Scripting Lattice FPGA Build Flow

Application Note

FPGA-AN-02073-1.0

October 2023

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 all associated risk is the responsibility entirely of the Buyer. The information provided herein is for informational purposes only and may contain technical inaccuracies or omissions, and may be otherwise rendered inaccurate for many reasons, and Lattice assumes no obligation to update or otherwise correct or revise this information. 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. LATTICE PRODUCTS AND SERVICES ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS, OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING ANY APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, "HIGH-RISK USES"). FURTHER, BUYER MUST TAKE PRUDENT STEPS TO PROTECT AGAINST PRODUCT AND SERVICE FAILURES, INCLUDING PROVIDING APPROPRIATE REDUDANCIES, FAIL-SAFE FEATURES, AND/OR SHUT-DOWN MECHANISMS. LATTICE EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES. 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

Figures

Tables

Acronyms in This Document

A list of acronyms used in this document.

Acronym	Definition
ELF	Executable Linking File
FPGA	Field Programmable Gate Array
LSE	Lattice Synthesis Engine
SDK	Software Development Kit
TCL	Tool Command Language

1. Lattice Scripted Flow Overview

1.1. Types of Tool Automation Scripts

There are three main types of scripts that are used in the Lattice FPGA tool development flow: TCL, batch, and makefile. Depending on the type of script, there are different types of commands and command syntaxes. Aside from that, there are also differences in the way each script is used in the tool development flow also varies depending on the type of script being used. For example, TCL and batch scripts are often used to build an FPGA project from a set of RTL files, while makefile scripts are used in an embedded design flow to compile C/C++ applications.

1.1.1. TCL Scripts

TCL (Tool Command Language) scripts consist of TCL commands which are used to interface with Lattice's main design tools. Most things, which can be done in Lattice Radiant™, Lattice Diamond®, or Lattice Propel™ user interfaces, have corresponding TCL commands that can be used to reproduce the user interface-level behavior in a scripted design flow. These types of commands can be used to do things such as rebuild a project, interface with a debug project, configure IP, and more.

Aside from Lattice specific TCL commands, there are also global TCL commands which are built into each TCL interpreter and are not unique to the Lattice tool flow. These types of TCL commands are often used in conjunction with tool specific TCL commands in order to improve the robustness of a TCL script. One such example would be to use the "file exists" TCL command in an if else statement in order to determine whether a file has been generated, with the if else clause being used to generate the file if necessary. An example script demonstrating this type of TCL command usage is shown in the TCL Command Build Script section.

TCL scripts require a TCL interpreter in order to execute. All of Lattice's main software tools: Radiant, Diamond, and Propel come with a built-in TCL console which can be used directly from each tool's respective user interface. A useful feature of these integrated TCL consoles it that as you use each tool's user interface, their respective TCL commands are output here. The advantage of this is that you can use the TCL commands that are output here in order to understand what commands you should use in a scripted flow in order to reproduce user interface behavior. Aside from the built-in TCL console, all the tools mentioned also have standalone interactive TCL console's which can be used to execute TCL commands and invoke TCL scripts entirely separately from a tool's user interface. For more information on how to invoke TCL scripts in the Lattice tool flow, refer to Invoking TCL Scripts in Lattice Diamond and Invoking TCL Scripts in Lattice Radiant.

1.1.2. Batch Scripts

Batch scripts consist of command-line level commands, which are a lower-level than TCL commands and typically require additional setup for the commands to be accessible and executable at the command-line level. This setup typically involves setting some environment variables in order to establish the location of the active Lattice tool installation. The primary use for batch scripts is to invoke core Lattice tool processes such as synthesis, MAP, or place and route in order to automate a design's build flow entirely separate from any Lattice tool's user interface. Something to keep in mind is that not all user interface-level functions have equivalent command-line batch mode commands.

1.1.3. Makefile Scripts

Makefile scripts are specific to the Lattice Propel tool flow and are primarily used to compile code for embedded C/C++ projects. It is not required for users to create their own makefile scripts as Lattice Propel automatically generates the required compilation scripts for a project. The primary usage for a customized makefile script in the Lattice tool flow is to build an embedded project differently than the default settings. One such example would be to generate an additional memory initialization file with a different format than hex such as binary. For more information about makefile scripts and how they fit into the Lattice Propel tool flow, refer to Lattice Propel SDK Makefile Scripting section.

2. Scripting with Lattice Diamond

2.1. General Information

The first thing to consider when creating a script to automate the Lattice Diamond tool flow is how much of the flow you want to script. If you intend to simply automate a design's build flow, either a TCL script or batch script can be used since both script types can use commands to run through synthesis, MAP, PAR, and bitstream generation. However, if you intend to reproduce some user interface-level functionality such as Reveal Analyzer debugging or memory initialization through ECO editor, you need to use a TCL script as there are no equivalent batch mode commands for either of these features.

To find more information about Diamond's various TCL and batch commands and their respective options, refer to the *Command Line Reference Guide* and *TCL Command Reference Guide* sections of the Diamond web help (*Help > Lattice Diamond Help*). Aside from the web help, each command also has built-in help information which can be accessed by typing *-help* following the name of the command (such as *synthesis -help*).

2.1.1. TCL Command Log

Another feature of the Lattice Diamond is the TCL command log, which can be accessed from the Diamond's report menu. As you develop your projects using the Diamond's user interface, a history of all the TCL commands that were executed from each session is stored in the TCL command log report. This report is useful as it can be used to easily understand the commands that should be used in order to reproduce user interface-level functionality in a TCL script. In most cases, these commands can be directly copied from this report to a TCL command script.

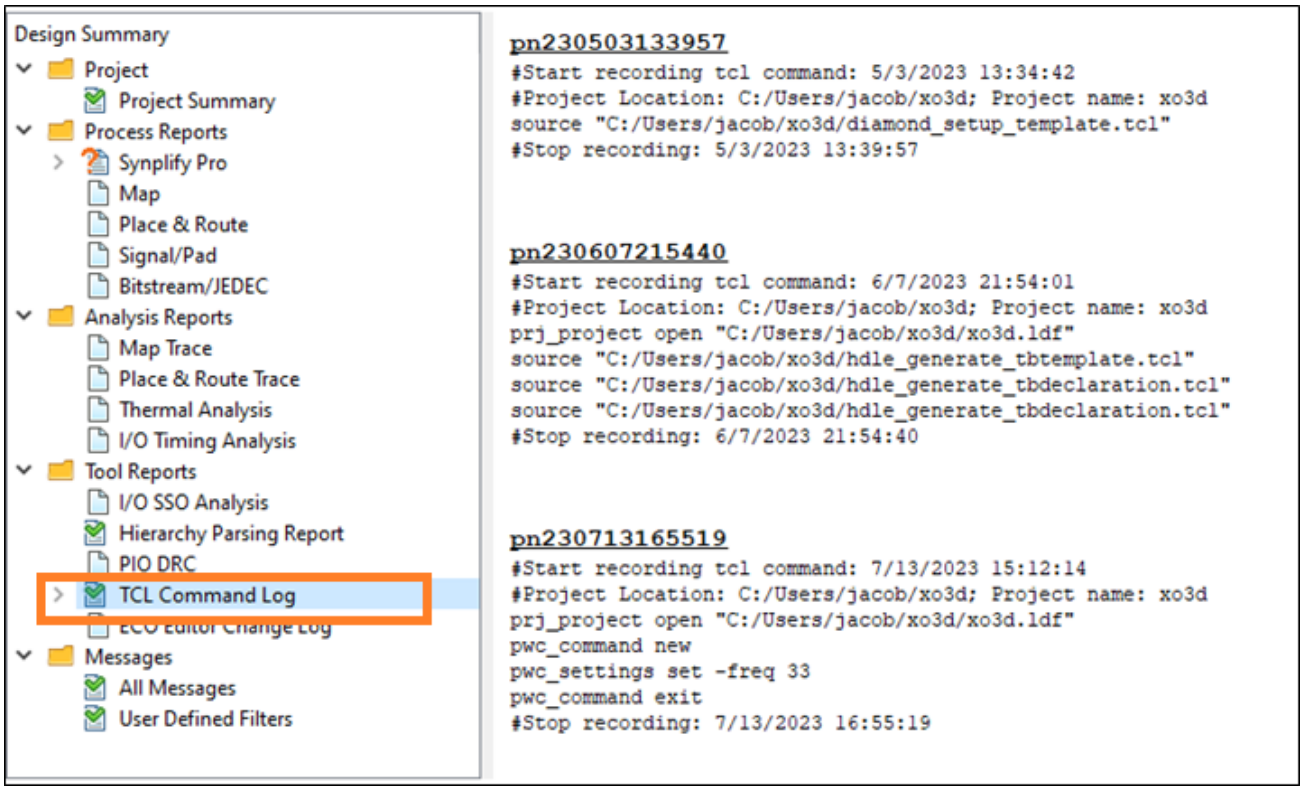

Figure 2.1. Lattice Diamond TCL Command Log Report

2.1.2. Finding Batch Commands

Another feature of Lattice Diamond is its ability to allow you to parse through the tool's output using *CTRL + F*. Using this method, you can parse through the Diamond's tool output in order to find the batch commands that are executed in order to build the project. This is useful to get an idea of the command's syntax and what options to use considering most batch commands have more options than their TCL equivalents. For example, when you search *synwrap* in the Diamond tools' output after running the synthesis, you can find the batch mode command that you need to run through synthesis in a batch script.

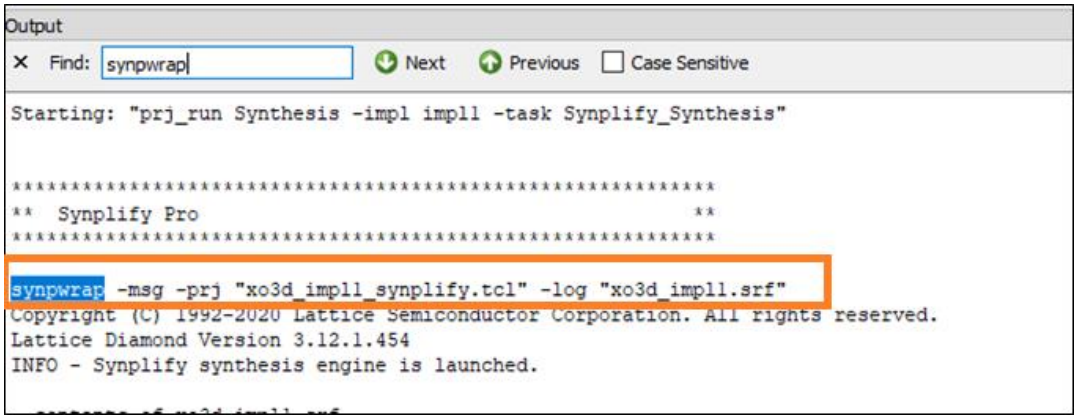

Figure 2.2. Searching Lattice Diamond Console Output for the synpwrap Command

2.2. Lattice Diamond TCL Scripting

2.2.1. Invoking TCL Scripts in Lattice Diamond

There are two main ways to invoke a TCL script in the Lattice tool flow, using either the built-in TCL console or by launching a script on tool startup.

In order to invoke a TCL script in the built-in TCL console, simply type *source* followed by the location and name of the TCL script you want to invoke (for example, *source /home/<user home directory>/projects/build.tcl*). Note that in this method of invoking TCL scripts, the *source* is a TCL command itself which can be used to invoke a TCL script from within a TCL script. For example, you can invoke TCL script 1 from the command-line using the *source* command, and then have the same TCL script #1 invoke another TCL script 2 by inputting another *source* command within the original script.

The second method for invoking TCL scripts in the Lattice tool flow is to invoke them on tool startup. To do this, launch the Diamond tool directly from the command line with the startup script specified as an option. The exact syntax and method for invoking a script on tool startup varies depending on the operating system.

- Windows
  - Launch Diamond user interface and run the script:
    - *<Diamond install path>/bin/nt64/pnmain.exe -t <TCL script location>/<TCL script name>.tcl*
    - Launch Diamond Console Mode and run the script:
    - <Diamond install path>/bin/nt64/pnmainc.exe <TCL script location>/<TCL script name>.tcl
- Linux
  - Launch Diamond user interface and run the script:
    - *<Diamond install path>/bin/lin64/diamond -t <TCL script location>/<TCL script name>.tcl*
  - Launch Diamond Console Mode and run the script:
    - *<Diamond install path>/bin/lin64/diamondc <TCL script location>/<TCL script name>.tcl*

2.2.2. Main Build Flow TCL Scripting

In general, the process for TCL scripting the main build flow in Diamond is simple and only requires a single command (*prj_run*) with a few command variants, assuming that a project is already created and has its associated files added. The general syntax for this command is *prj_run <Implementation stage> -impl <implementation name>*, where the implementation stage is the specific process that you want to run your project through (such as synthesis, MAP, and PAR). A useful feature of the *prj_run* command is only a single command is required to build an entire project from synthesis to bitstream generation, as *prj_run Export -impl …* automatically runs through all stages before bitstream generation. For example, if nothing has been run, then synthesis, MAP, and PAR automatically runs before running through bitstream generation.

- TCL Command – PRJ_RUN
  - Syntax – prj_run <implementation stage> -impl <implementation name> [-forceOne | -forceAll]
  - Command options:
    - Implementation stage:
      - synthesis (LSE and Synplify Pro®)
      - translate (Synplify Pro only)
      - MAP
      - PAR
      - Export
    - Implementation name – Name of the active project implementation.
    - -forceOne – Optional; Reruns the specified stage regardless of whether it has already been run or not. Default behavior is to not rerun a stage that is already ran if this option is not included.
    - -forceAll – Optional; Reruns the specified stage and all stages before it regardless of whether or not they have already been run. The default behavior is not to rerun a stage that is already ran if this option is not included.

2.3. Lattice Diamond Batch Scripting

2.3.1. Setting up the Batch Environment

Before Diamond's main build commands can be used in a batch script, there is some additional setup required. This setup is always at the beginning of each script and is required in order to configure the command line environment to recognize Lattice batch mode commands. The exact process for setting up the batch environment is operating system dependent, and generally involves setting a few environment variables that have to do with Diamond's installation path.

2.3.1.1. Setting Up the Batch Script in Windows

To setup the batch script in Windows:

1. Create a new .bat file.
2. Add the following lines at the beginning of the script:
   a. set PATH=<Diamond install path>/bin/nt64;<Diamond install path>/ispfpga/bin/nt64
   b. set FOUNDRY=<Diamond install path>/ispfpga
3. Configure the remainder of the script using batch mode commands for each stage of the Diamond project flow (synthesis, MAP, and PAR).
   a. For more information about this portion of the script, refer to section 2.3.2 Main Build Flow
   b. To run the script, the name in the Windows command line or powershell. For example, *C:/Users/<user home directory>/projects/my_xo2proj/build_design.bat*.

2.3.1.2. Setting Up the Batch Script in Linux

To setup the batch script in Linux:

1. Create a new .bat file by running the *touch <script name>.bat* command.
2. Make the batch script executable by running the *chmod +x <script name>.bat* command.

3. Add the following lines at the beginning of the script:
   a. export bindir=<Diamond install path>/bin/lin64
   b. source $bindir/diamond_env
4. Configure the remainder of the script using batch mode commands for each stage of the Diamond project flow (synthesis, MAP, and PAR).
5. To run the script, type the name in the Linux command line. For example, */home/<user home directory>/projects/xo2proj/build.bat*.

2.3.2. Main Build Flow Batch Scripting

In general, the process of batch scripting the main portion of the Diamond's build flow consists of a few commands that take the input of the previous command in order to generate the next intermediary file in the build flow. For the most part, the commands required to build a project from RTL to a programming file are the same and vary only depending on the tool you want to use for synthesis.

If you choose LSE as the synthesis tool, only the *synthesis* command is required. However, if selected synthesis tool is Synplify Pro, there are a few additional commands that are required. To perform synthesis with Synplify, the *synpwrap* command is used to interface with the Synplify Pro batch mode engine and generate an EDIF file. However, there is an additional translate stage that must be done before the synthesis output is passed to MAP which involves the *EDIF2NGD* and *NGDBUILD* commands to convert the EDIF file into an NGD.

Once the NGD file is generated using the two synthesis commands, the remainder of the batch mode flow is the same, requiring users to invoke MAP, PAR, and BITGEN in order to generate the programming file for the project. Figure 2.3 shows the Diamond batch mode tool flow.

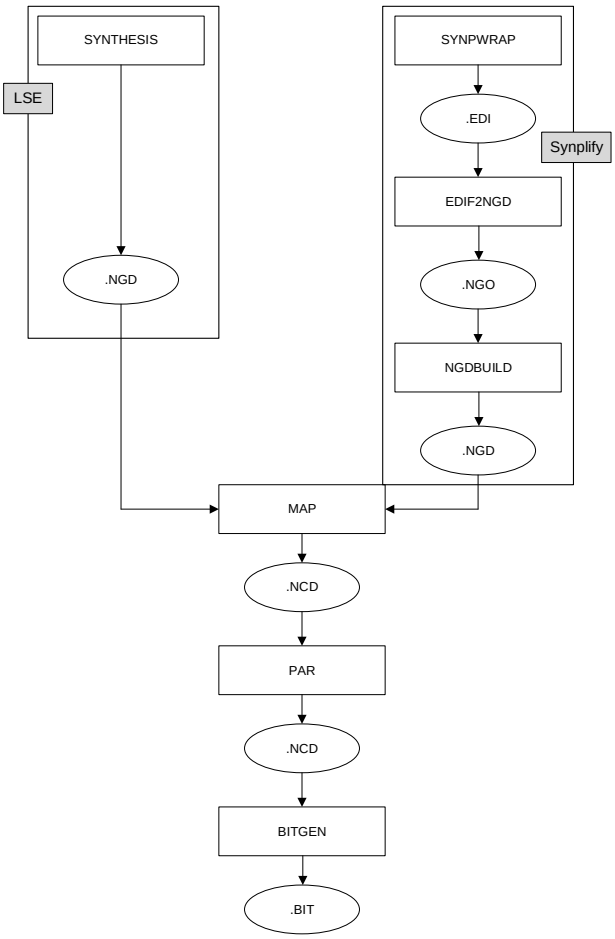

Figure 2.3. Lattice Diamond Batch Mode Tool Flow

3. Scripting with Lattice Radiant

3.1. General Information

When creating a script to automate the Lattice Radiant tool design flow, consider what parts of the Radiant tool development you want to script. If you want to automate the design's build flow, you can use either a TCL script or batch script since both script types can use commands to run through synthesis, MAP, PAR, and bitstream generation. However, if you intend to reproduce some user interface-level functionality, such as Reveal Analyzer debugging or memory initialization through ECO editor, you need to use a TCL script as there are no equivalent batch mode commands for either of these features.

To find out more information about the Lattice Radiant's various TCL and batch commands and their respective options, refer to the *Command Line Reference Guide* and *TCL Command Reference Guide* sections of the Diamond web help (*Help > Lattice Radiant Software Help*). Aside from the Radiant web help, each command also has a built-in help information, which can be accessed by typing *-help* following the name of the command (for example, *synthesis -help*).

3.1.1. Useful Reports

3.1.1.1. TCL Command Log Report

The TCL command log is one of the features of Lattice Radiant and can be accessed by going to *Reports > Misc Reports > TCL Command Log***.** As you develop your projects using the Lattice Diamond's user interface, a history of all TCL commands that are executed from each session is stored in this TCL command log report. This report is useful as it can be used to understand the commands that must be used in order to reproduce user interface-level functionality in the TCL script. In most cases, these commands can be directly copied from the report to a TCL command script.

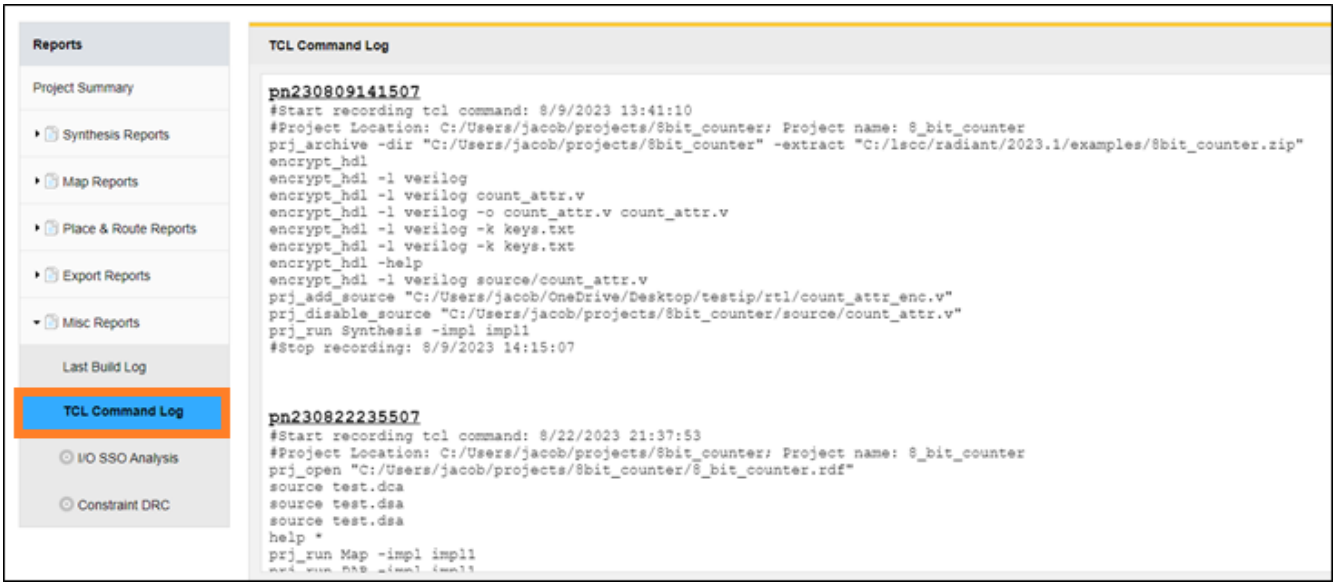

Figure 3.1. Lattice Radiant TCL Command Log Report

3.1.1.2. Last Build Log Report

Another useful feature of Lattice Radiant is its Last Build Log report. This report contains the output of Radiant's console from the last time the project was built and is easily parseable in order to find the batch mode commands that were executed by Radiant. By searching through this report for the specific batch mode build commands, you can easily identify what commands and command options you need to use in the batch mode scripts in order to rebuild the design from the command line.

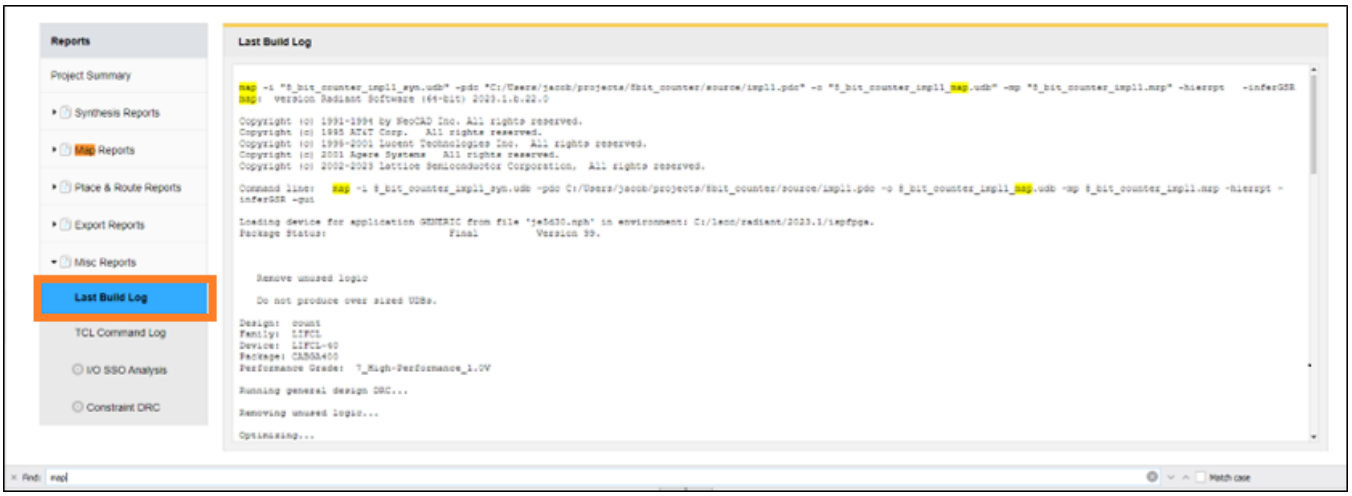

Figure 3.2. Lattice Radiant Last Build Log Report

3.1.2. Finding Batch Commands

Aside from the reports mentioned, another method for finding batch commands in Radiant is to directly parse through the console output using *CTRL + F*. Using this method, you can parse through the Radiant's tool output in order to find the batch commands that were executed in order to build their project. This is useful to get an idea of a command's syntax and what options to use considering most batch commands have more options than their TCL equivalents. For example, by searching for *par* in the Radiant tool output after running synthesis, you can easily find the batch mode command that you need to use in order to run *Place and Route* again using the same files in a batch script.

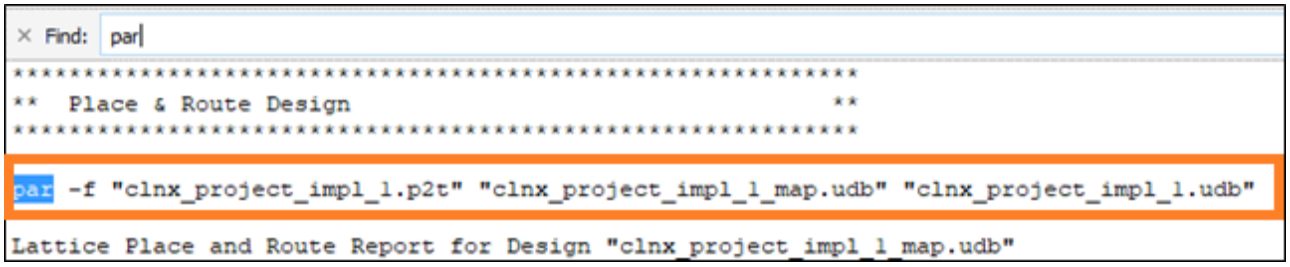

Figure 3.3. Lattice Diamond Console Search Output for the synpwrap Command

3.2. Lattice Radiant TCL Scripting

3.2.1. Invoking TCL Scripts in Lattice Radiant

There are two main ways to invoke a TCL script in the Lattice tool flow, using either the built-in TCL console or by launching a script on tool startup.

In order to invoke a TCL script in the built-in TCL console, simply type "source" followed by the location and name of the TCL script you want to invoke (for example, *source /home/<user home directory>/projects/build.tcl*). Something else to keep in mind regarding this method of invoking TCL scripts, is that *source* is a TCL command itself which can be used to invoke a TCL script from within a TCL script. For example, you can invoke TCL script 1 from the command-line using the "source" command, and then have the same TCL script 1 invoke another TCL script 2 by inputting another "source" command within the original script.

The second method for invoking TCL scripts in the Lattice tool flow is to invoke them on tool startup. To do this, Radiant must be launched directly from the command line with the startup script specified as an option. The exact syntax and method for invoking a script on tool startup varies depending on the operating system.

- Windows
  - Launch the Radiant user interface and run the script:
    - *<Radiant install path>/bin/nt64/pnmain.exe -t <TCL script location>/<TCL script name>.tcl*
  - Launch the Radiant Console Mode and run the script:
    - *<Radiant install path>/bin/nt64/pnmainc.exe <TCL script location>/<TCL script name>.tcl*
- Linux
  - Launch Radiant user interface and run the script:
    - *<Radiant install path>/bin/lin64/radiant -t <TCL script location>/<TCL script name>.tcl*
  - Launch Radiant Console Mode and run the script:
    - *<Radiant install path>/bin/lin64/radiantc <TCL script location>/<TCL script name>.tcl*

3.2.2. Main Build Flow TCL Scripting

In general, the process for the main build flow TCL scripting in Radiant is simple and only requires a single command (*pri_run*) with a few command variants, assuming that a project is already created and has its associated files added. The general syntax for this command is *prj_run <Implementation stage> -impl <implementation name>*", where the implementation stage is the specific process that you want to run your project through (such as synthesis, MAP, and PAR). A useful feature of the prj_run command is that only a single command is required to build an entire project from synthesis to bitstream generation, as the *prj_run Export -impl …*" automatically runs through all stages before bitstream generation. For example, if nothing has been run, then synthesis, MAP, and PAR automatically runs before running through bitstream generation.

- TCL Command: PRJ_RUN
  - Syntax – prj_run <implementation stage> -impl <implementation name> [-forceOne | -forceAll]
  - Command options:
    - Implementation stage:
      - synthesis (LSE and Synplify Pro)
      - MAP
      - PAR
      - Export
    - Implementation name – name of the active project implementation
    - -forceOne – Optional; Reruns the specified stage regardless of whether it has already been run or not. Default behavior is to not rerun a stage that is already ran if this option is not included.
    - -forceAll – Optional; Reruns the specified stage and all stages before it regardless of whether they have already been run or not. Default behavior is to not rerun a stage that is already ran if this option is not included.

3.3. Lattice Radiant Batch Scripting

3.3.1. Setting up the Batch Environment

Before the Radiant's main build commands can be used in a batch script, there is an additional setup required. This setup is always at the beginning of each script and is required in order to configure the command line environment to recognize Lattice batch mode commands. The exact process for setting up the batch environment is operating system dependent and involves setting a few environment variables that have to do with the Radiant's installation path.

3.3.1.1. Setting up the Batch Script in Windows

To set up the batch script in Windows:

1. Create a new .bat file.
2. Add the following two lines to the beginning of the script:
   - set PATH=<Radiant install path>/bin/nt64;< Radiant install path>/ispfpga/bin/nt64
   - set FOUNDRY=< Radiant install path>/ispfpga
3. Configure the remainder of the script using batch mode commands for each stage of the Radiant project flow (synthesis, MAP, and PAR).

a. For more information about this portion of the script, refer to Batch Scripting the Main Build Flow section.

4. To run the script, type the name into the Windows command line or powershell. For example, *C:/Users/Jacob/projects/clnx_soc_proj /build.bat*.

3.3.1.2. Setting up a Batch Script in Linux

To set up the batch script in Linux:

1. Create a new .bat file: *touch <script name>.bat*.
2. Make the batch script executable: *chmod +x <script name>.bat*.
3. Add the following two lines to the beginning of the script:
   - export bindir=<Radiant install path>/bin/lin64
   - source $bindir/radiant_env
4. Configure the remainder of the script using batch mode commands for each stage of the Radiant project flow (such as Synthesis, MAP, and PAR).
5. To run the script, simply type it's name in the Linux command line. For example, */home/<user home directory>/projects/clnx_soc/build.bat*.

3.3.2. Batch Scripting the Main Build Flow

Overall, the process of batch scripting the main portion of the Radiant's build flow consists of a few commands that take the input of the previous command in order to generate the next intermediary file in the build flow. For the most part, the commands required to build a project from RTL to a programming file are the same and only varies depending on the tool you want to use for synthesis.

If LSE is the synthesis tool of choice, only the *synthesis* command is required. However, if Synplify Pro is the selected synthesis tool, you must use the *synpwrap* command instead. From this point onwards, the remainder of the batch mode build flow is the same, requiring the *postsyn* command to convert the output of both synthesis engines into UDB format, which is used for the remainder of the flow.

Once a UDB file is generated by *postsyn*, the remainder of the batch mode flow is the same, which requires you to invoke MAP, PAR, and lastly BITGEN in order to generate the programming file for the project. Refer to Figure 3.4 for a more detailed graphic of the Diamond batch mode tool flow.

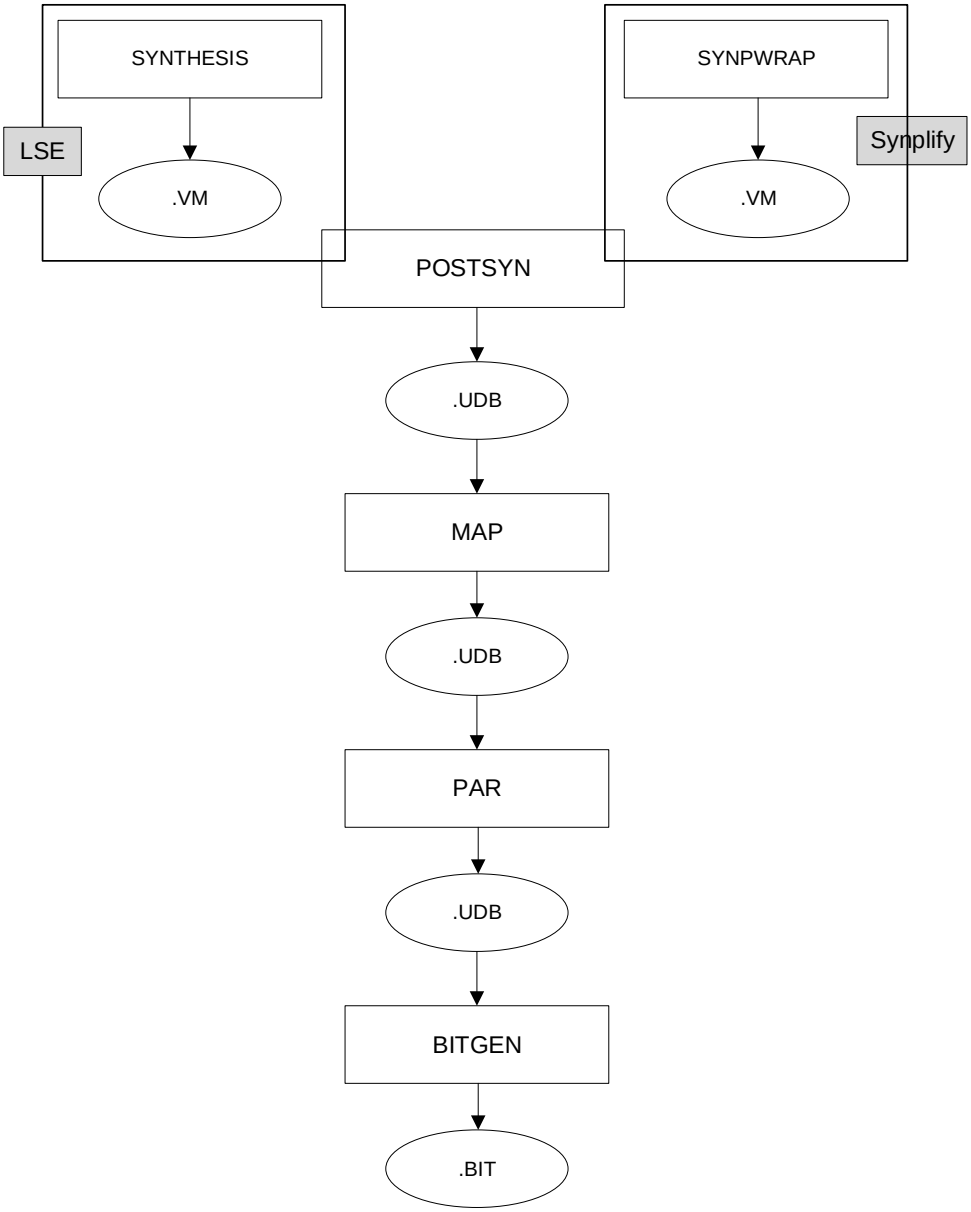

Figure 3.4. Lattice Radiant Batch Mode Tool Flow

4. Scripting with Lattice Propel

4.1. General Information

The two main types of scripts that can be used in the Lattice Propel embedded development flow are TCL and makefile. For Lattice Propel Builder, the TCL scripts can be used in order to reproduce user interface functionality and perform tasks such as create a project, generate IP, and manage address spaces separately from the Propel Builder's main user interface. Although the Propel Builder TCL scripts require a little more effort to setup, these scripts can be used to easily replicate the entire SoC project build flow without having to interact with the tool user interface.

Aside from TCL scripts, the other type of script that can be used in the Propel's embedded development flow is the makefile scripts. This script is used in the Lattice Propel SDK to compile codes and generate memory initialization files. As mentioned, these scripts are automatically generated before a project is built and require no user intervention in order to build a project. In general, the makefile script customization is done to generate additional files from compilation, mostly in a different format or type.

4.2. TCL Scripting Propel Builder

As mentioned before, most of the Propel Builder's user interface functionalities can be reproduced in a scripted flow using TCL commands. Table 4.1 lists the some of the frequently used Propel Builder TCL commands.

Table 4.1. Propel Builder Features and TCL Commands

User Interface Feature	Associated TCL Command(s)	Description
Project Creation	sbp_design new	Create a new SoC project from a template.
Project Opening	sbp_design open	Open an existing SoC project using .sbx file.
IP Generation	ipgen	Generate an IP from scratch or regenerate an existing IP.
IP Instantiation	sbp_add_component	Instantiate a generated IP to the active SoC design.
Sub-block Instantiation	sbp_add_sbxcomp	Instantiate a .sbx component as a sub-block to the active SoC design.
Adding Glue Logic	sbp_add_gluelogic	Implement custom glue logic in an SoC design, such as custom RTL, bus concatenation, an inverter, or bus split.
Port Creation	sbp_add_port	Create a new top-level port.
Port Editing	sbp_modify_port	Edit an existing top-level port to modify its name, width, or direction.
Port Exporting	sbp_export_pins, sbp_export_interface	Export any unconnected ports or interfaces of a component to the top-level of a design.
Connecting Components	sbp_connect_net, sbp_connect_interface_net, sbp_connect_group	Connect the ports or interfaces of a source component to another destination component.
Address Space Management	sbp_assign_addr_seg, sbp_design auto_assign_addresses	Manually or automatically set the base address for a memory mapped component in the active SoC project.
Address Space Verification	sbp_design drc	Perform a design rule check to ensure the validity of the active SoC design's address space and memory mapped components.
Generate RTL Wrapper	sbp_design generate	Regenerate all the RTL in the project, including IP and the top-level Verilog or VHDL wrapper.
Generate Tool Scripts	sbp_design pge dge	Generate TCL scripts to export the active SoC design to Diamond or Radiant depending on the current device.
Generate SW Project Files	sbp_design pge sge	Generate system platform definition files, IP drivers, and system environment XML file which is used in Propel SDK C/C++ projects.

For more information on any of the TCL commands mentioned, refer to the TCL Commands section in the Lattice Propel Builder user guide. In addition, all TCL commands have help information which can be accessed by typing *-help* after any TCL command (for example, *sbp_design -help*).

4.2.1. Invoking TCL Scripts

Like Lattice Radiant and Lattice Diamond, there are two ways to invoke TCL scripts in the Propel Builder, using either the built-in TCL console or by launching a script on the tool startup.

In order to invoke a TCL script in the built-in TCL console, type *source* followed by the location and name of the TCL script you want to invoke (for example, *source /home/<user home directory>/projects/build.tcl*). Note that *source* is actually a TCL command, which can be used to invoke a TCL script from within a TCL script. For example, you can invoke TCL script 1 from the command line using the *source* command. Then, the TCL script 1 invokes TCL script 2 by typing another *source* command within the original script.

Another method for invoking the TCL scripts in the Lattice tool flow is to invoke them on the tool startup. To do this, the Propel Builder must be launched directly from the command line with the startup script specified as an option. The exact syntax and method for invoking a script on tool startup varies depending on the operating system.

- Windows
  - Launch the Propel Builder user interface and run the script below:
    - *<Propel install path>/builder/rtf/bin/nt64/propelbld.exe <TCL script location>/<TCL script name>.tcl -gui*
  - Launch Propel Builder Console Mode and run the script below:
    - *<Propel install path>/builder/rtf/bin/nt64/propelbld.exe <TCL script location>/<TCL script name>.tcl*
- Linux
  - Launch Propel Builder user interface and run the script below:
    - *<Propel install path>/builder/rtf/bin/lin64/propelbldwrap <TCL script location>/<TCL script name>.tcl -gui*
  - Launch Propel Builder Console Mode and run the script below:
    - *<Propel install path>/builder/rtf/bin/lin64/propelbldwrap <TCL script location>/<TCL script name>.tcl*

4.3. Lattice Propel SDK Makefile Scripting

The purpose of the make utility in the Propel's embedded development tool flow is to compile and convert source code into executables that can run on actual hardware. A makefile script consists of a set of definitions and rules, which are used to perform all stages of the compilation flow in order to convert a project's source code into the final compilation output. Figure 4.1 shows a high-level overview of the contents of the Propel's autogenerated makefile script.

In the Propel project development flow, you are not required to create your own makefile scripts to compile your C/C++ projects, Propel automatically generates the necessary build script. In fact, it is recommended that you do not edit the generated makefiles as these are regenerated each time a project is rebuilt. This means any changes made directly to these files are overwritten. Instead of directly modifying the Propel generated makefiles, the generated makefile provides several hooks for you to integrate your own makefile scripts (such as makefile.init, makefile.defs, and makefile.targets). Refer to Customizing the Make Build Flow section on how to use these hooks and how they work.

Figure 4.1. High-Level Overview of Lattice Propel Auto-Generated Makefile Scripts

Aside from the optional makefile hooks, the overall structure of the makefile script can be summarized in three main sections. The first are the includes, which include the subdir.mk files for the entire project. These subdir.mk files contain makefile variable assignments and rules that describe how to assemble the source code. Next is the variable assignments, which define the main variables used to build the entire project. The variables are used in the final portion of the script, which are the build rules that describe what needs to be built, what items are required to build the project, and all the other steps required to generate the final memory initialization files.

The build rules begin from a set of source files, which are converted into assembly files by the GCC or G++ compiler that performs preprocessing, compilation, optimization, and code generation. These assembly files are then converted to object files by the assembler. Next, the linker script is used to combine the generated object files with any precompiled libraries in order to output an ELF (executable linking file). Finally, the GNU objcopy utility is used in order to generate the final output memory initialization file for the Propel design.

4.3.1. Customizing the Make Build Flow

As mentioned, the Propel SDK automatically generates the required makefile script and associated dependency files in order to compile a C/C++ project, so no user intervention is required in order to compile a design. The auto-generated makefile script provides three main entry points for you to customize your make build flow. Figure 4.2 shows lines 5, 18, and 49 contain three files called makefile.init, makefile.defs, and makefile.targets. These files are not automatically generated, and you must create this from scratch in the top-level C/C++ project directory if you want to customize your makefile build flow. If these files do not exist, the make utility skips over these includes and continue with the rest of the build flow.

Although the exact contents of the additional makefile scripts vary depending on your intention, the makefile.defs is well-suited for custom variable definitions due to its location in the makefile build flow. In addition, the makefile.targets is useful for adding additional targets, rules, outputs, or tasks that are not natively supported by default in the Propel SDK.

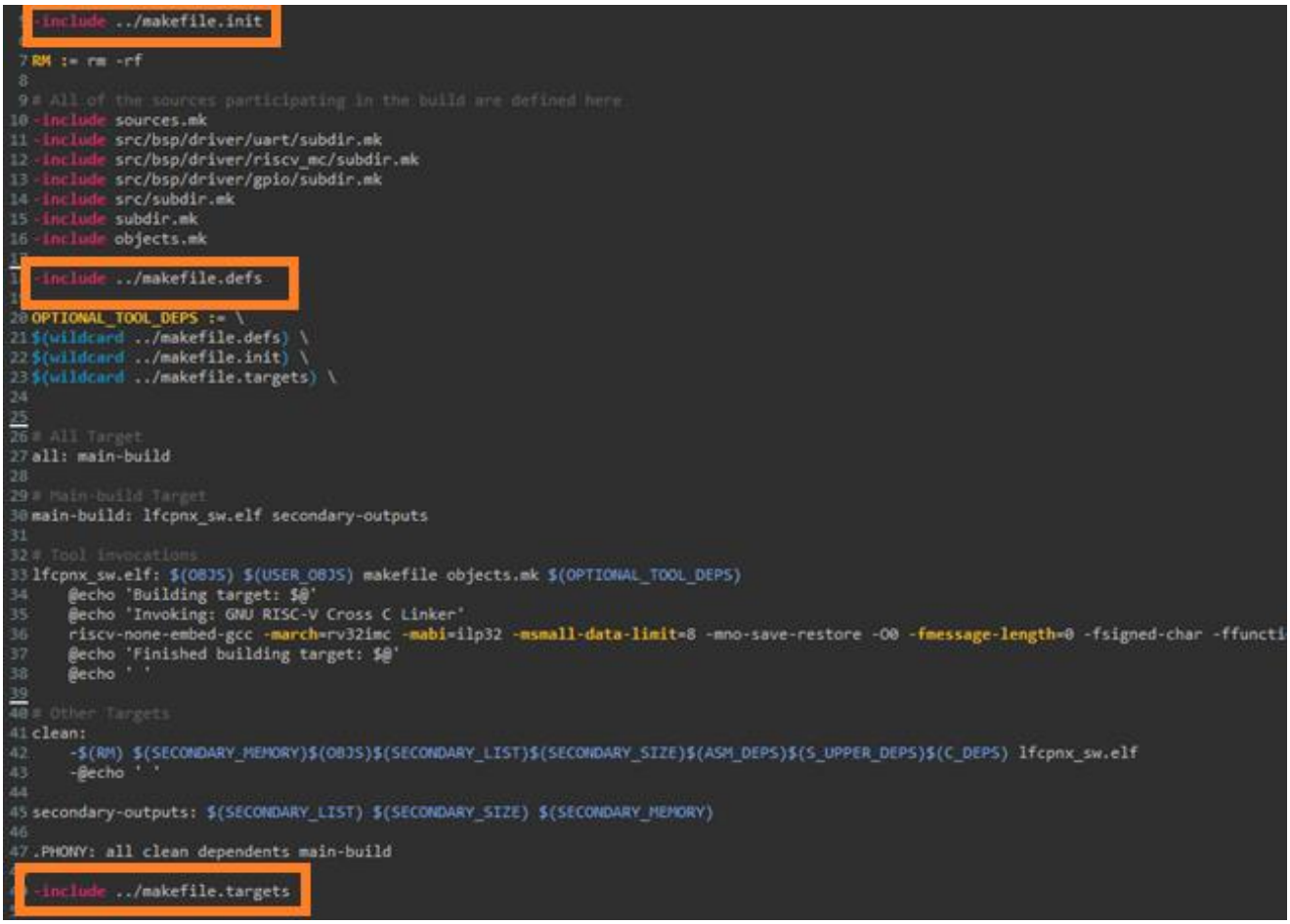

Figure 4.2. Auto-Generated Makefile Build Script Overview

One potential use case for the makefile customization is to automatically initialize a design's system memory IP each time a C/C++ project is built. Figure 4.3 shows how to create an additional makefile script called makefile.targets with a rule to do some task external from Propel SDK. The syntax for this file is fairly simple, initially requiring *secondary-outputs: <rule name>* at the top of the script. If there are multiple rules in the file, it needs to be included in the same line (for example, *secondary-outputs rule1 rule2 rule3…*). As shown from line 45 in Figure 4.2, the secondary-outputs tag is used as the entry point to the makefile build flow and invokes all rules within the target.

The next part of the makefile.targets script consists of the actual rules that are invoked by the secondary-outputs. The first line *sysmem_init:* is the name of the rule and indicates that the following lines must be executed as part of the rule. Next, the Propel Builder's interactive TCL console is invoked with the *ipgen.tcl* script in order to automatically regenerate the system memory IP with the updated memory initialization file.

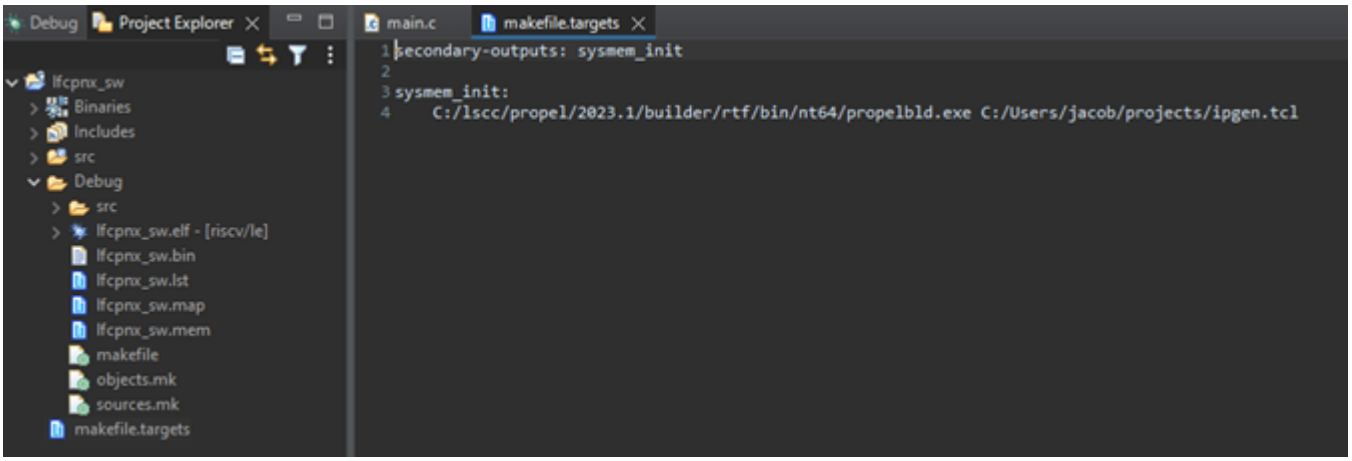

Figure 4.3. makefile.target Script Content to Automatically Initialize the Design System Memory IP

Figure 4.4 shows the contents of the *ipgen.tcl* TCL script. This script is straightforward, using the *set* command to set the project variables that specify the location of the system memory IP to be configured, its original source, and the parameter configuration file. The main portion of this script is lines 7 to 9, which utilizes the *exec* command to invoke the *ipgen.exe* command line executable, which is used to regenerate the system memory IP. The only other external file required in the usage flow is the IP parameter configuration file, as shown in Figure 4.5, which contains the IP parameters being set and the corresponding values. In this particular example, the location of the new memory initialization file is hardcoded, so the IP is regenerated each time this script is run.

```
1
2   set sysmem_dir "C:/Users/jacob/lfcpnx_soc/lfcpnx_soc/lib/latticesemi.com/ip/sysmem0/2.0.0"
3   set sysmem_src "C:/lscc/propel/2023.1/builder/rtf/ip/common/system_memory"
4
5   set sysmem_cfg "C:/Users/jacob/lfcpnx_soc/mem_cfg.cfg"
6
7   exec C:/lscc/propel/2023.1/builder/rtf/ispfpga/bin/nt64/ipgen.exe -o $sysmem_dir \
8   -ip $sysmem_src -name "sysmem0" -platform "Propel" -a "LFCPNX"  -p "LFCPNX-100" \
9   -t "LFG672" -sp "8_High-Performance_1.0V" -cfg $sysmem_cfg
```

Figure 4.4. ipgen.tcl Script Content Used to Regenerate the Design System Memory IP

```
1   {
2     "ADDR_DEPTH": 8192,
3     "PORT_COUNT": 2,
4     "INIT_MEM": true,
5     "INIT_FILE": "C:/Users/jacob/lfcpnx_soc/workspace/lfcpnx_sw/Debug/lfcpnx_sw.mem",
6     "REGMODE_S0": true,
7     "REGMODE_S1": true
8   }
```

Figure 4.5. mem_cfg.cfg File Content Used to Regenerate the Propel System Memory IP

5. Example Build Scripts

5.1. Lattice Radiant Build Flow

5.1.1. TCL Command Build Script

```
2   set proj_name "clnx_proj_tcl"
3   set proj_dir "/home/jmercado/my_designs/${proj_name}"
4   set proj_rdf "${proj_dir}/${proj_name}.rdf"
5
6   set device "LIFCL-40-8BG256C"
7   set speed "8_High-Performance_1.0V"
8
9
10  if { [file exists $proj_dir] == 0} {
11    file mkdir $proj_dir
12    cd $proj_dir
13
14    prj_create -name $proj_name -impl "impl_1" -dev $device -performance $speed -synthesis "synplify"
15
16    prj_add_source "/home/jmercado/rtl/count_attr.v"
17    prj_add_source "/home/jmercado/rtl/my_osc/my_osc.ipx"
18
19  } else {
20    prj_open $proj_rdf
21
22  }
23
24  puts "Starting to build the design"
25  prj_run Export -impl impl_1 -forceAll
26  prj_save
```

Figure 5.1. Lattice Radiant TCL Build Script Example

The following shows the description of the TCL command build script:

- Line 2 to Line 7
  - Basic project related parameter definitions used to simplify the script and improve its readability.
- Line 10 to Line 18
  - If portion of an if else block, which checks for the existence of the project directory.
  - If the project directory does not exist, the mkdir file is used to create it.
  - cd is used to change the current directory to the current project's directory.
  - prj_create is used to create a new Radiant project using the specified device settings.
  - To finish the project setup, prj_add_source is used to add the source RTL for the design.
- Line 19 to Line 22
  - Else portion of an if else block. In this block, it is assumed the project already exists. Therefore, there is nothing to be done to set it up.
  - cd is used to change the current directory to the project's directory.
- Line 24 to Line 26
  - puts is used to print out a statement indicating that the design is about to be built.
  - prj_run export runs through the entire design flow beginning from synthesis up until bitstream generation.
    - -forceAll option is used to force every stage of the build flow to rerun each time the script executes.

5.1.2. FPGA Build Flow Batch Script (Windows)

```
1  export bindir="/home/jmercado/lscc/radiant/2023.1/bin/lin64"
2  source $bindir/radiant_env
3
4  export proj_name="clnx_proj_tcl"
5  export proj_dir="/home/jmercado/my_designs/${proj_name}"
6
7  synthesis -a "LIFCL" -p "LIFCL-40" -t "CABGA256" -sp "8_High-Performance_1.0V" -path "${proj_dir}" \
8  -ver "${proj_dir}/source/count_attr.v" "${proj_dir}/source/my_osc/rtl/my_osc.v" -top "count" \
9  -output_hdl "${proj_name}_syn.vm"
10
11 postsyn -a "LIFCL" -p "LIFCL-40" -t "CABGA256" -sp "8_High-Performance_1.0V" -oc "commercial" -w \
12 -top -o "${proj_name}_postsyn.udb" "${proj_name}_syn.vm"
13
14 map -i "${proj_name}_postsyn.udb" -o "${proj_name}_map.udb"
15
16 par -w "${proj_name}_map.udb" "${proj_name}_par.udb"
17
18 bitgen -w "${proj_name}_par.udb"
```

Figure 5.2. Lattice Radiant Windows Batch Mode Script Example

The following shows the description of the FPGA build flow batch script in Windows:

- Line 1 and Line 2
  - These two commands are required for any Lattice Radiant or Diamond FPGA batch script in Linux.
  - These commands set an environment variable that indicates where the active Radiant installation is located and then sources a setup script to finish setting up the command line environment.
- Line 4 and Line 5
  - Basic project related parameter definitions used to simplify the script and improve its readability.
- Line 7 to Line 9
  - Command to invoke synthesis with LSE.
  - The output of synthesis is the *clnx_proj_tcl.vm* file.
- Line 11 and Line 12
  - Post-synthesis command used to convert the structural Verilog file from synthesis to UDB format.
  - The output of post-synthesis is the *clnx_proj_tcl_syn.vm* file.
- Line 14
  - Command used to run map for the active design using the UDB generated from post-synthesis.
  - The output of this command is the *clnx_proj_tcl_map.udb* file.
- Line 16
  - Command used to run place and route on the UDB file generated by MAP.
  - The output of this command is the *clnx_proj_tcl_par.udb* UDB file.
- Line 18
  - Command used to generate a bitstream based off the UDB generated by the place and route engine.
  - The output of this command is the *clnx_proj_tcl_par.bit* bitstream file.

5.1.3. FPGA Build Flow Batch Script (Linux)

```
1  set PATH=C:/lscc/radiant/2023.1/bin/nt64;C:/lscc/radiant/2023.1/ispfpga/bin/nt64
2  set FOUNDRY=C:/lscc/radiant/2023.1/ispfpga
3
4  set proj_name="crosslinknx_soc"
5
6  synthesis -a "LIFCL" -p "LIFCL-40" -t "CABGA256" -sp "8_High-Performance_1.0V" -path "${proj_dir}" \
7  -ver "${proj_dir}/source/count_attr.v" "${proj_dir}/source/my_osc/rtl/my_osc.v" -top "count" \
8  -output_hdl "${proj_name}_syn.vm"
9
10 postsyn -a "LIFCL" -p "LIFCL-40" -t "CABGA256" -sp "8_High-Performance_1.0V" -oc "commercial" \
11 -w -top -o "${proj_name}_postsyn.udb" "${proj_name}_syn.vm"
12
13 map -i "${proj_name}_postsyn.udb" -o "${proj_name}_map.udb"
14
15 par -w "${proj_name}_map.udb" "${proj_name}_par.udb"
16
17 bitgen -w "${proj_name}_par.udb"
```

Figure 5.3. Lattice Radiant Linux Batch Mode Example

The following shows the description of the FPGA build flow batch script in Linux:

- Line 1 and Line 2
  - These two commands are required for any Lattice Radiant or Diamond FPGA batch script in Windows.
  - These commands set the two required environment variables to configure the Windows command line interpreter to be able to recognize Lattice tool commands.
- Line 4
  - Basic project related parameter definitions used to simplify the script and improve its readability.
- Line 6 to Line 8
  - Command to invoke synthesis with LSE.
  - The output of synthesis is the *clnx_proj_tcl.vm* file.
- Line 10 and Line 11
  - Post-synthesis command used to convert the structural Verilog file from synthesis to UDB format.
  - The output of post-synthesis is the *clnx_proj_tcl_syn.vm* file.
- Line 13
  - Command used to run map for the active design using the UDB generated from post-synthesis.
  - The output of this command is the *clnx_proj_tcl_map.udb* file.
- Line 15
  - Command used to run place and route on the UDB file generated by MAP.
  - The output of this command is the *clnx_proj_tcl_par.udb* UDB file.
- Line 17
  - Command used to generate a bitstream based off the UDB generated by the place and route engine.
  - The output of this command is the *clnx_proj_tcl_par.bit* bitstream file.

5.2. Lattice Diamond Build Flow

5.2.1. TCL Command Build Script

```
1  set proj_dir "/home/lattice/Desktop/xo2_proj"
2  set proj_name "xo2_proj"
3
4  cd $proj_dir
5
6  prj_project open "${proj_dir}/${proj_name}.ldf"
7
8  prj_run Synthesis -impl impl1
9  prj_run Translate -impl impl1
10 prj_run Map -impl impl1
11 prj_run PAR -impl impl1
12 prj_run Export -impl impl1
```

Figure 5.4. Lattice Diamond TCL Build Script Example

The following shows the description of the TCL command build script:

- Line 1 and Line 2
  - Basic project related parameter definitions used to simplify the script and improve its readability.
- Line 4
  - Not required, but the *cd* command is used to change the current working directory to the project's directory.
- Line 8 to Line 12
  - Multiple *prj_run* commands are used to run each stage of the project development flow.
  - Alternatively, only the *prj_run Export -impl impl1* command can be used as this automatically runs at any stages before export, which have not been run already.
    - This example script includes individual *prj_run* commands for synthesis, MAP, PAR, and Export. However, this is not required and typically is only done to allow you more granularity in the script's functionality if you only want to run up to a certain process stage.

5.2.2. FPGA Build Flow Batch Script (Linux)

```
1  export bindir="/home/jmercado/lscc/diamond/3.12/bin/lin64"
2  source $bindir/diamond_env
3
4  export proj_name="counter"
5  export proj_dir="/home/jmercado/projects/${proj_name}"
6
7  synpwrap -prj "${proj_dir}/counter_project_impl1_synplify.tcl"
8
9  edif2ngd -l "MachXO3D" -d LAMXO3D-4300HC -path "$proj_dir" \
10 "${proj_dir}/${proj_name}.edi" "${proj_name}.ngo"
11
12 ngdbuild  -a "MachXO3D" -d LAMXO3D-4300HC  \
13 -p "/home/jmercado/lscc/diamond/3.12/ispfpga/se5c00/data"  \
14 -p "$proj_dir" "${proj_name}.ngo" "${proj_name}.ngd"
15
16 map -a "MachXO2" "${proj_name}.ngd" -o "${proj_name}_map.ncd"
17
18 par -w "${proj_name}_map.ncd" "${proj_name}_par.ncd"  \
19 "${proj_name}_map.prf"
20
21 bitgen -w "${proj_name}_par.ncd" "${proj_name}_map.prf"
```

Figure 5.5. Lattice Diamond Linux Batch Mode Example

The following shows the description of the FPGA build flow batch script in Linux:

- Line 1 and Line 2
  - These two commands are required for any Lattice Radiant or Diamond FPGA batch script in Linux.
  - These commands set an environment variable that indicates where the active Diamond installation is located, and then sources a setup script to finish setting up the command line environment.
- Line 3 and Line 4
  - Basic project related parameter definitions used to simplify the script and improve its readability.
- Line 7
  - Command used to run synthesis with Synplify Pro.
  - The output of this command is the *counter.edi* EDI file.
  - This command uses another tool generated TCL script, *counter_project_impl1_synplify.tcl*, to load the project settings for synthesis with Synplify Pro.
    - To easily generate this file, run through synthesis at least once beforehand in the Diamond user interface.
    - This file can be directly reused in a command line batch script flow.
- Line 9 to Line 14
  - *edif2ngd* and *ngdbuild* are used to run the translate stage of the Diamond project flow in order to convert the synthesis output into an NGD file which can be used by Diamond's mapper.
    - The output of edif2ngd is the *counter.ngo* file.
    - The output of ngdbuild is the *counter.ngd* file.
- Line 16
  - Command used to run map for the active design using the NGD file generated from the translate stage.
  - Output of this command is the *counter_map.ncd* file.
- Line 18 to Line 19
  - Command used to run place and route on the NCD file generated by MAP.
  - The output of this command is the *counter_par.ncd* NCD file.

- Line 21
  - Command used to generate a bitstream based off the UDB generated by the place and route engine.
  - The output of this command is the *counter_par.bit* bitstream file.

Appendix A. Lattice FPGA Build Scripts for Linux and Windows

```
set proj_name “clnx_proj_tcl”
set proj_dir “/home/jmercado/my_designs/${proj_name}”
set proj_rdf “${proj_dir}/${proj_name}.rdf”

set device “LIFCL-40-8BG256C”
set speed “8_High-Performance_1.0V”

if { [file exists $proj_dir] == 0} {
  file mkdir $proj_dir
  cd $proj_dir

  prj_create -name $proj_name -impl “impl_1” -dev $device -performance $speed -synthesis
“synplify"

  prj_add_source “/home/jmercado/rtl/count_attr.v”
  prj_add_source “/home/jmercado/rtl/my_osc/my_osc.ipx”

} else {
  prj_open $proj_rdf
}

puts “Starting to build the design”
prj_run Export -impl impl_1 -forceAll
prj_save
```

```
export bindir=”/home/jmercado/lscc/radiant/2023.1/bin/lin64”
source $bindir/radiant_env

export proj_name=”clnx_proj_tcl”
export proj_dir=”/home/jmercado/my_designs/${proj_name}”

synthesis -a “LIFCL” -p “LIFCL-40” -t “CABGA256” -sp “8_High-Performance_1.0V” -path
“${proj_dir} \
-ver “${proj_dir}/source/count_attr.v” “${proj_dir}/source/my_osc/rtl/my_osc.v” -top
“count” \
-output_hdl “${proj_name}_syn.vm”

postsyn -a “LIFCL” -p “LIFCL-40” -t “CABGA256” -sp “8_High-Performance_1.0V” -oc
“commercial” -w \
-top -o “${proj_name}_postsyn.udb” “${proj_name}_syn.vm”

map -i “${proj_name}_postsyn.udb” -o “${proj_name}_map.udb”

par -w “${proj_name}_map.udb” “${proj_name}_par.udb”

bitgen -w “${proj_name}_par.udb”
```

```
set PATH=C:/lscc/radiant/2023.1/bin/nt64;C:/lscc/radiant/2023.1/ispfpga/bin/nt64
set FOUNDRY=C:/lscc/radiant/2023.1/ispfpga

set proj_name=”crosslinknx_soc”

synthesis -a “LIFCL” -p “LIFCL-40” -t “CABGA256” -sp “8_High-Performance_1.0V” -path
“${proj_dir} \
-ver “${proj_dir}/source/count_attr.v” “${proj_dir}/source/my_osc/rtl/my_osc.v” -top
“count” \
-output_hdl “${proj_name}_syn.vm”

postsyn -a “LIFCL” -p “LIFCL-40” -t “CABGA256” -sp “8_High-Performance_1.0V” -oc
“commercial” -w \
-top -o “${proj_name}_postsyn.udb” “${proj_name}_syn.vm”

map -i “${proj_name}_postsyn.udb” -o “${proj_name}_map.udb”

par -w “${proj_name}_map.udb” “${proj_name}_par.udb”

bitgen -w “${proj_name}_par.udb”
```

```
set proj_dir “/home/lattice/Desktop/xo2_proj”
set proj_name “xo2_proj”

cd $proj_dir

prj_project open “${proj_dir}/${proj_name}.ldf”

prj_run Synthesis -impl impl1
prj_run Translate -impl impl1
prj_run MAP -impl impl1
prj_run PAR -impl impl1
prj_run Export -impl impl1
```

```
export bindir=”/home/jmercado/lscc/diamond/3.12/bin/lin64”
source $bindir/diamond_env

export proj_name=”counter”
export proj_dir=”/home/jmercado/projects/${counter}”

synpwrap -prj “${proj_dir}/counter_project_impl1_synplify.tcl”

edif2ngd -l “MachXO3D” -d LAMXO3D-4300HC -path “$proj_dir” \
“${proj_dir}/${proj_name}.edi” “${proj_name}.ngo”

ngdbuild -a “MachXO3D” -d LAMXO3D-4300HC \
-p “/home/jmercado/lscc/diamond/3.12/ispfpga/se5c00/data” \
-p “$proj_dir” “${proj_name}.ngo” “${proj_name}.ngd”

map -a “MachXO2” “${proj_name}.ngd” -o “${proj_name}_map.ncd”

par -w “${proj_name}_map.ncd” “${proj_name}_par.ncd” \
“${proj_name}_map.prf”

bitgen -w “${proj_name}_par.ncd” “${proj_name}_map.prf”
```

References

For more information, refer to:

- Lattice Radiant FPGA design software
- Lattice Diamond FPGA design software
- Lattice Propel FPGA design software
- Lattice Insights for Lattice Semiconductor training courses and learning plans

Technical Support Assistance

Submit a technical support case through www.latticesemi.com/techsupport.

For frequently asked questions, refer to the Lattice Answer Database at www.latticesemi.com/Support/AnswerDatabase.

Revision History

Revision 1.0, October 2023

Section	Change Summary
All	Initial release.

www.latticesemi.com