cadence®

Xcelium Simulator User Guide

Product Version XCELIUM AGILE
December 2024

Contents

1

Introduction to the Xcelium Simulator

The Cadence® Xcelium™ Logic Simulator implements a single-core engine that combines the high-performance of compiled code simulation with the accuracy, flexibility, and debugging capabilities of event-driven simulation.

With the Xcelium Simulator, all supported simulation styles leverage this high-performance engine. Optimizing compilers for each input language or format creates a sequence of instructions that are interleaved into a single, contiguous code stream. This code stream is effectively a custom-built engine for the specific blend of simulation languages or techniques represented by a particular design.

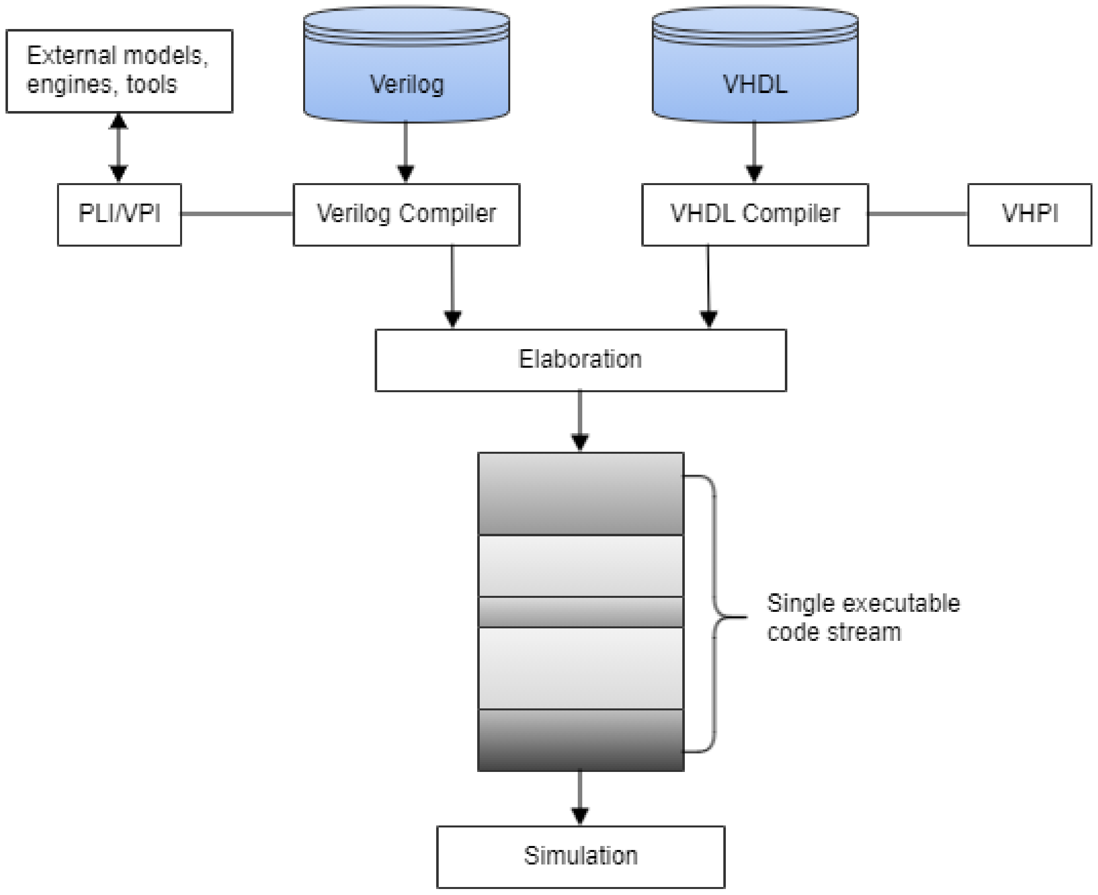
The supported simulation styles include:

- **XRUN Invocation:** This uses the *xrun* utility to specify all input files and command-line options. The *xrun* utility improves usability by taking files from different simulation languages. such as Verilog, SystemVerilog, VHDL, Verilog AMS, VHDL AMS, Specman ***e***, and files written in general programming languages like C and C++. Both single-step and multi-step *xrun* commands are valid.
- **Direct Invocation:** This uses explicit executables to parse, compile, elaborate, and simulate a design.
    - The *xmvlog* and *xmvhdl* utilities parse and compile files from different simulation languages, such as Verilog (`xmvlog`), SystemVerilog (`xmvlog -sv`), VHDL (`xmvhdl`), Verilog AMS (`xmvlog -ams`), and VHDL AMS (`xmvhdl -ams`). If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single packed library database file in the work library directory named `xcelium.d/worklib`.
    - The *xmelab* utility is a language-independent elaborator. It constructs a hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot, which is the representation of your design that the simulator uses to run the simulation. The snapshot is stored in the library database file along with the other intermediate objects generated by the compiler and elaborator.

- The *xmsim* utility loads the snapshot as its primary input. It then loads other intermediate objects referenced by the snapshot. In the case of interactive debugging, HDL source files and script files can also be loaded. Other data files can be loaded as demanded by the model being simulated (using `$read*` tasks or Textio). The output of the simulation is controlled by the model or debugger. This output can include result files generated by the model, Simulation History Manager (SHM) databases, Value Change Dump (VCD) files, and so on.

The following figure illustrates the simulation flow for a mixed Verilog/VHDL configuration:

Figure 1.1: Interleaved Code Generation



Related Topics

- Xcelium Product Licensing
- Xcelium Multi-Core Simulator User Guide
- Xcelium Simulator Tcl Command Reference
- Xcelium Design Simulation
- Xcelium Advanced Features

Xcelium Advanced Features

The latest iteration of the Xcelium Simulator leverages single-core and multi-core simulation technology for improved regression throughput. It is also optimized for advanced features like low-power simulation, multi-snapshot incremental elaboration, mixed-signal verification, X-propagation, and functional safety. These advanced features are summarized below.

Low-Power Simulation

Including a power intent file as another layer of abstraction enables advanced low-power techniques that entail using multiple, often switchable, power supplies. A design is split into basic units called power domains, which are a collection of instances in the design that share the same primary supplies, like power and ground. The supply for each domain can be shut off when that part of the design is not required to be functioning. This is called *power shutoff* or *power gating*. Power gating can be implemented with or without state retention; that is, keeping some or all of the registers powered during shutoff to facilitate faster recovery when power is restored. Each power domain also needs to be fully separated from other power domains, using appropriate isolators or level shifters on every signal that crosses the domains. Each domain can also be supplied with different voltage levels permanently, a technique called *multi-supply voltage* (MSV), or supplied with different voltage levels dynamically, a technique called *dynamic voltage and frequency scaling* (DVFS).

When enabling the low-power flow with, you must use one of the two available formats:

- The *Unified Power Format* (UPF) is the modern standard introduced in 2006 and approved by the IEEE 1801 committee in 2009.

  The latest iteration of the UPF standard is IEEE 1801-2018.
- The *Common Power Format* (CPF) is the legacy standard introduced by Cadence in 2005.

Multi-Snapshot Incremental Elaboration

Multi-snapshot incremental elaboration (MSIE) provides a form of parallel and incremental elaboration that can greatly decrease the time, elaboration memory footprint, and storage space required to elaborate the design. Depending on your environment, there can be various types of requirements to reduce the elaboration time. For instance, the first and simplest requirement is cutting down the monolithic elaboration time for each scratch build and/or reducing the next run of incremental build time. Other requirements include elaborating the different stable/unchanging and changing parts of the design separately and reusing them in the next runs. All requirements call for choosing the right MSIE flow.

The MSIE flows are implemented throughout the entire design, mainly, using two areas of the source code. Specifically:

- Parts of the design that are stable, requiring only occasional changes, are placed in one or more sections called ***primary partitions/snapshots***. The primary partitions contain portions of the design that can be used repeatedly without being re-elaborated when other portions of the design change.

  A primary partition is elaborated into a primary snapshot. The elaborator recognizes each primary snapshot as a new type of design unit. The Primary Snapshot (PRM) design unit is used along with Verilog Syntax Tree (VST) and Abstract Syntax Tree (AST) design units to create a simulation model.

- If there are any parts of the design that are not elaborated into a primary partition, they become part of the ***incremental partition/snapshot***. These are usually parts of the design that are small or that may change frequently.

  The incremental partition is elaborated into the simulation snapshot. A simulation snapshot that instantiates at least one primary snapshot is considered a multi-snapshot model and is using MSIE technology. Otherwise, it is just a normal snapshot.

Mixed-Signal Verification

Cadence's mixed-signal verification flow and methodology bring together the analog and digital sides. Integrating analog behavior modeling and analog and digital solvers into one flow, the methodology lets you balance the right amount of accuracy and speed based on your design requirements. The Xcelium Simulator with mixed-signal option:

- Provides design & verification solutions for analog, RF, memory, and mixed-signal SoCs.
- Covers advanced digital features such as UVM, SV Testbench, UPF/CPF, and SystemC®.
- Supports the simulation of languages such as Verilog-A, Verilog-AMS, VHDL-AMS,

SystemVerilog Real Number Modeling (SVRNM), SystemVerilog (SV) and mixed-signal features along with SPICE, and other digital-centric mixed-signal technologies, such as low-power + MS, code coverage + MS, functional safety +MS, and incremental elaboration + MS.

X-Propagation

Standard Verilog semantics define X as an *unknown* value. This value appears when simulations cannot acutely resolve signals to 1 or 0. Unfortunately, RTL simulations often mask X values by converting unknown values into known values, while gate-level simulations show additional Xs that will not exist in the real hardware. The simulation semantics that converts unknown (X) values to known values (for example, 0 or the false branch in an `if` statement) is a problem at RTL called X-optimism. These semantic differences can result in X bugs slipping to real silicon or getting detected only during gate-level simulation. And unresolved X issues can turn costly, adding complications as you familiarize yourself with the abstract synthesized logic while consuming time as you exhaustively simulate and synthesize designs for verification purposes.

Xcelium Safety

Xcelium Safety is part of the Cadence Safety Solution targeting safety-critical applications for faster ISO 26262 certification. It provides high-performance native engines for serial and concurrent fault injection and simulation. The native twin engines are complementary: the concurrent engine provides best-in-class throughput and is very useful in running the entire fault campaign, while the serial engine is useful in flow setup and fault debug. Both engines use an identical flow, and the command-line interface makes it easy to switch back and forth.

The native integration of the fault simulation engines enables unified compile, re-use of UVM testbench, and flows for both functional and safety verification, enhancing performance, productivity, and throughput, significantly. As part of the Cadence Safety Platform, the safety simulator is also integrated with the Midas Safety Platform for FMEDA-based functional safety verification. Fault campaign management (FCM) can be applied for optimized coverage-based test selection, smart run algorithms to reduce fault simulation time, and fault pruning using the JasperGold FSV App. The fault result database is also unified for merging diagnostic coverage across different engines as well as hierarchical merge from the IP to SoC level.

Related Topics

- Low-Power Simulation Guide (IEEE 1801)
- Multi-Snapshot Incremental Elaboration
- Using X-Propagation to Solve X-Optimism
- SystemC Simulation User Guide
- Xcelium Fault Simulator User Guide

2

Setting Up Your Environment

With Xcelium, all products are installed in a single hierarchy. Before you configure your environment for 32-bit or 64-bit mode, make a note of the path to the directory that contains the installation. Substitute the appropriate path wherever you see *`cds_install_path`* in the sections below.

When running the simulator, no environment configuration files are required; however, you can choose to use up to three files that help to manage your data and control the operation of the various Xcelium tools and utilities:

- `cds.lib`: Defines your design libraries and associates logical library names with physical library locations.
- `hdl.var`: Defines variables that affect the behavior of tools and utilities.
- `setuploc`: Specifies the search order that tools and utilities use when searching for the `cds.lib` and `hdl.var` files.

Configuring Your Environment for the 32-Bit Version

The 32-bit version of Xcelium is the default. To run this version of the software, you must do two things:

- Set the path variable so that it includes the path to the executables, which are in *`cds_install_path`*`/tools/bin`.

  ```
  set path = (cds_install_path/tools/bin $path)
  ```

- Set the following environment variable:

  ```
  CDS_LIC_FILE
  ```

  Set this variable so that it points to the port number and hostname of the machine on which the license daemon is running, as specified by the SERVER line in the license file. For example:

```
    setenv CDS_LIC_FILE 5280@hostname
```

When you run an executable in 32-bit mode, no special information is displayed saying that you are running in 32-bit mode. For example:

```
% xmvhdl -messages fa1.vhd
xmvhdl: 23.03-s001: (c) Copyright 1995-2023 Cadence Design Systems, Inc.
...
```

Configuring Your Environment for the 64-Bit Version

All Xcelium tools must be run in either 32-bit or 64-bit mode. For instance, you cannot compile HDL source files in 64-bit mode and then invoke the elaborator in 32-bit mode. To run the 64-bit version of the simulator, you must complete one of the following three tasks:

- Set the path to point to the 64-bit version.
- Set the path to point to the 32-bit version and then set either the `INCA_64BIT` or `CDS_AUTO_64BIT` environment variable.
- Set the path to point to the 32-bit version and then include the `-64bit` option on the command line when you run each of the simulator executables.

Setting Up Your PATH for the 64-Bit Simulator

If you want to run Xcelium in 64-bit exclusive mode, meaning that you do not intend to switch between 32-bit and 64-bit modes, this is the recommended configuration.

The 64-bit executables are located in *cds_install_path*`/tools/bin/64bit`.

Set the PATH variable so that the path to the 64-bit version precedes the path to the 32-bit version. For example:

```
set path = (cds_install_path/tools/bin/64bit cds_install_path/tools/bin $path)
```

Setting the INCA_64BIT or CDS_AUTO_64BIT Environment Variable

If you configure your Xcelium environment for the default 32-bit mode, you can then set either the `INCA_64BIT` or `CDS_AUTO_64BIT` environment variable to enable 64-bit mode. The advantage of using these environment variables is that you do not have to include an additional command-line option whenever you invoke the simulator.

- The `INCA_64BIT` variable is treated as a Boolean. You can set this variable to any value or to a null string.

  ```
  setenv INCA_64BIT
  ```

  ⚠ The `INCA_64BIT` environment variable does not affect other Cadence tools, such as IC tools. However, for Xcelium tools, the `INCA_64BIT` variable overrides the setting for the `CDS_AUTO_64BIT` environment variable. If the `INCA_64BIT` environment variable is set, all Xcelium tools will be run in 64-bit mode.

- The `CDS_AUTO_64BIT` variable is set to `INCLUDE:INCA`. The string `INCA` must be uppercase.

  ```
  setenv CDS_AUTO_64BIT INCLUDE:INCA
  ```

  Other Cadence tools, such as IC tools, also use the `CDS_AUTO_64BIT` environment variable to control the selection of 32-bit or 64-bit executables. The following table shows how you can set the `CDS_AUTO_64BIT` variable to run the Xcelium tools and IC tools in all modes.

  CDS_AUTO_64BIT Variable Settings

CDS_AUTO_64BIT Variable	Xcelium Tools	IC Tools
`setenv CDS_AUTO_64BIT ALL`	64-bit	64-bit
`setenv CDS_AUTO_64BIT NONE`	32-bit	32-bit
`setenv CDS_AUTO_64BIT EXCLUDE:`*`ic_binary`*	64-bit	32-bit
`setenv CDS_AUTO_64BIT EXCLUDE:INCA`	32-bit	64-bit

  Because all Xcelium tools must be run in either 32-bit mode or in 64-bit mode, do not use INCLUDE/EXCLUDE to include or exclude a specific binary, as in the following

  ```
  setenv CDS_AUTO_64BIT EXCLUDE:xmelab
  ```

  ⚠ If you set the `CDS_AUTO_64BIT` variable to exclude the Xcelium tools (`setenv CDS_AUTO_64BIT EXCLUDE:INCA`), all Xcelium tools are run in 32-bit mode. However, the `-64bit` command-line option overrides the environment variable.

To confirm which version you are currently set up to run, use the `xmbits` command. This command returns either 32 or 64. For example, if you have set up your path variable to run the 64-bit version, or if you have set the `INCA_64BIT` variable, the output of the `xmbits` command is as follows:

```
% xmbits
64
```

When you invoke a tool in 64-bit mode, the number 64 is shown on the copyright banner. For example, the following shows the banner if you run 64-bit xmelab:

```
% xmelab -messages -64bit worklib.top:module
xmelab(64): 23.03-s001: (c) Copyright 1995-2023 Cadence Design Systems, Inc.
```

If you have set your path variable so that you are pointing to the 64-bit version, or have set the `INCA_64BIT` environment variable, you can also use the `-version` option to see if you are configured to run the 64-bit executables. For example,

```
% xmelab -version
TOOL:  xmelab(64)      23.03-s001
```

Using the -64bit Command-Line Option

When you configure your Xcelium environment for the default 32-bit mode, you can also choose to enable 64-bit mode by specifying the `-64bit` option when you run a simulator command. For instance, if you are using *xrun*, include the `-64bit` option on the command line as below.

```
% xrun -64bit [other_options] input_files
```

If you are running in multi-step direct invocation mode, include the `-64bit` option when running each unique `xm*` command:

```
% xmvlog -64bit [other_options] verilog_source_files

% xmvhdl -64bit [other_options] vhdl_source_files

% xmsc -64bit [other_options] source_files

% xmsc_run -64bit [other_options] source_files

% xmelab -64bit [other_options] top_level_design_unit

% xmsim -64bit [other_options] snapshot_name
```

The `-64bit` option must be specified on the command line. You cannot include this option in an `hdl.var` file.

Related Topic

- Xcelium Library and Directory Structure

Xcelium Library and Directory Structure

The Xcelium library structure is organized according to a `Library.Cell:View` approach. These terms are explained below:

- **Library:** A collection of related cells that describe components of a single design (a *design library*) or common components used in many designs (a *reference library*). Each library is referenced by a logical name and has a unique physical directory associated with it. You can choose to define library names and map them to physical directories in the `cds.lib` file. The library used for your current design work is called the *working* or *work* library. You can choose to define the current work library either by defining the `WORK` variable in the `hdl.var` file or by using the `-work` command-line option.
- **Cell:** A cell is an object with a unique name stored in a library. Each Verilog module, macro module, or UDP, or each VHDL entity, architecture, package, package body, or configuration is a unique cell. The internal intermediate objects necessary to represent a cell are contained in the library database file (`.pak` file) stored in the library directory.
- **View:** A view is a version of a cell. These can be used to delineate between representations (schematic, VHDL, Verilog), abstraction levels (behavior, RTL, post-synthesis), status (experimental, released, golden), and so on. the internal intermediate objects necessary to represent a view are contained in the library database file (`.pak` file) stored in the library directory. For example, one view might contain the RTL representation of a particular Verilog module, while another view stores the behavioral representation. You could also have two different versions of the same cell—one with timing and one without timing.

Directory Structure

The generated directory structure is dependent on whether you run the simulator using the supported *xrun* invocation or traditional (direct invocation) mode.

By default, using *xrun invocation*:

- Xcelium automatically creates a work library called `worklib` in the directory called `xcelium.d`. The work library contains one `.pak` file.
- The *xrun* utility also creates a scratch subdirectory under the `xcelium.d` directory called:

```
run.<platform|platform.64>.<xrun_version>.d
```

By default, using *direct invocation*:

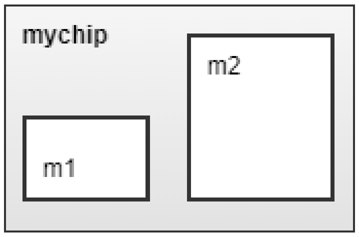
- Xcelium automatically creates a work library called `worklib` in the directory called `xcelium.d`. The work library contains one `.pak` file.

> In direct invocation mode, the compilation, elaboration, and simulation steps are executed by separate Xcelium commands.

- In this mode, no additional scratch subdirectory is created under `xcelium.d`.

Verilog Example

Consider a Verilog design where a module `mychip` instantiates two other modules, `m1` and `m2`.

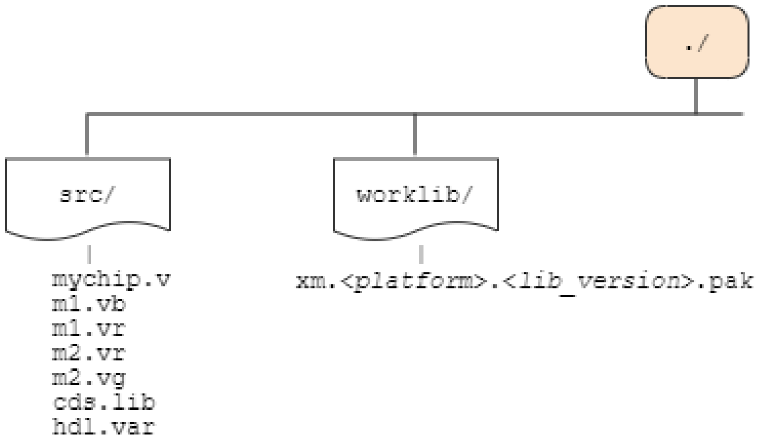


A single Verilog description of `mychip` is contained in the file `mychip.v`, but multiple descriptions of `m1` and `m2` have also been generated.

- For `m1`, there is both a behavioral and an RTL description, described in `m1.vb` and `m1.vr`, respectively.
- For `m2`, there is both an RTL and a synthesized gate-level representation, described in `m2.vr` and `m2.vg`, respectively.

Cell	Files	View
`mychip`	`mychip.v`	Structural
`m1`	`m1.vbm1.vr`	Behavioral/RTL
`m2`	`m2.vrm2.vg`	RTL/Gates

All of these source files reside in the `src` subdirectory, from which all tools are invoked. By default, the work library is saved in the `xcelium.d` directory; however, you can also choose to create a subdirectory at the same level as `src` and use that as the work library.



The `cds.lib` file located in the `src` directory includes the following statement which defines this alternate `worklib` for simulation:

```
# cds.lib file
DEFINE worklib ../worklib
```

The `hdl.var` file located in the `src` directory includes definitions of the `LIB_MAP` and `VIEW_MAP` variables, which can be used to specify the library and view mapping for Verilog design units.

⚠ The `LIB_MAP` and `VIEW_MAP` variables do not apply to VHDL.

```
# Define library mapping.
# Compile all files in src into worklib
DEFINE LIB_MAP (./ => worklib)

# Define view mapping.
# Files with .vb extension are compiled into view beh
# Files with .vr extension are compiled into view rtl
# Files with .vg extension are compiled into view gates
# Files with .v extension are compiled into view module
DEFINE VIEW_MAP (.vb => beh, \
                 .vr => rtl, \
                 .vg => gates, \
                 .v => module)
```

If you want to simulate the complete design, you can specify the following single `xrun` command:

```
% xrun -vlog_ext +.vb,.vg,.vr -top mychip mychip.v m1.vb m1.vr m2.vg m2.vr
```

Here, the `-vlog_ext` option adds `.vb`, `.vg`, and `.vr` to the list of legal Verilog file extensions. The -

`top` option is required by *xrun* to set the top-level design unit for elaboration and simulation.

Alternatively, you can use different commands to compile, elaborate, and simulate your design (also known as, direct invocation).

Start by invoking `xmvlog` to compile the design units in the available source files.

```
% xmvlog mychip.v m1.vb m1.vr m2.vg m2.vr
```

When the design is compiled, using either *xrun* or *xmlvog*, a cell and view are created for each module.

- The file `mychip.v` gets compiled into the default module view. The design unit in *`Lib.Cell:View`* notation is `working.mychip:module`.
- Each of the RTL representations, `m1.vr` and `m2.vr`, is compiled into its respective `rtl` view: `worklib.m1:rtl` and `worklib.m2:rtl`.
- The behavioral representation of `m1`, described in the file `m1.vb`, is compiled into `worklib.m1:beh`.
- The gate-level representation of `m2`, described in `m2.vg`, is compiled into `worklib.m2:gates`.
- The library directory (`worklib`) contains one `.pak` file that contains all of the intermediate objects created by the compiler.

To elaborate the design after invoking `xmvlog`, use the following `xmelab` command to specify the top-level module called `mychip` using *`Lib.Cell:View`* notation:

```
% xmelab worklib.mychip:module
```

Because there is only one view of `mychip` in the library, you can choose to omit the library and view specification as shown:

```
% xmelab mychip
```

When using either *xrun* or *xmelab*, the elaborator generates a simulation snapshot for the design. Intermediate objects created during this elaboration phase are stored in the `.pak` file. The snapshot is also a *`Lib.Cell:View`*. In this example, the snapshot is called `worklib.mychip:module`.

To simulate the design after invoking `xmelab`, use the following `xmsim` command to specify the name of the snapshot:

```
% xmsim worklib.mychip:module
```

Because there is only one snapshot in the library, you can choose to omit the library and view specification:

```
% xmsim mychip
```

Related Topics

- Managing Libraries Using the -v/-y Scheme
- XRUN Invocation
- Direct Invocation
- The cds.lib File
- The hdl.var File
- The setup.loc File

The cds.lib File

The `cds.lib` file is an ASCII text file that defines which libraries are accessible and where they are located. The file contains statements that map library logical names to their physical directory paths. During initialization, all tools that need to understand library names read the `cds.lib` file and compute the logical to physical mapping.

You can create a `cds.lib` file with any text editor. The following examples show how library bindings are specified in the `cds.lib` file with the `DEFINE` statement.

keyword	logical library name	physical location
`DEFINE`	*`lib_std`*	*`/usr1/libs/std_lib`*
`DEFINE`	*`worklib`*	*`../worklib`*

The `cds.lib` file cannot map the same library logical name to multiple physical paths. The tools issues a warning message and uses the last specified path. In the following example, `lib1` is mapped to `./otherlib`.

```
DEFINE lib1 ./lib
DEFINE lib1 ./otherlib
```

Multiple logical names can be mapped to the same physical location, as in the following example.

```
DEFINE lib1 ./pci_lib
DEFINE lib2 ./pci_lib
DEFINE lib3 ./pci_lib
```

While this is not a recommended practice, it can be useful in designs that contain legacy code. It is also useful in situations where multiple users are using the same library, but have defined different logical names in their respective `cds.lib` files, and have references to the library in their code.

You can have more than one `cds.lib` file. Use the `INCLUDE` or `SOFTINCLUDE` statements to include a `cds.lib` file within a `cds.lib` file. If you are doing a pure VHDL or a mixed-language simulation, you must use the `INCLUDE` or `SOFTINCLUDE` statement in the `cds.lib` file to include the default `cds.lib` file located in:

```
install_directory/tools/xcelium/files/cds.lib
```

This `cds.lib` file contains a `SOFTINCLUDE` statement to include a file called `cdsvhdl.lib`, which defines the Synopsys IEEE libraries included in the release. If you want to use the IEEE libraries that were shipped with Version 2.1 of the simulator instead of the Synopsys libraries, you must include the `cds.lib` file located in:

```
install_directory/tools/xcelium/files/IEEE_pure/cds.lib
```

A `cds.lib` file is not required to run the simulator. If you do not create a `cds.lib` file, the Synopsys libraries are used. All tools and utilities that require a `cds.lib` file use the same search mechanism to find the `cds.lib` file. Each tool that reads a `cds.lib` file also has a `-cdslib` option that you can use on the command line to specify a `cds.lib` file. This option overrides the default search mechanism.

Related Topics

- cds.lib Statements and Rules
- The Work Library
- The hdl.var File

cds.lib Statements and Rules

The following statements are supported for use in the `cds.lib` file:

- `DEFINE` *`<lib_name> <path>`*
- `UNDEFINE` *`<lib_name>`*
- `INCLUDE` *`<file>`*
- `SOFTINCLUDE` *`<file>`*
- `ASSIGN` *`<lib> <attribute> <path>`*
- `UNASSIGN` *`<lib> <attribute>`*

Statement	Description	Example

`DEFINE <lib_name> <path>`	Associates the logical library name specified with the *`lib_name`* argument with the physical directory path specified with the *`path`* argument.
	
	You cannot specify the same directory in multiple library definitions.
		`DEFINE ttl_lib /usr1/libraries/ttl_lib`
		`DEFINE ttl ./libraries/ttl`

`UNDEFINE <lib_name>`	Undefines the specified library. This command is used for removing any libraries that were defined in other files. If the specified *`lib_name`* was not previously defined, no error is generated.	`UNDEFINE ttl`
`INCLUDE <file>`	Reads the specified file as an addendum to the source `cds.lib` file. Use this statement to include the library definitions contained in the specified file. The included file does not have to be named `cds.lib`. An error message is printed if the file is not found or if a recursion is detected.	The following includes the `cds.lib file in /users/$USER`: `INCLUDE /users/$USER/cds.lib`
`SOFTINCLUDE <file>`	This statement works the same as the `INCLUDE` statement, except that no error messages are printed if the file does not exist.	The following includes the `cds.lib` file in the `$GOLDEN` directory: `SOFTINCLUDE $GOLDEN/cds.lib`

`ASSIGN <lib> <attribute> <path>`	Assigns an attribute to the library.
	
	⚠ `TMP` is the only supported attribute.
		The following defines the `iclib` library and assigns the attribute `TMP` to it. The value of `TMP` is `./ic_tmp_lib`.
		
		Example:
		
		`DEFINE iclib ./ic_lib`
		`ASSIGN iclib TMP ./ic_tmp_lib`

`UNASSIGN <lib> <attribute>`	Removes an assigned attribute from the library. No error is generated if the attribute has not been assigned to the library. However, if the library has not been defined, an error is generated. ⚠ `TMP` is the only supported attribute.	`UNASSIGN iclib TMP`

Rules

The following rules apply to the *cds.lib* file:

- Only one statement per line is allowed.
- Blank lines are allowed.
- Use the pound sign (#) or the double hyphen (--) to begin a comment. You must precede and follow the comment character with white space, a tab, or a new line.
  Examples: # this is a comment -- this is another comment.
- Keywords are identified as the first non-whitespace string on a line.
- Keywords and attributes are case insensitive.
- You can include symbolic variables (UNIX environment variables like $HOME and CSH

extensions such as ~ and ~user).

- Symbolic variables and library pathnames are in the file system domain and are case sensitive.
- You can enter absolute or relative file paths. Relative paths are relative to the location of the file in which they occur, not to the directory where the tool was invoked.
- Library names and path names reside within the file system name-space. For Verilog, non escaped library names are the same as the Verilog name; for VHDL, non escaped library names are resolved to lower-case. You cannot directly use escaped library names in a *cds.lib* file. To use an escaped name, run the `nmp` program in the *install_directory /tools/bin* directory to see how escaped library names are mapped to file system names. Then use the mapped name in the *cds.lib* file. The syntax for the `nmp` program is as follows.
  Remember the trailing space. `% nmp mapName {Verilog | NVerilog | Vhdl} Filesys` `` `\ illegal_name `` 'For example, to use the library named `Lib*`, you must use the library's escaped name format (`\Lib*`), since `" * "` is an illegal character. To determine the mapped file system name for `\Lib*, type: % nmp mapName Verilog Filesys '\Lib*` ' The `nmp` program returns `Lib#2a`. Use the mapped name (Lib#2a) in the *cds.lib* file.

Example

The following example contains most of the statements you can use in a *cds.lib* file. Comments begin with the pound sign (#).

```
# Assign /usr1/libraries/ic_library to the logical library name ic_lib

DEFINE ic_lib /usr1/libraries/ic_library

# Specify a relative path to library aludesign. The path is relative to this# cds.lib
file

DEFINE aludesign ./design

# Read cds.lib from the /users/$USERS directory.

INCLUDE /users/${USER}/cds.lib

# Read cds.lib from the $CADLIBS directory.

SOFTINCLUDE ${CADLIBS}/cds.lib

# Define a temporary directory and assign the TMP attribute to it. The directory#
```

```
./temp must exist and templib must be set to WORK in the hdl.var file in order# to
compile data into it.

DEFINE templib ./temp_lib

ASSIGN templib TMP ./temp
```

Related Topics

- The Work Library
- The hdl.var File

The Work Library

The library used for your current design work is called the work or working library. The work library is the library in which design units are compiled. Like other libraries, the directory path of the work library is defined in the `cds.lib` file.

There are several ways to specify which library is the work library. For Verilog, you can use:

- compiler directives in the source file
- `-work` command-line option, or
- variables defined in the `hdl.var` file

For VHDL, define the `WORK` variable in the `hdl.var` file or use the `-work` option on the command line.

Related Topic

- The cds.lib File

Binding One Library to Multiple Directories

To bind a library defined in the *cds.lib* file to a temporary storage directory, use the `ASSIGN` statement to assign the `TMP` attribute to the library. This allows multiple designers to reference a shared library, but store intermediate objects generated by the compiler or elaborator in separate design directories. When intermediate objects are read, the tools read whatever intermediate objects they need from the original library, and, if the objects are not in the original library, from the `TMP` library.

In the following example, a library called `asic_lib` is defined as `${PROJECT}/asic_lib`. A temporary storage directory called `work/design_lib` is created, and the `TMP` attribute is then assigned to `asic_lib` to bind this library to the temporary storage directory.

```
# Define the shared library

DEFINE asic_lib ${PROJECT}/asic_lib

# Assign a temp storage directory

ASSIGN asic_lib TMP ./work/design_lib
```

When you compile and elaborate a design that includes design units from the shared library, all new intermediate objects are stored in the `TMP` library instead of in the `asic_lib` library. Only one directory can be bound to a master library using the `TMP` attribute.

In the *cds.lib* file, you must define the library before referencing it with the `ASSIGN` statement. If the referenced library has not been defined before the `ASSIGN` statement is processed, the statement is ignored with a warning. Use the `UNASSIGN` statement to remove the `TMP` attribute before compiling your design units into the master library.

ⓘ If the elaborator needs to produce new intermediate files for a design unit that is in a read-only library with no explicit `TMP` library assigned, it automatically creates a `TMP` library and writes the intermediate files into this `TMP` library. If you have used the `-messages` option, the elaborator generates a message for each implicit `TMP` library that it creates.

These implicit TMP libraries are co-located with the design library that contains the snapshot produced by the elaborator. Each directory for an implicit `TMP` library is named `xm.`*`library_name`* (for example, `inca.std`, `inca.ieee`, `inca.userlib`).

When a snapshot is loaded, any required intermediate files are searched for in the following order:

- The original read-only shared library
- An explicit `TMP` library associated with the shared library, if this exists
- An implicit `TMP` library

Related Topics

- Xcelium Library and Directory Structure
- The cds.lib File
- Debugging cds.lib Files

Debugging cds.lib Files

Use the `xmhelp -cdslib` command to display information on the contents of `cds.lib` files. This helps identify errors and any incorrect settings within your `cds.lib` files.

The syntax and examples are shown below.

Syntax:

```
% xmhelp -cdslib [ cds.lib_file ]
```

Examples:

```
% xmhelp -cdslib
% xmhelp -cdslib ~/cds.lib
% xmhelp -cdslib ~/design/cds.lib
```

The following example shows how to display information about the contents of `cds.lib` files which is in the current working directory. The `xmhelp -cdslib` command displays the contents of the `cds.lib` file that would be used.

```
%xmhelp -nocopyright -cdslib
```

Parsing the ./cds.lib

```
cds.lib files:        // The cds.lib file in the working directory
includes
                      // the cds.lib file in tools/xcelium/files under the
                      // installation directory. That cds.lib file includes
                      // two other files.
```

1. `./cds.lib`
2. `/usr1/larrybird/nccoex/tools/xcelium/files/cds.lib`
   Included on line 2 of `./cds.lib`.
3. `/usr1/larrybird/nccoex/tools/xcelium/files/cdsvhdl.lib`
   Included on line 2 of `/usr1/larrybird/nccoex/tools/xcelium/files/cds.lib`.
4. `/usr1/larrybird/nccoex/tools/xcelium/files/cdsvlog.lib`
   Included on line 3 of `/usr1/larrybird/nccoex/tools/xcelium/files/cds.lib`.

Libraries defined in ./cds.lib

```
Line #  Filesys       Verilog  VHDL          Path

------  -------       -------  ----          ----

   1    worklib       worklib  WORKLIB       ./xcelium.d/worklib
```

Defined in `/usr1/larrybird/nccoex/tools/xcelium/files/cdsvhdl.lib`:

```
Line #  Filesys       Verilog       VHDL          Path

------  -------       -------       ----          ----

   1    std           std           STD
 /usr1/larrybird/nccoex/tools/xcelium/files/STD

   2    synopsys      synopsys      SYNOPSYS
/usr1/larrybird/nccoex/tools/xcelium/files/SYNOPSYS

   3    ieee          ieee          IEEE
/usr1/larrybird/nccoex/tools/xcelium/files/IEEE

   4    ambit         ambit         AMBIT
 /usr1/larrybird/nccoex/tools/xcelium/files/AMBIT

   5    vital_memory  vital_memory  VITAL_MEMORY
/usr1/larrybird/nccoex/tools/xcelium/files/VITAL_MEMORY
```

- In a `cds.lib` file, when the directory path specified in the `cds.lib` file does not exist or is inaccessible, the following warning message is generated. For example, you may have the following line in your `cds.lib` file, but you did not create a `./worklib` physical directory.

  ```
  DEFINE worklib ./worklib
  %xmvlog board.v
  xmvlog: *W,DLCPTH (./cds.lib,2): cds.lib Invalid path
  ```
  '/hm/belanger/xcelium/board/worklib' (`cds.lib` command ignored).
- If you have an `hdl.var` file but not the `cds.lib` file, the following messages are generated:

  ```
  %xmvlog board.v
  xmvlog: *W,DLNOCL: Unable to find a 'cds.lib' file to load in.
  xmvlog: *F,WRKBAD: logical library name WORK is bound to a bad library name
  'worklib'.
  ```

  The `DLNOCL` warning occurs when the tool could not find a `cds.lib` file using the search order specified in the `setup.loc` file.
  The `WRKBAD` error occurs when the work library is defined in the `hdl.var` file (for example, `DEFINE WORK worklib`), but the `cds.lib` file does not define the corresponding library (for example, `DEFINE worklib` *./worklib*).

The hdl.var File

The `hdl.var` file is an optional configuration text file that contains:

- Configuration variables, which determine how your design environment is configured. These include:
  - Variables that you can use to specify the work library where the compiler stores compiled objects and other derived data.
  - For Verilog variables (`LIB_MAP`, `VIEW_MAP`, `WORK`) that you can use to specify the libraries and views to search when the elaborator resolves instances.
- Variables that allow you to define compiler, elaborator, and simulator command-line options

and arguments.

- Variables that specify the locations of support files and invocation scripts.

If you do not have an `hdl.var` file, the simulator tools generate a warning message that an `hdl.var` file could not be found. For example, if you do not have an `hdl.var` file, and you define the work library by using the `-work` option on the command line, the parser generates the warning message and then compiles the source files.

⚠ Some variables that you can set in the `hdl.var`file, such as `XMVLOGOPTS`, `XMVHDLOPTS`, `XMELABOPTS`, and `XMSIMOPTS` can also be set as environment variables from the operating system. Variables that are set this way affect tool behavior, are overridden by variables set in the `hdl.var` file and, in some cases, can interact in undesirable ways with makefiles. No messages are generated telling you that a tool's behavior has been modified because a variable was set from the operating system.

All tools and utilities that use an `hdl.var` file use the same search mechanism to find the file. Each tool that reads an `hdl.var` file also has a `-hdlvar` option that you can use on the command line to specify an `hdl.var` file. This option overrides the default search mechanism.

You can have more than one `hdl.var` file. If you define the same variable in more than one file, the last variable read is used. For example, let us assume that you have the following *hdl.var* file in your current working directory. The `VERILOG_SUFFIX` variable defines recognized file extensions for Verilog source files.

```
INCLUDE ~/hdl.var
DEFINE VERILOG_SUFFIX (.ver)
DEFINE WORK ./worklib
```

The *hdl.var* file in your home directory is:

```
DEFINE VERILOG_SUFFIX (.vg)
```

The first line in the `hdl.var` file includes the `hdl.var` file in your home directory. This file sets the `VERILOG_SUFFIX` variable to `.vg`. The next line then sets the same variable to `.ver`. Only this suffix (`.ver`) is recognized as a valid suffix.

Now, suppose that the `hdl.var` file was written as follows:

```
DEFINE VERILOG_SUFFIX (.ver)
INCLUDE ~/hdl.var
DEFINE WORK ./worklib
```

In this case, the `VERILOG_SUFFIX` variable is first set to `.ver`, and then redefined to be `.vg`. Only the `.vg` suffix is recognized.

If you want both suffixes to be recognized, you could, for example, do the following:

```
# ./hdl.var
INCLUDE ~/hdl.var
DEFINE VERILOG_SUFFIX $VERILOG_SUFFIX (.ver)
DEFINE WORK worklib
# ~/hdl.var
DEFINE VERILOG_SUFFIX (.vg)
```

In this case, `VERILOG_SUFFIX` is first set to `.vg`. Then the `.ver` suffix is appended to this definition so that the compiler recognizes both suffixes.

Related Topics

- The hdl.var Statements and Rules
- The hdl.var Variables
- The setup.loc File
- The cds.lib File

The hdl.var Statements and Rules

The following statements are supported for use in the `hdl.var` file:

- `DEFINE` *`<variable> <value>`*
- `UNDEFINE` *`<variable>`*
- `INCLUDE` *`<filename>`*
- `SOFTINCLUDE` *`<filename>`*

Statement	Description

`DEFINE` *`<variable>`* *`<value>`*	Defines a *`<variable>`* and assigns a *`<value>`* to the variable.
	• A *`<variable>`* is an alphanumeric variable name.
	• The *`<value>`* is optional; if provided, it is either scalar or a list.
	For instance, the following defines the variable `WORK` to `worklib`.
	`DEFINE WORK worklib`
	The following defines the variable `VERILOG_SUFFIX` as the list `.v`, `.vg`, and `.vb`.
	`DEFINE VERILOG_SUFFIX (.v, .vg, .vb)`
	The following defines the variable `XMVLOGOPTS`, which specifies options for *xmvlog*.
	`DEFINE XMVLOGOPTS -messages -errormax 10 -update`

`UNDEFINE` *`<variable>`*	Removes the specified *`<variable>`* definition. This statement is useful for clearing definitions that were defined in other files; however, if the variable was not previously defined, the tool does not output an error message. For instance: `UNDEFINE XMUSE5X`

`INCLUDE <filename>`	Reads the specified `<filename>` as an `hdl.var` file.
	Use `INCLUDE` to include the variable definitions that are contained in the given file.
	• A `<filename>` specification can include an absolute or relative path. If it is relative, it is relative to the `hdl.var` file in which it is defined.
	For instance, the following specifies `my_hdl.var` as the `hdl.var` file using a relative path.
	`INCLUDE ~/my_hdl.var`
	The following specifies a custom `hdl.var` file using an absolute path to some user directory.
	`INCLUDE /users/${USER}/hdl.var`
	⚠ If the file is not found, a warning message is printed.

`SOFTINCLUDE <filename>`	`SOFTINCLUDE` is the same as the INCLUDE statement, except that no warning message is printed if the specified file is not found.
	For instance:
	`SOFTINCLUDE ~/hdl.var`
	`SOFTINCLUDE ${GOLDEN}/hdl.var`

Rules for Using the hdl.var file

The following rules apply to the `hdl.var` file:

- Only one statement per line is allowed.
- Keywords and variable names are case insensitive.
- Variable values, file names, and pathnames are case-sensitive.
- Begin comments with either the pound sign (#) or a double hyphen (--). The comment character must be either the first character in a line or preceded by white space.

- You can extend a statement over more than one line by using the escape character (\) as the last character of the line. For example: `DEFINE ALPHA (a,\ b,\ c)` is the same as `DEFINE ALPHA (a, b, c)`
- Left and right parentheses indicate the beginning and end of a list of values.
- Use a comma to separate values in a list.
- You can have a list containing zero elements. `DEFINE EMPTY_LIST ( )`
- Any character can be escaped using the backslash (\) escape character. Characters should be escaped if the meaning of the character is its ASCII value. For example, the following line defines the variable `JUNK` as `\dump\`. `DEFINE JUNK \\dump\\` The following example defines `LIST` as `a,b` and `c`. `DEFINE LIST (a\,b,c)`
- You can use tilde (~) in filename or value to specify:
    - ~ (or `$HOME`)
    - ~ `user` (home of `<user>`) The "~" must be the first non-whitespace character in filename or value. For example: `DEFINE DIR_RELATION ~/bin != ~larrybird/bin` expands to: `/usr/bin != /usr/larrybird/bin`
- The white space preceding and following a scalar value is ignored. In the following two lines, the variable `TEST` has the same value (this is a test): `DEFINE TEST` this is a test `DEFINE TEST` this is a test.
- You can use the dollar sign ($) in filename or value to indicate variable substitution. The syntax can be either `$ variable` or `${ variable }`, where the left and right braces ({ }) are real characters that mark the beginning and end of the variable name. Variable substitution first searches the `hdl.var` definitions and, if none are found, then searches for environment variables. The following example uses the environment variable `$SHELL` to define an *hdl.var* variable: `DEFINE MY_SHELL $SHELL` In the following example, `LIB_MAP` is defined as: (`./source/lib1/... => lib1`) Then the variable is redefined as (`./source/lib1/... => lib1, ./design => lib2`) `DEFINE LIB_MAP` (`./source/lib1/... => lib1`) `DEFINE LIB_MAP` (`$LIB_MAP, ./design => lib2`) In the following example, `ALPHA` is defined as first. Then `BETA` is defined as first == one. `DEFINE ALPHA first DEFINE BETA ${alpha} == one`
- When a scalar value or file name is specified as a relative path, the path is relative to the location of the `hdl.var` file in which it is defined.

Related Topics

- The hdl.var Variables
- The hdl.var File

The hdl.var Variables

The following variables are supported for use in an `hdl.var` file. The variable names can be entered in lowercase or in uppercase.

- FILE_OPT_MAP
- HALOPTS
- LIB_MAP
- SRC_ROOT
- VERILOG_SUFFIX
- VHDL_SUFFIX
- VIEW
- VIEW_MAP
- WORK
- XM_DBTIMEOUT
- XMELABOPTS
- XMHELP_DIR
- XMLOCK_INFO
- XMPROTECTOPTS
- XMSDFCOPTS
- XMSHELLOPTS
- XMSIMOPTS
- XMSIMRC
- XMUPDATEOPTS
- XMUSE5X

- XMVERILOGOPTS
- XMVHDLOPTS
- XMVLOGOPTS
- XMVLOGPRESOURCE
- XRUNOPTS

⚠ The variable definitions in the `hdl.var` file are treated as literal strings. Do not use quotation marks in the definitions unless you explicitly want them as part of the input. For example, use:

```
DEFINE XMVLOGOPTS -define foo=16'h03
```

instead of

```
DEFINE XMVLOGOPTS -define foo="16'h03"
```

which is the same as:

```
% xmvlog -define foo=\"16'h03\"
```

Related Topic

- Verilog Module and UDP Resolution

FILE_OPT_MAP

This variable maps Verilog or VHDL source files, or the directories that contain them, to command-line options. It enables you to compile different source files with different options.

⚠ You can define only one `FILE_OPT_MAP` variable in an `hdl.var` file.

Syntax

```
DEFINE FILE_OPT_MAP ( {filename|directory} => list_of_options [, {filename|directory}
=> list_of_options...)
```

- Specify either a *`filename`* or a *`directory`*, followed by a list of options separated by a space.
  - If a filename is specified, all options are applied to that file.
  - If a directory is specified, all options are applied to all files in that directory.

- The backslash character (`\`) can be used to define the variable over multiple lines.

Examples

The following maps the VHDL source file `file1.vhd` to the command-line options `-v93`, `-relax`, and `-linedebug`.

```
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -relax -linedebug)
```

The following is the same as the first example but also maps the Verilog source file `file1.v` to the command-line options `-view`, and `-v1995`.

DEFINE FILE_OPT_MAP (file1.vhd => -v93 -relax -linedebug, file1.v => -view view1 -v1995)

The following is the same as the second example but uses the backslash character to define `FILE_OPT_MAP` over multiple line.

```
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -relax -linedebug, \
            file1.v => -view view1 -v1995)
```

The following illustrates a more complex `FILE_OPT_MAP` definition taken from an `hdl.var` file:

```
# File: hdl.var
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug, \
                     ./src/vhdl/file2.vhd => -controlrelax ARSHCH -v93, \
                     ./src/vhdl => -v93, \
                     ./src/vlog/file1.v => -view view1 -v1995)
```

In this example:

- `file1.vhd` is compiled with the `-v93`, `-controlrelax`, and `-linedebug` options.
- `./src/vhdl/file2.vhd` is compiled with the `-controlrelax` and `-v93` options.
- All other files in the directory `./src/vhdl` are compiled with the `-v93` option.
- `./src/vlog/file1.v` is compiled with the `-view` and `-v1995` options.

HALOPTS

Defines HAL (Xcelium HDL Analysis) command-line options.

Syntax

```
DEFINE HALOPTS <list_of_options>
```

Example

The following example illustrates how to specify all checks globally in the `hdl.var` file:

```
DEFINE HALOPTS -check All
```

LIB_MAP

For the compiler, this variable maps files and directories to library names. Use the plus sign (`+`) to specify other files or directories that are not explicitly specified.

For the elaborator, this variable establishes the list of libraries to search and the order in which to search them when resolving Verilog instances.

Syntax

```
DEFINE LIB_MAP ( <directory>     => <library>, \
                 <directory>/... => <library>, \
                 <file>          => <library>, \
                 +               => <library> )
```

Example

```
DEFINE LIB_MAP ( ./design          => designlib, \
                 ./source/lib1/... => lib1, \
                 myfile.v          => mylib, \
                 +                 => worklib )
```

- All files in the directory `./design` are to be compiled into `designlib`.
- All files in `./source/lib1` and the directories below it are to be compiled into lib1.
- The file `myfile.v` is to be compiled into `mylib`.
- Any other file is to be compiled into `worklib`.

⚠ Using the `LIB_MAP` variable, you cannot map the same name to multiple libraries. For example, if you have Verilog and VHDL source files in the same directory, you cannot use `+` to map the Verilog files to one default library and map the VHDL files to a different default library. The default library would be the same for both Verilog and VHDL.

SRC_ROOT

This variable defines an ordered list of paths (as comma-separated values) to search for source files when you are updating a design either by running `xmsim -update` or by rerunning `xrun`. Using this variable, a legal path can be defined as:

- An absolute path statement
- A relative path statement
- An environment variable

The paths listed in the definition tell Xcelium where to look for source files if any design units are out-of-date.

Syntax

```
DEFINE SRC_ROOT (<list_of_paths>)
```

Example

```
DEFINE SRC_ROOT (~larrybird/source, $PROJECT)
```

VERILOG_SUFFIX

This variable defines valid file extensions for Verilog source files using a list of comma-separated values.

Syntax

```
DEFINE VERILOG_SUFFIX (<file_extension>, ...)
```

Example

```
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)
```

VHDL_SUFFIX

This variable defines one or more valid file extensions for VHDL source files. If multiple file extensions are defined, you must use a list of comma-separated values and enclose the argument in parentheses. The default value is:

```
DEFINE VHDL_SUFFIX (.vhd, .vhdl)
```

Syntax

```
DEFINE VHDL_SUFFIX <file_extension> | (<file_extension>, ...)
```

Examples

The following example defines a single valid file extension `.vhd`:

```
DEFINE VHDL_SUFFIX .vhd
```

The following example defines five valid file extensions `.vhd`, `.vhdl`, `.vsyn`, `.vrtl`, and `.vgate`:

```
DEFINE VHDL_SUFFIX (.vhd, .vhdl, .vsyn, .vrtl, .vgate)
```

VIEW

This variable sets the view *`name`*. A view is part of the Xcelium library structure organized according to the *`Library`*`.`*`Cell`*`:`*`View`* approach.

Using different views can be helpful when you want to delineate between different representations, abstraction levels, or status of a particular cell.

Syntax

```
DEFINE VIEW <name>
```

Example

```
DEFINE VIEW behavior
```

Related Topic

- Xcelium Library and Directory Structure

VIEW_MAP

For the *xmvlog* compiler, this variable maps files and file extensions to view names. The variable definition specifies that files or files with a particular extension are compiled with the given view name.

For the elaborator, this variable establishes a list of views to search when resolving instances.

Syntax

```
DEFINE VIEW_MAP ( <file_extension> => <view_name>, ... )
```

Example

```
DEFINE VIEW_MAP ( .v        => behav, \
                  .rtl      => rtl, \
                  .gate     => gate, \
                  myfile.v => gate )
```

WORK

This variable defines the current work library into which HDL design units are compiled.

You can also specify the work library with the `-work` command-line option. The command-line option overrides the definition in the `hdl.var` file.

Syntax

```
DEFINE WORK <library_name>
```

Example

DEFINE WORK worklib

XM_DBTIMEOUT

This variable sets a custom time, in seconds, that the tool must wait to check on a lock for the specified library.

By default, the tool waits for up to an hour to get the lock. This lock is required because another

By default, the tool waits for up to an hour to get the lock. This lock is required because another process may be accessing the library at the same time.

You can set this as either an `hdl.var` variable or as an environment variable.

Syntax

```
DEFINE XM_DBTIMEOUT <number>
```

Or:

```
setenv XM_DBTIMEOUT <number>
```

Example

The following shows how to set a timeout period of two hours using both supported methods.

hdl.var variable

```
DEFINE XM_DBTIMEOUT 7200
```

environment variable

```
% setenv XM_DBTIMEOUT 7200
```

XMELABOPTS

This variable sets a list of command-line options for the elaborator. The top-level design unit name(s) can also be included.

The command-line options that you specify with this variable are appended to the `xmelab` command. Hence, if an option specified in the `hdl.var` file is the same as one included on the command line, the option specified in the `hdl.var` file overrides the command-line specification. This happens because the last iteration of a given option on the command line takes precedence.

Also, be aware that Xcelium does not allow the following options in the variable definition:

- `-logfile`
- `-append_log`
- `-cdslib`
- `-hdlvar`

Including one or more of these options in the specified list results in one or more associated warnings.

Syntax

```
DEFINE XMELABOPTS <list_of_options>
```

Example

```
DEFINE XMELABOPTS -messages -errormax 10
```

XMHELP_DIR

This variable specifies the path to the directory where the help text files are located. These files are used by `xmhelp` to print more detailed information on the messages printed by other tools.

The help files are usually located in: `<install_directory>/tools/xcelium/files/help`

Use the `XMHELP_DIR` variable to use help files that are located in another directory.

Syntax

```
DEFINE XMHELP_DIR <path_to_help_files>
```

Example

```
DEFINE XMHELP_DIR <other_install>/tools/xcelium/files/help
```

XMLOCK_INFO

This variable specifies that XM tools can create files containing lock information for shared libraries when reading or writing to the library.

Design and verification engineers often work in a shared library environment using the simulator and other XM tools installed in a central location. In such an environment, the XM tools use a locking mechanism to prevent conflicts when multiple users attempt to read or write to the shared library. However, one can encounter a situation in which a process is waiting to execute because another process run by another user holds a shared or exclusive lock on a library. This results in warning messages like the ones below:

```
xmsim: *W,DLWTLK: Waiting for an exclusive lock on library LIB.
```

or:

```
xmsim: *W,DLWTLK: Waiting for a shared lock on library LIB.
```

To display information that tells you which user and process is holding the lock on a library so that you can determine if the lock is valid or if the process is hung, add `XMLOCK_INFO` to the `hdl.var` file.

Syntax

```
DEFINE XMLOCK_INFO
```

⚠ Setting this variable affects performance. The severity of the impact depends on the design and on the environment.

Example

Consider the following `hdl.var` specification when reviewing this section:

```
DEFINE XMLOCK_INFO
```

If all users must have this variable set, the variable first should be defined in a project-level `hdl.var` file and then should be included in the client-level `hdl.var` file(s) with an `INCLUDE` statement. Once this variable is set, the XM tools create a directory called `.lock_info` within the library that is being accessed. A file containing the library lock information is then created in this directory. Invoke the *xmls* utility with the `-lockinfo` option to display the lock information.

```
% xmls -lockinfo <library_name> [<library_name>...]
```

- If the `XMLOCK_INFO` variable has not been set, lock information files are not created. `xmls -lockinfo` generates a `NOINFO` warning and displays only the last process that has put a lock on the library.
- If the `XMLOCK_INFO` variable has been set, but a library is read-only, lock information files for that library cannot be created. A warning message `DLWRLI` is generated telling you that the information file could not be written. In this case, `xmls -lockinfo` generates a warning `NOINFO` and displays only the last process that has put a lock on the library.
- Lock information files are deleted when the lock on the library is released, or when exiting from an XM tool. The files are also deleted if the library is unlocked with the `xmpack -unlock` command.

  ```
  % xmpack -unlock <library_name>
  ```

XMPROTECTOPTS

This variable sets a list of command-line options for *xmprotect*.

Syntax

```
DEFINE XMPROTECTOPTS <list_of_options>
```

Example

```
DEFINE XMPROTECTOPTS -messages -nocopyright
```

XMSDFCOPTS

This variable sets a list of command-line options for `xmsdfc`.

Syntax

```
DEFINE XMSDFCOPTS <list_of_options>
```

Example

```
DEFINE XMSDFCOPTS -messages
```

XMSHELLOPTS

This variable sets a list of command-line options for `xmshell`.

Syntax

```
DEFINE XMSHELLOPTS <list_of_options>
```

Example

```
DEFINE XMSHELLOPTS -messages -errormax 10
```

XMSIMOPTS

This variable defines a list of command-line options for `xmsim`. A snapshot name can also be included.

The command-line options that you specify with this variable are appended to the `xmsim` command. Hence, if an option specified in the `hdl.var` file is the same as one included on the command line, the option specified in the `hdl.var` file overrides the command-line specification. This happens because the last iteration of a given option on the command line takes precedence.

Also, be aware that Xcelium does not allow the following options in the variable definition:

- `-logfile`
- `-append_log`
- `-cdslib`
- `-hdlvar`

Including one or more of these options in the specified list results in one or more associated warnings.

Syntax

```
DEFINE XMSIMOPTS <list_of_options>
```

Example

Consider the following `hdl.var` specification when reviewing this section:

```
DEFINE XMSIMOPTS -messages -errormax 10
```

The options defined using this variable are appended to the `xmsim` command. For instance, if you have defined the above `XMSIMOPTS` variable, then you enter the following command:

```
% xmsim -update worklib.top
```

The actual command line is as shown:

```
% xmsim -update worklib.top -messages -errormax 10
```

If an option is specified in the `hdl.var` file and is also included on the command line, the option specified in the `hdl.var` file overrides the option entered directly on the command line because the last option on the command line is used.

For instance, consider the following `xmsim` command that specifies an `-errormax` value of `5`.

```
% xmsim -errormax 5 worklib.top
```

Using the same defined `XMSIMOPTS` variable above form the `hdl.var` file, `-errormax 10` is appended to this command line. It overrides the `-errormax 5` option.

```
% xmsim -errormax 5 worklib.top -messages -errormax 10
```

XMSIMRC

This variable executes a command file when *xmsim* is invoked. This command file can contain commands, such as aliases, that you use with every simulation run.

Syntax

```
DEFINE XMSIMRC <path_to_file>
```

Example

```
DEFINE XMSIMRC /usr/design/rcfile
```

Or:

```
DEFINE XMSIMRC <install_dir>/tools/xcelium/files/xmsimrc
```

XMUPDATEOPTS

This variable sets a list of command-line options for *xmupdate*.

Syntax

```
DEFINE XMUPDATEOPTS <list_of_options>
```

Example

The following causes the compiler to use only the specified library, `my_lib`, when updating the model.

```
DEFINE XMUPDATEOPTS -library my_lib
```

XMUSE5X

This variable enables Xcelium to generate a packed library database file using the full Cadence 5.x library structure, in which the intermediate objects for each design unit are stored in their own subdirectories under the work library. It also creates three other files for each design unit: `master.tag`, `verilog.v`, and `pc.db`.

The full 5.x library structure and the additional files make it possible for tools that do not understand the default packed library structure to access the intermediate objects that are required by the simulator.

If you want to use a 5.x configuration file to control the binding of instances during elaboration, you must:

- Compile the source files with this variable defined.
- Or, specify the `-use5x` option on the `xmvlog` or `xmvhdl` command line.

Syntax

```
DEFINE XMUSE5X
```

XMVERILOGOPTS

ⓘ For simulations that still use the legacy `xmverilog` command, you can continue to set this variable for backward compatibility. However, be aware that Xcelium has replaced the `xmverilog` executable with `xrun`. Hence, invoking a simulation with the `xmverilog` command automatically invokes `xrun`.

This variable sets a list of command-line options for *xmverilog* and for *xmprep*. Both *xmverilog* and *xmprep* create the `hdl.var` file automatically if one does not exist. If you want to set frequently-used options, or if you want to set defaults for all users, you can create a custom `hdl.var` file before running these tools, and the tools will read from this `hdl.var` file instead.

Syntax

```
DEFINE XMVERILOGOPTS <list_of_options>
```

Example

```
DEFINE XMVERILOGOPTS +overwrite
```

XMVHDLOPTS

This variable sets a list of command-line options for the *xmvhdl* compiler. VHDL source file names can also be included.

However, be aware that Xcelium does not allow the following options in the variable definition:

- `-logfile`
- `-append_log`
- `-cdslib`
- `-hdlvar`

Including one or more of these options in the specified list results in one or more associated warnings.

Syntax

```
DEFINE XMVHDLOPTS <list_of_options>
```

Example

```
DEFINE XMVHDLOPTS -messages -errormax 10 -file ./proj_file
```

XMVLOGOPTS

This variable sets a list of command-line options for the *xmvlog* compiler. Verilog source file names can also be included.

However, be aware that Xcelium does not allow the following options in the variable definition:

- `-logfile`
- `-append_log`
- `-cdslib`
- `-hdlvar`

Including one or more of these options in the specified list results in one or more associated warnings.

Syntax

```
DEFINE XMVLOGOPTS <list_of_options>
```

Example

```
DEFINE XMVLOGOPTS -messages -errormax 10 -file ./proj_file
```

XMVLOGPRESOURCE

This variable defines a single source file or a macro file for the *xmvlog* compiler to load before any other source files specified on the command line.

Syntax

```
DEFINE XMVLOGPRESOURCE <filename>
```

Example

```
DEFINE XMVLOGPRESOURCE mydefine.v
```

XRUNOPTS

This variable specifies `xrun` command-line options.

Use this variable to define common options, project-wide options, and/or custom file extensions in the `hdl.var` file.

When specifying command-line options with this variable, be aware that only the `-hdlvar` option is not allowed. If you add this option to an `hdl.var` file using this variable and attempt to simulate a design using *xrun*, then the software outputs a warning.

Syntax

```
DEFINE XRUNOPTS <list_of_options>
```

Examples

Example 1:

```
DEFINE XRUNOPTS -ieee1364 -notimingchecks -access +rw
```

This defines options to enable read/write access, disable timing checks, and report on IEEE 1364 standard errors.

Example 2:

```
DEFINE XRUNOPTS -vlog_ext .v,.vg,.rtl -ieee1364 -notimingchecks -access +rw
```

This is similar to the first example but also defines custom Verilog file extensions by adding the `-vlog_ext` option to the list. This option is required when using *xrun* to specify custom file types for the Verilog parser.

Example 3:

```
INCLUDE <path_to_project_hdlvar>
DEFINE XRUNOPTS -ieee1364 -notimingchecks -access +rw
```

This example illustrates how to define project-wide options by using the INCLUDE statement for linking the `hdl.var` file in the work directory to some project-level `hdl.var` file.

Debugging hdl.var Files

Use the `xmhelp -hdlvar` command to display information about the contents of *hdl.var* files. This helps identify incorrect settings that may be contained within your *hdl.var* files.

Syntax:

```
% xmhelp -hdlvar [ hdl.var_file ]
```

Examples:

```
% xmhelp -hdlvar
% xmhelp -hdlvar ~/hdl.var
```

The following example shows how to display information about the contents of an *hdl.var* file. In the example, the `xmhelp -hdlvar` command displays the contents of the first *hdl.var* file found using the search order in the *setup.loc* file. In this example, the *hdl.var* file is in the current working directory.

```
%xmhelp -nocopyright -hdlvar

Parsing -HDLVAR file ./hdl.var.

hdl.var files:

1:  ./hdl.var
```

```
Variables defined:

Defined in ./hdl.var:

Line #  Name            Value

------  ----            -----

   5    LIB_MAP         ( /net/foghorn/usr1/belanger/chip1 => chip1 , \

                          /net/foghorn/usr1/belanger/libs/misc.v => misc )

   1    XMVLOGOPTS      -messages

   2    XMELABOPTS      -messages

   3    XMSIMOPTS       -messages

   4    VERILOG_SUFFIX  ( .v , .vlog )

   6    VIEW_MAP        ( .g => gates , .b => behav , .rtl => rtl )

   7    WORK            worklib
```

The setup.loc File

When using the `cds.lib` and `hdl.var` files to manage your simulation data and control the operation of the various Xcelium tools and utilities, you can also choose to write a `setup.loc` file, which specifies a custom location to search for these associated files. The tools and utilities that need to read library definition files (`cds.lib`) and configuration files (`hdl.var`) locate these files by:

- Searching the following directories in order for a `setup.loc` file:
    - The current work directory
    - `$CDS_WORKAREA` (user work area, if defined)
    - `$CDS_SEARCHDIR` (if defined)
    - `$HOME`
    - `$CDS_PROJECT` (project area, if defined)
    - `$CDS_SITE` (site-specific location, if defined)
    - `<cds_`*`install_directory`*`>/share`
- Using the search list above to find other setup files, if a `setup.loc` file *is* found.
- Searching the following directories in order for the setup files, if a `setup.loc` file *is not* found:

  - The current work directory
  - `$CDS_WORKAREA` (user work area, if defined)
  - `$CDS_SEARCHDIR` (if defined)
  - `$HOME`
  - `$CDS_PROJECT` (project area, if defined)
  - `$CDS_SITE` (site-specific location, if defined)
  - `<`*`cds_install_directory`*`>/share`
- Creating a directory called `xcelium.d` and then creating a default work library called `worklib` in that directory, if no setup files are found.

Related Topics

- Writing a setup.loc File
- The cds.lib File
- The hdl.var File

Writing a setup.loc File

To write a `setup.loc` file, copy the example in `<`*`cds_install_directory`*`>/share/cdssetup/setup.loc` and edit the file to list the directories in the order that you want them to be searched. You must put the `setup.loc` file in one of the following locations so that the Xcelium tools and utilities can find it:

- The current work directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- $HOME
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site-specific location, if defined)
- <cds_*install_directory*>/share

Rules for Writing a setup.loc File

The following rules apply when writing the `setup.loc` file:

- Only one entry per line is allowed.
- Use a semicolon (`;`), the pound sign (`#`), or a double hyphen (`--`) to begin a comment.
- The file can include:
    - `~`
    - `~user`
    - `$ environment_variable`
    - `$ {environment_variable}`

  By convention, environment variables are given uppercase names. See the documentation for your implementation of UNIX for complete details on setting environment variables.

  ⚠ If a directory specified in the `setup.loc` file references an environment variable that is not set, the location referenced by the variable is not searched and no warning or error message is issued.

- Relative paths in `setup.loc` files are relative to the current directory; they are not relative to the location of the file in which they occur or to the directory where the tool was invoked.

  ⚠ If a directory specified in the `setup.loc` file cannot be found or is not accessible, the search advances to succeeding locations without printing warning or error messages.

Related Topic

- The setup.loc File

Setting Universal Verification Methodology Environment Variables

The UVM installations provided by Xcelium are located as follows:

- `` `xmroot`/tools/methodology/UVM/CDNS-1.1d `` – path to UVM-SV 1.1d
- `` `xmroot`/tools/methodology/UVM/CDNS-1.2 `` – path to UVM-SV 1.2
- `` `xmroot`/tools/methodology/UVM/CDNS-1.2-ML `` – path to UVM Multi-Language
- `` `xmroot`/tools/methodology/UVM/CDNS-IEEE `` – path to UVM-IEEE

Depending on the UVM library you want to use, set the xrun/xmsim `-uvmhome` option as follows:

- `-uvmhome CDNS-1.1d`
- `-uvmhome CDNS-1.2`
- `-uvmhome CDNS-1.2-ML`
- `-uvmhome CDNS-IEEE`

⚠

- You do not need to set any environment variables to configure UVM.
- The default UVM installation is defined in the `` `xmroot`/tools/methodology/config `` file.
- If you are using UVM-ML, see additional configuration information in Compiling and Running Multi-Language Environments in the *UVM-ML Open Architecture User Guide*.

Troubleshooting the Xcelium Simulator Setup

The following sections describe how to recover from problems that may occur during your configuration of the simulator.

Libraries and Snapshots

Description: 32-bit and 64-bit libraries and snapshots are not compatible. For example, to simulate with 64 bits, the design must be compiled and elaborated with the 64-bit compiler and elaborator. If you build with the 32-bit versions and then try to simulate with 64 bits, the simulator generates an error telling you that the snapshot could not be found.

32-bit and 64-bit libraries and snapshots can coexist in the same physical directory. 64-bit libraries have the number 64 appended to their name. For example, a work library might contain the following two library files:

```
xlm.lnx86.066.pak
xlm.lnx8664.066.pak
```

Cannot Run the GUI Remotely on Another Host

Description: If you get an error message that says:

```
xmsim: can't open display, exiting . . .
```

Then you do not have permission to write to the remote host's display.

Add the following line to your `.xinitrc` file:

```
+host
```

In order to display output on a remote host, you may need to set the `DISPLAY` environment variable with the following line:

```
setenv DISPLAY <remote_host>:0
```

Cannot Run SimVision

Description: If you get the following error message:

```
X toolkit Error: Can't Open display (UNIX)
```

Then add this line to your `.xinitrc` file:

```
xhost +
```

SimVision Colors Not Optimal

Description: SimVision requires 24-bit color depth for optimum viewing and mapping of colors. See your system administrator if you think you need to increase your color depth.

Cannot Interact With the Simulator	
Description:	When you use the interrupt key to terminate a simulation started by a C shell script, the script terminates, but the simulation continues, and you cannot interact with the simulator. Let the simulation run to completion or obtain its process number, and include the number in a `kill` command. For example, to display the process number, use the `ps` command, then issue the following command: `kill` *`process_number`*

⚠ Running the scripts through the Bourne shell ensures that the interrupt key has the normal effect on scripts that initiate simulations.

3

Xcelium Help Information

The Xcelium Simulator supports multiple ways to access help for its tools, commands, and error messages at the command line. You can choose to:

- Get help on simulator executable commands
- Get help on simulator Tcl commands
- Get help on mnemonics output by the simulator

Additionally, the simulator returns exit codes for various error conditions. These codes help you to identify the error scenario, the category, and debug the reported issue.

Related Topics

- Getting Help on Tool Commands
- Getting Help on Simulator Tcl Commands
- Getting Help on Tool Messages
- Return Codes for Error Conditions

Getting Help on Simulator Tcl Commands

To get help on simulator (*xmsim*) Tcl commands:

- Use the `help` command.

  ```
  xcelium> help [ help_options ] [ command | all [ command_options ]]
  ```

  Examples

  ```
  xcelium> help database
  xcelium> help database -statement
  ```

  In addition to the set of interactive commands implemented for the simulator, the help command also displays help on standard Tcl commands. Only basic information is provided for these commands. See the following websites for information on Tcl commands and for

other information related to Tcl/Tk:
    - http://www.tcl.tk/
    - http://www.elf.org
    - http://dev.scriptics.com/
- Use the command reference manual in the online help.
- If you are using the SimVision analysis environment, some commands display pop-up forms that you have to fill in. Click the *Help* button on the form to get more information about that option.

Related Topics

- Getting Help on Tool Commands
- Getting Help on Tool Messages
- Return Codes for Error Conditions
- Xcelium Simulator Tcl Command Reference

Getting Help on Tool Commands

To display a simple list of options for any of the simulator tools and utilities, specify the tool or utility name followed by the `-help` option. The `-help` option displays a list of the command options for the specified tool with a brief description of each option.

Syntax

```
% tool_name -help
```

Examples

- `% xrun -help`
- `% xmvlog -help`
- `% xmvhdl -help`
- `% xmelab -help`
- `% xmsim -help`
- `% xmupdate -help`

xrun also includes an *extensive help system* with many options that let you display a list of all valid command-line options, recognized file types and their default file extensions, all options related to a particular subject or executable, aliases for options, the minimum characters that must be entered for an option, and so on.

Getting Enhanced Command Help Using XRUN

The *xrun* utility is a single executable that lets you use one command to invoke various tools to compile different types of files, elaborate the design, and simulate a snapshot. Because *xrun* can invoke several tools, each of which has its own set of command-line options, the number of options that you can use with one command is large. This utility also includes several command options that let you change the display of the output in various ways.

The following table describes options to display the output in various ways.

Option	Description
`-helpalias`	Displays the different ways to enter an option.
`-helpall`	Displays a list of every supported *xrun* option.
`-helpargs`	Displays a list of source files specified on the command line, with their type, and the list of the command-line options being used.
`-helpfileext`	Displays all supported source file types, the default file extensions defined for each type, and the command-line option you can use to change the set of defined file extensions.
`-helphelp`	Displays all the *xrun* options that control help.
`-helpxmverliog`	Displays all the *xmverilog*-related options.
`-helpshowmin`	Displays the minimum characters required for each supported dash option.
`-helpsubject <`*`subject`*`>`	Displays a list of options for the specified subject.

`-helpusage`	Displays a list of every supported option, with each option followed by two fields: • **`flags`**: The flags displayed in this field can be: ◦ `ARG`: This option requires an argument. ◦ `HARD`: In a subsequent *xrun* invocation, adding or removing this option, or changing the argument to this option, causes some (or all) executables to be rerun. ◦ `NOMULT`: This option can be specified once on the command line. ◦ `PSC`: This option can have a list of arguments separated by a comma. ◦ `PSP`: This option can have a list of arguments separated by a `+`. ◦ `PSE`: This option can have a list of arguments separated by a `=`. ◦ `PSFS`: This option can have a list of arguments separated by a `/`. ◦ `PSSP`: This option can have a list of argument enclosed in quotes. For example: `"a b c"` ◦ `SCMUL`: Thos option can have a list of arguments. • **`execs`**: This field lists the executables to which the option applies.
`-helpverbose`	Displays the options with verbose text descriptions.
`-helpwidth <`*`width`*`>`	Sets the maximum width for command help.

Related Topics

- Checking for Compatible Legacy Options Using XRUN
- Getting Help on Simulator Tcl Commands
- Getting Help on Tool Messages
- Return Codes for Error Conditions

Checking for Compatible Legacy Options Using XRUN

When simulating your design with the *xrun* utility, you might want to verify which of the supported legacy options still work in the *xrun* flow. To do this, specify the legacy options you want to use and add the `-checkargs` option on the command line:

```
% xrun [legacy_options] -checkargs
```

For example:

```
% xrun -nocopyright -ieee1364 -v93 -access +r \
xor_verify.e xor.v xor_specman.vhd -top xor_top -checkargs

Checking arguments. Following are the command line arguments recognized by xrun:
   Minus ("-") options:
        -nocopyright -ieee1364 -v93 -checkargs
   Paired minus ("-") options:
        -access +r
        -top xor_top

   Source file arguments:
        xor_verify.e
        xor.v
        xor_specman.vh
```

Related Topic

- Getting Help on Tool Commands

Getting Help on Tool Messages

To display extended help on the brief messages generated by the compiler, elaborator, and simulator, use the *xmhelp* utility.

Syntax

```
% xmhelp [ options ] tool_name message_code
```

You can enter the `message_code` argument in lowercase or in uppercase.

Examples

- `% xmhelp xmvlog BADCLP`
- `% xmhelp xmvlog badclp`
- `% xmhelp xmelab cuvwsp`
- `% xmhelp xmsim`

Related Topics

- Getting Help on Tool Commands
- Getting Help on Simulator Tcl Commands
- Return Codes for Error Conditions

Return Codes for Error Conditions

The XM binaries (*xmvlog*, *xmvhdl*, *xmelab*, *xmsim*) and the *xmshell*, *xmls*, *xmupdate*, and *xmrelocate* utilities return detailed exit codes by default.

The following table shows the exit code returned for the various error conditions.

Exit Code	Condition
0	Success or *W (warning) condition
1	*E (error) condition
2	*F (fatal) condition
3	*W (warning) with -warnstatus option
-1	*internal* error condition
-2	SIGILL signal condition
-3	SIGABRT signal condition
-4	SIGFPE signal condition

-5	SIGBUS signal condition
-6	SIGSEGV signal condition
-7	SIGXCPU signal condition
-8	SIGXFSZ signal condition
-9	SIGHUP signal condition
-10	SIGQUIT signal condition
-11	SIGINT signal condition
-12	SIGTERM signal condition
-16	exec failed
-17	exec failed and core file was dumped
-18	exec failed and there was a system error

The return codes are stored in the `$status` variable. You can access this variable with an `echo $status` (or `echo $?`) command. For example:

```
% xmvlog -nocopyright -bogus_option source.v

xmvlog: *F,BADOPT: unknown or ambiguous options (-bogus_option).

% echo $status

2
```

Accessing the return value of an executable can be useful in a script. For example, the following code checks the return value of *xmelab* to decide if *xmsim* should be executed:

```
xmelab ....
    if ($status == 0) then
        execute xmsim ....
    else
        exit;
    end if;
```

4

Xcelium Design Simulation

The Xcelium Simulator supports two methods of invocation. No environment setup files are required to run the simulator.

- *XRUN invocation* is the default method that you can use to run the simulator by specifying all input files and command-line options using one or more *xrun* command lines. This utility simplifies the invocation process by letting you use just one tool for design simulation instead of executing multiple tools separately to piece together a snapshot that can then be simulated manually with *xmsim*.

  The most basic way to use *xrun* is to list the files that comprise the simulation on a single command line, along with all command-line options that *xrun* will pass to the appropriate compiler, elaborator, and simulator. For instance:

  ```
  % xrun [options] <hdl_source_file1> <hdl_source_file2> ...
  ```

  In addition to single-step mode, the simulator also supports two-step and three-step *xrun* invocation.

- Direct invocation is the traditional method that uses separate tools to compile, elaborate, and simulate a design. In this case, the model must first be compiled and the design hierarchy defining the model must then be elaborated before you can run the simulation.

  For instance, consider a mixed Verilog/VHDL design where the Verilog source files are compiled with *xmvlog* and the VHDL source files are compiled with *xmvhdl*. The top-level design unit is specified as an argument to *xmelab* to elaborate the design and generate the snapshot, Finally, *xmsim* is invoked with the name of the snapshot to simulate the design.

  ```
  % xmvhdl [options] <vhdl_source_files>
  % xmvlog [options] <verilog_source_files>
  % xmelab [options] <top_level_design_unit>
  % xmsim [options] <snapshot_name>
  ```

Related Topics

- XRUN Invocation
- Direct Invocation

XRUN Invocation

The Xcelium Simulator supports specifying all input files and command-line options using the *xrun* utility. With its built-in flexible use model, you can use *xrun* to enable single-core and/or multi-core engine features from one or more command lines by running the tool in single-step or multi-step mode. The first time you run the simulator with the `xrun` command, it:

- Creates a scratch directory called `xcelium.d`.
- Creates a scratch subdirectory under the `xcelium.d` directory called

  `run.<`*`platform|platform.`*`64>.<`*`xrun_version`*`>.d`

  For example:

  `run.lnx86.17.04.d/`

  The *xrun* utility creates files and directories under this subdirectory to support tool operations. As a convenience, a symbolic link named `run.d` is created that points to this *xrun* scratch subdirectory.
- Parses the command line.
- Invokes the appropriate compiler for each file specified on the command line.
  - Design units contained in HDL design files are compiled into the work library `xcelium.d/worklib`.
  - Verilog design units specified in `-y` libraries or `-v` library files are compiled into libraries that have the same names. These libraries are stored in subdirectories of `run.d/xllibs`.
  - The output from the Specman ***e*** compiler, `sn_compile.sh`, is stored in subdirectories under the `run.d` directory.

  For example, the following command compiles `top.v` into the work library (`xcelium.d/worklib`). Design units in `./libs` are compiled into a library called `libs` (`xcelium.d/run.d/xllibs/libs`), and design units in `./models` are compiled into a library called `models` (`xcelium.d/run.d/xllibs/models`).

  ```
  % xrun top.v -y ./libs -y ./models +libext+.v
  ```

> ⚠ When using the `-y` option, the module name requested in the Verilog code instance must be identical to the file name found in the source directory. The file extension listed using `-libext` must also be the same. If *`<module_name>`*`.`*`<ext>`* is not identical, then the tool will be unable to read the file that contains the definition.

- Invokes the elaborator (*xmelab*) to elaborate the design and generate a simulation snapshot.
- Invokes the simulator (*xmsim*) to simulate the snapshot.

The output of all tools is written to a common log file called `xrun.log` in the directory in which *xrun* was invoked. You can change the name of the log file with the `-l` option. For example:

```
% xrun -l run1.log ...
```

For redirecting the output of certain executables, such as *hal* or *xmsim*, from the `xrun.log` file, see the -log_* command-line options.

Saving Path Information From XRUN Binaries

When using *xrun*, the real paths of all the Xcelium binaries used during simulation (*xrun*, *xmvlog*, *xmvhdl*, *xmelab*, and *xmsim*) are saved to a file named `xcelium_bin_paths.log` by default in the `xcelium.d` directory. For instance, consider a simple Verilog design (`top.v`) that is run using the `xrun` command below:

```
% xrun top.v
```

The tool creates the file `xcelium_bin_paths.log`, in the `xcelium.d` directory, and saves the path information from all binaries.

```
<xcelium_install_dir>/tools.lnx86/inca/bin/xrun
<xcelium_install_dir>/tools.lnx86/inca/bin/xmvlog
<xcelium_install_dir>/tools.lnx86/inca/bin/xmelab
<xcelium_install_dir>/tools.lnx86/inca/bin/xmsim
```

Where *`<xcelium_isntall_dir>`* is the absolute path to the Xcelium installation in your environment. You can choose to change the way the simulator saves the path information in the `xcelium_bin_paths.log` file by specifying the `-log_install_path` option on the `xrun` command line. If you specify this option with the `dynamic` argument, the tool instead saves the symlink paths of each binary used for simulation.

Now, consider a VHDL design (`top.vhd`) that is run using the following `xrun` command:

```
% xrun top.vhd
```

In this case, the tool still creates the `xcelium_bin_paths.log` file by default in the `xcelium.d` directory, but saves the absolute path information for *xmvhdl* instead of *xmvlog*.

```
<xcelium_install_dir>/tools.lnx86/inca/bin/xrun
<xcelium_install_dir>/tools.lnx86/inca/bin/xmvhdl
<xcelium_install_dir>/tools.lnx86/inca/bin/xmelab
<xcelium_install_dir>/tools.lnx86/inca/bin/xmsim
```

To specify a custom directory in which to save the `xcelium_bin_paths.log` file, you must include `-xmlibdirname <directory_path>` on the `xrun` command line:

```
% xrun top.vhd -log_install_path dynamic -xmlibdirname vhdlSnapshot
```

In this case, the tool saves the log file to the custom directory called `vhdlSnapshot` with dynamic symlink path information.

```
<xcelium_install_dir>/tools/bin/xrun
<xcelium_install_dir>/tools/bin/xmvhdl
<xcelium_install_dir>/tools/bin/xmelab
<xcelium_install_dir>/tools/bin/xmsim
```

XRUN History

The history of each *xrun* invocation is written to an `xrun.history` file by default. This file records information about the environment settings and each run in sequence by date and timestamp, where the timestamps of the environment (*e#*) and each associated command (*s#*) are identical. The environment settings are written to an `env.history.<date_timestamp>` log file inside the `xcelium.d/env.d` directory. The path of the file is relative to the run directory.

For example:

```
more xrun.history
#e1(23Aug2023:16:51:17): source xcelium.d/env.d/env.history.23Aug2023_16_51_17
s1(23Aug2023:16:51:17): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_1
#e2(23Aug2023:16:51:36): source xcelium.d/env.d/env.history.23Aug2023_16_51_36
s2(23Aug2023:16:51:36): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_2
#e3(23Aug2023:16:51:47): source xcelium.d/env.d/env.history.23Aug2023_16_51_47
s3(23Aug2023:16:51:47): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_3 -
clean
```

✓ To change the name of the `xrun.history` file, specify the `-history_file` option on the `xrun` command line.

Alternatively, you can choose to turn off `env.history.*` file generation by specifying the `-noenvhistory` option on the command line. The next example shows how the simulator also stops dumping the environment (*e#*) information to the `xrun.history` file for that step.

```
more xrun.history
#e1(23Aug2023:16:51:17): source xcelium.d/env.d/env.history.23Aug2023_16_51_17
s1(23Aug2023:16:51:17): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_1
s2(23Aug2023:16:51:36): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_2 -
noenvhistory
s3(23Aug2023:16:51:47): xrun -sv tb.sv -input inp.tcl -access +wr -seed 1 -define RUN_3 -
clean -noenvhistory
```

You can turn off `xrun.history` file generation by specifying the `-nohistory` option on the command line; however, in this case, the tool continues to save `env.history.<date_timestamp>` log files inside the `xcelium.d/env.d` directory. To turn off both `xrun.history` and `env.history.*` file generation, you must specify `-nohistory` and `-noenvhistory` on the same xrun command line.

XRUN Input Files

The input files specified on the `xrun` command line must have a full, local, or relative path. The type of a file is determined by its file extension. The following table shows the default mapping of file extensions to file type.

File Type	File Extensions
Verilog	`.v, .vp, .vs, .V, .VP, .VS`
Verilog 1995	`.v95, .V95, .v95p, .V95P`
SystemVerilog	`.sv, .SV, .svp, .SVP, .svi, .svh, .vlib, .VLIB`
VHDL	`.vhd, .vhdp, .vhdl, .vhdlp,` `.VHD, .VHDP, .VHDL, .VHDLP`
VHDL configuration	`.vhcfg`
Specman ***e***	`.e, .E`
Compiled ***e***-library	`.elib`
Verilog-AMS	`.vams, .VAMS`
VHDL-AMS	`.vha, .VHA, .vhams, .VHAMS, .vhms, .VHMS`
VIP Library	`.viplib`

PSL file for Verilog	`.pslvlog`
PSL file for VHDL	`.pslvhdl`
PSL file for SystemC	`.pslsc`
C	`.c`
C++	`.cpp, .cc`
Assembly	`.s`
Compiled object	`.o`
Compiled archive	`.a`
Dynamic library	`.so, .sl`
SPICE file	`.scs, .sp`

Specman Input Files

When specifying Specman input files on the command line, one or more precompiled ***e***-library files may be specified given the following rules:

- `.elib` files and `.e` files both support the `-nosncomp` option. If you combine this option with ***e*** files and ***e***-library files on the same command line, then *xrun* loads the `.elib` files first.
- `.elib` files are only supported with the default generation engine, IntelliGen. Attempting to compile or link ***e***-library files with the `-pgen` option results in an error.

The *xrun* utility only supports the compilation of ***e*** files on top of previously compiled ***e***-library files. It does not support the creation of ***e***-library files.

If the `SPECMAN_PATH` environment variable has been set, *xrun* scans the specified directories to find ***e*** and ***e***-library files. You can also use the `-snpath` option to list directories to scan when loading ***e*** files or compiling ***e*** and ***e***-library files. Any path you specify with this command-line option is prefixed to paths defined in the `SPECMAN_PATH` environment variable.

Single-Step XRUN Invocation

When using the *xrun* utility in single-step mode, Xcelium supports specifying all input files and all command-line options on one command line.

Command Syntax

The `xrun` command uses the following simple syntax in single-step mode:

```
% xrun [options] <source_files>
```

Example

Consider the mixed-language example below, which consists of four source files: `top.v`, `middle.vhd`, `sub.v`, and `verify.e`.

```
% xrun -ieee1364 -v93 -access +r -gui verify.e top.v middle.vhd sub.v
```

In this case:

- The files `top.v` and `sub.v` are recognized as Verilog files and are compiled by the Verilog parser *xmvlog*. The `-ieee1364` option is passed to the *xmvlog* compiler.
- The file `middle.vhd` is recognized as a VHDL file and is compiled by the VHDL parser *xmvhdl*. The `-v93` option is passed to the *xmvhdl* compiler.
- The file `verify.e` is recognized as a Specman ***e*** file and is compiled using `sn_compile.sh`.
- After compiling the files, *xrun* then calls *xmelab* to elaborate the design. The `-access` option is passed to the elaborator to provide read access to simulation objects.
- After the elaborator has generated a simulation snapshot, *xmsim* is invoked with both the SimVision and Specview graphical user interfaces.

Multi-Step XRUN Invocation

When using the *xrun* utility in multi-step mode, Xcelium supports a flexible use model that has the same behavior and feature set as traditional direct invocation.

Command Syntax

The following syntax is required for *xrun* in two-step mode:

```
% xrun -elaborate [other_options] <source_files>
% xrun -R
```

The following syntax is required for *xrun* in three-step mode:

```
% xrun -compile [compilation_options] <source_files>
% xrun -elaborate -elabonly -top [lib.]cell[:view] [elaboration_options]
% xrun -R
```

Multi-Step Invocation Options

The following table defines the *xrun* options that are required to enable the different modes of multi-step invocation:

-compile	Parses and compiles the specified source files but does not elaborate. Required for three-step mode.
-elaborate	Parses and compiles source files. elaborates the design, and generates a simulation snapshot but does not simulate. Required for two-step and three-step mode. If source files are already compiled, the tool recompiles any changed design units automatically. To skip this for re-elaboration, you must specify `-noupdate` on the command line.
-elabonly	Elaborates the design and generates a simulation snapshot only but does not parse or simulate. Required for three-step mode.
-R	Simulates the last snapshot generated by the *xrun* utility. Required for two-step and three-step mode. If a simulation is run with the `-snapshot` or `-name` option to specify a name for the simulation snapshot, you must also include this same specification when invoking `xrun -R`.
-top	Specifies the top-level HDL design unit to be elaborated and simulated in `[lib.]cell[:view]` format. Required for three-step mode. You can use multiple `-top` options to specify multiple top-level units.

Example

Consider the mixed-language example below, which consists of three HDL source files: `buf.v`, `and2.v`, and `top.vhd`. In this case, no `cds.lib` file is created to define libraries. All design units are compiled into a default work library called `worklib`. The following commands show how to simulate the design in direct invocation mode:

```
% xmvlog buf.v
% xmvlog and2.v
% xmvhdl -v93 top.vhd
% xmelab worklib.top:a
% xmsim worklib.top:a
```

These steps can be replicated with the following `xrun` command in single-step mode:

```
% xrun buf.v and2.v -v93 top.vhd -top top:a
```

You can also choose to simulate the design with `xrun` in two-step mode:

```
% xrun -elaborate buf.v and2.v -v93 top.vhd -top top:a
% xrun -R
```

Or in three-step mode:

```
% xrun -compile buf.v and2.v -v93 top.vhd
% xrun -elaborate -elabonly -top top:a
% xrun -R
```

Customizing XRUN

Although the *xrun* utility normally improves on the traditional method of using separate tools to compile, elaborate, and simulate a design, there are certain situations where the resulting command line can be complex and/or cumbersome. For such cases, you can choose to further customize *xrun* by including command-line options to override the default behaviors of the tool.

To customize your simulation using *xrun*, specify one or more of the following options:

- **`-f`** | **`-F`**: Uses an arguments file to specify files and options
- **`-*_ext`**: Changes the default set of file extensions
- **`-xmlibdriname`**: Changes the name of the scratch directory
- **`-xmlibdirpath`**: Changes the path to the scratch directory
- **`-work`**: Changes the name of the work library
- **`-filemap`**: Compiles source files with specific options

- **`-simtmp`**: Changes the save location of scratch area files
- **`-snapshot`**: Specifies a unique snapshot name
- **`-makelib`**: Compiles files into multiple libraries
- **`-fast_recompilation`**: Enables enhanced recompilation

In addition to the above options, *xrun* also includes functionality that can help you debug an application failure or manage the severity of output message codes (mnemonics). For more information on these features, see Customizing Crash Reports with XRUN and Changing the Severity of Messages.

Using an Arguments File to Specify Files and Options

An *xrun* command can become quite lengthy when adding all the options and files required to simulate an advanced design.

To simplify the command line, specify an arguments file. The *xrun* utility provides two options: `-f` and `-F`.

```
% xrun -f xrun.args         // Scans for files relative to the
                            // xrun invocation directory.
% xrun -F ./args/xrun.args  // Scans for files relative to the location of
                            // the arguments file xrun.args.
```

The named arguments file can specify a list of input files and command-line options. For example:

```
./vhdl_src/file.vhd
./vlog_src/file.v
./psl/file.pslvlog
-ieee1364
-notimingchecks
-access +rw
```

When specifying input files, the wildcard character `*` is supported, as shown:

```
/vlog/*
../rtl/*.v
../rtl/count*.vhd
../rtl/*count.vhd
../rtl/c*nt.vhd
/usr1/libs/rtl*54*stl/*.v
```

You can also use environment variables in an arguments file. The syntax is `${env_var}`. For example, if you set the environment variable `SRC` to point to a directory that contains source files, the arguments file can contain the variable `${SRC}`, as in the following example:

```
${SRC}/source1.v
${SRC}/source2.v
${SRC}/*.sv
```

Only single-line comments are supported in the arguments file. Multi-line comments are not supported. Comments must begin with a pound sign (`#`), two forward slashes (`//`), or two dashes (`--`). For example:

```
// File: xrun.args
-- This is a comment
# This is another comment
-ieee1364
source.v
```

Specifying XRUNROOT in an Arguments File

Designs that use the Universal Verification Methodology (UVM) support the `_XRUNROOT_` environment variable. This variable provides an alternative method for referencing the location of the UVM installation with the `-uvmhome` option or the location of the Cadence extensions to UVM with the `-uvmexthome` option when creating a `-f` or `-F` arguments file. For example, the following arguments file uses the `_XRUNROOT_` environment variable to point to the Cadence extensions for version 1.2 of UVM:

```
// args file
-clean
-uvmhome /UVM/2014/UVM12/uvm-1.2
-uvmexthome ${_XRUNROOT_}/methodology/UVM/CDNS-1.2
```

Using the `-f` option, you can then specify this file with on the command line, as shown:

```
% xrun -f args <source_files>
```

Generating a Command File to Use as an Arguments File

When using the *xrun* utility across different terminals in your Xcelium environment, you might find that you are using the same command and setting the same environment variables over and over again to keep consistent settings. To avoid this and improve the convenience of the tool, you can specify the `-gen_xrun_opts` option on the command line, which expands an original `xrun` command and any defined settings and then saves this information to a specified filename in an easy-to-read format.

You can use this generated command file to correlate the following information used in the original run:

- All command options and their arguments.
- Any umbrella options (in comments) with a list of their sub-options.

- Any set environment variables (in comments) with their values expanded.
- Any `-f`/`-F` files (in comments) with their contents dumped in list format.

  In the case of `-F` files, the directory of the file is appended as a prefix of the files passed with this option.

For example:

```
% xrun -f run.f -gen_xrun_opts args_file
```

After running the above command, all command options, arguments, and environment variables are expanded and dumped to the file named `args_file`. Later on, you can use `args_file` with the `-f` option to run the same simulation with *xrun* in single-step mode.

```
% xrun -f args_file
```

XRUN Utility File Types and Extensions

The *xrun* utility uses the file extensions of the input files specified on the command line to determine their file type. Each recognized file type has a built-in, predefined set of file extensions. To change, or add to, the list of file extensions mapped to a given language, there is a command-line option that you can use for each file type.

The following table shows the recognized file types, the set of built-in, predefined file extensions for each language type, and the command-line option you can use to change, or add to, the list of file extensions mapped to a given language.

File Type	Defined File Extensions	Option
Verilog	`.v, .V, .vp, .VP, .vs, .VS`	`-vlog_ext`
Verilog 1995	`.v95, .V95, .v95p, .V95P`	`-vlog95_ext`
SystemVerilog	`.sv, .SV, .svp, .SVP, .svi, .svh, .vlib, .VLIB`	`-sysv_ext`
VHDL	`.vhd, .VHD, .vhdl, .VHDL, .vhdp, .VHDP, .vhdlp, .VHDLP`	`-vhdl_ext`
VHDL configuration	`.vhcfg, .vhcfgp`	`-vhcfg_ext`
Specman ***e***	`.e, .E`	`-e_ext`
Verilog-AMS	`.vams, .VAMS`	`-amsvlog_ext`

VHDL-AMS	`.vha`, `.VHA`, `.vhams`, `.VHAMS`, `.vhms`, `.VHMS`	`-amsvhdl_ext`
PSL file for Verilog	`.pslvlog`	`-propvlog_ext`
PSL file for VHDL	`.pslvhdl`	`-propvhdl_ext`
PSL file for SystemC	`.pslsc`	`-propsc_ext`
C	`.c`	`-c_ext`
C++	`.cpp`, `.cc`	`-cpp_ext`
Assembly	`.s`	`-as_ext`
Compiled object	`.o`	`-o_ext`
Compiled archive	`.a`	`-a_ext`
Dynamic library	`.so`, `.sl`	`-dynlib_ext`
SPICE file	`.scs`, `.sp`	`-spice_ext`

For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. If you have Verilog files with other extensions (for example, `.rtl` and `.vg`), you must specify that these are valid extensions for Verilog files. Using the `-vlog_ext` option, you can:

- Replace the list of built-in, predefined extensions with a new list. For example, the following option specifies that the valid extensions for Verilog files are `.v`, `.rtl`, and `.vg`:

  ```
  -vlog_ext .v,.rtl,.vg
  ```

- Add extensions to the list of built-in, predefined extensions by using a plus sign (`+`) argument. For example, the following option adds `.rtl` and `.vg`.

  ```
  -vlog_ext +.rtl,.vg
  ```

  This is the same as:

  ```
  -vlog_ext .v,.vp,.vs,.V,.VP,.VS,.rtl,.vg
  ```

> If a file has no extension, you can also use the plus sign (`+`) argument to specify its file type. For example, the following option specifies the SystemVerilog file `mysvtop`:
>
> ```
> % xrun -sysv_ext +, mysvtop
> ```

You can also include extension options in the definition of the `XRUNOPTS` variable in an `hdl.var` file. For example:

```
#hdl.var file
DEFINE XRUNOPTS -vlog_ext .v,.vg,.rtl [other_options]
```

The *xrun* utility generates an error if it encounters a file with an extension it does not recognize. In addition to using one of the extension options to override the set of recognized extensions for a particular file type, you can use the `-default_ext` option to specify the file type for files with unrecognized extensions.

The argument to the `-default_ext` option is a string that represents a file type.

Argument to -default_ext	**File Type**
`verilog`	Verilog HDL
`vcnf`	Verilog configuration
`verilog95`	Verilog 1995 HDL
`systemverilog`	SystemVerilog HDL
`vhdl`	VHDL HDL
`vhcfg`	VHDL configuration
`e`	Specman ***e***
`verilog-ams`	Verilog-AMS HDL
`vhdl-ams`	VHDL-AMS HDL
`psl_vlog`	PSL file for Verilog
`psl_vhdl`	PSL file for VHDL
`psl_sc`	PSL file for SystemC

`c`	C file
`cpp`	C++ file
`assembly`	Assembly
`o`	Compiled object
`a`	Compiled archive
`so`	Dynamic library
`scs`	SPICE file

For example, suppose that you have Verilog files with `.v`, `.vlog`, and `.vg` file extensions. Because `.v` is a defined file extension for Verilog files, *xrun* recognizes files with a `.v` extension. However, the tool is unable to determine the file type for files with `.vlog` or `.vg` extensions because these extensions do not map to any file type. If you use the `-default_ext verilog` option, *xrun* treats all files with these undefined extensions as Verilog files.

Modifying the Scratch Directory

By default, *xrun* generates a scratch directory during simulation called `xcelium.d` and saves it to the current working directory.

- You can choose to change the name of this directory using the `-xmlibdirname` option, or
- You can choose to change the location of this directory using the the `-xmlibdirpath` option

Changing the Name of the Scratch Directory

To change the name of the default scratch directory, specify `-xmlibdirname` on the *xrun* command line using the following syntax:

```
% xrun -xmlibdirname <name> [other_options] <source_files>
```

For example:

```
% xrun -xmlibdirname XRUN_libs dut.v tb.v
```

This command changes the name of the scratch directory from its default name `xcelium.d` to `XRUN_libs`. This is saved to the current working directory by *xrun*.

Changing Name and Path Information Using -XMLIBDIRNAME

When specifying a custom name for the `xcelium.d` scratch directory, you can also include path information. Path statements passed to the `-xmlibdirname` option can be relative or absolute.

To specify a relative path using `-xmlibdirname`, use the following syntax:

```
% xrun -xmlibdirname <relative_path>/<name> [other_options] <source_files>
```

For example:

```
% xrun -xmlibdirname ../XRUN_libs dut.v tb.v
```

This command changes the name of the scratch directory to `XRUN_libs`, but instead of saving it to the current working directory, the tool saves the directory using a relative path that is located above the current working directory.

To specify an absolute path using `-xmlibdirname`, use the following syntax:

```
% xrun -xmlibdirname <absolute_path>/<name> [other_options] <source_files>
```

For example:

```
% xrun -xmlibdirname /tests/RTL/dut/XRUN_libs dut.v tb.v
```

This command specifies an absolute path to `XRUN_libs` with multiple hierarchical levels. The directory `/tests/RTL/dut` must exist in order for *xrun* to create the new `XRUN_libs` location. If this location does not exist, *xrun* halts and generates an `NWRKDRA` error.

Changing the Path to the Scratch Directory

Altrernatively, when specifying a relative path for storing new libraries, you can use the `-xmlibdirpath` option on the *xrun* command line.

⚠ You can use this option by itself or together with `-xmlibdirname`; however, when using `-xmlibdirname` together with `-xmlibdirpath`, the directory name specified with `-xmlibdirname` must not contain any path information.

To specify a relative path using -xmlibdirpath, use the following syntax:

```
% xrun -xmlibdirpath <relative_path>/<name> [other_options] <source_files>
```

For example:

```
% xrun -xmlibdirpath ./libraries dut.v tb.v
```

This command generates a new `xcelium.d` directory in the relative location `./libraries`, which is located in the current working directory. Note that `./libraries` must already exist in order to avoid an error.

```
% xrun -xmlibdirname XRUN_libs -xmlibdirpath ../libraries dut.v tb.v
```

This second command adds the `-xmlibdirname` option to change the name of the scratch directory to `XRUN_libs`. Here, the `-xmlibdirpath` option is used to save the custom scratch directory to the relative location `../libraries`, which is one level above the current working directory. Note that `../libraries` must already exist in order to avoid an error. That is, if `../libraries` is available, then *xrun* generates `XRUN_libs` at the specified location.

Changing the Name of the Work Library

All design units in HDL files are compiled into a work library called `worklib` (located within the `xcelium.d` directory tree).

To change the name of this library, you can:

- Define the work library by using the `WORK` variable in an `hdl.var` file. For example:

  ```
  # hdl.var
  DEFINE WORK mylib
  ```

- Add the `-work` option to the *xrun* command line using the following syntax:

  ```
  % xrun -work <name> [other_options] <source_files>
  ```

Examples:

```
% xrun -work mylib [other_options] dut.v tb.v
```

The above command creates a work library called `mylib`. Using the default scratch directory, this work library is saved to `xcelium.d/mylib`.

```
% xrun -xmlibdirname XRUN_libs -work mylib [other_options] input_files
```

This second command also creates a work library called `mylib`. In this case. the `-xmlibdirname` option changes the name of the default scratch directory to `XRUN_libs`, meaning that the work library is saved to `XRUN_libs/mylib`.

Compiling Source Files with Specific Options

When source files are compiled using *xrun*, compiler options specified on the command line apply to all listed files. For example, the following command includes four options.

```
% xrun -v93 -controlrelax ARSHCH -ieee1364 -linedebug top.v middle.vhd sub.v
```

The `-v93` and `-controlrelax` options are VHDL-specific and are passed to the *xmvhdl* compiler when the file `middle.vhd` is compiled. The `-ieee1364` option is Verilog-specific and is passed to the *xmvlog* compiler when `top.v` and `sub.v` are compiled. The `-linedebug` option is common to both Verilog and VHDL and is passed to both compilers.

You can also include these options in the definition of the `XRUNOPTS` variable in an `hdl.var` file. For example:

```
# hdl.var
DEFINE XRUNOPTS -v93 -relax -ieee1364 -linedebug
```

```
% xrun top.v middle.vhd sub.v
```

In some cases, it might be necessary to compile some Verilog or VHDL source files with a specific option, or set of options, and other files with different options. There are two ways to do this:

- Use the `-filemap` and `-endfilemap` options on the `xrun` command line.

  These options start and end a filemap collection. The syntax is as follows:

  ```
  -filemap source_file [source_file ...] list_of_options -endfilemap
  ```

  Example:

  ```
  % xrun file1.v -filemap file2.v -view view1 -v1995 -endfilemap -assert
  ```

  In this example, `file1.v` is processed with `-assert` only. The file `file2.v` is processed with `-assert`, `-view`, and `-v1995`.

  See the description of the `-filemap` option for more details on using the `-filemap` and `-endfilemap` options.

- Define the `FILE_OPT_MAP` variable in an `hdl.var` file.

  To define the `FILE_OPT_MAP` variable, specify either a filename or a directory, followed by a list of options. The format is as follows:

  ```
  DEFINE FILE_OPT_MAP ( {filename | directory} => list_of_options[, {filename |
  directory} => list_of_options ...)
  ```

  The options are separated by a space.

If a filename is specified, the options are applied to that file. If a directory is specified, the options are applied to all files in that directory.

The backslash character (`\`) can be used to define the variable over multiple lines.

```
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug)
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -linedebug, file1.v => -view view1 -v1995)
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug, \
                     file1.v => -view view1 -v1995)
```

Only one `FILE_OPT_MAP` variable can be defined in an `hdl.var` file.

Example:

```
# File: hdl.var
DEFINE FILE_OPT_MAP (file1.vhd => -v93 -controlrelax ARSHCH -linedebug, \
                     ./src/vhdl/file2.vhd => -controlrelax ARSHCH -v93, \
                     ./src/vhdl => -v93, \
                     ./src/vlog/file1.v => -view view1 -v1995)
```

In this example:

- `file1.vhd` is compiled with the `-v93`, `-controlrelax`, and `-linedebug` options.
- `./src/vhdl/file2.vhd` is compiled with the `-controlrelax` and `-v93` options.
- All other files in the directory `./src/vhdl` are compiled with the `-v93` option.
- `./src/vlog/file1.v` is compiled with the `-view` and `-v1995` options.

Not all parser options can be used in the definition of the `FILE_OPT_MAP` variable or with the `-filemap`/`-endfilemap` options. The following options are not supported:

`-ams`	`-hdlvar`	`-ovl`
`-cdslib`	`-ial`	`-smartorder`
`-cds_implicit_tmpdir`	`-xmerror`	`-smartscript`
`-cds_implicit_tmponly`	`-xmfatal`	`-specificunit`
`-cmdfile`	`-neverwarn`	`-status`

`-define`	`-nocopyright`	`-sv`
`-design_top`	`-nolink`	`-zlib`
`-errormax`	`-nowarn`	`-zparse`

Changing the Storage Location of Scratch Area Files

During simulation, *xrun* stores scratch area files, such as the following, in a temporary directory under the scratch subdirectory.

- `xmsim.args`
- `xmsim.env`
- `xrun.args`

By default, the scratch subdirectory is called `run.`*`<platform|platform.64>`*`.`*`<xrun_version>`*`.d`, and a symbolic link `run.d` is created that points to the same location. The temporary directory is named *`<hostid>`*`_`*`<process_id>`*`[.`*`<n>`*`]` as shown below.

`xcelium.d/run.d/`*`<hostid>`*`_`*`<process_id>`*`[.`*`<n>`*`]`

> See Specifying a Snapshot Name for information on changing the default name of the scratch subdirectory.

If `xcelium.d` no longer has write permission, then *xrun* generates a SUBSCR warning and automatically uses `/tmp` as the storage location.

As an alternative to the above, use the command-line option `-simtmp` to specify a custom location for the temporary directory *`<hostid>`*`_`*`<process_id>`*`[.`*`<n>`*`]`, which does not depend on `xcelium.d` write permissions. Specifying a custom location can also help to reduce NFS traffic to the same `xcelium.d` or `tmp` directory when parallel simulations are launched.

The directory specified to `-simtmp` can be relative or absolute.

Example:

```
// test.sv
module test;
   reg a;
endmodule
```

```
% xrun -elaborate test.sv
% mkdir -p /usr2/tmp1/myscratch
% xrun -R -simtmp /usr2/tmp1/myscratch
```

Here, the first command uses *xrun* to parse and compile the `test.sv` source file, elaborate the design, and generate a simulation snapshot. The second command creates the directory `/usr2/tmp1/myscratch` using an absolute path specification. The final command uses *xrun* to simulate the snapshot. The `-simtmp` option specifies that the scratch area files are to be stored at the custom `myscratch` location.

If you want to retain the temporary directory and all scratch related files, add the option `-noremovescratch` to the `xrun` command line.

Specifying a Snapshot Name

After invoking the appropriate compiler to compile the source files specified on the command line, *xrun* invokes *xmelab* to elaborate the design. The elaborator generates a simulation snapshot that is the input to the simulator (*xmsim*). Simulation snapshots are given default names in *`lib.cell:view`* format.

- For Verilog, the default snapshot name is:

  ```
  library.top_level_module_name:view
  ```

  where *`library`* is the working library (default as `worklib`), and *`top_level_module_name`* is the name of the top-level module (or the name of the first top-level module encountered when parsing the source files). The *`view`* name is the file extension of the file that contains the description of the top-level module. For example, if the name of the top-level module is `top`, and the module is described in `test.v`, and this module is compiled into the default `worklib` library, the snapshot is called `worklib.top:v`.

- For VHDL, the default snapshot name is:

  ```
  library.top_level_entity_name:architecture
  ```

  For example, if the top-level entity is called `counter_test` and the architecture is called `bench`, the snapshot is called `worklib.counter_test:bench`.

To give different elaborations of your design unique snapshot names, use the `-snapshot` option. For example, the following command generates a snapshot called `worklib.run1:v`.

```
% xrun -snapshot run1 file1.v file2.v
```

The `-snapshot` option also changes the name of the scratch subdirectory under the `xcelium.d` directory. *xrun* uses this directory as a scratch area to create files and pass them between tools and to track information necessary to execute *xrun* multiple times. By default, this subdirectory is called:

`run.<platform|platform.64>.<xrun_version>.d`

For example:

```
run.lnx86.16.11.d/
```

As a convenience, a symbolic link named `run.d` is created that points to the *xrun* scratch subdirectory.

If you specify `-snapshot run1` on the `xrun` command line, the tool generates a directory called `run1.lnx86.16.11.d/`. The symbolic link is `run1.d`.

Both `-snapshot` and `-name` are synonyms for the same option.

Enabling Enhanced Recompilation

When working with a larger design, or a design that requires temporary access to distinct source IP files, the default *xrun* recompilation flow can reduce simulator performance or introduce unnecessary recompilation and re-elaboration. For instance, the following command sequence compiles multiple source files, two of which are source IP files. A temporary source IP file, `ip2.v`, is removed after elaborating the initial snapshot:

```
% xrun -compile ip1.v
% xrun -compile ip2.v
% xrun -compile dut.v
% xrun -elaborate -top top test.v
% rm ip2.v
% xrun -elaborate -top top test.v
```

With the default recompilation flow, *xrun* references the initial snapshot during the second elaboration step to check the files and timestamps. Because there is a reference to `ip2.v` in the initial snapshot, and this file was removed, *xrun* initiates an unnecessary recompilation during the second elaboration and the following message is printed to the screen:

```
    Recompiling... reason: unable to stat
    '/./examples/inca_test/recompile/testcase2/ip2.v'.
```

Furthermore, even in cases where no files have changed, *xrun* still reads the snapshot and checks all associated file timestamps, which can reduce performance on larger designs.

You can specify the `-fast_recompilation` option on the command line to enable the enhanced recompilation flow. This option allows *xrun* to avoid unnecessary recompilation and re-elaboration while improving performance with larger designs.

When using enhanced recompilation, Cadence recommends that you organize each *xrun* command in such a way that it builds its own separate work library instead of the default `xcelium.d` work library. You can use the `-xmlibdirname` option to specify a custom name in place of the default `xcelium.d` work library.

For instance, the `ip2.v` example above changes as follows if you specify `-fast_recompilation` to use the enhanced recompilation flow. The `-xmlibdirname` option specifies separate libraries for each compilation step, and these generated libraries are defined using the `-reflib` option for the final elaboration step.

```
% xrun -compile -fast_recompilation -xmlibdirname iplib1 -work ip1 ip1.v
% xrun -compile -fast_recompilation -xmlibdirname iplib2 -work ip2 ip2.v
% xrun -compile -fast_recompilation -xmlibdirname dutlib -work dut dut.v
% xrun -elaborate -fast_recompilation -top top -reflib iplib1/ip1 \
      -reflib iplib2/ip2 -reflib dutlib/dut test.v
% rm ip2.v
% xrun -elaborate -fast_recompilation -top top -reflib iplib1/ip1 \
      -reflib iplib2/ip2 -reflib dutlib/dut test.v
```

Each `xrun` command line must specify the `-fast_recompilation` option, in order for the enhanced recompilation flow to function correctly. If any *xrun* command line does not specify this option, then the following error results:

```
xrun: *E,FST_INCONSISTENT: Inconsistent usage of Fast Recompilation.
```

Customizing Crash Reports with XRUN

Specify the `XLM_ERRORLOG` environment variable to enable *xrun* to generate a report whenever there is an application failure (internal error or crash) with one of the core binaries which *xrun* invokes, such as *xmvlog*, *xmelab*, or *xmsim*. The crash report automatically records the user environment, software version, exit status, stack trace (and so on), to assist in crash investigations. You can customize the crash report behavior by specifying `XLM_ERRORLOG` with one or more parameters.

This feature is supported on LINUX only for this Xcelium release.

For instance, one key capability of this feature is to save all crash reports to a centralized location, by defining either the `DIR` or `AUTO_DIR` parameter. Each parameter can be set as an absolute path or as a path relative to the application startup directory.

```
setenv XLM_ERRORLOG 'DIR=/home/jsmith/errorlogs'
```

Or:

```
setenv XLM_ERRORLOG 'DIR=../<directory_name>/errorlogs'
```

With `DIR` or `AUTO_DIR`, you can collect all crash information in one place, including Cadence Support Investigation (CSI) reports, making it easier to send everything to Cadence for analysis. Subsequently, this information may assist in debugging critical software problems and may also provide a means of tracking essential data, which can be useful when attempting to identify key failure trends across specific applications.

You can define the following parameters, as name-value pairs, when specifying `XLM_ERRORLOG`.

ⓘ You only need to specify an explicit value if you want to override the default.

Parameter	Description
`EXTENDED=[TRUE | FALSE]`	Set the `EXTENDED` parameter to enable or disable the collection of additional information for debug analysis. This parameter takes a Boolean `TRUE/FALSE` or `YES/NO` value as its argument. The default is `YES/TRUE`. If set to `YES/TRUE`, additional system information is recorded for analysis (including the values of environment variables that differ from their default values). If set to `NO/FALSE`, no additional information is recorded. For more details, see Reporting Additional Crash Data.
`ALWAYS=[YES | NO]`	Set the `ALWAYS` parameter to enable or disable debug mode. This parameter takes a Boolean `TRUE/FALSE` or `YES/NO` value as its argument. The default is `NO/FALSE`. If set to `YES/TRUE`, *xrun* always generates debug information regardless of whether or not there is an application failure. If set to `NO/FALSE`, *xrun* generates debug information only if there is an application failure.
`STARTINFO_FILE=<filename>`	Set the `STARTINFO_FILE` parameter to specify a file with which to log each session from beginning to end. For more details, see Crash Trend Reporting.

`SCRIPT=<path/to/script>`	Set the `SCRIPT` parameter to specify a custom shell script file to execute when any of the core *xrun* binaries fail. You can set the variable as:
	`setenv XLM_ERRORLOG "SCRIPT=/path/to/<file>.sh"`
	The script file passes the following arguments:
	`-status <exit_status>`
	`-log <path_to_CDS.log_file>`
	`-t <crash_time>`
	`-dir <log_directory>`
	`-exe <path_to_application_binary>`

`DIR=<path>`	Set the `DIR` parameter to specify a unique centralized location to save the reports. If you specify a directory that does not currently exist, *xrun* automatically creates the directory. ⓘ If you do not set this parameter, and `AUTO_DIR` is not set, *xrun* saves the reports to the default `/tmp` directory.

`AUTO_DIR=<path>`	Set the `AUTO_DIR` parameter to specify a unique location to group the reports together by month and year.
	`<path>/year_month`
	For example:
	`% setenv XLM_ERRORLOG "AUTO_DIR=autoDir"`
	`% ls -a autoDir/18_04/`
	`...`
	ⓘ If you do not set this parameter, and `DIR` is not set, *xrun* saves the reports to the default `/tmp` directory.

`ALL_ERRORS=[TRUE | FALSE]`	Set the `ALL_ERRORS` parameter to control how *xrun* reports on crash information.
	
	This parameter takes a Boolean `TRUE/FALSE` value as its argument. The default is `FALSE`.
	
	If set to `TRUE`, *xrun* includes all user-generated errors in the crash report. In the `xrun.log` file, the following statement is appended when the process is terminated:
	
	`Process was terminated by USER with #### signal`
	
	If set to `FALSE`, *xrun* includes only system-generated errors in the crash report. In the `xrun.log` file, the following statement is appended when the process is terminated:
	
	`Process was terminated with #### signal`
	
	Optionally, you can run the `cdsLibDebug` and `cdsinfo` commands to add more information to the crash report. The `cdsLibDebug` command enables you to get a list of `cds.lib` files and library definitions. It also enables you to find syntax errors in `cds.lib` files. The `cdsinfo` command provides information on design libraries, such as `DMTYPE` or `NAMESPACE`.

`NAME_FMT=<`*`keywords`*`>`	Set the `NAME_FMT` parameter to define the crash report name. By default, the crash report is named as follows: `crashReport_<`*`DATE`*`>_<`*`TIME`*`>_<`*`SUBVERSION`*`>_<`*`USER`*`>_<`*`HOST`*`>.log` Alternatively, you can change the name by specifying different keywords. For details, see Customizing the Crash Report Name.

`USE_DEBUGGER=[YES | NO]`	Set the `USE_DEBUGGER` parameter to control debug behavior after simulation. By default, *xrun* reports on the status of all core binaries that it invokes after each process has exited. The one exception is the simulator (*xmsim*), which *xrun* executes but which does not return control back to *xrun* after finishing a run. This parameter takes a Boolean `TRUE/FALSE` or `YES/NO` value as its argument. The default is `YES/TRUE`. If set to `YES/TRUE`, *xmsim* attempts to run the debugger immediately after an internal failure to get additional stack information. If set to `NO/FALSE`, *xmsim* does not use the debugger. For details, see Specifying a Runtime Debugger.
`DISABLED=[YES | NO]`	Set the `DISABLED` parameter to switch off the crash reporting functionality. This parameter takes a Boolean `TRUE/FALSE` or `YES/NO` value as its argument. The default is `NO/FALSE`. If set to `YES/TRUE`, *xrun* does not generate a crash report. If set to `NO/FALSE`, *xrun* ignores the parameter and generates a crash report.
`SAVELOGS=[ALWAYS | ONCRASH | OFF]`	Set the `SAVELOGS` parameter to enable *xrun* to call the log collection script and automatically save the current `xrun.log` file. Accepted values are: • `ALWAYS` • `ONCRASH` • `OFF` The default, `ONCRASH`, saves a copy of `xrun.log` to the crash directory on exit.

`LOG_EMAIL=<app:email>`	Set the `LOG_EMAIL` parameter to send a system-generated notification to all the e-mail addresses specified for the listed application, in the case of application failure (that is, a non-zero exit).
	
	For example:
	
	`LOG_EMAIL=xmvlog:abc@cadence.com,xyz@customer.com`
	
	You can also specify multiple applications/binaries:
	
	`LOG_EMAIL=xmvlog:abc@cadence.com,xyz@customer.com,`
	`xmelab:abc@cadence.com,pqr@customer.com`
	
	If `$XLM_ERRORLOG LOG_EMAIL=<app:email>` is set, and if there is an application failure, then the first 50 lines of the `xrun.log` file comes as an attachment in the resulting e-mail notification.

`GRAPH=[YES | NO]`	Set the `GRAPH` parameter to control GUI mode. This parameter takes a Boolean `TRUE/FALSE` or `YES/NO` value. The default is `NO/FALSE`, meaning GUI mode is off. If set to `YES/TRUE`, GUI mode is enabled.

Example:

```
setenv XLM_ERRORLOG "SAVELOGS=ONCRASH EXTENDED=TRUE ALWAYS=NO
STARTINFO_FILE=/home/genel/startTimes_`date +%b%Y` DIR=/home/genel/crashes
NAME_FMT=crashReport_%DATE_%TIME_%SUBVERSION_%USER_%HOST.log USE_DEBUGGER=YES DISABLED=NO
LOG_EMAIL=xmvlog:abc@cadence.com,xyz@customer.com"
```

Specifying a Runtime Debugger

By default, *xrun* reports on the status of all core binaries that it invokes after each process has exited. The one exception is the simulator (*xmsim*), which *xrun* executes but which does not return control back to *xrun* after finishing a run.

Set the `USE_DEBUGGER` parameter to `YES`, to have *xmsim* attempt to run the debugger defined in your `$PATH` immediately after a simulator internal error to get additional stack information.

Older versions of debuggers, such as GDB, may not correctly handle Xcelium code, which can result in the generation of incomplete stack traces. It is therefore recommended that your `PATH` be set to the latest version of GDB, for example.

If `USE_DEBUGGER` is set to `YES`, but no debugger is set in your `$PATH`, then the crash report notes that a debugger was not available on exit.

```
debug:Warning : no platform debugger found
debug:Check your system configuration
```

Additionally, you can use the `DEBUGGER` variable, which attaches the specified debugger to the current process even if `USE_DEBUGGER` is not set. For example:

```
setenv DEBUGGER /opt/gdb/8.1/bin/gdb
```

If the `DEBUGGER` variable is not defined, and if there is no debugger in your `$PATH`, then no additional stack trace information is saved to the crash report.

Reporting Crash Trends

You can track data from the beginning to the end of a session by adding the `STARTINFO_FILE` parameter to the `XLM_ERRORLOG` environment variable.

Cadence recommends logging session information only on a local, rather than a global, site basis to ensure that *xrun* performance is not impacted.

After application exit, whether as a result of a crash or normal exit, *xrun* collects a range of system data when this parameter is defined. The resulting information can then be analyzed to measure trends. For example, the number of session starts against the number of system crashes.

If the `STARTINFO_FILE` parameter is set to point to a writable file, each *xrun* session is logged.

This log includes details of the program version, process ID, username, start/end dates, and also times, or, in the case of a crash, the start/crash date and time are recorded. Details of the process ID and crash report file name are also included. This can be either an absolute file path or the filename (which can be created and updated using the `DIR` parameter or `/tmp`).

The log file can greatly increase in size following numerous start/stop session recordings. It is therefore suggested that you consider making the files date dependent on a daily, weekly, or monthly basis. For example, `/tmp/startinfo_04_2018`. You can set the file name to automatically change using the following environment variable:

```
setenv XLM_ERRORLOG "STARTINFO_FILE=/home/jsmith/startTimes_`date +%b%Y`"
```

For example:

```
[ 7555@mymachine@jsmith ] start date/time:Sat Mar 21 06:37:57 2018 xrun XCELIUM18.03.001
plat:lnx86
```

```
crash report file :: /tmp crashReport_03212018_0638_XCELIUM18.03.001_jsmith_mymachine.log
```

`STARTINFO_FILE` contains the name of the "startinfo file" under `XLM_ERRORLOG` (or under `/tmp` if `DIR` is not set). The "startinfo file" is created automatically if it does not exist, or it can point to an existing file.

⚠ If you shutdown an *xrun* process using the `kill -9` command, the `STARTINFO_FILE` does not record the exit/crash.

To send the stack trace to the log file when the application hangs, type the following at the command prompt:

```
kill -s USR1 <pid>
```

Availability of Signal Description Information for Crashes

The crash report also displays the termination signal name and number, for example `signal:Hangup(1)`. This provides additional information on the type of crash that has occurred, such as internal application errors recorded as, for example, `#11 (segFault)` or `#4 (illegal instruction)`. Alternatively, if the software was forcefully interrupted it notes that action, for example `#15 (sigterm)` and `#6 (sigabort)`.

Hence, when a signal description is available to describe the type of crash, the STARTINFO_FILE displays it as follows:

```
[7555@mymachine@jsmith] 11(SegFault):Wed Mar 11 13:16:03 2018 xmelab XCELIUM18.03.001
lnx86
```

It should be noted, however, that signal numbers and descriptions may vary from platform to platform.

Reporting Additional Crash Data

Add the `EXTENDED` parameter to the `XLM_ERRORLOG` environment variable, and set its value to `TRUE`, if you require additional crash data to be collected.

If `EXTENDED` is activated, the crash report records details of your environment variable settings and the content of any `.cdsinit` or `.cdsenv` files.

- If `.cdsinit` exists, the content of the file is included in the crash data.
- If `.cdsenv` exists, the name and value of any cds environment variables that differ from their

default values are included in the crash data.

The crash report can also provide `checkSysConf` and `xdpyinfo` information.

- For `checkSysConf`, the report includes the results of checks against the necessary pre-requisites to run Cadence software on the machine that the crash took place.
- For `xdpyinfo`, the report includes the results of checks against the available graphics display.

The `EXTENDED` parameter is set to `TRUE` by default.

Customizing the Crash Report Name

Each crash report uses the following default filename format:

```
"crashReport_<DATE>_<TIME>_<SUBVERSION>_<USER>_<HOST>.log"
```

Optionally, you can customize this filename format by setting the `NAME_FMT` parameter and specifying only those keywords which you want to use to redefine the generated filename. The following keywords are available:

Keyword	Is Substituted by...
`%DATE`	The current date
`%TIME`	The current time
`%VERSION`	The application version
`%SUBVERSION`	The application subversion
`%PID`	The process ID
`%USER`	The user name
`%HOST`	The host name

For example:

```
XLM_ERRORLOG='NAME_FMT=crashReport_%DATE_%PID_%SUBVERSION_%USER_%HOST.log'
```

could generate a report name of:

```
/home/jsmith/errorlogs/crashReport_02042018_1014_XCELIUM18.03.001_jsmith_msc080.log
```

There might be instances where several crash reports are generated at the same time, meaning that the timestamp is same for all reports. In such a scenario, the newly-generated crash report name is appended by a unique identifier. For example, `<generated-crash-report-name>_1.log`, `<generated-crash-report-name>_2.log` ... `<generated-crash-report-name>_N.log`, to avoid overwriting existing reports.

Changing the Severity of Messages

When simulating a design with *xrun*, you can expect to see message codes (mnemonics) for up to five different message types printed to the screen or saved to the log file.

From the command line, you can choose from a handful of options to help fine-tune the severity level of these messages.

For instance, use the following command-line options to increase the severity level of warnings and errors:

- `-xmerror`: This option increases the severity level of the specified warning message from warning to error.
- `-xmfatal`: This option increases the severity level of the specified warning or error message.
    - If the specified message is a warning, the severity level is increased to error.
    - If the specified message is an error, the severity level is increased to fatal.

To decrease the severity level of warnings and soft errors, use the command-line options below:

- `-xmnote`: This option decreases the severity level of the specified warning message from warning to note.
- `-xmwarn`: This option decreases the severity level of the specified soft error to a warning.

If you want to disable printing a specified warning or note, use `-nowarn`. Each message type is defined in the table below.

Type	Code	Description
Note	`N`	This message is for informational use only. It does not require any action.
Warning	`W`	This message identifies a possible issue in your source file. The issue might not prevent the tool from running, but you have the option to investigate and resolve it.

Error	`E`	This message identifies an actual issue in your source file. The issue prevents the software from running and requires action in order to resolve it.
Soft Error	`SE`	This message identifies something in the design that would normally be considered an error but, under certain circumstances, might result in a correct simulation. The issue can be downgraded to a warning using the `-xmwarn` command-line option.
Fatal	`F`	This message identifies a critical issue with the software. Critical issues require action from Cadence for resolution.

Using a Control File to Change the Severity Level

As an alternative to changing the severity level by manually specifying options on the command line, or in an arguments file using `-f`, *xrun* also supports a message control file. The message control file is an ASCII file that you can use as a single source to adjust the severity level of mnemonics printed to the standard output. Using this file can help reduce the amount of manual effort required to fine-tune messages from one Xcelium release to the next. For instance, you might use the message control file to specify different severity levels for the same mnemonic across different Xcelium binaries (such as *xrun*, *xmvlog*, *xmvhdl*, *xmelab*, and so on).

To specify a message control file, use the `-xmfile_msgcntl` option on the `xrun` command line. For example:

```
% xrun -xmfile_msgcntl xm_msg.ctl [other_options] foo.v
```

Only one message control file is supported. If you specify `-xmfile_msgcntl` multiple times on the `xrun` command line, an error results.

For more information on the format of the message control file, see The Message Control File.

Related Topic

- Listing the Severity of All Tool Messages

The Message Control File

The format of the message control file consists of one command per line, with each line having two fields as shown.

```
<command> [<xcelium_tool_name>.]<mnemonic>[:[<xcelium_tool_name>.]<mnemonic>...]
```

Where:

command	Specifies a value for controlling the severity level of the specified mnemonic. The following command values are supported: • `xmerror`: increases the severity of the specified warning to error. • `xmfatal`: increases the severity of the specified warning or error to fatal. • `xmnote`: decreases the severity of the specified warning to note. • `xmwarn`: decreases the severity of the specified soft error to a warning. • `xmnowarn`: disables printing the specified warning or note.
mnemonic	Specifies a list of one or more message codes (mnemonics) using the following syntax rules: • Use a colon `:` to separate each mnemonic in the list, when specifying more than one mnemonic for a given command. • Use a message type identifier (N, W, E, SE, F) to control the severity level of all mnemonics of that type. ◦ The wildcard character `*` is allowed for message types. For instance, `*W` adjusts all warnings and `*E` adjusts all errors. If you specify `*` without also specifying a type, then all message types are implied. ◦ If the message control file includes a line with an explicit *mnemonic* specification, that explicit specification overrides any corresponding message type specification. ⚠ If an invalid or unrecognized mnemonic is specified in the message control file, that mnemonic is ignored.

`xcelium_tool_name`	Specifies one of the binaries supported by *xrun*. Include this argument if you want to change the severity level of a given mnemonic but apply the change only to the specified tool. If the *`xcelium_tool_name`* argument is not included, then the severity level change applies to all tools invoked by *xrun*. ⚠ If an invalid or unrecognized tool is specified in the message control file, that tool name is ignored.

Comments are supported using the following formats:

```
# This is the first comment
// This is the second comment
/* This is the third comment */
```

Control File Precedence Rules

Commands are listed in the message control file from lowest to highest precedence. That is, a command specification listed at the bottom of the control file overrides any specification listed near the top of the control file.

Additionally, if a mnemonic is specified using a command-line control option, and that same mnemonic is specified in a message control file that you pass to the simulator with `-xmfile_msgcntl`, the command-line option specification takes precedence, and the message control file specification is ignored.

Control File Examples

The first example illustrates a common use case where a wildcard command changes the severity of all messages from warnings to errors, followed by another command which lists exceptional warning mnemonics for which the severity should not change.

⚠ Previously, this use case could result in multiple CNTSEV messages. In this Xcelium release, these spurious messages are no longer generated.

```
% cat xm_msg1.ctl
# upgrade all warnings to errors except OPSUPP & OPOBSO
xmerror *W
xmwarn OPSUPP:OPOBSO

% cat xm_msg2.ctl
```

```
// This illustrates comment style 2
// Note how you can also use separate xmwarn lines
xmwarn OPSUPP
xmwarn OPOBSO

% cat xm_msg3.ctl
# The first command upgrades UVMHOME & UVMINS to fatal errors
xmfatal UVMHOME:UVMINS
# The second command upgrades NOPERF to an error, but only for xmvlog
xmerror xmvlog.NOPERF
```

With Xcelium, there can be duplicate messages with different meanings between the supported tool binaries. This second example shows how to use the message control file to modify a specific mnemonic, adding modifications to the message which apply to different tools:

```
% cat xm_msg4.ctl
/* This illustrates comment style 3 */
/* The first command downgrades DLCPTH to a note */
xmnote DLCPTH

/* The second command disabled the DLCPTH warning complete for xrun */
xmnowarn xrun.DLCPTH

/* The third command keeps the DLCPTH warning for xmvlog */
xmwarn xmvlog.DLCPTH

/* The fourth command upgrades DLCPTH to an error, but only for xmsim */
xmerror xmsim.DLCPTH

# After resolving the above commands:
# xrun should not complain about DLCPTH at all
# xmvlog should show DLCPTH as warning
# xmelab should show DLCPTH as note
# xmsim should show DLCPTH as error
```

Related Topic

- Changing the Severity of Messages

Listing the Severity of All Tool Messages

After using the message control file and/or one of the severity control options (`-xmfatal`, `-xmerror`, `-xmwarn`, and `-xmnote`), you may want to verify the log file and check that a tool is, in fact, changing the severity of all output messages (mnemonics). To save tool mnemonics and their severities, use the `-xmout_msgsev` option on the *xrun* command line. This option takes the name of a supported Xcelium tool (like *xrun*) as an argument. For example:

```
% xrun -xmnote BADPRF -xmout_msgsev xrun 2bit_adder.v 2bit_adder_test.v
```

You can specify multiple tool names by separating each name with a colon (`:`). For each specified tool, a separate `<tool>.msgsev` file is saved in the working directory. This file is an ASCII file with a two-column format. One column contains the message severities, and the other contains the mnemonics.

```
TOOL: <header>
<severity> <mnemonic_1>
<severity> <mnemonic_2>
<severity> <mnemonic_3>
...
```

The contents of the `<tool>.msgsev` file changes if:

- You have write permission to the `xcelium.d` directory. In this case, the file shows the effective severity of any updated mnemonics.
- You do not have write permission to `xcelium.d`. In this case, the file shows only the default severity of mnemonics.

Precedence Rules for Message Severity Updates

If you specify multiple severity control options on a single *xrun* command line with `-xmfile_msgcntl` and/or `-xmout_msgsev`, the following precedence rules apply to message severity updates:

Command-Line Control Options	-xmfile_msgctnl	-xmout_msgsev	Updates to logfiles	Updates in *.msgsev
Yes	No	Yes	Yes	Yes
Yes	Yes	Yes	Yes	Yes
No	Yes	Yes	Yes	Yes

If you do not have write permission to the `xcelium.d` directory, the precedence behavior changes as below:

Command-Line Control Options	-xmfile_msgctnl	-xmout_msgsev	Updates to logfiles	Updates in *.msgsev
Yes	No	Yes	Yes	No
Yes	Yes	Yes	Yes	No
No	Yes	Yes	xrun only	No

Related Topics

- Changing the Severity of Messages
- The Message Control File

Compiling Files into Multiple Libraries with XRUN

Use the `-makelib` option when you want to compile different files into different libraries. This option precompiles design units in the specified files into a reference library (or *makelib* collection). If top-level design files are compiled, the reference library is scanned for components instantiated in the design.

The *xrun* command supports `-makelib` with the following file types:

- Verilog
- SystemVerilog
- VHDL
- Verilog-AMS
- C
- C++
- SystemC

The syntax of the `-makelib` option is below:

```
-makelib path_to_library[:logical_name] source_files [-roelab|-rwelab] [-parallel] [-
endlib]
```

The following are some of the common tasks that you can perform using the `-makelib` option:

- Specifying the Source Files to Compile
- Specifying the Reference Library

- Precompiling a Read-Only Library for Elaboration
- Enabling Parallel MAKELIB Compilation
- Using Other Options Within a MAKELIB Collection

Specifying the Source Files to Compile

When building a reference library (or *makelib* collection), the list of files that you want to compile can either be listed individually following the `-makelib` option, or they can be listed in a file specified with the `-f` or `-F` option. For whichever scenario you choose, the list of files in a makelib collection is terminated by:

- A `-end` or `-endlib` option
- Another option that specifies a collection of files to be processed together

  These options are:

  - Another `-makelib` option
  - `-cpost`
  - `-snstage`

For example:

```
% xrun -compile -v93 -makelib mylib file1.vhd file2.vhd -endlib ...
```

Or:

```
% xrun -compile -v93 -makelib mylib -f mylib_files.txt -endlib ...
```

In addition, *xrun* offers limited support for changing the default file extensions of source files specified within a makelib collection. Use:

- `-c_ext` to change the default extension for C source files.
- `-cpp_ext` to change the default extension for C++ source files.
- `-vlog_ext` to change the default extension for Verilog source files.
- `-vhdl_ext` to change the default extension for VHDL source files.

⚠ If you specify any `*_ext` option in a `-makelib` collection other than those listed above, then *xrun* outputs a COLOPT error.

For example:

```
% xrun -makelib libc -c_ext +.iamc foo.iamc -endlib test.v
```

This command specifies a custom source file extension `.iamc` for the C files specified in the makelib collection `libc`.

```
% xrun -makelib libvhdl -vhdl_ext +.myvhdl bar.myvhdl -endlib tb_top.vhd
```

This command specifies a custom source file extension `.myvhdl` for the VHDL files specified in the makelib collection `libvhdl`.

```
% xrun -makelib svlib -sysv_ext .mysv test.mysv -endlib
```

This command uses the `-sysv_ext` option, which is not allowed in a makelib collection. In this case, *xrun* outputs a COLOPT error and halts the simulation.

Referencing a Library with Precompiled Files

After using `-compile` and `-makelib` to precompile files into one or more reference libraries (or makelib collections), you can reference the library using the `-reflib` option.

For instance, consider two source files, `bv_images_p.vhd` and `bv_images_pb.vhd` , that are precompiled into a custom library `./libs/plib` using the following *xrun* command:

```
% xrun -v93 -compile -makelib ./libs/plib bv_images_p.vhd bv_images_pb.vhd -endlib
```

To reference the library, after the design is elaborated, use the `-reflib` option:

```
% xrun -v93 -reflib ./libs/plib inverter.vhd counter_4bit.vhd counter_32bit.vhd
counter_32bit_tb.vhd -top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

The specified reference library `./libs/plib` is added to the list of libraries scanned by *xrun*. This library path is the same path specified to the `-makelib` option.

⚠ When reinvoking the simulation with SimVision, any modifications made to source files within precompiled libraries, or within files compiled with either the `-makelib` or `-reflib` option do not take effect.

Specifying the Reference Library

By default, the logical name of the reference library (or *makelib* collection) is the terminating name in the provided path. If no directory hierarchy is specified, the library is stored inside the `xcelium.d` scratch directory.

For example:

```
% xrun -makelib lib1 file1.vhd file2.vhd -makelib lib2 file3.vhd -endlib file4.vhd
```

In this case:

- `file1.vhd` and `file2.vhd` are compiled into the library `lib1`. The physical path of the library is `xcelium.d/lib1`.
- `file3.vhd` are compiled into the library `lib2`. The physical path of the library is `xcelium.d/lib2`.
- `file4.vhd` are compiled into the default library `worklib`.

If a directory hierarchy is specified, *xrun* creates the directory and uses that directory for the library storage. For example, the following option compiles `file.vhd` into the library `lib1`. The physical path of the library is `./lib1`.

```
% xrun -makelib ./lib1 file.vhd -endlib ...
```

The following option compiles `buf.v` and `and2.v` into the library gates. The physical path of the library is `/usr1/myarea/gates`.

```
% xrun -makelib /usr1/myarea/gates buf.v and2.v -endlib ...
```

⚠ *xrun* generates an error if the root path to the specified directory (`/usr1/myarea` in the example) does not exist.

If you want to create a different logical name for the library, add `:`*`logical_name`* to the end of the path. For example, the following option creates a library called `rtllib`. The physical directory for the library is `/usr1/libs`.

```
-makelib /usr1/libs:rtllib source_files ...
```

The `-makelib` option adds libraries to the search path used when the entire design is being elaborated. If you have libraries defined in a `cds.lib` file, the `-makelib` option adds the libraries at the end of the list defined in the `cds.lib` file.

Precompiling a Read-Only Library for Elaboration

To precompile a read-only library for elaboration, use the `-roelab` option with `-compile` or inisde a makelib collection with `-makelib`. This option changes the default *xrun* behavior by applying a read-only property to all 32-bit and 64-bit pakfiles (on all platforms).

Syntax:

```
% xrun -compile -roelab [other_options] source_files
```

or

```
% xrun -compile -makelib path_to_library [source_files] -roelab -endlib
```

where:

- If you specify `-roelab` outside of a makelib collection, this applies to `worklib`.
- If you specify `-roelab` inside of a makelib collection, this applies to the specified `-makelib` library.
- You can specify `-roelab` to a makelib collection with or without included source files.
- A library marked as read-only stays read-only until it's explicitly reverted back to writable using the `-rwelab` option.

Once marked as read-only, the specified library is not written to by the elaborator; however. the parser can still write VSTs to this read-only library.

For example:

```
% xrun -compile -makelib lib1 lib1.v -roelab -endlib
...
xrun: *N,ROELAB: Library lib1 is read-only for the elaborator.
file: lib1.v
...
```

In this case, a read-only pakfile is output with the name `xm.lnx86.069.pak`, which is saved to `xcelium.d/lib1`. Use `-rwelab` to revert the library back to a default writable mode. As with `-roelab`, you can use `-rwelab` in a makelib collection with or without included source files.

```
% xrun -compile -makelib lib1 -rwelab -endlib
...
xrun: *N,RWELAB: Library lib1 is writable for the elaborator. Previously, it was read-only
for the elaborator.
```

```
% xrun -compile -makelib lib1 -roelab -endlib
...
xrun: *N,ROELAB: Library lib1 is read-only for the elaborator. Previously, it was writable
for the elaborator.
```

⚠ Specifying either `-roelab` or `-rwelab` outside of a makelib collection without `-compile` results in an RO_RW_ELAB_WRK error.

Enabling Parallel MAKELIB Compilation

By default, *xrun* compiles each reference library (or *makelib* collection) serially, or one after another, which can result in a longer scratch compilation for highly complex designs. To cut down on this time commitment, you can choose to compile each makelib collection in parallel by adding the `-parallel` option as shown:

```
% xrun -makelib mylib dff.v -parallel -endlib
```

⚠ The `-parallel` option is valid only when specified inside makelib collections. Specifying this option inside any other collection, or specifying it on the `xrun` command line outside of a makelib collection, results in a COLOPT2 error.

If a mix of both serial and parallel makelib collections is specified on the `xrun` command line,

- Serial makelib collections are compiled first.
- Parallel makelib collections are compiled next.

  Parallel makelib collections can reference libraries that were compiled during the serial compilation step.
- All remaining source files are directly compiled last.

  Any direct-file compilations can reference libraries that were compiled during the serial and parallel compilation steps.

Using Other Options Within a MAKELIB Collection

You can specify other command-line options within a reference library (or *makelib* collection). In general, only options that go to the compilers (*xmvlog*, *xmvhdl*, or *xmsc_run*) are allowed within a makelib collection. These options can be specified following the `-makelib` option, or they can be in a file included with the `-f` or `-F` option. The specified options are applied only to source files within the collection.

ⓘ By default, global options that are specified on the command line outside of a makelib collection are processed first and apply to all `-makelib` options. If you do not want these global options to take priority over the `-makelib` options, then you can choose to change the default processing order with `-mklib_opts_order` or `-set_default_mklib_opts_order`.

The *xrun* command supports the following options within a makelib collection only when using the compatible compiler(s):

Option	Is valid only when using...

	xmvlog	xmvhdl	xmsc_run
`-cds_implicit_tmponly`	✓	✓	
`-debug`			✓
`-default_spice_oomr`	✓		
`-errormax`		✓	
`-ieee1364`	✓		
`-ignore_spice_oomr`	✓		
`-libcell`	✓		
`-liborder`			
`-librescan`	✓		
`-libverbose`			
`-lps_ft_graph`		✓	
`-xmerror`	✓	✓	
`-xmfatal`	✓	✓	
`-xmlibdirname`	✓		
`-xmlibdirpath`	✓		
`-xmshare`	✓		
`-xmuid`	✓		
`-xmvhdl_args`		✓	
`-xmvhdlargs`		✓	
`-xmvlog_args`	✓		
`-xmvlogargs`	✓		
`-neverwarn`	✓	✓	
`-nocopyright`	✓	✓	✓

`-nopragmawarn`	✓	✓	
`-nostdout`	✓	✓	
`-noupdate`	✓	✓	
`-ovmlinedebug`	✓		
`-partialdesign`	✓		
`-pragma`	✓	✓	
`-status`		✓	✓
`-u`	✓		
`-unbuffered`			✓
`-uptodate_messages`	✓	✓	
`-uvmaccess`	✓		
`-uvmlinedebug`	✓		
`-uvmpackagename`	✓		
`-v1995`	✓		
`-v2001`			
`-zlib`	✓	✓	

In addition, *xrun* offers limited support for changing the default file extensions of source files specified within a makelib collection. Use:

- `-c_ext` to change the default extension for C source files.
- `-cpp_ext` to change the default extension for C++ source files.
- `-vlog_ext` to change the default extension for Verilog source files.
- `-vhdl_ext` to change the default extension for VHDL source files.

If you specify any `*_ ext` options in a makelib collection other than those listed above, then *xrun* outputs a COLOPT error. For instance, the following command is invalid:

```
% xrun -makelib svlib -sysv_ext .mysv test.mysv -endlib
```

Examples

The following `xrun` command compiles two files (`bv_images_p.vhd` and `bv_images_pb.vhd`) into a library called `plib` (in the physical directory `/usr/proj/plib`). Design units in the other VHDL files are compiled into the library `worklib`.

For Verilog, *xrun* can automatically detect top-level units in the design. However, *xrun* does not automatically detect top-level VHDL units, and you must specify the top-level VHDL unit with the `-top` or `-vhdltop` option.

```
% xrun -v93 -makelib /usr/proj/plib bv_images_p.vhd bv_images_pb.vhd -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

In the following command, the two files to be compiled into library `plib` are listed in a file called `plib_files.txt`, which is specified with the `-f` option.

```
% xrun -v93 -makelib /usr/proj/plib -f plib_files.txt -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

The following command includes two options within the makelib collection. These options apply only to the files within the collection.

```
% xrun -v93 -makelib /usr/proj/plib -novitalcheck -nobuiltin \
bv_images_p.vhd bv_images_pb.vhd -endlib \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

In the following example, the two VHDL source files are compiled into library `plib` using the `-compile` option. Library `plib` is then referenced with the `-reflib` option. The argument to `-reflib` is the path to the library.

```
% xrun -v93 -compile -makelib /usr/proj/plib bv_images_p.vhd bv_images_pb.vhd
```

```
% xrun -v93 -reflib /usr/proj/plib inverter.vhd counter_4bit.vhd \
counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

The `-v93` option enables VHDL-93 features for the entire design, including the specified library `plib`. If you were to specify a library that does not require VHDL-93 features (for instance, a Verilog design for mixed-language use), then you could compile that library without the `-v93` option and use `-reflib` to specify it instead of `plib`.

If you include a logical name for the library, the logical name must also be included with `-reflib`. For example:

```
% xrun -v93 -compile -makelib /usr/proj/plib:mypack \
bv_images_p.vhd bv_images_pb.vhd
%
% xrun -v93 -reflib usr/proj/plib:mypack \
inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
-top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

If you use the `-c` option to both compile source files and elaborate the design, top-level design files must be included on the command line. For example:

```
% xrun -c -makelib lib1 counter.v clock.v ff.v -endlib top.v
% xrun -c -makelib lib1 counter.vhd clock.vhd ff.vhd -endlib top.v \
  -top worklib.top
```

If you have an existing `cds.lib` file that defines libraries, and have created the physical directories for the libraries, you can use the `-makelib` option to compile files into the libraries. The argument to `-makelib` must match the path in the `cds.lib` file because you cannot have two libraries with the same name. For example, suppose that you have a `cds.lib` file that defines a library called `plib` as follows:

```
# cds.lib
DEFINE plib ./libs/plib
```

To compile files into library `plib`, the argument to `-makelib` must match the path specified in the `cds.lib` file.

```
% xrun -v93 -compile -makelib ./libs/plib bv_images_p.vhd bv_images_pb.vhd
```

Because the `cds.lib` file contains the library reference, it is not necessary to include the `-reflib` option to refer to the library.

```
% xrun -v93 inverter.vhd counter_4bit.vhd counter_32bit.vhd counter_32bit_tb.vhd \
  -top WORKLIB.COUNTER_32BIT_TEST:BENCH
```

Changing the Default MAKELIB Processing Order

When processing a makelib collection, the default Xcelium behavior is to give priority to all *xrun* options specified outside of the collection. The simulator processes and applies these global options first before any options specified after `-makelib`. However, this default behavior may not match the expected behavior for certain designs, and the processing order becomes particularly important for such cases.

To change the Xcelium default behavior, there are two available options:

- `-set_default_mklib_opts_order [local_only | local_then_global | global_then_local]`
- `-mklib_opts_order [local_only | local_then_global | global_then_local]`

You control the processing order by specifying one of the following arguments:

`global_then_local`	The options outside of a `-makelib` are processed before those that are inside of it. This is the default.
`local_then_global`	The options that are present within a `-makelib` are processed before those that are outside of it.
`local_only`	Only those options that are present within a `-makelib` are processed. Global options are ignored.

Use `-set_default_mklib_opts_order` when you want to globally control the default processing order of options specified inside and outside of all makelib collections on a given command line.

Use `-mklib_opts_order` when you want to control the processing order of options specified inside and outside of a particular `-makelib`. Each `-mklib_opts_order` option must be included within an associated `-makelib` to affect any change on that given makelib collection. If both `-set_default_mklib_opts_order` and `-mklib_opts_order` are specified on the same `xrun` command line, `-mklib_opts_order` takes precedence.

Additionally, you can use `-mklib_exclude_option` within an associated `-makelib` to apply a filter on the options specified outside of a given makelib collection. This option uses the following syntax:

```
-mklib_exclude_option "<option_name>"
```

Where the `"<option_name>"` must be an actual minus option enclosed in double quotation marks.

Example

Consider an example where a file `top.sv` instantiates three module instances `a1`, `a2`, and `a3`.

top.sv

```
module top();
  a1 a1_inst();
  a2 a2_inst();
  a3 a3_inst();
endmodule
```

Each module instance is defined in its own separate HDL file, as shown, and includes a basic numbered foo file.

a1.sv

```
`include "foo1.v"
`include "foo11.sv"

module a1();
  foo1 f1();
  foo11 f11();
  parameter p1 = `R1;

  initial begin
    $display("R1 is %d",p1);
  end
endmodule
```

a2.sv

```
`include "foo2.v"
`include "foo22.sv"

module a2();
  foo2 f2();
  foo22 f22();
  parameter pp2 = `R2;

  initial begin
    $display("R2 is %d",pp2);
  end
endmodule
```

a3.sv

```
`include "foo3.v"
`include "foo33.sv"

module a3();
  foo3 f3();
  foo33 f33();
  parameter pp3 = `R3;

  initial begin
    $display("R3 is %d",pp3);
  end
endmodule
```

To run this example, you can specify the following `xrun` command:

```
% xrun top.sv -q -clean -incdir dir0 -define R1=5 -define R2=6 -define R3=7 \
-makelib lib1 a1.sv -incdir dir1 -incdir dir11 -define R1=1 \
-mklib_opts_order local_only -endlib -makelib lib2 a2.sv -incdir dir2 \
-incdir dir22 -define R2=2 -mklib_opts_order local_then_global -endlib \
-makelib lib3 a3.sv -incdir dir3 -incdir dir33 -define R3=3 \
-mklib_opts_order global_then_local -mklib_exclude_option "-incdir" -endlib
...
xcelium> run
R1 is           1
R2 is           2
R3 is           7
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

In the first makelib collection, the `-mklib_opts_order` option sets the `local_order`, so only the options within that makelib collection are used. The simulator searches `dir1` and `dir11` for `` `include `` files but not `dir0`. The value of the `R1` macro is set to `1`.

In the second makelib collection, the `-mklib_opts_order` option sets a `local_then_global` order. The simulator searches `dir2` and `dir22` for `` `include `` files before searching `dir0`. The value of the `R2` macro is set to `2`.

In the third makelib collection, the `-mklib_opts_order` option sets a `global_then_local` order. In this case, `-mklib_exclude_option` also applies a filter on the global `-incdir` option. The simulator searches `dir3` and `dir33` for `` `include `` files but not `dir0` because of the specified filter. However, the global `-define R3=7` specification takes priority over the `-define R3=3` inside the makelib collection.

Direct Invocation

The Xcelium Simulator supports a traditional direct invocation (or three-step) use model, where you compile, elaborate, and simulate your design by running separate tools from the command line. When running a design in direct invocation mode, you must:

- Compile the model first using *xmvlog* or *xmvhdl*.
- Elaborate the design hierarchy used to define the model using *xmelab*.
- Simulate the design by invoking a run on the command line using *xmsim*.

Use the *xrun* utility to simplify the invocation process and specify all input files and options with a single command, instead of invoking multiple tools. For more information, see XRUN Invocation.

See Simulation Command-Line Options for a full description of the `xmsim` command syntax, and for complete definitions of all `xmsim` command-line options.

Related Topics

- Compilation
- Elaboration
- Simulation
- cds.lib Statements and Rules
- The hdl.var Variables

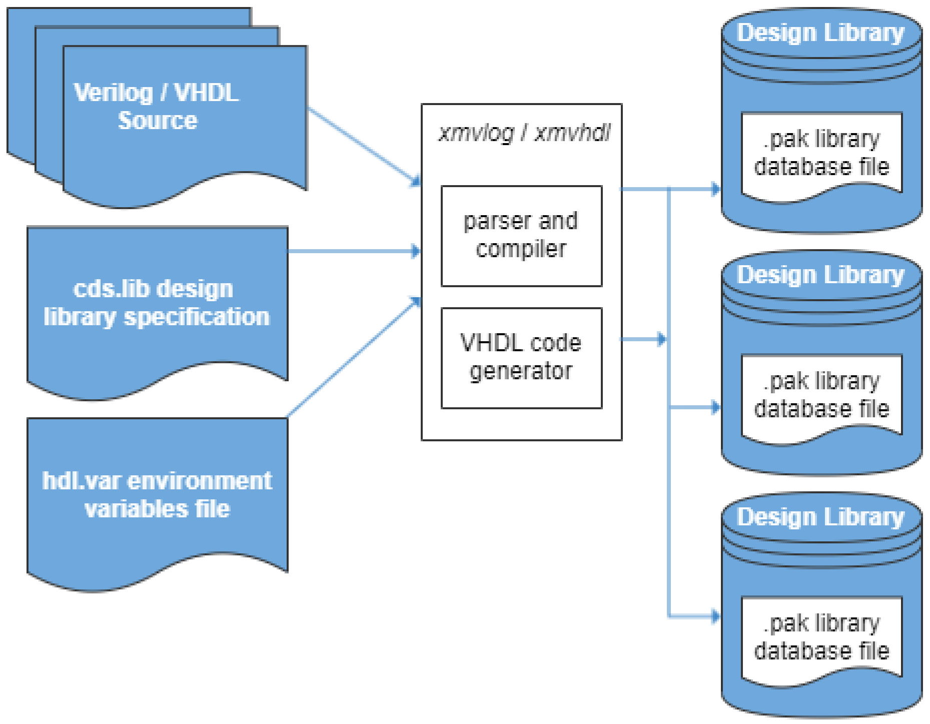
Compilation

Xcelium performs syntactic and static semantic checking on Verilog and VHDL design unit(s). If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. By default, these intermediate objects are stored in a single packed library database file in the work library directory.

- Use the *xmvlog* utility to compile the design unit(s) in Verilog source files (modules, macromodules, or UDPs).
- Use the *xmvhdl* utility to compile the design unit(s) in VHDL source files.

The XMVLOG/XMVHDL Flow

The following figure illustrates the *xmvlog*/*xmvhdl* process flow:


Verilog / VHDL Source
cds.lib design library specification
hdl.var environment variables file
xmvlog / xmvhdl
parser and compiler
VHDL code generator
Design Library
.pak library database file
Design Library
.pak library database file
Design Library
.pak library database file

You can simply invoke a compiler to compile your Verilog or VHDL source files, and that compiler automatically creates a default work library. All design units are compiled into this library. The work library is called `worklib`. This library is saved to a scratch directory called `xcelium.d`, which is under the current directory. In other words, if you have not created `cds.lib` or `hdl.var` files, all design units are compiled into `<`*`current_working_directory`*`>/xcelium.d/worklib`.

Invoking the compiler and letting the tool compile the design units into a default work library automatically is a convenient feature that lets you start using the simulator quickly. However, this method does not provide you with any control over the work library or where it is located. For example, compilation fails if a file contains the following `library` clause, `use` clause, or configuration specification:

```
library lib1;
use lib1.all;
...
for O1:or2
   use entity lib1.or2(or2_a);
```

If you want more control over the libraries into which design units are compiled, you must:

- Create a `cds.lib` file. This file contains statements that define your libraries and that map logical library names to physical directory paths.

- Specify which library is the work library. You can do this by defining variables in an `hdl.var` file, by using command-line options, or by using compiler directives.

Additionally, if you change any of the design units in the hierarchy, you must recompile the parts of the design that have changed and re-elaborate the design hierarchy. You can automatically recompile all out-of-date design units in the hierarchy and re-elaborate the design by:

- Running *xmupdate*. This utility runs *xmvlog* to recompile any changed modules and/or *xmvhdl* to recompile any changed VHDL units (or vice versa). It then runs *xmelab* to re-elaborate the design. The *xmelab* command automatically invokes the *xmsdcf* utility to recompile the SDF source file if it detects a change in the file. The elaborator then generates a new snapshot. Use *xmupdate* when you want to update the snapshot without simulating the complete design.
- Including the `-update` option on the `xmsim` command. This option calls *xmupdate*, which recompiles any changed design units, recompiles the SDF file if necessary, re-elaborates the design, generates a new snapshot, and then invokes the simulator. Use `xmsim -update` if you want to update and then simulate your design.

Compressed Source Files

Compressed files save storage space when managing larger source files. With *xmvlog* and *xmvhdl*, you can use compressed source files on the command line like uncompressed source files. Each compressed file should contain only one source file. The `-y` option also supports compressed files. Archives containing directories with multiple files or multiple directories are not supported.

Currently, the Xcelium compilers recognize the following archive formats:

- Gnu zip compression (`.gz`)
- Standard compression (`.Z`)

Ensure that your compressed source files make use of the following naming convention:

```
<filename>.<hdl_suffix>.<compressed_file_extension>
```

Related Topics

- Compiling with XMVLOG
- Compiling with XMVHDL
- Updating Design Changes at Simulation Time
- *xmupdate*

Compiling with XMVLOG

To compile design units in Verilog source files with *xmvlog*, you can:

- Specify a list of source files on the command line. For example:

  ```
  % xmvlog src1.v src2.v src3.v
  ```

- Specify a list of compressed source files on the command line. For example:

  ```
  % xmvlog src1.v.gz src2.v.gz src3.v.gz
  ```

- Compile the files using design-top compilation.

  Use the *xmparse* utility for design-top compilation.

  With design-top compilation, you specify the top-level of the design instead of specifying all source files on the command line. The top-level is specified by defining the `DESIGN_TOP` variable in a file called a compilation command file, which is passed as an argument to the `xmparse -cmdfile` option.

  The compilation command file also contains variables that define the list of directories to be searched for locating the design files, and the path to a naming rules file that describes how design unit names map to the names of the source files containing their definitions.

  ```
  % xmparse -messages -cmdfile comp_file.txt
  ```

  For a pure Verilog design, you can also compile source files using design-top compilation by specifying the compilation command file on the `xmvlog` command line with the `-cmdfile` option. For example:

  ```
  % xmvlog -messages -cmdfile comp_file.txt
  ```

See Compiling Files by Specifying the Design-Top Level for more information.

- Specify a design unit name with the `-unit` option. For example:

  ```
  % xmvlog -unit worklib.mymod
  ```

  Use the `-unit` option to recompile a specific design unit that has been compiled previously and that you have subsequently edited. The argument to the -unit option is the name of the design unit. You cannot specify a source file name.

- Specify a design unit with the `-specificunit` option and a source file name. For example:

  ```
  % xmvlog -specificunit worklib.arith alu.v
  ```

  Use the `-specificunit` option to compile one design unit in a source file that contains multiple design units.

Arguments to the `xmvlog` command can occur in any order, except that parameters to options must immediately follow the option that they modify.

For each command-line argument that is not an option, or a parameter to an option, Xcelium will treat it as a filename when running this command. For each filename, *xmvlog* first tries to open the file as specified. If this fails, each file extension that is specified with the `VERILOG_SUFFIX` variable is appended to the name, and *xmvlog* tries to open the file. The default file extension is `.v`. If no match is found, *xmvlog* tries the list of possible suffixes in the `hdl.var` variable `VIEW_MAP`. If all suffixes are exhausted, *xmvlog* generates an error.

Compilation of Design Units into Library.Cell:View

After you run the *xmvlog* compiler, HDL design units—modules, macromodules, and user-defined primitives (UDPs)—are compiled into a *`library.cell:view`* format. For example, a Verilog module called `counter` might be compiled into:

```
worklib.counter:module
```

In this case:

- The *`library`* name (`worklib`) is the logical name of the library into which the Verilog/VHDL design unit has been compiled. This library is called the *work* library.
- The *`cell`* is the name of the design unit. For Verilog, a design unit can be a module, macromodule, or UDP, and the cell is always set to the name of the design unit.

- The *`view`* is the name associated with a version of a cell. Views can be used to delineate between representations (schematic, VHDL, Verilog), abstraction levels (behavior, RTL, post-synthesis), status (experimental, released, golden), and so on. For instance, you might have one view that is the RTL representation of a particular unit and another view that is the behavioral representation, or you might have two different versions of a cell, one with timing and one without timing.

  Alternatively, you can let the compiler use predefined default view names. For *xmvlog*, the predefined view names are `module` for a module or macromodule and `udp` for a UDP. For *xrun*, the predefined view name for a module or macromodule is the file extension of the file that contains the definition. For example, if you compile a module in file `file.v`, the default view name is `v`. If you compile a module `file.vlib`, the default view name is `vlib`. A compiled module in `file.sv` has the default view name of `sv`.

  > The *xrun* command uses the file extensions of specified input files to determine their file type. Each recognized file type has a built-in, predefined set of file extensions. For example, the default file extensions for Verilog files are `.v`, `.vp`, `.vs`, `.V`, `.VP`, and `.VS`. For each file type, there is a command-line option that you can use to change, or add to, the list of file extensions mapped to a given language. For example, if you have Verilog files with extensions `.rtl` and `.vg`, you can specify that these are valid extensions for Verilog files with the `-vlog_ext` option. After changing the list of valid extensions, or adding the new extensions to the list of predefined extensions, a compiled module in a file called `file.vg` will have a view name of `vg`. See XRUN Utility File Types and Extensions for more information.

  Or you can specify the view name by defining variables in an `hdl.var` file, using command-line options, or using the `` `view `` compiler directive.

Related Topics

- The hdl.var Variables

Using the -libmap Option

The `-libmap` command-line option specifies a library map file. This file can contain:

- Library declarations
  A library declaration associates a logical library name with a source file or set of source files. The design units in specified source files are compiled into the associated library. For example:

```
library rtlLib *.v;
library gateLib *.vg;
```

All libraries in a library mapping file must be declared in the `cds.lib` file.

- References to other library map files
  An `include` statement can be used to include the contents of a library map file in another library map file.

The `xmvlog -libmap` option can be used to specify a library map file that contains library declarations. When specified, this option takes precedence over any `LIB_MAP` variable specification in the `hdl.var` file.

⚠ The elaborator looks up for cell definitions in the default work library, `worklib`, even when this library is not present in the library map provided by the user. This works only when the `-WORK` flag is not used with elaborator, which specifies a WORK library.

Example:

Below, a library map file called `lib.map` contains certain library declarations and a configuration block:

```
library rtlLib "top.v";
library aLib "adder.v";
library gateLib "adder.vg";
config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib;
endconfig
```

When you compile the source files with the `-libmap` option, the library declarations are parsed and analyzed, and the declarations are used to determine the compilation of design units into libraries. Configurations specified in the file, if any, are checked for syntax only.

In this example:

- Design units in file `top.v` are compiled into the library called `rtlLib`.
- Design units in file `adder.v` are compiled into the library called `aLib`.
- Design units in file `adder.vg` are compiled into the library called `gateLib`.

```
% xmvlog -nocopyright -messages -libmap lib.map top.v adder.v adder.vg
file: top.v
        module rtlLib.top
                errors: 0, warnings: 0
        module rtlLib.foo
                errors: 0, warnings: 0
```

```
file: adder.v
        module aLib.adder
                errors: 0, warnings: 0
        module aLib.foo
                errors: 0, warnings: 0
file: adder.vg
        module gateLib.adder
                errors: 0, warnings: 0
        module gateLib.foo
                errors: 0, warnings: 0
```

Using the -work and -view Options

The `-work` and `-view` command-line options take precedence over any `WORK` and `VIEW` variable definitions in the `hdl.var` file.

- The argument to `-work` is a library name.
- The argument to `-view` is a view name.

Example:

Below, the `-work` and `-view` options are used to override the variable definitions in an `hdl.var` file. The `hdl.var` file is as shown:

```
DEFINE XMVLOGOPTS -messages
DEFINE XMELABOPTS -messages
DEFINE XMSIMOPTS  -messages

DEFINE LIB_MAP (/test/xcelium/board => designlib, \
                                  + => worklib)

DEFINE VIEW_MAP (.v    => behav, \
                 .rtl  => rtl, \
                 +     => module)
```

All source files have a `.v` extension, and are in the `board/` directory.

```
;# Compile all .v source files.
;# Use -work to compile the design units into the library called testlib.
;# Use -view to specify a view name of gate for all design units.
% xmvlog -nocopyright -work testlib -view gate *.v
file: board.v
    module testlib.board:gate
        errors: 0, warnings: 0
file: clock.v
    module testlib.m555:gate
```

```
        errors: 0, warnings: 0
file: counter.v
    module testlib.m16:gate
        errors: 0, warnings: 0
file: ff.v
    module testlib.dEdgeFF:gate
        errors: 0, warnings: 0
```

Using the -specificunit Option

The `-specificunit` command-line option can compile one design unit from a source file that contains multiple design units. This option takes a `[`*`library`*`.]`*`cell`*`[:`*`view`*`]` specification as its argument. If you:

- Specify the design unit by itself, the compiler uses default values for the library and view names.
- Specify the design unit and a library name, the design unit is compiled into that library.
- Specify the design unit and a view name, the design unit is given that specified view.

Example:

The following example uses the `-specificunit` option to compile one design unit in a file that contains multiple units. The `hdl.var` file is as shown:

```
DEFINE XMVLOGOPTS -messages
DEFINE XMELABOPTS -messages
DEFINE XMSIMOPTS -messages

DEFINE LIB_MAP (/test/xcelium/alu => worklib, \
                                + => designlib)
DEFINE VIEW_MAP (.v => behav, .rtl => rtl, .g => gate, + => module)
```

Using the definitions in this file, source files with a `.v` extension in the `alu/` directory are compiled into `worklib` with a view name of `behav`.

```
;# Use -specificunit to compile only the design unit called arith.
;# Module arith is compiled into worklib.arith:behav
% xmvlog -nocopyright -specificunit arith 16bit_alu.v
file: 16bit_alu.v
    module worklib.arith:behav
        errors: 0, warnings: 0

;# Compile only the design unit called arith. Specify the library in the
;# argument to -specificunit and a new view name of rtl.
;# Module arith is compiled into designlib.arith:rtl.
% xmvlog -nocopyright -specificunit designlib.arith:rtl 16bit_alu.v
file: 16bit_alu.v
```

```
    module designlib.arith:rtl
        errors: 0, warnings: 0
```

Conditionally Compiling Source Code

Use the conditional compilation compiler directives (`` `ifdef``, `` `else``, and `` `endif``) to conditionally include lines of a Verilog HDL source description during compilation. The `` `ifdef`` compiler directive checks whether a variable name is defined either in the source code or on the command line. If the variable name is defined, the compiler includes the lines in the source description.

There are two ways to define `` `ifdef`` variables:

- Use the `` `define`` compiler directive. For example:

  ```
  `define debug
  ```

  This defines the `debug` text macro, which can then be used repetitively throughout your design. If this defined macro is also assigned a value, the tool assigns that value whenever it encounters the defined macro. These defined text macros are especially useful for constant values.

  > ⓘ Defined text macros have a limit of 2048 characters.

- Use the `-define` option on the *xrun* or *xmvlog* command line.

  To control conditional compilation, define a variable name as an empty text macro. The syntax is:

  ```
  -define text_macro_name
  ```

  For example,

  ```
  -define debug
  -define sun3
  ```

  You can also have multiple `-define` arguments on a single command line as shown below:

  ```
  -define sun4 -define structural
  ```

If you define the same macro name differently using a `` `define`` compiler directive and a `-define` command-line option, the command-line option overrides the compiler directive.

The compiler does not check the syntax for any ignored group of lines. However, even though *xrun/xmvlog* does not check the syntax of this text, it must conform to the lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Example 1

The following example shows you how to define a macro on the command line with the `-define` option. The source code is as follows:

```
module test;
  initial
    begin
   `ifdef debug     // If debug is defined, execute the following line.
     $display("debug is defined.");
   `else            // If debug is not defined, execute the following line.
     $display("debug is not defined.");
   `endif
    end
endmodule
```

Direct Invocation Mode:

```
% xmvlog -nocopyright test.v    ;# Compile with xmvlog. Macro debug is not
                                ;# defined in the model or on the command line.
% xmelab -nocopyright worklib.test      ;# Elaborate with xmelab.
% xmsim -nocopyright test               ;# Invoke the simulator.
xcelium> run                    ;# Model displays "debug is not defined."
debug is not defined
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
% xmvlog -nocopyright -define debug test.v     ;# Compile with xmvlog.
                                ;# Macro debug is defined on the command line.
% xmelab -nocopyright worklib.test
% xmsim -nocopyright test
xcelium> run               ;# Run the simulation. Model displays "debug is defined."
debug is defined
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

XRUN Invocation Mode:

```
% xrun -nocopyright -q -define debug test.v   ;# Macro debug is defined on the
                                              ;# command line.
Top level design units:
        test
xcelium> run
debug is defined.
xmsim: *W,RNQUIE: Simulation is complete.
xcelium> exit
```

Example 2

This example shows how to use the `` `define `` compiler directive to specify a user-defined macro that includes the `$display` system task, The following code defines the macro as a string and uses the `%s` format specifier:

```
`define teststring "THIS_IS_A_STRING"
module foo();
  initial
  $display("teststring = %s", `teststring);
endmodule
```

```
% xrun -nocopyright display_macro.v
file: correct.v
        module worklib.foo:v
                errors: 0, warnings: 0
...
...
Loading snapshot worklib.foo:v .................... Done
xcelium> run
teststring = THIS_IS_A_STRING
...
```

Built-In Conditional Compilation Text Macros

There are two conditional compilation text macros that, if used in your source Verilog HDL, are always defined. The text macros are:

- `XCELIUM`

  The tool parses any code enclosed within `` `ifdef XCELIUM`` and `` `endif``, and the code runs in the simulator.

- `CDS_TOOL_DEFINE`

  The tool parses any code enclosed within `` `ifdef CDS_TOOL_DEFINE`` and `` `endif``, and the code runs in any Cadence tool that uses this HDL parser.

  ⚠ Some Cadence tools may not implement this text macro.

  For example:

```
module test();
...
...

  `ifdef CDS_TOOL_DEFINE
  // This code is used by all Cadence tools that use the
  // Verilog HDL parser, and is ignored by all other tools.
  // It is not necessary to use -define CDS_TOOL_DEFINE on the command line.
    ...
    ...
  `endif

  // Other HDL code
  ...
  ...

endmodule
```

Defining Macros on the Command Line

When specifying the `-define` option to define a macro on the *xrun* or *xmvlog* command line, you can choose to:

- Define a variable name as an empty text macro. For example:

  `-define structural`

  Empty text macros are used to define variable names to control conditional compilation. See Conditionally Compiling Source Code.

- Define a macro name as a string. The `-define` option has the following syntax when it defines a macro name as a string:

  `-define` *`macro_name`* `=` *`macro_string`*

  For example, the following option defines the macro name `gate` as the string `or`:

  `-define gate=or`

  Enclose the *`macro_string`* in quotation marks if it includes characters that you explicitly want as part of the input:

  `-define foo "16'h03"`

You can define only one macro with each `-define` on the command line, but the number of `-define` options on the command is unlimited. The option cannot define macros of more than one line.

If you define the same macro name differently with a `` `define `` compiler directive and a command line `-define` option, the command-line option overrides the compiler directive.

Example

After compiling the code in the following example, the `` `define `` compiler directive defines the macro `gate` as an AND gate. The resulting model is then elaborated with *xmelab*. After simulating the elaborated snapshot with *xmsim*, the value of `c` changes from `x` to `0` to `1`.

% more def.v

```
module def();
`define gate and
reg a, b;
`gate (c, a, b);
initial
    begin
        #1;
        a = 0;
        b = 1;
        $display("a=", a, " b=", b, " c=", c);
        #5;
        $display("a=", a, " b=", b, " c=", c);
        a = 1;
        #5;
        $display("a=", a, " b=", b, " c=", c);
        $finish;
    end
endmodule
```

```
% xmvlog -nocopyright def.v
% xmelab -nocopyright def
% xmsim -nocopyright def
xcelium> run
a=x   b=x   c=x
a=0   b=1   c=0
a=1   b=1   c=1
Simulation complete via $finish(1) at time 11 NS + 0
./def.v:17 $finish;
xcelium> exit
```

```
:# The gate is simulated using the command-line definition.
% xmvlog -nocopyright -define gate=or def.v
`define gate and
                |
xmvlog: *W,MACNDF (def.v,2|16): text macro `gate' not redefined
```

```
        using command line definition.
% xmelab -nocopyright def
% xmsim -nocopyright def
xcelium> run
a=x   b=x   c=x
a=0   b=1   c=1
a=1   b=1   c=1
Simulation complete via $finish(1) at time 11 NS + 0
./test.v:16 $finish;
xcelium> exit
```

Compiling with XMVHDL

There are several ways to compile the design units in VHDL source files with *xmvhdl*:

- Specify a list of source files on the command line. For example:
  ```
  % xmvhdl -messages src1.vhd src2.vhd src3.vhd
  ```
- Specify a list of compressed source files on the command line. For example:
  ```
  % xmvhdl -messages src1.vhd.gz src2.vhd.gz src3.vhd.gz
  ```
- Compile the files using design-top compilation.
  Use the *xmparse* utility for design-top compilation.

  With design-top compilation, you specify the top-level of the design instead of specifying all source files on the command line. The top-level is specified by defining the `DESIGN_TOP` variable in a file called a *compilation command file*, which is passed as an argument to the `xmparse -cmdfile` option.

  The compilation command file also contains variables that define the list of directories to be searched for locating the design files, and the path to a naming rules file that describes how design unit names map to the names of the source files containing their definitions.

  ```
  % xmparse -messages -cmdfile comp_file.txt
  ```

  For a pure VHDL design, you can also compile source files using design-top compilation by specifying the compilation command file on the xmvhdl command line with the `-cmdfile` option. For example:

  ```
  % xmvhdl -messages -cmdfile comp_file.txt
  ```

See Compiling Files by Specifying the Design-Top Level for more information.

- Specify a design unit name with the `-unit` option. For example:

  ```
  % xmvhdl -messages -unit work.arith:arch
  ```

  Use the `-unit` option to recompile a specific design unit that has been compiled previously and that you have subsequently edited. The argument to the `-unit` option is the name of the design unit. You cannot specify a source file name.

- Specify a design unit with the `-specificunit` option and a source file name. For example:

  ```
  % xmvhdl -specificunit arith alu.vhd
  ```

  Use the `-specificunit` option to compile one design unit in a source file that contains multiple design units.

The arguments to the `xmvhdl` command can occur in any order except that parameters to options must immediately follow the option they modify.

xmvhdl treats each command-line argument that is not an option or a parameter to an option as a filename. For each filename, it first tries to open the file as specified. If this fails, each file extension specified with the `VHDL_SUFFIX` variable in the `hdl.var` is appended to the name and *xmvhdl* tries to open the file. The default file extensions are `.vhd` and `.vhdl`. If all suffixes are exhausted, *xmvhdl* generates an error. See `VHDL_SUFFIX` for more information on this variable.

After you run the *xmvhdl* compiler, your HDL design units are compiled into a *`library.cell:view`* format. For example, a VHDL module called `test_adder` might be compiled into:

```
worklib.test_adder:entity
```

In this case:

- The *`library`* name (`worklib`) is the logical name of the library into which the Verilog/VHDL design unit has been compiled. This library is called the *work* library.
- The *`cell`* and *`view`* names depend on the kind of VHDL design unit parsed.

Design Unit	Cell Name	View Name
Entity	Entity name	`entity`
Architecture	Corresponding entity name	Architecture name
Package	Package name	`package`
Package body	Corresponding package name	`body`

Configuration	Configuration name	`configuration`

For example, if the work library is `work`, and you compile a source file that contains an entity called `drink_machine`, the entity is compiled into `work.drink_machine:entity`.

If the source file contains an architecture called `rtl` for entity `drink_machine`, the architecture is compiled into `work.drink_machine:rtl`.

Using the -smartorder Option

By default, source files that you specify as input to the *xmvhdl* compiler are analyzed in the order in which they are listed on the command line, and the design units in a particular source file are analyzed in the order in which they appear in the file. This means that the design units within a source file, and the source files listed on the command line, must be ordered so that there are no unresolved dependencies for any of the design units in a design file when that file is being compiled. You can use the `-smartorder` command-line option to compile design files without having to establish a compile order beforehand. For example:

```
% xmvhdl -smartorder file1.vhd file2.vhd file3.vhd
```

The above command first examines the design files to identify any dependencies, establishes the correct compilation order, and then analyzes the design units. All design units in the specified files are then compiled into the work library (specified with the `WORK` variable or the `-work` command-line option).

⚠ Order-independent compilation using the `-smartorder` option is intended to be used for compiling design files into a single work library. Design library (reference library) files must be precompiled into their respective library locations before the analysis of the design files. Design files and the files comprising the various libraries cannot be compiled into different locations with the same `xmvhdl` call.

If you are compiling by specifying a compilation command file (design-top compilation), the `-smartorder` option is ignored.

Using the -smartlib Option

To compile design files without having to establish a compile order beforehand, use the `-smartorder` option. However, be aware that this order-independent compilation has a major limitation: Design library (reference library) files must be precompiled into their respective library locations before the analysis of the design files. Design files and the files comprising the various libraries cannot be compiled into different locations with the same `xmvhdl` call.

The `-smartlib` command-line option:

- Enables order-independent compilation (that is, it turns on the `-smartorder` option).
- Enables the compilation of different design files into different libraries.

The `-smartlib` option uses the library information for design units of a design file to determine the library into which the design file needs to be compiled. If it is possible to uniquely determine the library into which a design unit is to be compiled, the design file is compiled into that library. If it is not possible to uniquely determine the library, the design file is compiled into the work library (specified with the `WORK` variable or the `-work` command-line option).

For example, the following file `file1.vhd` has a dependency on a package `pack1`, which is defined in `file2.vhd`.

```
-- file1.vhd                                -- file2.vhd
library ieee;                               library ieee;
use ieee.std_logic_1164.all;                use ieee.std_logic_1164.all;

library testlib;                            package pack1 is
use testlib.pack1.all;                        function ...
                                            end package pack1;
entity E is
  ...                                       package body pack1 is
end E;                                        function ...
                                            end package body pack1;
architecture ARCH of E is
begin
  ...
end ARCH;
```

If you only use the `-smartorder` option, *xmvhdl* identifies this dependency and compiles `file2.vhd` first. The file is compiled into the work library (`worklib` in this example). However, the `use` clause in `file1.vhd` references package `pack1` in library `testlib`. Compilation of `file1.vhd` fails because design unit `pack1` cannot be found in library `testlib`.

In the example shown above, `file1.vhd` contains the following `use` clause:

```
use testlib.pack1.all;
```

This `use` clause indicates that package `pack1` is in the library `testlib`.

To correctly compile `file2.vhd` into library `testlib`, use `-smartorder` together with the `-smartlib` option as below:

```
% xmvhdl -smartorder -smartlib file1.vhd file2.vhd
```

If it is not possible to uniquely determine the library, the file is instead compiled into the work library. In this example, since there is no indication for the library into which `file1.vhd` is to be compiled, it is compiled into the work library.

Compiling Files by Specifying the Design-Top Level

With Xcelium, you can choose to compile a design by specifying the top-level design unit when you invoke *xmvlog* or *xmvhdl* instead of specifying each and every source file on the command line. The top-level design unit is specified in a file called the *compilation command file*, which is passed as an argument to the `xmparse -cmdfile` option. The compilation command file also specifies the path to a file that contains the rules that describe the mapping of design unit names to source file names, and the list of directories to be searched for locating the design files.

✓ You can use design-top compilation for compiling Verilog, VHDL, or mixed Verilog-VHDL designs.

To compile your design files using design-top compilation, you must:

1. Write a compilation command file.
   A compilation command file is required. The path to this file must be specified as the argument to the `-cmdfile` option.

   See Writing a Compilation Command File.
2. Write a naming rules file.
   The naming rules file contains the naming rules that describe how design unit names map to the names of the source files containing their definitions.

   See Name Rule Syntax.

   If a rules file is not used, a default set of naming rules is used. The default naming rules are shown in Default Naming Rules.
3. Specify the library/libraries into which you want to compile the design units.
   You can specify the library mapping by using the `-work` command-line option, by defining the WORK variable in the `hdl.var` file, or by defining the `LIB_MAP` variable in the `hdl.var` file.

   See Specifying the Library Mapping.
4. Use the `-cmdfile` *`compilation_command_file`* option.
   This `xmparse` command-line option specifies that design-top based compilation is to be used, and the argument provides the path to the compilation command file. The syntax is:

```
% xmparse -cmdfile compilation_command_file [other_xmparse_options]
```

See *xmparse* for details on this utility.

⚠ Using the `-cmdfile` option specifies that design-top compilation is to be used. You cannot specify design file names on the command line. If you use the `-cmdfile` option and also include design file names on the command line, the parser uses the compilation command file.

For a pure Verilog design, you can compile files using design-top compilation by using the `-cmdfile` option on the `xmvlog` command line.

```
% xmvlog -cmdfile compilation_command_file [other_options] ...
```

For a pure VHDL design, you can use the `-cmdfile` option on the `xmvhdl` command line.

```
% xmvhdl -cmdfile compilation_command_file [other_options] ...
```

Writing a Compilation Command File

A compilation command file contains the following definitions:

- The design top
  You specify the design top by defining the `DESIGN_TOP` variable.
  - The definition can be one of the following for Verilog:
    - The name of the top-level module for the design
      DEFINE DESIGN_TOP topmodule
    - The name of the design file that contains the top-level design unit
      DEFINE DESIGN_TOP (design.v)
  - The definition can be one of the following for VHDL:
    - The top-level entity:architecture pair
      `DEFINE DESIGN_TOP (top:arch)`
    - The top-level entity
      `DEFINE DESIGN_TOP (top:)`
    - A configuration
      `DEFINE DESIGN_TOP (conf1)`
    - The name of the design file that contains the top-level design unit
      `DEFINE DESIGN_TOP (design.vhd)`

⚠ You can use the `-design_top` option on the `xmparse` command line to specify the design top. The argument to the `-design_top` option overrides the definition of the `DESIGN_TOP` variable in the compilation command file.

- The path to a naming rules file
  By default, the parser uses a set of default naming rules to describe how design unit names map to the names of the source files containing their definitions. The default naming rules are shown in Default Naming Rules.

  You can write your own naming rules file to describe your naming conventions. See Name Rule Syntax for details on writing a naming rules file.

  If you use a naming rules file, you must specify the path to the naming rules file by defining the `RULES_FILE` variable in the compilation command file.

  ```
  DEFINE RULES_FILE (rules_file_path)
  ```

  The rules file path can be absolute or relative.
- The list of directories to be searched for locating the design files
  You specify the directories to search for design files by defining the `SEARCH_PATH` variable. The syntax is as follows:

  ```
  DEFINE SEARCH_PATH ( directory_path[, directory_path ...] )
  ```

  The paths can be absolute paths or relative paths. Environment variables can be used in the paths.

  The parser searches each directory in the list, in the order that they appear, for the file name until a match is found. The search stops as soon as a match is found. If a path is not found, or if a path does not have executable rights, the path is skipped with a warning message.

  Example:

  ```
  DEFINE SEARCH_PATH ( /usr/user1/project/dir1, \
                       /usr/user1/project/dir2, \
                       /usr/user1/project/dir3 )
  ```
- The `REDUMP_ON_UPDATE` variable
  This value of this variable (`ON` or `OFF`) controls whether a source file whose contents have not

changed is recompiled when updating with the *xmupdate* utility or an `xmvlog -update`, `xmvhdl -update`, `xmelab -update`, or `xmsim -update` command.

The default value of the `REDUMP_ON_UPDATE` variable is `ON`.
See Updating Design Changes at Simulation Time for more information.

- The `VLOG_ARGS` and `VHDL_ARGS` variables
  These variables define the command-line options for the *xmvlog* and *xmvhdl* parsers. For example:

  ```
  DEFINE VLOG_ARGS (-messages -ieee1364)
  DEFINE VHDL_ARGS (-messages -v93)
  ```

  You cannot include parser options on the `xmparse` command line.

Example Compilation Command File

The following is an example compilation command file.

```
DEFINE DESIGN_TOP conftop
DEFINE SEARCH_PATH (./src1, ./src2, ./src3, ./top)
DEFINE RULES_FILE (./rules/name_rules.rules)
DEFINE REDUMP_ON_UPDATE (off)
DEFINE VLOG_ARGS (-messages -ieee1364)
DEFINE VHDL_ARGS (-messages -v93)
```

Name Rule Syntax

A naming rules file contains the naming rules that describe how design unit names map to the names of the source files containing their definitions. You can specify one or more name rules for each design unit type. The format for a name rule is:

```
(DESIGN_UNIT_TYPE (NAMING_RULE) [(NAMING_RULE) ...])
```

`DESIGN_UNIT_TYPE` is one of the following:

- `ENTITY`
- `ARCH`
- `PACKAGE`
- `PACKAGEBODY`
- `CONFIG`

- `MODULE`
- `UDP`

A *`NAMING_RULE`* takes the following form:

```
([str]<DESIGN_UNIT_VARIABLE>[str])
```

where [str] can be any string that acts as prefix or suffix, and *`<DESIGN_UNIT_VARIABLE>`* is the name of the current design unit. The *`<DESIGN_UNIT_VARIABLE>`* can be:

- `<ENTITY>`

  This variable can be used in naming rules for design unit types `ENTITY`, `ARCH`, and `CONFIG`.
- `<ARCH>`

  This variable can be used in naming rules for design unit type `ARCH`.
- `<PACKAGE>`

  This variable can be used in naming rules for design unit type `PACKAGE` and `PACKAGEBODY`.
- `<CONFIG>`

  This variable can be used in naming rules for design unit type `CONFIG`.
- `<MODULE>`

  This variable can be used in naming rules for design unit type `MODULE`.
- `<UDP>`

  This variable can be used in naming rules for design unit type `MODULE` and `UDP`.

The following is an example naming rules file:

```
(ENTITY (e_<ENTITY>))
(ARCH (A_<ENTITY>_<ARCH>) (a_<ARCH>))
(PACKAGE (p_<PACKAGE>)
(PACKAGEBODY (p_<PACKAGE>_body)(<PACKAGE>body))
(CONFIG (c_<CONFIG>))
(MODULE (m_<MODULE>))
(UDP (u_<UDP>))
```

The following table shows example naming rules for each design unit type.

In the examples shown in the following table, it is assumed that all source file names are in lowercase and that the file extension for all source files is `.v` for Verilog and `.vhd` for VHDL. See Source File Names and File Extensions for more information.

Design Unit Type	Comments	Examples

`ENTITY`	A naming rule for design unit type `ENTITY` can take only variable `<ENTITY>`.	An entity `foo` is in file `foo.vhd`.
		`(ENTITY (<ENTITY>))`
		
		An entity `foo` is in file `e_foo.vhd`.
		`(ENTITY (e_<ENTITY>))`
		
		An entity `foo` is in file `e_foo_ver1.vhd`.
		`(ENTITY(e_<ENTITY>_ver1))`
		
		An entity `foo` is in file `foo.entity.vhd`.
		`(ENTITY(<ENTITY>.entity))`

`ARCH`	A naming rule for design unit type `ARCH` can take variables `<ENTITY>` and/or `<ARCH>`.	Architecture `rtl` is in file `a_rtl.vhd`.
		`(ARCH(a_<ARCH>))`
		
		Architecture `rtl` of entity `foo` is in file `a_foo.vhd`.
		`(ARCH(a_<ENTITY>))`
		
		Architecture `rtl` of entity `foo` is in file `e_foo_rtl_arch.vhd`.
		`(ARCH(e_<ENTITY>_<ARCH>_arch))`
		
		Architecture `rtl` of entity `foo` is in file `rtl.foo.arch.vhd`.
		`(ARCH(<ARCH>.<ENTITY>.arch))`
		
		⚠ If a top-level configuration is not specified, and the rules file contains the rule `(ARCH(<ENTITY>.<ARCH>))` the `ARCH` name (`<ARCH>`) is not known ahead of time. *xmparse* searches for an architecture in a file `ENTITY.*`. This is a wildcard search, and the wrong files could be selected.

`PACKAGE`	A naming rule for design unit type `PACKAGE` can take only variable `<PACKAGE>`.	A package `pack` is in file `pack.vhd`. `(PACKAGE(<PACKAGE>))` A package `pack` is in file `p_pack.vhd`. `(PACKAGE(p_<PACKAGE>))`
`PACKAGEBODY`	A naming rule for design unit type `PACKAGEBODY` can take only variable `<PACKAGE>`.	A package body for a package `pack` is in file `p_pack_body.vhd`. `(PACKAGEBODY(p_<PACKAGE>_body))` A package body is in file `pack.body.vhd`. `(PACKAGEBODY(<PACKAGE>.body))`
`CONFIG`	A naming rule for design unit type `CONFIG` can take variables `<CONFIG>` and/or `<ENTITY>`.	A configuration `conftop` of entity `top` is in file `conftop.vhd`. `(CONFIG(<CONFIG>))` A configuration `conftop` of entity `top` is in file `c_conftop_top.vhd`. `(CONFIG(c_<CONFIG>_<ENTITY>))`
`MODULE`	A naming rule for design unit type `MODULE` can take variables `<MODULE>` and/or `<UDP>`.	A module `foo` is in file `foo.v`. `(MODULE (<MODULE>))` A module `foo` is in file `m_foo.v`. `(MODULE (m_<MODULE>))` A module `foo` is in `m_foo_ver1.v`. `(MODULE(m_<MODULE>_ver1))`
`UDP`	A naming rule for design unit type `UDP` can take only variable `<UDP>`.	A UDP `latch` is in file `latch.v`. `(UDP(<UDP>))` A UDP `latch` is in file `u_latch.v`. `(UDP(u_<UDP>))` A UDP `latch` is in file `latch_udp.v`. (UDP(<UDP>_udp)) A UDP `latch` is in file `u_latch_ver1.v`. `(UDP(u_<UDP>_ver1))`

Multiple rules are allowed for a design unit type. For example:

```
(CONFIG(<CONFIG>)(c_<CONFIG>_<ENTITY>))
(MODULE (<MODULE>) (m_<MODULE>))
```

For VHDL secondary units (architecture and package body), the parser first checks to see if the unit is defined in the source file that defines the corresponding primary unit (entity or package) before proceeding to search according to the naming rules. For example, suppose that source files contain the following design units:

Source File	Contains...
`e_fa_beh.vhd`	entity `fa_beh`
`a_fa_beh.vhd`	architecture `fa_beh_a` of entity `fa_beh`
`e_ha_beh.vhd`	entity `ha_beh` architecture `ha_beh_a` of entity `ha_beh`

When determining the source file for architecture `ha_beh_a`, the parser first checks if this secondary design unit is defined in the same file as the corresponding entity (`ha_beh`). Because it is, the parser associates the filename `e_ha_beh.vhd` with both design units. However, after determining that architecture `fa_beh_a` is not described in the same file as the corresponding entity (`fa_beh`), the parser must use a naming rule to determine the source file name. Therefore, in this example, the following two naming rules would be required:

```
(ENTITY(e_<ENTITY>))
(ARCH(a_<ENTITY>))
```

Source File Names and File Extensions

All source filenames in the design are assumed to be lowercase, except for cases where an escaped name has been used for the design unit.

For example, if the design contains two entities called `ent1` and `ENT2`, and the naming rule for entities is:

```
(ENTITY (e_<ENTITY>))
```

The parser searches for source files called `e_ent1.vhd` and `e_ent2.vhd`.

However, for an entity called `\Ent3\`, the parser performs a case-sensitive search for a source file called `e_Ent3.vhd`.

For Verilog, the parser recognizes source files with a `.v` extension by default. If other file extensions are used, you must define the `VERILOG_SUFFIX` variable in the `hdl.var` file. For example:

```
DEFINE VERILOG_SUFFIX (.v, .vlog, .vext)
```

For VHDL, the parser recognizes source files with `.vhd` and `.vhdl` file extensions by default. If other file extensions are used, you must define the `VHDL_SUFFIX` variable in the `hdl.var` file. For example:

```
DEFINE VHDL_SUFFIX (.vhd, .vhdl, .vhext)
```

The *xmvlog* and *xmvhdl* parsers use the list of file extension specified with the `VERILOG_SUFFIX` and `VHDL_SUFFIX` variable to determine the file extension to be appended to the filename.

For example, using the `VERILOG_SUFFIX` definition shown above, if you have a module called `fa_beh`, and the following naming rule

```
(MODULE(m_<MODULE>))
```

The parser searches for a file in the following order: `m_fa_beh.v`, `m_fa_beh.vlog`, `m_fa_beh.vext`.

> For VHDL, you can specify the name of the design file that contains the top-level design unit when you define the `DESIGN_TOP` variable in the compilation command file. The extension of this file has the highest priority. If a file is not found, the parser then looks for a file using the extensions specified with `VHDL_SUFFIX`.

For example, if the design top was specified as:

```
DEFINE DESIGN_TOP (entity1.vhext)
```

and the `VHDL_SUFFIX` variable was defined as:

```
DEFINE VHDL_SUFFIX (.vhdl, .vhd)
```

then the file for a design unit `ent1dep` is searched in the following order:

```
ent1dep.vhext
ent1dep.vhdl
ent1dep.vhd
```

Default Naming Rules

If a naming rule for a particular design unit type is not specified, the default rule(s) shown below for that design unit type is used. If the path to a naming rules file is not specified in the compilation command file, all defaults apply.

```
(ENTITY(<ENTITY>)(<ENTITY>.entity))
(ARCH(<ENTITY>.<ARCH>))
(PACKAGE(<PACKAGE>)(<PACKAGE>.pkg))
(PACKAGEBODY(<PACKAGE>.body))
(CONFIG(<CONFIG>))
(MODULE (<MODULE>))
(UDP (<UDP>))
```

Specifying the Library Mapping

There are several ways to specify which library, or libraries, design units should be compiled into. When compiling files using design-top compilation, it is recommended that you explicitly define the library mapping by defining the `LIB_MAP` variable in the `hdl.var` file. For example:

```
DEFINE LIB_MAP ( ./src/vhdl/design/                 => designlib, \
                 /hm/xyz/abc/fpfile.vhd             => lib1, \
                 /root/designs/main/src/verilog/... => lib2, \
                 /root/designs/main/src/vhdl/...    => lib3, \
                 myfile.v                           => mylib \
                 +                                  => commonlib )
```

See `LIB_MAP` for more information.

Enabling Line Breakpoints Using -linedebug

To enable support for setting line, process, and subprogram breakpoints, use the compiler `-linedebug` option. This option also enables you to single-step through HDL code.

Due to the global nature of this option, it can have a severe impact on simulation performance. If you specify `-linedebug` on the command line when you compile your files (using `xrun` or `xmvlog`), it disables the available optimizations for RTL modules and UVM SystemVerilog code and sets the default access for all simulation objects to read/write/connectivity at elaboration time. You can perform a selective application by compiling specific files with this option. For example:

```
% xmvlog -linedebug [other_options] test1.v
% xmvlog [other_options] test2.v test3.v
```

Using -classlinedebug to Single-Step Through Class Based Objects

To optimize single-step debug, specifying the `-classlinedebug` option using `xrun` or `xmvlog`. This option specifies an alternative single-step debug for class-based objects only. It compiles the dynamic testbench portions of the HDL code as if `-linedebug` were specified for these portions only, excluding the module-based RTL portions and optimizing them for better simulation performance. For example:

```
% xmvlog -sv -classlinedebug [other_options] test1.sv
```

Or:

```
% xrun -sv -claslinedebug [other_options] test1.sv
```

The following table summarizes the optimizations available when specifying `-linedebug`, `-`

`classlinedebug`, or `-uvmlinedebug` on the `xrun` or `xmvlog` command line.

Option	Module	UVM SystemVerilog Code	UVM Library
`none`	optimize	optimize	optimize
`-linedebug`	debug	debug	optimize
`-uvmlinedebug`	debug	debug	debug
`-classlinedebug`	optimize	debug	debug
`-classlinedebug` and `-uvmlinedebug`	optimize	debug	debug
`-linedebug` and `-uvmlinedebug`	debug	debug	debug
`-classlinedbug` and `-linedebug`	debug	debug	debug

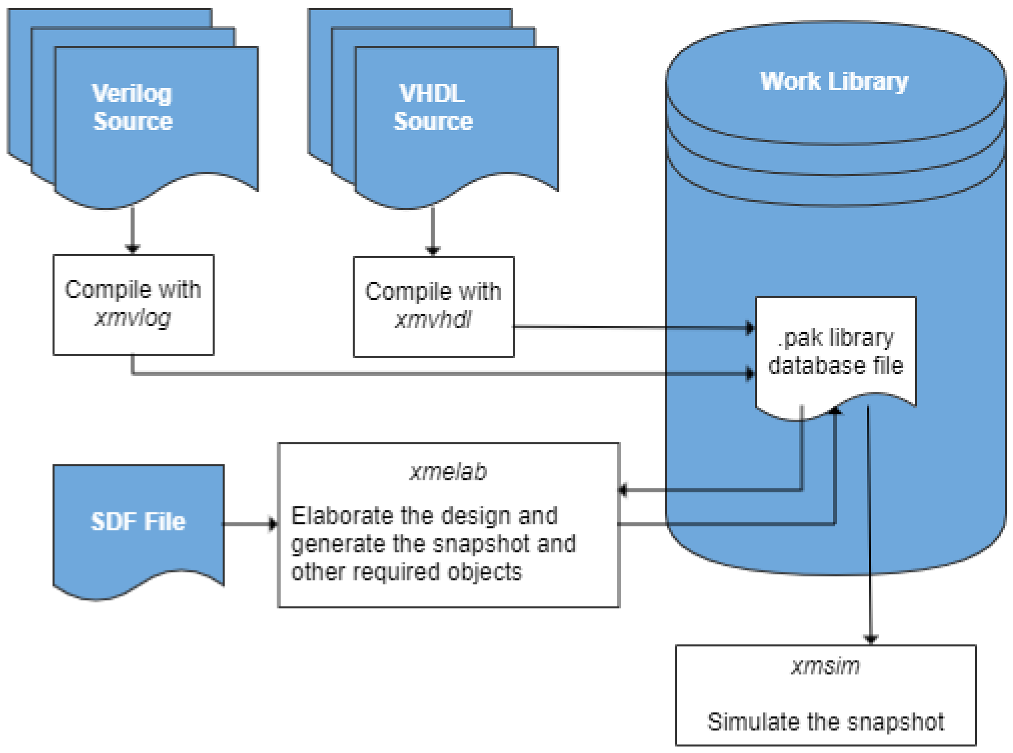
⚠ UVM classes in UVM libraries are not excluded with `-classlinedebug`, meaning that single-step debug is available for these classes. There is no way to distinguish between UVM and non-UVM classes at this time.

Elaboration

The *xmelab* utility is a language-independent elaborator. It constructs a design hierarchy based on the instantiation and configuration information in the design, establishes signal connectivity, and computes initial values for all objects in the design. The elaborated design hierarchy is stored in a simulation snapshot, which is the representation of your design that the simulator uses to run the simulation. The snapshot is stored in the library database file along with the other intermediate objects generated by the compiler and elaborator.

The XMELAB Flow

The following figure illustrates the *xmelab* process.



By default, the elaborator performs the following actions:

- It marks all simulation objects in the design as having no read, write, or connectivity (load and driver) access. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve simulation performance. However, turning off access to the HDL data structures means that you cannot access these objects from a point outside the HDL code through Tcl commands or through PLI, VPI, VHPI. You can set the global visibility access to simulation objects with the `-access` option when you invoke *xrun* or *xmelab*. You can also use the `-afile` option to include an access file, which lets you set the visibility access for particular instances or portions of a design.
- It forks off a single code generation process. If you are running on a multicore machine, you can use the `-mccodegen` option to fork off multiple code generator processes. With multiple code generation processes running in parallel, the amount of time the elaborator spends in code generation can be decreased.

When you change any of the design units in the hierarchy, you must recompile the design units that you have changed and re-elaborate the design hierarchy. You can automatically recompile all out-of-date design units and re-elaborate the design by:

- Running *xmupdate*. This utility runs *xmvlog* to recompile any changed Verilog design units and *xmvhdl* to recompile any changed VHDL units, and then runs *xmelab* to re-elaborate the design. *xmelab* also automatically invokes the *xmsdfc* utility to recompile the SDF source file if it detects a change in the file. The elaborator then generates a new snapshot. Use *xmupdate* when you want to update the snapshot, but do not want to simulate. See *xmupdate* for more information.
- Including the `-update` option on the `xmsim` command. This option calls *xmupdate*, which recompiles any changed design units, recompiles the SDF file if necessary, re-elaborates the design, generates a new snapshot, and then invokes the simulator. Use `xmsim -update` if you want to update and simulate.

You must elaborate the entire design at least once before you can use either feature to automatically update the design.

Related Topics

- Elaborating a Design with Multiple Top-Level VHDL Design Units
- Updating Design Changes at Simulation Time

Elaborating with XMELAB

The *xmelab* utility is invoked with command-line options and arguments. You can specify the options and arguments in any order, but parameters to options must immediately follow the options that they modify.

If you are running the simulator in single-step invocation mode with *xrun*, the tool automatically figures out what the top-level Verilog modules are, and the snapshot name uses the library and cell of the first top-level module that is processed (if all modules are in one file, this is the first top-level module in the file). If the top-level design unit is not Verilog, you must use the `-top` option to specify the top-level unit. If you want to compile your source files and elaborate a design with *xrun*, use the `-elaborate` option. This stops the simulator after elaboration and saves a snapshot in *Lib.Cell:View* format, assuming the `-snapshot` option was not used to explicitly name the snapshot.

The argument to the `xmelab` command can be:

- The *Library.Cell:View* name(s) of the top-level HDL design unit(s). Design units specified on the command line cannot be instantiated in the design.
  The top-level unit(s) specified on the command line can be:
  - One or more Verilog top-level units.
  - One or more VHDL top-level units.

- One or more Verilog top-level units and one or more VHDL top-level units.

  Syntax:

  ```
  % xmelab [options] [lib.]cell[:view] [[lib.]cell[:view]...]
  ```

  - You must specify the cell (top-level unit name).
  - You must specify the library if a top-level unit with the same name exists in more than one library.

    ⚠ Alternatively, use the `-auto_top_cell_binding` option to instead bind the top cell using the default library scanning order. If using this option results in a successful binding of the top cell, then a note like the one displayed below is output to the screen:

    ```
    xmelab: *N,BIATS: More than one unit matches for top cell 'foo',
    automatically selected top is 'module libfoo1.foo:sv (VST)', among:
     module libfoo2.foo:sv (VST)
     module libfoo1.foo:sv (VST)
    ```

- If there are multiple views (Verilog views or VHDL architectures) of the top-level unit(s), the easiest, and recommended, thing to do is to specify the view on the command line. If you do not specify the library and the view, *xmelab* uses the following rules to resolve the reference to the top-level design unit:

  - Search the library defined with the `WORK` variable in the `hdl.var` file.
    If the cell exists in the work library, search for the views of the cell.
    If one view is found, use that view.
    If more than one view exists in the work library:

      For Verilog, generate an error message.
      For VHDL, use the most-recently analyzed architecture.

    If no views for the cell are found, generate an error message.
    If the cell exists in more than one library, or if the cell does not exist in any library, generate an error.
    If the cell exists in one library, search for the views in that library.

      If one view is found, use that view.
      If more than one view exists in the library:

        For Verilog, generate an error message.
        For VHDL, use the most-recently analyzed architecture.

If no views for the cell are found, generate an error message.

- If the WORK variable is not defined in the hdl.var file, or if the cell does not exist in the work library, search the libraries defined in the `cds.lib` file.

  Examples:

```
% xmelab [options] test         One top-level design unit (top-level
                                Verilog module or top-level VHDL entity)
% xmelab [options] test top     Multiple top-level design units
```

> ⚠ If you are running in single-step invocation mode with *xrun*, you must use the `-top` option to specify the top-level VHDL unit(s) in a pure VHDL design, or the top-level unit(s) in a mixed Verilog/VHDL design. *xrun* can automatically determine the top-level design unit(s) for a pure Verilog design. For example, assuming that `test` and `top` are both top-level VHDL entities, the `xrun` command requires two `-top` options.

```
% xrun [options] -top test -top top source_files
```

- The name of a Verilog configuration.
  - If the configuration is specified in a library map file, use the `-libmap` option to specify the library map file.

    ```
    % xmelab -libmap library_map_file [other_options] configuration_name
    ```

  - If you are running the simulator using *xrun*, use the following command. It is not necessary to use the `-top` option to specify the top-level design unit.

    ```
    % xrun -libmap library_map_file [-compcnfg] [other_options] source_files
    ```

  - If the configuration is specified in a Verilog source file, specify the name of the configuration.

    ```
    % xmelab [options] configuration_name
    ```

    or, if you are using *xrun*:

    ```
    % xrun [-compcnfg] [other_options] source_files
    ```

    > ⚠ If the configuration is in a Verilog source file, but you are using a library map file that contains library declarations to control the compilation of design units into libraries, you must include the `-libmap` option to specify the library map file. The elaborator searches libraries in the order in which they are specified in the library map file.
    >
    > ```
    > % xmelab -libmap library_map_file [other_options] configuration_name
    > ```

- The name of a 5.x (or legacy) configuration.

Specifying the full legacy library structure, and any additional files, makes it possible for tools that do not understand the default packed library structure to access the required intermediate objects. For example, you must compile the Verilog or VHDL source files with the `-use5x` option if you want to use a 5.x configuration file to control the binding of instances during elaboration.

Syntax:

```
% xmelab [options] [lib.]cell[:view]
```

You can choose to enable automatic generation of a VHDL configuration file by specifying the `-conffile` option on the `xmelab` command line. The syntax for this command is as follows:

```
xmelab [options] -conffile configuration_filename [lib.]cell[:view]
```

Elaborating a Design with Multiple Top-Level VHDL Design Units

The Xcelium simulator supports designs with multiple VHDL top levels.

If you are simulating in single-step invocation mode with *xrun*, the tool can automatically determine the top-level design unit(s) for a pure Verilog design. However, you must use the `-top` option to specify the top-level VHDL unit(s) in a pure VHDL design, or the top-level unit(s) in a mixed Verilog/VHDL design. For example:

```
% xrun -top entity1 -top entity2 [other_options] top1.vhd top2.vhd
```

In this example, `top1.vhd` and `top2.vhd` are recognized as VHDL files and are compiled using *xmvhdl*. The top-level design hierarchies specified for top-level designs `entity1` and `entity2` are then elaborated with *xmelab* and then simulated using *xmsim*. The multiple `-top` options are required to specify the multiple VHDL tops of the design.

If you are running in multi-step invocation mode, the top-level design units are specified on the `xmelab` command line.

```
% xmvhdl [compile_options] top1.vhd top2.vhd
% xmelab [elab_options] entity1 entity2
% xmsim snapshot_name
```

Default Debug Scope

If multiple VHDL tops are specified, the default debug scope is set to the first VHDL top-level entity specified on the command line.

For a design that has both VHDL and Verilog tops, the default scope is set to the first VHDL top specified on the command line.

Resolving Hierarchical Names

In Verilog, you can reference any object in the design unambiguously by providing a full hierarchical path to the object beginning with the name of the top level. For example, if there are two top-level Verilog modules `top1` and `top2`, all names can be resolved by using absolute paths such as the following:

```
top1.inst1.inst2.w
top2.inst1.inst2.w
```

VHDL, however, uses a colon (`:`) to represent the VHDL top. This results in ambiguity if the design has multiple VHDL top levels. For example, if there are two top-level entities, `top1` and `top2`, the following hierarchical name is not valid because the `:` cannot be resolved to either top. It is not clear if the signal `w` lies in the `top1` or in the `top2` hierarchy.

```
:inst1:inst2:w
```

For designs with multiple VHDL tops, many enhancements have been implemented so that you can include the name of the top entity in hierarchical names. The following syntax for hierarchical names can be used if there are multiple VHDL tops:

```
top_entity_name:hierarchical_instance_path_name:signal_name
```

For example:

```
top1:inst1:inst2:w
```

The following delimiters are supported:

- colon (`:`)
- period (`.`)
- forward slash (`/`)

The following hierarchical names are identical:

```
top1:inst1:inst2:w
top1.inst1.inst2.w
top1/inst1/inst2/w
```

The following syntax is also supported in case of multiple VHDL tops:

```
/top1/inst1/inst2/w
```

⚠ For a design with a single VHDL top, only hierarchical names beginning with `:` are valid.

Using a hierarchical name that includes the name of the top-level entity to resolve ambiguity when there are multiple VHDL tops affects several areas, including the following:

- Supported interfaces
  Interfaces, such as Tcl, VHPI, and VDA, which accept hierarchical names, require that the top entity name be specified in absolute hierarchical paths to resolve ambiguity.

  For Tcl commands that accept hierarchical names, (for example, `probe`, `value`, `drivers`, `deposit`, and `force`), an absolute hierarchical path to an object must start with the top entity name in case of multiple VHDL tops. For example:

  ```
  value top1:inst1:inst2:w
  force top2.d2/a '1'
  ```

  If there are multiple VHDL tops, specifying a name that starts with `:` results in an error.

  In addition, the hierarchical object name reporting from commands reports hierarchical names with the top VHDL entity name.
- The XLM utilities `xm_mirror`, `xm_deposit`, `xm_force`, `xm_release`, and `xmreadmem` support hierarchical names with the top VHDL entity name.
- VHDL attributes `instance_name and `path_name
  If there are multiple VHDL tops, the instance name and the path name must start with the name of the top-level entity, not `:`.
- `-generic` and `-gpg` options
  These elaborator options are used to specify the value for all VHDL generics that match the specified name. These options accept object names, instance_name:object_name, and hierarchical names. The options set the value of generics matched in all top-level VHDL hierarchies.

VHPI interface

The tag `vhpiRootInstK` returns the handle for the top-level entity in a VHDL design. To support multiple VHDL tops, a new tag `vhpiRootInsts` has been introduced, which supports 1-to-many mappings. Support of `vhpiIterator` over this tag has been added to return the handles to multiple VHDL tops present in a design. The tag `vhpiScan` can then be used on this handle to access the top-level entities.

For VHPI attributes `vhpiFullNameP` and `vhpiFullCaseNameP`, the return string includes the top entity name in case of multiple VHDL tops.

Restrictions

Specifying a VHDL top and a Verilog top with the same name is not permitted even if the cases are different, as VHDL is case insensitive. The elaborator generates an error if the names of the tops are the same.

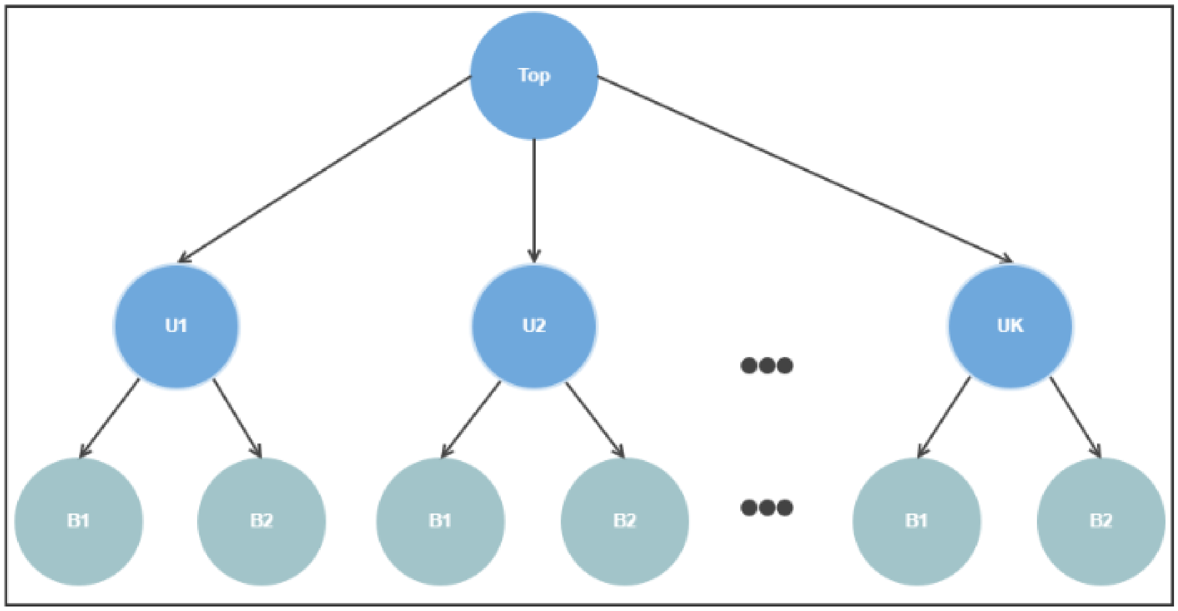
Elaborating a Design with Replicated Blocks

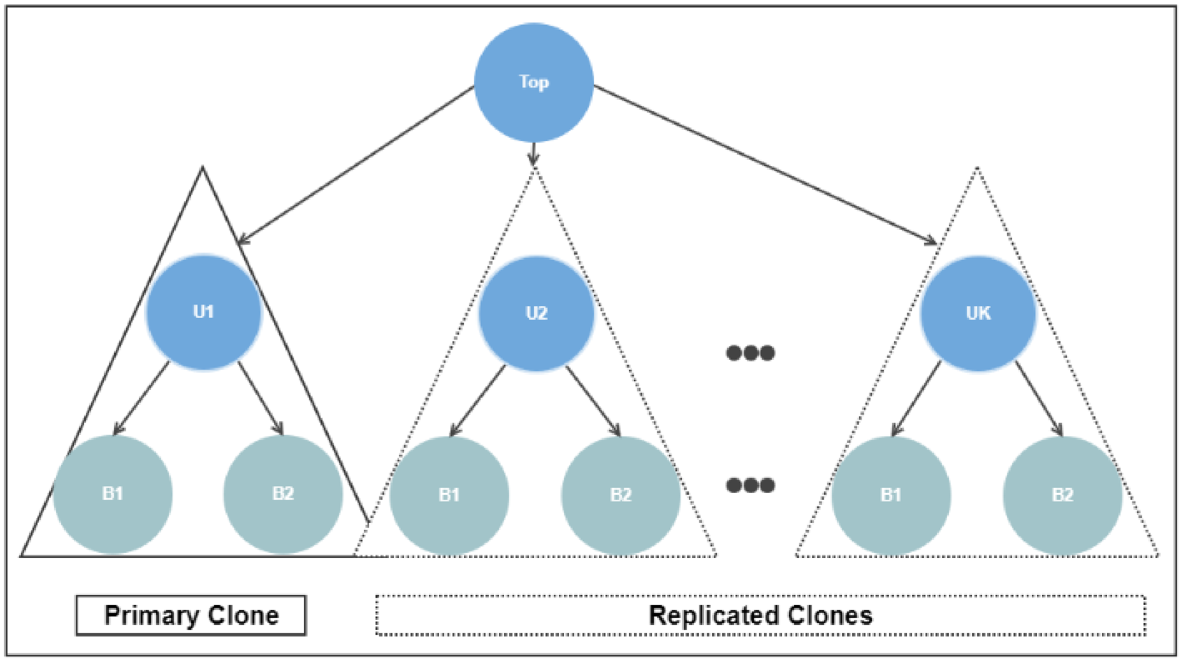
In the monolithic elaboration flow of Xcelium, each instance of the design hierarchy is elaborated individually. This requires repetitive elaboration for designs with replicated functional blocks. To reduce build time and memory for creating a design hierarchy for repetitive designs, use the `-xmclone` option during elaboration. The `-xmclone` option automatically partitions the design and finds one or more replicated blocks keeping one representative instance hierarchy for each replicated set and the non-replicated portion of the design. It then elaborates the replicated blocks by cloning them to create a final snapshot for simulation. All these steps are performed in a single elaboration build flow. To clone a design hierarchy, the `-xmclone` option automatically identifies one or more replicated blocks.

The `-xmclone` option offers the following benefits:

- Faster build performance with minor overhead on simulation performance (no greater than 5%)
- Reduces peak memory
- No external input is needed

The following figures illustrate a replicated design hierarchy and the elaboration flow with the `-xmclone` option. The `-xmclone` option automatically identifies clones based on heuristics that identify candidates for cloning. In the following figure, `U2 … UK` are clones of instance `U1`. It builds the instance `U1`, replicates instances `U2 ... UK`, and builds a complete instance hierarchy for run time.


Top
U1
U2
UK
B1
B2
B1
B2
B1
B2



To enable cloning for a design hierarchy, specify the `-xmclone` option with xmelab. For example:

```
xrun test.sv -xmclone
```

Alternatively, you can use the `-xmclone_top` option to specify specific module names if different modules are instantiated multiple times in the design hierarchy. In this case, partitions are created in the design for the specified modules only.

If the elaborator cannot find any replicated hierarchy that qualifies for cloning, the tool does not clone the hierarchy, and the elaboration proceeds with the normal flow. Note that to qualify for cloning, all parameters in the replicated hierarchy must have the same values.

Currently, cloning is not supported for the following technologies:

- Power Playback
- IXCOM
- VHDL
- Functional safety
- Fault simulation
- Jupiter
- Multi-Core
- MSIE/DSS (Parallel/Incremental Build)
- Mixed Signal - Embedded Hierarchical Dynamic Voltage Supply (DVS)

If any technologies mentioned above are present in the design, the tool does not clone any hierarchy, and the elaboration proceeds with the normal flow.

Consider the following example that illustrates the usage of the `-xmclone` option to clone a design hierarchy.

```
module tb;
   top #(1) t1();
   top #(2) t21();
   top #(2) t22();
   top #(2) t23();
   top #(2) t24();
   top #(3) t3();
endmodule // tb

module top #(parameter p = 0);
   child #(p) u1();
   child #(p+2) u2();
   initial $display ("top");
endmodule // top
module child #(parameter q = 0);
   reg [q:0] r;
   initial $display (q);
endmodule // child
```

Run the following command to clone the hierarchy mentioned above:

```
xrun test.sv -xmclone
```

Output

```
[CLONE] Searching for instances to clone ...

[CLONE] Done.

[CLONE] Clone set for design unit 'worklib.top'
[CLONE]    1: tb.t21
[CLONE]    2: tb.t22
[CLONE]    3: tb.t23
[CLONE]    4: tb.t24

 Building instance overlay tables: .......... Done
 Generating native compiled code:
  worklib.child:sv <0x262daa9c>
   streams:   1, words:   415
  worklib.child:sv <0x4067e1cd>
   streams:   1, words:   415
  worklib.top:sv <0x1184246a>
   streams:   1, words:   369
  worklib.child:sv <0x2236678f>
   streams:   1, words:   415
  worklib.top:sv <0x2c35c8b6>
   streams:   1, words:   369
  worklib.child:sv <0x7c2c7449>
   streams:   1, words:   415
  worklib.child:sv <0x42d26b0e>
   streams:   1, words:   413
  worklib.top:sv <0x5756110e>
   streams:   1, words:   369
 Building instance specific data structures.
 Loading native compiled code:     .................... Done
 Design hierarchy summary:
            Instances  Unique
  Modules:         19       3
  Registers:       12       1
  Initial blocks:  18       2
 Writing initial simulation snapshot: worklib.tb:sv
Loading snapshot worklib.tb:sv .................... Done
xcelium> source /grid/avs/install/xcelium/AGILE7/latest/tools/xcelium/files/xmsimrc
xcelium> run
          1
          3
top
          3
          5
```

```
top
          2
          4
top
          2
          4
top
          2
          4
top
          2
          4
top
xmsim: *W,RNQUIE: Simulation is complete.
```

Related Topic

- Elaboration

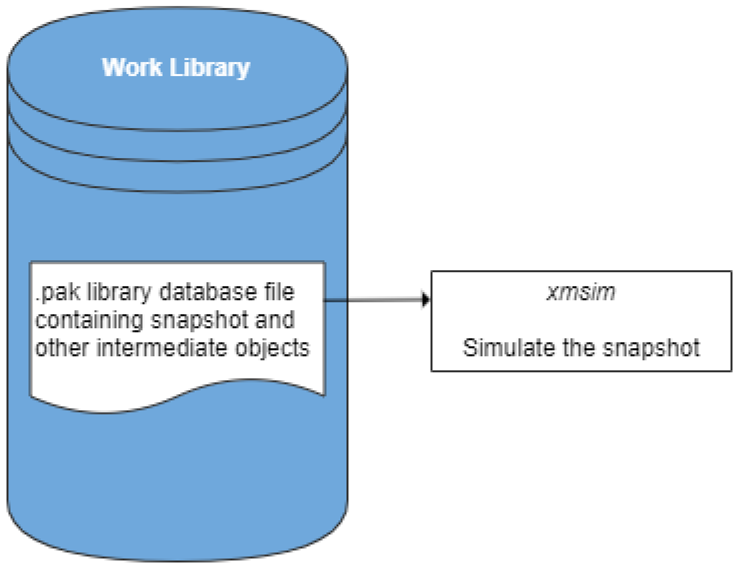
Simulation

The *xmsim* utility loads the *snapshot* as its primary input. It then loads other intermediate objects referenced by the snapshot. In the case of interactive debugging, HDL source files and script files can also be loaded. Other data files can be loaded as demanded by the model being simulated (via `$read*` tasks or Textio).

The outputs of simulation are controlled by the model or debugger. These outputs can include result files generated by the model, Simulation History Manager (SHM) databases, Value Change Dump (VCD) files, and so on.

The XMSIM Flow

The following figure illustrates the *xmsim* process flow.


Work Library
.pak library database file containing snapshot and other intermediate objects
xmsim
Simulate the snapshot

Rules for Resolving the Snapshot Reference

A snapshot name uses the `[lib.]cell[:view]` format. If you do not specify a library or a view, *xmsim* uses the following rules to resolve the snapshot reference on the command line (assuming that the command line is `% xmsim top`):

1. Check if the `WORK` variable set in the `hdl.var` file.

```
YES => Does WORK.top exist?
    YES => How many views of WORK.top have snapshots?
        1 => Simulate this snapshot.
            More than 1 => Error message
              (More than one snapshot matches "top")
    NO => Go to Step 2.
NO => Go to Step 2.
```

2. Search all libraries in the `cds.lib` file.

```
    Does LIB*.top exist?
        YES => How many views of LIB*.top have snapshots?
            1 => Simulate this snapshot.
                More than 1 => Error message
                  (More than one snapshot matches "top".)
        NO => Error message
                  (Snapshot "top" does not exist in the libraries.)
```

Compressed File Support when Using User-Defined Options

Compressed files save storage space when managing larger files. Both *xrun* and *xmsim* allow you to use compressed files with user-defined plus options.

The simulator recognizes and extracts the following archive formats:

- Gnu zip compression (`.gz`)
- Standard compression (`.z`)

For example:

```
% xmsim [lib.]cell[:view] +TESTFILE=VER.FULLSCAN.test.verilog.gz
```

Multiple compressed files are supported. If multiple compressed files are processed together, then *xmsim* extracts each file into a `/tmp` directory. Once every file is decompressed and the simulation is complete, the `/tmp` directory is deleted.

Related Topics

- Elaborating with XMELAB
- Introduction to SDF Timing Annotation

Invoking the Simulator

You can invoke the simulator (*xmsim*) in two modes:

- **Non-interactive mode:** Automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input. See Invoking the Simulator in Noninteractive Mode.
- **Interactive mode:** Stops the simulation at time 0 and returns the `xcelium>` prompt. See Invoking

the Simulator in Interactive Mode.

In either mode, you can invoke the simulator with or without the SimVision analysis environment.

The syntax for invoking the simulator is:

```
% xmsim [options] snapshot_name
```

The options and arguments can occur in any order except that parameters to options must immediately follow the option they modify.

The *snapshot_name* argument is a `[lib.]cell[:view]` specification. You must specify the cell and only one snapshot can be specified.

- If a snapshot with the same cell name exists in more than one library, the easiest (and recommended) thing to do is to include the library name on the command line.
- If there are multiple views that contain snapshots, the easiest (and recommended) thing to do is to include the view name on the command line.

If you do not specify a library or a view, *xmsim* uses a set of rules to resolve the snapshot reference on the command line. See Rules for Resolving the Snapshot Reference.

Invoking the Simulator in Non-interactive Mode

Invoking *xmsim* in noninteractive mode automatically starts the simulation.

- To invoke *xmsim* in non-interactive mode without the SimVision analysis environment, type:

  ```
  % xmsim [-run] [other_options] snapshot_name
  ```

  Examples:

  ```
  % xmsim worklib.top:module
  % xmsim -run worklib.top:module
  ```

  The `-run` option is not required.

- To enable the processing of commands and other input from a script file, type:

  ```
  % xmsim [-run] [other_options] -input <filename>.tcl snapshot_name
  ```

  Example:

  ```
  % xmsim -input input.tcl worklib.top:module
  ```

- To invoke *xmsim* in non-interactive mode with the SimVision analysis environment, type:

  ```
  % xmsim -gui -run [other_options] snapshot_name
  ```

  Example:

  ```
  % xmsim -gui -run worklib.top:module
  ```

The `-run` option is required.

If you have included the `-tcl` option in your `XMSIMOPTS` variable in the `hdl.var` file because you invoke the simulator in interactive mode most of the time, use `-batch` to override `-tcl` and simulate in non-interactive mode.

Example:

```
% xmsim -batch worklib.top:module
```

If you want to run in noninteractive mode, but do not want the simulator to exit at the end of the simulation, use both the `-run` and `-tcl` options. The following combination of options runs *xmsim* in noninteractive mode, but returns the `xcelium>` prompt instead of exiting:

```
% xmsim -run -tcl snapshot_name
```

Include the `-exit` option on the command line if you want the simulator to exit under conditions that would normally stop the simulation and put it in interactive mode.

The following examples illustrate the various options for invoking the simulator in noninteractive mode. Other `xmsim` command options are not shown.

`% xmsim [-run] top`	Invokes *xmsim* and runs the simulation.
`% xmsim -gui -run top`	Invokes *xmsim* with the SimVision analysis environment and runs the simulation.
`% xmsim -run -tcl top`	Invokes *xmsim* and runs the simulation. When the simulation is completed or interrupted, returns the `xcelium>` prompt instead of exiting.
`% xmsim -batch top`	Use `-batch` if you want to simulate in noninteractive mode, but have the `-tcl` option included with the `XMSIMOPTS` variable in the `hdl.var` file.

Invoking the Simulator in Interactive Mode

You can interact with your design throughout a simulation by instructing *xmsim* to stop at the beginning of the simulation so that you can enter interactive mode at simulation time 0.

If you are using the command-line interface, use the `-tcl` option when you invoke the simulator.

```
% xmsim -tcl [other_options] snapshot_name
```

Example:

```
% xmsim -tcl worklib.top
```

If you are using the SimVision analysis environment, *xmsim* automatically stops at the beginning of the simulation.

```
% xmsim -gui [-tcl] [other_options] snapshot_name
```

Examples:

```
% xmsim -gui worklib.top
% xmsim -gui -tcl worklib.top
```

The `-tcl` option is not required when running SimVision.

Updating Design Changes at Simulation Time

When you change design units in the hierarchy, you *must* recompile them and re-elaborate the design hierarchy.

If you want to update *and* simulate, use the `-update` option with the `xmsim` command to automatically recompile and re-elaborate all out-of-date design units. This option calls *xmupdate* to recompile any changed design units, re-elaborate the design, and generate a new snapshot. It then invokes the simulator and loads the new snapshot.

If you only want to update the snapshot, run the *xmupdate* utility. See *xmupdate* for more information.

The purpose of *xmupdate* is to provide quick design change turnaround when you have edited a design unit. The modifications to design units cannot cross file boundaries to modify other files. Do not use *xmupdate* (or `xmsim -update`) after adding a design unit, a source file, or compiler directives to the design. For instance, *xmupdate* does not update correctly if you edit a source file to define a new macro, or if you change a design unit in a way that introduces a new cross-file dependency. In these cases, it's recommended to recompile the design with `xmvlog` or `xmvhdl -update`.

Use the `-nosource` option with `-update` if you recompile selected parts of your design and then want to automatically re-elaborate and load the new snapshot into the simulator. For instance, suppose you have edited two source files, `first.v` and `second.v`, and only want to include the changes in `second.v`. You can recompile the file `second.v` and then use `xmsim -update -nosource`.

```
% xmsim -update -nosource snapshot_name
```

You can also use `-nosource` when you have changed one design unit in a file with more than one design unit. In this case, follow the steps below to recompile only the unit you have changed and then use `xmsim -update -nosource` to re-elaborate the design and invoke the simulator.

1. Recompile the changed design unit.

   ```
   % xmupdate -unit [lib.]cell[:view]
   ```

   or

   ```
   % xmvlog -unit [lib.]cell[:view]
   ```

2. Use `xmsim -update -nosource` as below.

   ```
   % xmsim -update -nosource snapshot_name
   ```

If you make a change to any SDF-related file (the SDF source file, the compiled SDF file, the SDF configuration file, or the SDF command file), and then execute an `xmsim -update` command, the elaborator automatically re-annotates the design using the new, up-to-date files. SDF source files that have changed are automatically recompiled.

The following example shows how to use `xmsim -update` to automatically recompile, re-elaborate, and load a new snapshot.

```
;# After editing a design unit, use -update to automatically recompile,
;# re-elaborate, and reinvoke the simulator.

;# The -update option calls xmupdate to recompile the file counter.v, which
;# contains the changed module, m16.

% xmsim -nocopyright -messages -update -tcl board

Updating snapshot worklib.board:module (SSS), reason: file `./counter.v' is newer than
expected.

       expected: Mon May 3 15:20:43 1999

       actual: Tue Jun 1 08:52:57 1999

Updating: module worklib.m16:module (VST)

file: ./counter.v

    module worklib.m16:module

       errors: 0, warnings: 0

;# xmupdate re-elaborates the design hierarchy.

Updating: snapshot worklib.board:module (SSS)

Update of snapshot worklib.board:behav (SSS) successful.

;# The -update option automatically reinvokes the simulator.

Loading snapshot worklib.board:module ................... Done

xcelium>
```

The following example shows how to recompile one module in a file containing multiple modules before using `-update`.

```
;# The source file 16bit_alu.v contains four modules. After editing module
;# logic, use xmupdate-unit to update only that module.

% xmupdate -nocopyright -messages -unit worklib.logic:module

Updating: module worklib.logic:module (VST)

file: ./16bit_alu.v

    module worklib.logic:module

       errors: 0, warnings: 0

;# Then use xmsim -update -nosource to re-elaborate, generate a new snapshot,

;# and reinvoke the simulator.

% xmsim -nocopyright -messages -tcl -update -nosource alu_16

Updating snapshot worklib.alu_16:module (SSS), reason: dependent module
worklib.logic:module (VST) is newer than expected.

       expected: Tue Oct 8 11:19:04 1996

       actual: Tue Oct 8 11:20:31 1996

Updating: snapshot worklib.alu_16:module (SSS)

Update of snapshot worklib.alu_16:module (SSS) successful.

Loading snapshot worklib.alu_16:module .................... Done

xmsim: *W,VSCNEW: file `./16bit_alu.v' is newer than expected by module

worklib.alu_16:module (VST).

xcelium>
```

Including User-Defined Plus Options

Use the `$test$plusargs` system function in your Verilog code to check for the presence of a plus run-time option on the command line.

Example 1:

```
initial

  if ($test$plusargs("dumpon"))
  begin
    $dumpfile("results.vcd");
    $dumpvars;
  end
```

In the above code example, a user-defined option named `+dumpon` is defined to output values to a VCD file if the option is specified on the `xmsim` command line as below. No recompilation is necessary.

```
% xmsim +dumpon snapshot_name
```

Example 2:

Alternatively, you can choose to use `$test$plusargs` to set up different stimuli that then control the simulation.

```
initial
  if ($test$plusargs("test01"))
  $readmemh("test01.dat", stim_mem);

  if ($test$plusargs("test02"))
  $readmemh("test02.dat", stim_mem);

  if ($test$plusargs("test03"))
  $readmemh("test03.dat", stim_mem);

  if ($test$plusargs("test04"))
  $readmemh("test04.dat", stim_mem);
```

For instance, using the above `$test$plusargs` system functions, you can change the stimulus without rebuilding your design.

This method is also an easy way to run multiple simulations in parallel, as shown:

```
% xmsim snapshot_name +test01 &
% xmsim snapshot_name +test02 &
% xmsim snapshot_name +test03 &
% xmsim snapshot_name +test04 &
```

Providing Interactive Commands from a File

You can load a file containing simulator commands by specifying an input file. This is useful when you want to load a file of Tcl commands or aliases, or when you want to reproduce an interactive session by re-executing a file of commands saved from a previous simulation run (a key file). When *xrun* or *xmsim* has processed all of the commands in the input file, or if you interrupt processing, input reverts back to the terminal.

There are three ways to execute the commands in an input file:

- Specify the input file with the `-input` option when you invoke the simulator.

  When you use the -input option, commands contained in the input file are executed at the beginning of the simulation session.

- Specify the input file with the Tcl `input` command.
- Execute the Tcl `source` command.

The behavior of the `input` command and the `-input` option is different from the behavior of the `source` command in the following ways:

- With the `source` command, execution of the commands in the script stops if a command generates an error. With the `input` command or with the `-input` option, the contents of the file are read in place of standard input at the next Tcl prompt, as if you had typed the commands at the command-line prompt. This means that errors do not stop the execution of commands in the script.
- The `input` command and the `-input` option echo commands to the screen as they are executed, along with any command output or error messages. The `source` command, on the other hand, displays the output of only the last command in the file. Output from the model (for example, the output of `$display`, `$monitor`, or `$strobe` tasks, or the output of stop points) is printed to the screen.

If you are using the SimVision analysis environment:

1. Select *File – Source Command Script*.

   This opens the *Select SimVision Command Script* form.
2. Select the type of input file from the *Files of type* drop-down menu.
3. Enter the name of the file that contains your Tcl commands in the *Filename* field.
4. Click the *Open* button.

Command-Line Parameters for the -input Option

When using the `-input` option on the *xrun* or *xmsim* command line, you can specify either a filename or a simulator command as an argument.

To specify a filename, use a Tcl script file in ASCII format as input. A Tcl script file lists each of the Tcl commands that you want to run on separate lines.

```
% xrun -input setup.tcl tb.v dut.v
```

Or:

```
% xmsim -input setup.tcl worklib.top:module
```

To specify a simulator command, use the @ symbol before the command. You must enclose everything in quotation marks if the command takes an argument, modifier, or option.

For instance:

```
% xrun -input "@probe -screen *; run; exit" ffnand.v
```

Or:

```
% xmsim -input "@probe -screen *; run; exit" worklib.ffnand:v
```

You can include more than one `-input` option of either form on the command line. The input files (or commands) are processed in the order in which they appear on the command line.

Example

The following example shows how to use the `-input` option when you invoke the simulator using *xmsim*.

```
;# Create the input file using a text editor.
;# Key files can also be used as input files.
;* Command input file: set_break.tcl
stop -create -line 27
run
value data
run 50
value data
run 50
value data
run
```

```
;# Run xmsim with the -input option to specify the input file.
;# The simulator executes the commands in the file.
% xmsim -nocopyright -messages -input set_break.tcl hardrive
Loading snapshot worklib.hardrive:module .................... Done
xcelium> stop -create -line 27
Created stop 1
xcelium> run
0 FS + 0 (stop 1: ./hardrive.v:27)
./hardrive.v:27 repeat (2)
xcelium> value data
4'hx
xcelium> run 50
Ran until 50 NS + 0
xcelium> value data
4'h0
xcelium> run 50
at time 50 clr =1 data= 0 q= x
Ran until 100 NS + 0
xcelium> value data
```

```
4'h0
xcelium> run
at time 150 clr =1 data= 1 q= 0
at time 250 clr =1 data= 2 q= 1
...
...
...
at time 3250 clr =0 data=15 q= 0
at time 3350 clr =0 data=15 q= 0
Simulation complete via $finish(1) at time 3400 NS + 0

;# Control reverts back to the terminal after xmsim
;# has executed all commands in the file.
xcelium> exit
```

Starting the Simulation with Tcl or SimVision

With Xcelium, there are two alternative ways to start or resume a simulation using Tcl or SimVision:

- If you are using the Tcl command-line interface, use the `run` command. This command has several options that let you control when the simulation is to stop:
  - **`-delta`**: Run to the beginning of the next delta cycle or to a specified delta cycle.
  - **`-next`**: Run one behavioral statement, stepping over any subprogram calls.
  - **`-return`**: Run until the current subprogram (task, function, procedure) returns.
  - **`-step`**: Run one behavioral statement, stepping into subprogram calls.
  - **`-timepoint`**: Run for a specified number of time units.
  - **`-phase`**: Run to the beginning of the next phase of the simulation cycle. The two phases of a simulation cycle are signal evaluation and process execution.
  - **`-process`**: Run until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a `process` statement. In Verilog, it is an `always` block, `initial` block, or one of several kinds of anonymous behaviors that can be scheduled to run.
- If you are using the SimVision analysis environment, use the commands on the *Simulation* menu. To simulate for a specified number of time units, set a time breakpoint before starting the simulation. You can also control the simulation by using the simulation buttons on the simulation toolbar.

Related Topic

- Setting a Time Breakpoint

Saving and Restarting the Simulation with Tcl or SimVision

You can save and restart the simulation state at any time. Creating simulation checkpoints is especially useful for large simulations where you might want to save the simulation state at regular intervals. Another common use is to save the simulation state after the circuit has been initialized so that future simulations can begin at that point rather than from time 0.

When you save the simulation state, the simulator creates a new snapshot. To restart the simulation at a later time, you must load the saved snapshot.

The current simulation state that is saved in the snapshot includes the simulation time and all object values, scheduled events, annotated delays, the contents of the memory allocated for access type values, and file pointers. It does not include aspects of the debugging environment such as breakpoints, probes, Tcl variables, and GUI configuration. PLI/VPI callbacks and handles are saved under certain circumstances. Please refer to the PLI/VPI manuals for details.

You cannot save a snapshot if the simulator is in the process of executing sequential HDL code. If the simulation is in a state that cannot be saved, you must use Tcl `run -clean` to run the simulation until the currently running sequential behavior (if any) suspends itself at a delay or event control or a VHDL wait statement.

- If you are using the Tcl command-line interface, use the `save` command to save the simulation state and the `restart` command to load a saved snapshot.

  See `save` and `restart` for details on these commands. The documentation for the `save` command includes an example of saving and restarting.

- If you are using the SimVision analysis environment, select Simulation – Save Checkpoint and fill in the Save Checkpoint form to save the simulation state.

  To restart, select *Simulation – Restart from Checkpoint* and fill in the *Restart from Checkpoint* form.

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
- Breakpoints set at the time that you execute the restart remain set.

⚠ If you set a breakpoint that triggers, for example, every 10 ns (that is, at time 10, 20, 30, and so on) and restart with a snapshot saved at time 15, the breakpoint triggers at 20, 30, and so on, not at time 25, 35, and so on.

- Forces and deposits in effect at the time you issue a save command are still in effect when you restart.

If you exit the simulation and then invoke the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug environment when you restart, make sure that you save the environment with the `save -environment` specification. This creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the Tcl `source` command after restarting or the `-input` command-line option when you invoke the simulator to execute the script.

For example:

```
% xmsim worklib.top
xcelium> (Open a database, set probes, set breakpoints, deposits, forces, etc.)
xcelium> run 100 ns
xcelium> save worklib.top:ckpt1
xcelium> save -environment ckpt1.tcl
xcelium> exit
% xmsim -tcl worklib.top:ckpt1
xcelium> source ckpt1.tcl
```

Some operating systems may impose a limit on the size of a file. If a library database exceeds this limit, you will not be able to add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded. If you reach this limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example:

  ```
  xcelium> save -simulation -overwrite snap1
  ```

- Save snapshots to a separate library. For example:

  ```
  % mkdir xcelium.d/snaplib
  % xmsim -f xmsim.args
  xcelium> run 1000 ns
  xcelium> save -simulation snaplib.snap1
  xcelium> run 1000 ns
  xcelium> save -simulation snaplib.snap2
  ```

⚠ In addition to creating a directory for the library, the library must be defined in the `cds.lib` file.

- Remove snapshots using the *xmrm* utility. For example:

  ```
  % xmrm -snapshot worklib.snap1
  ```

Resetting the Simulation with Tcl or SimVision

With Xcelium, you can reset the currently loaded model to its original state at time zero using Tcl or SimVision.

- If you are using the Tcl command-line interface, use the `reset` command.

  The documentation for the Tcl `save` command includes an example of resetting the simulation.
- If you are using the SimVision analysis environment, select *Simulation – Reset to Start*. The time-zero snapshot, created by the elaborator, must still be available.

  You can also click the *Reset* button on the simulation toolbar.

When you reset the simulation to its state at time 0, the debug environment remains the same.

- Tcl variables remain as they were before the reset.
- SHM and VCD databases remain open, and probes remain set.

  ⚠ VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.
- Breakpoints remain set.
- The SimVision waveform viewer window remains the same.

Forces and deposits that were in effect at the time you issued the `reset` command are removed.

If you exit the simulation instead of resetting, databases are closed, probes and breakpoints are deleted, and Tcl variables are reset to their default values.

Reinvoking the Simulation with SimVision

With Xcelium, if you are using the SimVision analysis environment, you can reinvoke the simulation session at any time, even after the simulation has finished. When you reinvoke, an `exit` command is issued and then the tool (*xrun* or *xmsim*) is invoked again to recompile any changed design units, re-elaborate the design, generate a new snapshot, and load the updated snapshot.

To reinvoke, select *Simulation – Reinvoke Simulator*.

You can reinvoke the simulation inside the graphical environment as many times as you want. If you want to customize the reinvoke options and arguments, set the *Prompt before reinvoke* option on the SimVision *Preferences* form. This lets SimVision display the *Reinvoke* form, which lists the command-line arguments you used when you invoked *xrun* or *xmsim* originally.

Be aware that if you edit the text field on the *Reinvoke* form to remove the `-gui` option, it is not possible to reinvoke from the Tcl `xcelium>` prompt.

- **Single-Step XRUN Reinvocation**

  Using a single *xrun* command line, if the *Prompt before reinvoke* option is set, the *Reinvoke* form displays, letting you edit the *xrun* options and arguments that are specified before running the simulation again. If you add an option that affects elaboration (`-access`, for example), the design is re-elaborated and a new snapshot is created. If the *Prompt before reinvoke* option is not set, the *Reinvoke* form does not appear and the simulator uses the same *xrun* command-line options and arguments specified for the original run.

- **Multi-Step XRUN Reinvocation**

  Using separate *xrun* command lines to compile and/or elaborate a design requires that you then use *xrun* with the `-r` or `-R` option to simulate the snapshot. If you are using the SimVision analysis environment with this multi-step use model, selecting *Simulation – Reinvoke Simulator* will invoke *xrun* again using the original options used to simulate the snapshot. If the *Prompt before reinvoke* option is set, you can edit the reinvoke options before running the simulation; however, multi-step *xrun* introduces the following limitations and restrictions:

  - If you remove the `-R` or `-r` option, a NOMOPT error will result.
  - If you add compilation and/or elaboration options to the prompt, these will be discarded.

  When you reinvoke, your current setup is automatically saved and restored. This includes:

  - SHM databases
  - Probes
  - Breakpoints
  - All signals that were displayed in the SimVision waveform viewer

  Forces and deposits that were in effect at the time you reinvoke are removed.

- **XMSIM Reinvocation**

  Using the *xmsim* command line in direct invocation mode works like *xrun* in single-step mode. If you have set the *Prompt before reinvoke* option on the *Preferences* form, SimVision displays the *Reinvoke* form, which lists the command-line arguments you used when you invoked *xmsim* originally. If you want to change the *xmsim* options and arguments, edit the text field. Click *Yes* to reinvoke the simulation. Click *No* to cancel the reinvoke and remain in the current simulation session.

If the *Prompt before reinvoke* option on the *Preferences* form is not set, the Reinvoke form does not appear, and *xmsim* is invoked with the same *xmsim* command-line options and arguments specified for the original run.

Exiting the Simulation with Tcl or SimVision

With Xcelium, there are multiple methods to exit the simulator using Tcl and SimVision:

- If you are using the Tcl command-line interface, use either the `exit` or `finish` command.

  The `exit` command is a built-in Tcl command. It halts execution and returns control to the operating system.

  The `finish` command also halts execution and returns control to the operating system. This command takes an optional argument that determines what type of information is displayed after exiting.

  - `0`: Prints nothing (same as executing finish without an argument).
  - `1`: Prints the simulation time.
  - `2`: Prints simulation time and statistics on memory and CPU usage.

- If you are using the SimVision analysis environment, there are two ways that you can stop simulating with SimVision:

  - Disconnect from the simulation. This leaves the simulation running but makes it unavailable from the SimVision user interface.

    To disconnect from a simulation, click the *Disconnect* button in the simulator tab of the Console window.

    You can reconnect to the simulation by choosing *File – Open Simulation* and selecting the simulation from the *Connect to Simulation Session* form.

  - Terminate the simulation. This stops the simulator. You cannot reconnect to the simulation after you have terminated it.

    To terminate a simulation, click the *Terminate* button in the simulator tab of the Console window.

You can also select *File - Exit SimVision*.

5

Extending Snapshots to Add Source Files

You can extend an elaborated snapshot to add additional source files by using the elaborator `-extendsnap` option. For instance, suppose that you have compiled source files for a DUT and have already generated a snapshot for it. You can then choose to compile new files with some code to test the DUT (HDL files or ***e*** files, for example) and re-elaborate the design to extend the first, or *primary*, snapshot. The elaborator uses the primary snapshot and the newly compiled files to create a unified snapshot. By using the `-extendsnap` option, you can avoid rewriting complex scripts or altering the verification environment.

- All compiled libraries used to build the primary snapshot and the `.pak` file must be available when elaborating the unified snapshot.
- Using the `-extendsnap` option does not perform incremental elaboration. A new snapshot is built by doing a full elaboration of the design.

  There is no performance gain, and memory consumption increases because a full snapshot is loaded in memory before elaboration starts again.

⚠ Do not use the `-extendsnap` option to extend a primary snapshot that has missing instantiations and that was elaborated using the `-partialdesign` option. In the current release, you cannot use the `-extendsnap` option to add files that make connections to missing instantiations in the primary snapshot.

The argument to `-extendsnap` is the name of the pre-elaborated (primary) snapshot:

```
-extendsnap snapshot_name
```

Extending Snapshots with XRUN

With *xrun*, include the `-extendsnap` option and the new source files on the command line to generate the unified snapshot.

For example, consider a scenario where you have compiled VHDL files for a DUT and have generated a snapshot, as follows:

```
xrun -c dut.vhd -top dut
```

Suppose that the name of the snapshot is `worklib.dut:dut_a`.

Now you want to add a Verilog testbench. The new file (`tb.v`) instantiates the DUT and is now the new top.

To extend the primary snapshot, create a unified snapshot, and simulate the design, use the following command:

```
xrun -extendsnap worklib.dut:dut_a tb.v -input sim.tcl
```

Here, because *xrun* can automatically detect Verilog tops, specifying the Verilog top with the `-top` option is not necessary.

⚠ In this example, the new snapshot is called `worklib.tb:v`. The name of the snapshot generated by the second elaboration must be different from the name of the primary snapshot specified with `-extendsnap`. An error is generated if the snapshot names are the same.

Extending Snapshots in Direct Invocation Mode

In direct invocation mode, the `-extendsnap` option is specified during the elaboration step when generating the unified snapshot.

For example, consider the same scenario above where you compile VHDL files for a DUT and generate a snapshot.

The *primary* snapshot is generated with the following two commands:

```
xmvhdl -messages dut.vhd
xmelab -messages dut
```

This generates the `worklib.dut:dut_a` snapshot.

To extend the snapshot, compile the Verilog testbench file and elaborate with `-extendsnap`.

For the elaboration step, both the primary snapshot and the Verilog top (`tb`) must be specified as arguments on the command line:

```
xmvlog -messages tb.v
xmelab -messages -extendsnap worklib.dut:dut_a tb
xmsim worklib.tb:module -input sim.tcl
```

Different Command-Line Options

If different command-line options have been used in the first elaboration (primary snapshot) and the second elaboration (elaboration with the `-extendsnap` option), the second elaboration uses the union of options specified for both elaborations. If options conflict, a warning is issued and the option specified in the second elaboration takes precedence. For example, if the DUT was elaborated with `-access +r` and the second elaboration with `-access +rwc`, the latter specification is used.

6

Verilog Module and UDP Resolution

One of the most important operations that occur during elaboration is binding (or linking). Binding is the process of selecting which design units are instantiated at each node of the hierarchy. Each Verilog module or user-defined primitive (UDP) that is instantiated in another, higher-level module is bound to a particular `[lib.]cell[:view]`.

You can control the binding of instances in several ways. The listed order of each section below reflects the order of precedence used by the elaborator, from lowest to highest.

- The Default Binding Mechanism
- Default Configuration Using a Library Map File
- The `uselib Compiler Directive
- The -binding Option

  This option is used to force the binding of a cell to a particular library and view. It overrides the default binding mechanism and the `` `uselib `` compiler directive.

- Verilog Configurations

  A Verilog configuration specifies the explicit rules to be used for binding each instance in the design.

⚠ The term *binding* refers to the process of selecting a particular `[lib.]cell[:view]` for each module or UDP that is instantiated in another, higher-level, module. These binding rules do not apply to the top-level module(s) that you specify on the command line.

The Default Binding Mechanism

The following are the default binding rules that the tool performs during elaboration in order of precedence:

1. For a particular module, if an instance has already been bound during the current elaboration, that same binding applies.

2. If no binding is found, the tool uses libraries that are listed with the `LIB_MAP` variable and, beginning with the library where the parent module was found, it searches the view names that are listed with the `VIEW_MAP` variable in order, beginning with the view that has the same view name as the parent module. If a view that has the same name as the view of the parent module is found, the tool uses that binding.
3. If no binding is found, the tool continues with the next view in the `VIEW_MAP` variable. It continues searching the views in order, wrapping around to the first view, if necessary.
4. If no binding is found, the tool moves on to the next library listed in the `LIB_MAP` variable and repeats the view search using the same steps above.
5. If no binding is found, the tool searches the library where the parent module was found to determine a possible binding.
    - If one binding exists, the tool uses it.
    - If more than one binding exists, the tool exits with an error.
6. If no binding exists, the tool searches all known libraries.
    - If one binding exists, the tool uses it.
    - If more than one binding exists, the tool exits with an error.
      The following example uses the `-makelib` command-line option to compile a module into a library named `lib_a1`.

      ```
      // a1.v
      module a;
      initial $display("from a1.v");
      endmodule
      ```

      ```
      % xrun -compile -makelib ./lib_a1 a1.v -endlib
      ```

      A second *xrun* command is then specified to compile another module with the same name into a different library named `lib_a2`.

      ```
      //a2.v
      module a;
      initial $display("from a2.v");
      endmodule
      ```

      ```
      % xrun -compile -makelib ./lib_a2 a2.v -endlib
      ```

The module `a` is instantiated in a Verilog source file `top.v`. The libraries `lib_a1` and `lib_a2` are defined in the `cds.lib` file, as shown:

```
# cds.lib
define lib_a1 ./lib_a1
define lib_a2 ./lib_a2

//top.v
module top
  a inst();
endmodule;
```

The following *xrun* command shows the error message that is generated when multiple bindings exist. All possible bindings are listed as part of the error message.

```
% xrun top.v
xrun: 16.11-p001: (c) Copyright 1995-2016 Cadence Design Systems, Inc.
file: top.v
        module worklib.top:v
                errors: 0, warnings: 0
                Caching library 'worklib' ....... Done
        Elaborating the design hierarchy:
                Caching library 'worklib' ....... Done
xmelab: *E,MULVLG: Possible bindings for instance of design unit 'a' in
'worklib.top:v' are:
        lib_a2.a:v
        lib_a1.a:v

  a inst();
       |
xmelab: *E,CUVMUR (./top.v,3|7): instance 'top.inst' of design unit 'a' is
unresolved in 'worklib.top:v'.
```

- If no binding exists, the tool exits with an error.

If no binding is possible, *xmelab* generates an error message similar to the following example:

```
xmelab: *E,CUVMUR: instance of module/UDP 'bot' is unresolved in 'worklib.top:module'.
```

If the `LIB_MAP` variable has not been defined, *xmelab* searches the libraries listed in the `cds.lib` file. The libraries are searched in order, beginning with the library in which the parent module was found, and wrapping around to the first library, if necessary.

If you have not defined the `VIEW_MAP` variable, xmelab searches only the default views (`module` and `udp`). If you define a `VIEW_MAP` variable, and you want the elaborator to search for the default views, you must have an entry in the `VIEW_MAP` variable for `module` and `udp`.

⚠ Use the `-libverbose` option to display binding messages as depicted in the example below:

```
% xmelab -libverbose top
```

The following three examples illustrate the binding rules. The following files are used in the examples:

```
# cds.lib file
DEFINE worklib ./worklib
DEFINE designlib ./designlib
DEFINE source ./source
```

```
# hdl.var file
DEFINE VIEW_MAP (.v    => behav, \
                 .rtl  => rtl, \
                 .gate => gate )
DEFINE LIB_MAP (./designlib => designlib, \
                ./source    => source, \
                +           => worklib )
```

```
// File top.rtl                  // File ./designlib/foo.v
module top ();                      module foo ();
   foo a ();                           ...;
   foo b ();                           ...;
endmodule                           endmodule
```

Example 1

In the first example, module `top` in `top.rtl` is compiled into `worklib.top:rtl`, and module `foo` in `./designlib/foo.v` is compiled into `designlib.foo:behav` using the definitions of the `LIB_MAP` and `VIEW_MAP` variables.

xmelab first searches the library in which the parent module (`top`) was found (`worklib`) for a view that has the same view name as the parent module (`rtl`). It then searches `worklib` for a `gate` or `behav` view. No binding is found, so *xmelab* moves to the next library that is listed with the `LIB_MAP` variable (`designlib`), where it looks first for a view called `rtl` and then for a view called `gate` before finding view `behav`.

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.

% xmvlog -nocopyright -messages top.rtl
```

```
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The VIEW_MAP variable maps files with a .v
;# extension to a view called behav. Compilation produces designlib.foo:behav.

% xmvlog -nocopyright -messages ./designlib/foo.v
file: ./designlib/foo.v
    module designlib.foo:behav
        errors: 0, warnings: 0

;# Elaborate the top-level module, top. Use the -libverbose option to display
;# messages. xmelab looks in worklib for the same view as the parent (rtl),
;# then for a gate or behav view. It then does the same view search in designlib.

% xmelab -nocopyright -messages -libverbose worklib.top:rtl
        Elaborating the design hierarchy:
Resolving design unit `foo' at `top.a'.
                Caching library `worklib' ....... Done
        library: `worklib' views: `rtl' `gate' `behav' -> not found
                Caching library `designlib' ....... Done
        library: `designlib' views: `rtl' `gate' `behav' -> found
Resolved design unit `foo' at `top.a' to `designlib.foo:behav'.
Resolved design unit `foo' at `top.b' to `designlib.foo:behav'.
...
         Writing initial simulation snapshot: worklib.top:rtl
```

Example 2

In the next example, module `top` in `top.rtl` is compiled into `worklib.top:rtl`. Module `foo` in `./designlib/foo.v` is compiled into `designlib.foo:foo` using the `-view` command-line option.

xmelab first searches the library in which the parent module (`top`) was found (`worklib`) for a view that has the same view name as the parent module (`rtl`). It then searches `worklib` for a `gate` or `behav` view. No binding is found, so *xmelab* moves to the next library that is listed with the `LIB_MAP` variable (`designlib`) and then to the third library (`source`).

xmelab then searches for a possible binding in the library in which the parent module (`top`) was found (`worklib`). When it does not find a possible binding, *xmelab* searches all known libraries. One view (`designlib.foo:foo`) is found in the library `designlib`.

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.
```

```
% xmvlog -nocopyright -messages top.rtl
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called foo.
;# Compilation produces designlib.foo:foo.

% xmvlog -nocopyright -messages ./designlib/foo.v -view foo
file: ./designlib/foo.v
    module designlib.foo:foo
        errors: 0, warnings: 0

;# Elaborate the top-level module, top. xmelab looks in worklib for the same
;# view as the parent (rtl), then for a gate or behav view. It then does the
;# same view search in designlib, and then in source. xmelab then searches
;# worklib for a possible binding. When xmelab does not find a possible binding,
;# it searches all known libraries. One view (designlib.foo:foo) is found in
;# library designlib.

% xmelab -nocopyright -messages -libverbose worklib.top:rtl
        Elaborating the design hierarchy:
Resolving design unit 'foo' at 'top.a'.
                Caching library 'worklib' ....... Done
        library: 'worklib' views: 'rtl' 'gate' 'behav' -> not found
                Caching library 'designlib' ....... Done
        library: 'designlib' views: 'rtl' 'gate' 'behav' -> not found
                Caching library 'source' ....... Done
        library: 'source' views: 'rtl' 'gate' 'behav' -> not found
Resolved design unit 'foo' at 'top.a' to 'designlib.foo:foo'.
xmelab: *W,CUSRCH: Resolved design unit 'foo' at 'top.a' to 'designlib.foo:foo' through
a global search of all libraries.
Resolved design unit 'foo' at 'top.b' to 'designlib.foo:foo'.
...
...
        Writing initial simulation snapshot: worklib.top:rtl
```

Example 3

In the third example, module `top` in `top.rtl` is compiled into `worklib.top:rtl`. Module `foo` in `./designlib/foo.v` is compiled twice using the `-view` command-line option: into `designlib.foo:foo` and into `designlib.foo:bar`.

Using the search mechanism described above, *xmelab* finds two possible bindings: `designlib.foo:foo` and `designlib.foo:bar`. *xmelab* generates error messages saying that it found multiple bindings and that no binding was possible.

```
;# Compile top.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.top:rtl.

% xmvlog -nocopyright -messages top.rtl
file: top.rtl
    module worklib.top:rtl
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called foo.
;# Compilation produces designlib.foo:foo.

% xmvlog -nocopyright -messages ./designlib/foo.v -view foo
file: ./designlib/foo.v
    module designlib.foo:foo
        errors: 0, warnings: 0

;# Compile ./designlib/foo.v. The LIB_MAP variable maps files in this directory
;# to the designlib library. The -view option creates a view called bar.
;# Compilation produces designlib.foo:bar.

% xmvlog -nocopyright -messages ./designlib/foo.v -view bar
file: ./designlib/foo.v
    module designlib.foo:bar
        errors: 0, warnings: 0

% xmelab -nocopyright -messages -libverbose worklib.top:rtl
     Elaborating the design hierarchy:
Resolving design unit 'foo' at 'top.a'.
                Caching library 'worklib' ....... Done
        library: 'worklib' views: 'rtl' 'gate' 'behav' -> not found
                Caching library 'designlib' ....... Done
        library: 'designlib' views: 'rtl' 'gate' 'behav' -> not found
                Caching library 'source' ....... Done
        library: 'source' views: 'rtl' 'gate' 'behav' -> not found
xmelab: *E,MULVLG: Possible bindings for instance of design unit 'foo' in
'worklib.top:rtl' are:
        designlib.foo:foo
        designlib.foo:bar
xmelab: *E,CUVMUR: instance 'top.a' of design unit 'foo' is unresolved in
'worklib.top:rtl'.
```

Related Topic

- Verilog Module and UDP Resolution

Default Configuration Using a Library Map File

You can use the *library map file* as a method for controlling the default binding of instances like Verilog modules and user-defined primitives (UDPs). The concept of the library map file was first introduced in the IEEE 1364-2001 standard. This file can contain library declarations, which map source files, or sets of source files, to libraries. For example:

```
library rtlLib  "top.v";
library aLib    "adder.v";
library gateLib "adder.vg";
```

In this example, the declarations specify that all design units in the file `top.v` are compiled into the library called `rtlLib`. Design units in the file `adder.v` are compiled into the library called `aLib`. Design units in the file called `adder.vg` are compiled into the library called `gateLib`.

Libraries listed in library declarations must be defined in the `cds.lib` file.

⚠ In addition to library declarations, a library map file can also contain configuration declarations in `config-endconfig` blocks. These declarations comprise an explicit set of rules that specify the exact source code description to be used to represent each instance in a design. See Verilog Configurations for more information.

To use the library mappings in a library map file, you must compile the source files with the `-libmap` command-line option to specify the name of the library map file. If you specify a library map file, the definition of the `LIB_MAP` variable in the `hdl.var` file, if any, is ignored.

If you then elaborate the design with the `-libmap` option, and if the library map file does not contain a configuration, the libraries listed in the library declarations are searched to bind instances. Libraries are searched beginning with the library where the parent module was found and wrapping around to the first library, if necessary.

Example 1

The following files are used in the example:

```
# cds.lib file
DEFINE worklib ./worklib
DEFINE designlib ./designlib
DEFINE source ./source
DEFINE rtlLib ./rtlLib
DEFINE aLib ./aLib
DEFINE gateLib ./gateLib
```

```
# hdl.var file
```

```
DEFINE LIB_MAP (./designlib => designlib, \
                ./source    => source, \
                +           => worklib )

// Library map file: lib.map
library rtlLib top.v;
library aLib adder.v;
library gateLib adder.vg;

// File top.v                  // File adder.v              //File adder.vg
module top();                   module adder(...);          module adder(...);
  adder a1(...);                  // rtl                      // gate-level
  adder a2(...);                  foo f1(...);                foo f1(...);
endmodule                         foo f2(...);                foo f2(...);
                                endmodule                   endmodule
module foo(...);
  // rtl                        module foo(...);            module foo(...);
endmodule                         // rtl                      // gate-level
                                endmodule                   endmodule

;# Compile the source files with the -libmap option. The LIB_MAP variable in
;# the hdl.var file is ignored, and the mappings in the library map file are used.
;# Design units in top.v are compiled into rtlLib.
;# Design units in adder.v are compiled into aLib.
;# Design units in adder.vg are compiled into gateLib.
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -nocopyright -libmap lib.map -libverbose \
       -top rtlLib.top:module -vlog_ext +.vg top.v adder.v adder.vg
```

If you are running in direct invocation mode, the commands for this example are as shown below:

```
% xmvlog -nocopyright -messages -libmap lib.map top.v adder.v adder.vg
file: top.v
    module rtlLib.top
        errors: 0, warnings: 0
    module rtlLib.foo
        errors: 0, warnings: 0
file: adder.v
    module aLib.adder
        errors: 0, warnings: 0
    module aLib.foo
        errors: 0, warnings: 0
file: adder.vg
```

```
    module gateLib.adder
        errors: 0, warnings: 0
    module gateLib.foo
        errors: 0, warnings: 0
```

```
;# Elaborate the top-level module, rtlLib.top:module.
;# xmelab searches the libraries in the library declarations in the library map
;# file. Libraries are searched beginning with the library where the parent module was
found and wrapping around to the first library, if necessary.
```

```
% xmelab -nocopyright -messages -libmap lib.map -libverbose rtlLib.top:module
        Elaborating the design hierarchy:
Resolving design unit 'adder' at 'top.a1'.
                Caching library 'rtlLib' ....... Done
        library: 'rtlLib' views: 'module' 'udp' -> not found
                Caching library 'aLib' ....... Done
        library: 'aLib' views: 'module' -> found
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:module'.
Resolving design unit 'foo' at 'top.a1@adder<module>.f1'.
        library: 'aLib' views: 'module' -> found
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:module'.
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:module'.
Resolved design unit 'adder' at 'top.a2' to 'aLib.adder:module'.
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'aLib.foo:module'.
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'aLib.foo:module'.
        Building instance overlay tables: .................... Done
        ...
        ...
        Writing initial simulation snapshot: rtlLib.top:module
```

Example 2

In Example 1, the libraries declared in the library declarations are searched to find a binding. By default, libraries are searched beginning with the library where the parent module was found.

You can use the `-libname` option to override the default library search order. For example, to override the default search order so that library `gateLib` is searched first, use `-libname gateLib`.

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -libmap lib.map -libname gateLib -libverbose \
       -top rtlLib.top:module -vlog_ext +.vg top.v adder.v adder.vg
```

If you are running in direct invocation mode, the commands for this example are as shown below:

```
% xmvlog -nocopyright -libmap lib.map top.v adder.v adder.vg
% xmelab -nocopy -messages -libmap lib.map -libname gateLib \
         -libverbose rtlLib.top:module
        Elaborating the design hierarchy:
Resolving design unit 'adder' at 'top.a1'.
                Caching library 'gateLib' ....... Done
library: 'gateLib' views: 'module' -> found
Resolved design unit 'adder' at 'top.a1' to 'gateLib.adder:module'.
Resolving design unit 'foo' at 'top.a1@adder<module>.f1'.
        library: 'gateLib' views: 'module' -> found
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'gateLib.foo:module'.
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'gateLib.foo:module'.
Resolved design unit 'adder' at 'top.a2' to 'gateLib.adder:module'.
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'gateLib.foo:module'.
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:module'.
        Building instance overlay tables: .................... Done
        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

Related Topic

- Verilog Module and UDP Resolution
- Specifying the Configuration in a Library Map File

The `uselib Compiler Directive

Use the `` `uselib `` compiler directive to override the default search mechanism for binding instances like Verilog modules and user-defined primitives (UDPs). You can either apply special simulator extensions to `` `uselib `` or you can use legacy Verilog-XL syntax.

`uselib Extensions

Syntax:

```
`uselib lib = library_name
`uselib view = view_name
```

A library and a view can be specified with one directive, as follows:

```
`uselib lib = library_name view = view_name
```

The library and the view can be specified in any order.

Each `` `uselib `` directive explicitly defines a library and/or view search that resolves the instances that follow it until the elaborator encounters another `` `uselib `` directive, which redefines the search. An empty `` `uselib `` directive or a `` `nouselib `` directive makes the preceding `` `uselib `` directives ineffective.

`uselib Syntax

The standard Verilog-XL syntax for `` `uselib `` and the extensions for the Xcelium simulator are shown in the following table:

XL Usage	Xcelium Simulator Extensions
`` `uselib dir = `` *`lib_directory_name`*	`` `uselib lib = `` *`library_name`* Based on the mapping of source directories to library names in the `LIB_MAP` variable.

`` `uselib file = `` *`lib_file_name`*	`` `uselib lib = `` *`library_name`*
	`view = ` *`view_name`*
	
	Based on the mapping of source directories to library names in the `LIB_MAP` variable and on the mapping of file extensions to view names in the `VIEW_MAP` variable.

`libext = ` *`file_extension`*	`` `uselib view = `` *`view_name`* Based on the mapping of file extensions to view names in the `VIEW_MAP` variable.

⚠ If you do not specify both the library and the view in the `` `uselib `` directive, the missing mapping is taken from the `hdl.var` file. If the mapping of library or view cannot be made using the `LIB_MAP` or `VIEW_MAP` variables, all libraries defined in the `cds.lib` file are searched.

If you have a Verilog-XL design and run *xmprep*, an `hdl.var` file is created in the current directory. This `hdl.var` file contains `LIB_MAP` and `VIEW_MAP` variables based on the information in your `` `uselib `` compiler directives. See *xmprep* for more information.

`uselib Examples

The following two examples show you how to use the `` `uselib `` compiler directive to force the binding of one particular instance of a module.

Example 1

The following files are used in the first example. Notice that the `` `uselib `` directive in the file `top.v` specifies both the library and the view. The instance `a` of module `foo` is bound to `source.foo:rtl`, as specified in the `` `uselib `` compiler directive.

```
// hdl.var file
DEFINE VIEW_MAP ( .v    => behav, \
                  .rtl  => rtl, \
                  .gate => gate )


DEFINE LIB_MAP ( ./designlib => designlib, \
                 ./source    => source, \
                  +          => worklib )


// File: source/top.v
module top ();

`uselib lib=source view=rtl
foo a();    // You want to use the rtl view in the library called source
`uselib

foo b();
bar c();

endmodule


           module foo                                  module bar
source.foo:behav compiled from foo.v         source.bar:behav compiled from bar.v
source.foo:rtl compiled from foo.rtl         source.bar:rtl compiled from bar.rtl
designlib.foo:rtl compiled with -work


;# Compile all .v and .rtl files. The LIB_MAP variable maps source files in the
;# source directory to the library called source. The VIEW_MAP variable maps
;# files with a .v extension to a view called behav and .rtl files to a view
;# called rtl.

% xmvlog -nocopyright -messages source/foo.v
```

```
file: source/foo.v
    module source.foo:behav
        errors: 0, warnings: 0

% xmvlog -nocopyright -messages source/bar.v
file: source/bar.v
    module source.bar:behav
        errors: 0, warnings: 0

% xmvlog -nocopyright -messages source/foo.rtl
file: source/foo.rtl
    module source.foo:rtl
        errors: 0, warnings: 0

% xmvlog -nocopyright -messages source/foo.rtl -work designlib
file: source/foo.rtl
    module designlib.foo:rtl
        errors: 0, warnings: 0

% xmvlog -nocopyright -messages source/bar.rtl
file: source/bar.rtl
    module source.bar:rtl
        errors: 0, warnings: 0

% xmvlog -nocopyright -messages source/top.v
file: source/top.v
    module source.top:behav
        errors: 0, warnings: 0

;# Elaborate the top-level module.
;# Instance foo a is bound to source.foo.rtl.
;# Instances foo b and bar c are bound to the behav views using the default
;# binding mechanism.

% xmelab -nocopyright -messages -libverbose top
        Elaborating the design hierarchy:
Resolving design unit 'foo' at 'top.a' (`uselib at ./source/top.v,3).
                Caching library 'source' ....... Done
        library: 'source' views: 'rtl' -> found
Resolved design unit 'foo' at 'top.a' to 'source.foo:rtl' (`uselib at
./source/top.v,3).
Resolving design unit 'foo' at 'top.b'.
        library: 'source' views: 'behav' -> found
Resolved design unit 'foo' at 'top.b' to 'source.foo:behav'.
Resolving design unit 'bar' at 'top.c'.
        library: 'source' views: 'behav' -> found
```

```
Resolved design unit 'bar' at 'top.c' to 'source.bar:behav'.
        Building instance overlay tables: .................... Done
...
...
        Writing initial simulation snapshot: source.top:behav
```

Example 2

The second example is identical to the first example, except that the `` `uselib `` directive specifies only the view. The following is the file `top.v`:

```
module top ();

`uselib view=rtl

foo a();
`uselib

foo b();
bar c();

endmodule
```

The view specified in the compiler directive is used. This overrides any view specified with the `-binding` option. However, because the library is not specified, the libraries specified in the `LIB_MAP` variable in the `hdl.var` file are searched in order for an `rtl` view for module `foo`.

In this example, the library `designlib` is searched first. Because there is a design unit called `designlib.foo:rtl`, this binding is used.

The following shows the output of the elaborator:

```
        Elaborating the design hierarchy:
Resolving design unit 'foo' at 'top.a' (`uselib at ./source/top.v,3).
                Caching library 'designlib' ....... Done
        library: 'designlib' views: 'rtl' -> found
Resolved design unit 'foo' at 'top.a' to 'designlib.foo:rtl' (`uselib at
./source/top.v,3).
Resolving design unit 'foo' at 'top.b'.
                Caching library 'source' ....... Done
        library: 'source' views: 'behav' -> found
Resolved design unit 'foo' at 'top.b' to 'source.foo:behav'.
Resolving design unit 'bar' at 'top.c'.
        library: 'source' views: 'behav' -> found
Resolved design unit 'bar' at 'top.c' to 'source.bar:behav'.
        Building instance overlay tables: .................... Done
...
```

```
...
        Writing initial simulation snapshot: source.top:behav
```

Related Topic

- Verilog Module and UDP Resolution

The -binding Option

Use the `-binding` option on the *xrun* or *xmelab* command line to force the binding of a cell to a particular library and view. The syntax is:

```
-binding [lib.]cell[:view]
```

This option overrides the default view and library search mechanism. The specified cell is bound to an explicit library and view that you specify.

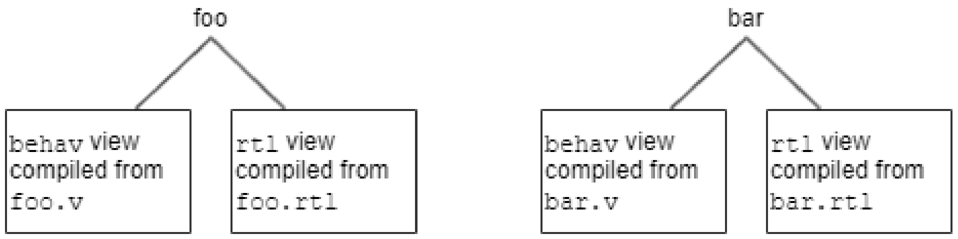
Remember that once the first instance has been resolved, all instances of the same Verilog module or user-defined primitive (UDP) are resolved the same way. Use a configuration to force different bindings for modules or UDPs with the same name. The following example shows you how to use the `-binding` option to force the binding of a cell to a particular view. The following files are used in the example:

```
// hdl.var file
DEFINE VIEW_MAP (.v => behav, \
.rtl => rtl, \
.gate => gate)

DEFINE LIB_MAP (./designlib => designlib, \
./source => source, \
+ => worklib)

// File top.v
module top();
foo a(); # You want to use the rtl
foo b(); # view for these instances.

bar c(); # You want to use the behav view for this instance.
endmodule
```


foo
bar
behav view compiled from foo.v
rtl view compiled from foo.rtl
behav view compiled from bar.v
rtl view compiled from bar.rtl

```
;# Compile top.v. The VIEW_MAP variable maps files with a .v extension to a
;# view called behav. Compilation produces worklib.top:behav.
```

`% xmvlog -nocopyright -messages top.v`

```
file: top.v
    module worklib.top:behav
        errors: 0, warnings: 0


;# Compile foo.v. Compilation produces worklib.foo:behav.
;# Compile foo.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.foo:rtl.

% xmvlog -nocopyright -messages foo.v foo.rtl
file: foo.v
    module worklib.foo:behav
        errors: 0, warnings: 0
file: foo.rtl
    module worklib.foo:rtl
        errors: 0, warnings: 0


;# Compile bar.v. Compilation produces worklib.bar:behav.
;# Compile bar.rtl. The VIEW_MAP variable maps files with a .rtl extension to
;# a view called rtl. Compilation produces worklib.bar:rtl.
```

`% xmvlog -nocopyright -messages bar.v bar.rtl`

```
file: bar.v
    module worklib.bar:behav
        errors: 0, warnings: 0
file: bar.rtl
```

```
    module worklib.bar:rtl
        errors: 0, warnings: 0


;# Elaborate top. To resolve the first instance of foo, the library where the
;# parent is located (worklib) is searched for a view that matches the view of
;# the parent (behav). After resolving this instance, any following instances
;# of foo receive the same binding.

% xmelab -nocopyright -messages -libverbose worklib.top:behav
        Elaborating the design hierarchy:
Resolving design unit 'foo' at 'top.a'.
                Caching library 'worklib' ....... Done
        library: 'worklib' views: 'behav' -> found
Resolved design unit 'foo' at 'top.a' to 'worklib.foo:behav'.
Resolved design unit 'foo' at 'top.b' to 'worklib.foo:behav'.
Resolving design unit 'bar' at 'top.c'.
        library: 'worklib' views: 'behav' -> found
Resolved design unit 'bar' at 'top.c' to 'worklib.bar:behav'.
        Building instance overlay tables: .................... Done
        ...
        ...
Writing initial simulation snapshot: worklib.top:behav


;# Elaborate top. Use -binding to force binding of instances of foo to the rtl view.
;# The first instance of foo is bound to the rtl view. Other instances of foo
;# use the same binding.

% xmelab -nocopyright -messages -libverbose -binding foo:rtl worklib.top:behav
        Elaborating the design hierarchy:
Resolved design unit 'foo' at 'top.a' to 'worklib.foo:rtl' (-binding switch).
Resolved design unit 'foo' at 'top.b' to 'worklib.foo:rtl' (-binding switch).
Resolving design unit 'bar' at 'top.c'.
                Caching library 'worklib' ....... Done
        library: 'worklib' views: 'behav' -> found
Resolved design unit 'bar' at 'top.c' to 'worklib.bar:behav'.
        Building instance overlay tables: ........... Done
        ...
        ...
        Writing initial simulation snapshot: worklib.top:behav
```

Related Topics

- Verilog Module and UDP Resolution
- Verilog Configurations

Verilog Configurations

Configurations were introduced in the IEEE 1800-2017 standard as a way to facilitate sharing of Verilog/SystemVerilog designs and/or design groups. A configuration is an explicit set of rules that specifies the exact source code description to be used to represent each instance in a design. In other words, the *binding* of instances such as Verilog modules and user-defined primitives (UDPs).

See Section 33 of the IEEE 1800-2017 standard for details on configuring the contents of a design by using configurations.

With Xcelium, a Verilog configuration can be specified as:

- An external file, called a *library map file*, which is included at elaboration time with the `-libmap` command-line option.
- Part of a *Verilog source file*.

Verilog Configuration Behavior Enhancement
From Xcelium 21.03 MAIN onwards, the behavior of bindings with Verilog configuration is strictly compliant as per the LRM. Currently, this LRM-compliant behavior is achieved using the `-vcfg_no_default_bind` option.

```
xmelab: *N,VCFGNDB: The behaviour of bindings with Verilog configurations will be
made strictly compliant as per LRM in an upcoming future release. Currently, LRM
compliant behaviour can be achieved using -vcfg_no_default_bind.
```

Owing to the LRM adherence, the following behavior is modified as per Chapter 33 of the IEEE Std 1800-2017:

- Does not use any other binding mechanism if failed to bind using Verilog configuration [33.6.1],
- Only looks for cell definition in libraries mentioned in the library list specified by the specific rule in the Verilog configuration [33.4.1.5],
- Only looks for cell definition in libraries mentioned in the library lists specified by the rules for parent instances in the Verilog configuration [33.4.1.5],
- Sub-configs shall not inherit library lists from parent configurations [33.4.2],
- Precedence of instance rules over cell rules in the configuration [33.4.1.3].
- Does not apply any Verilog configuration on the instantiation of SV bind statements written in the compilation unit [23.11].

Configurations specified in a library map file and compiled Verilog configurations can both be used to specify bindings for a mixed Verilog-VHDL design. See Configuring a Mixed-Language Design with a Verilog Configuration for more information.If the Verilog configuration is specified in both a library map file and in a Verilog source file, the simulator uses the configuration information in the library map file and ignores the configuration specified in the source files.

Deprecation Warning
The ability to add configurations in the libmap file is in deprecation, and will be removed in a future Xcelium release. Use or migrate to compiled configurations, which can be defined directly in the Verilog source and offer enhanced functionality.

Related Topic

- Verilog Module and UDP Resolution

Specifying the Configuration in a Library Map File

You can specify a configuration as an external *library map file* to bind Verilog modules and user-defined primitives (UDPs). This file is processed at elaboration time. To use the configuration in a library map file, you:

1. Create the library map file. This file contains one or more configuration blocks and each block contains a set of rules to use for binding. It uses the following format:

   ```
   config <name>;
     <binding_rule_1>;
     <binding_rule_2>;
     ...
   endconfig
   ```

2. Invoke *xmelab* with the `-libmap` option to specify the name of the library map file, and include the name of the actual configuration as an argument to this command.

   ```
   % xmelab -libmap library_map_file [other_options] configuration_name
   ```

   If running the simulator using *xrun*, use the `-top` option to specify the configuration.

   ```
   % xrun -libmap library_map_file -top name [other_options] source_files
   ```

   If the name of the configuration matches the name of a top-level design unit, you must append `:config` to the specified name. This identifies the argument as an explicit configuration name. For example:

   ```
   % xrun -libmap library_map_file -top name:config [other_options] source_files
   ```

⚠ In addition to configurations, a library map file can also contain library declarations, which can be used to control the compilation of design units in source files into specified libraries. If the library map file includes library declarations, in addition to configuration declarations, the library map file must be specified on the command line when you compile the source files (`xmvlog -libmap` or `xrun -libmap`).

The following examples show you how to use Verilog configurations. The `cds.lib` file, `hdl.var` file, and source code used for the examples is as follows:

```
# File: cds.lib
DEFINE worklib ./worklib
DEFINE rtlLib ./rtlLib
DEFINE aLib ./aLib
DEFINE gateLib ./gateLib
```

```
# File: hdl.var
DEFINE XRUNOPTS -default_ext verilog
DEFINE LIB_MAP (./top.v    => rtlLib, \
                ./adder.v  => aLib, \
                ./adder.vg => gateLib, \
                 +         => worklib)

DEFINE VIEW_MAP (.v => rtl, \
                 .vg => gate )
```

```
// File top.v                // File adder.v            // File adder.vg
module top();                 module adder(...);          module adder(...);
  adder a1(...);                // rtl                      // gate-level
  adder a2(...);                foo f1(...);                foo f1(...);
endmodule                       foo f2(...);                foo f2(...);
                              endmodule                   endmodule

module foo(...);
  // rtl                      module foo(...);            module foo(...);
endmodule                       // rtl                      // gate-level
                              endmodule                   endmodule
```

Example 1

In this example, the compilation of design units into libraries is determined by the `LIB_MAP` and `VIEW_MAP` variables in the `hdl.var` file.

```
% xmvlog -nocopyright top.v adder.v adder.vg
```

Compilation results in the following library structure:

Lib.Cell:View	From ...
`rtlLib.top:rtl`	`top.v`
`rtlLib.foo:rtl`	`top.v`
`aLib.adder:rtl`	RTL from `adder.v`
`aLib.foo:rtl`	RTL from `adder.v`
`gateLib.adder:gate`	Gate-level from `adder.vg`
`gateLib.foo:gate`	Gate-level from `adder.vg`

To use the RTL representations of `adder` and `foo`, which are in the library `aLib`, create a library map file (called `lib.map` in this example) that contains the following configuration:

```
config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;
endconfig
```

- The `design` statement names the library and cell of the top-level module in the design hierarchy.
- The `default liblist` clause defines the default library search order. In this example, the library `aLib` is searched before `rtlLib` and `gateLib`.

```
;# Elaborate the design.
;# Use the -libmap option to specify the name of the library map file.
;# Specify the name of the configuration as the argument to the xmelab command.

% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,1), from default liblist rule).
Resolved design unit 'adder' at 'top.a2' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,1), from default liblist rule).

        Building instance overlay tables: .................... Done
        ...
        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

To use the gate-level representations of `adder` and `foo`, specify the `gateLib` library first in the `default liblist` clause, as follows:

```
config cfg;
  design rtlLib.top;
  default liblist gateLib aLib rtlLib;
endconfig
```

If you are running in single-step invocation mode with *xrun*:

```
% xrun -libmap lib.map -top cfg -libverbose \
       -vlog_ext +.vg top.v adder.v adder.vg
```

The simulation completes successfully as long as the files `cds.lib` and `hdl.var` are located in your working directory. If you choose to run the simulation without these files, then you must run *xrun* using the `-makelib` option as shown.

```
% xrun -libmap lib.map -top cfg -libverbose -vlog_ext +.vg \
  -makelib rtlLib top.v -endlib -makelib aLib adder.v -endlib \
  -makelib gateLib adder.vg -endlib
```

⚠ If the library map file contains multiple configurations with the same name, the last configuration is used.

Example 2

This example is similar to Example 1, except that the library map file contains library declarations in addition to the configuration. The library declarations specify that:

- Design units in `top.v` are to be compiled into `rtlLib`.
- Design units in `adder.v` are to be compiled into `aLib`.
- Design units in `adder.vg` are to be compiled into `gateLib`.

```
library rtlLib  top.v;
library aLib    adder.v;
library gateLib adder.vg;

config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;
endconfig
```

The source files are compiled with the `-libmap` option to specify the name of the library map file.

```
% xmvlog -nocopyright -messages -libmap lib.map top.v adder.v adder.vg
file: top.v
    module rtlLib.top:rtl
        errors: 0, warnings: 0
    module rtlLib.foo:rtl
        errors: 0, warnings: 0
file: adder.v
```

```
    module aLib.adder:rtl
        errors: 0, warnings: 0
    module aLib.foo:rtl
        errors: 0, warnings: 0
file: adder.vg
    module gateLib.adder:gate
        errors: 0, warnings: 0
    module gateLib.foo:gate
        errors: 0, warnings: 0
```

```
% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'adder' at 'top.a2' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Building instance overlay tables: ................... Done

        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -libmap lib.map -top cfg -libverbose -vlog_ext +.vg top.v adder.v adder.vg
```

Example 3

To use the RTL representation of the `top.a1` adder (and its descendants), and the gate-level representation of the `top.a2` adder, use the following configuration:

```
library rtlLib  top.v;
library aLib    adder.v;
library gateLib adder.vg;

config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;
  instance top.a2 liblist gateLib;
endconfig
```

In this example, the `instance` clause specifies the specific instance to which the following `liblist` clause applies. Because the `liblist` is inherited, all descendants of `top.a2` inherit its `liblist`.

```
% xmvlog -nocopyright -libmap lib.map top.v adder.v adder.vg
% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'adder' at 'top.a2' to 'gateLib.adder:gate' (using Verilog config
'cfg' rule at (./lib.map,8)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).

        Building instance overlay tables: .................... Done|
        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -libmap lib.map -top cfg -libverbose -vlog_ext +.vg top.v adder.v adder.vg
```

Example 4

To use the RTL representation of `adder` from `aLib`, and the gate-level representation of `foo` from `gateLib`, use the following configuration:

```
library rtlLib  top.v;
library aLib    adder.v;
library gateLib adder.vg;

config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;
  cell foo use gateLib.foo;
```

```
endconfig
```

In this example, the cell clause selects all cells named foo and binds them to the gate-level representation in gateLib.

```
% xmvlog -nocopyright -libmap lib.map top.v adder.v adder.vg
% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).
Resolved design unit 'adder' at 'top.a2' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./lib.map,8)).

        Building instance overlay tables: .................... Done
        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -libmap lib.map -top cfg -libverbose -vlog_ext +.vg top.v adder.v adder.vg
```

If multiple `cell` clauses configure the same cell, a warning is issued and the first matching `cell` clause is used for binding.

Example 5

This example illustrates the use of a hierarchical configuration.

Suppose that you have developed the `adder` module in this example and have written a configuration that binds the instances of `foo` as follows:

```
// File adder.v
module adder(...);
// rtl
foo f1(...);        // Use rtlLib.foo for this instance
foo f2(...);        // Use gateLib.foo for this instance
endmodule
```

```
module foo(...);
// ...
endmodule
```

The configuration is as follows:

```
library rtlLib  top.v;
library aLib    adder.v;
library gateLib adder.vg;


config cfg2;
  design aLib.adder;
  default liblist gateLib aLib rtlLib;
  instance adder.f1 liblist rtlLib;
endconfig
```

When you elaborate the top-level of the design (`rtlLib.top`), you can then use the configuration `cfg2` shown above for one of the `adder` instances. For example:

```
// File top.v
module top();
  adder a1(...);        // Use aLib.adder for this instance
  adder a2(...);        // Use the rules in configuration cfg2 for this instance
endmodule

module foo(...);
  // rtl
endmodule
```

To specify this special set of binding rules for instance `a2`, you can bind the instance directly to the configuration `cfg2` by using an `instance` clause with a `use` clause. In this example, the configuration is as follows:

```
library rtlLib  "top.v";
library aLib    "adder.v";
library gateLib "adder.vg";

config cfg1;
  design rtlLib.top;
  default liblist aLib rtlLib;
// Use the rules in cfg2 to resolve the bindings for top.a2 and its descendants
  instance top.a2 use cfg2;
endconfig
```

```
config cfg2;
  design aLib.adder;
  default liblist gateLib aLib rtlLib;
  instance adder.f1 liblist rtlLib;
endconfig
```

In this example, the configuration `cfg1` specifies that the configuration `cfg2` is to be used to resolve the bindings of instance `top.a2` and its descendants. The `design` statement in `cfg2` defines the binding for the `top.a2` instance itself. The other rules in `cfg2` define the rules for binding the descendants of `top.a2`.

```
% xmvlog -nocopyright -libmap lib.map top.v adder.v adder.vg
% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg1
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg1' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg1' at (./lib.map,5), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg1' at (./lib.map,5), from default liblist rule).
xmelab: *W,DLCILIB: Library name 'rtllib' not found, defaulting to 'rtlLib'.  Please
see xmhelp on this error.
xmelab: *W,DLCILIB: Library name 'rtllib' not found, defaulting to 'rtlLib'.  Please
see xmhelp on this error.
Resolved design unit 'adder' at 'top.a2' to 'aLib.adder:rtl' (using Verilog config
'cfg1' rule at (./lib.map,9)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'rtlLib.foo:rtl' (using
Verilog config 'cfg2' rule at (./lib.map,15)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg2' at (./lib.map,12), from default liblist rule).

        Building instance overlay tables: .................... Done
        ...
        Writing initial simulation snapshot: rtlLib.top:rtl
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -libmap lib.map -top cfg1 -libverbose top.v adder.v adder.vg
```

⚠ DLCILIB Warning
The warning `DLCILIB` appears because the elaborator, after unsuccessfully searching for a particular cell in a library, downcases the library name string and makes another attempt for a case-insensitive search for that cell in the VHDL namespace. While searching the VHDL namespace, this warning gets issued. The tool does it only for those library names that are provided to the tool in the capital/camel case.

Related Topic

- Verilog Module and UDP Resolution
- Default Configuration Using a Library Map File

Specifying the Configuration in a Verilog Source File

In addition to supporting a configuration specified in a library map file that you include with the `-libmap` option, the simulator supports compiled Verilog configurations (configurations specified in a Verilog source file). Compiled Verilog configurations:

- Let you use compiler directives such as `` `define ``, `` `ifdef ``, and `` `include `` in the config block.
- Remove the requirement of using the `xrun -top` option to specify the top-level design unit. With a compiled configuration, the config block is treated as a separate design unit, and *xrun* can automatically detect the design tops.
- Make it easier to migrate designs from other simulators that support compiled configurations.

To compile a Verilog source file that contains a configuration, you must specify the `-compcnfg` option on the `xrun` or `xmvlog` command line. For example:

```
xrun -compcnfg [other_options] <source_files>
```

Or:

```
xmvlog -compcnfg [other_options] <source_files>
```

This option allows the use of `config`, `endconfig`, `design`, `cell`, and other Verilog keywords associated with configurations to be used as identifiers in the design.

⚠ You cannot use the `-rmkeyword` option to remove keywords associated with Verilog configurations when using the `-compcnfg` command-line option.

Hierarchical (or nested) configurations are supported, as they are with configurations specified in a library map file.

When using a compiled configuration, the configuration information is in a Verilog source file. You can still use a separate library map file to control the compilation of design units into libraries. The library map file is included at compilation time and at elaboration time with the `-libmap` option. The simulator, however, provides several ways to control the compilation of design units into libraries, and using a library map file is not required. See Compilation of Design Units into Library.Cell:View for more information on how you can compile design units into libraries.

The Verilog configuration can be in its own source file, or the config block can be included in a source file with other HDL code.

The configuration is always compiled with a view name of `config` even if the other HDL source in the same file is being compiled into a different view (using the `VIEW_MAP` or `VIEW` variable, the `-view` command-line option, the `` `view `` directive, and so on).

The following examples show you how to use compiled configurations. Below are the `cds.lib` file and `hdl.var` files used in the examples:

```
# File: cds.lib
DEFINE worklib ./worklib
DEFINE rtLib ./rtlLib
DEFINE aLib ./aLib
DEFINE gateLib ./gateLib

# File: hdl.var
DEFINE LIB_MAP (./top.v    => rtlLib, \
                ./adder.v  => aLib, \
                ./adder.vg => gateLib, \
                +          => worklib)

DEFINE VIEW_MAP (.v => rtl, \
                 .vg => gate)
```

Example 1

In this example, the following library map file is used to control the compilation of design units into libraries.

```
// File: lib.map
library rtlLib  top.v;
library aLib    adder.v;
library gateLib adder.vg;
```

The library declarations in this file specify that:

- Design units in `top.v` are to be compiled into `rtlLib`.
- Design units in `adder.v` are to be compiled into `aLib`.
- Design units in `adder.vg` are to be compiled into `gateLib`.

The Verilog configuration is included in the source file `top.v`.

```
// File: top.v
config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;
  instance top.a2 liblist gateLib;
endconfig

module top();
  adder a1();
  adder a2();

  initial
    $display("\tInstance is %m\tLibrary is rtlLib\tModule is top");
endmodule

module foo();
  initial
    $display("\tInstance is %m\tLibrary is rtlLib\tModule is foo");
endmodule
```

This configuration specifies that the elaborator is to use the RTL representation of the `top.a1` adder (and its descendants), and the gate-level representation of the `top.a2` adder. In this example, the `instance` clause specifies the specific instance to which the following `liblist` clause applies. Because the `liblist` is inherited, all descendants of `top.a2` inherit its `liblist`.

```
;# Compile the source files.
;# Use the -libmap option to include the library map file
;# and control the compilation of design units into libraries.
;# Use -compcnfg to allow keywords to be used as identifiers.

% xmvlog -nocopyright -messages -compcnfg -libmap lib.map top.v adder.v adder.vg
file: top.v
        config rtlLib.cfg:config          // View name of cfg is :config
        module rtlLib.top:rtl
        module rtlLib.foo:rtl
file: adder.v
        module aLib.adder:rtl
        module aLib.foo:rtl
file: adder.vg
        module gateLib.adder:gate
        module gateLib.foo:gate


;# Elaborate the design.
;# Use the -libmap option to specify the name of the library map file.
;# Specify the name of the configuration as the argument to the xmelab command.

% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./top.v,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./top.v,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./top.v,1), from default liblist rule).
Resolved design unit 'adder' at 'top.a2' to 'gateLib.adder:gate' (using Verilog config
'cfg' rule at (./top.v,4)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./top.v,4)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./top.v,4)).

        Building instance overlay tables: .................... Done
...
...
        Writing initial simulation snapshot: rtlLib.cfg:config

;# Simulate the design.
% xmsim -nocopyright rtlLib.cfg
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -default_ext verilog -compcnfg -libmap lib.map -top cfg -libverbose \
top.v adder.v adder.vg
```

Example 2

This example is the same as the previous example, except that the configuration has been removed from `top.v` and added to a separate file called `config.v`. The configuration is compiled into the library `rtlLib`.

```
// File: lib.map
```

```
library rtlLib  top.v, config.v;
library aLib    adder.v;
library gateLib adder.vg;

// File: config.v
config cfg;
  design rtlLib.top;
  default liblist aLib rtlLib gateLib;

  instance top.a2 liblist gateLib;
endconfig

;# Compile the source files, including config.v.
;# Use the -libmap option to include the library map file.
;# Use -compcnfg to allow keywords to be used as identifiers.

% xmvlog -nocopyright -compcnfg -libmap lib.map \
top.v adder.v adder.vg config.v

;# Elaborate the design.
;# Use the -libmap option to specify the name of the library map file.
;# Specify the name of the configuration as the argument to the xmelab command.

% xmelab -nocopyright -messages -libmap lib.map -libverbose cfg
        Elaborating the design hierarchy:
Resolved design unit 'adder' at 'top.a1' to 'aLib.adder:rtl' (using Verilog config
'cfg' at (./config.v,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f1' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./config.v,1), from default liblist rule).
Resolved design unit 'foo' at 'top.a1@adder<module>.f2' to 'aLib.foo:rtl' (using
Verilog config 'cfg' at (./config.v,1), from default liblist rule).
Resolved design unit 'adder' at 'top.a2' to 'gateLib.adder:gate' (using Verilog config
'cfg' rule at (./config.v,5)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f1' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./config.v,5)).
Resolved design unit 'foo' at 'top.a2@adder<module>.f2' to 'gateLib.foo:gate' (using
Verilog config 'cfg' rule at (./config.v,5)).

        Building instance overlay tables: .................... Done
...
...
        Writing initial simulation snapshot: rtlLib.cfg:config

;# Simulate the design.
```

```
% xmsim -nocopyright rtlLib.cfg
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -default_ext verilog -compcnfg -libmap lib.map -top cfg -libverbose \
top.v adder.v adder.vg config.v
```

Printing Concise Binding Reports Using Configuration Rules

Use the `-configverbose` option on the `xrun` or `xmelab` command line, instead of `-libverbose`, if you want a more concise report of instance bindings when using a Verilog configuration. The simulator prints the resolution of:

- `instance` USE rules
- `cell` USE rules
- the first hierarchical instance bound using a non-default `liblist` rule.

The output of both `-libverbose` and `-configverbose` is dumped to the *xrun* log file by default. However, you can use the `-save_libverbose` *`filename`* option to dump the output in a separate log file instead of the *xrun* log file.

For instance, using the example from the Specifying the Configuration in a Verilog Source File topic:

```
% xrun -default_ext verilog -compcnfg -libmap lib.map -top cfg -libverbose \
top.v adder.v adder.vg config.v
```

The instance resolution printed to the display changes as shown:

```
...
Resolved design unit 'adder' at 'top.a2' to 'gateLib.adder:gate' through liblist
gateLib (using Verilog config 'cfg' rule at (./config.v,6)).
...
```

In the next example, the `-makelib` option is used on the `xrun` command line to specify each library. The configuration is defined in the file `top.v`.

```
// File: top.v
config cfg;
       design top;
       default liblist rtlLib aLib;
       instance top.fooInst liblist aLib;
       cell dut liblist aLib;
endconfig

module top();
       adder adderInst();
       foo fooInst ();
       dut dutInst ();
endmodule



module foo();
       dut fooDutInst();
endmodule
```

```
// File adder.v
module adder ();
endmodule

module foo ();
endmodule

module dut();
       foo dutFooInst();
endmodule
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -makelib rtlLib top.v -endlib -configverbose \
-makelib aLib adder.v -endlib -compcnfg -top cfg
```

```
// Design Hierarchy
// File: top.v
        config rtlLib.cfg:config
        module rtlLib.top:rtl
        module rtlLib.foo:rtl

// File: adder.v
        module aLib.adder:rtl
        module aLib.foo:rtl
        module aLib.dut:rtl

// Output of -configverbose
Resolved design unit 'foo' at 'top.fooInst' to 'aLib.foo:rtl' through liblist aLib
(using Verilog configuration).
```

```
Resolved design unit 'dut' at 'top.dutInst' to 'aLib.dut:rtl' through liblist aLib
(using Verilog configuration).
```

Unused Verilog Configuration Rules

The tool displays the unused rules from your Verilog configuration, along with the reasons why those rules were left unused, in the elaboration logs. The following sections discuss these different scenarios and the possible warnings that you may encounter.

Invalid Hierarchical Instance Name in Instance Rules

You can use `instance` in a Verilog configuration for writing rules for a specific instance.

For instance, consider this design using the Verilog files `top.v` and `adder.v`.

```
// File: top.v
config cfg;
       design top;
       default liblist rtlLib aLib;
       instance top.fooInst liblist aLib;
       cell dut liblist aLib;
endconfig

module top();
       adder adderInst();
       foo fooInst ();
       dut dutInst ();
endmodule

module foo();
       dut  fooDutInst();
endmodule

module dut();
       adder dutAdderInst();
endmodule

// File: adder.v
module adder();
endmodule

module foo();
endmodule

module dut();
```

```
       foo dutFooInst();
endmodule
```

If you are running in single-step invocation mode with *xrun*, the command line for this example is as follows:

```
% xrun -makelib rtlLib top.v -endlib -makelib aLib adder.v -endlib -compcnfg -top cfg
```

```
// Design Hierarchy
// File: top.v
        config rtlLib.cfg:config
        module rtlLib.top:rtl
        module rtlLib.foo:rtl

// File: adder.v
        module aLib.adder:rtl
        module aLib.foo:rtl
        module aLib.dut:rtl
```

However, if the instance name associated is an invalid hierarchical name, then you may encounter one of the scenarios listed below.

1. Invalid instance name in the Verilog configuration's (`cfg`) `instance` rule.

   ```
   instance top.foo_Inst liblist aLib;
   ```

   Analyzing the design, it is evident that `top` does not have any instance `foo_Inst`. Hence, the tool shall give the following warning:

   ```
   xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtLib.cfg'
   (file: './top.v' line: '1') was/were not used to resolve any instances:
           instance top.foo_Inst liblist  aLib ;
                                |
   Hierarchical name component lookup failed at  "foo_Inst" in rtlLib.top:rtl
   ```

2. Valid instance name in the Verilog configuration (`cfg`), but the `design top` mentioned in the `design` statement does not have that instance under its scope.

   ```
   design top;
             instance dut.dutAdderInst liblist aLib;
   ```

   As the design top in the Verilog configuration has been set to `top` module, `dut` module is not visible under its scope and hence the rule is left unused.

   ```
   xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtlLib.cfg'
   ```

```
(file: './top.v' line: '1') was/were not used to resolve any instances:
        instance dut.b1 liblist  aLib ;
                  |
Hierarchical name component lookup failed at  "dut" in rtlLib.top:rtl
```

3. Sometimes, it might be clearly evident that an instance is a part of a module, but that module's instance had previously been bound to another module's definition.

```
instance top.fooInst liblist aLib;
instance top.fooInst.fooDutInst liblist aLib;
```

   The instances which are below its hierarchy are searched from the bound definition of the module.

```
Resolved design unit 'foo' at 'top.fooInst' to 'aLib.foo:rtl' (using Verilog
configuration).
```

   As `fooInst` instance has been bound to `aLib.foo:rtl`, there is no instance `fooDutInst` in the definition of the module. Hence, the tool shall display the following warning:

```
xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtlLib.cfg'
(file: './top.v' line: '1') was/were not used to resolve any instances:
        instance top.fooInst.fooDutInst liblist  aLib ;
                              |
Hierarchical name component lookup failed at  "fooDutInst" in aLib.foo:rtl
```

4. Similar to the previous use-case, even if the order of instance rules are interchanged, the result remains the same, as the binding starts from the beginning of the design, irrespective of the order of rules in a Verilog configuration.

```
instance top.fooInst liblist aLib;
instance top.fooInst.fooDutInst liblist aLib;

Resolved design unit 'foo' at 'top.fooInst' to 'aLib.foo:rtl' (using Verilog
configuration).
xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'worklib.cfg'
(file: './cnfg.v' line: '1') was/were not used to resolve any instances:
        instance top.fooInst.fooDutInst liblist  aLib ;
```

```
                         |
Hierarchical name component lookup failed at  “fooDutInst” in aLib.foo:rtl
```

5. If a Verilog configuration is compiled as part of the design, but it is not mentioned using `-top` on the command line.

   ```
   % xrun -makelib rtlLib top.v -endlib -makelib aLib adder.v -endlib -compcnfg
   ```

   The configuration is left unused in this case, and the simulator does not display a warning. You can change this behavior by adding either `-libverbose or -configverbose` to the command line. Then, the `VCFGUC` warning is printed to the display:

   ```
   xmelab: *W,VCFGUC: The Verilog configuration 'rtlLib.cfg' (file: './top.v' line:
   '1') was/were not used to resolve any instances.
   ```

6. If a Verilog sub-configuration is compiled as a part of the design and is also mentioned as a sub-configuration in a config rule.

   ```
   instance top.fooInst use rtlLib.sub_config:config;
   ```

   If the rule is left unused and the sub-configuration is also not used for any other binding, only the unused rule warning is displayed in default logs:

   ```
   xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtlLib.cfg'
   (file: './top.v' line: '1') was/were not used to resolve any instances:
           instance top.fooInst use rtlLib.sub_config:config;
   ```

   Note that if `-libverbose` or `-configverbose` are used on the command line, then the simulator also displays an additional warning:

   ```
   xmelab: *W,VCFGUC: The Verilog configuration 'rtlLib.sub_config:config' (file:
   './top.v' line: '1') was/were not used to resolve any instances.
   ```

Unknown Library in the LIBLIST Clause

You can use `liblist` selection clause in the Verilog Configuration to define an ordered set of libraries to be searched. If one or more libraries are invalid, then you may encounter any of the following scenarios.

1. Invalid library name in the `default liblist` rule.

```
default liblist rtlLib aLib WRONG_LIB;
```

Since, `WRONG_LIB` does not exist in the design, the tool shall display a warning with a list of all unknown libraries in a Verilog configuration.

```
xmelab: *W,VCFGUL: Unknown library(ies) found in Verilog configuration
'rtlLib.cfg' (file: './top.v' line: '1'):
       WRONG_LIB
```

2. Invalid library name in the `instance liblist` clause.

```
instance top.fooInst liblist aLib WRONG_LIB;
```

Since, `WRONG_LIB` does not exist in the design, the tool shall display a warning with a list of all unknown libraries in a Verilog configuration.

```
xmelab: *W,VCFGUL: Unknown library(ies) found in Verilog configuration
'rtlLib.cfg' (file: './top.v' line: '1'):
       WRONG_LIB
```

If this instance rule is left unused because of the unknown library present, then the tool shall also give the following warning:

```
xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtlLib.cfg'
(file: './top.v' line: '1') was/were not used to resolve any instances:
       instance top.fooInst liblist aLib WRONG_LIB;
              Unknown library(ies) found: WRONG_LIB
```

3. Invalid library name in the cell liblist clause.

```
cell foo liblist aLib WRONG_LIB;
```

Since, `WRONG_LIB` does not exist in the design, the tool shall display the following warning:

```
xmelab: *W,VCFGUL: Unknown library(ies) found in Verilog configuration
'rtlLib.cfg' (file: './top.v' line: '1'):
       WRONG_LIB
```

If this cell rule is left unused because of the unknown library present, then the tool shall also give the following warning:

```
xmelab: *W,VCFGUUR: The following rule(s) in Verilog configuration 'rtlLib.cfg'
(file: './top.v' line: '1') was/were not used to resolve any instances:
```

```
cell foo liblist aLib WRONG_LIB;
        Unknown library(ies) found: WRONG_LIB
```

7

Access to Simulation Objects

By default, at elaboration time, Xcelium marks all simulation objects in the design as having no read or write access and disables access to connectivity (load and driver) information. Turning off these three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve simulation performance. However, certain features force exceptions to this optimized mode. For instance:

- Using objects as arguments to *user-defined* system tasks or functions.

  These objects are automatically given read, write, and connectivity access. However, no access is given to objects used as arguments to *built-in* system tasks or functions by default. Using a construct that does not have a value as an argument (a module instance, for example) has no effect on access capabilities.

- Enabling PLI/VPI routines that modify delays with the `-anno_simtime` option.

  This option automatically provides read and write access to simulation objects at elaboration time.

- Enabling coverage with the elaborator `-coverage` option.

  This option automatically provides read access to specific objects in the design to ensure the accurate scoring of coverage.

You can also choose to change the default behavior of the tool customize your own access by:

- Using an access file
- Specifying global access with `-access`
- Specifying access for Verilog regs with `-accessreg`

Each method describes how to turn the access to simulation objects on or off. If access is turned on for an object, access to that object is guaranteed. However, turning access off for an object does not always mean that access to that object is restricted. Turning access off is a hint to the elaborator to not reference that object but, in some cases, the elaborator grants access to objects if there is no performance benefit to denying access. For instance, if signals at different levels of the hierarchy are part of the same net, and if access to the net is turned on at one level of the hierarchy, but turned off at another level of the hierarchy, access to all signals of the net is turned on because there is no performance benefit to turning it off at other levels of the hierarchy.

Related Topics

- Simulation Constraints for Snapshots with Default Access
- Guidelines for Access Control

Simulation Constraints for Snapshots with Default Access

Simulating a snapshot with limited visibility into simulation constructs has significant performance advantages, and this is the obvious choice in some situations. For example, simulating with default access is recommended if you are running regression simulations with self-checking tests. The disadvantage is that you have minimal debugging capability. You cannot access simulation objects from a point outside the HDL code, through Tcl commands, through PLI/VPI/VHPI, or using certain SimVision features. For instance:

- You cannot probe and generate waveforms for objects that do not have read access. You cannot display the value of these objects with Tcl commands such as `value` or `describe`. PLI/VPI/VHPI applications cannot get the values of objects tagged as having no read access.
- You cannot modify the values of objects that do not have write access. Write access is required for forcing or depositing values with Tcl `force` and `deposit` commands, or for a PLI/VPI/VHPI application to put values to objects.
- You cannot display the drivers of a particular wire or register if the object does not have connectivity access. Connectivity access is required for tracing signals in the Trace Signals sidebar, and for a PLI/VPI/VHPI application to scan for loads or drivers of an object.

To access design objects during simulation, you must use debug command-line options. These options provide visibility into different parts of a design and enable debugging features, but disable optimizations performed inside the simulator. Because these options slow down performance, you should provide the minimum amount of access possible by specifying only the kind of access you need for specific objects, instances, or portions of the design.

Default Mode and Tcl Commands

Many Tcl commands access and set values and probes on objects. Other commands provide load and driver information. With no read, write, or connectivity access to objects, these commands generate warning or error messages or they display output that does not include some objects or object values. For example:

- The `force` command prints an error if the object that is being forced does not have write access.
- The `deposit` command prints an error if the object to which a value is to be deposited does not have read/write access.

- The `value` command prints an error if any of the objects given as arguments do not have read access.
- The `probe` command prints an error if any of the objects given as arguments do not have read access. If the argument to this command is a scope, objects within that scope that do not have read access are excluded from the probe, and a warning message is printed. No waveforms are generated for these objects.
- The `drivers` command cannot display the drivers of a particular wire or register if the object does not have connectivity access.
- The `describe` command output does not include the value of an object if the object does not have read access.

Default Mode and PLI/VPI/VHPI Applications

If you run in the default mode, PLI/VPI/VHPI applications:

- Cannot get the values of objects tagged as having no read access.
  If the value of an object that does not have read access is requested, the PLI interface returns the value as a strong X in the requested format. If the requested format is decimal, integer, or time, the value is 0. If the format is in the form of a double, the return value is 0.0.

  If PLI 1.0 routines are being used, the ACC flag `acc_error_flag` is set to non-zero. This error can be detected in the VPI interface by calling `vpi_chk_error()` after a call to `vpi_get_value()` or by registering a callback for `cbPLIError`.
- Cannot put values to objects tagged as having no write access.
  Trying to put a value to a non-writable object is ignored and results in an error. If an event handle is requested from the `vpi_put_value()` routine, the return value is NULL.

  If PLI 1.0 routines are being used, the ACC flag `acc_error_flag` is set to non-zero. This error can be detected in the VPI interface by calling `vpi_chk_error()` after a call to `vpi_put_value()` or by registering a callback for `cbPLIError`.
- Cannot place value change callbacks on objects tagged as having no read access.
  Requesting a value change callback on a non-readable object is ignored and results in an error. Calling `acc_vcl_add()` on the object results in the ACC flag `acc_error_flag` being set to non-zero. Calling `vpi_register_cb()` on the object results in a NULL being returned. This

error can be detected using `vpi_chk_error()` immediately after the call to `vpi_register_cb()` or by registering a callback for `cbPLIError`.

- Cannot place callbacks on objects for force and release if the objects are tagged as having no read access.
- Cannot scan for loads or drivers of an object without connectivity access.
  If the object does not have read connectivity access, scanning for loads results in an error. Scanning for drivers on an object that does not have read access also generates an error.
- The simulated net for a net without read access is the original net.

Testing for the Visibility of an Object

The PLI 1.0, VPI, and VHPI interfaces allow an application to test for the accessibility of an object before working on it. The following macros are defined in `vxl_acc_user.h`:

- accWriteAccess
- accReadAccess
- accConnectivityAccess

You can use these macros with `acc_object_of_type()` and `acc_object_in_typelist()`.

The following properties are included in `vpi_user_cds.h`:

- vpiWriteAccess
- vpiReadAccess
- vpiConnectivityAccess

These Boolean properties can be used with the `vpi_get()` routine. They return TRUE if the object is read/write accessible and return FALSE otherwise.

These properties are accessible from any object that can have a value. In addition, the calls `vpi_get(vpiWriteAccess, NULL)` and `vpi_get(vpiReadAccess, NULL)` return FALSE if any object in the design has limited visibility. In Verilog-XL, these properties are accessible and always return TRUE.

The property `vhpiAccessP` is defined in `vhpi_user.h`. You can use this property to test the visibility of VHPI objects: `vhpi_get(vhpiAccessP, objHandle)`.

Controlling the Display of PLI Error and Warning Messages

By default, the simulator displays all warning and error messages that are generated when an error is detected due to a PLI read, write, or connectivity access violation. You can suppress the display of these access violation messages by using the `-plinooptwarn` command-line option (`xmsim -plinooptwarn` or `xrun -plinooptwarn`). If you use this option, a warning message is displayed only once when the first violation is detected. The message is displayed again if an access violation is detected after a reset or a restart.

Default Mode and the SimVision Debug Environment

The SimVision debug tools cannot display the value of objects that do not have read access. The current value for an object without read access is shown as "No Read Access". Trying to execute commands to show the value of an object, to set an object breakpoint, or to set a probe on objects without read access results in an error message.

Other operations require write access. For example, objects must have write access in order for you to deposit or to force a value.

Connectivity access affects the Trace Signals sidebar. For signals that do not have connectivity access, no action is taken. In other cases, an object may have connectivity access, but some drivers have been removed or have been optimized. In these cases, a warning is displayed telling you that the signal has no drivers.

Guidelines for Access Control

You can trade off simulation performance for debuggability using the access control mechanism in the simulator. By default, the simulator runs in maximum performance mode. This means that you have minimal debug access if you run in default mode. If you are going to use Tcl interactive commands or PLI/VPI/VHPI routines to debug and/or analyze the design, then some amount of access is required.

Cadence recommends following these guidelines for access control:

- General Access Control Guidelines
- Access Requirements for Tcl Interactive Commands
- Access Requirements for PLI/VPI/VHPI Functions and Callbacks

General Access Control Guidelines

Except for dynamic objects, no special access is required for viewing the hierarchy or for finding the names of objects (nets, regs, variables, scopes, and so on) in the design.

- Read access (`+R`) is required for probing nets, regs, and variables (including setting PLI callbacks) and getting the value of these objects.
- Write access (`+W`) is required to interactively set the value of simulation objects (depositing or forcing variables). Write access automatically provides read access.

  ⚠ With the exception of system tasks and functions, HDL constructs, such as a Verilog `force` statement, do not require any special access.

- Connectivity access (`+C`) is required to get driver and load information about a specific net, reg, or other variables. Connectivity access automatically provides write and read access.
- Compiling with the `-linedebug` option is required for setting breakpoints at source lines or for applying statement callbacks. Using this option automatically provides read, write, and connectivity access which can have a severe impact on performance.
- Elaborating with the `-anno_simtime` option (`xmelab -anno_simtime`) is required if you want to use PLI/VPI routines that modify delays at simulation time. Using this option automatically provides read, write, and connectivity access.

These general guidelines can be summarized as follows: Specify only the type(s) of access required for your debugging purposes, and try to set access controls on specific scopes, nets, regs, ports, and so on, by using an access file.

For example, if the only reason you need access is to save waveform data, use `xmelab -access +R`. If possible, use an access file to set read access only on the scopes or individual variables that you want to probe. The following access file, for example, can be used to save waveform data for ports only. The `DEFAULT` keyword specifies the default access for all instances, and the second statement specifies read access for ports.

```
DEFAULT -rwc
PATH ...<PORT> +r-wc
```

The following access file can be used to save waveform data for all nets and regs only:

```
DEFAULT -rwc
PATH ...<WIRE> +r-wc
PATH ...<REG> +r-wc
```

The following access file can be used if you want to save waveform data for ports only and perform scan tests by writing to scanflops tagged as cells with `` `celldefine ``:

```
DEFAULT -rwc
PATH ...<PORT> +r-wc
CELLINST worklib.dff_scan.module +rw-c
```

Access Requirements for Tcl Interactive Commands

The following table lists Tcl commands and the type of access they require:

Tcl Command	Access Requirement
`alias`	None
`call`	Depends on the C function or PLI/VPI/VHPI routine
`database`	None
`deposit`	+W
`describe`	None (+R to see object values)
`drivers`	+C
`finish`	None
`force`	+W
`help`	None

`probe`
`[-create]`
`-delete`
`-disable`
`-enable`
`-show`
	
	+R (for all desired objects)
	None
	None
	None
	None

`release`	None
`reset`	None
`restart`	None

`run`	None
`save`	None

`scope`
`-describe`
`-names`
`-sort`
`-drivers`
`-list`
`[-set]`
`-show`
	
	None (+R to see object values)
	None
	None
	+C
	None
	None
	None

`status`	None

`stop`
`[-create]`
`-condition`
`-continue`
`-delbreak`
`-execute`
`-if`
`-line`
`-name`
`-object`
`-skip`
`-time`
`-delete`
`-disable`
`-enable`
`-show`
	
	+R (if condition must include simulation object values)
	None
	None
	None
	+R (if condition must include object values)
	-linedebug
	None
	+R
	None
	None
	None
	None
	None
	None

`time`	None
`value`	+R
`version`	None

Access Requirements for PLI/VPI/VHPI Functions and Callbacks

PLI 1.0 Function	Access Requirement
`acc_vcl_add()`	+R
`acc_fetch_value()`	+R
`acc_fetch_paramval()`	+R
`acc_fetch_paramval_mtm()`	+R
`acc_set_value()`	+W
`acc_next_driver()`	+C
`acc_next_load()`	+C
`acc_append_delays()`	-anno_simtime
`acc_fetch_delays()`	-anno_simtime
`acc_replace_delays()`	-anno_simtime
`acc_append_pulsere()`	-anno_simtime
`acc_fetch_pulsere()`	-anno_simtime
`acc_replace_pulsere()`	-anno_simtime
`acc_set_pulsere()`	-anno_simtime

`misc_tf reason codes`
 `reason_paramvc`
	+R

VPI Function	Access Requirement
`vpi_get_value()`	+R
`vpi_put_value()`	+W
`vpi_get_delays`	-anno_simtime
`vpi_iterate(vpiDriver)`	+C

`vpi_iterate(vpiLoad)`	+C
`vpi_put_delays`	-anno_simtime

`vpi_register_cb()`
`cbValueChange`
`cbForce/cbRelease`
`cbAssign/cbDeassign`
`cbStmt`
	
	+R
	+R (on the expression)
	+R (on the register being assigned)
	-linedebug (on desired module)

VHPI Function	**Access Requirement**
`vhpi_get_value()`	+R
`vhpi_put_value()`	+W

`vhpi_iterator()`
`vhpiContributors`
`vhpiDrivers`
	
	+C
	+C

`vhpi_register_cb()`
`vhpiCbValueChange`
`cbStm`
	
	+R
	-linedebug

Using an Access File

An access file is a text file that lets you specify the type of access (read, write, and/or connectivity) that you want for particular instances and portions of the design.

To provide access to simulation objects during elaboration, do the following:

1. Create an access file. Xcelium supports two types of access file generation. You can:
    a. Manually write an access file, or
    b. Automatically generate an access file with `-genafile`.
2. Specify the access file on the command line. Once your access file is ready, you include it on the command line when you invoke the elaborator.

The access file can also contain PLI map file information. This map file associates user-defined system tasks and system functions with functions in a PLI application. See Including a PLI Map File for more information.

Guidelines to Write an Access File

To provide access to simulation objects, you can manually write an access file and then pass it on to the elaborator.

An access file consists of a set of lines, each of which specifies the desired access capability for instances in the design, hierarchical portions of the design, or value constructs used as arguments to user-defined system tasks or functions. Each line in the access file begins with a keyword. You can enter keywords in uppercase or lowercase. Each line must be terminated by a carriage return.

Some keywords require a hierarchical path argument. The syntax for specifying hierarchical paths is language-neutral. You can use either the VHDL path element separator (a colon), or the Verilog path element separator (a period) to separate the path elements. A path that starts at the VHDL top-level must begin with a colon.

Two wildcard extensions can be used in hierarchical path specifications:

- An asterisk (`*`) matches any instance in the current scope.
- Three dots (`...`) used as a suffix matches any instance in the hierarchy below the current scope.

The access specifications in the access file are similar to those used with the `-access` command-line option. Use `r`, `w`, and `c` for read, write, and connectivity access, respectively. Use a plus sign to turn on the specified access, and a minus sign to turn off the specified access. Objects that are given connectivity or write access are also given read access. In an access file, you can also specify `+d`. This specification gives all drivers of a net read access if the net has connectivity access. This provides a convenient way to provide access so that an application can scan for all drivers and read their values.

⚠ With the `-access` option, the plus sign is the default if you do not specify a plus or minus sign. In an access file, you must use the plus sign to turn on access.

You can also include comments in the file.

- Begin one-line comments with two slashes (`//`).
- Begin multiple-line comments with `/*` and end the comment with `*/`.

Here is a simple example access file:

```
// Read access for all instances in the 2nd levels of hierarchy
PATH *.*    +r-wc

// Read and write access for all instances of sub that are two levels under top
PATH top.*.sub +rw-c

//Read access, but no write access, for all instances below top.sub
PATH top.sub... +r-wc
```

Objects derive their access from an exact match. If there is no exact match for an object, the access is derived from the closest matching wildcard. If an object cannot be matched with a wildcard, the default access is used. For example, if you have the following two lines in an access file, `top.u1` and all instances inside `u1` have full access enabled because the first line specifies the more complete path.

```
PATH top.u1.* +rwc
PATH top.*.* -rwc
```

Access File Syntax

The syntax of the access file is as follows:

```
Access_file ::= Line
Line ::=
    DEFAULT Access
    | BASENAME [Path] [Access]
    | PATH Path Access
    | CELLINST Access
    | CELLLIB Cell Access
  |UDPPATH [FULL_PATH] [Access]
    | UDPMODULE [MODULE_NAME] [Access]
    | $UDTF Stfname [<REG> | <WIRE>] Access
    | INCLUDE File
    | Comments

Path ::=
    Name
    | Path.Name            NOTE: You can use the VHDL separator (:)
                                 or the Verilog separator (.).
    | Path...
    | Path...Key
    | Path.Key

Cell ::= A name in Lib.Cell:View format

Name ::= Legal Verilog or VHDL name

Stfname ::=
    Name of a user-defined system task or function
    | *

File ::= Path to an access file to include

Key ::=
    <REG>
    | <WIRE>
    | <INTEGER>
    | <TIME>
    | <REAL>
    | <PRIMITIVE>
    | <ASSIGN>
    | <EVENT>
    | <PORT>
    | <PORTIN>
    | <PORTOUT>
    | <PORTINOUT>
    | <SIGNAL>
    | <VARIABLE>

Access ::= Modifier Capability
```

```
Modifier ::= + | -
Capability ::= R | W | C | D | RW | RC | WC | RWC | CD

Comments ::=
    // text
    | /* text */
```

The keywords that you can use in an access file are listed in the table below:

`DEFAULT` *`Access`*	Customizes the default access for all instances. This overrides any access that you specify with the `-access` option.
`BASENAME [`*`Path`*`] [`*`Access`*`]`	Specifies the starting point for the path in all subsequent `PATH` statements. The specified path cannot contain any wildcard characters. If you do not specify a path, the `BASENAME` is set to null.

`PATH` *`Path Access`*

Specifies the access for all instances and constructs that match the path specification.

If a path ends with an object name, the named object gets the specified access. The following example turns on read and connectivity access for `top.foo.io`. The `+d` specifies that all drivers of a net with connectivity access also gain read access:

```
PATH top.foo.io +rcd-w
```

Objects from named blocks or HDL tasks and functions get their access from the containing scope if they do not have an exact match.

You can use wildcard characters in the hierarchical path argument. The two wildcard characters are:

- An asterisk (`*`) matches any instance at the current scope.
- Three dots (`...`) used as a suffix to match any instance in the hierarchy below the current scope.

The following example turns on read and write access for the top two levels of the design:

```
PATH * +rw-c          // Read, write access for top
level
PATH *.* +rw-c        // Read, write access for
second level
PATH ... -rwc         // No access to objects below
the second level
```

In addition to normal Verilog or VHDL hierarchical paths, such as `top.u1.foo` and `:U1:foo`, and hierarchical names using wildcard characters, there are several keys that let you specify access for particular constructs. If the path ends with a key, all objects of a class that matches the key get the specified access. These keys are:

- `<REG>` (Matches Verilog registers not declared as `integer`, `real`, or `time`)

- `<INTEGER>`, `<REAL>`, `<TIME>` (Matches the corresponding type of register declaration)
- `<WIRE>` (Matches Verilog wires)
- `<SIGNAL>` (Matches VHDL signals and ports)
- `<VARIABLE>` (Matches VHDL variables)
- `<PORT>` (Matches all Verilog wires and registers declared as module ports, and all VHDL ports)
- `<PORTIN>`, `<PORTOUT>`, `<PORTINOUT>` (Matches only ports of the corresponding mode)
- `<PRIMITIVE>` (Matches Verilog instances of primitives)
- `<ASSIGN>` (Matches Verilog blocking assignment statements)
- `<EVENT>` (Matches Verilog named events)

If an object can be matched by either a `<PORT*>` or `<WIRE>` key, the `<PORT*>` key is used.

`CELLINST` *`Access`*

Specifies the access for all instances that are tagged as cells and their subhierarchy. Instances are tagged as cells either with the `` `celldefine `` compiler directive or by using the `-y` or `-v` options with *xrun*.

The access that you specify with `CELLINST` overrides all wildcard paths that match a cell instance. However, an object in a cell instance matched by an exact path is annotated using the access from the exact path.

`CELLLIB` *`Cell Access`*	Specifies access using *`lib.cell:view`* format.

You can use the asterisk (`*`) as a wildcard character in any part of the *`lib.cell:view`* specification. This overrides the access for all matching modules. However, be aware that any access explicitly set with the `PATH` keyword overrides the access set using `CELLLIB` with this wildcard.

You can also use three dots (`...`) as a suffix to allow access down in the hierarchy for matching modules.

⚠ If a module contains all three rules, namely, `CELLINST`, `CELLLIB` and `CELLLIB...`, `CELLINST` gets the highest priority, followed by `CELLLIB` and then `CELLLIB...`

```
CELLLIB worklib.top:* ... +r
CELLLIB worklib.mid1:* +w → mid1 gets +w
```

If both, the three dots (`...`) and `CELLLIB` rules are specified for a module, then the `CELLLIB` rule takes precedence.

```
CELLLIB worklib.mid1:* ... -w
CELLLIB worklib.mid1:* ... +w → mid1 gets +w
```

If there is an exact match, a max union takes place.

```
CELLLIB worklib.mid1:* ... -w
CELLLIB worklib.mid1:* ... +w → mid1 gets +w
```

OR

```
CELLLIB worklib.mid1:*  -w
CELLLIB worklib.mid1:* +w → mid1 gets +w
```

`UDPPATH [`*`FULL_PATH`*`] [`*`Access`*`]`	Specifies access to a UDP-based instance. This keyword affects access on UDP output ports and wires connected to these respective ports only. The *FULL_PATH* is a full hierarchical path starting from the top module. You can use wildcard characters in the hierarchical path argument. The two wildcard characters are: • An asterisk (`*`) matches any instance at the current scope. • Three dots (`...`) used as a suffix to match any instance in the hierarchy below the current scope. The VHDL separator (`:`) and/or the Verilog separator (`.`) are allowed in the path statement.

`UDPMODULE [`*`MODULE_NAME`*`] [`*`Access`*`]`

Specifies access to a UDP-based module or primitive.

You can use wildcard characters when specifying a *MODULE_NAME*. The two wildcard characters are:

- An asterisk (`*`) matches any instance of the module at the current scope.
- Three dots (`...`) used as a suffix to match any module in the hierarchy below the current scope.

Precedence Rules

The `UDPPATH`/`UDPMODULE` keywords have slightly higher precedence than their counterparts. Specifically, `UDPPATH` > `PATH` and `UDPMODULE` > `CELLLIB`. In case of a conflict between `PATH` and `UDPPATH`, `UDPPATH` takes precedence; and between `CELLLIB` and `UDPMODULE`, `UDPMODULE` takes precedence. `UDPPATH` and `UDPMODULE` also take precedence over `-nocellaccess`.

Unsupported Rules

The following rules are not supported with `UDPMODULE`:

- Marking output register access for all the UDPs with the matching instance name `udp_inst` below the top module.

```
UDPMODULE *.udp_inst +rw
```

- Marking output register access for all the UDPs with the matching instance name `udp_inst` below the top module {2nd level only}.

```
UDPMODULE *.*.udp_inst +rw
```

- Marking output register access for all the UDPs instantiated inside the `m1` instance of the mid module.

```
UDPMODULE top.m1.* +rw
```

`$UDTF` *`Stfname`* `[<REG> | <WIRE>]` *`Access`*	Specifies the access for value constructs used as arguments to a user-defined system task or function. By default, these constructs are given full access (`+rwc`). Use this keyword to turn off different kinds of access. ⚠ The default access for constructs used as arguments to built-in system tasks and functions is `-rwc`. You cannot use `$UDTF` to modify this access.
`INCLUDE` *`File`*	Include the contents of the specified access file.

Access File Syntax Examples

Example 1:

Normally, the Xcelium default is no access. This example shows how to set read-only access as the default.

```
DEFAULT +r-wc
```

Example 2:

The following example uses the `BASENAME` and `PATH` keywords to provide access to multiple design objects.

```
BASENAME top.u1.foo         // Start all subsequent paths with top.u1.foo
PATH bar +rwc               // top.u1.foo.bar has +rwc
PATH u3 -rwc                // top.u1.foo.u3 has -rwc
BASENAME                    // Remove previous basename specification
PATH top.u3 +rwc            // top.u3 has +rwc
```

Example 3:

This example also uses the `PATH` keyword to provide access but adds the `<WIRE>`, `<REG>`, and `<PRIMITIVE>` keys to the specified hierarchical path statements.

```
PATH top.<WIRE> +rw-c         // Read, write access for wires in second level
PATH top.sub...<REG> +rw-c    /* Read, write access for regs in instances
                               below top.sub */
PATH top.sub.foo.<PRIMITIVE> +rw-c  // Read, write access for primitives in foo
```

Example 4:

The following example uses `CELLINST` to specify that all cells in the design can be fully optimized, while the upper levels have full debug access.

```
CELLINST -rwc      // No access to objects in cell instances
PATH ... +rwc      // Enable full access to objects above cells
```

Example 5:

This example uses the CELLLIB keyword to provide access using different *`lib`*`.`*`cell`*`:`*`view`* specifications.

```
CELLLIB worklib.m16:module +rw-c  // Read, write access for worklib.m16:module
CELLLIB worklib -rwc              // No access to objects in worklib
CELLLIB worklib.* +rwc            // Read, write, connectivity acces to all cells in 
worklib
CELLLIB worklib.m16:* +rwc        // Read, write, connectivity access to all views in 
worklib.m16
CELLLIB *.m16 +rwc                // If cell m16 exists in the libraries, set +rwc 
acess
```

Example 6:

The following example uses `CELLLIB` to turn off access to all objects in the library `asic`. Read access is then granted to a single register `r1`.

```
// Full access to all objects
PATH ... +rwc
// No access to objects in the library asic
CELLLIB asic -rwc
/* Read access to register r1 in the instance top.u1.as01, an instance of a part in the 
library asic */
PATH top.u1.as01.r1 +r-wc
```

Example 7:

The example below is of a module `top` that has two module instances `mid1` and `mid2`. The module `mid1` also has one instance of `bot`.

```
module top ();
reg c1;
   mid2 m21();
   mid1 m11 ();
endmodule

module  mid1 ();
reg mida;
```

```
   bot b11 ();
endmodule

module bot ();
reg a1;
```

You can add `CELLLIB` to your access file with three dots (`...`) to provide access to `top` and all instances `m21`, `m11`, and `b11` as shown:

```
CELLLIB worklib.top:* ... +C
```

Example 8:

The following examples show how to prove access to UDPs using the `UDPPATH` and `UDPMODULE` keywords.

```
UDPPATH top.a +rw    // If "a" is a UDP, provide access to the
                     // output register and the connected wires
UDPPATH top.* +rw    // Provide access to the output registers and connected
                     // wires of any UDPs instantiated inside top
UDPPATH top.*.* +rw  // Provide access to the output registers and connected
                     // wires of any UDPs instantiated below top (1st level only)
UDPPATH top… +rw     // Provide access to the output registers and connected wires of
all
                     // UDPs including and below top (as deep as the hierarchy goes)

UDPMODULE udp_mod +rw   // Provide access to output the register
                        // and connected wire of each instance "udp_mod".
UDPMODULE mymod.* +rw   // Provide access to the output registers and connected wires
                        // of all the UDPs instantiated inside MODULE mymod
UDPMODULE mymod.*.* +rw // Provide access to the output registers and connected wires
                        // of all the UDPs instantiated below MODULE mymod (1st level
only)
UDPMODULE mymod… + rw   // Provide access to the output registers and connected wires
of all
                        // UDP instances including & below MODULE mymod (as deep as
hierarchy goes)
```

Example 9:

By default, value constructs used as arguments to user-defined tasks or functions are given full access. This example shows how to specify customized access for the construct `$mytask`.

```
$UDTF $mytask +r-wc         // Read-only access for $mytask
$UDTF $mytask <REG> +w      // Write access for all registers
                            // in the module containing $mytask
$UDTF $mytask <WIRE> +r     // Read access for all wires
```

```
                                   // in the module containing $mytask
```

Optionally, you can use `*` to provide custom access to all user-defined system tasks and functions.

```
$UDTF * +r-wc
```

Access File Warning Messages

There are several conditions that result in warning messages. These include:

- If an access mode for an object is both enabled and disabled, access is enabled, and a warning is issued. For example, in the following access file, top.u1.u3 gets read access.

  ```
  PATH top.u1.u3 +r-wc
  ...
  ...
  PATH top.u1.u3 -rwc
  ```

- A warning is issued if an object is named in an access file, but is not used in the elaborated design. For example, a warning is generated if you have the following line in the access file, but `top.u1.u3` is not used in the design.

  ```
  PATH top.u1.u3 +r-wc
  ```

  No warning is generated if wildcards are used. For example, if you have the following line in the access file, no warning is issued if `b` is not found in the design.

  ```
  PATH top.*.b +r-wc
  ```

Related Topic

- Using an Access File

Generating an Access File Using -genafile

If you know that multiple simulation runs require the same types of access on the same objects, you can automatically generate an access file by including the `-genafile` elaborator-time option using *xrun* or *xmelab*. When you simulate, the objects that are accessed by Tcl commands or by a PLI application are monitored along with the types of access required for each object. When the simulation exits, an access file is created with the specified filename.

```
% xrun -genafile access_file source_files
```

Or:

```
% xmelab -genafile access_file [lib.]cell[:view]
```

For instance, consider a scenario where you generate an access file named `access.txt`. You can then include this access file in subsequent runs with the `-afile` option.

```
% xrun -afile access.txt source.v
```

Or:

```
% xmelab -afile access.txt worklib.top
```

If you use `-genafile`, any request for access reduction with the `-access` or `-afile` command-line options is ignored. The `describe` command does not affect the specifications inserted into the access file.

> If you want to reinvoke a simulation in SimVision, select *Edit – Preferences* and make sure that the *Prompt before reinvoke* option is set. When you then select *Simulation – Reinvoke Simulator*, the *Reinvoke* form appears and you can edit the text field that contains your original `xrun` command-line options to change `-genafile` to `-afile`.

Related Topic

- Using an Access File

Specifying the Access File on the Command Line

After you create your access file, you must then pass it on to the elaborator to provide access to simulation objects.

To specify the access file on the command line, use the `-afile` option.

For *xrun* mode, the syntax is as shown:

```
% xrun -afile access_file source_file
```

Or, when you invoke the elaborator in direct invocation mode:

```
% xmelab -afile access_file [lib.]cell[:view]
```

For example:

```
% xrun -afile afile.af source.v
```

Or:

```
% xmelab -afile afile.af worklib.top
```

You can use more than one `-afile` option to include multiple access files. If you use multiple `-afile` options, the contents of the different files are combined before the access is actually set. For example:

```
% xmelab -afile afile1.af -afile afile2.af worklib.top
```

Including a PLI Map File

In addition to specifying access to simulation objects, an access file can also contain PLI map file information. A PLI map file associates user-defined system tasks and system functions with functions in a PLI application. The file contains a line for each user-defined system task or system function your application needs. In each line, you specify:

- The name of the system task or system function.
- Additional specifications for the system task or system function.

  For a user-defined system function, you must specify the size of the return value.

  Other, optional, specifications include the name of the call function, the name of the check function, the name of the misc function, and the data value passed as the first argument to the call, check, and misc routines.

The PLI map file information can be:

- Included in an access file, which is specified with the `xmelab -afile` option.
- Created as a separate file. You can include this file at elaboration time using the `-afile` option, or at simulation time with the `-plimapfile` option. If passed at elaboration time, the system tasks and functions defined in the file are known to both *xmelab* and *xmsim*. If passed at simulation time, the system tasks and functions defined in the file are known only to *xmsim*.

See the section "Using a PLI/VPI Map File" in the chapter "Using VPI" in the *VPI User Guide and Reference* for details on the PLI map file.

Related Topic

- Using an Access File

Specifying Global Access Using -access

Use the elaborator-time option, `-access`, with *xrun* or *xmelab* to turn on read, write, or connectivity access for all simulation objects in a design.

⚠ Providing read, write, and connectivity access to all objects in the design can have a severe impact on performance. Set the minimum access necessary for your debugging purposes. For example, if the only reason you need access is to save waveform data, use the option to provide only read access. If you only need to save waveform data for specific instances or portions of the design, specify read access for those portions of the design in an access file, and include the access file with the `-afile` option.

Objects that are given write access are also given read access. Objects that are given connectivity access are given write and read access.

Syntax:

```
-access [+] [-] access_specification
```

The *`access_specification`* argument can be:

- `r` (read access)
- `w` (write access)
- `c` (connectivity access)
- Any combination of the three access types

Use the plus sign to turn on the specified access. This is the default if no plus or minus sign is used with `-access`. Use the minus sign to turn off the specified access.

The `+` and `-` options apply to all subsequent `r`, `w`, or `c` specifications until the next `+` or `-`.

Examples:

- Read access only:
  `-access +r` (same as `-access r`)
- Write access:
  `-access +w` (same as `-access w`)

  Objects given write access are also given read access.
- Read/Write access:

`-access +rw` (same as `-access +r+w` and `-access rw`)

- Connectivity access:
  `-access +c` (same as `-access c`)

  Objects given connectivity access are also given write and read access.

You can also use multiple `-access` options. For example,

-access +r -access +c

⚠ Objects that are given connectivity access are given write access, and objects that are given write access are given read access. With the following set of options, all objects are given connectivity access and, therefore, write and read access.

```
-access +c -access -rw
```

Use the following general rules when deciding what kind of access you want to specify:

- Read access is required if you want to probe objects in the design and generate an SHM, VCD, or EVCD database. This lets you use the SimVision waveform viewer to view waveforms, and most Tcl commands and SimVision features. Read access is also required for getting signal values with, for example, the `value` or `describe` command, or `vpi_get_value()`.
- Write access is required if you want to deposit values using, for example, the `force` or `deposit` command, or `vpi_put_value()`.
- Connectivity access is required in order to show load or driver information. This kind of access is required, for example, by the drivers command and by the Trace Signals sidebar.

Specifying Access for Verilog regs

The `-access` option sets the visibility access for all objects in a design during elaboration. Use the `-accessreg` option to set the access for Verilog regs only during elaboration.

The `-accessreg` option has the same behavior as the `-access` option, except that it applies access only to the following Verilog object types:

- `reg`
- `integer`

- `real`
- `time`
- `event`

If both `-access` and `-accessreg` are included on the same *xrun*/*xmelab* command line, the read, write, or connectivity access specified by the `-accessreg` option is used for the above objects.

8

Managing Libraries Using the -v/-y Scheme

Normally, Xeclium compiles an entire design using the `library.cell:view` approach; however, the `-v` and `-y` library management scheme is the former scheme for binding and library management using Xcelium tools and some legacy designs continue to depend on this method. Be aware that this scheme is not mentioned in any Verilog or SystemVerilog LRM.

⚠ This method is only available when using Verilog and SystemVerilog HDL designs.

Using Library Files

To use a library file, specify the `-v` command-line option with the name of the library file that you want to use. The *xrun* utility scans this file for module and user-defined primitive (UDP) definitions that cannot be resolved in the specified source files and parses them when a match is found.

The following example shows how to use the `-v` command-line option:

```
% xrun source1.v -v libfile.v
```

Using Library Directories

To use a library directory, specify the `-y` command-line option with the path to the library directory. The *xrun* utility scans this directory for files containing definitions of modules or primitives that are unresolved in the specified source file.

The following example shows how to use the `-y` command-line option:

```
% xrun source1.v -y /usr/me/proj/lib/cmos +libext+.v
```

Files in library directories may contain just one module or UDP definition, or they may be complete hierarchies. If they are hierarchical, then the top level of the hierarchy should be the first module declared in the file.

Library directory files are not scanned unless they have the same name as an unresolved module or UDP that has been instantiated within the normal source text.

Checking the Syntax of Library Files

With *xrun*, modules and user-defined primitives are not parsed unless they have the same name as an unresolved module or UDP that has been instantiated within any normal source files (or other library files) specified. Therefore, syntax and semantic errors in those modules and UDPs are not detected. To completely check library file errors, you can have *xrun* parse all definitions within a library file by invoking *xrun* without the `-v` option as shown:

```
% xrun source1.v libfile.v
```

Related Topics

- Xcelium Library and Directory Structure
- Library Scan Precedence
- Compiling -v/-y Modules into a Single Library
- Specifying File Extensions in Library Directories

Library Scan Precedence

When *xrun* finds an instance of a module or user-defined primitive (UDP) that cannot be resolved in the source description files, it scans for a definition in the library files or directories that you specify on the command line. Once *xrun* finds a definition, it resolves the reference and ignores all subsequent definitions of the module or UDP that it encounters.

If your library files or directories contain multiple modules or UDPs with the same name, then the scan precedence *xrun* uses to find definitions becomes important.

The *xrun* command has three possible methods of scan precedence:

default scan precedence	If an unresolved instance is in a source file, *xrun* automatically scans the library files/directories in the order in which they are entered on the command line.
+liborder scan precedence	Enables you to direct the compiler to start a library file/directory search according to where the first instance of an unresolved module is detected.
+librescan scan precedence	Enables you to specify the order in which *xrun* scans the library files/directories to resolve all undefined module and UDP instances from both source files and libraries.

Default Scan Precedence

The default scan precedence that *xrun* uses to search for definitions in library files or directories is as follows:

- If the unresolved instance is in a source file, *xrun* scans the library files or directories in the order in which they are entered on the command line. This process begins with the left-most library file or directory, no matter where the source file appears on the command line. After *xrun* scans the left-most library, it scans the others in the order in which they appear.
- If the unresolved instance is in a library file or directory, *xrun* scans in the following:
  a. The library file or a directory that contains the unresolved instance.
  b. If the instance remains unresolved, *xrun* scans the remaining libraries, beginning with the one that follows the library containing the unresolved instance.

     This process continues in a circular manner—that is, *xrun* scans the libraries as they follow on the command line and then "wraps around" to the left-most library until it visits each one.

When you use the default scan precedence, you should enter library files and directories on the command line in the order in which you want them scanned.

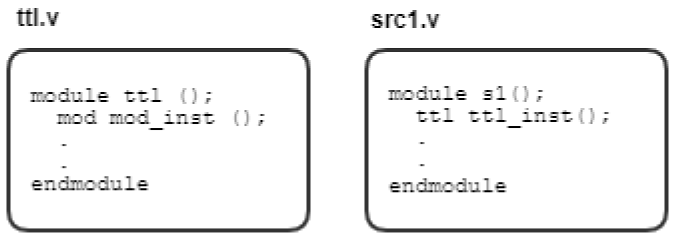
Consider the following example:

```
% xrun src1.v -y /usr/lib/NMOS src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS
```

In this example, if source file `src2.v` instantiates a module that is not defined in source files `src1.v` or `src2.v`, *xrun* scans for a definition first in `/usr/lib/NMOS`, then in `/usr/lib/TTL/ttl.v`, and finally in `/usr/lib/CMOS`.

However, if the module is neither instantiated nor defined in either source file, but is instead instantiated in the library file `/usr/lib/TTL/ttl.v`, then *xrun* looks first for a definition in `/usr/lib/TTL/ttl.v`, and then in `/usr/lib/CMOS`, and finally in `/usr/lib/NMOS`.

For instance, consider the following figure:



```
module ttl ();
  mod mod_inst ();
  .
  .
endmodule
```

```
module s1();
  ttl ttl_inst();
  .
  .
endmodule
```

A module `ttl` defined in `/usr/lib/TTL/ttl.v` instantiates another module `mod`, which is neither instantiated nor defined in either source file. The software searches for `mod` in the order described above because module `s1` in source file `src1.v` instantiated `ttl`.

If you have multiple modules or UDPs with the same name in your libraries, you can control how *xrun* resolves undefined instances with the default scan precedence in the following ways:

- Enter library files and directories on the command line in the order in which you want them scanned for module and UDP definitions. For example:

  ```
  % xrun src1.v src2.v -y /usr/lib/new -y /usr/lib/old
  ```

  In the previous example, *xrun* first scans the definitions of all undefined modules and UDPs from the source files in `/usr/lib/new`. If any instances remain unresolved, *xrun* scans `/usr/lib/old`. This method may not work if a library module contains an unresolved instance of another module or UDP.

  If your libraries contain multiple modules with the same name that have unresolved instances, you should use the `+librescan` option.

- Use library directory filename extensions and the `+libext+` command-line option. As described in Specifying File Extensions in Library Directories, *xrun* searches for the extensions listed after `+libext+` from left to right in the order in which you specify them on the command line.

  To make one set of files within a library directory take precedence over another set, use different extensions on the two sets and list the extension of the dominant set immediately after the `+libext+` option. For example, a library named `/usr/lib` contains multiple modules with the same name. The newer versions of these modules have a `.v2` extension; the older versions have a `.v1` extension. To give the modules with the `.v2` extension higher precedence, enter the following command:

  ```
  % xrun src1.v src2.v -y /usr/lib +libext+.v2+.v1+
  ```

  The module and UDP definition library files with the `.v2` extension are used to resolve undefined instantiations first because the `.v2` extension appears first after the plus option.

+liborder Scan Precedence

The `+liborder` option allows you to order a library search according to where the first instance of an unresolved module is detected. The compiler can find instances of unresolved modules in source descriptions or in library files. When the origin is a source description, `+liborder` directs the compiler to start searching in the library file or directory immediately following that source file. However, if the module instance is detected in a library file, `+liborder` initiates the search in that library.

In either case, if the module remains unresolved, `+liborder` directs *xrun* to scan the remaining library files and directories in a circular order—that is, to scan libraries as they follow on the command line and then “wrap around” to preceding libraries that *xrun* has not yet visited.

For example, suppose that you add the `+liborder` option to the command line, as in the following example:

```
% xrun source1.v -v lib1.v source2.v -v lib2.v +liborder
```

Now suppose that the compiler detects an instance of the unresolved module `dff` in the source description `source2.v`. To resolve the module, *xrun* first searches for a description of `dff` in `lib2.v`. If the module remains unresolved, the search continues in `lib1.v`.

For an instance of `dff` that appears in `source1.v`, the compiler searches for the module definition first in `lib1.v`, and then in `lib2.v`.

Now consider the following command:

```
% xrun src1.v -y /usr/lib/NMOS \
src2.v -v /usr/lib/TTL/ttl.v -y /usr/lib/CMOS +liborder +libext+.a+.b++
```

Suppose that the compiler finds an unresolved module `ttl` in the library `/usr/lib/TTL/ttl.v`. The `+liborder` option directs the search for a description of `ttl` in the following manner:

1. Scan the library file `/usr/lib/TTL/ttl.v`.
2. Scan library files `ttl.a`, `ttl.b`, and `ttl` in the order listed in the directory `/usr/lib/CMOS`, if `ttl` remains unresolved.
3. Scan library files `ttl.a`, `ttl.b`, and `ttl` in the order listed in the directory `/usr/lib/NMOS`, if `ttl` is still unresolved.

Suppose that the description of `ttl` is detected in one of the library files in the directory `/usr/lib/CMOS`. However, in the process of resolving `ttl`, *xrun* detects an instance of a new unresolved module `ttl_buf` in the description of `ttl`. The `+liborder` option directs the compiler to search for a description of `ttl_buf` in the following manner:

1. Scan the library files `ttl_buf.a`, `ttl_buf.b`, and `ttl_buf` in the order listed in the directory `/usr/lib/CMOS`.

2. Scan the library files `ttl_buf.a`, `ttl_buf.b`, and `ttl_buf` in the order listed in the directory `/usr/lib/NMOS`, if `ttl_buf` remains unresolved.
3. Scan the library file `/usr/lib/TTL/ttl.v`, if `ttl_buf` is still unresolved.

The option `+liborder` is especially useful for resolving multiple descriptions of modules or primitives that have the same name. For example, suppose you want to compare the timing performance of two modules called `ttl_fast`, each from a different library.

To make this comparison, you must create two source descriptions—`test_ttl_1.v` and `test_ttl_2.v`—that each contain instantiations of `ttl_fast`. It is critical that the compiler resolve all instances of `ttl_fast` in `test_ttl_1.v` from one library and all instances of `ttl_fast` in `test_ttl_2.v` from the other library. The following command accomplishes this task:

```
% xrun test_ttl_1.v -v lib1.v test_ttl_2.v -v lib2.v +liborder
```

⚠ You cannot use the `+librescan` option with the `+liborder` option.

+librescan Scan Precedence

The `+librescan` option allows you to specify one order in which *xrun* scans library files and directories to resolve all undefined module and UDP instances from both source files and libraries. The behavior of `+librescan` depends on the location of the undefined instance—that is, it depends on whether the instance is located in a source file, a library file, or a file within a library directory.

- *The Source File Scenario*

  When the undefined instance is located in a source file, `+librescan` acts the same as the default scan precedence—scanning begins with the left-most library on the command line and continues through the remaining libraries from left to right.

- *The Library File Scenario*

  When the undefined instance is located in a library file, and `+librescan` is in effect, *xrun* does not continue to search the library file for the matching definition; instead, it begins to scan through the left-most library on the command line, and then scans the remaining libraries in the order in which they appear.

- *The Library Directory Scenario*

  Finally, when the undefined instance is located in a library directory file, and `+librescan` is in effect, *xrun* scans the library directory file first to try to resolve the instance. If it remains unresolved, *xrun* then begins to scan the left-most library on the command line, followed by the remaining libraries in the order that they appear.

The following example includes three library files: `lib.orig.v`, `lib.revised.v`, and `lib.latest.v`. Library `lib.orig.v` contains the original versions of all the modules. Library `lib.revised.v` contains revised versions of many of the modules from `lib.orig.v`. The library `lib.latest.v` contains the latest revisions of just a few of the modules. To resolve all undefined instances with the most up-to-date modules, use the following command:

```
% xrun source.v -v lib.latest.v -v lib.revised.v -v lib.orig.v +librescan
```

In this example, if `lib.orig.v` instantiates a module that is defined in that library, *xrun* looks for a definition of the module instance first in `lib.latest.v`, then in `lib.revised.v`, and finally in `lib.orig.v`.

⚠ You cannot use the `+librescan` option with the `+liborder` option.

Summary of Library Scan Precedence

The order that *xrun* uses to search libraries for definitions of modules and UDPs depends on the location of the unresolved instance and the type of scan precedence in use. However, in all cases, all definitions in the source files specified on the command line are given first precedence. The differences between the types of scan precedence are shown in Table 7-1.

Table 7-1 Variance of Scan Precedence

Location	Scan Precedence	Library Search Order
Source file	default and `+librescan`	1. The left-most library file or directory on the command line. 2. The remaining libraries in the order in which they appear on the command line .

Source file	`+liborder`	1. The library file or directory that follows the source files on the command line. 2. The remaining libraries in a circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited.
Library file	`+librescan`	1. The left-most library file or directory on the command line. 2. The remaining libraries in the order in which they appear on the command line.
Library file	default and `+liborder`	1. The library file that contains the instance. 2. The remaining libraries in a circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited.
File in a Library Directory	`+librescan`	1. The file in a library directory that contains the instance. 2. The left-most library file or directory on the command line. 3. The remaining libraries in the order in which they appear on the command line.

File in a Library Directory	default and `+liborder`	1. The file in the library directory that contains the instance. 2. The other files in that library directory. 3. The library file or directory that immediately follows on the command line. 4. The remaining libraries in circular order, scanning libraries as they follow on the command line, and then “wrapping around” to the left-most library and any following libraries as yet unvisited.

Compiling -v/-y Modules into a Single Library

By default, *xrun* compiles `-v` library files and `-y` library directories to their own separate libraries, which can impact the performance of the Xcelium Simulator at elaboration time. As an alternative, you can specify one of the two options below on the `xrun` command line to change this default behavior and improve elaboration performance:

`-enable_single_yvlib`	This option enables the tool to compile all specified `-v` files and `-y` directories into a single library named `single_yvlib`.
`-enable_work_yvlib`	This option enables the tool to compile all specified `-v` files and `-y` directories into a single `worklib` directory, and can be helpful if your design has a `worklib` requirement.

Supporting Compilation Units in Library Files

A compilation unit is a collection of one or more source files compiled together.

SystemVerilog extends Verilog by allowing declarations outside of a module, interface, package, or program. Each compilation unit has a compilation unit scope (cu-scope), which contains all of the external declarations made across all files within the compilation unit. Unlike global declarations, which are shared by all of the modules that make up a design, cu-scope declarations are visible only to the source files that make up the compilation unit.

Prior to release 13.2, the *xrun* -v and -y library management scheme did not support the compilation unit scope properly. Going forward, the defined behavior is to encapsulate all parsed elements in a compilation unit scope from the first source file parsed to the last library file parsed, and so on, until all encountered module and UDP instances are resolved.

For example, consider the following design:

File / Directory	Filename	Module Definition	Instantiations
Source file	**`top.v`**	top	`moda::inst_a()`

`lib1.v` (*Library file*)	**`lib1.v`**	`mod11`
		`mod21`
			`dummy::inst_dummy1()`
			`dummy::inst_dummy2()`

`lib2` (*Library directory*)	**`moda.v`**
	`dummy.v`
		`moda`
		`dummy`
			`modb::inst_b()`
			`dummy::inst_dummy3()`

`lib3` (*Library directory*)	**`modb.v`** **`dummy.v`**	`modb` `dummy`	- `dummy::inst_dummy5()`
`lib3` (*Library directory*)	**`modc.v`**	modc	dummy::inst_dummy6()

`lib4.v` (*Library file*)	**`lib4.v`**	`mod41`
		`mod42`
			`dummy::inst_dummy7()`
			`dummy::inst_dummy8()`

And suppose you use the following `xrun` command to search the libraries for definitions of modules and UDPs that are unresolved in `top.v`:

```
% xrun top.v +libext+.v+ -v lib1.v -y lib2 -y lib3 -v lib4.v
```

In this example, *xrun* starts by parsing the source file `top.v`, then the library file `lib1.v`, followed by the library directory file `/lib2/moda.v`, another library directory file `/lib3/modb.v`, and finally the library file `lib4.v`. The cu-scope contains all definitions outside of the module's scope using the specified files only. The other files in the library directory are not included in the scope.

Protecting Compilation Unit Scope Elements

There are situations when the same library file may be parsed multiple times to bind unresolved instances. In order to protect cu-scope elements, you should use unique compile-time directives as shown:

```
`ifndef DONE
    `define DONE
    bind moda checker inst_chk();
`endif
```

Specifying File Extensions in Library Directories

You can specify the files in a library directory that you want *xrun* to use to resolve module and user-defined primitive (UDP) definitions by specifying their file extensions. If you choose to specify files with no file extensions, then each filename in a library directory must match the name of the module or UDP that they contain. To specify library directory file extensions, use the `+libext` or `-libext` command-line option.

`+libext`	Enter `+libext` on the command line followed immediately (no spaces) by the strings of characters that make up each extension. You surround each extension string with two `+` signs. Since the `+` signs separate one extension string from another, the final `+` sign in the argument is optional.
	For example, the following two command lines are equivalent:
	`% xrun source1.v -y /usr/me/lib/cmos +libext+.v`
	`% xrun source1.v -y /usr/me/lib/cmos +libext+.v+`
	This example specifies the files named `<module_or_UDP_name>.v` in the library directory `/usr/me/lib/cmos`.

`-libext`	Enter `-libext` on the command line followed immediately by a space and then the strings of characters that make up the file extension. When specifying multiple extension strings with this option, you must separate each string with a comma. For instance: `% xrun source1.v -y /usr/me/lib/cmos -libext .v` This example is equivalent to the `+libext` example above.

Specifying Multiple Library Directory File Extensions

With *xrun*, you can choose to specify multiple library directory file extensions using either `+libext` or `-libext`. If a file that has the first extension is not found, then *xrun* tries each extension that follows it on the command line until the file is found or until *xrun* has tried all the listed extensions. All specified extensions must follow a single `+libext` or `-libext` option on the command line, as in the following examples:

```
% xrun source1.v -y /usr/me/lib/cmos +libext+.v+.v2+
```

or

```
% xrun source1.v -y /usr/me/lib/cmos -libext .v,.v2
```

The extension string need not contain a period character. The *xrun* command concatenates each string specified to the ends of the names of the modules and UDPs that need to be resolved while searching a library directory. This means that all the extensions shown below are valid when using the `+libext` option:

```
% xrun source1.v -y /usr/me/proj/lib/cmos +libext+.v+_version_3+64+
```

And the same is true when using `-libext` on the command line:

```
% xrun source1.v -y /usr/me/proj/lib/cmos -libext .v,_version_3,64
```

Suppose the modules `NAND2`, `MUX`, and `ADDER` cannot be resolved in `source1.v` on the command line shown above. In this case, *xrun* scans the following files if they reside in the library directory `/usr/me/proj/lib/cmos`:

```
NAND2.v MUX_version_3 ADDER64
```

In some situations, you may want *xrun* to scan the library directory files that have no extensions (or null extensions) along with the files that do have extensions. With the `+libext` option, you can specify a null extension as two adjacent `+` signs, like this: `++`.

Here is an example:

```
% xrun source1.v -y /usr/proj/lib/cmos +libext++.v+
```

With the `-libext` option, you can specify a null extension as an empty value in the comma-separated list of file extensions:

```
% xrun source1.v -y /usr/proj/lib/cmos -libext ,.v
```

Both commands above direct *xrun* to look first for files with no extension, and then for files with the extension `.v`.

9

Timing Delays and Race Conditions in Gate-Level Netlists

The simulation of a gate-level netlist with no timing delays is prone to race conditions. This typically occurs when you are simulating with no SDF delay annotation or when using a zero delay option (`-delay_mode_zero`). With no proper clock delay balancing, the clock and data signals can propagate instantaneously, making the data from one stage available at downstream stages before the appropriate clock edge is available.

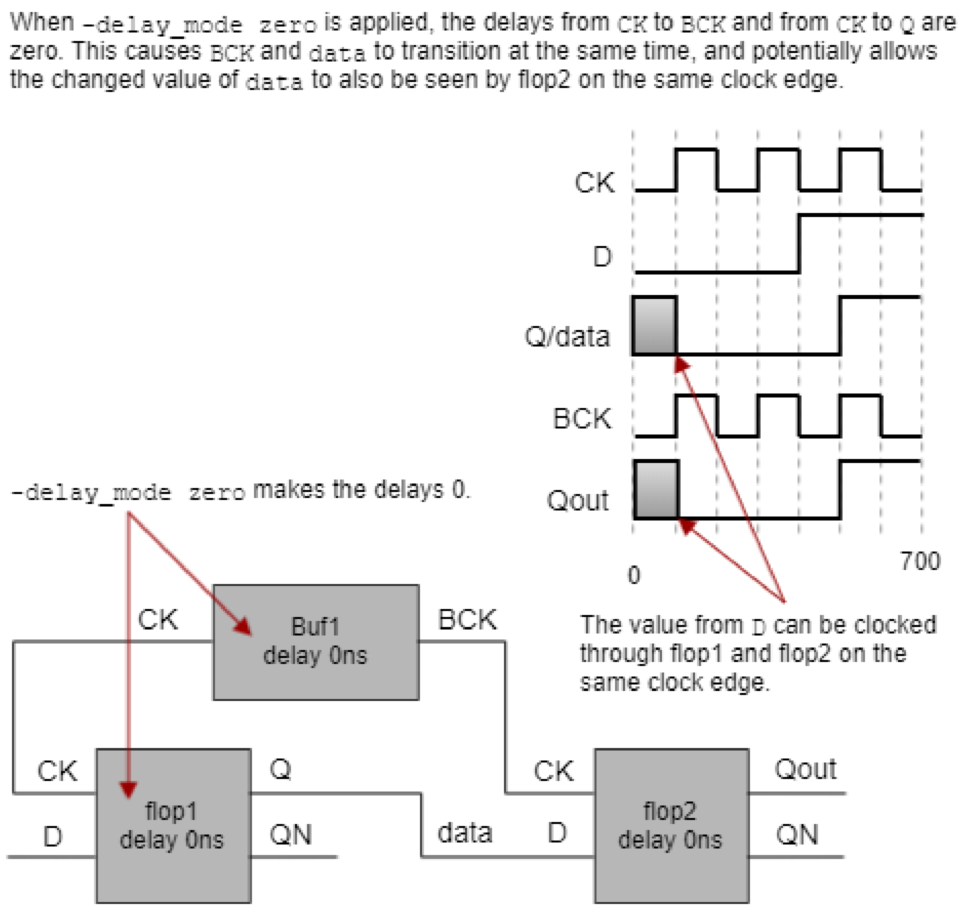
Delay Options to Prevent Race Conditions

The following *xmelab* delay options can help you avoid race conditions caused by zero timing delays:

`-seq_udp_delay`	Applies a delay value to the input/output paths of sequential UDPs. Use the `-seq_udp_delay` option with a delay specification value to set all delays to zero (as if you are using `-delay_mode zero`) except for sequential UDPs. The specified delay value overrides any delay specified for sequential UDPs in the design, in `specify` blocks, through SDF annotation, and so on. The option also removes any timing checks associated with sequential UDPs. The shift register example in Figure 1 illustrates the problem of race conditions occurring in a gate-level netlist with a zero delay. Figure 2 shows how the `-seq_udp_delay` option avoids the race.

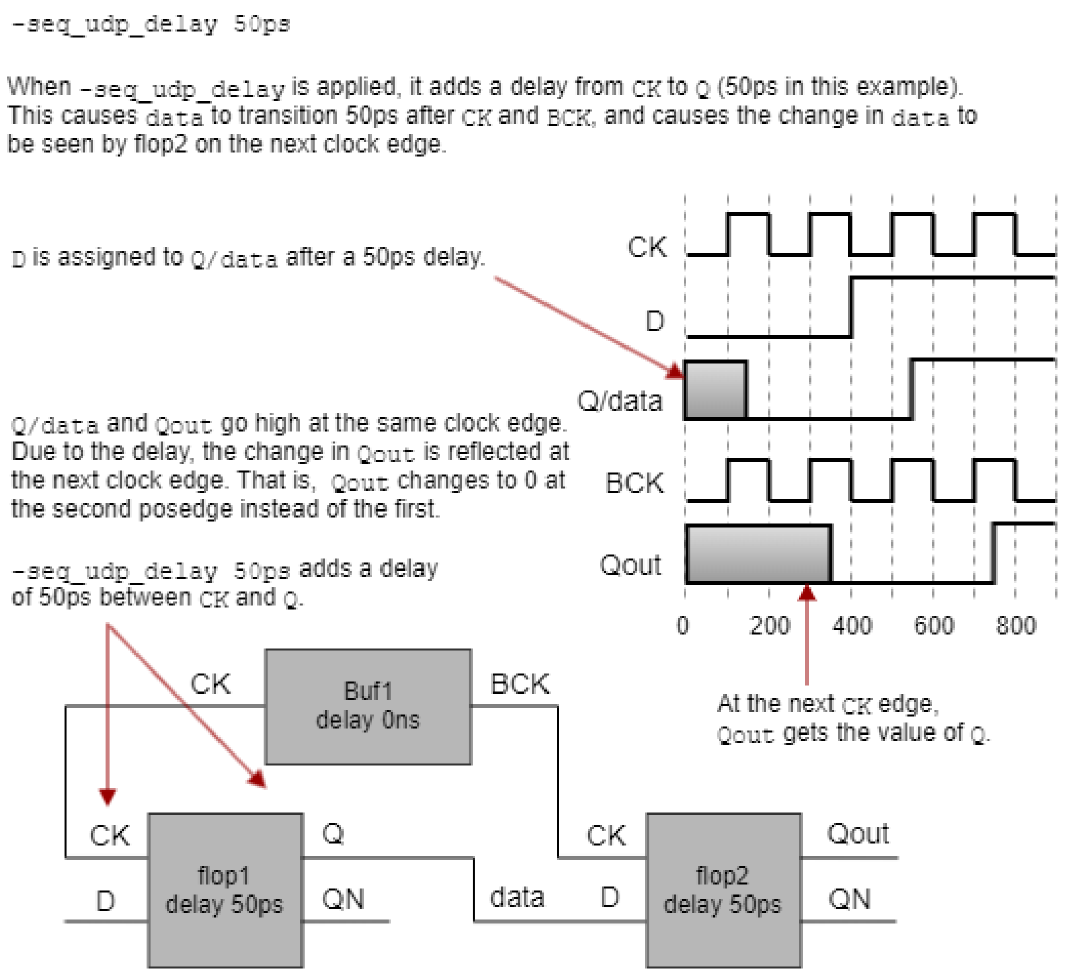
`-add_seq_delay`	Applies a delay value to the input/output paths of un-delayed sequential UDPs. The `-add_seq_delay` option updates un-delayed sequential UDPs with a specific delay value. The option applies the delay to only those sequential UDPs that do not have a path delay already defined in the instantiation. Unlike `-seq_udp_delay`, this option preserves all other delay values in the design, such as module path delays, specify blocks, SDF annotation, and so on. This option is useful for avoiding race conditions, but comes at the expense of performance improvements that can be gained with `-seq_udp_delay`, which removes all other delays and timing checks.
`-delta_sequdp_delay`	Adds a delta delay to sequential UDPs as a way to settle the flow of logic value transitions being transferred through each logic stage (such as scan chains). A delta delay provides a minimum delay when sequential UDPs do not include explicitly specified delays.
`-sequdp_nba_delay`	Adds a non-blocking delta delay to the input/output paths of sequential UDPs. The `-sequdp_nba_delay` option adds a nonblocking delta delay to sequential UDPs at the end of the simulation cycle, after all values have settled and when nonblocking assignments are evaluated. This differs from `-delta_sequdp_delay` which delays to the *active* region of the next delta cycle the outputs of sequential UDP instances that do not explicitly specify delays. The `-sequdp_nba_delay` option delays to the *NBA* region of the current delta cycle the outputs of sequential UDP instances that do not explicitly specify delays. Both options provide a minimum delay when sequential UDP instances do not explicitly specify delays.

Figure 1 No Timing Delay Creates a Race Condition


When -delay_mode zero is applied, the delays from CK to BCK and from CK to Q are zero. This causes BCK and data to transition at the same time, and potentially allows the changed value of data to also be seen by flop2 on the same clock edge.
CK
D
Q/data
BCK
Qout
0
700
-delay_mode zero makes the delays 0.
The value from D can be clocked through flop1 and flop2 on the same clock edge.
CK
Buf1 delay 0ns
BCK
CK
D
flop1 delay 0ns
Q
QN
data
CK
D
flop2 delay 0ns
Qout
QN

The following figure illustrates the effect of using `-seq_udp_delay` with a delay specification of `50ps`, for the shift register example in Figure 1.

Figure 2 Setting a Timing Delay on Sequential UDPs Avoids a Race Condition


-seq_udp_delay 50ps
When -seq_udp_delay is applied, it adds a delay from CK to Q (50ps in this example). This causes data to transition 50ps after CK and BCK, and causes the change in data to be seen by flop2 on the next clock edge.
D is assigned to Q/data after a 50ps delay.
Q/data and Qout go high at the same clock edge. Due to the delay, the change in Qout is reflected at the next clock edge. That is, Qout changes to 0 at the second posedge instead of the first.
-seq_udp_delay 50ps adds a delay of 50ps between CK and Q.
CK
D
Q/data
BCK
Qout
0
200
400
600
800
At the next CK edge, Qout gets the value of Q.
CK
Buf1
delay 0ns
BCK
CK
D
flop1
delay 50ps
Q
QN
data
CK
D
flop2
delay 50ps
Qout
QN

Related Topics

- Specifying -add_seq_delay with -seq_udp_delay
- Using Delta Delays to Avoid a Race Condition

Specifying -add_seq_delay with -seq_udp_delay

You can specify the `-add_seq_delay` option with the `-seq_udp_delay` option. In this case, the instance-specific `-add_seq_delay` value overrides the delay value applied with `-seq_udp_delay` (it does not add to the prior delay).

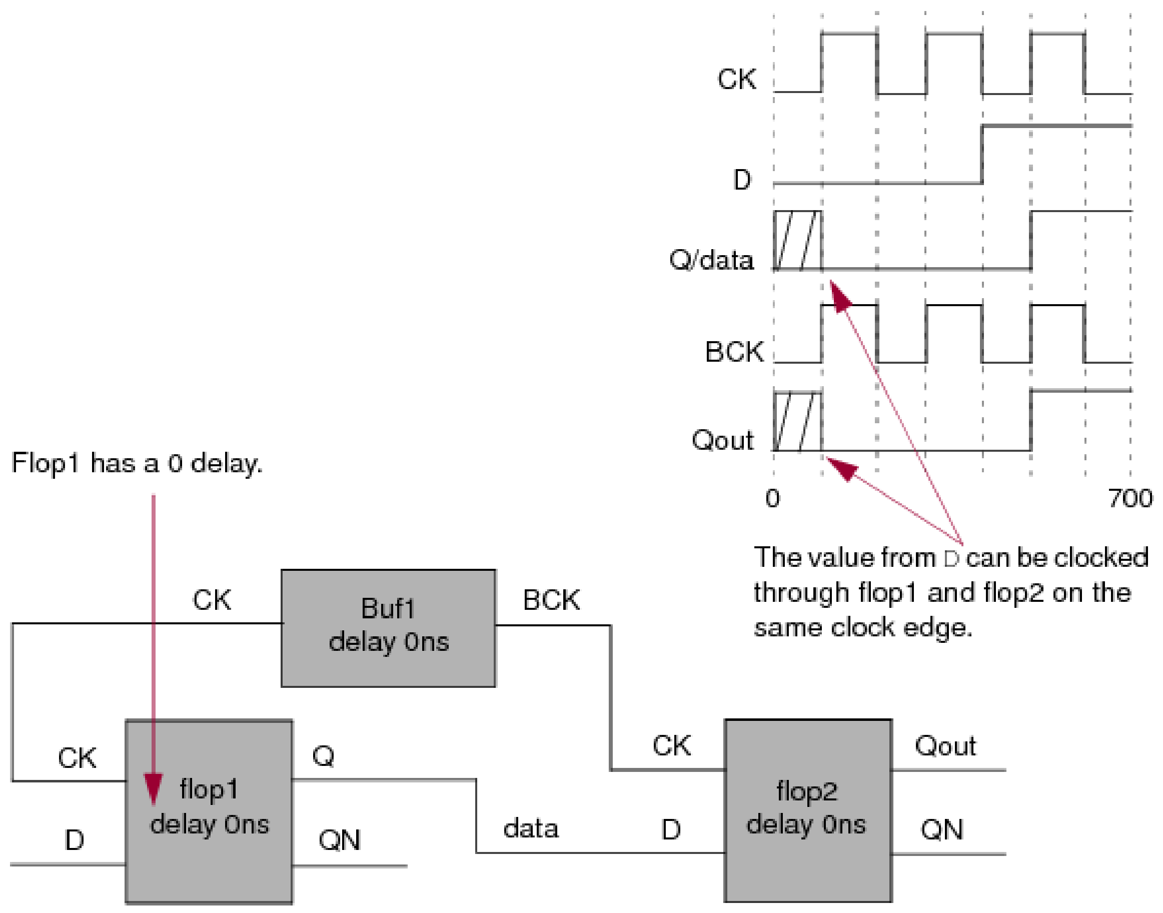
In the following example, the `-add_seq_delay` option applies a `40ps` delay to the instance `top.dut_top.u3`, while the `-seq_udp_delay` option applies a `20ns` delay to all other UDPs:

```
-seq_udp_delay 20ns -add_seq_delay top.dut_top.u3=40ps
```

The shift register example in Figure 3 illustrates a race condition occurring in a flip-flop with a zero delay. Figure 4 shows how applying `-seq_udp_delay` with a specific delay value avoids the race.

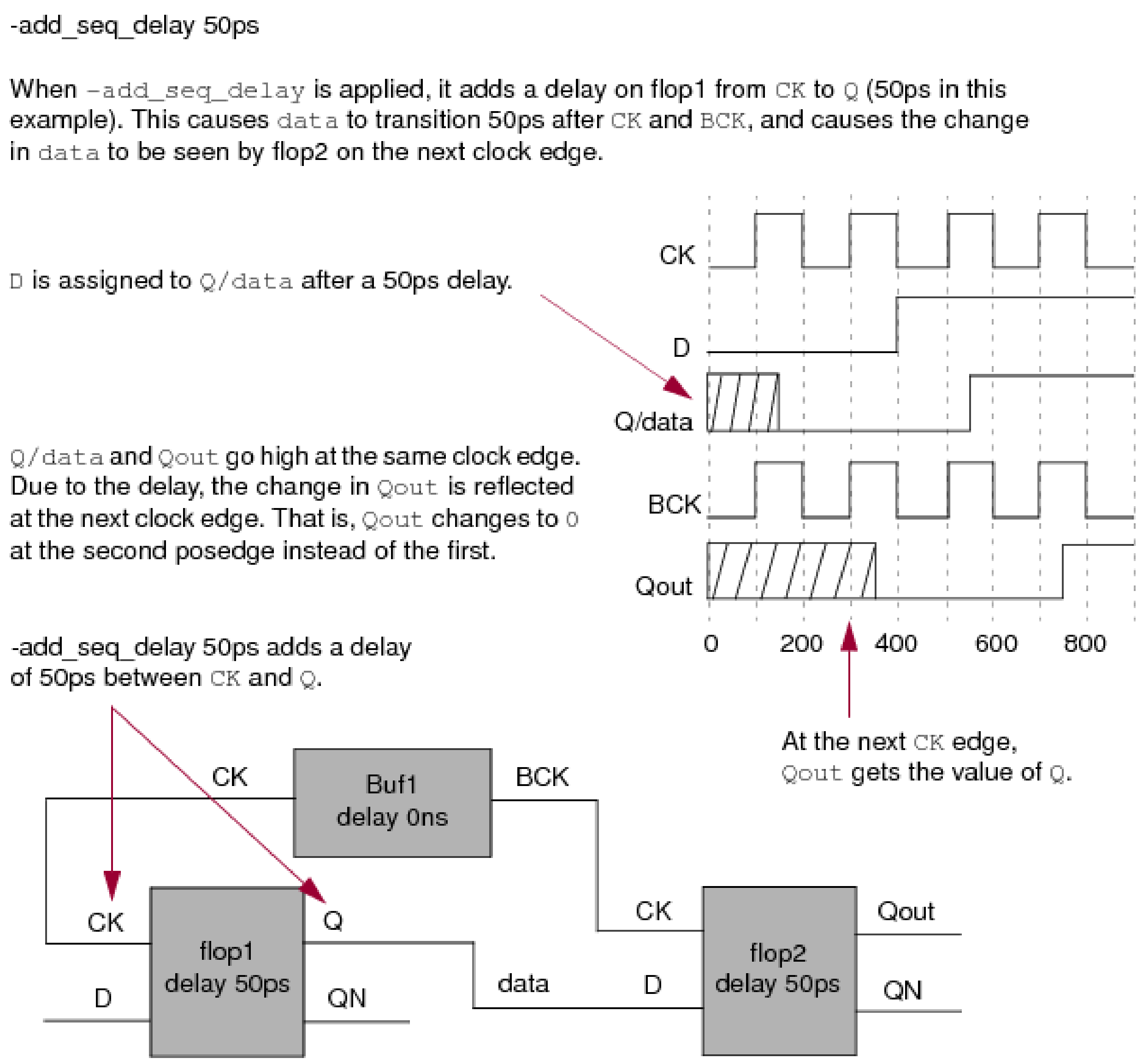
Figure 3 No Delay On a Sequential UDP Creates a Race Condition

The delays from `CK` to `Q` and from `CK` to `BCK` are zero. This causes `BCK` and `data` to transition at the same time, and potentially allows the changed value of `data` to also be seen by flop2 on the same clock edge.



The following figure illustrates the effect of using `-add_seq_delay` with a delay specification of `50ps`, for the shift register example above in Figure 3.

Figure 4 Adding a Sequential Delay Avoids a Race Condition


-add_seq_delay 50ps

When `-add_seq_delay` is applied, it adds a delay on flop1 from `CK` to `Q` (50ps in this example). This causes `data` to transition 50ps after `CK` and `BCK`, and causes the change in `data` to be seen by flop2 on the next clock edge.

`D` is assigned to `Q/data` after a 50ps delay.

`Q/data` and `Qout` go high at the same clock edge. Due to the delay, the change in `Qout` is reflected at the next clock edge. That is, `Qout` changes to `0` at the second posedge instead of the first.

-add_seq_delay 50ps adds a delay of 50ps between `CK` and `Q`.

At the next `CK` edge, `Qout` gets the value of `Q`.

Related Topic

- Using Delta Delays to Avoid a Race Condition

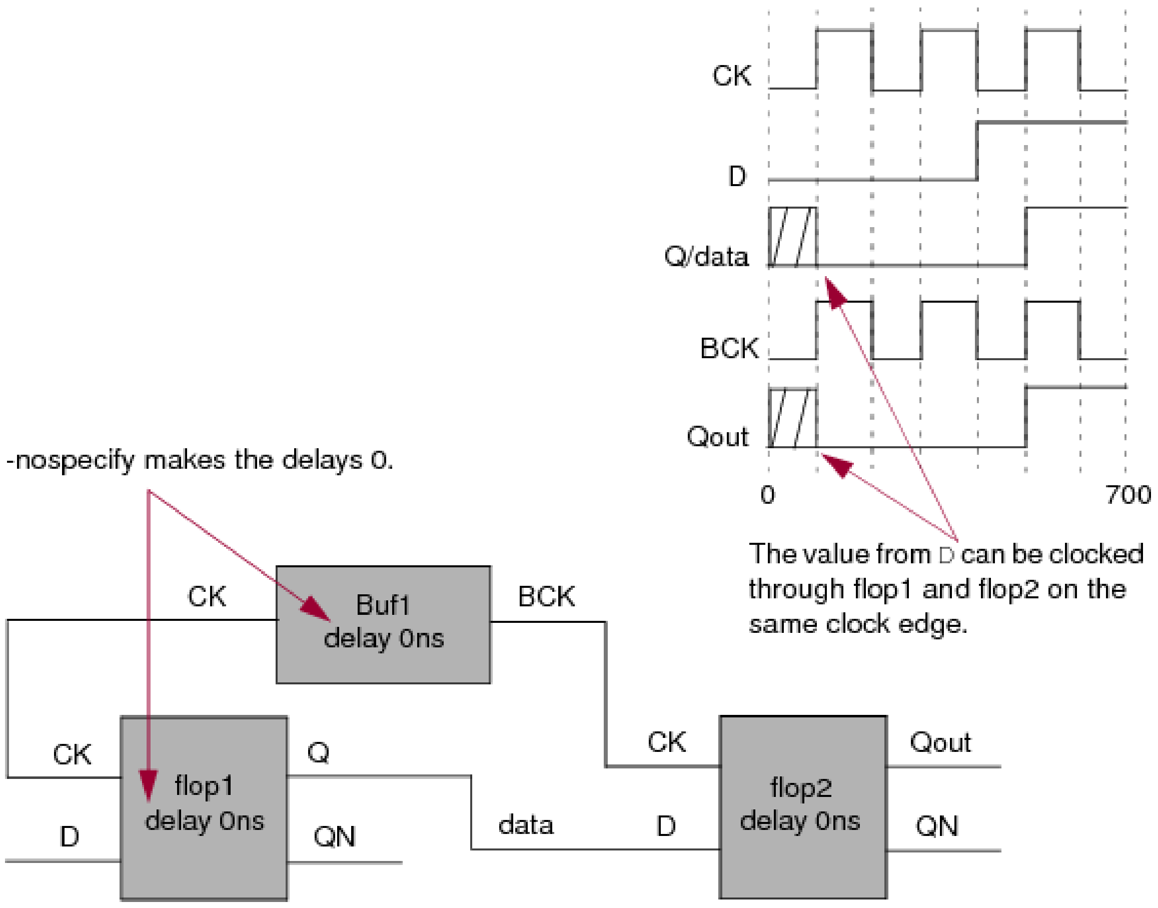
Using Delta Delays to Avoid a Race Condition

Both `-delta_sequdp_delay` and `-sequdp_nba_delay` are useful when running zero delay simulations with the `-nospecify` option (which disables SDF annotation and the timing features of specify blocks). Adding a delta delay to sequential UDPs allows the combinational logic to settle before the simulator updates the output of the sequential logic.

The shift register example in Figure 5 illustrates the problem of a race condition occurring in a gate-level netlist simulating with the `-nospecify` option. Figure 6 shows how the `-delta_sequdp_delay` option avoids the race. Figure 7 shows how the `-sequdp_nba_delay` option avoids the race.

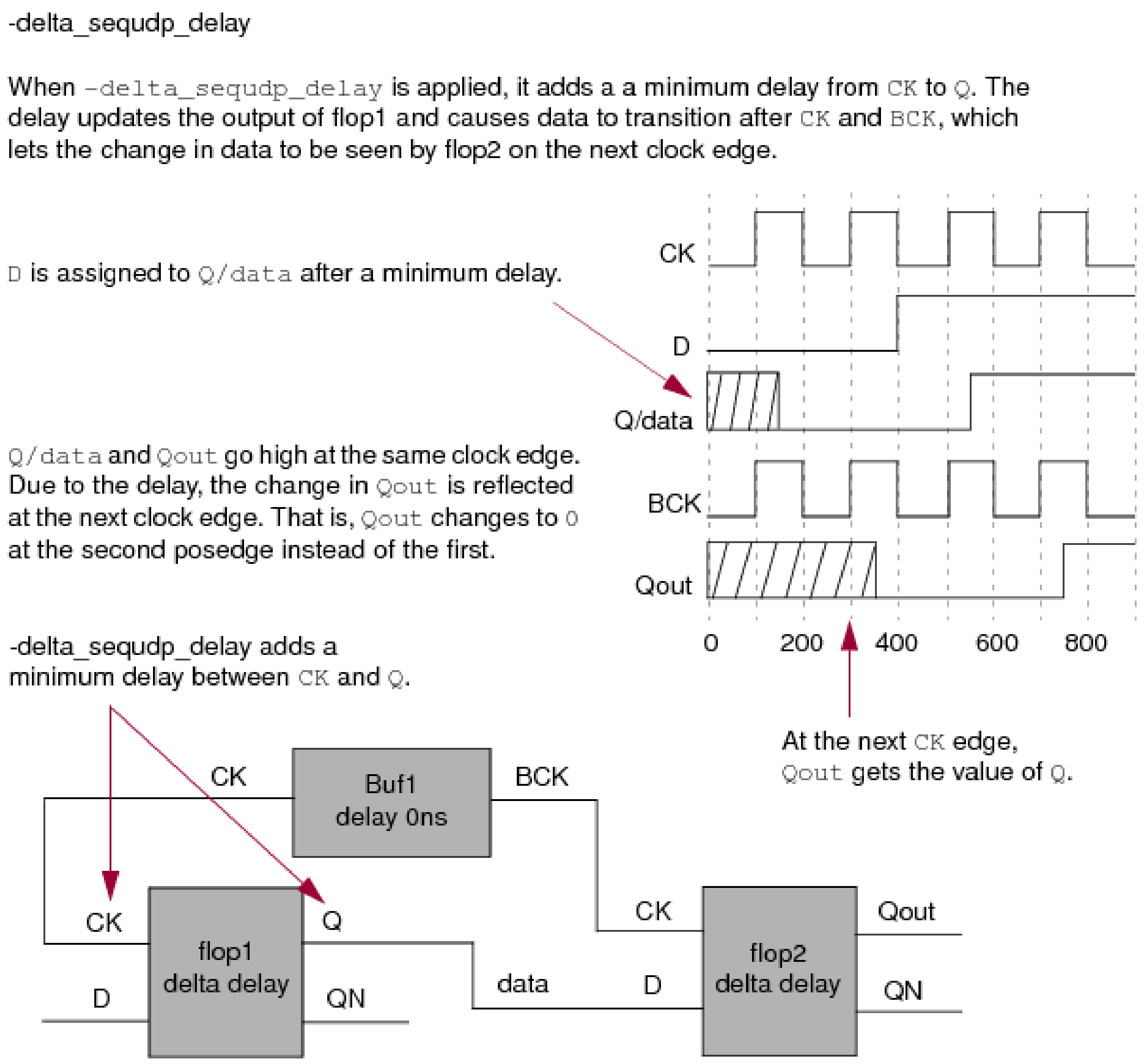
Figure 5 Simulating With -nospecify Creates a Race Condition

When `-nospecify` is used, the delays from `CK` to `BCK` and from `CK` to `Q` are zero. This causes `BCK` and `data` to transition at the same time, and potentially allows the changed value of `data` to also be seen by flop2 on the same clock edge.


CK
D
Q/data
BCK
Qout
0
700
The value from D can be clocked through flop1 and flop2 on the same clock edge.
-nospecify makes the delays 0.
CK
Buf1
delay 0ns
BCK
CK
flop1
delay 0ns
Q
D
QN
data
CK
D
flop2
delay 0ns
Qout
QN

The following figure illustrates the effect of using `-delta_sequdp_delay` for the shift register example in Figure 5.

Figure 6 Specifying a Delta Delay Avoids a Race Condition



-delta_sequdp_delay

When `-delta_sequdp_delay` is applied, it adds a a minimum delay from `CK` to `Q`. The delay updates the output of flop1 and causes data to transition after `CK` and `BCK`, which lets the change in data to be seen by flop2 on the next clock edge.

`D` is assigned to `Q/data` after a minimum delay.

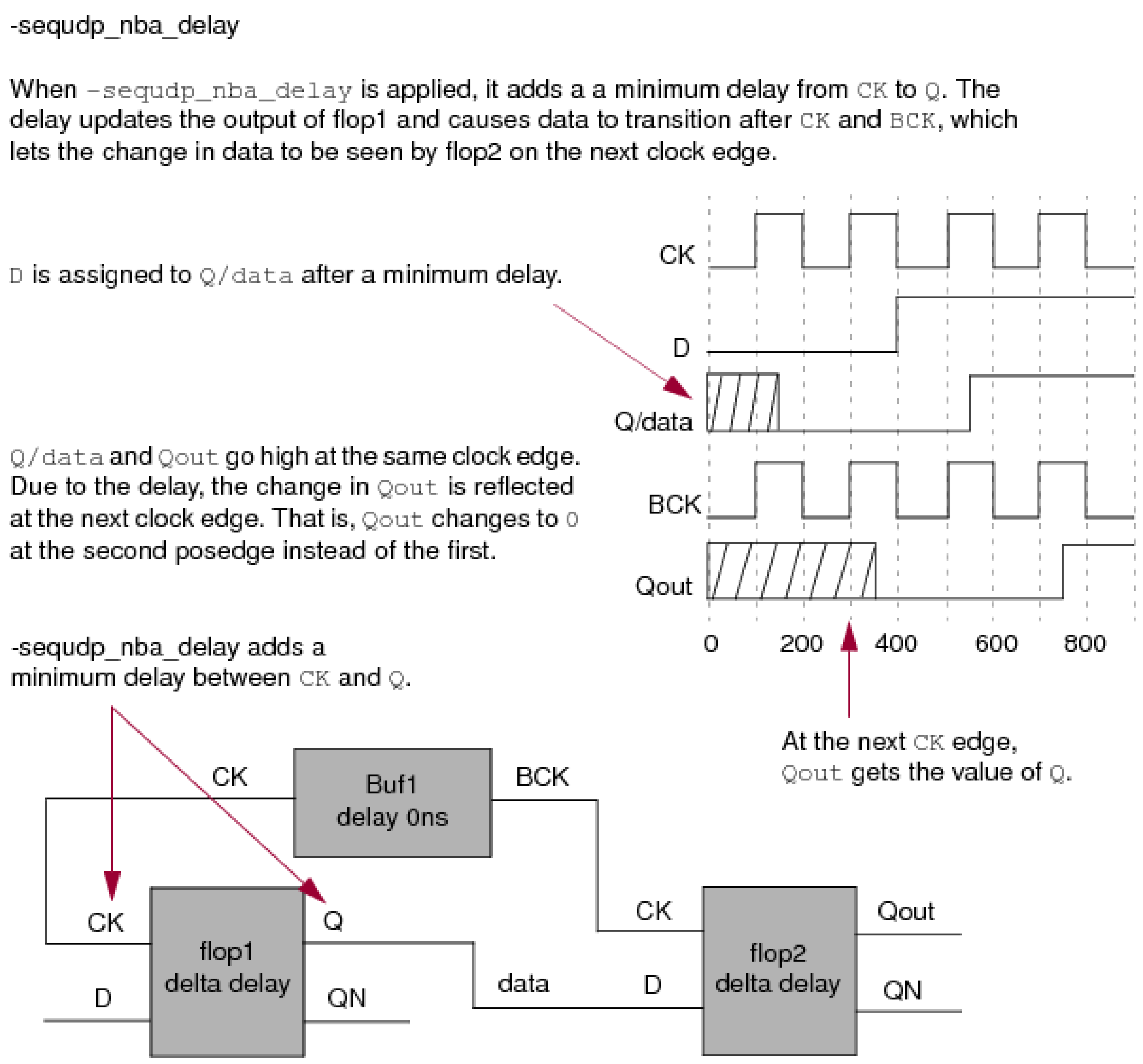
`Q/data` and `Qout` go high at the same clock edge. Due to the delay, the change in `Qout` is reflected at the next clock edge. That is, `Qout` changes to `0` at the second posedge instead of the first.

-delta_sequdp_delay adds a minimum delay between `CK` and `Q`.

At the next `CK` edge, `Qout` gets the value of `Q`.

Figure 7 illustrates the effect of using `-sequdp_nba_delay` for the same shift register example.

Figure 7 Specifying a Nonblocking Delta Delay Avoids a Race Condition



-sequdp_nba_delay

When `-sequdp_nba_delay` is applied, it adds a a minimum delay from `CK` to `Q`. The delay updates the output of flop1 and causes data to transition after `CK` and `BCK`, which lets the change in data to be seen by flop2 on the next clock edge.

`D` is assigned to `Q/data` after a minimum delay.

`Q/data` and `Qout` go high at the same clock edge. Due to the delay, the change in `Qout` is reflected at the next clock edge. That is, `Qout` changes to `0` at the second posedge instead of the first.

-sequdp_nba_delay adds a minimum delay between `CK` and `Q`.

At the next `CK` edge, `Qout` gets the value of `Q`.

Related Topic

- Specifying -add_seq_delay with -seq_udp_delay

10

Delay Modes in Xcelium

Delay modes let you alter the delay values specified in your Verilog models using command-line options and compiler directives with Xcelium. Depending on which method you choose, delay modes can be defined on a global or module basis. For example, if you assign a specific delay mode to a module, all instances of that module simulate in that mode. The delay mode of each module is then determined at elaboration time and cannot be altered dynamically.

Once specified, the selected delay mode controls only structural delays (delays assigned to gate and switch primitives, UDPs, and nets), path delays, timing checks, and delays on continuous assignments. Any other delays simulate as specified regardless of the delay mode.

The following are the delay modes that you can explicitly select at elaboration time using the `-delay_mode` option and the Xcelium default mode if no delay mode is selected.

Punit Delay Mode	In precision unit delay mode, the simulator ignores all timing checks. It sets all path delays to simulate with the value of one simulation time unit, where one simulation time unit is equal to one unit of module precision.
Path Delay Mode	In this mode, the simulator derives its timing information from specify blocks. If a module contains a specify block with one or more module path delays, all structural and continuous assignment delays within that module (except `trireg` charge decay times) are set to zero. In path delay mode, `trireg` charge decay remains active. The module simulates with "black box" timing, with module path delays only. You can specify distributed delays that cannot be overridden by the path delay mode using the `DelayOverride$ specparam` or PLI access routines. When a path delay mode simulation encounters a distributed delay-locked by either mechanism, module path delays and the distributed delay simulate concurrently. Modules with no module path delays simulate distributed delay mode when path delay mode is selected.

Distributed Delay Mode	Distributed delays are delays on nets, primitives, or continuous assignments--in other words, delays other than those specified in procedural assignments and specify blocks. The simulator ignores all module path delay information in distributed delay mode and uses all distributed delays and timing checks. You can override specified delay values with PLI access routines. The simulator ignores the `DelayOverride$ specparam` in the distributed delay mode.
Unit Delay Mode	The simulator ignores all module path delay information and timing checks in-unit delay mode. It converts all non-zero structural and continuous assignment delay expressions to a unit delay of one simulation time unit. To override the effect of the unit delay mode for specific delays, you can use: • PLI access routines • the `DelayOverride$ specparam` in a specify block
Zero Delay Mode	Zero delay mode is similar to unit delay mode in that all module path delay information, timing checks, and structural and continuous assignment delays are ignored. To override the effect of the zero delay mode for specific delays, you can use: • PLI access routines • the `DelayOverride$ specparam` in a specify block
No Delay Mode	If you do not specify a delay mode, or if you specify `-delay_mode` none, delays simulate as defined in the model's source description files. You can also specify path delays and distributed delays in the same module. Such delays simulate together only when the simulation is in this default mode.

Reasons to Select a Delay Mode

Replacing integer path or distributed delays with global zero or unit delays can considerably reduce the simulation time. During design debugging phases, you can use delay modes when checking circuit logic is more important than detailed timing checks. You can also speed up simulation during debugging by selectively disabling delays in sections of the model where timing is not currently a concern. If these are significant portions of a large design, the time saved may be substantial.

The *distributed* and *path* delay modes allow you to develop or use modules that define both path and distributed delays and then choose the path or the distributed delays at elaboration time. This feature allows you to use the same source description with multiple tools and select the appropriate delay mode when using the sources with the simulator. You can set the delay mode for the simulator by placing a compiler directive for either the distributed or the path mode in the module source description file or by specifying a global delay mode at elaboration time.

Related Topics

- VPI/VHPI Reference
- Setting a Delay Mode
- Module Behavior with Delay Modes
- Timescales and Simulation Time Units
- Overriding Delay Values
- Delay Mode Example
- Decompiling with Delay Modes

Setting a Delay Mode

There are two ways to set a delay mode:

- Using the command-line options to set a global delay mode when you invoke the elaborator.
- Using the compiler directives in the source file to set delay modes specific to particular modules.

The order of precedence in delay mode selection from highest to lowest is as follows:

- Command-line options
- Compiler directives

- Default – no delay mode

Command-Line Options

Use the `-delay_mode` option on the `xmelab` or `xrun` command line to set the global delay mode. You can specify one of the following arguments: `unit`, `zero`, `path`, `punit`, `distributed`, and `none`. If more than one `-delay_mode` option is specified on the command line, the elaborator issues a warning and selects the mode with the highest precedence. The table below lists the `-delay_mode` arguments from highest to lowest priority:

Option/Argument	Result
`-delay_mode punit`	The design simulates in precision unit delay mode. All timing checks are ignored.
`-delay_mode path`	The design simulates in path delay mode, except modules with no module path delays.
`-delay_mode distributed`	The design simulates in distributed delay mode.
`-delay_mode unit`	The design simulates in unit delay mode.
`-delay_mode zero`	Modules simulate in zero delay mode.
`-delay_mode none`	Delays simulate as specified in the source (default).

With *xrun*, you can use delay mode plus options to specify the global delay mode in place of `-delay_mode`. In this case, `+delay_mode_punit` has the highest precedence. Delay mode plus options are supported only using single-step *xrun* mode.

Compiler Directives

Use compiler directives to select a delay mode for all instances of the same module. The compiler directive must precede the module definition. The compiler directives are:

- `` `delay_mode_path ``
- `` `delay_mode_distributed ``
- `` `delay_mode_unit ``
- `` `delay_mode_zero ``

When the compiler encounters a delay mode directive in a source file, it applies that delay mode to all modules defined from that point onwards until it meets a directive specifying a different delay mode or the end of the compilation. Delay modes specified with a compiler directive remain active across file boundaries. You can use the `` `resetall `` compiler directive at any point to return the source to the default delay mode (no mode selected). The recommended usage is to place `` `resetall `` at the beginning of each source text file, followed immediately by the directives desired in the file. You can override all compiler directives by using the command-line options.

Related Topics

- `+delay_mode_punit`
- Delay Modes in Xcelium
- Module Behavior with Delay Modes
- Timescales and Simulation Time Units
- Overriding Delay Values
- Delay Mode Example
- Decompiling with Delay Modes

Module Behavior with Delay Modes

The following table summarizes the rules governing the behavior of a module for which a particular delay mode is in effect.

	Punit	**Path****	**Distributed**	**Unit**	**Zero**	**Default**
module path delays	set to 1*	used	ignored	ignored	ignored	used
timing checks	ignored	used	used	ignored	ignored	used
delays specified by access routine	PLI access routines work in all delay modes					
override by `DelayOverride$`	ignored	used	ignored	used	used	ignored
treatment of distributed delays	ignored	ignored	used as defined	set to 1***	set to 0	used as defined

* module path delays are set to one simulation time unit in punit mode, where the simulation unit is 1 module precision
** path mode is ignored in modules containing no path information
*** non-zero values are set to one simulation time unit

Timescales and Simulation Time Units

When working with delay modes, you should consider the way delay modes use timescales and simulation time units. When you select the unit delay mode, each explicit delay gets converted to a value of one, *measured in simulation time units*--that is, the value of the smallest *time_precision* argument specified by a `` `timescale `` compiler directive in *any* of your model's description files.

For example, you can specify an explicit delay for a gate as follows:

```
nand #5 g1 (qbar, q, clear);
```

When a model uses timescales, the delay of five units is measured in timescale units. That is, its simulation value is five times the unit of time specified in a controlling timescale directive. (In the absence of any timescale directives, the delay is a relative value. It is used to schedule events in the correct relative order.) For example, the gate shown above might be controlled by the following timescale directive:

```
`timescale 1 us/1 ns
```

This directive causes the simulation delay value for the nand gate g1 to be five microseconds. Elsewhere in the model, you might have the following timescale directive, which gives the smallest precision argument specified for the model:

```
`timescale 10 ns/1 ps
```

The above timescale value sets the simulation time unit as one picosecond, so the five-microsecond delay on nand gate `g1` is 5,000,000 picoseconds. When selecting the unit delay mode, your five-microsecond delay on `g1` gets converted to one picosecond.

The following example shows how delay times change from the default mode when the unit delay mode is selected.

```
`timescale 1 ns/1 ps
module alpha (a, b, c);
    input b, c;
    output a;
    and #2 (a, b, c);
endmodule

`timescale 100 ns/1 ns
```

```
module beta (q, a, d, e);
    input a, d, e;
    output q;
    wire f ;
    xor #2 (f, d, e);
    alpha g1 (q, f, a);
endmodule
```

```
`timescale 10 ps/1 fs
module gamma (x, y, z);
    input y,z;
    output x;
    reg w;
    initial
    #200 w = 3;
     ...
endmodule
```

Delay mode selection controls the delays in the above example with the following results:

- **zero delay mode** - no delays on gates; delay on assignment to register `w` is 2 ns, as specified because delay modes do not affect behavioral delays
- **unit delay mode** - delays of one femtosecond on gates; delay on assignment to register `w` is 2 ns, as specified because delay modes do not affect behavioral delays
- **path delay mode** - distributed delays are used because no module paths are defined
- **distributed delay mode** - distributed delays are used
- **default delay mode** - default delays are used

Related Topics

- Delay Modes in Xcelium
- Setting a Delay Mode
- Overriding Delay Values
- Delay Mode Example
- Decompiling with Delay Modes

Overriding Delay Values

You can use one of the following two methods to override the effect of a delay mode selection:

- PLI access routines
- the `DelayOverride$` specparam in a specify block

PLI Access Routines and Delays

You can use a PLI access routine to override a structural delay set by a delay mode. This method can provide structural delay values in a module regardless of the method used to define the module's delay mode. The PLI routines that set delay values are the following:

- `acc_append_delays`
- `acc_replace_delays`

An application can use the `acc_fetch_delay_mode` access routine to retrieve delay mode information.

Refer to the *PLI 1.0 User Guide and Reference* and the *VPI User Guide and Reference* for more information.

⚠ In a PLI access routine, the delay value is measured in the timescale units of the module containing the gate.

DelayOverride$ specparam

Modules frequently need distributed delays on sequential elements to prevent race conditions. Sometimes such a module also needs path delays. Use the `DelayOverride$` specparam to ensure that these essential delays are not overridden in path, unit, or zero delay modes.

The `DelayOverride$` specparam lets you specify a delay on a particular instance of a primitive or UDP that occurs during the zero, unit, or path delay modes. The delay provided by this mechanism replaces the distributed delay that the zero, unit, or path delay mode overrides. You must also provide a distributed delay to take effect during the distributed and default delay modes.

To use `DelayOverride$`, include it in the specify block section of the module that contains the instance to be controlled. The specparam uses the `DelayOverride$` prefix followed by the primitive or UDP instance name, with no space between. The syntax is as follows:

```
specparam DelayOverride$object_name = literal_constant_value;
```

The *`object_name`* is a primitive or UDP instance name. If you specify no object name, the simulator overrides all delays on gate primitives and UDPs in that module. The *`literal_constant_value`* is the number that provides the value for the delay. The number can be any of the following:

- a decimal integer
- a based number (for example, `2'b10`)
- a real number
- a *`min`*`:`*`typ`*`:`*`max`* expression composed of any one of the above three number formats

The following example shows how to use the `DelayOverride$` specparam:

```
module
    ...
    nand #5 g1 (q, qbar, preset) ;
    ...
    specify
        ...
        specparam DelayOverride$g1= 5;
        ...
    endspecify
    ...
endmodule
```

When using `DelayOverride$`, the delay override value is measured in simulation time units--that is, the module's timescale is ignored.

⚠ In Verilog HDL, when you use `+delay_mode_path` with the `DelayOverride$` specparam, some gates are assigned zero delay, and some are assigned the override value. In the Xcelium simulator, all gates are assigned the override value.

Related Topics

- Delay Modes in Xcelium
- Setting a Delay Mode
- Timescales and Simulation Time Units
- Delay Mode Example
- Decompiling with Delay Modes

Delay Mode Example

Consider the following example, `test.v`, which includes an instantiated module that simulates explicit delays on certain gates.

```
`timescale 1ns/1ps
module delay(in, out1, out2, out3, out4, transport, inertial);
   input in;
   output out1, out2, out3, out4, transport;
   output inertial;
   reg transport;
   wire inertial;
   always @in
      begin
         transport <= #10 in;
      end
   assign #10 inertial = in;

   not#0 g1(out1, in);
   not#5 g2(out2, in);
   not#10 g3(out3, in);
   not g4(out4, in);

   specify
      ( in => out4) = 4;
   endspecify
endmodule

module wrapper;
   reg IN;
   wire OUT1, OUT2, OUT3, OUT4, TRANSPORT, INERTIAL;
   delay inst(IN, OUT1, OUT2, OUT3, OUT4, TRANSPORT, INERTIAL);
   initial begin
      IN = 1'b0;
      #50 IN = 1'b1;
      #50 IN = 1'b0;
      #50 $finish();
   end
endmodule
```

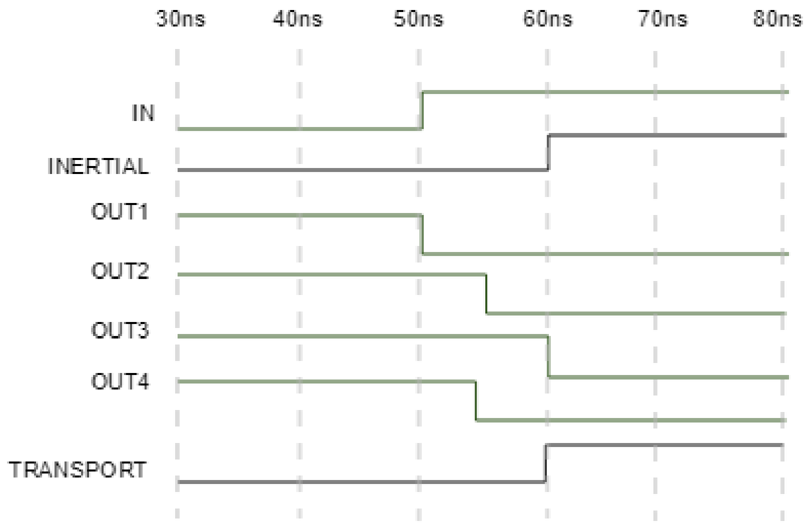
To run this example in direct invocation mode, specify the following commands:

```
% xmvlog test.v
% xmelab -access +rwc wrapper
% xmsim -tcl -profile -input "@probe -screen *; run; exit" wrapper
```

Alternatively, to run this example in single-step *xrun* mode, specify the following:

```
% xrun -access +rwc -tcl -profile -input "@probe -screen *; run; exit" test.v
```

The HDL is coded such that this results in the following waveform by default:



where,

- `g1` (out1) has zero delay
- `g2` (out2) has a distributed delay of 5 simulation time units
- `g3` (out3) has a distributed delay of 10 simulation time units
- `g4` (out4) has a path delay of 4 simulation time units
- `inertial` and `transport` both have a delay of 10 simulation time units

The following table shows how the simulation delays are executed when you select one of the global delay modes:

unit delay	Adding `-delay_mode unit` to the `xmelab` or `xrun` command line converts all non-zero structural and continuous assignment delay expressions to a delay of one simulation time unit. This changes the `inertial` delay, as well as the delays on gates `g2` and `g3`. The delay on gate `g4` is ignored since it is a module path delay. The `transport` delay and the zero delay on gate `g1` behave the same as when no delay mode is specified.

zero delay	Adding `-delay_mode zero` to the `xmelab` or `xrun` command line ignores all module path delay information, timing checks, and structural and continuous assignment delays coded in the HDL source. This changes the `inertial` delay, and the delays on gates `g2`, `g3`, and `g4`. Gate `g1` is already a zero delay, and remains unchanged. The `transport` delay behaves the same as when no delay mode is specified.
distributed delay	Adding `-delay_mode distributed` to the `xmelab` or `xrun` command line ignores `specparam` and module path specifications. Only gate `g4` is ignored in this case since it is a module path delay. All other delay specifications behave the same as when no global delay mode is specified.
path delay	Adding `-delay_mode path` to the `xmelab` or `xrun` command line uses the module path delay information in the HDL source and ignores all distributed delay specifications. This changes the `inertial` delay, and the delays on gates `g1`, `g2`, and `g3`. The `transport` delay and the module path delay on gate `g4` behave the same as when no delay mode is specified.
punit delay	Adding `-delay_mode punit` to the `xmelab` or `xrun` command line ignores all timing checks and sets all path delays to simulate with the value of one simulation time unit. Only gate `g4` is changed in this case since it is a module path delay. All other delay specifications behave the same as when no delay mode is specified.

Related Topics

- Delay Modes in Xcelium
- Setting a Delay Mode
- Timescales and Simulation Time Units
- Overriding Delay Values
- Decompiling with Delay Modes

Decompiling with Delay Modes

When decompiling a Verilog HDL source using `$list` or the `-d` compile-time option, the delay values displayed are the ones being simulated--not the ones in the original description. If delays have been added using PLI access routines, these are not shown in the decompilation.

Macro Module Expansion and Delay Modes

When a delay mode is in effect, all macro module instances within the scope of that delay mode are expanded before the delay mode information is processed. This rule means that a macro module instance inherits the delay mode of the module in which it is expanded.

Related Topics

- Delay Modes in Xcelium
- Setting a Delay Mode
- Timescales and Simulation Time Units
- Overriding Delay Values
- Delay Mode Example

11

Setting Pulse Controls

In the Xcelium simulator, both module path delays and interconnect delays are simulated as transport delays by default. There is no command-line option to enable the transport delay algorithm. You must, however, set pulse control limits to see transport delay behavior. If you do not set pulse control limits, the limits are set equal to the delay by default, and no pulses having a shorter duration than the delay will pass through. That is, if you do not set pulse control limits, module path delays and interconnect delays are simulated as transport delays, but the results look as if the delays are being simulated as inertial delays. Full pulse control is available for both types of delays. You can:

- Set global pulse control for both module path delays and interconnect delays.
- Set global pulse limits for module path delays and interconnect delays separately in the same simulation.
- Narrow the scope of module path pulse control to a specific module or to particular paths within modules using the `PATHPULSE$` specparam.
- Specify whether you want to use *On-Event* or *On-Detect* pulse filtering.

Related Topics

- PATHPULSE Keyword
- PATHPULSEPERCENT Keyword

Setting Global Pulse Control

To set global pulse limits for both module path delays and interconnect delays, you can use the `-pulse_r` and the `-pulse_e` options when you invoke the elaborator.

If you want to set pulse control for module path delays and interconnect delays separately in the same simulation, use the following sets of options:

- `-pulse_r` and `-pulse_e` to set limits for path delays
- `-pulse_int_r` and `-pulse_int_e` to set limits for interconnect delays

Based on the global pulse control setting, the simulator takes one of the following actions:

- Reject the output pulse (the state of the output is unaffected).
- Let the output pulse through (the state of the output reflects the pulse).
- Filter the output pulse to the error state. This generates a warning message and then maps to the `x` state.

⚠ Pulse widths are measured at the output, not at the input.

The action that the simulator takes depends on the delay value and a window of acceptance. The simulator calculates the window of acceptance from the following two values that you supply as arguments to the options. Both arguments are a percentage of the delay.

- *reject_percent*
- *error_percent*

Syntax

```
-pulse_r reject_percent -pulse_e error_percent [Lib.]Cell[:View]
```

Example

```
% xmelab -pulse_r 50 -pulse_e 80 top_mod
```

The calculation of the limits is as follows:

```
reject_limit = (reject_percent / 100) * delay
error_limit = (error_percent / 100) * delay
```

For example, the command line shown above specifies a `reject_percent` of 50% and an *error_percent* of 80%. This means that, for a module path delay of 50 time units, the reject limit is 25 time units (50% of 50 time units) and the error limit is 40 time units (80% of 50 time units).

Using the reject limit and error limit calculations, the simulator acts on pulses according to the following rules:

- Reject if 0 <= pulse < (reject limit).
- Set to error if reject limit <= pulse < (error limit).
- Pass if pulse >= error limit.

Therefore, in the example, in which the module path delay is 50 time units, the following are the reject, error, and pass values:

Output Pulse Width	Result
0 - 24	Reject
25 - 39	Set to error
40+	Pass

To generate an error whenever a module path pulse is less than the module path delay, use the following values:

```
-pulse_r 0 -pulse_e 100
```

The values of *reject_percent* and *error_percent* must fall between 0 and 100, with *reject_percent <= error_percent*.

The default values for *reject_percent* and *error_percent* are 100. If you omit only the *reject_percent*, its default value is 100. If you omit only the *error_percent*, its value is set to the *reject_percent*. If the *reject_percent* exceeds the *error_percent*, a warning is issued and the *error_percent* is reset to equal the *reject_percent*. For example, the *error_percent* in the following command line is reset to 100 because the *reject_percent* has the default value of 100.

```
% xmelab top -pulse_e 80
```

In the following command line, the *error_percent* is set to the *reject_percent* of 50.

```
% xmelab top -pulse_r 50
```

The following example shows how to use the `-pulse_r` and `-pulse_e` options to set global path pulse control. In this example, module path delay = 50 time units.

```
% xmelab -pulse_r 60 -pulse_e 90 hardrive
```

In this example, a module path delay of 50 time units has a reject limit of 30 time units (60% of 50 time units). The error limit is 45 time units (90% of 50 time units).

- Pulses smaller than 30 time units are rejected.
- At 30 through 44 time units, pulses are set to the error state and then mapped to the `x` state.
- At 45 time units and above, pulses are passed through.

Use the `-epulse_no_msg` option when you invoke the simulator to suppress the display of error messages for pulses smaller than *error_percent*.

```
% xmsim -epulse_no_msg top
```

Use the `-epulse_onevent` or `-epulse_ondetect` option when you invoke the simulator to specify the type of pulse filtering that you want to use.

Related Topics

- Setting Pulse Controls
- Setting Pulse Control for Specific Modules and Module Paths
- Pulse Filtering Style

Setting Pulse Control for Specific Modules and Module Paths

You can override global pulse control for module paths by declaring `PATHPULSE$` specparams in specify blocks. The `PATHPULSE$` specparam narrows the scope of module path pulse control to a specific module or to particular paths within modules.

Use the `-pathpulse` command-line option to enable the `PATHPULSE$` specparams.

⚠ Standard Delay Format (SDF) annotation can provide new values for pulse limits of both module path delays and interconnect delays. This annotation method operates independently of the `PATHPULSE$` specparam construct, and the `-pathpulse` option is not needed when pulse control values are provided by SDF annotation.

Syntax

```
pulse_control_specparam
::= PATHPULSE$ = ( reject_limit [, error_limit ]);
|| PATHPULSE$ module_path_source $ module_path_destination =
                      ( reject_limit [, error_limit ]);
```

If a module path source and a module path destination are specified, the pulse control is applied to the specific module path. If the source and destination are not included, the pulse control is applied to all paths declared within the module. If both path-specific `PATHPULSE$` specparams and a non-path-specific `PATHPULSE$` specparam are specified in the same module, the path-specific specparams take precedence.

The sources and destinations in *`module_path_source`* and *`module_path_destination`* can be scalar nets or vector nets, but they cannot be bit-selects or part-selects. The pulse handling characteristics you specify for paths beginning in a vector and ending in a vector automatically apply to all module paths connecting the two vectors.

Values assigned to the `PATHPULSE$` specparam define the pulse handling windows in time units (not percentages, as in the `-pulse_r` and `-pulse_e` options). The first value represents the reject limit; the second value is the error limit. If you supply only one value, both limits are set to the same value.

Example

The following example illustrates how to use `PATHPULSE$` specparams to set path pulse controls on specific paths. You must use the `-pathpulse` option when you elaborate this design.

```
specify
    (clk => q) = 12;
    (data => q) = 10;
    (clr, pre *> q) = 4;
specparam
    PATHPULSE$ = 3;
    PATHPULSE$clk$q = (2, 9);
    PATHPULSE$clr$q = 1;
endspecify
```

In this example:

- The second `PATHPULSE$` specparam sets a reject limit of 2 and an error limit of 9 for the path `(clk=>q)`.
- The third `PATHPULSE$` specparam sets reject and error values of 1 for the path `(clr*>q)`. Note that a pulse control limit is specified for the first input signal `clr` in module path `(clr, pre => q)`, but no pulse control limit is specified for `pre`, the second signal in the path. Pulse limits for this path are not affected by `PATHPULSE$clr$q`.

  In Verilog-XL, you must use the `+expand_specify_vectors` option to obtain this behavior. Without this option, all signals in module paths with multiple inputs or outputs have the same delays and pulse handling. That is, without the `+expand_specify_vectors` option, the third `PATHPULSE$` specparam sets reject and error values of 1 for both `clr=>q` and `pre=>q`.

- The first `PATHPULSE$` specparam sets reject and error values of 3 for the path `(data=>q)`.

Related Topics

- Setting Pulse Controls
- Setting Global Pulse Control
- Pulse Filtering Style

Pulse Filtering Style

The simulator provides the following methods of pulse filtering.

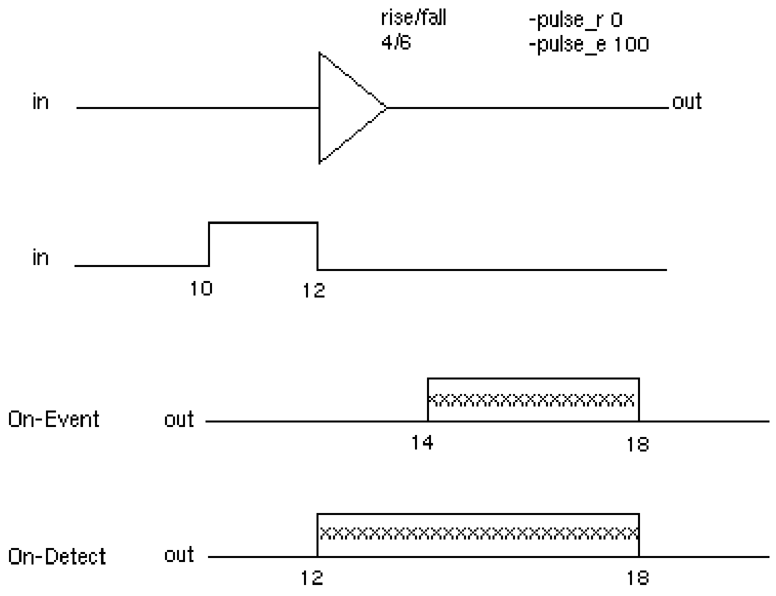
- **On-Event method**: Filters pulses so that the transition *to* X occurs after the normal delay is calculated for the scheduled input transition. The transition *from* X occurs after the normal delay is calculated for the new output state.
- **On-Detect method**: Filters pulses so that the transition to X occurs immediately upon the detection of the pulse error. The X state remains until the normal delay is calculated for the new output state.

The On-Detect allows for more pessimism when filtering pulses to the X state, producing a longer X region. This method more closely reflects the output caused by nearly simultaneous inputs that result in scheduling output events at the same time

This topic includes an example that illustrates the difference between the *On-Event* and *On-Detect* methods of pulse filtering. It also discusses anomalies associated with schedule cancellation, which occurs when narrow pulses or nearly simultaneous transitions occur at model inputs.

On-Event vs. On-Detect Pulse Filtering

The following figure uses a simple buffer with asymmetric rise/fall times and pulse limits equal to the delay to illustrate the difference between On-Event and On-Detect pulse filtering. An output waveform is shown for both On-Event and On-Detect.



With a rise delay of 4 and a fall delay of 6, the simulator schedules the delay for times 14 and 18 based on the input transitions. If pulse limits are set equal to the delay, the simulator generates an error whenever a module path pulse is less than the module path delay. If you use the On-Event pulse filter, the X state region exists from time 14 to 18. If you use the On-Detect pulse filter, the X state region extends from the closing edge of the violating input transition (time 12) to the normally calculated delay for the new output state (time 18).

You can specify the pulse filtering style by using elaborator (*xmelab*) command-line options or by using keywords in a `specify` block. The command-line options specify the type of pulse filtering to use for all module path and interconnect delays. The `specify` block keywords let you specify the pulse filtering method to use for specific paths.

Command-line options override keywords in `specify` blocks. If you do not specify any keywords or command-line option, On-Event is the default.

The command-line options are:

- `-epulse_onevent` (default)
- `-epulse_ondetect`

Example

```
% xmelab -epulse_ondetect top
```

The `pulsestyle_onevent` and `pulsestyle_ondetect` keywords can be used in a `specify` block to specify the pulse filtering style for particular paths. The syntax is:

```
pulsestyle_onevent path_output ...;
pulsestyle_ondetect path_output ...;
```

The pulse filtering style keywords must be defined for an output prior to any path declarations for that output.

Example 1

In this example, no keywords are specified within the `specify` block. If no command-line option is specified, the default On-Event method is used.

```
specify
    (a => out) = (2,3);
    (b => out) = (3,4);
endspecify;
```

Example 2

In this example, the `pulsestyle_ondetect` keyword is used to apply the On-Detect pulse filtering method to outputs `out` and `out_b`.

```
specify
    pulsestyle_ondetect out;
    (a => out)=(2,3);
    (b => out)=(4,5);
    pulsestyle_ondetect out_b;
    (a => out_b)=(5,6);
    (b => out_b)=(3,4);
endspecify;
```

Because multiple output declarations are allowed for the same keyword, the following line could have been used in the above example:

```
pulsestyle_ondetect out, out_b;
```

Example 3

In the following example, the default On-Event is applied to output `out` prior to the `(a => out)` path statement. This is then followed by the `pulsestyle_ondetect` keyword, which applies the On-Detect method to `out`. This generates an error because an output path cannot have both methods applied to it.

```
specify
    (a => out)=(2,3);
    pulsestyle_ondetect out;
    (b => out)=(3,4);
endspecify;
```

Pulse Filtering and Canceled Schedules

A schedule is canceled when a delay schedules a transition to occur before a previously scheduled transition. By default, the presence of canceled schedules is not indicated with an X state region. You can cause specify path outputs to use the X state to indicate the presence of canceled schedules by using the `-epulse_neg` command-line option when you invoke the elaborator or by using keywords in the `specify` block.

The `-epulse_neg` option turns on the X state display for all specify paths. The default is `-epulse_noneg`.

Example

```
% xmelab -epulse_ondetect -epulse_neg top
```

Use the `showcancelled` and `noshowcancelled` keywords in a specify block to turn on the display for particular paths.

Syntax

```
showcancelled path_output ...;
noshowcancelled path_output ...;
```

The keywords must be defined for an output prior to any path declarations for that output.

Command-line options override keywords in `specify` blocks. If you do not specify any keywords or command-line option, canceled schedules are not indicated.

For example, in the following `specify` block, the `showcancelled` keyword turns on the X state display for canceled schedules, and the `pulsestyle_ondetect` keyword specifies On-Detect style pulse filtering:

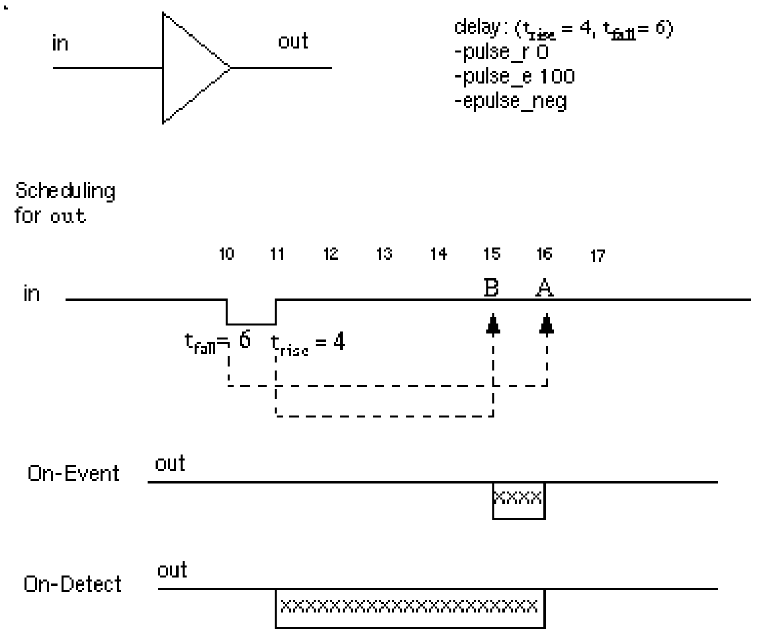
```
specify
    showcancelled out;
    pulsestyle_ondetect out;
    (a => out)=(2,3);
    (b => out)=(4,5);

    showcancelled out_b;
    pulsestyle_ondetect out_b;
    (a => out_b)=(5,6);
    (b => out_b)=(3,4);
endspecify;
```

Because multiple output declarations are allowed for the same keyword, the following specify block produces the same result as the example above:

```
specify
    showcancelled out,out_b;
    pulsestyle_ondetect out,out_b;
    (a => out)=(2,3);
    (b => out)=(4,5);
    (a => out_b)=(5,6);
    (b => out_b)=(3,4);
endspecify;
```

The following figure shows the X state region that occurs with a canceled schedule for each method of pulse filtering. Neither On-Event nor On-Detect provides a "correct" answer. Select the method that you want to use based on your library characterization and best judgement.



The events in this figure occur as follows:

1. At time 10, a 1->0 transition on the input causes the simulator to schedule event A at time 16 (10 + 6).
2. At time 11, a 0->1 transition on the input causes the simulator to schedule event B at time 15 (11 + 4).
3. Because event B is scheduled to occur before event A, the schedule for A is canceled. If you include the `-epulse_neg` option, an X state region is produced based on the pulse filtering method you use, as follows:

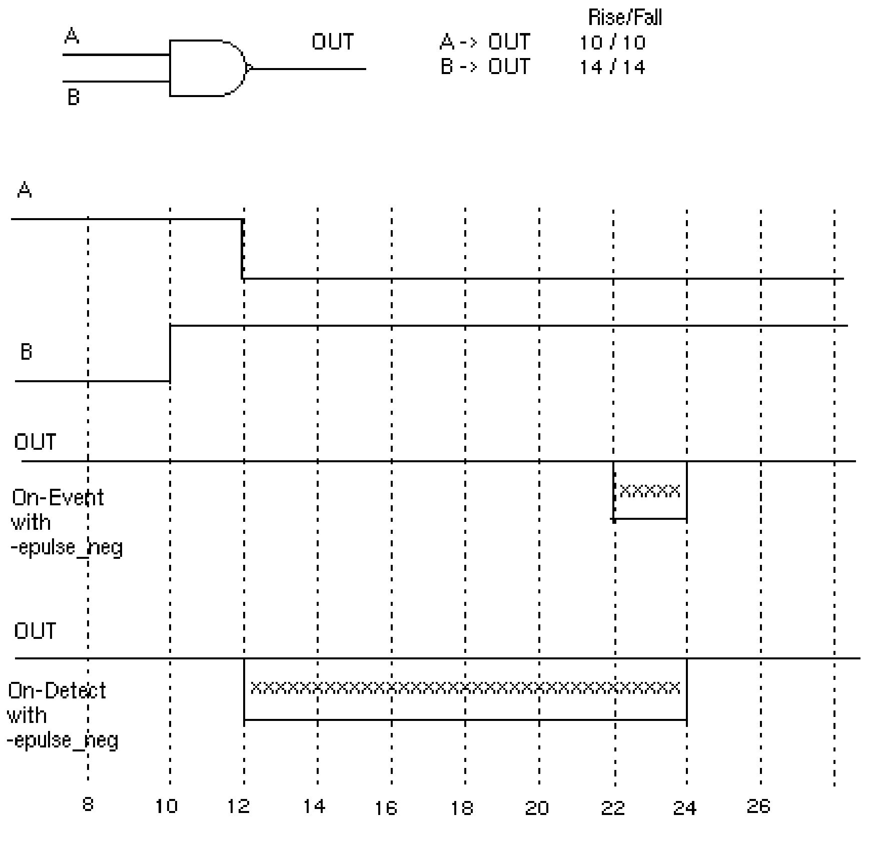
- On-Event pulse filtering produces an X state region on `out` that begins at the time of the second schedule, B, and that ends at the time of the canceled scheduled event, A, which is replaced with a schedule to the new logic state (in this case, 1).

- On-Detect pulse filtering produces an X state region on `out` that begins at the time the schedule was canceled and ends at the time of the canceled schedule A, which is replaced with a schedule to the new logic state (in this case, 1).

The following two examples show what happens when two inputs arrive at nearly the same time (closer together in time than the difference in delays) and cause a schedule cancellation. In the first example, the output events occur at different times; in the second example, the output events are scheduled at the same time.

Example 1: Nearly Simultaneous Switching Inputs (different event time)

The condition shown in this example is similar to the one described above in the buffer case, except that multiple signals are involved. The figure shows the waveform for a two-input NAND gate where input A is 1 and input B is 0.


A
B
OUT
Rise/Fall
A -> OUT 10 / 10
B -> OUT 14 / 14
A
B
OUT
On-Event with -epulse_neg
XXXXX
OUT
On-Detect with -epulse_neg
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
8
10
12
14
16
18
20
22
24
26

At time 10, input `B` makes a 0->1 transition, which schedules the output to make the 1->0 transition at time 24. At time 12, input `A` transitions 1->0, scheduling the output to transition from 0->1 at time 22.

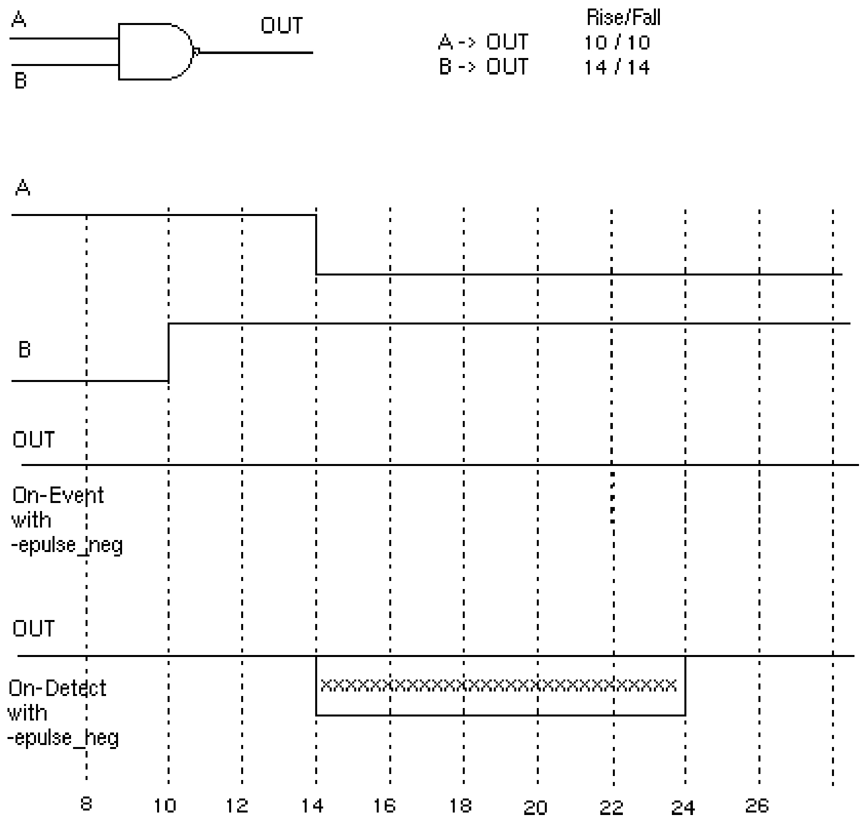
Because the second input (`A`) causes a schedule to be placed on the output prior to the one already scheduled from the `B` transition, the output takes on an X state to reflect the uncertainty of the output state between the two schedules.

If the input events cause the output to transition in the same direction, the X state is ignored because this is just a timing difference rather than an event that causes the output to create a momentary pulse.

Example 2: Nearly Simultaneous Switching Inputs (same event time)

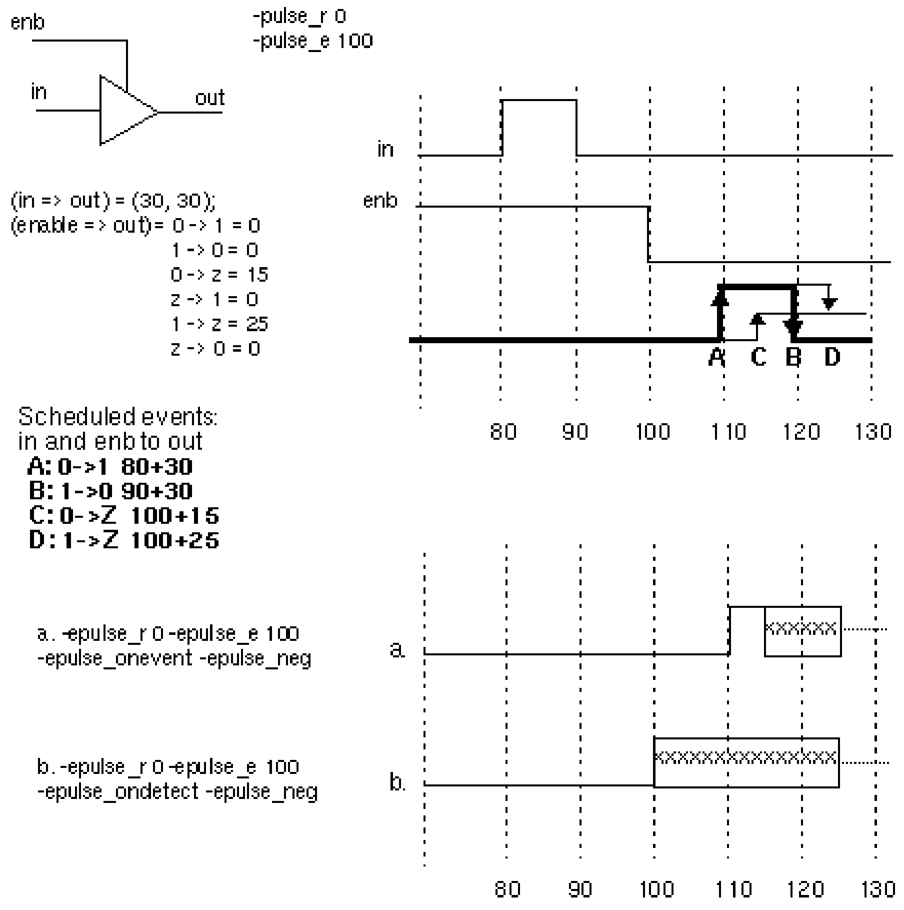
This example shows the result of nearly simultaneous input events causing output events to be scheduled at the same time. The transitions on inputs `A` and `B` both cause output events to be scheduled at time 24.

In this case, On-Event mode does not reflect that the event has occurred. This is because both output events occur at the same time, so no delta with an X can be shown. The On-Detect mode provides a longer time period in which the uncertainty can be indicated with an X.



The following is a more complex example of canceled schedules, using two inputs to `out`. This example also illustrates the effects of delay recalculation.

Delay recalculation takes place when a schedule is canceled, causing a change in the state previous to the new state. The delay must be recalculated based on the new state from which the signal is transitioning. The following example shows a delay being calculated for a 0 -> Z transition, but the schedule to 0 is canceled, and so a 1 -> Z transition delay must be calculated.



The waveforms in the previous figures are explained as follows:

The transport delay algorithm recalculates the 0->z delay based on a 1->z transition. This changes the time of the transition to `z` on the output to time 125. However, if that change is done, then the 1->0 transition at time 120 no longer needs to be canceled, producing a canceled schedule dilemma.

Wave `a` is produced because the On-Event filter causes an `x` to appear on the output due to the 1->0 schedule at time 120 being canceled. The `e` state extends from time 115 to 125.

Wave `b` is produced because the On-Detect filter extends the `e` state to the edge of the event that caused the pulse to occur, which is the transition on `enb` at time 100.

Related Topics

- Setting Pulse Controls
- Setting Global Pulse Control
- Setting Pulse Control for Specific Modules and Module Paths

12

Solving Race Conditions at Inputs

To solve race conditions between control/data signals for sequential flip-flops, you can use the `-congruency` elaboration option. With this option, the Xcelium solves the race conditions at inputs of flops (sequential always blocks) in a consistent and deterministic manner.

To solve such race conditions, perform the following steps:

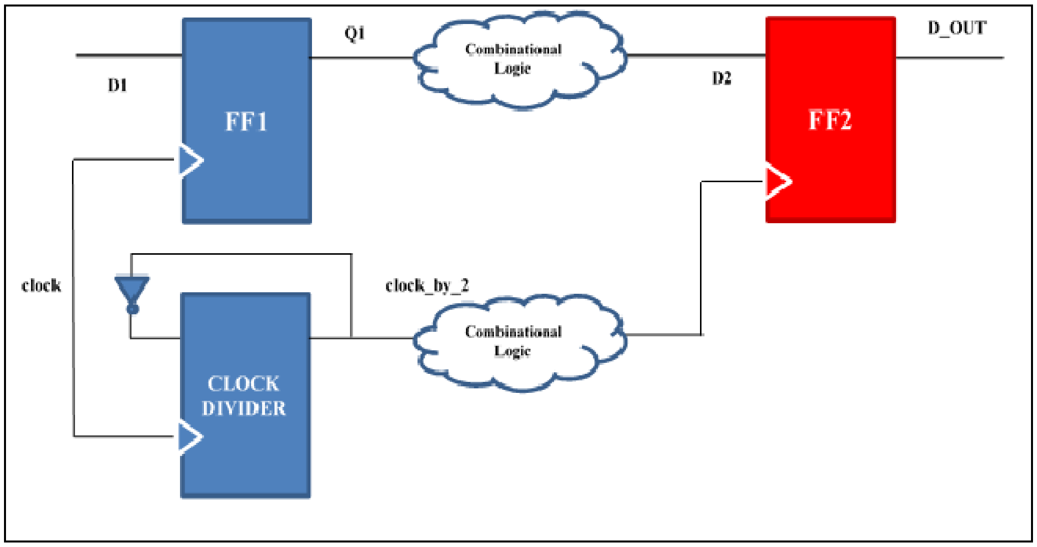
1. Identify `always` blocks that are sequential and supported with Congruency.
2. Identify the *control signal*" (`clk`/`set`/`rst`/`...`) and *data signals* of identified `always` blocks.
3. These identified sequential blocks always use the OLD values of the data signals. That is, the sequential blocks use value of the data signals from the beginning of the simulation time. This solves control/data race conditions in a deterministic manner.

⚠ The value of the data signals from beginning of the simulation time are used only within the sequential blocks identified to exhibit congruency behavior. In all other blocks, which have not been qualified as congruent, the updated value is used.

Generally, clock/data races happen if basic assumptions of NBA/BA are broken:

- Data signals are not assigned with NBA
- Clocks signals are assigned with NBA

In the following figure, there is a race between clock and data, as either the new or old value of data (D2) can be latched at FF2.



Example

In the following example, the `always` block:

```
always @(posedge clk)
Q <= d;
```

is replaced by:

```
assign #prepone d` = d;
always @(posedge clk)
Q <= d`;
```

Supported Datatypes

The supported datatypes can be divided into the following two categories.

Integral Datatypes	Aggregate Datatype
int	Multi-dimensional arrays
shortint	Packed/unpacked arrays
longint	Associative arrays
integer	
byte	
reg	

bit	
logic	
Packed Arrays	

Limitations of the -congruency Option

The current implementation does not support the following scenarios:

- Sequential block with assertions

```
always @(posedge clk_div2) begin
    data_div2 <= data_div1;
    assert(data_div1 != data_div2); // Immediate assertion inside always block
end
always @(posedge clk_div2) begin
    data_div2 <= data_div1;
    assert property (@(posedge clk_div2) data_div1 |=> data_div2); // Concurrent
assertion inside always block
end
```

- Sequential blocks that reads signals using hierarchical identifiers (OOMRs).

```
module top;
 bit clk_div2 = 0;
  int data_div2 = 0;
  always @(posedge inst1.clk_div1)
     clk_div2 <= ~clk_div2;
  always @(posedge clk_div2) begin
    data_div2 <= inst1.data_div1; // Reading OOMR
  end

test inst1();
endmodule

module test;
  bit clk_div1 = 0;
  int data_div1 = 0;
  always @(posedge clk_div1)
  begin
    data_div1 <= data_div1 + 1;
  end

  initial  forever // Input Clk
```

```
    #5 clk_div1 = ~clk_div1;
endmodule
```

- Read from interface port using dotted identifier.

```
module top;
  …
  ifc ifinst[0:4] ();
  …
  dut dutI(ifinst[0], clk, derived_clk, rst_n, store);
  …
endmodule

interface ifc();
  logic [7:0] d = 8'd0;
endinterface

module dut(ifc ifc1, input clk, derived_clk, rst_n, output logic [7:0] q);
  always @(posedge derived_clk or negedge rst_n) begin
    if (~rst_n) begin
      q <= 8'b0
    end
    else begin
      q <= ifc1.d;
    end
  end
endmodule
```

- Sequential blocks in verification elements (checker).

```
checker tester;
reg clk;
int data1, data2;
always_ff @(posedge clk)
    data2 <= data1;
endchecker
```

- Self-triggering always blocks are not identified as sequential.

```
always @(posedge clk_div1 or negedge clk_div3) begin
    if(clk_div1)
    data_div3 <= data_div1;
    else
    clk_div3 <= ~clk_div3;
end
```

- A sequential block with blocking or continuous assignments.

```
always @(posedge clk_div1)
```

```
begin
    data_div1 = data_div1 + 1;
end
always @(posedge clk_div2) begin
  assign  data_div2 = data_div1;
end
```

- Sequential blocks using fork…join instead of being…end

```
always @(posedge clk_div1)
fork
    data_div1 <= data_div1 + 1;
    $display("value of data_div1 is %d at time %t",data_div1,$time);
join
```

- Functions which read or modify a global variable

```
int data_div1 = 0;
int data_div2 = 0;
function int f1();
  return data_div1;  // global variable being read
endfunction
always @(posedge clk_div1)
    data_div2 <= f1();
```

- More than 3 events in sequential block's sensitivity list (because of limitation of present flop analysis)

```
always @(posedge rst2 or posedge rst0 or posedge rst1 or posedge clk) begin
      if (rst0)
        out2 <= in0;
      else if (rst1)
        out2 <= in1;
      else if (rst2)
        out2 <= in1;
      else
        out2 <= in2;
   end
```

- When system calls are present in a sequential flop, except `$display` and `$monitor`, the flop is not marked congruent. For `$display` and `$monitor`, when these system calls are present inside a congruent always block, updated ("new") value of congruent signal gets displayed.
- The following elements in sequential block:
    - Struct, full or member access, in RHS of assignment

- Tasks
- VPI/DPI calls
- Any non-SystemVerilog elements (VHDL, SystemC)
- Immediate and procedural assertions
- Covergroups
- Ref ports;

- `-sparsearray` compilation option
- The following technologies:
  - X-Propagation
  - AMS

  ⚠ In presence of the option/technologies listed above, elaboration issues the following error:

  ```
  xmelab: *E,CNGUNSP: Congruent behavior (-congruency option) is not
  supported with <option/technology>. Remove the relevant option(s) and
  rerun.
  ```

- In the case of MSIE, the congruency configuration file currently does not support select or deselect rules with hierarchical instance paths or descendants wildcard (`...`).
- Sequential UDPs
- Assertion (*calsim*) generated always blocks
- Dynamic SystemVerilog elements in a sequential block. Dynamic datatypes presently implemented in the tool are:
  - Queue
  - Dynamic array
  - Associative array
  - Class
  - Virtual interface
  - Semaphore
  - String

  - Mailbox
  - Process

Related Topics

- Identifying always Blocks as Sequential and Qualification for Congruency
- Congruency Configuration File
- Behavior of Congruent Signals
- Congruency Log File

Congruency Configuration File

By default, when using the `-congruency` option, the feature is applied to the entire design. However, you can use the configuration file to selectively apply congruency to some portions of the design. In absence of any select/deselect rules, the feature is applied to the entire design, by default. The configuration file determines on which design components (modules or instances) congruency is are enabled. You can also apply congruency module-wise under a certain instance hierarchy (including from top).

You can pass the configuration file using the `-cong_file` option at elaboration.

The following are use cases and their effects:

- If only `-congruency` option is present, then apply congruency on the full hierarchy.
- If only `-cong_file` switch is specified with the `-congruency` option and:
  - if select/deselect commands are present in the configuration file, then apply congruency on the specified hierarchy.
  - if *no* select/deselect commands are present in the configuration file, then do not apply congruency on any hierarchy.
- If both `-congruency` and `-cong_file` are present and:
  - if select/deselect commands are present in the configuration file, then apply congruency on the specified hierarchy.
  - if no select/deselect commands are present in configuration file, then apply congruency on full hierarchy.

Congruency Configuration File Commands

The following table describes the commands that you can specify in the congruency configuration file. For instance, to add or remove congruency from certain design unit(s).

Command	Syntax	Description
`select_congruency`	`select_congruency ([[(-design_unit|-module)] <`*`design_unit_name`*`>]) | (-instance <`*`hierarchical_instance_path`*`>)`	Enables congruency for one (or more) design units (modules, interfaces) or, one (or more) instances of a design unit.
`deselect_congruency`	`deselect_congruency ([[(-design_unit|-module)] <`*`design_unit_name`*`>]) | (-instance <`*`hierarchical_instance_path`*`>)`	Disables congruency for one (or more) design units (modules, interfaces) or, one (or more) instances of a design unit.
`set_log`	`set_log [ -on | -off ]`	Enables or disables logging as explained in the Logging section.
`set_display_default`	`set_display_default [ -on | -off ]`	Makes `$display` present in congruent blocks to show "old" or "updated" value. By default `$display` shows an "old" value.
`set_sc_mc_align`	`set_sc_mc_align [ -on | -off ]`	Aligns single-core congruency implementation with that of multi-core engine. By default, the single-core implementation has its own support set. When this switch is set to `-on`, it might disable support that aligns with the multi-core engine. For example, support for MDA congruent objects is disabled with the switch set to `-on`.

Applying Congruency Commands

If there are multiple `select_congruency` and `deselect_congruency` commands in a configuration file, then each command builds on the previous command.

Generate blocks in a module/instance are considered as part of the module/instance in selection and deselection commands used at elaboration. Selection/deselection cannot be applied on generate blocks explicitly. The arguments to these commands support wildcard expressions and specified hierarchy.

The following wildcards are supported:

Wildcard	Description	Example
`*`	Matches any text from current location until next delimiter or end of string	`select_congruency -module test_mod*` Matches all names beginning with `test_mod`, such as `test_mod1`, `test_module`.
`?`	Matches any single character	`select_congruency -module test_mod?` Matches all names beginning with `test_mod` followed by a single character such as `test_mod1` and `test_mod2`.
...	Matches that module/instance path and all its descendants	`select_congruency -module mod...` Matches all instances of mod and all their descendants.

Examples

The following are examples of the commands that you can use in the configuration file:

- `select_congruency -module m2`

  Enables congruency for all the instance of module `m2`.

- `deselect_congruency -module m2`

  Disables congruency for all the instance of module `m2`.

- `select_congruency -instance top.m1`

  Enables congruency for the instance `top.m1`.

- `deselect_congruency -instance top.m1`

  Disables congruency for the instance `top.m1`.

- `select_congruency -instance top.m1...`
  Enables congruency for the instance `top.m1` and the hierarchy below it.
- `deselect_congruency -instance top.m1...`
  Disables congruency for the instance `top.m1` and the hierarchy below it.
- `select_congruency -module m2...`
  Enables congruency for all instances of module `m2` and the hierarchy below these instances.
- `deselect_congruency -module m2...`
  Disables congruency for all instances of module m2 and the hierarchy below these instances.

Congruency Log File

A log file is generated when a verbose switch is specified, or `set_log [-on]` command is given in congruency configuration file. This log file lists `always` blocks in the design that are found to be congruent (with their control and data signals) and `always` blocks that are not congruent.

```
Always block identified as a congruent flip flop

Instance location:    top.m1
Control signals:      clk
<file name>: <line number>
```

```
Always block not identified as a congruent flip flop

Instance location:    top.m2
Control signals:      clk
<file name>: <line number>
```

The log file is generated under a test directory and named as `sc_flip_flop_identification.log.gz`.

Related Topics

- Solving Race Conditions at Inputs
- Identifying always blocks as sequential and qualification for congruency
- Behavior of congruent signals

Identifying always Blocks as Sequential and Qualification for Congruency

When the `-congruency` option is specified to a solve race condition, every `always` block is checked if it is sequential or not. For every sequential block, you can differentiate between its “control signals” (clk/set/rst/…) and its “data signals”. The simulation uses the old values of the “data signals” when running the `always` block.

The rules for identifying the `always` blocks as sequential conform to section 5.2.2 of IEEE 1364.1 Synthesizable LRM. These sequential blocks are divided in 3 categories:

- Flops without set/reset
- Flops with asynchronous set/reset
- Flops with synchronous set/reset

Identifying Flops without Set/Reset

As per section [5.2.2] in IEEE 1364.1 Synthesizable LRM, an `always` block is identified as FF, which:

- Contains a single edge event
- Writes to its outputs using non-blocking assignments
- Does not contain multiple event list (for example, an event does not appear in the middle of the `always` block)
- May use blocking assignments only for temporary variables (which are used within the block)

⚠ From the above list, bullet 4 does not qualify for congruency with the current implementation.

Identifying Flops with Asynchronous Set/Reset

As per section [5.2.2] in IEEE 1364.1 Synthesizable LRM, an always block is identified as an asynchronous FF, which:

- Contains only edge events
- Has an if/else tree as stated in LRM text, with the conditions having the right polarity (according to whether the edge was posedge or negedge)

- Follows same conditions as stated for FFs without set/reset with respect to writing to output and temporary variables

Identifying Flops with Synchronous Set/Reset

Section [5.2.2] in IEEE 1364.1 Synthesizable LRM states:

"NOTE—No specific style is required to infer edge-sensitive storage device with synchronous set/reset. A synthesis tool may optionally choose to or not to infer such a storage device. See the sync_set_reset attribute on how it can be used to infer a device with synchronous set/reset."

Based on the above description, this implementation identifies flops with synchronous set/reset similarly to regular FFs with a clock (and their synchronous set/reset would be identified as "data signals").

Flop Identification Mechanism

This feature uses existing sequential block analysis in the elaborator. It is expected that this analysis conforms to the earlier sub-sections referred from IEEE 1364.1 Synthesizable LRM.

Behavior of Congruent Signals

For a sequential block (which has been marked congruent), as explained in the above section, the data signals read in it are marked congruent as well. This means, that in the given always block, old value of the data signal is read.

⚠ If this congruent signal is read elsewhere in a non-congruent block (or context), the updated value is read, that is, it does not exhibit congruent behavior.

For the following example, while "old" value of "a" and "b" is read in the sequential block context at line 13, its "updated" value is read in the combinational block at line 9.

```
1 module mod (clk, rst_n, a, b, q, y);
2
3  output q, y;
4  input  clk, rst_n;
5  input  a, b;
6  reg   q, y;
7
8  always @ (a or b)
9    y <= a ^ b;
```

```

 always @ (posedge clk or negedge rst)
   if (!rst_n) q <= 1’b0;
   else        q <= a ^ b;
endmodule
```

Displaying Congruent Signals/Blocks

To fetch the value of a data signal of sequential `always` blocks, various utilities continues to show (output) updated values. Some of them are listed below:

- `$display` system task
- probe, SHM, VCD (other signal dumping means)
- Waveform display

Delays in Assignments

This implementation honors the delay in assignments, as used in the below examples, and is not be affected by them.

```
always @(posedge clk)
# <delay> Q <= d;
always @(posedge clk)
Q <= # <delay> d;
always @(posedge clk)
# <delayi> Q <= # <delay> d;
```

In future, there could be a switch proposed to ignore the delays.

Related Topic

- Solving Race Conditions at Inputs
- Congruency Configuration File

13

Detecting Race Conditions at Inputs of Flip-Flops

Starting Xcelium 22.02-a, you can use the `-race_detect` command-line option to enable race detection of register and wire data objects present in the design. The option does not detect simulation races arising due to scenarios where data read/write is not involved. Refer the support set and limitations in relevant sections of this chapter.

The detected races are reported as warnings in the simulation logs and are explained below.

Simulation Warnings

Mnemonic	Indicates
`RACERAW`	read-after-write race
`RACEWAR`	write-after-read race
`RACEWAW`	write-after-write race
`RACECLKD`	clock-data races for sequential flops (updates using NBA statement)
`RACECTRL`	control-control races for sequential flops

RAW and CLKD Races
In System Verilog, it is recommended to use a blocking assignment (BA) statement to update the clock signal and a non-blocking assignment (NBA) statement to update the data signal. This ensures that the logic that is synchronized to the clock reads the 'old' value of the data. If a clock signal update is detected to be in the NBA region, resulting in the latest value read of the data, `RACECLKD` is flagged instead of the `RACERAW` race type.

> ⚠
> - Since `WAR` and `WAW` races are not expected to affect the simulation behavior; by default, these warnings are not reported, and their respective counts in the summary report at the end of the simulation will be 0, even if those races were present.
> - You can enable the reporting of `RACEWAR` and `RACEWAW` using the `-unique_raceall` option.

These warnings also include information on:

- Simulation time at which the race occurred.
- Complete hierarchical path to the object (wire or register) on which the race occurred.
- The source information (file name, line number) for the contexts involved in the current race scenario.

Elaboration Warning for potential Clock-Data Races

Mnemonic	Indicates
`RDNBACLK`	It indicates that the clock signal is updated using the non-blocking assignment (NBA) which may cause unexpected simulation behavior due to clock-data race condition.

User Configuration File

The user configuration file limits the design hierarchy for which race detection is turned on. The configuration is passed to elaboration using the `-racedt_config` command-line option.

The user configuration file has the following commands:

- `select`

  Enables the race detection for one (or more) design units (module, interfaces) or one (or more) instances of the design unit.

  ```
  select ([[(-design_unit|-module)] <design_unit_name>]) | (-instance
  <hierarchical_instance_path>)
  ```

- `deselect`

  Turns off race detection for one (or more) design units (module, interfaces) or one (or more) instances of the design unit.

```
deselect ([[(-design_unit|-module)] <design_unit_name>]) | (-instance
<hierarchical_instance_path>)
```

- If multiple `select` and `deselect` commands exist in a configuration file, each builds on the previous command.
- Generate blocks and UDPs in a module/instance are considered part of the module/instance in selection and deselection commands used at elaboration. Selection/deselection cannot be applied explicitly to generate blocks/UDPs.

- `qualify_write_op -on/-off`
  By default, `-race_detect` tracks a signal only if it is 'read' accessed. If the signal is never 'read' accessed and only 'write' accessed, it is not qualified for `-race_detect`. Use this setting to enable (`qualify_write_op -on`) 'write' only signals too for `-race_detect`.

Wildcard	Description	Example
`*`	Matches any text from the current location until the next delimiter or end of a string	`select -module test_mod*` Matches all names beginning with `test_mod`, such as `test_mod1`, `test_module`, etc.
`?`	Matches any single character	`select -module test_mod?` Matches all names beginning with `test_mod` followed by a single character, such as `test_mod1` and `test_mod2`.
…	Matches that module/instance path and all its descendants	`select -module mod...` Matches all instances of the `mod` and all of their descendants.

The following are the use cases and their results concerning `-race_detect` and `-racedt_config` switches:

1. When only the `-race_detect` switch is present, race detection is applied to the complete

hierarchy.

2. When only `-racedt_config` switch is present:
   - `select/deselect` command is present in the configuration file, then Race Detection is applied to the specified hierarchy.
   - No `select/deselect` command is present in the configuration file, then Race Detection is not applied to any hierarchy.
3. When both `-race_detect` and `-racedt_config` switches are present:
   - `select/deselect` command is present in the configuration file; then, Race Detection is applied to the specified hierarchy.
   - No `select/deselect` command is present in the configuration file, and then Race Detection is applied to the complete hierarchy.

The following table provides examples of the commands that can be used in the configuration file:

Command	Impact
`(de)select -module m2`	(Dis)enables race detection for all the instances of the module `m2`.
`(de)select -instance top.m1`	(Dis)enables race detection for the instance `top.m1`
`(de)select -instance top.m1...`	(Dis)enables race detection for the hierarchy under the instance `top.m1`, including it
`(de)select -module m2...`	(Dis)enables race detection for the hierarchy under all instances of the module `m2`, including those instances

Race Detector Options

The following options are added to Race Detector:

Option Name	Description
`-race_allowall`	By default, races for the signals present only under Sequential Flops and Sequential UDPs are reported. Use this option to enable all signals for race detection.

`-unique_raceall`	Use this option to enable `RACEWAR` and `RACEWAW` races with `-race_detect`. The `-unique_raceall` option will enable the race counts for `RACEWAR` and `RACEWAW` also, if such races are present.
`-raceall`	Use this option to display all race warnings. Presently, unique races are displayed by default.
`-race_entime0`	Use this simulation option to enable races for a simulation time of 0FS. By default, `-race_detect` does not report any time 0 races.
`-raceoutput`	Use this option to dump all warnings and messages related to race detection into a separate file. Example: `-raceoutput dump_race.out`

Related Topics

- Race Detector Support Set
- Limitations of Race Detector
- Race Detector Log File and Summary
- Race Conditions Examples

Race Detector Support Set

Starting Xcelium 24.09-agile release, Race detection supports MSIE and monolithic flow.

Support for Edge-sensitive Sequential UDPs

Starting Xcelium 22.10-a release, the support of `-race_detect` is extended to data object updates by edge-sensitive sequential User Defined Primitives (UDPs). It is important to note that:

- Data object updates of Mixed UDPs (with both level-sensitive and edge-sensitive table entries) are not supported. However, if the output field does not change (i.e., '`-`') for each level-sensitive row, then that will be considered pure edge-sensitive sequential UDP, and thus, data object updates of such UDPs will be supported for `-race_detect`.

- Race warning is reported only when the output of UDP changes. If no change in UDP output is detected, no race will be reported, even if the input UDP terminal has a race.

Example 4 and Example 5 illustrate the scenarios.

Support for Testbenches

Starting with the Xcelium 24.03-a release, the support for the register and wire data objects present in the SystemVerilog program, assertion, and checker scope is added. The tool now supports race detection for the signal updates in the reactive region set and re-NBA (Non-blocking Assignments) region set. See Example 10 for more information.

⚠ The re-inactive region in a testbench is not supported.

Related Topics

- Limitations of Race Detector
- Race Detector Log File and Summary
- Race Conditions Examples

Limitations of Race Detector

The following limitations apply to Race Detector:

- It only warns for the races of register and wire signals.
- Any external read/writes on signals, like force/deposit/VPI, are not supported.
- Processes/signals with the following conditions are not qualified for race detection:
  - Self-triggering always blocks
  - Signals that are of 'auto' scope
  - Signals that are 'formal' arguments of functions/tasks
- Control-control races are not supported for wires. Currently, it is only supported for registers.

- Races with dynamic datatypes, such as dynamic arrays, associative arrays, queues, etc., are not supported.
- Races that do not involve data read/write at the point of occurrence are not reported.

  For example:

```
// Oscillator
// There is a race between call of task 'osc' and disabling of it.
always @(pllPowerDown) begin
   if(pllPowerDown) begin
      disable osc;
      vco_clk = pVcoPowerDownVal;
   end
end

always @(pllPowerDown) begin
   if(!pllPowerDown)
     #0 osc;
end
```

- All warnings, except for `RACECTRL`, are reported only if multiple read/write operations are detected on the same signal. A race occurring due to an update of two different signals is not supported.

  For example:

```
// If valid and pd are changed in the same region and timestamp,
// then result1 may observe a glitch.

assign valid_p1 = valid;
assign pd_p1 = pd;
assign result1 = valid & pd_p1;
```

- Races due to static variable initialization of a class member are not supported.

  For example:

```
function int f();
   return A::stable ? 1 : 0;
endfunction

static bit y = f();     // Value of y could be 0 or 1, a race

class A;
  static bit stable = 1;
endclass
```

Related Topics

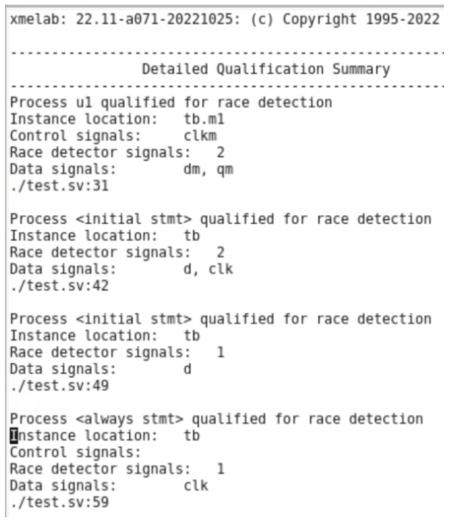
- Race Detector Support Set
- Race Detector Log File and Summary
- Race Conditions Examples

Race Detector Log File and Summary

Race Detector Log File

This log file is generated when the `-RACEDT_VERBOSE` option is added to the command line or the `set_log [-on]` command is given in the configuration file. This log file lists RTL processes in the design that are qualified for race detection (with their control signals) and those that do not qualify.

This is displayed in the example below:



```
xmelab: 22.11-a071-20221025: (c) Copyright 1995-2022

---------------------------------------------------------
                Detailed Qualification Summary
---------------------------------------------------------
Process u1 qualified for race detection
Instance location:   tb.m1
Control signals:     clkm
Race detector signals:   2
Data signals:        dm, qm
./test.sv:31

Process <initial stmt> qualified for race detection
Instance location:   tb
Race detector signals:   2
Data signals:        d, clk
./test.sv:42

Process <initial stmt> qualified for race detection
Instance location:   tb
Race detector signals:   1
Data signals:        d
./test.sv:49

Process <always stmt> qualified for race detection
Instance location:   tb
Control signals:
Race detector signals:   1
Data signals:        clk
./test.sv:59
```

The log file is generated under the test directory and named *rd_flip_flop_identification.log.gz*.

Race Detector Summary

At the simulation exit, a summary section is displayed in the simulation log file or in the output file specified with `-raceoutput`.

This is displayed in the example below:

```
xmsim: *N,RDSUMRP: Race Detection Summary
--------------------------------------------
    Number of races reported : 3
    Read-After-Write     : 3          Write-After-Read     : 0          Write-After-
Write    : 0    Clock-Data-Race      : 0

--------------------------------------------
```

⚠ In a scenario when no race is found, the following Note is displayed:

```
xmsim: *N,RDSUMRPN: No races were found.
```

Race Conditions Examples

Example 1: Read-After-Write race scenarios for sequential flops

```
module d_ff(q,d,clk,rst);
 output reg q;
 input d,clk,rst;

 always@(posedge clk)
         if(!rst)
             q<=1'b0;
         else
             q<=d;

 endmodule

 module tb;
 wire q;
 reg d,rst;
 reg clk=1'b0;
 d_ff inst(q,d,clk,rst);

 initial
     begin
        rst=0;
     #5
         rst=1;d=0;
     #2
         d=1;
     #10
         $finish;
     end
 always
 #1 clk=~clk;
 endmodule
```

Run Command:

```
xrun -sv test.sv -race_detect -raceall -log xrun.log
```

Race Detection Output:

```
xmsim: *W,RACERAW: Detected a race at time 5 NS + 1 on signal
"tb.inst@d_ff<module>.rst". Signal read at ./test.sv:6 after being written at
./test.sv:17.
xmsim: *W,RACERAW: Detected a race at time 5 NS + 1 on signal "tb.inst@d_ff<module>.d".
Signal read at ./test.sv:9 after being written at ./test.sv:17.
xmsim: *W,RACERAW: Detected a race at time 7 NS + 1 on signal "tb.inst@d_ff<module>.d".
Signal read at ./test.sv:9 after being written at ./test.sv:17.
```

Example 2: Read-After-Write race scenarios for sequential flops

```
module top(input clk, input in0, output reg out0);
     reg temp1;

     always @(posedge clk)
         temp1 = in0;
     always @(posedge clk)
         out0 <= temp1;
 endmodule

 module tb();
     reg d, q;
     reg clk = 1'b0;

     always #5 clk = ~clk;

     initial begin
         #5 d = 1'b1; #15 $finish;
     end

     top t(clk, d, q);
 endmodule
```

Run Command:

```
xrun -sv test.sv -race_detect -log xrun.log
```

Race Detection Output:

```
xmsim: *W,RACERAW: Detected a race at time 5 NS + 1 on signal "tb.t@top<module>.in0".
Signal read at ./test.sv:5 after being written at ./test.sv:20.
```

Example 3: Clock data (RACECKLD) race scenarios for sequential flops

```
module test();
  reg clk=1'b0,dclk=1'b0;
  always #5 clk =~clk;
  reg d ,d1=1'b1;
  reg q;
always@(posedge clk)
    dclk<=~dclk;
always@(posedge clk)
    d<=d1;
always@(posedge dclk)
    q<=d;
initial
     #20 $finish;
endmodule
```

Run Command:

```
xrun -sv -clean -access +rwc -race_detect -racedt_verbose test.sv -log_xmsim xmsim.log
```

Race Detection Output:

```
xmsim: *W,RACECLKD: Detected a Clock-Data race at time 5 NS + 1 with clock "test.dclk"
written at ./test.sv:7 and data "test.d" written at ./test.sv:9 which impacts
./test.sv:11.
```

Example 4:   Read-After-Write race scenarios for Sequential UDPs

```
primitive udp (qu, clku, du);
output qu;
input clku, du;

reg qu;

//initial qu = 0;

table
// obtain output on rising edge of clku
// clku         du        qu       qu+
   (01)         0   :   ?   :   0   ; //line12
   (01)         1   :   ?   :   1   ; //line13
   (0?)         1   :   1   :   1   ;
   (0?)         0   :   0   :   0   ;
// ignore negative edge of clk
   (?0)         ?   :   ?   :   -   ;
```

```
// ignore d changes on steady clk
   ?          (??)      :   ?   :   -   ;
endtable

endprimitive


module m (qm, clkm ,dm);
input clkm, dm;
output qm;

udp u1 (qm, clkm, dm);

endmodule

module tb;
reg  d, clk;
wire q;


m m1 (q, clk, d); // line 38

initial begin
$monitor ("t=%0t q=%d clk=%d d=%d ", $time,q,clk,d );
clk =0;
d=0;
#11 $finish ();
end

initial begin

#2 d=1; // line 49
#1 d=0; // line 50
#2 d=1;
#3 d=0;
#1 d=1;
#1 d=0;
end

always #1 clk = ~clk;

endmodule
```

Race output:

```
xmsim: *W,RACERAW: Detected a race at time 3 NS + 1 on signal "tb.d". Signal read at
./test.sv:31 after being written at ./test.sv:40.
```

Example 5: Read-After-Write race scenarios for Sequential UDPs (having not pure edge-sensitive UDP)

```
primitive ffudpt(Q, D, C, NOTIFIER);
output Q;
input D, C;
  reg Q;
input NOTIFIER;

  table
  // D   C   NOTIFIER : reg :  Q  ;
     0   r      ?     :  ?  :  0  ; // capture 0
     1   r      ?     :  ?  :  1  ; // capture 1
     ?   n      ?     :  ?  :  -  ; // ignore falling edge
     *   ?      ?     :  ?  :  -  ; // stable on static clock
     0   *      ?     :  0  :  -  ; // reduce pessimism
     1   *      ?     :  1  :  -  ; // reduce pessimism
     ?   ?      *     :  ?  :  x  ; // any NOTIFIER change;
  endtable
endprimitive


primitive mux2udpt
(Z, S, D0, D1);
output Z;
input S, D0, D1;

table
//      S  D0 D1   Z
        0  0  ?  : 0 ;
        0  1  ?  : 1 ;
        0  x  ?  : x ;
        1  ?  0  : 0 ;
        1  ?  1  : 1 ;
        1  ?  x  : x ;
        x  0  0  : 0 ;
        x  1  1  : 1 ;

endtable
endprimitive
```

```
module no_x_out (out, in, clk);
    input in;
    input clk;
    output out;
    parameter NY_NO_X_OUT_ASSIGN = 0;
      reg out;
      generate if (NY_NO_X_OUT_ASSIGN == 1) begin
        always @(posedge clk) begin
          out = in;
        end
      end else begin
        always @(posedge clk) begin
          out <= in;
        end
      end
      endgenerate
endmodule

module top (
  C,
  CP,
  SE,
  SI,
  Q);

  output Q ;
reg notifier;
   input C ;
   input CP ;
   input SE ;
   input SI ;

  parameter NY_STDCELL_DLY_C2Q = 0;
  parameter NY_NO_X_OUT_ASSIGN = 1;

  wire    q_l1;
  wire D_AND_CBar;
  wire CBar;
  wire QN;
      wire Q_INT_temp;

  not n0 (CBar, C);
  and ad0 (D_AND_CBar, QN, CBar);

 mux2udpt u0(Z,SE,D_AND_CBar,SI);
```

```
ffudpt ff(Q_INT_temp, Z, CP, notifier);
//no_x_out #(NY_NO_X_OUT_ASSIGN) u_no_x_Q1 (Q_INT, Q_INT_temp);
no_x_out  u_no_x_Q1 (Q_INT, Q_INT_temp, CP);

  buf   bq(Q, Q_INT);
  not   nq(QN, Q_INT);

endmodule


module tb;
reg C, CP, SE, SI;
wire Q;
top top_inst (C, CP,SE,SI, Q);

initial begin
CP=1'b0;
C=1'b0;
SE=1'b0;
SI=1'b0;
#5;
C=1'b1;
SI=1'b1;
SE=1'b1;
#5 $finish ();
end
always #1 CP = ~CP;

endmodule
```

Race Output:

```
xmsim: *W,RACERAW: Detected a race at time 7 NS + 1 on signal
"tb.top_inst@top<module>.u_no_x_Q1@no_x_out<module>.in". Signal read at ./test.sv:52
after being written at ./test.sv:85.
```

In Example 5, '`mux2udpt`' UDP did not qualify for `-race_detect`, as it was not pure edge-sensitive UDP.

Example 6: Read-After-Write race scenarios for sequential flops (struct aggregate)

```
1 module top;
  typedef struct {
    reg r_pack;
    reg clk;
  } s;
  s s1 = '{1'b0 , 1'b1};
endmodule


module test();
  reg clk=1'b0,r1;
always #2 clk=~clk;
always@(posedge clk)
  top.s1.r_pack=1'b1;
always@(posedge clk)
  r1=top.s1.r_pack;
initial
    #8 $finish;
endmodule
```

Race Output:

```
xmsim: *W,RACERAW: Detected a race at time 2 NS + 0 on signal "top.s1" at "{r_pack}".
Signal read at ./test.sv:16 after being written at ./test.sv:14.
```

Example 7: Read-After-Write race scenarios for sequential flops (array aggregate)

```
module top;
 reg [15:0]r_pack;
endmodule

module test();
reg clk=1'b0,r1;
reg [0:2]c=3'd4;

always #2 clk=~clk;

always@(posedge clk)
    begin
   top.r_pack[c]=2'b11;
   $display("reg c = %d",c);
   end
always@(posedge clk)
   r1=top.r_pack[4];
initial
    #20  $finish;
endmodule
```

Race Output:

```
xmsim: *W,RACERAW: Detected a race at time 2 NS + 0 on signal "top.r_pack" at "[4]".
Signal read at ./test.sv:17 after being written at ./test.sv:13.
```

Example 8: With different indices (no race expected)

```
module top;
 reg [15:0]r_pack;
endmodule

module test();
reg clk=1'b0,r1;
reg [0:2]c=3'd4;

always #2 clk=~clk;

always@(posedge clk)
    begin
   top.r_pack[c]=2'b11;
   $display("reg c = %d",c);
   end
always@(posedge clk)
   r1=top.r_pack[5];
initial
    #20  $finish;
endmodule
```

Race Output:

```
xmsim: *N,RDSUMRPN: No races were found.
```

Example 9: Clock-clock race scenarios for sequential flops

```
module test();
reg clk=1'b0,d=1'b0, reset=1'b1;

always #5 clk =~clk;

initial #5 reset = ~reset;

always@(posedge clk or posedge reset)
if (reset)
  d = 0;
else
  d<=!d;

initial begin
   #2 $display(d);
   #18 $finish;
end

endmodule
```

Run Command:

```
xrun -sv -clean -race_detect -racedt_verbose test.sv
```

Race Output:

```
always@(posedge clk or posedge reset)
     |
xmsim: *W,RACECTRL (./test1.sv,8|5): Detected a Control-Control race at time 5 NS + 1
(or at the preceding time-slot) for the above shown process. Details of control signals
changing simultaneously is as below.
  test.clk changed at ./test1.sv:4
  test.reset changed at ./test1.sv:6
```

Example 10: Support for Testbench

```
module test();
reg clk=1'b0;
always #5 clk=~clk;
logic l1,l2;
reg a=1'b1;
property p1;
 @(posedge clk) !a;
endproperty
c1: assert property(p1)
  else
    begin
       l1=1'b1; //12 line no.
    end
c2:assert property(p1)
    else
      begin
       l2=l1;  //17 line no.
      end
endmodule
```

Run Command:

```
xrun -sv test.sv -clean -race_detect -racedt_verbose -unique_raceall -nodeadcode -
race_allowall
```

Race Output:

```
xmsim: *W,RACEWAR: Detected a race at time 45 NS + 1 on signal "test.l1". Signal
written at ./test.sv:12 after being read at ./test.sv:17.
```

Example 11: Support with MSIE

```
package my_pck;
reg latch_out;
reg q_out;
reg clk_r = 0;
reg d = 0;
reg reset = 1;
endpackage
module latch_and_flop(input wire d_async, input wire clk, input wire nreset, output
q);
import my_pck::*;
 always @(*)
   if(!nreset)
```

```
     latch_out <= 0;
     else if(clk)
     latch_out <= q_out;

     always@(posedge clk or negedge nreset) begin
     if(!nreset)
       q_out <= 0;
       else
       q_out <= d_async;

     end
     assign q =  latch_out;

endmodule


module top;
import my_pck::*;

wire clk;
wire d_async =d;
wire nreset = reset;

assign clk = clk_r;

always #2 clk_r = ~clk_r;

wire q;

    latch_and_flop l1(d_async,clk,nreset,q);

    initial $monitor("%t q = %d clk = %d d = %d",$time,q,clk,d_async);
    initial #20 $finish;
    initial begin
        #10 d = 1;
    end

endmodule
```

Run Command:

```
xrun -reduce_messages -race_detect -racedt_verbose -sv -clean -automsie test.sv
```

Race Output:

```
q_out <= d_async;
```

```
              |
xmsim: *W,RACERAW (.test.sv,20|14): Detected a race at time 10 NS + 1 on signal
"top.l1.d_async", read at statement shown above after being written at ./test.sv:33.
```

14

Using PLI

The Programming Language Interface (PLI) is a public-domain C-language procedural interface and interface mechanism that lets you access and modify data dynamically in the data structure that results from compiling Verilog HDL source descriptions and elaborating the design hierarchy. The PLI provides a library of C-language functions that can directly access data within the data structure.

Some applications of the Programing Language Interface include:

- Customized debugging routines
- Applications that read data, such as test vectors, from a file and that pass the data to the simulator
- Simulation models written in C and dynamically linked into a Verilog simulation
- Event-driven callback routines
- Delay calculators
- Backannotating routines

The PLI has been standardized by the IEEE. The standard is described in the *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*.

The IEEE standard describes the PLI routines in terms of three generations:

- Task/function routines (TF routines)
  These routines are used primarily for operations involving user-defined system task/function arguments and for utility functions, such as setting up callback mechanisms and writing data to output devices. The TF routines are sometimes called *utility* routines.
- Access routines (ACC routines)
  These routines provide access into a Verilog HDL structural description. The ACC routines are used to access and modify information, such as delay values and logic values on a wide variety of objects that exist in a Verilog description.

  The TF and ACC routines are sometimes referred to as PLI 1.0.

In addition to the information on the PLI TF and ACC mechanism contained in the IEEE standard, refer to the PLI 1.0 User Guide and Reference for details on these routines.

- Verilog Procedural Interface (VPI) routines
  These routines provide an object-oriented access for both Verilog HDL structural and behavioral objects. The VPI routines are a superset of the functionality of the TF and ACC routines.

  Using the VPI rather than PLI 1.0 is strongly recommended because VPI is more complete, is easier to learn and to use, and can result in better simulation performance.

To use the PLI with the Xcelium™ Simulator, you write a standard C language application that calls PLI routines. You then integrate your application either by compiling and linking the application and the simulator object modules into a new set of executables (static linking), or by compiling and linking the application into a shared library (dynamic linking). If a PLI application has been compiled into a dynamic shared library, you can use the `-loadpli1` or `-loadvpi` command-line option to load the library and to register the system tasks defined in the application at run time.

Related Topic

- VPI/VHPI Reference

PLI Map File

A PLI map file associates user-defined system tasks and system functions with functions in a PLI application. The file contains a line for each user-defined system task or system function your application needs. In each line, you specify:

- The name of the system task or system function.
- Additional specifications for the system task or system function.

  For a user-defined system function, you must specify the size of the return value.

  Other optional specifications include the name of the call function, the name of the check function, the name of the miscellaneous function, and the data value passed as the first argument to the call, check, and miscellaneous routines.

The PLI map file can be created separately, which you can include at elaboration time using the `-afile` option or at simulation time with the `-plimapfile` option. If passed at elaboration time, the system tasks and functions defined in the file are known to both *xmelab* and *xmsim*. If passed at simulation time, the system tasks and functions defined in the file are known only to *xmsim*.

```
xmelab -afile plimapfile.file ...
xrun -afile plimapfile.file ...
```

```
xmsim -plimapfile plimapfile.file ...
xrun -plimapfile plimapfile.file ...
```

You can also include the information in an *access file*. An access file is a text file that lets you specify the type of access (read, write, connectivity) that you want for particular instances and portions of the design. An access file must be included at elaboration time, so if you include the PLI map information in an access file, use the `-afile` option, as shown above.

Related Topics

- Access to Simulation Objects
- Using a PLI/VPI Map File

Debugging PLI Applications

Running the simulator with PLI applications can sometimes result in errors or in a crash. This topic provides information on how to debug these problems.

Multi-Step Mode

If running the simulation using direct invocation (multi-step) mode—with separate tools to compile, elaborate, and simulate the design—you can debug a PLI-related crash as follows:

1. Invoke the *gdb* debugger on *xmsim*.

   ```
   % gdb xmsim
   ```

2. Set a breakpoint on the `ncdbg_fatal` function. This function is called immediately after a fatal error has been encountered and reported in your application but before the executable calls `exit()`.

   ```
   (gdb) b ncdbg_fatal
   ```

3. Specify the following `handle` commands so that the debugging session can continue without interruption by a `SIGSEGV` or `SIGBUS` signal.

   ```
   (gdb) handle SIGSEGV pass nostop noprint
   (gdb) handle SIGBUS pass nostop noprint
   ```

4. Specify the `run` command to run *xmsim* . For example:

```
(gdb) run xmsim_options snapshot_name
```

or:

```
(gdb) run -f arguments_file
```

5. Specify the `where` command to see a stack trace leading back to the code where the problem occurred.

   ```
   (gdb) where
   ```

Single-Step Mode Using XRUN

If running the simulator in single-step mode—with the *xrun* command—you can debug a PLI-related crash as follows:

1. Invoke *gdb* on *xmsim*.

   ```
   % gdb xmsim
   ```

2. Set a breakpoint on the `ncdbg_fatal` function, and specify the same `handle` commands, as shown for multi-step mode above.

   ```
   (gdb) b ncdbg_fatal
   (gdb) handle SIGSEGV pass nostop noprint
   (gdb) handle SIGBUS pass nostop noprint
   ```

3. Run the simulator using the `xmsim.args` file that *xrun* generated in the `xcelium.d/run.d` directory.

   ```
   (gdb) run -f xcelium.d/run.d/xmsim.args
   ```

4. Specify the `where` command to see a stack trace leading back to the code where the problem occurred.

   ```
   (gdb) where
   ```

Debugging with C++ Dynamic Libraries

In some cases, when using C++ dynamic libraries, the debugger cannot properly handle symbolic information in the dynamic library and will not provide the necessary debug information. A simple workaround for this issue is to build static executables.

To debug the PLI crash:

1. Remove the `xcelium.d` directory.

   ```
   % rm -rf xcelium.d
   ```

2. Rerun `xrun`, including the following options on the command line:

```
% xrun +xmelabexe+./ncelabC +xmsimexe+./ncsimC -f run.args
```

> Both the plus options `+xmelabexe` and `+xmsimexe` and minus options `-xmelabexe` and `-xmsimexe` are supported.

3. Invoke the debugger, but debug using `ncsimC`.

```
% gdb ./ncsimC
(gdb) b ncdbg_fatal
(gdb) handle SIGSEGV pass nostop noprint
(gdb) handle SIGBUS pass nostop noprint
(gdb) run -f xcelium.d/snap.nc/xmsim.args
    Breakpoint 1, ncdbg_fatal ()
(gdb) where
```