtype_indication	不支持复合体（阵列和记录）的已解析子类型。
waveform	不支持不受影响的对象。
waveform_element	不支持空波形元素，因为仅在保护信号范围内，此类元素才有关联。

Verilog 语言支持例外

下表列出了受支持的 Verilog 语言支持的例外情况。

表 55：**Verilog 语言支持例外**

Verilog 构造	例外
编译器指令构造	
`unconnected_drive	不支持
`nounconnected_drive	不支持
属性	
attribute_instance	不支持
attr_spec	不支持
attr_name	不支持
原语门电路和开关类型	

表 55：**Verilog 语言支持例外** (续)

Verilog 构造	例外
cmos_switchtype	不支持
mos_switchtype	不支持
pass_en_switchtype	不支持
生成的例化	

generated_instantiation	不支持 module_or_generate_item 替代项。
	源自 IEEE 标准（请参阅《IEEE 标准 Verilog 硬件描述语言 (IEEE-STD-1364-2001)》第 13.2 款）的生成结果：
	`generate_item_or_null ::=`
	`generate_conditonal_statement |`
	`generate_case_statement |`
	`generate_loop_statement |`
	`generate_block |`
	`module_or_generate_item`
	仿真器支持的生成结果：
	`generate_item_or_null ::=`
	`generate_conditional_statement|`
	`generate_case_statement |`
	`generate_loop_statement |`
	`generate_blockgenerate_condition`

genvar_assignment	部分支持。
	所有生成块都必须命名。
	源自标准（请参阅《IEEE 标准 Verilog 硬件描述语言 (IEEE-STD-1364-2001)》第 13.2 款）的生成结果：
	`generate_block ::=`
	`begin`
	`[ : generate_block_identifier ]`
	`{ generate_item }`
	`end`
	仿真器支持的生成结果：
	`generate_block ::=`
	`begin:`
	`generate_block_identifier {`
	`generate_item }`
	`end`

源文本构造	
库源文本	
library_text	不支持
library_descriptions	不支持
library_declaration	不支持
include_statement	这表示引用库映射文件内的 include 语句（请参阅《IEEE 标准 Verilog 硬件描述语言 (IEEE-STD-1364-2001)》第 13.2 款）。它并不引用 `include 编译器指令。
系统时序检查命令	
$skew_timing_check	不支持
$timeskew_timing_check	不支持

表 55：**Verilog 语言支持例外** (续)

Verilog 构造	例外
`$fullskew_timing_check`	不支持
`$nochange_timing_check`	不支持
系统时序检查命令实参	
`checktime_condition`	不支持
PLA 建模任务	
`$async$nand$array`	不支持
`$async$nor$array`	不支持
`$async$or$array`	不支持
`$sync$and$array`	不支持
`$sync$nand$array`	不支持
`$sync$nor$array`	不支持
`$sync$or$array`	不支持
`$async$and$plane`	不支持
`$async$nand$plane`	不支持
`$async$nor$plane`	不支持
`$async$or$plane`	不支持
`$sync$and$plane`	不支持
`$sync$nand$plane`	不支持
`$sync$nor$plane`	不支持
`$sync$or$plane`	不支持
值更改转储 (VCD) 文件	

`$dumpportson`
`$dumpports`
`$dumpportsoff`
`$dumpportsflush`
`$dumpportslimit`
`$vcdplus`
	不支持

Vivado 仿真器快捷参考指南

下表提供了常用 AMD Vivado™ 仿真器命令的快捷参考和示例。

<table>
<tr><th colspan="3">解析 HDL 文件</th></tr>
<tr><td colspan="3">Vivado 仿真器支持三种 HDL 文件类型：Verilog、SystemVerilog 和 VHDL。您可使用 XVHDL 和 XVLOG 命令解析受支持的文件。</td></tr>
<tr><td>解析 VHDL 文件</td><td colspan="2">xvhdl file1.vhd file2.vhd
xvhdl -work worklib file1.vhd file2.vhd
xvhdl -prj files.prj</td></tr>
<tr><td>解析 Verilog 文件</td><td colspan="2">xvlog file1.v file2.v
xvlog -work worklib file1.v file2.v
xvlog -prj files.prj</td></tr>
<tr><td>解析 SystemVerilog 文件</td><td colspan="2">xvlog -sv file1.v file2.v
xvlog -work worklib -sv file1.v file2.v
xvlog -prj files.prj
如需了解有关 PRJ 文件格式的信息，请参阅 工程文件 (.prj) 语法。</td></tr>
<tr><th colspan="3">其他 xvlog 和 xvhdl 选项</th></tr>
<tr><td>xvlog 和 xvhdl 主要选项</td><td colspan="2">如需获取命令选项的完整列表，请参阅 表 15：xelab、xvhd 和 xvlog 命令选项。
xvlog 和 xvhdl 的主要选项如下所示：</td></tr>
<tr><td></td><td>主要选项</td><td>适用于：</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog 和 xvhdl</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog 和 xvhdl</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog 和 xvhdl</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog 和 xvhdl</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog, xvhdl</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvhdl, vlog</td></tr>
<tr><td></td><td>xelab、xvhdl 和 xvlog xsim 命令选项</td><td>xvlog 和 xvhdl</td></tr>
<tr><th colspan="3">细化和生成可执行快照</th></tr>
<tr><td colspan="3">解析后，您可在 Vivado 仿真器中使用 XELAB 命令来细化设计。XELAB 用于生成可执行快照。
您可跳过解析器阶段，直接调用 XELAB 命令并传递 PRJ 文件。XELAB 用于调用 XVLOG 和 XVHDL 来解析文件。</td></tr>
</table>

用法	`xelab top1 top2`	细化具有如下 2 个顶层设计单元的设计：`top1` 和 `top2`。在此示例中，设计单元是在 `work` 库中编译的。
	`xelab lib1.top1 lib2.top2`	细化具有如下 2 个顶层设计单元的设计：`top1` 和 `top2`。在此示例中，设计单元是分别在 `lib1` 和 `lib2` 中编译的
	`xelab top1 top2 -prj files.prj`	细化具有如下 2 个顶层设计单元的设计：`top1` 和 `top2`。在此示例中，设计单元是在 `work` 库中编译的。`files.prj` 文件包含如下条目： `verilog <libraryName> <VerilogDesignFileName> vhdl <libraryName> <VHDLDesignFileName> sv <libraryName> <SystemVerilogDesignFileName>`
	`xelab top1 top2 -s top`	细化具有如下 2 个顶层设计单元的设计：`top1` 和 `top2`。在此示例中，设计单元是在 `work` 库中编译的。编译后，`xelab` 会生成名为 `top` 的可执行快照。如无 `-s top` 开关，`xelab` 会通过串联单元名称来创建快照。
命令行帮助和 xelab 选项	`xelab -help` 表 15：xelab、xvhd 和 xvlog 命令选项	
运行仿真		
解析后，细化和编译阶段即告成功；xsim 会加载可执行快照以运行仿真。		
用法	`xsim top -R`	对设计进行仿真直至完成。
	`xsim top -gui`	打开 Vivado 仿真器工作空间 (GUI)。
	`xsim top`	在 Tcl 模式下打开 Vivado Design Suite 命令提示符。您可从中调用如下选项： `run -all run 100 ns`
重要快捷方式		
您可通过单、双和三阶段方式调用解析、细化和可执行生成与仿真。		

	三阶段	`xvlog bot.v`
		`xvhdl top.vhd`
		`xelab work.top -s top`
		`xsim top -R`

	双阶段	`xelab -prj my_prj.prj work.top -s top`
		`xsim top -R`
		其中 `my_prj.prj` 文件包含：
		`verilog work bot.v`
		`vhdl work top.vhd`

	单阶段	`xelab -prj my_prj.prj work.top -s top -R`
		其中 `my_prj.prj` 文件包含：
		`verilog work bot.v vhdl work top.vhd`

注释：如果设计包含 UVM 构造，则需向 xvlog 和 xelab 命令传递 `-L uvm`		
Vivado 仿真 Tcl 命令		

常用 Tcl 命令如下所示。要获取完整列表，请在 Tcl 控制台中调用下列命令：
`load_features simulator`
`help -category simulation`
如需了解有关任意 Tcl 命令的信息，请输入：`-help <Tcl_command>`		

Vivado 仿真器常用 Tcl 命令：	`add_bp`	在某一行 HDL 源代码中添加断点。
	`add_force`	将某个信号、连线或寄存器的值强制设为指定值。如需获取 Tcl 命令示例，请参阅 使用 Force 命令。
	`current_time` `now`	报告当前仿真时间。如需获取 Tcl 脚本内此命令的示例，请参阅 使用 -tclbatch 文件。
	`current_scope`	报告或设置当前工作中 HDL 的作用域。如需了解更多信息，请参阅 “Scope” 窗口。
	`get_objects`	根据指定模式，获取一个或多个 HDL 作用域内的 HDL 对象列表。如需获取命令使用示例，请参阅：SAIF Tcl 命令示例。
	`get_scopes`	获取 HDL 子作用域列表。如需了解更多信息，请参阅 “Scope” 窗口。
	`get_value`	获取选定的 HDL 对象（变量、信号、连线或寄存器）的当前值。如需了解更多信息，请在 Tcl 控制台中输入 `get_value -help`。
	`launch_simulation`	使用 Vivado 仿真器启动仿真。
	`remove_bps`	从仿真中移除断点。
	`report_drivers`	为 HDL 连线或信号对象打印驱动程序和当前驱动值。如需了解更多信息，请参阅：使用 report_drivers Tcl 命令。
	`report_values`	打印给定 HDL 对象（变量、信号、连线或寄存器）的当前仿真值。
	`restart`	将仿真回绕至加载后状态（就像重新加载设计一样）；时间设为 0。
	`set_value`	将 HDL 对象（变量、信号、连线或寄存器）设为指定值。如需了解更多信息，请参阅：附录 I: Vivado 仿真器 Tcl 命令中的值规则。
	`step`	单步执行仿真直至下一个语句。请参阅 单步执行仿真。

AMD

附录 L

使用赛灵思仿真器接口

赛灵思仿真器接口 (XSI) 是一个 C/C++ 应用编程接口 (API)，用于对接 AMD 的 Vivado 仿真器 (xsim)，该仿真器支持 C/C++ 程序充当 HDL 设计的测试激励文件。通过使用 XSI，C/C++ 程序即可控制用于托管 HDL 设计的 Vivado 仿真器的活动。

C/C++ 程序通过下列方法对仿真进行控制：

- 设置 HDL 设计的顶层输入端口的值
- 指令 Vivado 仿真器按特定仿真时间量来运行仿真

此外，C/C++ 程序可以读取 HDL 设计的顶层输出端口的值。

执行以下步骤以在 C/C++ 程序中使用 XSI：

1. 准备 XSI API 函数，以供通过动态链接来调用
2. 使用 API 函数编写 C/C++ 测试激励文件代码
3. 编译并链接 C/C++ 程序
4. 将 Vivado 仿真器与 HDL 设计一起封装到共享库中

准备 XSI 函数用于动态链接

AMD 建议使用动态链接来间接调用 XSI 函数。虽然此方法所涉及的步骤多于直接调用 XSI 函数，但动态链接允许您的 HDL 设计与 C/C++ 程序的编译保持彼此独立。您可以随时编译和加载自己的 HDL 设计，即使在 C/C++ 程序持续运行时也是如此。

要通过动态链接来调用函数，您的程序需要执行以下步骤：

1. 打开包含该函数的共享库。
2. 按名称查找函数，获取指向该函数的指针。
3. 使用函数指针调用该函数。
4. 关闭共享库（可选）。

步骤 1、2 和 4 需要使用操作系统专用的库调用，如下表所示。请参阅您的操作系统文档以获取有关这些函数的详细信息。

表 57：**操作系统专用的库调用**

功能	Linux	Windows
打开共享库	`void *dlopen(const char *filename, int flag);`	`HMODULE WINAPI LoadLibrary(_In_ LPCTSTR lpFileName );`
按名称查找函数	`void *dlsym(void *handle, const char *symbol);`	`FARPROC WINAPI GetProcAddress(_In_ HMODULE hModule,_In_ LPCSTR lpProcName );`
关闭共享库	`int dlclose(void *handle);`	`BOOL WINAPI FreeLibrary(_In_ HMODULE hModule );`

XSI 要求您从两个共享库调用函数：内核共享库和您的设计共享库。内核共享库随附于 Vivado 仿真器，称为 `librdi_simulator_kernel.so` (Linux) 或 `librdi_simulator_kernel.dll` (Windows)。它驻留在以下目录中：

```
<Vivado Installation Root>/lib/<platform>
```

其中，`<platform>` 是 `lnx64.o` 或 `win64.o`。运行程序时，请确保将该目录包含在库路径中。在 Linux 上，将该目录包含在环境变量 `LD_LIBRARY_PATH` 中，在 Windows 上，将其包含在环境变量 `PATH` 中。

Vivado 仿真器会在编译 HDL 设计的过程中创建您的设计共享库（如 准备设计共享库 中所述），此共享库称为 `xsimk.so` (Linux) 或 `xsimk.dll` (Windows)，通常驻留在以下位置：

```
<HDL design directory>/xsim.dir/<snapshot name>
```

其中，`<HDL design directory>` 是在其中创建您的设计共享库的目录，`<snapshot name>` 是您在创建此库期间指定的快照名称。

您的 C/C++ 程序会调用驻留在您的设计共享库中的 XSI 函数 `xsi_open()`，并从内核共享库调用所有其他 XSI 函数。

Vivado 仿真器随附的 XSI 代码示例会将 XSI 函数整合到名为 `Xsi::Loader` 的 C++ 类中。该类会接受两个共享库的名称、在内部执行必要的动态链接步骤，并将所有 XSI 函数公开作为该类的成员函数。以此方式封装 XSI 函数即可消除直接调用动态链接操作系统函数的需求。您可在 Vivado 安装的以下位置中找到该类的源代码，并可将其复制到您自己的程序中：

```
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.h
<Vivado Installation Root>/examples/xsim/verilog/xsi/counter/xsi_loader.cpp
```

要使用 `Xsi::Loader`，只需按如下示例所示传递两个共享库的名称来对其进行例化即可：

```
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/xsimk.so",
"librdi_simulator_kernel.so");
```

编写测试激励文件代码

使用 XSI 的 C/C++ 测试激励文件通常使用以下步骤：

1. 打开设计。
2. 提取每个顶层端口的 ID。
3. 重复以下步骤直至完成仿真：
    a. 在顶层输入端口上设置值。
    b. 按特定时间量运行仿真。
    c. 提取顶层输出端口的值。
4. 关闭设计。

下表列出了用于每个步骤的 XSI 函数及其等效的 `Xsi::Loader` 成员函数。如需获取每个 XSI 函数的用法详细信息，请参阅 XSI 函数参考。

表 58：**Xsi: :Loader 成员函数**

活动	XSI 函数	Xsi::Loader 成员函数
打开设计	xsi_open	open
提取端口 ID	xsi_get_port_number	get_port_number
设置输入端口值	xsi_put_value	put_value
运行仿真	xsi_run	run
提取输出端口值	xsi_get_value	get_value
关闭设计	xsi_close	close

您可在以下 Vivado 安装位置找到使用 XSI 的 C++ 程序示例：

```
<Vivado Installation Root>/examples/xsim/<HDL language>/xsi
```

编译 C/C++ 程序

您可使用 XSI 示例程序作为指南。每个示例均可提供一个或两个脚本用于编译和运行示例。请参阅您的编译器文档以获取有关编译程序的详细信息。在 Linux 上，编译和运行是双步进程。

1. 在 C 语言的 shell 中，运行 source set_env.csh
2. 调用 run.csh

在 Windows 上，只需运行批处理文件 `run.bat` 即可。

请注意脚本中的以下内容：

1. 编译行通过 `-I` 指定包含含有 `xsi.h` include 文件的目录。
2. 编译 C++ 程序期间并未提及设计共享库或内核共享库。

XSI include 文件驻留在以下位置：

```
<Vivado Installation Root>/data/xsim/include/xsi.h
```

准备设计共享库

生成基于 XSI 的有效 C/C++ 程序的最后一步需编译 HDL 设计并将其与 Vivado 仿真器封装在一起以成为您的设计共享库。只要 HDL 设计源代码存在更改，即可重复此步骤。

注意！如果您要在程序持续运行的同时为自己的 C/C++ 程序重新构建设计共享库，请务必先在程序中关闭此设计，然后再执行此步骤。

创建设计共享库，具体方法是在 HDL 设计上调用 xelab，并包含 -dll 开关以指令 xelab 生成共享库代替常用快照，搭配 Vivado 仿真器的用户界面一起使用。

例如：

在 Linux 命令行中输入以下命令以创建位于 `./xsim.dir/design/xsimk.so` 的设计共享库：

```
xelab work.top1 work.top2 -dll -s design
```

其中，`work.top1` 和 `work.top2` 均为顶层模块名称，`design` 则是快照名称。

如需了解有关编译 HDL 设计的更多详细信息，请参阅 xelab、xvhdl 和 xvlog xsim 命令选项。

XSI 函数参考

本节以明文（直接 C 语言调用）形式和 `Xsi::Loader` 成员函数形式展示了每个 XSI API 函数。明文形式的函数会取用 `xsiHandle` 实参，而成员函数则不会取用该实参。`xsiHandle` 包含有关已打开的 HDL 设计的状态信息。明文形式的 `xsi_open` 会生成 `xsiHandle`。`Xsi::Loader` 内部包含 `xsiHandle`。

xsi_close

```
void xsi_close(xsiHandle design_handle);
void Xsi::Loader::close();
```

此函数用于关闭 HDL 设计，清空与设计关联的存储器。调用该函数即可结束仿真。

xsi_get_error_info

```
const char* xsi_get_error_info(xsiHandle design_handle);
const char* Xsi::Loader::get_error_info();
```

此函数用于返回遇到的最后一个错误的字符串描述。

xsi_get_port_number

```
XSI_INT32 xsi_get_port_number(xsiHandle design_handle, const char*
port_name);
int Xsi::Loader::get_port_number(const char* port_name);
```

此函数会为请求的 HDL 设计顶层端口返回整数 ID。随后，您可使用此 ID 在 `xsi_get_value` 调用和 `xsi_put_value` 调用中指定端口。`port_name` 是端口名称，对于 Verilog，此名称区分大小写，对于 VHDL 则不区分大小写。如果不存在含指定名称的端口，则该函数会返回 `-1`。

代码示例：

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
```

xsi_get_status

```
XSI_INT32 xsi_get_status(xsiHandle design_handle);
int Xsi::Loader::get_status();
```

该函数用于返回仿真状态。状态可能为以下标识符之一：

表 59：**Xsi 仿真状态标识符**

状态代码标识符	描述
`xsiNormal`	无错误。
`xsiError`	仿真遇到 HDL 运行时错误。
`xsiFatalError`	仿真遇到错误状况，导致 Vivado 仿真器无法继续。

代码示例：

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
...
if (loader.get_status() == xsiError)
    printf("HDL run-time error encountered.\n");
```

xsi_get_value

```
void xsi_get_value(xsiHandle design_handle, XSI_INT32 port_number, void*
value);
int Xsi::Loader::get_value(int port_number, void* value);
```

该函数用于提取端口 ID `port_number` 所表示的端口的值。该值置于该值指向的存储缓冲器中。如需了解有关获取端口 ID 的信息，请参阅 xsi_get_port_number。

重要提示！您的程序必须为缓冲器分配足够的存储器才能调用该函数。要确定缓冲器的必要大小，请参阅 Vivado 仿真器 VHDL 数据格式 和 Vivado 仿真器 Verilog 数据格式。

代码示例：

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Buffer for value of port "count"
s_xsi_vlog_logicval count_val = {0X00000000, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
...
int count = loader.get_port_number("count");
loader.get_value(count, &count_val);
```

xsi_open

```
typedef struct t_xsi_setup_info {
    char* logFileName;
    char* wdbFileName;
} s_xsi_setup_info, *p_xsi_setup_info;
xsiHandle xsi_open(p_xsi_setup_info setup_info);
void Xsi::Loader::open(p_xsi_setup_info setup_info);
bool Xsi::Loader::isopen() const;
```

此函数用于打开 HDL 设计以便进行仿真。要使用此函数，必须首先例化 `s_xsi_setup_info` 结构体以传递给该函数。将 `logFileName` 用作为仿真 log 日志文件的名称，或者使用 `NULL` 禁用日志记录。如果波形追踪已开启（请参阅 xsi_trace_all），那么输出波形数据库 (WDB) 文件名称为 `wdbFileName`。使用 `NULL` 作为 `xsim.wdb` 的默认名称。如果波形追踪已关闭，那么 Vivado 仿真器会忽略 `wdbFileName` 字段。

提示：为保护您的程序以免将来更改 XSI API，AMD 建议您先将 `s_xsi_setup_info` 结构体置零，然后再填充字段，如 xsi_open 中所示。

该函数的普通（非加载器）形式会返回 `xsiHandle`，此 C 语言对象包含有关设计的进程状态信息，可搭配所有其他普通形式的 XSI 函数一起使用。函数的加载器形式无返回值。但您可以通过查询 `isopen` 成员函数来检查加载器是否已打开设计，如已调用 `open` 成员函数，那么该成员函数会返回 true。

示例

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
info.logFileName = NULL;
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
```

xsi_put_value

```
void xsi_put_value(xsiHandle design_handle, XSI_INT32 port_number, void*
value);
void Xsi::Loader::put_value(int port_number, const void* value);
```

该函数用于将 `value` 中存储的值存入端口 ID `port_number` 所指定的端口。如需了解有关获取端口 ID 的信息，请参阅 xsi_get_port_number。`value` 指针指向您的程序必须分配和填充的存储缓冲器。如需了解有关值的正确格式的信息，请参阅 Vivado 仿真器 VHDL 数据格式 和 Vivado 仿真器 Verilog 数据格式。

注意！为最大程度提升性能，Vivado 仿真器不会对您传递给 `xsi_put_value` 的值的大小和类型执行任何检查。将值传递给 `xsi_put_value` 时，如果其大小和类型与端口的大小和类型不匹配，则可能导致程序和 Vivado 仿真器出现不可预测的行为。

代码示例：

```
#include "xsi.h"
#include "xsi_loader.h"
...
// Hard-coded Buffer for a 1-bit "1" Verilog 4-state value
const s_xsi_vlog_logicval one_val = {0X00000001, 0X00000000};
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

xsi_restart

```
void xsi_restart(xsiHandle design_handle);
void Xsi::Loader:: restart();
```

此函数用于将仿真复位至仿真时间 0。

xsi_run

```
void xsi_run(xsiHandle design_handle, XSI_UINT64 time_ticks);
void Xsi::Loader::run(XSI_INT64 step);
```

该函数会采用内核精度单位按指定的给定时间量来运行仿真。内核精度单位是设计的所有 HDL 源文件之间指定的时间精度的最小单位。例如，如果设计具有两个源文件，其中一个指定精度为 1 ns，另一个指定精度为 1 ps，那么内核精度单位为 1 ps，因为这是两个时间单位间较小的单位。

Verilog 源文件可使用 `` `timescale `` 指令来指定时间精度。

示例：

```
`timescale 1ns/1ps
```

在此示例中，`/` (1 ps) 后的时间单位即时间精度。VHDL 不具有等效的 `` `timescale `` 指令。

此外，您可通过使用 xelab 命令行选项 --timescale、--override_timeprecision 和 --timeprecision_vhdl 来调整内核精度单位。如需了解有关如何使用这些命令行选项的信息，请参阅 xelab、xvhdl 和 xvlog xsim 命令选项。

注释： xsi_run 会保持阻塞，直至耗尽指定的仿真运行时为止。您的程序与 Vivado 仿真器共享单一执行线程。

xsi_trace_all

```
void xsi_trace_all(xsiHandle design_handle);
void Xsi::Loader:: trace_all();
```

在 xsi_open 后调用此函数即可为 HDL 设计的所有信号开启波形追踪。开启波形追踪的情况下运行仿真会导致 Vivado 仿真器生成波形数据库 (WDB) 文件，其中包含设计中每个信号的所有事件。WDB 的默认文件名为 xsim.wdb。要指定不同的 WDB 文件名，请在调用 xsi_open 时设置 s_xsi_setup_info 结构体的 wdbFileName 字段，如以下代码示例所示。

代码示例：

```
#include "xsi.h"
#include "xsi_loader.h"
...
Xsi::Loader loader("xsim.dir/mySnapshot/
xsimk.so","librdi_simulator_kernel.so");
s_xsi_setup_info info;
memset(&info, 0, sizeof(info));
char wdbName[] = "test.wdb"; // make a buffer for holding the string
"test.wdb"
info.wdbFileName = wdbName;
loader.open(&info);
loader.trace_all();
```

仿真完成后，可以在 Vivado 中打开 WDB 文件检验信号的波形。如需了解有关如何在 Vivado 中查看 WDB 文件的更多信息，请参阅 打开先前保存的仿真运行。

重要提示！ 编译 HDL 设计时，必须在 xelab 命令行上指定 -debug all 或 -debug typical。如无 -debug 命令行选项，Vivado 仿真器则不会记录波形数据。

Vivado 仿真器 VHDL 数据格式

本节描述了如何在 VHDL 值之间进行转换，以及搭配 XSI 函数 xsi_get_value 和 xsi_put_value 一起使用的存储缓冲器的格式。

IEEE std_logic 类型

VHDL std_logic 和 std_ulogic 的单个位元在 C/C++ 中呈现为单个字节（字符或无符号字符）。下表显示了 std_logic/std_ulogic 的值及其等效的 C/C++ 值。

表 60：**std_logic/std_ulogic 值及其等效的 C/C++ 值**

std_logic 值	C/C++ 字节值（十进制）
'U ‘	0
‘X ‘	1
‘0 ‘	2
‘1 ‘	3
‘Z ‘	4
‘W ‘	5
‘L ‘	6
‘H ‘	7
‘_ ‘	8

代码示例：

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : std_logic;
const char one_val = 3; // C encoding for std_logic '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL bit 类型

VHDL `bit` 类型的单个位元在 C/C++ 中呈现为单个字节。下表显示了 `bit` 的值及其等效的 C/C++ 值。

表 61：**位元的值及其等效的 C/C++ 值**

bit 值	C/C++ 字节值（十进制）
'0 ‘	0
‘1 ‘	1

代码示例：

```
// Put a '1' on signal "clk," where "clk" is defined as
// signal clk : bit;
const char one_val = 1; // C encoding for bit '1'...
int clk = loader.get_port_number("clk");
loader.put_value(clk, &one_val); // set clk to 1
```

VHDL 字符类型

单个 VHDL `character` 值在 C/C++ 中呈现为单个字节。VHDL `character` 值与 C/C++ `char` 字面值完全相同，并且也与其 ASCII 数值相等。例如，VHDL 字符值 `m` 与 C/C++ `char` 字面值 `m` 或十进制值 109 相等。

代码示例：

```
// Put a 'T' on signal "myChar," where "myChar" is defined as
// signal myChar : character;
const char tVal = 'T';
int myChar = loader.get_port_number("myChar");
loader.put_value(myChar, &tVal);
```

VHDL 整数类型

单个 VHDL `integer` 值在 C/C++ 中呈现为 `int`。

代码示例：

```
// Put 1234 (decimal) on signal "myInt," where "myInt" is defined as
// signal myInt : integer;
const int intVal = 1234;
int myInt = loader.get_port_number("myInt");
loader.put_value(myInt, &intVal);
```

VHDL 实数类型

单个 VHDL `real` 值在 C/C++ 中呈现为 `double`。

代码示例：

```
// Put 3.14 on signal "myReal," where "myReal" is defined as
// signal myReal : real;
const double doubleVal = 3.14;
int myReal = loader.get_port_number("myReal");
loader.put_value(myReal, &doubleVal);
```

VHDL 阵列类型

VHDL 阵列在 C/C++ 中呈现为 C/C++ 类型的阵列，此 C/C++ 类型表示 VHDL 阵列的元素类型。下表显示了 VHDL 阵列及其 C/C++ 等效类型的示例。

表 62：**VHDL 阵列及其 C/C++ 等效类型**

VHDL 阵列类型	C/C++ 阵列类型
`std_logic_vector`（`std_logic` 的阵列）	`char [ ]`
`bit_vector`（`bit` 的阵列）	`char [ ]`
`string`（`character` 的阵列）	`char [ ]`
`integer` 的阵列	`int [ ]`
`real` 的阵列	`double [ ]`

VHDL 阵列在 C/C++ 中的组织方式为：VHDL 阵列的左侧索引映射到 C/C++ 阵列元素 0，右侧索引映射到 C/C++ 元素 <阵列大小> - 1。

C/C++ 阵列索引	0	1	2		<阵列大小> - 1

VHDL 阵列（left TO right）索引	left	left + 1	left + 2		right
VHDL 阵列（left DOWNTO right）索引	left	left – 1	left – 2		right

代码示例：

```
// For the following VHDL definitions
// signal slv : std_logic_vector(7 downto 0);
// signal bv : bit_vector(3 downto 0);
// signal s : string(1 to 11);
// type IntArray is array(natural range <>) of integer;
// signal iv : IntArray(0 to 3);
// do the following assignments
//
// slv <= "11001010";
// bv <= B"1000";
// s <= "Hello world";
// iv <= (33, 44, 55, 66);
const unsigned char slvVal[] = {3, 3, 2, 2, 3, 2, 3, 2}; // 3 = '1', 2 = '0'
loader.put_value(slv, slvVal);
const unsigned char bvVal[] = {1, 0, 0, 0};
loader.put_value(bv, bvVal);
const char sVal[] = "Hello world"; // ends with extra '\0' that XSI ignores
loader.put_value(s, sVal);
const int ivVal[] = {33, 44, 55, 66};
loader.put_value(iv, ivVal);
```

Vivado 仿真器 Verilog 数据格式

Verilog 逻辑数据是使用 `xsi.h` 中定义的以下结构体以 C/C++ 来编码的：

```
typedef struct t_xsi_vlog_logicval {
    XSI_UINT32 aVal;
    XSI_UINT32 bVal;
} s_xsi_vlog_logicval, *p_xsi_vlog_logicval;
```

Verilog 值的每个四态位都占据 `aVal` 中的一个位元位置以及 `bVal` 中的对应位元位置。

表 64：**Verilog 值映射**

Verilog 值	aVal 位值	bVal 位值
0	0	0
1	1	0
X	1	1
Z	0	1

对于二态 SystemVerilog 位值，`aVal` 位用于保存位值，对应 `bVal` 位则不使用。AMD 建议您在为 xsi_put_value 组合二态值时，将 `bVal` 置零。

Verilog 矢量在 C/C++ 中的组织方式为：Verilog 矢量的右索引映射到 `aVal/bVal` 位元位置 0，左索引映射到 `aVal/bVal` 位元位置 <vector size> - 1。

aVal/bVal 位元位置	<vector size> 到 31	<vector size> - 1	<vector size> - 2	...	1	0
索引归属于 `wire [left:right] vec`（其中 left > right）	未使用	left	left - 1	...	right + 1	right
索引归属于 `wire [left:right] vec`（其中 left < right）	未使用	left	left + 1	...	right - 1	right

例如，下表显示了 Verilog 和以下 Verilog 矢量的 C/C++ 等效矢量。

```
wire [7:4] w = 4'bXX01;
```

Verilog 位索引				7	6	5	4
Verilog 位值				X	X	0	1
C/C++ 位元位置	31	...	4	3	2	1	0
`aVal` 位值	未使用	...	未使用	1	1	0	1
`bVal` 位值	未使用	...	未使用	1	1	0	0

含超过 32 个元素的 Verilog 矢量的 C/C++ 表示法是 `s_xsi_vlog_logicval` 阵列，其中，Verilog 矢量最右侧的 32 个位映射到 C/C++ 阵列的元素 0。该 Verilog 矢量的后 32 个位映射到 C/C++ 阵列的元素 1，以此类推。例如，下表显示了将 Verilog 矢量

```
wire [2:69] vec;
```

映射到 C/C++ 阵列的方式

```
s_xsi_vlog_logicval val[3];
```

表 67：**Verilog 索引范围**

Verilog 索引范围	C/C++ 阵列元素
vec[38:69]	val[0]
vec[6:37]	val[1]
vec[2:5]	val[3]

因此，`vec[2]` 映射到 `val[3]` 位元位置 3，`vec[69]` 映射到 `val[0]` 位元位置 0。

多维 Verilog 阵列映射到 `s_xsi_vlog_logicval` 或 `s_xsi_vlog_logicval` 阵列的各个位的方式与在将 Verilog 阵列映射到 C/C++ 之前先将其以按行为主的顺序来平铺是相同的。

例如，二维阵列

```
reg [7:0] mem[0:1];
```

的处理方式与映射到 C/C++ 之前先复制到矢量相同：

```
reg [15:0] vec;
vec[7:0] = mem[1];
vec[8:15] = mem[0];
```

AMD

附录 M

附加资源与法律声明

查找其他文档

技术信息门户网站

AMD 技术信息门户网站是旨在使用您的网页浏览器提供健全的文档搜索和导航的在线工具。要访问该技术信息门户网站，请转至 https://docs.amd.com。

注释：单击链接将打开英语版本，但您可从下拉列表中选择简体中文版本（如可用）。请注意，简体中文版本可能比英语版本旧。

Documentation Navigator

Documentation Navigator (DocNav) 是预安装的工具，支持访问 AMD 自适应计算文档、视频和支持资源，您可在其中通过筛选和搜索来查找信息。要打开 DocNav，请执行以下操作：

- 在 AMD Vivado™ IDE 中，单击“Help” → “Documentation and Tutorials”。
- 在 Windows 上，单击“Start”（开始）按钮并选中“Xilinx Design Tools” → “DocNav”。
- 在 Linux 命令提示中输入 `docnav`。

注释：如需了解有关 DocNav 的更多信息，请参阅《Documentation Navigator 用户指南》(UG968)。

注释：您无法从 DocNav 访问简体中文版本。请使用设计中心网页。

设计中心

AMD 设计中心提供了根据设计任务和其他主题整理的文档链接，可供您用于了解关键概念以及常见问题解答。要访问设计中心，请执行以下操作：

- 在 DocNav 中，单击“Design Hubs View”选项卡。
- 转至设计中心网页。

支持资源

如需获取答复记录、技术文档、下载以及论坛等支持资源，请访问技术支持。

参考资料

以下技术文档是非常实用的补充资料，可配合本指南一起使用：

1. 《Vivado Design Suite 用户指南：版本说明、安装和许可》(UG973)
2. 《Vivado Design Suite 用户指南：系统级设计输入》(UG895)
3. 《Vivado Design Suite 用户指南：采用 IP 进行设计》(UG896)
4. 《Vivado Design Suite 用户指南：使用 Vivado IDE》(UG893)
5. 《Vivado Design Suite 用户指南：使用 Tcl 脚本》(UG894)
6. 《Vivado Design Suite 7 系列 FPGA 和 Zynq 7000 SoC 库指南》(UG953)
7. 《Vivado Design Suite Tcl 命令参考指南》(UG835)
8. 《Vivado Design Suite 用户指南：功耗分析与优化》(UG907)
9. 《Vivado Design Suite 用户指南：使用约束》(UG903)
10. 《Vivado Design Suite 教程：逻辑仿真》(UG937)
11. 《Vivado Design Suite 用户指南：设计流程概述》(UG892)
12. 《Vivado Design Suite 属性参考指南》(UG912)
13. 《Vivado Design Suite 用户指南：综合》(UG901)
14. 《编写高效的测试激励文件》(XAPP199)
15. 《IEEE 标准 VHDL 语言参考手册》(IEEE-STD-1076-1993)
16. 《IEEE 标准 Verilog 硬件描述语言》(IEEE-STD-1364-2001)
17. 《适用于 SystemVerilog 的 IEEE 标准 - 统一硬件设计、规范和验证语言》(IEEE-STD-1800-2009)
18. 《标准延迟格式规范 (SDF)》(IEEE-STD-1497-2004)
19. 《推荐的电子产品设计 IP 加密与管理实践》(IEEE-STD-P1735)

指向第三方仿真器相关附加信息的链接

1. Questa Advanced Simulator/ModelSim 仿真器：
   - http://www.mentor.com/products/fv/questa/
   - http://www.mentor.com/products/fv/modelsim/
2. Synopsys VCS 仿真器： https://www.synopsys.com/verification/simulation/vcs.html
3. Active-HDL 仿真器： https://www.aldec.com/en/support/resources/documentation/articles/1579
4. Riviera PRO 仿真器：https://www.aldec.com/en/support/resources/documentation/articles/1525

培训资料

AMD 提供多种多样的培训课程和 QuickTake 视频，可帮助您进一步了解本文档中提出的概念。使用以下链接获取相关培训资料：

1. 使用 Vivado Design Suite 设计 FPGA 1 培训课程
2. 使用 Vivado Design Suite 设计 FPGA 2 培训课程
3. 使用 Vivado Design Suite 设计 FPGA 3 培训课程
4. Vivado Design Suite QuickTake 视频：如何使用 Zynq 7000 Verification IP 通过仿真来进行验证和调试
5. Vivado Design Suite QuickTake 视频教程：逻辑仿真
6. Vivado Design Suite QuickTake 视频教程

修订历史

下表列出了本文档的修订历史。

章节	修订综述
2024 年 5 月 30 日 2024.1 版	
常规更新。	整个文档。

请阅读：重要法律声明

本文档所示信息仅做参考，其中可能包含不准确的技术信息、疏漏和印刷错误。受诸多原因影响，此处所含信息可能发生更改，也可能无法准确呈现，这些原因包括但不限于产品和路线图变更、组件和主板版本更改、新增模型和/或产品发布、不同制造商之间存在的产品差异、软件更改、BIOS 刷新、固件升级等。任何计算机系统均存在安全性漏洞风险，无法彻底阻止也无法缓解这类风险。AMD 没有任何义务来更新或者以任何其他方式纠正或修改这些信息。但 AMD 保留随时修改这些信息和更改文档内容的权利，AMD 没有任何义务将此类修改或更改通知任何人。此处信息“按原样”提供。AMD 对于本文档内容不作任何陈述或保证，并且对于这些信息中可能出现的任何不准确、错误或疏漏问题不承担任何责任。对于有关任何暗含的非侵权、适销性及适合特定用途的保证，AMD 特此声明不承担任何责任。无论在任何情况下，对于任何人因使用此处包含的任何信息而形成的依赖或者引发的任何直接、间接、特殊或其他后果性损害，AMD 概不负责，即使 AMD 已明确获悉存在发生此类损害的可能性也是如此。

关于与汽车相关用途的免责声明

如将汽车产品（部件编号中含“XA”字样）用于部署安全气囊或用于影响车辆控制的应用（“安全应用”），除非有符合 ISO 26262 汽车安全标准的安全概念或冗余特性（“安全设计”），否则不在质保范围内。客户应在使用或分销任何包含产品的系统之前为了安全的目的全面地测试此类系统。在未采用安全设计的条件下将产品用于安全应用的所有风险，由客户自行承担，并且仅在适用的法律法规对产品责任另有规定的情况下，适用该等法律法规的规定。

版权声明

© Copyright 2012-2024 AMD 公司，版权所有。AMD、AMD 箭头标识、Artix、Kintex、UltraScale、UltraScale+、Versal、Virtex、Vivado、Zynq 及其组合均为 Advanced Micro Devices, Inc. 的商标。“AMBA”、“AMBA Designer”、“Arm”、“ARM1176JZ-S”、“CoreSight”、“Cortex”、“PrimeCell”、“Mali”和“MPCore”为 Arm Limited 在美国和/或其他国家或地区的商标。“PCI”、“PCIe”和“PCI Express”均为 PCI-SIG 拥有的商标，且经授权使用。 此出版物中所使用的其他产品名称仅用于标识目的，可能是其各自所属公司的商标。