Disclaimers

Lattice makes no warranty, representation, or guarantee regarding the accuracy of information contained in this document or the suitability of its products for any particular purpose. All information herein is provided AS IS, with all faults and associated risk the responsibility entirely of the Buyer. Buyer shall not rely on any data and performance specifications or parameters provided herein. Products sold by Lattice have been subject to limited testing and it is the Buyer's responsibility to independently determine the suitability of any products and to test and verify the same. No Lattice products should be used in conjunction with mission- or safety-critical or any other application in which the failure of Lattice's product could create a situation where personal injury, death, severe property or environmental damage may occur. The information provided in this document is proprietary to Lattice Semiconductor, and Lattice reserves the right to make any changes to the information in this document or to any products at any time without notice.
Contents

Acronyms in This Document ........................................................................................................................................6
1. Introduction ...............................................................................................................................................................7
  1.1. Purpose and Overview ........................................................................................................................................7
  1.2. Audience .............................................................................................................................................................7
2. Important Things to Know .........................................................................................................................................8
  2.1. Simulation Flow Overview ................................................................................................................................8
    2.1.1. General Simulation Flow ............................................................................................................................8
    2.1.2. Project vs Directory-Based Simulations ....................................................................................................8
  2.2. Precompiled Device Libraries ...........................................................................................................................9
  2.3. Encrypted Files ..................................................................................................................................................10
3. Recommended Usage Flows ......................................................................................................................................12
  3.1. ModelSim User Interface ................................................................................................................................12
    3.1.1. Initial Setup (Project-based Flow) .............................................................................................................12
    3.1.2. Initial Setup (Directory-based Flow) .........................................................................................................14
    3.1.3. Simulation Flow ...........................................................................................................................................14
  3.2. Simulation Wizard ...............................................................................................................................................14
    3.2.1. Initial Setup ................................................................................................................................................15
    3.2.2. Simulation Flow #1 (from ModelSim User Interface) .................................................................................17
    3.2.3. Simulation Flow #2 (from Radiant User Interface) ......................................................................................17
  3.3. Simulation Wizard and Custom Script ...............................................................................................................19
    3.3.1. Simulation Flow (ModelSim User Interface) .............................................................................................19
  3.4. Simulation Script ...............................................................................................................................................19
    3.4.1. Project-based Simulation Scripts ..............................................................................................................19
    3.4.2. Directory-based Simulation Scripts .........................................................................................................20
  3.5. Propel Verification Projects .............................................................................................................................21
    3.5.1. Initial Setup ................................................................................................................................................21
    3.5.2. Simulation Flow (ModelSim User Interface) .............................................................................................23
    3.5.3. Simulation Flow (Propel Builder User Interface) ......................................................................................24
4. ModelSim Usage Tips ..............................................................................................................................................25
  4.1. Tips for Script Based Simulations ......................................................................................................................25
    4.1.1. Do versus TCL Scripts ................................................................................................................................25
    4.1.2. Invoking Scripts ...........................................................................................................................................25
    4.1.3. Compiling Multiple Files ............................................................................................................................26
  4.2. Simulation Tips and Tricks ................................................................................................................................27
    4.2.1. Simulation Configurations ...........................................................................................................................27
    4.2.2. Saving and Loading user interface Layouts ...............................................................................................29
    4.2.3. Creating a Waveform Do Script .................................................................................................................31
    4.2.4. Creating and Using Custom Libraries .......................................................................................................32
  4.3. Simulating Different Process Stages ...................................................................................................................34
    4.3.1. Generating Post-RTL Simulation Files .......................................................................................................34
    4.3.2. Using Post-RTL Simulation Netlists ........................................................................................................35
    4.3.3. Simulating Post-RTL with Timing ...............................................................................................................36
5. Important TCL Commands and Options ...............................................................................................................38
  5.1. add wave ............................................................................................................................................................38
  5.2. cd ........................................................................................................................................................................38
  5.3. do .........................................................................................................................................................................39
  5.4. project .................................................................................................................................................................39
  5.5. run ......................................................................................................................................................................40
  5.6. vcom ...................................................................................................................................................................41
  5.7. view .....................................................................................................................................................................41
5.8. vlib .......................................................................................................................42
5.9. vlog .......................................................................................................................42
5.10. vmap ....................................................................................................................44
5.11. vsim ....................................................................................................................44
5.12. wave ....................................................................................................................45
6. Useful TCL Commands and Options .....................................................................47
   6.1. formatTime ........................................................................................................47
   6.2. layout load ........................................................................................................47
   6.3. onbreak ..............................................................................................................48
   6.4. onelaberror ......................................................................................................48
   6.5. onerror ..............................................................................................................48
   6.6. onfinish .............................................................................................................48
   6.7. precision ..........................................................................................................49
   6.8. quietly ..............................................................................................................49
   6.9. quit ....................................................................................................................49
   6.10. radix .................................................................................................................50
   6.11. verror .............................................................................................................50
   6.12. where ..............................................................................................................51
7. Example Scripts ......................................................................................................52
   7.1. Directory Based Simulation Script .................................................................52
   7.2. Post-MAP with Timing Simulation Script .......................................................53
   7.3. Waveform Do Script ......................................................................................54
   7.4. Compiling Files to a Custom Library .............................................................55
References ..................................................................................................................56
Technical Support Assistance ..................................................................................57
Revision History ......................................................................................................58
Figures

Figure 2.1. where TCL Command Output in ModelSim's Transcript View ................................................................. 9
Figure 2.2. Example of Session Key for a User Encrypted RTL File............................................................................. 10
Figure 2.3. Public Key Example for Mentor from Radiant 3.2 key.txt File ................................................................. 11
Figure 2.4. Example of Additional IEEE Encrypted File Generated for Diamond IP in Propel ............................... 11
Figure 3.1. Location of the New Project Option in the ModelSim User Interface ..................................................... 12
Figure 3.2. Create a Project Window in the ModelSim User Interface ................................................................. 13
Figure 3.3. Add Item to Project Window from the ModelSim User Interface ......................................................... 13
Figure 3.4. Initial Page of Simulation Wizard ........................................................................................................ 15
Figure 3.5. Second Page of Simulation Wizard .................................................................................................... 15
Figure 3.6. Third Page of Simulation Wizard ....................................................................................................... 16
Figure 3.7. Fourth Page of Simulation Wizard ...................................................................................................... 16
Figure 3.8. Final Page of Simulation Wizard ......................................................................................................... 17
Figure 3.9. Simulation Wizard Project File Location in Radiant ............................................................................. 18
Figure 3.10. Skip to End Button Location on the First Page of Simulation Wizard ............................................... 18
Figure 3.11. Simulation Mode and Debug Mode Parameters for the RISC-V MC IP, v2.2.1 ................................. 22
Figure 3.12. Initialize Memory Setting Location for the System Memory IP, v1.1.2 ................................................ 23
Figure 4.1. Method for Invoking Scripts using the ModelSim User Interface ....................................................... 25
Figure 4.2. Adding a Simulation Configuration to a ModelSim Project ............................................................... 27
Figure 4.3. Simulation Configuration Setup Initial Page .......................................................................................... 28
Figure 4.4. Simulation Configuration Added to Project ......................................................................................... 29
Figure 4.5. Location of the Save Layout As Option .................................................................................................. 30
Figure 4.6. Save Layout as Naming Option ........................................................................................................... 30
Figure 4.7. Layout Setting in the ModelSim User Interface ..................................................................................... 30
Figure 4.8. ModelSim Layout Configuration Window .......................................................................................... 31
Figure 4.9. Save Waveform Display Format Window ........................................................................................... 32
Figure 4.10. Creating a Custom ModelSim Library ................................................................................................ 33
Figure 4.11. Library Creation Window ................................................................................................................ 33
Figure 4.12. Example modelsim.ini Modification to Add Full Library Path .......................................................... 34
Figure 4.13. Radiant Process Toolbar ................................................................................................................ 34
Figure 4.14. Diamond Process Navigator Tab ...................................................................................................... 35
Figure 4.15. SDF Tab View of the Start Simulation Window .................................................................................... 36
Figure 4.16. Add SDF Entry Window .................................................................................................................. 37
## Acronyms in This Document

A list of acronyms used in this document.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUT</td>
<td>Design Under Test</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>OEM</td>
<td>Original Equipment Manufacturer</td>
</tr>
<tr>
<td>TCL</td>
<td>Tool Command Language</td>
</tr>
</tbody>
</table>
1. Introduction

Mentor ModelSim™ is a multi-language simulation environment with support for languages such as VHDL, Verilog, System Verilog, and C. The version of ModelSim that is installed with Lattice Diamond®, Lattice Radiant™, and Lattice Propel™ is a licensed version that can be used to simulate projects as a standalone, or directly from each respective Lattice software tool.

1.1. Purpose and Overview

Project simulation plays a crucial role in the FPGA design process and is one of the main ways to ensure the functionality of a design is correct as it is being developed. Aside from RTL or functional simulations, Lattice Radiant and Lattice Diamond both also support various types of post-synthesis simulations.

The purpose of this document is to introduce the most important information to know before using ModelSim to simulate a project. Aside from that, this document also contains some recommended usage flows for simulating Lattice projects, as well as some additional usage guidelines and tips. The overall goal for the document is for its audience to know how to simulate their projects in ModelSim, either using the user interface or TCL scripts.

1.2. Audience

The intended audience for this document includes design and verification engineers using Lattice FPGA devices. This document is specifically targeted to anyone using Lattice Diamond, Lattice Radiant, or Lattice Propel with varying amounts of experience using simulation tools like ModelSim. The technical guidelines for this document assume that readers have at least some expertise with digital design, FPGAs, and/or embedded systems.
2. Important Things to Know

This section of the document outlines important information to know about how projects can be simulated using ModelSim. Aside from that, some basic information regarding Lattice’s FPGA device libraries and simulating encrypted files are also discussed.

2.1. Simulation Flow Overview

2.1.1. General Simulation Flow

In general, the simulation flow is fairly similar from project to project. Typically, the main requirements are some RTL which consists of a design under test (DUT) and a testbench. Both the testbench and DUT are compiled to a library, usually called work, which is used to indicate the working files for the project. The main difference between the two methods for simulation have to do with how files are organized. Directory-based simulations require more user effort to setup, compile, and simulate, while project-based simulations are more integrated to the ModelSim user interface.

Once the user RTL has been compiled to work, the next step is to invoke the ModelSim VSIM simulator. This is done using the VSIM command, or by selecting Simulate > Start Simulation from the ModelSim menu bar. The VSIM command requires user to specify the testbench user want to simulate.

Additionally, libraries such as work or some of Lattice’s pre-compiled device libraries are also often linked when invoking VSIM. The reason this is because many Lattice IP contain device specific primitives that are within these additional FPGA device libraries. One of the most commonly used libraries for both Diamond, Radiant, and Propel is pmi_work, which contains modules for several Lattice primitives.

Once VSIM has been invoked successfully, the final step is to advance the simulation for some amount of time. The reason for this is because using VSIM only invokes the ModelSim simulator and keeps the simulation at time = 0 until the simulation is advanced. Depending on the project, the next steps from this point onwards will vary depending on how the user wants to observe the simulation results. Generally, however, these next steps involve opening the waveform display and adding waveforms to observe various signals during the simulation.

2.1.2. Project vs Directory-Based Simulations

Ultimately, there are two main flows users can follow to simulate their projects in ModelSim, with the main difference being how files are organized for each method. For directory-based simulation flows, local directories are used to manage the user RTL files for a project. For project-based simulations, the ModelSim user interface is used to manage the user RTL files for a simulation.

2.1.2.1. Advantages and Disadvantages for each Flow

In general, project-based simulations are better integrated with ModelSim since everything can be managed within the ModelSim user interface. The advantage of this is that this simulation flow requires little to no knowledge of ModelSim’s TCL commands or options and enables users to easily manage and simulate their projects via the user interface. The main disadvantage of this flow, however, is that not every TCL command has a corresponding setting in the ModelSim user interface. What this means is that some functionality is lost when simulating with only the ModelSim user interface. Similar to directory-based simulations, users can still automate project-based simulations using TCL scripts. However this requires different TCL commands and is not as straightforward as it is for directory-based simulations.

The advantage of directory-based simulation flows is essentially the opposite of project-based simulations. Directory-based simulations can be more straightforward to setup, but also require a better understanding of ModelSim’s various TCL commands and options. The ModelSim user interface can also be used to simulate directory-based projects, however this method is more complicated than it is for project-based simulations, as directory-based simulations are not as integrated to the ModelSim user interface as project-based simulations are. Because of this, TCL scripts are almost always used to manage, compile, and simulate the files for directory-based simulations.
2.1.2.2. Managing Libraries and Settings

The first thing to know about ModelSim’s settings is that they are stored in a master file called `modelsim.ini`. Each of Lattice’s software tools (Diamond, Radiant, and Propel) have their own `modelsim.ini` file corresponding to the settings for the ModelSim version that is associated with that tool’s installation.

When setting or library changes are made within the ModelSim user interface, those changes are saved to the active `modelsim.ini` file. Similarly, users can also directly modify these setting files to set some settings. This second method for editing the setting file is recommended, as not all settings within the active `modelsim.ini` file can be modified within the ModelSim user interface.

One of the main differences between project-based and directory-based simulation flows is the format for their respective `modelsim.ini` files. For directory-based simulations, `vmap -c` is often used to create a copy of the master `modelsim.ini` file in the current directory. The reason for this is so any library or setting modifications made are specific for that simulation, and do not impact any other simulations. This local copy of the `modelsim.ini` file contains all the same settings as the active `modelsim.ini` file and references the library mappings from it as well.

For project-based simulations, the settings from the active `modelsim.ini` file are copied to the `<project name>.mpf` file that is generated when the project is created. The contents of this file are very similar to a local `modelsim.ini` file, and contain all the same settings and library mappings, and can also be modified in the same way.

Depending on whether user using a directory-based, or project-based simulation flow, or are not currently simulating a project, the active `modelsim.ini` file may change. If a project is open, the `.mpf` file associated with that project will be the one active. If there is no project open, then ModelSim will default to the `modelsim.ini` file in the current directory. If there is no `modelsim.ini` in the current directory, then ModelSim resorts to the master `modelsim.ini` file as the active setting file.

**Note:** an easy way to tell the active setting file is to use the `where` TCL command as shown in Figure 2.1.

```
ModelSim> where
# Current directory is: C:/Users
# Project is: C:/lsoc/radiant/3.2/modeltech/win32loem//./modelsim.ini
```

**Figure 2.1. where TCL Command Output in ModelSim’s Transcript View**

Below is the order of precedence for an active `modelsim.ini`: `<project name>.mpf`:

- Local `modelsim.ini`
- Master `modelsim.ini`

2.2. Precompiled Device Libraries

As mentioned before, each version of ModelSim within each Lattice software tool installation has its own unique library mappings within its master `modelsim.ini` file. These precompiled libraries are crucial, as they contain simulation models for many Lattice specific primitives and hard IP blocks. Without these precompiled libraries, ModelSim is unable to resolve the references to various primitives within the design, which can cause simulations to error out.

The same is true for non-ModelSim simulators, which also require these device specific Lattice simulation libraries as well to resolve undefined instances. Because of this, the `cmpl_libs` TCL script needs to be used to compile Lattice’s device libraries for use in a separate third-party simulation tool, since that tool does not have these crucial libraries installed by default. For more information about the `cmpl_libs` TCL command, what simulators it supports library compilation for, and how it can be used to compile Lattice’s device libraries for use in a third-party tool, refer to the Radiant or Diamond web help pages.

One of the most used types of precompiled libraries are device-specific libraries, such as `lifcl` for CrossLink™-NX, or `machxo3d` for the MachXO3D™ family devices. These libraries are often required when device specific IP is involved in a design, as it contains the references to the primitives that are used within that IP. For Diamond, there are two type device libraries: Verilog and VHDL. Verilog device libraries are denoted with the `ovi_` prefix (such as `ovi_machxo3d`), while VHDL libraries do not have any prefix (such as `machxo3d`). Functionally, there should be no difference between the two libraries, so either can be used to the same effect.
Although Radiant also has ovi and non-ovi device libraries, both are compiled using Verilog source files. For example, this means there is no difference between libraries such as lifcl and ovi_lifcl in Radiant other than their names. Propel Builder follows a similar naming format, but only contains the Verilog device libraries for both Diamond and Radiant.

Aside from device specific libraries, two more important libraries to know about are pmi_work and uaplatform. Similar to the device libraries mentioned before, these two libraries also contain Lattice specific primitives. However, the main difference is that these primitives are common between multiple device families and are not necessarily unique to a single family. If a simulation is unable to resolve some instances despite linking the device libraries mentioned before, these two libraries are good options to try linking to resolve those missing instances.

For more information on how to create and compile to a custom library in ModelSim, refer to Compiling Files to a Custom Library.

One last important reason to use Lattice’s precompiled device libraries is because each version of ModelSim that is included with Diamond, Radiant, or Propel has an instance limit in place, which causes performance to degrade once the limit is passed. Because Lattice’s precompiled device libraries are watermarked, they do not count towards this instance limit. For this reason, it’s important that the pre-compiled device libraries are used when applicable, to prevent any performance degradation from happening.

2.3. Encrypted Files

This section covers how ModelSim is used to simulate encrypted files, and not how to encrypt the user RTL itself. For more information about encrypting user RTL, refer to the Radiant web help page for the encrypt_hdl TCL command.

The main thing to know about code that has been encrypted, is that there are two main keys required for a third-party tool to be able to read user encrypted RTL. The first key is the called the session key, and is located within the encrypted RTL file as shown in Figure 2.2. If a file was encrypted using the encrypt_hdl command, there are several session keys in the encrypted RTL file. Each session key that is generated corresponds to a key for that a different third-party vendor can use to decrypt the encrypted RTL.

![Figure 2.2. Example of Session Key for a User Encrypted RTL File](image-url)
Aside from the session key, the other important key for third-party tools like ModelSim to be able to read encrypted RTL is the public key. Public keys can be located in either the encrypted RTL itself, or in a separate key file. The public keys for Lattice’s various supported third-party vendors can be found in a key.txt file within each tool’s installation directory. Figure 2.3 shows the key owner and key name fields for each third-party vendor’s public key matches the session key. However, the keys themselves are different.

![Vendor the key is for](image)

**Figure 2.3. Public Key Example for Mentor from Radiant 3.2 key.txt File**

Below is the list of key.txt file for various Lattice tools:

- Radiant – `<Radiant install path>/ispfpga/data/key.txt`
- Propel – `<Propel install path>/builder/rtf/ispfpga/data/key.txt`

As in the list of key locations above, Diamond does not have a key.txt file like Propel or Radiant. The reason for this is because Diamond does not support IEEE1735 encryption. For encrypted IP that are generated in Propel, the default IEEE encryption is converted to blowfish encryption so the project can be synthesized in Diamond while still being encrypted. A limitation of Diamond’s default encryption, however, is that it is not supported by ModelSim. Because of this, Propel generates an additional set of RTL files for each Diamond IP specifically for simulation. **Figure 2.4** shows the IEEE encrypted version of the IP that is intended for simulation is denoted using `_ieee1735`.

![Figure 2.4. Example of Additional IEEE Encrypted File Generated for Diamond IP in Propel](image)
3. Recommended Usage Flows

There are several different flows that can be followed to simulate a project. This section of the document introduces some of the various simulation flows and explains the main advantages and disadvantages for each.

3.1. ModelSim User Interface

For user interface-based simulations, there are two slightly different set of steps that are required to setup and simulate a project. In this section of the document, the initial setup for project-based and directory-based simulations are covered.

3.1.1. Initial Setup (Project-based Flow)

To set up based on project:

1. Go to File > New > Project to create a new project.

2. Define a Project Name and select a Project Location.
   a. (Optional) Choose a different Default Library Name than work.
   b. (Optional) Select a different modelsim.ini to copy from using the Copy Settings From field.
      - Copy Library Mappings – copies the library mappings from the selected modelsim.ini file, directly into the <project name>.mpf that is created.
      - Reference Library Mappings – references the library mappings from the selected modelsim.ini file. This causes the library mappings to be synchronous, so changes to the original modelsim.ini’s library mappings also affect the new project.
3. Click OK to create the new project.
4. Add user RTL and testbenches to the project using the Add Existing File option from the popup window. If the popup disappears, right click anywhere in the Project Tab > Add to Project > Existing file.

5. Compile the files in the project to work.
   a. Right click anywhere in the Project Tab > Compile > Compile All or Compile Out-of-date.
   b. Selecting Compile > Compile All or Compile Out-of-date from the menu bar also works.
   c. If compilation fails, try setting the Compilation Order so files with dependencies are compiled after the files they are dependent on.
   d. Files at the top of the list are compiled first.
6. Select Simulate > Start Simulation... from the menu bar.
7. In the Design tab, expand the work library using the plus icon next to its name. Select the testbench for the project. The Design Unit(s) field should update to reflect the testbench module selection.
8. In the Libraries tab, use the -L field and Add... button to specify which libraries to link. Typically, this is work, <device library>, pmi_work, and uaplatform.
9. Click OK to begin simulating.
3.1.2. Initial Setup (Directory-based Flow)

To set up based on directory:
1. Switch to the directory the work library should be created in.
   a. Select File > Change Directory ... or type cd <directory path> to switch to the correct directory.
2. Select Simulate > Start Simulation... from the menu bar.
3. In the Design tab, expand the work library using the plus icon next to its name.
   a. Select the testbench for the project. The Design Unit(s) field should update to reflect the testbench module selection.
4. In the Libraries tab, use the -L field and Add... button to specify which libraries to link.
   a. Typically this is work, <device library> and pmi_work.
5. Click OK to begin simulating.

3.1.3. Simulation Flow

To start the simulation flow:
1. Access the work library.
   a. Project-based: open the ModelSim project.
   b. Directory-based: change directory to the directory work is mapped to.
2. Select Simulate > Start Simulation... from the menu bar.
3. In the Design tab, expand the work library using the plus icon next to its name.
   a. Select the testbench for the project. The Design Unit(s) field should update to reflect the testbench module selection.
4. In the Libraries tab, use the -L field and Add button to specify which libraries to link.
   a. Typically, this is work, <device library> and pmi_work.
5. Click OK to begin simulating.

Advantages:
- Requires no knowledge of ModelSim TCL commands.
- Easier to interpret what settings to use for a simulation.
- Can use project-based or directory-based simulation flows.

Disadvantages:
- Requires several icon presses to setup, compile, and simulate a project.
- Repeatability is hindered due to the repeated user interface selections that are required.

3.2. Simulation Wizard

In this section of the document, the initial setup and setup for general simulations using the Simulation Wizard flow are covered. Although the screen captures is from Radiant’s version of Simulation Wizard, the Simulation Wizard for both Diamond and Radiant work the same and are very similar.

The main difference between the two software tools are the process stages that each Simulation Wizard can generate a .mdo simulation script for. Refer to Simulating Different Process Stages for information about simulating Diamond and Radiant projects at various process stages.
3.2.1. Initial Setup

To start initial setup:

1. Launch Simulation Wizard.
   a. Select the Simulation Wizard icon from the tool bar.
   b. Or select Tools > Simulation Wizard from the menu bar.

2. Click Next once Simulation Wizard opens.
   a. This initial page describes how Simulation Wizard works.

   ![Figure 3.4. Initial Page of Simulation Wizard](image)

3. Select a name and location for the project using the Project Name and Project Location fields.
   a. A new directory called `<Project name>` is created wherever the selected project location is.

   ![Figure 3.5. Second Page of Simulation Wizard](image)

4. Select a Process Stage to simulate the project at.
   a. These selections are only available once the required simulation files for each respective process stage have been generated.

5. Click Next.
6. Select the files required for the simulation.
   a. By default, all files in the project that are set for **Synthesis and Simulation** or **Simulation** are included.
   b. Use the **Add** or **Remove** icons to add or remove files.
   c. The Export source file list icon generates a file containing the full paths of the listed RTL files.
   d. If **Automatically set simulation compilation file order** is enabled, the order of compilation is automatically determined by Simulation Wizard.
      - Otherwise, use the **Up** and **Down** green arrow icons to manually set the order of compilation.
      - Files at the top are compiled first and should have no dependencies in the files compiled after them.

   ![Figure 3.6. Third Page of Simulation Wizard](image)

7. Click **Next** once all the required files for simulation have been added and the compilation order is correct.

8. Select the top testbench module from the **Simulation Top Module** dropdown list of options.
   a. Simulation Wizard parses through all the included files to suggest valid simulation testbench modules.
   b. Ensure the correct testbench module is selected before proceeding.

![Figure 3.7. Fourth Page of Simulation Wizard](image)
9. Click **Next**.

10. Review the settings in the top half of the final summary window to ensure the previous selections are correct.
    a. Each option listed corresponds to a setting that was selected in the previous pages Simulation Wizard four.
    b. Use the **Back** button to return to a previous page to correct any incorrect settings.

11. Select the last few settings according to how user want the script to be generated, and what to do afterwards.
    a. **Run Simulator** launches ModelSim and invoke the generated .mdo script.
    b. **Add top-level signals to waveform display** adds the *add wave /* command to the generated script, adding all the waveforms from the simulation testbench to the waveform display.
    c. **Run Simulation** adds the *run command to the final part of the simulation script, advancing it in time.
        - Use the two boxes to the right to select how long to advance a simulation in time.
        - The box on the left is the amount of time to advance, and the right box are the time units.

![FIGURE 3.8. Final Page of Simulation Wizard](image)

12. Click **Finish**.

### 3.2.2. Simulation Flow #1 (from ModelSim User Interface)

1. Invoke the generated .mdo script.
   a. Type `do <path of Simulation Wizard project>/<project name>.mdo` in the Transcript window.
   b. Or select **File > Load > Macro File** > select the <project name>.mdo file located where the Simulation Wizard project was generated.
      - By default only .do and .tcl files are displayed.
      - Use the dropdown on the bottom right to view all files.

### 3.2.3. Simulation Flow #2 (from Radiant User Interface)

To start initial setup:

1. Launch Radiant and open the project.

2. Double click `<project name>.spf` from the list of files in the project under **Script Files**.
   a. This is the Simulation Wizard project file and can be used to access existing projects.
3. Click **Skip to End** to skip to the final page of the Simulation Wizard window.
   a. If user want to make changes to any selections for the Simulation Wizard project, click **Next**.

4. On the final page, ensure **Run Simulation** is enabled and then click **Finish**.
   a. Doing this launches, ModelSim invokes the generated simulation script automatically.

**Advantages**
- Requires little knowledge of ModelSim TCL commands or TCL scripting.
- Can automatically generate a .mdo simulation script, invoke ModelSim, and run a simulation.

**Disadvantages**
- Limited customization options via the user interface, generates a basic script.
- Generated simulation script is project-based and does not support directory-based script generation.
- Invoking the script through Simulation Wizard in Diamond or Radiant regenerates the script, making it difficult to keep changes that were made to the script itself.
3.3. Simulation Wizard and Custom Script

Refer to Simulation Wizard for more detailed information about the initial Simulation Wizard project setup. The initial flow is the same and only requires Simulation Wizard to generate a .mdo as a starting point.

3.3.1. Simulation Flow (ModelSim User Interface)

1. Launch ModelSim.
2. Invoke the generated .mdo script.
   a. Type `do <path of Simulation Wizard project>/<project name>.mdo` in the Transcript window.
   b. Or select File > Load > Macro File ... > select the <project name>.mdo file located where the Simulation Wizard project was generated.
      • By default, only .do and .tcl files are displayed.
      • Use the dropdown on the bottom right to view all files.
   c. Refer to Invoking Scripts for alternate methods of invoking simulation scripts.

Advantages
• Easily generate a simulation script as a starting point.
• Unlimited customizability with the baseline script.

Disadvantages
• Requires knowledge of ModelSim’s TCL commands and TCL scripting basics.
• Generated script is project-based, so changing it to be directory-based requires additional effort.

3.4. Simulation Script

Depending on whether user using a project-based or directory-based simulation approach, the commands that user would use to simulate the project using scripts will vary. In this section, user will review the general structure for each type of script and the main considerations user should make to determine what options user should use with each command. For additional information about the most commonly used options for each of these commands, refer to Important TCL Commands and Options. For information about additional TCL commands refer to Useful TCL Commands and Options, or the ModelSim Command Reference Manual. For example scripts, checkout Directory Based Simulation Script and Post-MAP with Timing Simulation Script.

3.4.1. Project-based Simulation Scripts

The main TCL commands used for project-based simulation scripts are `project`, `vlog`, `vcom`, `vsim`, `view wave`, `add wave`, and `run`. Similar to any other simulation flow, project-based simulations can be broken down into four main parts: setting up a work library, compiling files to work, running the simulation, and interpreting results.

3.4.1.1. Project-based Script Commands

The following is the list of project-based script commands:
• Setting up a work library:
  • `project new`
  • `project open`
  • `project addfile`
• Compiling files:
  • `project compileall`
  • `project compileoutofdate`
  • `vlog`
  • `vcom`
3.4.2. Directory-based Simulation Scripts

The main TCL commands used for directory-based simulation scripts are `vlib`, `vmap`, `vlog`, `vcom`, `vsim`, `view wave`, `add wave`, and `run`. Similar to any other simulation flow, directory-based simulations can be broken down into four main parts: setting up a work library, compiling files to work, running the simulation, and interpreting results.

### 3.4.2.1. Directory-based Script Commands

The following is the list of directory-based script commands:

- Setting up a work library:
  - `vlib`
  - `vmap`
- Compiling files:
  - `vlog`
  - `vcom`
- Running the simulation:
  - `vsim`
  - `run`
- Viewing results:
  - `view wave`
  - `add wave`

The main difference between directory-based, and project-based simulation scripts are the commands used to create a work library and compile files to it. Since the `project` command can only be used for project-based simulation scripts, the `vlib` command must instead be used to create a work library.

Aside from that, another frequently used command is `vmap`, which is used to map the logical library created by `vlib` to a physical directory. If a library is not mapped using `vmap`, its library mapping in the active modelsim.ini file uses a relative path instead of an absolute one. The disadvantage of using a relative path is that ModelSim is not able to locate that library if the user is in a different directory.
Once the work library is correctly setup, the next few commands the user should use for directory-based simulation scripts are `vlog` and `vcom`, which are used to compile Verilog or VHDL files to the work library. For directory-based simulation scripts, its up to the user to manage which files are compiled and when. Because of this, its always important to always keep track of the files that are files that are required for the simulations.

Finally, once all the design and testbench modules are compiled to work, the last step of the simulation flow is to run the simulation and view the results. The commands used in this step are exactly the same as for project-based simulation scripts. These commands are `vsim` which is used to invoke the ModelSim simulator, `view wave` which opens ModelSim’s waveform display window, and `add wave` which adds the signals that user wants to track. Lastly, the `run` command is used to advance the simulation forward in time since invoking the ModelSim simulator with `vsim` keeps the time at t=0.

Refer to Directory Based Simulation Script for an example simulation script that uses a directory-based approach.

### 3.4.2.2. Simulation Flow

1. Launch ModelSim.
2. Invoke the simulation script.
   a. Type `do <script location>` in the Transcript window or select File > Load > Macro File > select the `<script location>`.
   b. Refer to Invoking Scripts for alternate methods of invoking simulation scripts.

**Advantages**

- Unlimited customizability.
- Can use project-based or directory-based simulation flows.

**Disadvantages**

- Requires knowledge of ModelSim’s TCL commands and TCL scripting basics.

### 3.5. Propel Verification Projects

#### 3.5.1. Initial Setup

To start initial setup:

1. Edit the CPU component and enable Simulation Mode and disable Debug Enable.
   a. Doing this generates the RISC-V processor in a way that is compatible for simulations.
   b. Disabling debug mode removes a JTAG hub component that does not have a simulation model.
2. Edit the system memory component and initialize it with the .mem file from the Propel SDK C project.

3. Regenerate the SoC project using the Generate icon from the toolbar.

4. Switch to the verification project using the Switch Verification and Design icon from the toolbar.

5. Add, configure, and connect verification IP.
   a. The exact setup for this step varies from project to project.

6. Generate simulation environment using the Generate icon from the toolbar.
   a. Doing this generates several additional files and folders for the simulation environment in the <Builder project location>/verification/sim directory:
      b. flist.f – file list that contains the paths for all RTL files in the simulation environment.
      c. <project name>_v.sv – basic simulation testbench to add stimulus to the SoC project. Generated according to the IP and connections in the Verification project schematic setup.
      d. msim.do – simulation do script that compiles all RTL for the SoC and verification project, invokes the ModelSim simulator, and advances the simulation.
      e. wave.do – waveform do script that adds all signals from components in the SoC project.
      f. work – location of the work library generated by the simulation script.

7. Edit the generated testbench and simulation script.
   a. msim.do, wave.do, and <project name>_v.sv are good candidates to edit and add additional functionality to the simulation environment.
8. Launch ModelSim using the **Launch Simulation** icon 🚀 from the toolbar.
   a. This launches Propel’s version of ModelSim and invokes the generated simulation .do script.
9. Avoid regenerating the verification project to prevent files from being overwritten from step 4.
   a. Only regenerate to add verification IP or make changes to the verification project setup.
   b. Modifying the DUT doesn’t require simulation environment regeneration. Since the DUT is instantiated in the test bench, any changes will synchronize due to the simulation script recompiling each file.

### 3.5.2. Simulation Flow (ModelSim User Interface)

1. Launch ModelSim from Propel Builder.
   a. This step is important, so the correct libraries are mapped.
2. Invoke the generated .mdo script.
   a. Type `do <script location>` in the Transcript window.
   b. Or select **File > Load > Macro File ...** > select the `<script location>`.
   c. Refer to **Invoking Scripts** for alternate methods of invoking simulation scripts.
3.5.3. Simulation Flow (Propel Builder User Interface)

1. Launch Propel Builder and open the Verification project.
2. Select the **Launch Simulation** icon from the toolbar.
   a. Doing this launches Propel’s version of ModelSim and invokes the generated simulation .do script.
   b. Avoid regenerating the simulation environment beforehand if changes were made to the simulation script or testbench to prevent those files from being overwritten.

**Advantages**
- Requires little-to-no knowledge of TCL commands or scripting.
- Generates a basic simulation environment for users to perform a behavioral simulation for their SoC.

**Disadvantages**
- Only supports RTL simulation.
- No support for VHDL or mixed-language projects without editing the generated simulation script.
4. ModelSim Usage Tips

This section of the document covers some various tips and suggestions to simplify and enhance the usage of ModelSim for simulations.

4.1. Tips for Script Based Simulations

4.1.1. Do versus TCL Scripts

The two main types of scripts that are used in ModelSim are do (.do) and TCL (.tcl) scripts. Functionally, both script types are similar to each other and execute commands in a sequential order. One of the main differences between do and TCL scripts, however, is that do scripts can use a few additional commands such as ONBREAK, ONELABERROR, ONERROR, and ONFINISH, which are useful for runtime condition handling and do not execute sequentially.

For example, placing ONERROR {resume;} at the top of a simulation script causes it to resume execution when it encounters an error, regardless of when or where the error is encountered. For more information and examples for these commands, refer to onbreak, onelaberror, onerror, and onfinish.

4.1.2. Invoking Scripts

The two main ways to invoke a script in ModelSim are using either the do or source commands. Typically, the do command is used to invoke do scripts, while the source command is used to invoke TCL scripts. However, either can be used interchangeably in ModelSim. The reason for this is because do is a superset of source, meaning there is no functional difference between using do or source to invoke a TCL script. One thing to consider, however, is whether the script has any do specific commands like onbreak or onerror. For scripts containing these commands, the do command should be used.

As mentioned before, either the do or source commands can be used to invoke a script in ModelSim. To invoke a script, simply type either do or source in ModelSim’s transcript area, followed by the location of the script the user wants to invoke. If the script is in the current directory, the user can simply type the name of the script to invoke it. Similarly, a script can invoke another scripts using either of these commands.

Figure 4.1. Method for Invoking Scripts using the ModelSim User Interface

Aside from that, another way to invoke a script is to select File > Load > Macro File from the ModelSim’s menu bar as shown in Figure 4.1. If the user is unable to select the macro file, try clicking the transcript area first. Once Macro File has been selected, a new file explorer window opens and the user can select the script to invoke. By default, this window only display .do and .tcl files, so the user should use the dropdown at the bottom right of the window to select a different file type if the script uses some other extension. Once the user located the script to invoke, select and then click the Open button to invoke it in ModelSim. Doing this is functionally, the same as typing do <script name and location in ModelSim’s transcript area.
4.1.3. Compiling Multiple Files

Depending on the language of the files the user wants to compile, there are in general three main ways to compile multiple files to a ModelSim library. The VCOM command is used to compile VHDL modules and has a -f method which can be used to compile multiple files from a master file list. The VLOG command, which is used to compile Verilog and System Verilog files, also has a -f method as well as a +incdir+ method, and a -y method for compiling multiple files. The main limitation for these methods of compilation is that neither VCOM or VLOG support mixed language compilation. This means that the user cannot compile VHDL and Verilog files with the same VCOM or VLOG commands and must compile them separately.

The important thing to know about compiling multiple files with single TCL commands is that each module is treated as its own independent design unit. This means that using any method for compiling multiple files is functionally the same as typing multiple VLOG or VCOM to compile each individual file. Aside from that, the VLOG command also has a -mfcs option which can be used to change this compilation behavior so that all compiled files are treated as a single design unit.

4.1.3.1. -f Compilation Method

Requirements
• Separate file list file with full or relative paths to additional files to compile.

Usage: vlog -work <library> -f <file list>

Advantages
• Command is simple to use and understand.
• Can be used by both VLOG or VCOM.

Disadvantages
• Requires an additional file list file.
• Compiles all files in the file list, regardless of whether they are required or not.

4.1.3.2. +incdir+ Compilation Method

Requirements
• Include for additional files to compile in the top file being compiled first.

Usage: vlog +incdir=<search path> -work <library> <top file>

Advantages
• Easiest command to setup if all `include RTL are in the same few directories.

Disadvantages:
• Cannot compile VHDL files that are include.
• Requires additional +incdir+ to search for files located in a different directory.

4.1.3.3. -y Compilation Method

Requirements
• Top module with missing design units.
• -y option for various locations to search for design units to compile.
• +libext+ option to specify the file extensions to search for with the -y option.

Usage: vlog -work <library> <top file> -y <file search path> +libext+<file extension>

Advantage
• Can easily resolve missing instances to compile necessary files.
• Easy to manage which files to compile after the initial setup.
• Only compiles files that are necessary.

Disadvantage
• Complicated initial setup requires specific directory and file structure.
• Cannot compile VHDL files.
4.2. Simulation Tips and Tricks

The purpose of this section is to introduce various quality-of-life tips to improve the usage of ModelSim. These tips are suggestions and can be used to simplify various portions of the simulation process. All these tips can be used interchangeably and are not mutually exclusive with each other. The only exception is for Simulation Configurations which can only be used for project-based simulations and does not have an equivalent mechanic for directory-based simulations.

4.2.1. Simulation Configurations

A simulation configuration is an easy way for project-based simulations to reinvoke the ModelSim VSIM simulator with some specific settings. Functionally, using a simulation configuration is the same as typing a VSIM command to ModelSim’s transcript. The advantage of simulation configurations however is that once they are setup, users only need to double click the configuration in their project tab to reinvoke the ModelSim simulator with those specific settings.

This method is especially useful project for projects with multiple testbenches or sets of options used to simulate a design in different ways. In situations like this, multiple simulation configurations could be used to easily reinvoke the ModelSim simulator with a different set of VSIM command options to change the way the project is simulated.

4.2.1.1. Creating a Simulation Configuration

To create a simulation configuration:

1. Select Project > Add to Project > Simulation Configuration.
   a. Right clicking the Project tab, then selecting Add to Project > Simulation Configuration also works.

![Figure 4.2. Adding a Simulation Configuration to a ModelSim Project](image)
2. Define a Simulation Configuration Name.

3. Select the folder to place the simulation configuration in using the Place in Folder dropdown option.
   a. By default, the configuration is added to the top-level project folder.

4. Select a top module to simulate from the Design tab.
   a. Select the dropdown next to Work, then select the module name corresponding to the testbench.
   b. The Design Unit(s) field should update to reflect the testbench selection.

5. Select which libraries to link for missing design instances in the Libraries tab.

6. Select settings in the VHDL, Verilog, SDF, or Others tabs according to the projects simulation requirements.

7. Click Save to add the configuration to the project.
   a. The simulation configuration is visible in the Projects tab as shown in Figure 4.4.
4.2.1.2. Launching a Simulation Configuration

Once a simulation configuration has been added to a ModelSim project, it can be easily executed to reinvoke the ModelSim VSIM simulator with the specified simulation settings. To invoke the ModelSim simulator using a simulation configuration, double click the name of the configuration from the Projects tab as shown in Figure 4.4.

4.2.2. Saving and Loading user interface Layouts

A useful aspect of ModelSim is its user interface Layout feature, which can be used to alternate between various user interface configurations. As its name implies, the usefulness of user interface layouts is that it can be used to create multiple different user interface layouts depending on which windows user want to see, how user want them positioned, how they should be sized, and more.

ModelSim comes with a few different user interface layouts by default, such as NoDesign and Simulate. These default layouts are used by ModelSim during various parts of the ModelSim usage to automatically update the user interface layout and appearance. For example, when the VSIM simulator is invoked, the Simulate layout is loaded, which opens the waveform window as well as a few other debug views.

4.2.2.1. Creating a Layout

Creating a user interface Layout is simple. The first step in doing so is to modify the existing user interface layout and configuration according to the preference. This step does not require any active simulation to be running to include debugging views like the objects or waveform windows. Any layouts that are created are not project specific, and can be used regardless of the version of ModelSim that users are using or the project that users are simulating.
4.2.2.2. Loading a Layout

Depending on the intentions for the custom layout that was created, there are a few different ways to load a layout in ModelSim. The first and most simple way to load a custom layout involves the ModelSim user interface. To select a different layout to load, select the arrow next to the **Layout** option in order to expand the list of layouts to select from. From the dropdown list of layouts that appears, select the layout the user wants to load in order to load that layout.

Similarly, another way to load a user interface layout in ModelSim involves the **Layout** tab dropdown from ModelSim’s menu bar as shown in **Figure 4.5**. To load a different user interface layout this way, select the layout the user wants to load from the list of layouts that appear at the bottom of the dropdown.
Aside from that, another way to select which layouts to load is to set the default layouts that ModelSim automatically loads. To do this, select **Layout > Configure** from ModelSim’s menu bar. As shown in Figure 4.8, doing this opens the Configure Window Layouts window. Within this window, select a new layout to load automatically using the **When no design loaded** and **When a design is loaded** fields. The **when no design is loaded** selection is active whenever ModelSim is opened, and the **when design is loaded** option only activates when the VSIM simulator is invoked.

![Figure 4.8. ModelSim Layout Configuration Window](image)

The advantage of this method for selecting a ModelSim layout is that ModelSim automatically switches between these layouts as user simulate the project. The drawback of this method, however, is that there are only two stages that can be set which are when a design is loaded and when a design is not yet loaded. Aside from the methods mentioned, one last way to load layouts involves the layout load TCL command. As the name implies, this TCL command can be used in scripts in order to automatically load a specific user interface layout during a scripts execution. For more information about this command, refer to layout load.

### 4.2.3. Creating a Waveform Do Script

The Waveform scripts are a useful way to load a specific waveform configuration with signals and settings without having to manually add or modify any signals. An example waveform do script is shown in Post-MAP with Timing Simulation Script. Typically, there are two main commands used in a waveform do script. The first command is **add wave**, which is used to add signals to the waveform display. This command has several additional options which correspond to various settings for that signal. For more information about the **add wave** TCL command, refer to add wave.

Aside from that, another command that is typically used with waveform do scripts is **onerror**. Adding **onerror {resume}** to the top of user waveform do script allows the script to continue executing if it encounters an error. This is particularly useful for designs that are a work in progress and are frequently changing. The reason for this is because the **onerror** command allows the script to continue running if there are some signals that have been renamed or removed from the design. For more information about the **onerror** TCL command, refer to onerror.

### 4.2.3.1. Saving a Waveform Format as a Script

A useful feature of ModelSim is that its waveform display can be used to automatically generate a waveform do script. Depending on the modifications that were made to the waveform display, a do script containing the equivalent **add wave** TCL commands are generated. This do script can be invoked to repopulate a waveform display with the exact same settings as before.
4.2.3.2. Saving a Waveform Configuration
To save a waveform configuration:
1. Configure the waveform as shown in Figure 4.9.
2. Select File > Save Format from the menu bar.
   a. CTRL + S is another way to save the waveform configuration.

![Figure 4.9. Save Waveform Display Format Window](image)

3. Define a name and location to save the waveform do script.
   a. By default, the script is called wave.do and saved to the current ModelSim directory.
4. Enable Waveform formats.
   a. This setting saves all the changes that were made to the waveform display to a script.
5. Click OK.

Once the waveform format is saved as a script, it can easily be invoked like any other script in ModelSim. The only requirement is that the waveform window is already open. A useful way to invoke this script from a separate script, is using the do TCL command, or the -do VSIM option.

4.2.4. Creating and Using Custom Libraries
Custom libraries are a useful method for storing design or verification modules that are commonly used in other testbenches or designs. Custom libraries are particularly useful for modules that do not change frequently, as users can compile that file to a custom library knowing that they do not need to recompile it like other design or testbench files associated with the project.

There are two main ways to create a custom library, and in this section, the user focuses on how to do so using the ModelSim user interface. For more information on creating a custom library using TCL scripts, refer to the example script in Compiling Files to a Custom Library.

4.2.4.1. Creating a Custom Library
To create a custom library:
1. Switch to the directory user want to create the library in.
   a. Use the CD TCL command, or select File > Change Directory from the menu bar.
2. Select File > New > Library from the menu bar.
3. Select a **new library and a logical mapping to it**.
   a. This creates a new ModelSim library and maps it in the active modelsim.ini file.

4. Define a **Library Name**.
   a. The **Library Physical Name** option automatically updates to match.
   b. The **Library Physical Name** is the name of the folder that is created.

5. Click **OK**.
   a. A new library is created and mapped relative to the current directory.
   b. Edit the modelsim.ini file to add the full library mapping.
   c. By default, the mapping added to the active modelsim.ini file is relative to the current directory causing it to be inaccessible for projects/simulations in other directories.
   d. The modelsim.ini file the full mapping is added to controls which projects can use the library.
   e. `<project name>.mpf` – library is only accessible from the current project.
   f. **Local modelsim.ini** – only simulations from the current directory can access the library.
   g. **Master modelsim.ini** – library is accessible for all local modelsim.ini and `<project name>.mpf` files that reference the master modelsim.ini file (recommended).
4.3. Simulating Different Process Stages

Both Diamond and Radiant support post-RTL simulation and can be used to generate post-synthesis simulation netlists and standard delay format (SDF) files to simulate projects with timing. The usefulness of post-RTL simulation is that it allows the user to check the functionality of our projects at various points in the Diamond or Radiant project implementation flow. Aside from actually testing the full design on device, these post-RTL simulation methods enable us to better understand how the project likely functions.

4.3.1. Generating Post-RTL Simulation Files

The main requirement for any of the post-RTL simulation options available in ModelSim is to generate the appropriate files required for simulation using either tool's process toolbar. Although both Diamond and Radiant support various kinds of post-RTL simulation, there are some differences between the processes stages that each software can generate simulation files for.

Figure 4.13 shows the Radiant process toolbar which only supports Verilog post-synthesis and gate-level simulation file generation. Enabling either of these options generates a .vo simulation netlist file that can be used to simulate a project at that process stage. Additionally, for gate-level simulations there is also another SDF file that is generated, which can be used to simulate the netlist with timing.
Figure 4.14 shows Diamond process navigator tab, which supports both VHDL (.vho) and Verilog (.vo) simulation netlist generation. Additionally, another difference is that Diamond can generate a post-MAP, and post-PAR simulation netlist file, as well as SDF files for both process stages to simulate the project with timing. Once the correct option has been enabled, a simulation netlist file can be generated for that respective process stage by running Diamond through it or by double clicking the option itself.

![Diamond Process Navigator Tab](image)

**Figure 4.14. Diamond Process Navigator Tab**

### 4.3.2. Using Post-RTL Simulation Netlists

Once the simulation netlist has been generated for the process stage user want to simulate the project at, there are two main ways to simulate that project's simulation netlist. The first way is to use TCL commands or scripts, and the second way is using the ModelSim user interface. To simulate a project post-synthesis using the ModelSim user interface, simply compile the .vo or .vho post-RTL simulation netlist file to the work library, and then simulate the project as normal. For more information about the general simulation flow using the ModelSim user interface, refer to ModelSim User Interface.

A useful feature of Diamond and Radiant’s Simulation Wizard tools are that they can be used to generate a post-RTL simulation script for the process stage user want to simulate the project at. Once the correct simulation files have been generated for a process stage, that option becomes available for selection in the second page of the Simulation Wizard setup window.

Aside from that method of generating a post-RTL simulation script, the general process for creating a custom post-RTL simulation script is fairly simple. The only difference between this flow and any other RTL simulation flow is that user would first need to compile the .vo or .vho post-RTL simulation netlist file using either the VLOG or VCOM TCL commands. Once the correct simulation netlist has been compiled to the working library, the final step is to invoke the ModelSim simulator using the VSIM command. For post-RTL simulation, there are no additional TCL commands or options that are required. Refer to Post-MAP with Timing Simulation Script for an example of a post-RTL simulation script.
4.3.3. Simulating Post-RTL with Timing

Similar to the previous section, the two main ways to simulate a project with timing are to use either TCL commands or scripts or the ModelSim user interface.

4.3.3.1. Simulating Post-RTL with Timing using TCL Commands

Similar to simulating different process stages, both Diamond and Radiant’s Simulation Wizard tools can be used to generate post-RTL simulation scripts with timing. The main limitation of this, however, is that Diamond’s Simulation Wizard cannot generate a post-MAP with timing script, although the software tool itself can generate a post-MAP SDF file. To generate a post-RTL with timing simulation script using Simulation Wizard, select the correct process stage with timing option in the second page of the Simulation Wizard setup window.

Aside from using Simulation Wizard to generate a post-RTL with timing script, another way to simulate a project with timing involves the VLOG, VCOM, and VSIM TCL commands. Similar to post-RTL simulations, user must also compile the .vo or .vho simulation netlist file using the VLOG or VCOM commands depending on the file extension. The reason for this is because each SDF file is annotated using the nets corresponding to the simulation netlist file that was generated for that process stage. Because of this, any SDF file can only be used with the simulation netlist file that was generated with it for that process stage.

Once the post-RTL simulation netlists have been compiled to work, the only other difference in the script-based simulation flow is to use either the -sdfmin, -sdftyp, or -sdfmax options for the VSIM command in order to annotate the cells in the simulation netlist with either their minimum, typical, or maximum standard delay format values. For more information about these command options, refer to vsim for information about the command options or to Post-MAP with Timing Simulation Script for an example script.

4.3.3.2. Simulating Post-RTL with Timing Using the ModelSim User Interface

To simulate post-RTL with timing:

1. Compile .vo or .vho simulation netlist file to work.
   a. Refer to ModelSim User Interface for information compiling files to work using the ModelSim user interface.
2. Select Simulate > Start Simulation from the ModelSim menu bar.
3. Select the correct test bench to simulate from the Design tab.
4. Select the correct libraries to link to resolve missing design instances using the Libraries tab.
5. Click Add in the SDF Files section of the SDF tab.

![Figure 4.15. SDF Tab View of the Start Simulation Window](image-url)
6. Click **Browse** to locate and select the SDF file for the project.
7. Choose the delay type to add using the **Delay** field.
   a. **typ** = typical delay value, **min** = minimum delay value, and **max** = maximum delay value.
8. Select the region to apply the delay format to using the **Apply to Region** field.
   a. For a testbench **top_tb** instantiating a DUT called **hw_soc**, user would need to enter `/top_tb/hw_soc` to apply the delay to the correct module.
9. Click **OK**.
10. Repeat steps 5-9 to apply more standard delay values to cells as required.
11. Click **OK** to begin the simulation.

![Add SDF Entry Window]

**Figure 4.16. Add SDF Entry Window**
5. Important TCL Commands and Options

ModelSim has many different TCL commands that are used to automate certain parts of the project simulation flow. The most frequently used TCL commands and options for TCL scripting are included in this section of the document.

This document does not outline all the TCL commands or TCL command options in ModelSim. For more detailed information about these TCL commands or other TCL commands or options that were not mentioned in this section, refer to the ModelSim Command Reference Manual.

Access the ModelSim command reference manual by opening the SW tool's version of ModelSim, then selecting Help in the menu bar, then PDF Documentation, and then Reference Manual.

Each of the commands mentioned in this section has a description, syntax, arguments, and example subsection. Arguments in between arrows < > indicate the type of argument, which is typically some user input like a file name. Arguments in brackets [ ] are optional, while all other arguments are required.

5.1. add wave

Description: Adds an object to the waveform display window.

Syntax: add wave <arguments>

Arguments

- <object name>
  - Adds the specified object to the waveform display window.
  - Use * to add all the signals from the current module.
  - Use / to add signals from different levels of the project’s hierarchy.

- {<object name> {signal_1 signal_2 ...}}
  - Creates a user-defined bus called <object name> containing signal_1, signal_2,...

Example

- add wave top_tb/dut_wrap/*
  - Adds all the top-level signals from the dut_wrap module that is instantiated by top_tb.
- add wave {my_bus {led_0 led_1 led_2 led_3}}
  - Adds a new custom bus containing the led_0 – led_3 signals from the top module.

5.2. cd

Description: Changes the current directory to the one specified following the command.

This should not be used for project-based simulations as projects are tied to a specific directory. Changing the directory after a project is open closes that project.

Syntax: cd <directory path>

Arguments

- <directory path>
  - Specifies the full or relative path that user want the current ModelSim directory to be changed to.

Example

- cd C:/Users/john/Documents/my_projects/counter
  - Changes to the absolute directory called counter.
- cd ../scripts/plugins
  - Goes two directories back from the current directory, then into the plugins directory.
5.3. do
Description: Invokes a script or executes a set of commands.
Syntax: do <file name> [<parameters>]
Arguments
• <file name>
  • Name of the script that user want to invoke.
  • Can be a full or relative path depending on the location of the script.
• [<parameters>]
  • (Optional) set of parameters to pass to the script being invoked.
Example
• do C:/Users/john/Documents/scripts/plugins/my_script.do
  • Invokes the my_script.do script located at the specified full path.

5.4. project
Description: Performs a variety of different operations on a project. These commands can only be used for and on ModelSim projects and do not work with directory-based simulations.
Syntax: project <arguments>
Arguments
• addfile <file name> [folder name>
  • Adds the specified file to the current project.
  • If a <folder name> is specified, the file is added to that project folder.
• addfolder <folder name> [parent folder>
  • Creates a new folder within the active project.
  • If a <parent folder> is specified, the new folder is added as a sub-folder of the parent.
• calculateorder
  • Determines the optimal compilation order for the files in the project by compiling each file and moving the files that error out to the bottom of the order.
• close
  • Closes the active project.
• compileall [-n]
  • Compiles all the files in the active project.
  • -n outputs the list of commands that would be executed to compile each file in the project, without actually invoking those commands or compiling any files.
• compileoutofdate [-n]
  • Compiles all files that have been updated since their last compilation (the date/time stamp from each file are used to determine which have been modified).
  • -n outputs the list of commands that would be executed to compile each file in the project, without actually invoking those commands or compiling any files.
• new <directory> <project name> [default lib>] [initial .ini>] [0 | 1]
  • Creates a new project in the specified <directory> called <project name>.
  • <default lib> is the name of the library the project defaults to (default is work).
  • <initial .ini> is used to specify a specific modelsim.ini to use as the source for the new project.
  • If no <initial .ini> is specified, the active modelsim.ini file is used instead.
  • [0 | 1] is used to determine how to reference the source modelsim.ini file.
  • 0 is the default option, and copies all the library mappings from the source modelsim.ini to the new .mpf file, causing them to be local for the new project.
  • 1 references the library mappings from the source modelsim.ini.
• open <project .mpf>
  • Opens the specified project file.
  • If a project is currently open, ModelSim closes it before opening the new project.
• removefile <file name>
  • Removes the specified file from the active project.

Example
• project new . clnx_sim_proj C:/Users/john/Documents/modelsim.ini 1
  • Creates a new project called clnx_sim_proj in the current directory.
  • References the library mappings from C:/Users/john/Documents/modelsim.ini.

5.5. run

description: Advances a simulation for a specified amount of time, or specific number of time steps. This can only be used after the VSIM command has been invoked to begin a simulation.
syntax: run <arguments>

Arguments
• <no argument>
  • Advances the simulation for the default amount of time (100 ns).
• <time steps> [<time units>]
  • Advances the simulation by the specified number of time steps and time units. Time steps can be floating point, as long as the precision does not exceed the specified timescale precision in the RTL.
  • Valid time units: fs, ps, ns, µs, ms, sec
  • If no time unit is specified, the default unit (ns) is used.
• -all
  • Advances the simulation forever, or until a breakpoint such as $finish is encountered in the testbench.
• -continue
  • Continues running a simulation that has encountered an event or breakpoint.
• -next
  • Advances the simulation until the next event or breakpoint.

Example
• run 10.43 µs
  • Advances the simulation for exactly 10.43 microseconds.
• run -all
  • Advances the simulation indefinitely until a breakpoint is encountered.
5.6. **vcom**

**Description:** Compiles a VHDL design unit into a specified library.

**Syntax:** `vcom [options] <file name>`

**Options**

- `-87 | -93 | -2002 | -2008`
  - Specifies which VHDL standard to compile the specified file with.
  - By default, if no standard is specified 2002 is selected.
- `-error <message number>, [<message number>, ...]`
  - Changes the specified message number(s) level of severity to error.
- `(-F | -file | -f) <filename>`
  - Specifies a .f file list with the path of other VHDL files to compile using the same instances.
  - Simplifies the process for compiling multiple VHDL files with the same settings.
- `-work <library name>`
  - Specifies the ModelSim library to compile the VHDL design units to.
  - By default, if no library is specified the files are compiled to work.

**Example**

- `vcom -f flist.f -2008 -work alt_work top.vhd`
  - Compiles top.vhd, and all the files specified in flist.f using VHDL-2008 standard.
  - Each VHDL design unit is compiled to the library called `alt_work`.
- `vcom top.vhd -warning 2213`
  - Compiles the top.vhd design unit to work.
  - Changes the severity of the error message number 2213 to warning.

5.7. **view**

**Description:** Opens the specified window. If no argument is input after the command, a list of all the currently active windows are returned.

**Syntax:** `view <arguments>`

**Arguments**

- `<window type>`
  - Type of window to open
  - Valid window types:
    - assertions, atv, browser, calltree, canalysis, capacity, classgraph, classtree, covergroups, dataflow, details, duranked, exclusions, fcovers, files, fsmlist, fsmview, instance, library, list, locals, memdata, memory, msgviewer, objects, process, profile details, project, ranked, runmgr, schematic, source, stackview, structural, structure, tracker, transaction, transcript, uvmdetails, watch, wave
  - `-height <# of pixels>`
    - Specifies the height of the window in pixels.
  - `-title <window title>`
    - Specifies a new name for the window.
-width <# of pixels>
  - Specifies the width of the window in pixels.
-dock | -undock
  - Docks or undocks the specified window to the main ModelSim user interface instance.
-x <# of pixels>
  - Specifies the location of the upper left-hand x-coordinate of the window in number of pixels.
  - Must be a non-negative integer.
-y <# of pixels>
  - Specifies the location of the upper left-hand y-coordinate of the window in number of pixels.
  - Must be a non-negative integer.

Example
- view wave -undock -width 1920 -height 1080
  - Opens the waveform window, undocking it from the main modelsim user interface.
  - Waveform window is 1920 x 1080 pixels, occupying the full monitor display.

5.8. vlib
Description: Creates a design library.
Syntax: vlib [options] <library name>
Arguments
- -dirpath <directory path>
  - Used to specify a different directory to create a library in.
  - If this is not specified, the new library is created in the current directory.
- (-lock | -unlock) <design unit>
  - Locks or unlocks a specific design unit within a library from being recompiled.
- -locklib | -unlocklib
  - Locks or unlocks an entire design library from being recompiled.
- -override_precision
- -override_timescale
- <library name>
  - Name of the library user want to create.
Example
- vlib -dirpath C:/alt_path/new_lib lib_ex
  - Creates a new design library called lib_ex located at C:/alt_path/new_lib/lib_ex
- vlib -unlocklib lib_ex
  - Unlocks the library called lib_ex if it was locked before.
  - If it was not locked previously, nothing will happen.

5.9. vlog
Description: Compiles a Verilog or System Verilog design unit into a specified library.
Syntax: vlog [options] <file name>
Arguments
- -93
  - Specifies that the VHDL interface to Verilog modules uses VHDL-1993.
- +define+ <macro name>[=<macro text>]
  - Defines a macro via the command line that is called <macro name> and has value <macro text>.
  - +define+<macro name 1>=<macro text 1>+<macro name 2>=<macro text 2>+.. can be used to define multiple macros in the same command.
-error <message number>[, <message number>, ...]
  - Changes the severity level of the specified message to error.

- (F | -file | -f) <file name>
  - Specifies another file that specifies the paths for other design units to compile.
  - Used to compile multiple design units easily with the same compilation settings.

+incdir+<directory path>
  - Specifies the location of directories to search for files located within includes.
  - By default, VSIM searches the current directory, followed by any directories specified with +incdir+.

+libext+<suffix>
  - Used in conjunction with the -y VLOG option to specify the extension of files to search for compilation.

-mfcu
  - Treats each file in a command as a single compilation unit.
  - Opposite behavior of -sfcu.

-permissive
  - Allows messages from the LRM group of errors to be treated as warnings.

-override_precision
  - Overrides the precision specified in RTL with whatever is set with the -timescale command.

-override_timescale=<time units>/<time precision>
  - Overrides the timescale and precision specified in RTL with <time units> and <time precision>.

-sfcu
  - Treats each file in a command as its own separate compilation unit (default vlog behavior).
  - Opposite behavior of -mfcu.

-source
  - Displays the associated line of code that an error is generated from.
  - By default, only the error message and line number were displayed.

-suppress {<message number> | <message group>}[, {<message number> |<message group>}, ...]
  - Prevents the specified message, or groups of messages from displaying.

-sv
  - Enables System Verilog features and keywords.
  - By default compilation follows IEEE standard 1364-2005 and ignores System Verilog keywords.
  -timescale=<time units>/<time precision>
    - Specifies a default timescale for all design units being compiled that do not already have an explicit timescale definition.

-warning <message number>[,<message number>]
  - Changes the severity of the specified message(s) to warning.

-warning error
  - Changes the severity of all warning messages to error.

-work <library name>
  - Specifies a different library to compile design units to.
  - If no -work library is specified, files are default compiled to work.

-y <library directory>
  - Specifies a source library containing module definitions, packages, interfaces, and primitives to search for that are not already part of an existing library.
  - Requires the +libext+ option to specify the file type extensions to search for.

<file name>
  - Name or path of the specific Verilog file to compile.

Example
vlog -work work -sv -permissive -timescale 1ns/100ps top_tb.v
  - Compiles the modules in top_tb.v to work with a timescale and resolution of 1 ns by 100 ps.
  - Uses System Verilog features and keywords during compilation and demotes some errors to warnings.
vlog top_tb.v -sv -mfuc -y C:/project/rtl +libext+.v+.vh+.sv+.svh
  - Compiles the top module top_tb.v to work.
  - Searches C:/project/rtl for any unresolved instances with extensions .v, .vh, .sv, and .svh and then compiles those files to work as a single compilation unit.

5.10. **vmap**

**Description:** Defines a mapping between a logical library and a physical library by modifying the active modelsim.ini file.

**Syntax:** vmap <arguments>

**Arguments**

- `-c`
  - Copies the active modelsim.ini to the current directory.
  - Do not use this command with any other options.
- `-del <logical name>`
  - Deletes a mapping from the specified logical library from the active modelsim.ini file.
- `<logical name> <path>`
  - Maps a logical library to the library at the specified path.
- `-modelsimini <path>`
  - Loads an alternate modelsim.ini file that replaces the currently active .ini when executing this command.

**Example**

- `vmap -c`
  - Creates a copy of the active modelsim.ini file in the current directory.
- `vmap new_work C:/Users/john/my_dir/work33`
  - Maps the logical library *new_work* to the physical directory C:/Users/john/my_dir/work33.

5.11. **vsim**

**Description:** Invokes the VSIM simulator and begins a simulation.

**Syntax:** vsim <arguments>

**Arguments**

- `-default_radix <radix>`
  - Overrides the default radix ModelSim setting.
  - Valid radices: ascii, binary, decimal, hexadecimal, octal, symbolic, unsigned.
- `-do {<command string> | <do file>}`
  - Executes the set of commands in <command string> or within <do file> once VSIM has loaded.
- `-error <message number> [, <message number>, ...]`
  - Sets the selected message numbers severity to error.
- `-f <file name>`
  - Specifies a separate file with more VSIM command options.
  - Enables complex command reuse without having to retype arguments each time.
- `-g <Name>=<Value>`
  - Assigns a value to all VHDL generics or Verilog parameters that are unassigned or do not have explicit values yet.
  - Multiple arguments can be entered for each respective generic or parameter as a space-separated list.
- `-lib <library name>`
  - Specifies the working library to look for design units.
  - If no -lib is specified, the default library is *work*.
- `-L <library name>`
  - Specifies an additional library to look for design units after searching *work* or the library specified with -lib.
  - Libraries are searched in the order their -L is linked in the command line.
• -modelsimini <path>/modelsim.ini
  • Specifies an alternate ModelSim setting file to use in place of the active one.
• -permissive
  • Downgrades some error messages in the LRM group of errors to warnings.
• -sdffmin | -sdfftyp | -sdfmax [@<delay scale>] [<module name>=<sdf file location>]
  • Annotates <module name> with either the minimum, typical, or maximum standard delay values specified in the standard delay format file located at <sdf file location>.
  • Scales the delay value that is applied by <delay scale> if one is specified. Otherwise the default is x1.
• -supress <message number> [, <message number>, ...]
  • Prevents the specified <message number> from appearing during simulation.
• -t [<multiplier>] <time unit>
  • Specifies the time resolution for the simulation.
  • Valid time units: fs, ps, ns, µs, ms, sec.
• +transport_int_delays
  • Selects transport mode with pulse control for single-source nets.
  • If this is not enabled, by default pulses smaller than the delay from the sdf are filtered out.
• +transport_path_delays
  • Selects transport mode for path delays.
  • If this is not enabled, by default pulses smaller than the delay from the sdf are filtered out.
• -warning <message number> [, <message number>, ...]
  • Changes the severity level of <message number> to warning.
• -warning_error
  • Reports all warnings as errors.

Example
• vsim -lib work -L ovi_lifcl work.top_tb
  • Invokes the ModelSim simulator, links the work and ovi_lifcl libraries to resolve instances.
  • Uses the top_tb module that was compiled to work as the simulation top module.
• vsim -lib work -L ovi_lifcl -override_timescale=1ns/10ps top_tb -do {view wave; add wave /*; run 50ns;}
  • Invokes the ModelSim simulator, links the work and ovi_lifcl libraries to resolve instances.
  • Uses the top_tb module that was compiled to work as the simulation top module.
  • Sets the timescale in the included RTL to be 1 ns by 10 ps.
  • Opens the waveform display, adds all signals from the top module, and advances the simulation for 50 nanoseconds.

5.12. wave
Description: Collection of various commands that are used to make modifications to the waveform window. Requires that VSIM is already running, and that the waveform window is open.
Syntax: wave <arguments>
Arguments
• zoom in <zoom factor>
  • Zooms in the waveform display by the specified zoom factor.
  • If no zoom factor is specified, the display is zoomed in 2x.
• zoom out <zoom factor>
  • Zooms out the waveform display by the specified zoom factor.
  • If no zoom factor is specified, the display is zoomed out 2x.
• zoom full
  • Zooms out the waveform display to show the entire waveform.
• zoom range <left-side time value> <right-side time value>
  • Sets the left and right edge for the waveform display to the left and right-side values.
Example

- do {run -all; wave zoom full;}
  - Runs the simulation until a breakpoint is encountered, and then zooms out the entire waveform display.
- wave zoom range 5ns 20ns
  - Zooms into the range beginning with 5 ns and ending with 20 ns.
6. Useful TCL Commands and Options

This section outlines some additional TCL commands and options that are useful, but less commonly used for TCL scripting in ModelSim. For more detailed information about these TCL commands, or other TCL commands or options that were not mentioned in this section, refer to the ModelSim Command Reference Manual.

Each of the commands mentioned in this section has a description, syntax, arguments, and example subsection. Arguments in between arrows < > indicate the type of argument, which is typically some user input like a file name. Arguments in brackets [ ] are optional, while all other arguments are required.

6.1. formatTime

**Description:** Changes the global setting for all time values displayed in the ModelSim user interface.

Each argument requires either a + prefix to enable the setting, or – prefix to disable it. Using the command with no arguments returns the current setting values for each setting. The default is for all of these to be disabled (-).

**Syntax:** formatTime [<arguments>]

**Arguments**

- +commas | -commas
  - Adds (+) or removes (-) commas from time values.
- +nodefunits | -nodefunits
  - Does not include (+) or does include (-) default units in time values.
- +bestunits | -bestunits
  - Enables (+) or disables (-) unit swapping to.

**Example**

- formatTime +commas
  - Changes 1234567 ns to 1,234,567 ns in the user interface.
- formatTime +nodefunits
  - Changes 1234567 ns to 1234567 in the user interface.
- formatTime +bestunits
  - Changes 1200000 ns to 1.2 ms in the user interface.

6.2. layout load

**Description:** Opens the specified layout.

**Syntax:** layout load <layout name>

**Arguments**

- <layout name>
  - Name of the layout to load.

**Example**

- layout load my_layout
  - Loads the custom layout called my_layout.
6.3. onbreak

Description: This used in DO scripts to execute a command, set of commands, or script whenever a breakpoint is encountered in source code.

Syntax: onbreak <commands>

Arguments
- <commands>; {<commands>; …}
  - Command, set of commands, or script to execute when a breakpoint is encountered in source code.

Example
- onbreak break_script.tcl
  - Invokes the break_script.tcl script whenever a breakpoint is encountered.

6.4. onelaberror

Description: This used in DO scripts to execute a command, set of commands, or script whenever an error is encountered during elaboration when VSIM is invoked.

Syntax: onelaberror <commands>

Arguments
- <commands>; {<commands>; …}
  - Command, set of commands, or script to execute when an error is encountered during elaboration of VSIM.

Example
- onelaberror history
  - Prints a history of all commands executed when an elaboration error is encountered during VSIM.

6.5. onerror

Description: This used in DO scripts to execute a command, set of commands, or script whenever an error is encountered during the master scripts execution.

Syntax: onerror <commands>

Arguments
- <commands>; {<commands>; …}
  - Command, set of commands, or script whenever an error is encountered during the master script’s execution.

Example
- onerror {resume}
  - Resumes the script’s execution if an error is encountered.

6.6. onfinish

Description: This used in DO scripts to execute a command, set of commands, or script whenever a $finish is encountered during a simulation.

Syntax: onfinish <commands>

Arguments
- <commands>; {<commands>; …}
  - Command, set of commands, or script whenever a $finish is encountered during the simulation.

Example
- onfinish simstats
  - Invokes the simstats command, reporting the statistics for the current simulation when a $finish is encountered during the simulation.
6.7. precision

**Description:** This controls how numbers are displayed in the Objects, Wave, Locals, and List windows. Maximum precision value cannot exceed 17. The default precision is 6.

**Syntax:** precision [<digits> [#]]

**Arguments**
- `<digits> [#]`
  - Number of digits to display in the user interface.
  - # forces trailing zeros in the precision.
- no argument
  - Returns the current precision value.

**Example**
- precision 3
  - Results in 3 digits of precision.
- precision 5#
  - Results in 5 digits of precision with any missing decimals set to zero.
  - For example: 6.43 -> 6.4300.

6.8. quietly

**Description:** This turns off transcript echoing for the specified command.

**Syntax:** quietly <command name>

**Arguments**
- `<command name>`
  - ModelSim command to turn off transcript echoing for.

**Example**
- quietly vlog
  - Turns off transcript echoing for the VLOG command.

6.9. quit

**Description:** This quits the ModelSim simulator or ModelSim itself.

**Syntax:** quit <arguments>

**Arguments**
- `-f | -force`
  - Forces ModelSim to exit.
- `-sim`
  - Exits the VSIM simulator.
- `-code <integer>`
  - Exits Modelsim, reporting the exit code `<integer>`. 
6.10. radix

Description: This sets the default radix to use for a simulation.

Syntax: radix <radix type> [ <radix setting> ]

Arguments
- <radix types>
  - binary
  - Displays signals in binary format.
  - octal
  - Displays signals in octal format.
  - decimal | signed
  - Displays signals in signed decimal format.
  - hexadecimal
  - Displays signals using lower case hexadecimal values.
  - HEXADECIMAL
  - Displays signals using upper case hexadecimal values.
  - unsigned
  - Displays signals in unsigned decimal format.
  - ascii
  - Displays signals using 8-bit character encoding.
- <radix settings>
  - enumnumeric
    - Causes enumerations to appear as numbers.
  - enumsymbolic
    - Displays enumerations as symbols (default behavior).
  - Used to reverse the behavior of -enumnumeric.
  - showbase
    - Displays number of bits and radix for each signal.
    - E.g. the hexadecimal number 5532 will display as 32'h5532.

6.11. verror

Description: This reports additional information for a specific ModelSim message number, set messages, or type of messages. Useful in understanding why some errors are happening, and how to resolve them.

Syntax: verror <arguments>

Arguments
- <message number> [, <message number>, …]
  - Reports information about the specified message(s).
- -kind <tool>
  - Reports all messages from the specified ModelSim tool.
- -pedanticerrors | -permissive | -suppressibleerrors | -all
  - Used along with -kind to search for specific types of messages from a ModelSim tool.
  - Pedantic errors are messages with a strict interpretation of the Verilog LRM.
  - Permissive messages are warnings with less strict interpretation of Verilog LRM.
  - Suppressible errors are messages that can be suppressed.

Example
- verror 3033
  - Reports additional information about the error message #3033 to the transcript window.
• `verror -kind vcom -permissive`
  • reports all possible messages from the VCOM command.
  • Only messages of the permissive type are reported.

6.12. **where**

**Description:** This displays the current ModelSim directory, and active ModelSim settings (modelsim.ini). The reported ModelSim setting file is the one that library mappings are referenced from.

**Syntax:** where
7. Example Scripts

This section contains several example scripts that demonstrate various ways to simulate a project in ModelSim or do other ModelSim-related tasks such as compile custom libraries. The purpose of these scripts is to introduce users to the basic structure and some of the most used TCL commands for ModelSim scripts. Additionally, these scripts can also be used as a starting point for own custom ModelSim scripts.

7.1. Directory Based Simulation Script

The following shows how the directory-based simulation script works:

- `onerror (resume)` command causes the script to resume execution if it encounters an error.
- `cd` changes to the directory of the simulation project.
- The `if` block creates and maps a new work library if one does not already exist.
- The `vlog` command uses the `-y` option to compile the required files for simulation.
- `vsim` is used to invoke the ModelSim simulator simulating the top_tb module and linking the work and ovi_lifcl libraries.
- Additionally, all signals from the testbench, and from the uart instance in the DUT module are added to the waveform window, followed by the simulation advancing 50 ns and zooming out the waveform display.
7.2. Post-MAP with Timing Simulation Script

```plaintext
set proj_path C:/demo/postmap_sim
set proj_name postmap_timing

project new $proj_path $proj_name
project addfile $proj_path/rtl/top_impl1_mapvo.vo
project addfile $proj_path/rtl/top_tb.v

vlog -work work $proj_path/rtl/top_impl1_mapvo.vo
vlog -work work $proj_path/rtl/top_tb.v

vsim -lib work -L ovi_machxo3l \ 
+transport_path_delays +transport_int_delays top_tb \ 
-sdfmax /top_tb/DUT=$proj_path/rtl/top_impl1_mapvo.sdf

view wave
add wave /*
run 100ns
zoom wave full
```

The following shows how the post-MAP with timing simulation script works:

- `set` command is used to set variables for the location and name of the project.
- `project` commands are used to create a new project and add the simulation netlist and testbench to it.
- `vlog` command is used to compile the simulation netlist and testbench to work.
- `vsim` command is used to invoke the ModelSim simulator, linking the work and ovi_machxo3l libraries.
- `+transport_path_delays` and `+transport_int_delays` vsim options are used to change how signal delays are calculated through cells.
- `-sdfmax` option is used to apply the maximum delay from the specified SDF file to the simulation netlist.
7.3. Waveform Do Script

The following shows how the waveform DO script works:

- `onerror` command is used to resume script execution if an error is encountered for whatever reason.
- `add wave` command with various options are used to add signals to the waveform display into different groups, colors, and radices.
7.4. Compiling Files to a Custom Library

The following shows how to compile files to a custom library:

- **onerror** (resume)
- set libname mylib
- set libpath C:/demo/my_sim_lib
- vlib $libname
- vmap $libname $libpath
- vlog -work -sv $libname -f flist_v.f
- vcom -work $libname -f flist_vhd.f

The following shows how to compile files to a custom library script:

- **onerror** command causes script to resume execution if it runs into some error.
- **set** commands are used to set variables for the name and location of the new library.
  - The **libpath** variable assumes that directory already exists.
- **vlib** command is used to create a new library called `<lib name>`.
- **vmap** command is used to map the library `<lib name>` to the physical directory located at `<lib path>`.
  - This updates the library mappings in the active modelsim.ini file.
  - Ensure the active modelsim.ini is the master modelsim.ini so the library is accessible from any project.
- **vlog** command is used to compile Verilog and System Verilog modules from `<file>` to the `<lib name>` library.
- **vcom** command is used to compile VHDL modules from `<file>` to the `<lib name>` library.
References

- Lattice Radiant Software 3.2 User Guide
- Lattice Diamond 3.12 User Guide
- Lattice Propel 2.2 Builder User Guide
Technical Support Assistance
Submit a technical support case through www.latticesemi.com/techsupport.
## Revision History

**Revision 1.0, October 2022**

<table>
<thead>
<tr>
<th>Section</th>
<th>Change Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>