Synthesis User Guide (UG018)

All Achronix Devices

Achronix
Data Acceleration

Copyrights, Trademarks and Disclaimers

Notice of Disclaimer

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter 1 : Overview

This user guide describes how to synthesize an RTL design to generate a synthesized gate-level netlist for implementation in an Achronix device. Suggested optimization techniques are also included.

A high-level overview of the Achronix design flow is shown in figure below.

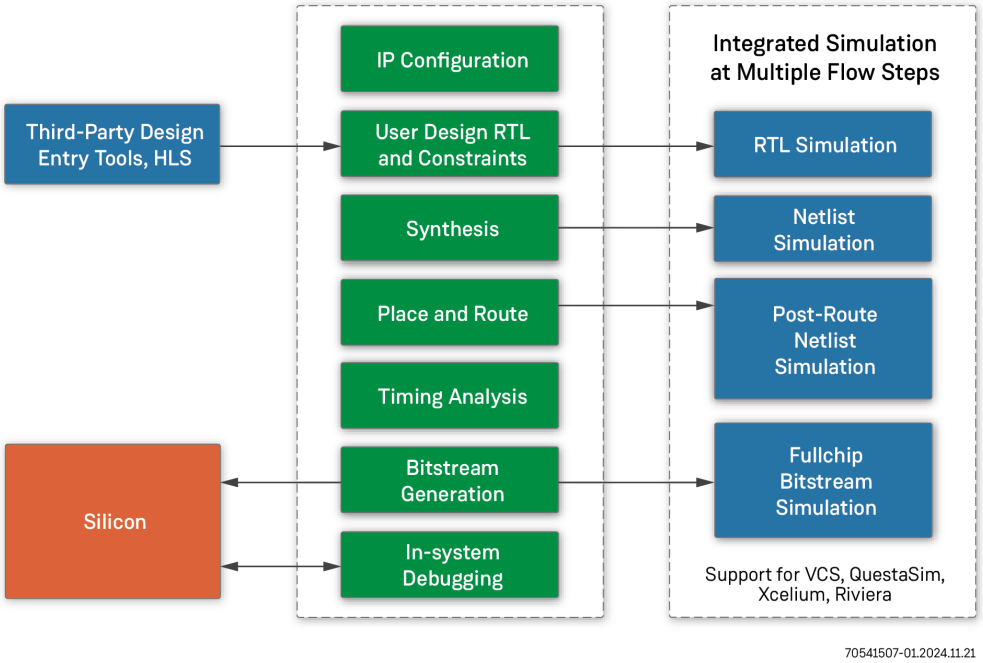


Figure 1 • Achronix Design Flow

Synthesis Flows

There are three main synthesis flows supported by the ACE tools suite:

- **ACE-Driven Integrated Synthesis** (page 4), where ACE owns and manages the synthesis project definition, and synthesis is run via the built-in ACE flow step.
- **Synplify-Pro-Driven Integrated Synthesis** (page 14), where Synplify Pro owns and manages the synthesis project definition, and synthesis is run via the built-in ACE flow step.
- **Stand-Alone Synthesis in Synplify Pro** (page 23), where Synplify Pro is run completely outside of ACE to generate the synthesized gate-level netlist, which is then added to the ACE project. The built-in ACE synthesis flow step is not run in this case.

Synplify Pro does not support batch mode with node-locked licenses In order to run the integrated synthesis flow steps, a floating license setup is required for Synplify Pro since ACE calls Synplify Pro in batch mode.

Chapter 2 : ACE-Driven Integrated Synthesis

As of ACE version 10.0, synthesis is now a fully integrated flow step in ACE. For designers, the simplest and easiest synthesis flow to use is the ACE-driven integrated synthesis flow. In this flow, end users do not need to leave ACE to configure or run synthesis. Users can stay in ACE and manage all aspects of design synthesis, including synthesis project setup, synthesis options configuration, running synthesis to compile the design, error reporting and log viewing, and report viewing.

In this scenario, ACE is the master of the Synplify Pro project and runs Synplify Pro from within the ACE **Run Synthesis** flow step.

> **Caution!**
>
> Users should not open the ACE-generated Synplify project file and make changes in Synplify Pro in this flow, because ACE will re-generate the Synplify project file from the ACE project file settings each time synthesis is run, and any changes made in Synplify Pro will be lost. To manage a Synplify project file using Synplify Pro, refer to section. **Synplify-Pro-Driven Integrated Synthesis** (page 14).

Synthesis Project Setup in ACE

To simplify the download, install, and licensing process, Synplify Pro is now included in the base ACE install package. Users no longer need to find the compatible version of Synplify Pro, and download it separately from ACE.

The ACE installer on Windows, and the ACE installer script on Linux have been updated to automatically install Synplify Pro as part of the ACE installation. Users no longer need to install Synplify Pro separately. In addition, some of the ACE and Synplify Pro license installation and configuration is now automated in the ACE install process. See the *ACE Installation and Licensing Guide* (UG002)[1] for more details.

As of ACE 10.2 users no longer need to set the $ACX_SYNPLIFY_TOOL_PATH environment variable. ACE now searches for the Synplify Pro installation according to the following order of precedence:

1. If ACX_SYNPLIFY_TOOL_PATH is set, use it, otherwise;
2. Check if $SYNPLIFY_HOME is set, and search for it there, otherwise;
3. Check to see if synplify_pro is available inside the ACE install at <ace_install>/Synplify/bin/synplify_pro(.exe on Windows), otherwise;
4. Check to see if synplify_pro is available on the $PATH env variable, otherwise;
5. Error out

Now launch ACE to get started.

Create an ACE Project

In the Projects View, click the ( ) **Create Project** toolbar button. Follow these steps to create the project:

1. In the Create Project Dialog, enter (or browse to) the desired path to the ACE project top-level directory in the Project Directory field.

1 https://www.achronix.com/documentation/ace-installation-and-licensing-guide-ug002

2. Enter the desired ACE project name in the **Project Name** field and click **Finish**.

The new project will now appear in the Projects view. See "Creating Projects" or "Working with Projects and Implementations" in the *ACE Users Guide* (UG070)[2] for more details.

Add the Design Files and Set Project Options

In the Projects view, click the project to select it. Follow these steps to add the design source files for synthesis and place and route:

1. Click the ( ) **Add Source Files** toolbar button and select **Add RTL Files**.
2. In the Add RTL Files dialog, browse to the source RTL directory and select all of the RTL files by holding down the **CTRL** key and clicking each file name.
3. Click the **Open** button to add the RTL files to the project. Repeat this process as needed until all the RTL files are added to the project.
4. Click the ( ) **Add Source Files** toolbar button and select **Add Synthesis Constraint Files**.
5. In the "Add Synthesis Constraint Files" dialog, browse to the constraints directory and select all of the synthesis constraints files by holding down the **CTRL** key and clicking each file name.
6. Click the **Open** button to add the synthesis constraint files to the project. Repeat this process as needed until all the synthesis constraints files are added to the project.
7. Click the ( ) **Add Source Files** toolbar button and select **Add Place and Route Constraint Files**.

   Notes

   - This and the following steps to add place-and-route constraint Files are optional and are not required for running synthesis. These instructions only apply to continue running the flow through Place and Route.
   - If a previously generated a synthesized gate level netlist exists and has been added it as a place-and-route netlist file in the ACE project, remove the netlist from the ACE project prior to running the integrated synthesis flow in ACE. ACE will automatically add the generated synthesized netlist to the ACE project as part of the Run Synthesis flow step.

8. In the "Add Place and Route Constraint Files" dialog, browse to the place-and-route constraints directory and select all of the files by holding down the **CTRL** key and clicking each file name.
9. Click the **Open** button to add the place-and-route constraint files to the project. Repeat this process as needed until all the place-and-route constraint files are added to the project.

For instructions on adding simulation files to the ACE project, please see the *Simulation User Guide* (UG072)[3] or the "ACE Quickstart Tutorial" in the *ACE Users Guide* (UG070)[4].

2 https://www.achronix.com/documentation/ace-user-guide-ug070
3 https://www.achronix.com/documentation/simulation-user-guide-ug072
4 https://www.achronix.com/documentation/ace-user-guide-ug070

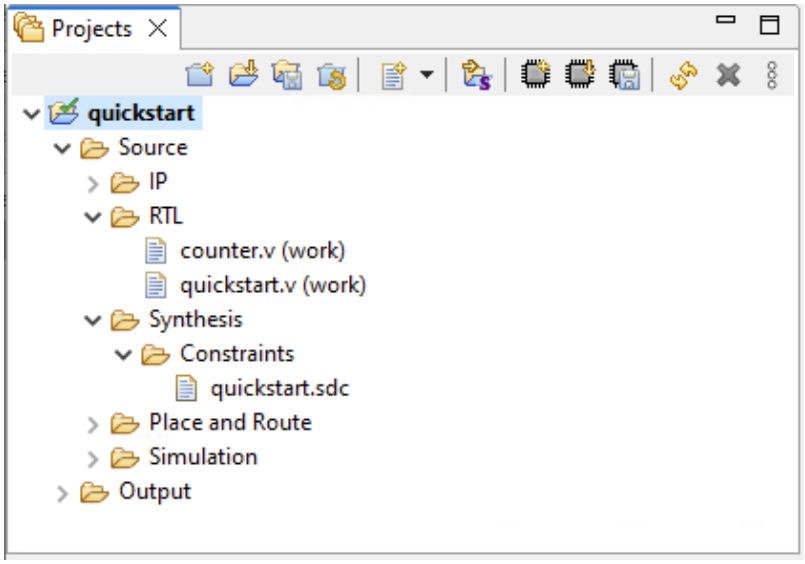


Figure 2 • Synthesis Project Source Files

In the Options View, follow these steps to configure the project options:

1. Expand the "Project Options" section and select the target device for the design.
2. In the Project Options section, scroll down and enter the semicolon-separated list for the HDL include path. For example:

```
D:/test_dir/src/rtl;D:/test_dir/src/tb
```

> **Notes**
>
> - The HDL include path applies to both synthesis and simulation.
> - The include path does not need to be added to the ACE libraries in `<ace_install>/libraries`. The Run Synthesis flow step will automatically add this to the include path in the generated Synplify Pro project file.

3. In the "Project Options" section, scroll down and enter the space-separated list of any HDL define symbols needed for the design in "HDL Defines". For example:

```
ADDR_WIDTH=16 DATA_WIDTH=8
```

> **Note**
>
> The HDL defines applies to both synthesis and simulation.

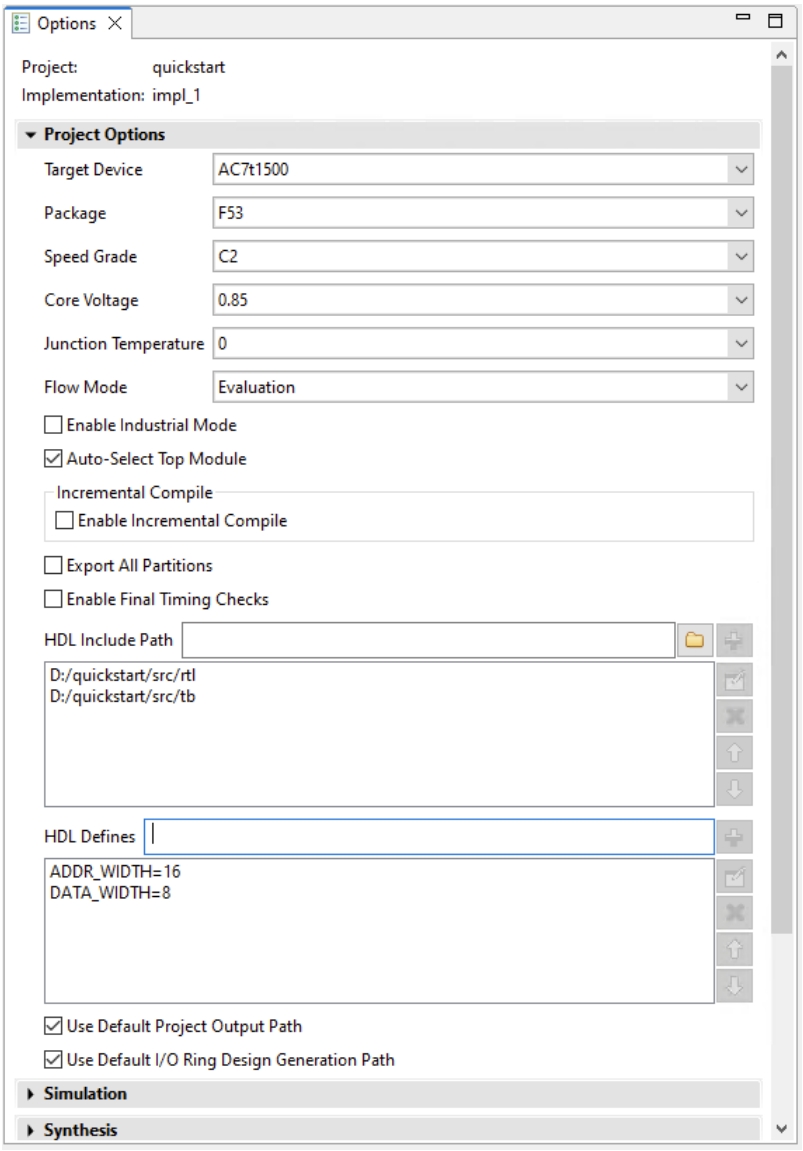


Figure 3 • Synthesis Project Options

Synthesis Options Configuration

Once the source files are added and the project options are set, the synthesis implementation options must also be set. In "Options View", scroll down to the "Synthesis" section and click to expand the section to show the synthesis implementation options. Ensure that the option the **ACE-Driven Synthesis** is checked.

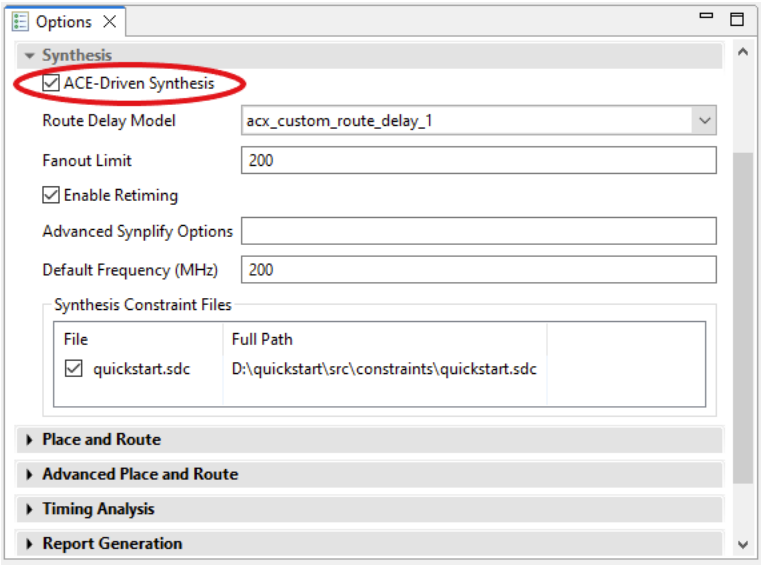


Figure 4 • Synthesis Implementation Options

Caution!

In order to run the ACE-driven integrated synthesis flow, the **ACE-Driven Synthesis** option must be checked (`syn_ace_driven_synthesis` project option is set to 1). If it is not checked, then project is using the Synplify-Pro-Driven Integrated Synthesis (page 14) flow instead.

Configure the remaining implementation options as needed for the design. Any Synplify Pro options that are not directly exposed in the ACE GUI can be set using the "Advanced Synplify Options" field. Simply enter a TCL formatted list of option-value pairs, for example:

```
{{option1 value1} {option2 value2}}
```

Synthesis implementation options can be explored automatically to find the best options for the design by using the ACE multiprocess feature as described in Synthesis Integration with Multiprocess Option Exploration (page 25).

Running Synthesis to Compile the Design

To run synthesis from within ACE, ensure that the **Run Synthesis** flow step is enabled (the checkbox is checked):

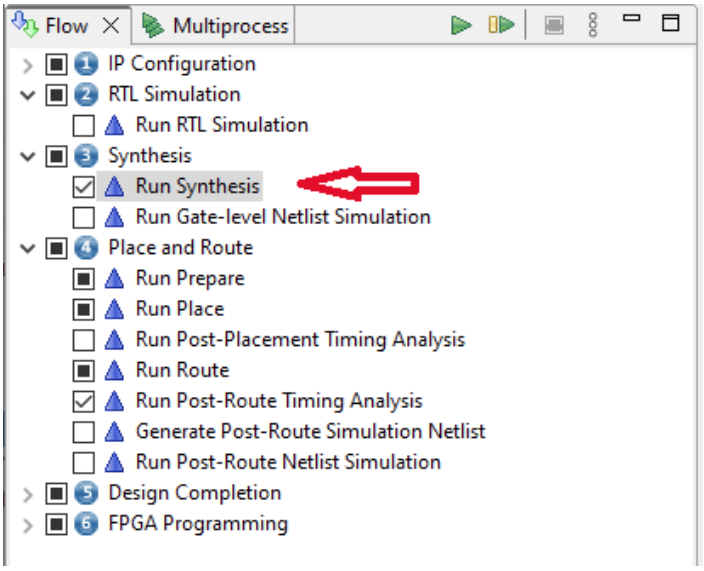


Figure 5 • Enabling the Synthesis Flow Step

To run just the Run Synthesis flow step, perform one of the following:

- Double-click on the Run Synthesis flow step
- Right-click on the Run Synthesis flow step and select **Run Selected Flow Step**
- Call `run -step run_synthesis` from the TCL console

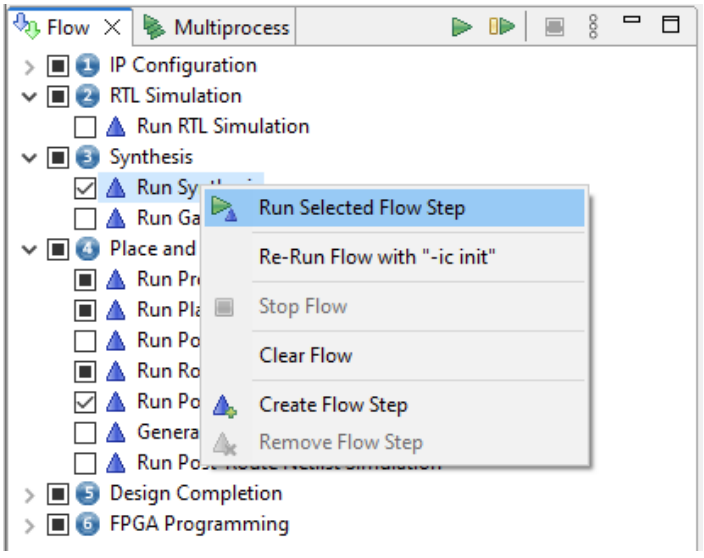


Figure 6 • Running the Run Synthesis Flow Step

The Run Synthesis flow step can be run from within the context of the overall flow by:

- Clicking on the **Run Flow** toolbar button to run the entire flow
- Call `run` from the TCL console to run the entire flow

If a subsequent flow step is run, ACE will automatically run all incomplete prerequisite and enabled flow steps between the selected flow step and the last completed flow step. For example, double-clicking on the **Run Post-Route Timing Analysis** flow step and none of the previous steps are complete, ACE will automatically start running the enabled flow steps in order from the beginning of the flow, including Run Synthesis if it is enabled.

The Run Synthesis flow step runs synthesis using the configuration set in the ACE project options. In this flow ACE is the master of the synthesis project (the `syn_ace_driven_synthesis` project option is set to 1).

The source synthesis project file will be automatically generated from the ACE project settings and managed by ACE in the Project → Output → (impl) → syn directory.

All output from the underlying synthesis tool is streamed to the ACE TCL console and ACE log file. If synthesis fails, ACE will catch the error and will mark the Run Synthesis flow step state as an error with a red X and stop the flow from running any further. If synthesis succeeds, ACE will mark the Run Synthesis flow step as complete with a green check-mark icon.

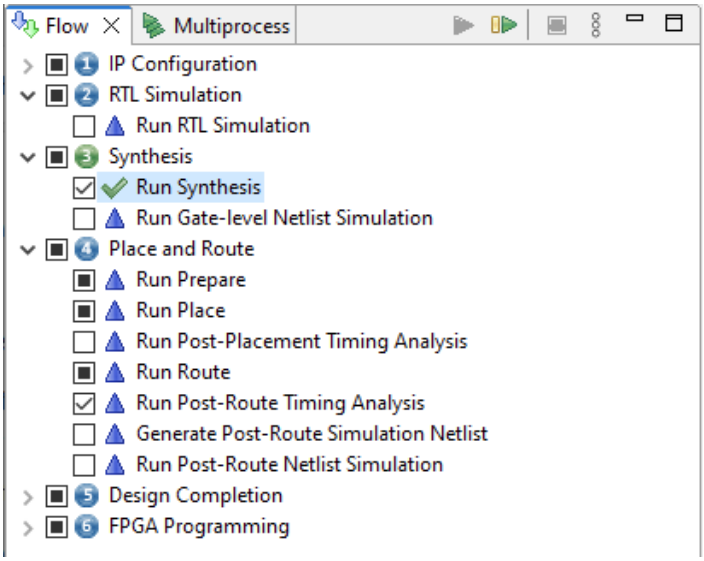


Figure 7 • Synthesis Completed Successfully

Synthesis Reports and Messages

Once synthesis completes, ACE will automatically open any relevant synthesis reports and log files in the ACE GUI Editor Area.

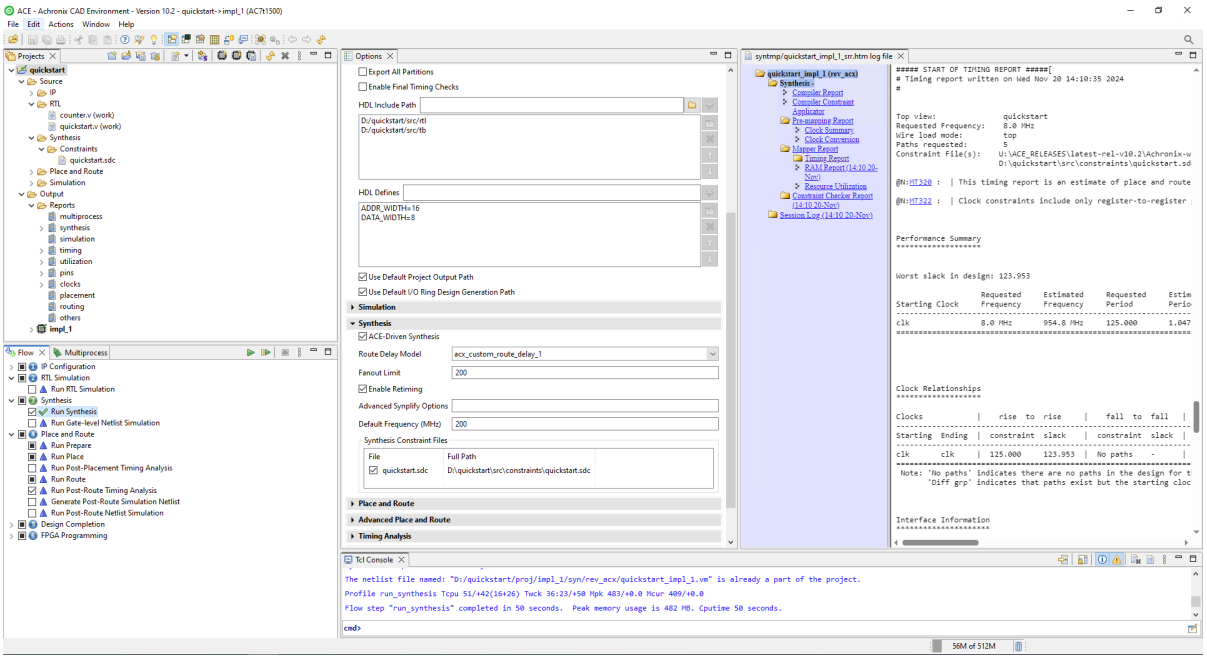


Figure 8 • Synthesis Reports and Messages

These reports can be found later on in the ACE Projects View under the Project → Output → Reports → synthesis virtual folder. ACE automatically organizes all reports in a central location for easy access.

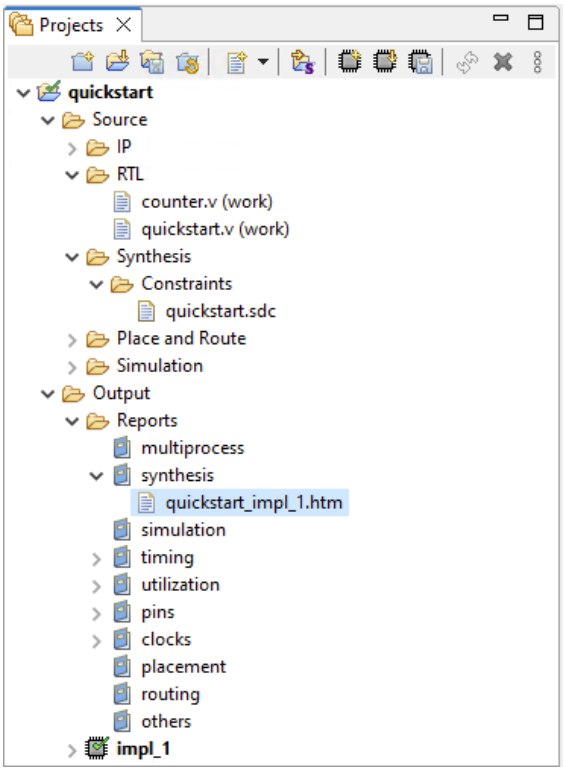


Figure 9 • ACE Project Reports Virtual Folders

Opening Synplify Project File in Synplify Pro

As of ACE 10.1, ACE allows the user to open the generated Synplify Pro project file under Project→Output→(impl)→syn directory in Synplify Pro GUI. To open this file in Synplify Pro, in the Projects View tree, right-click on the ACE-generated Synplify Pro project file, i.e., **Output → <active_impl> → syn → <project_name_impl_name>.prj** and select **Open Project in Synplify**. For more details, refer to the section Opening Synplify in ACE in the *ACE Users Guide* (UG070)[5].

Caution!

Users can use this feature to view project file settings and schematics in Synplify Pro GUI. The ACE-generated Synplify Pro project file should not be updated in this flow because ACE will re-generate the Synplify Pro project file from the ACE project file settings each time synthesis is run, and any changes made in Synplify Pro will be lost. To manage a Synplify Pro project file using Synplify Pro, refer to section. Synplify-Pro-Driven Integrated Synthesis (page 14).

5 https://www.achronix.com/documentation/ace-user-guide-ug070

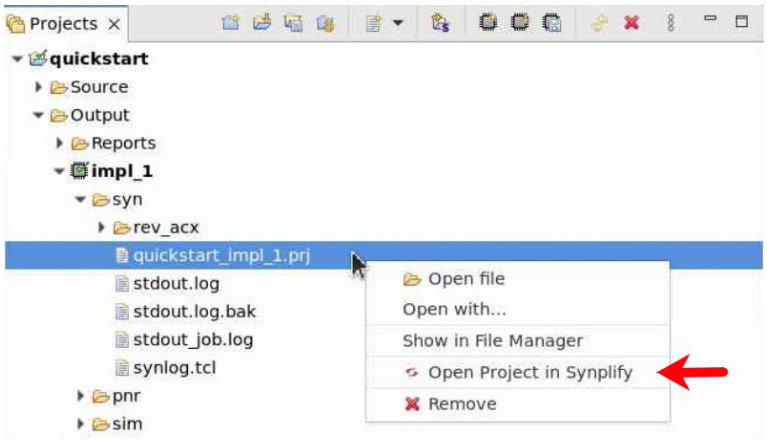


Figure 10 • Open Project in Synplify Pro for ACE-Driven Integrated Synthesis

Chapter 3 : Synplify-Pro-Driven Integrated Synthesis

As of ACE 10.0, synthesis is now a fully integrated flow step in ACE. In this hybrid flow, end users configure and manage their synthesis project in Synplify Pro and run synthesis from inside ACE. This capability enables users who are comfortable using the Synplify GUI to take advantage of the integrated Run Synthesis flow step in ACE and the automated synthesis implementation option exploration offered in the ACE multiprocess feature.

In this scenario, Synplify Pro is the master of the Synplify project file, and ACE calls Synplify Pro from within the ACE Run Synthesis flow step. In this flow, users must disable (uncheck) the **ACE-Driven Synthesis** synthesis implementation option in ACE (`syn_ace_driven_synthesis` project option is set to 0), and set the **Project Override Path** option to point to the source Synplify project file being managed in Synplify Pro.

When the Run Synthesis flow step is run, ACE reads in the Project Override Path project file, overrides a subset of the implementation options (to enable multiprocess), and generates a local modified copy of the project file to run from within ACE. Users *should not* open the ACE-generated Synplify project file and make changes in Synplify Pro in this flow because ACE will re-generate the Synplify project file from the ACE project file settings each time synthesis is run, and any changes made in Synplify Pro will be lost. To manage a Synplify project file using Synplify Pro, open the Project Override Path project file in Synplify instead. For more details, For more details, refer to the section Opening Synplify in ACE in the ***ACE Users Guide*** **(UG070)**[6].

Configuring the Synthesis Project in Synplify Pro

The first step is to create a new synthesis project and configure the synthesis options as documented in the section, "**Managing Projects in Synplify Pro** (page 27)".

Synthesis Project Setup in ACE

To simplify the download, install, and licensing process, Synplify Pro is now included in the base ACE install package. Users no longer need to find the compatible version of Synplify Pro, and download it separately from ACE.

The ACE installer on Windows, and the ACE installer script on Linux have been updated to automatically install Synplify Pro as part of the ACE installation. Users no longer need to install Synplify Pro separately. In addition, some of the ACE and Synplify Pro license installation and configuration is now automated in the ACE install process. See the ***ACE Installation and Licensing Guide*** **(UG002)**[7] for more details.

As of ACE 10.2 users no longer need to set the $ACX_SYNPLIFY_TOOL_PATH environment variable. ACE now searches for the Synplify Pro installation according to the following order of precedence:

1. If ACX_SYNPLIFY_TOOL_PATH is set, use it, otherwise;
2. Check if $SYNPLIFY_HOME is set, and search for it there, otherwise;
3. Check to see if synplify_pro is available inside the ACE install at <ace_install>/Synplify/bin/synplify_pro(.exe on Windows), otherwise;
4. Check to see if synplify_pro is available on the $PATH env variable, otherwise;
5. Error out

6 https://www.achronix.com/documentation/ace-user-guide-ug070
7 https://www.achronix.com/documentation/ace-installation-and-licensing-guide-ug002

Now launch ACE to get started.

Create an ACE Project

In the Projects View, click the ( ) **Create Project** toolbar button. Follow these steps to create the project:

1. In the Create Project Dialog, enter (or browse to) the desired path to the ACE project top-level directory in the Project Directory field.
2. Enter the desired ACE project name in the **Project Name** field and click **Finish**.

The new project will now appear in the Projects view. See "Creating Projects" or "Working with Projects and Implementations" in the *ACE Users Guide* (UG070)[8] for more details.

Add the Design Files and Set Project Options

In this flow, RTL files or synthesis constraints files do not need to be added to the ACE project since the synthesis project is outside of ACE. Also, the HDL Include Path, HDL Defines do not need to be configured. These settings will all be automatically imported from the Synplify Pro project file specified in the **Project Override Path** when the Run Synthesis flow step is run.

> **Caution!**
>
> Paths containing environment or TCL variables are not supported as part of the automatic import of settings from the Synplify Pro project file.

In the Projects view, click the project to select it. Follow these steps to add the design source files for synthesis and place and route:

1. Click the ( ) **Add Source Files** toolbar button and select **Add Place and Route Constraint Files**.

   > **Notes**
   >
   > - This and the following steps to add place-and-route constraint Files are optional and are not required for running synthesis. These instructions only apply to continue running the flow through Place and Route.
   > - If a previously generated a synthesized gate level netlist exists and has been added it as a place-and-route netlist file in the ACE project, remove the netlist from the ACE project prior to running the integrated synthesis flow in ACE. ACE will automatically add the generated synthesized netlist to the ACE project as part of the Run Synthesis flow step.

2. In the "Add Place and Route Constraint Files" dialog, browse to the place-and-route constraints directory and select all of the files by holding down the **CTRL** key and clicking each file name.
3. Click the **Open** button to add the place-and-route constraint files to the project. Repeat this process as needed until all the place-and-route constraints files are added to the project.

8 https://www.achronix.com/documentation/ace-user-guide-ug070

For instructions on adding simulation files to the ACE project, see the *Simulation User Guide* (UG072)[9] or the "ACE Quickstart Tutorial" in the *ACE Users Guide* (UG070)[10].

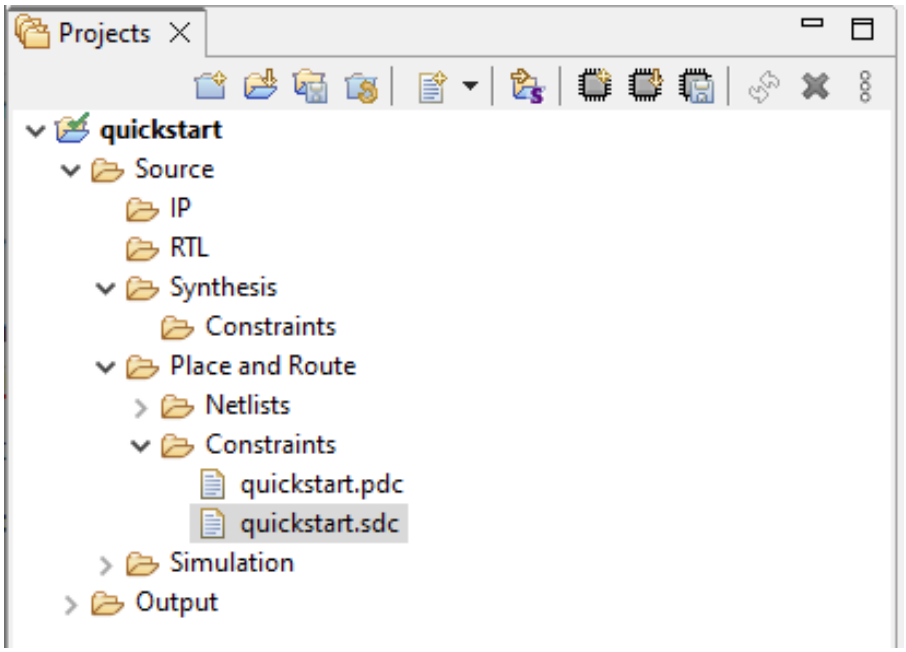


Figure 11 • ACE Project Source Files

In the Options View, follow these steps to configure your project options, expand the "Project Options" section and select the target device for the design.

> **Note**
>
> The HDL Include Path or HDL Defines do not need to be set to run synthesis. These options may need to be configured if running simulation from within ACE. See the *Simulation User Guide* (UG072)[11] for details.

9 https://www.achronix.com/documentation/simulation-user-guide-ug072
10 https://www.achronix.com/documentation/ace-user-guide-ug070
11 https://www.achronix.com/documentation/simulation-user-guide-ug072

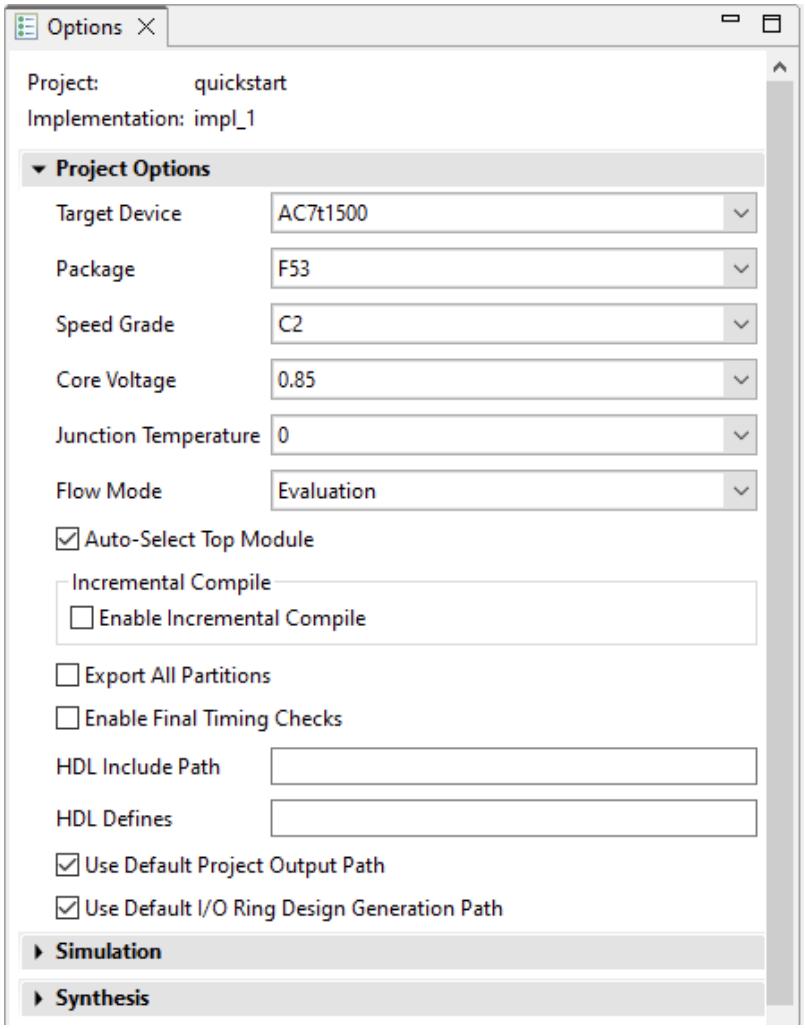


Figure 12 • Synthesis Project Options

Synthesis Options Configuration

Once the source files are added and the project options are set, the synthesis implementation options must also be set. In "Options View", scroll down to the "Synthesis" section and click to expand the section to show the synthesis implementation options. Ensure that the option **ACE-Driven Synthesis** is unchecked.

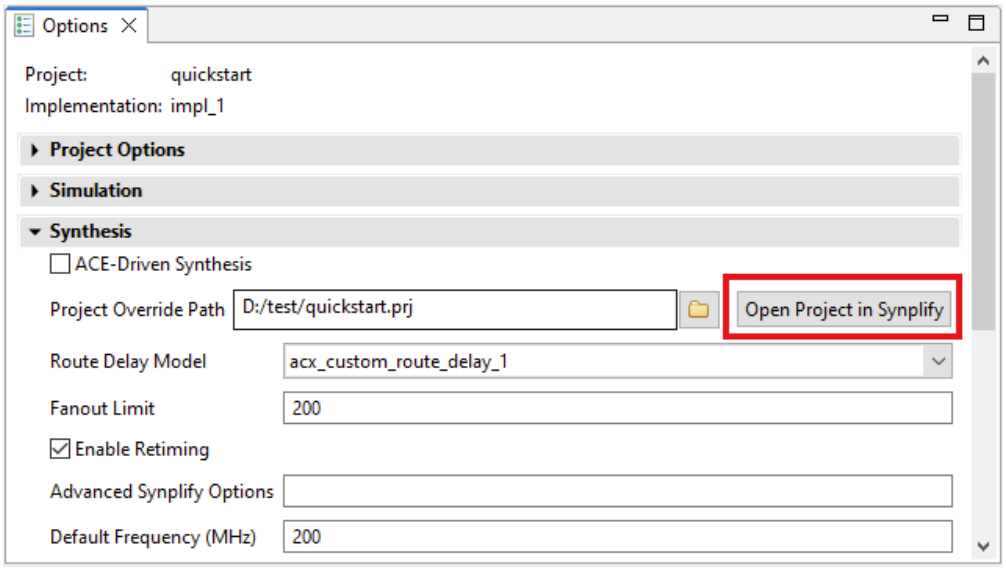


Figure 13 • Opening a Synplify Pro Project File that is Specified as Project Override Path

> **Caution!**
>
> In order to run the Synplify-driven integrated synthesis flow, the **ACE-Driven Synthesis** option must be unchecked (`syn_ace_driven_synthesis` project option is set to 0) and have the "Project Override Path" option set to point to the source Synplify project file. If the **ACE-Driven Synthesis** checkbox is checked, then the project is using the ACE-Driven Integrated Synthesis (page 4) flow instead.

Configure the remaining implementation options as needed for the design.

Synthesis implementation options can be explored automatically to find the best options for the design by using the ACE multiprocess feature as described in Synthesis Integration with Multiprocess Option Exploration (page 25).

Running Synthesis to Compile the Design

To run synthesis from within ACE, ensure that the **Run Synthesis** flow step is enabled (the checkbox is checked):

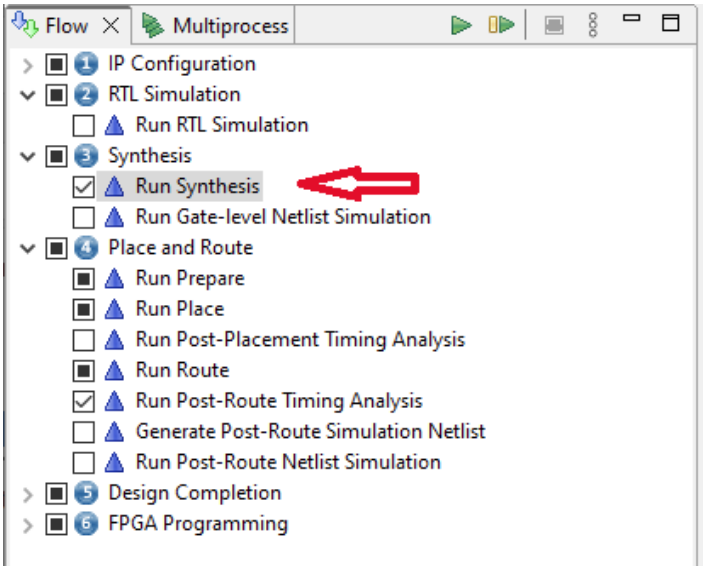


Figure 14 • Enabling the Synthesis Flow Step

To run just the Run Synthesis flow step, perform one of the following:

- Double-click on the Run Synthesis flow step
- Right-click on the Run Synthesis flow step and select **Run Selected Flow Step**
- Call `run -step run_synthesis` from the TCL console

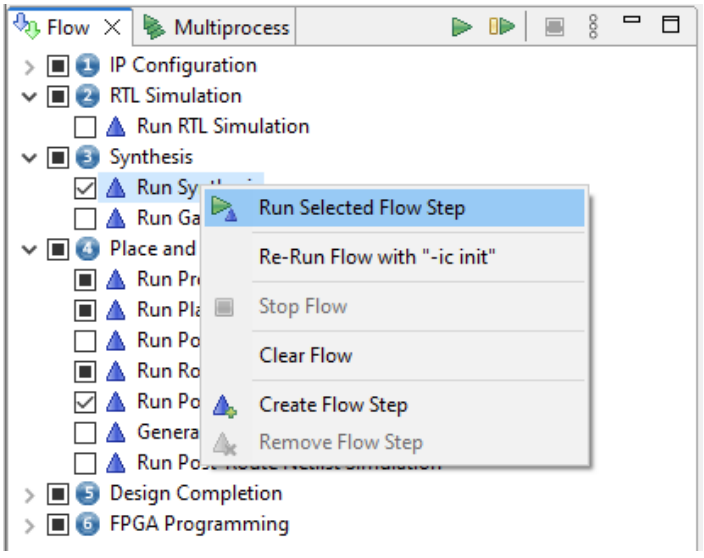


Figure 15 • Running the Run Synthesis Flow Step

The Run Synthesis flow step can be run from within the context of the overall flow by:

- Clicking on the **Run Flow** toolbar button to run the entire flow
- Call `run` from the TCL console to run the entire flow

If a subsequent flow step is run, ACE will automatically run all incomplete prerequisite and enabled flow steps between the selected flow step and the last completed flow step. For example, double-clicking on the **Run Post-Route Timing Analysis** flow step and none of the previous steps are complete, ACE will automatically start running the enabled flow steps in order from the beginning of the flow, including Run Synthesis if it is enabled.

The Run Synthesis flow step reads in the Project Override Path project file, overrides a subset of the implementation options (to enable multiprocess), and generates a local modified copy of the project file to run from within ACE. In this flow Synplify Pro is the master of the synthesis project (the `syn_ace_driven_synthesis` project option is set to 0).

> **Note**
>
> For each implementation, ACE generates a locally modified copy of the synthesis project file in the Project → Output->(impl) → syn directory.

All output from the underlying synthesis tool is streamed to the ACE TCL console and ACE log file. If synthesis fails, ACE will catch the error and will mark the Run Synthesis flow step state as an error with a red X and stop the flow from running any further. If synthesis succeeds, ACE will mark the Run Synthesis flow step as complete with a green check-mark icon.

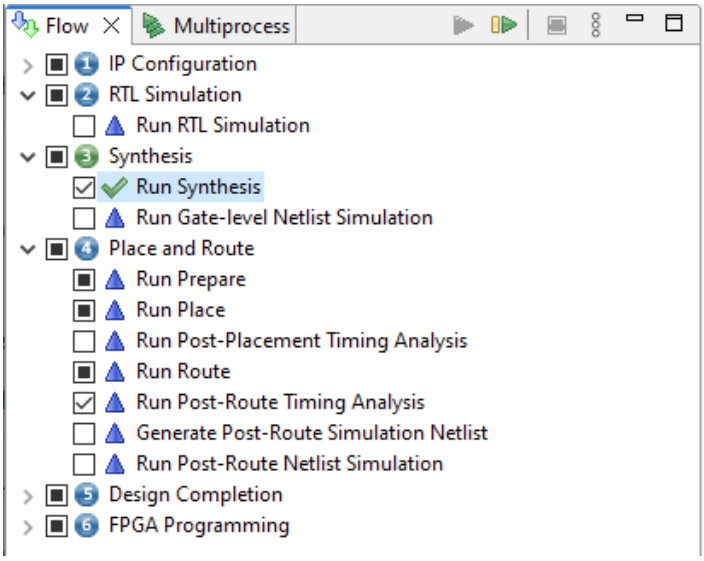


Figure 16 • Synthesis Completed Successfully

Synthesis Reports and Messages

Once synthesis completes, ACE automatically opens any relevant synthesis reports and log files in the ACE GUI Editor Area.

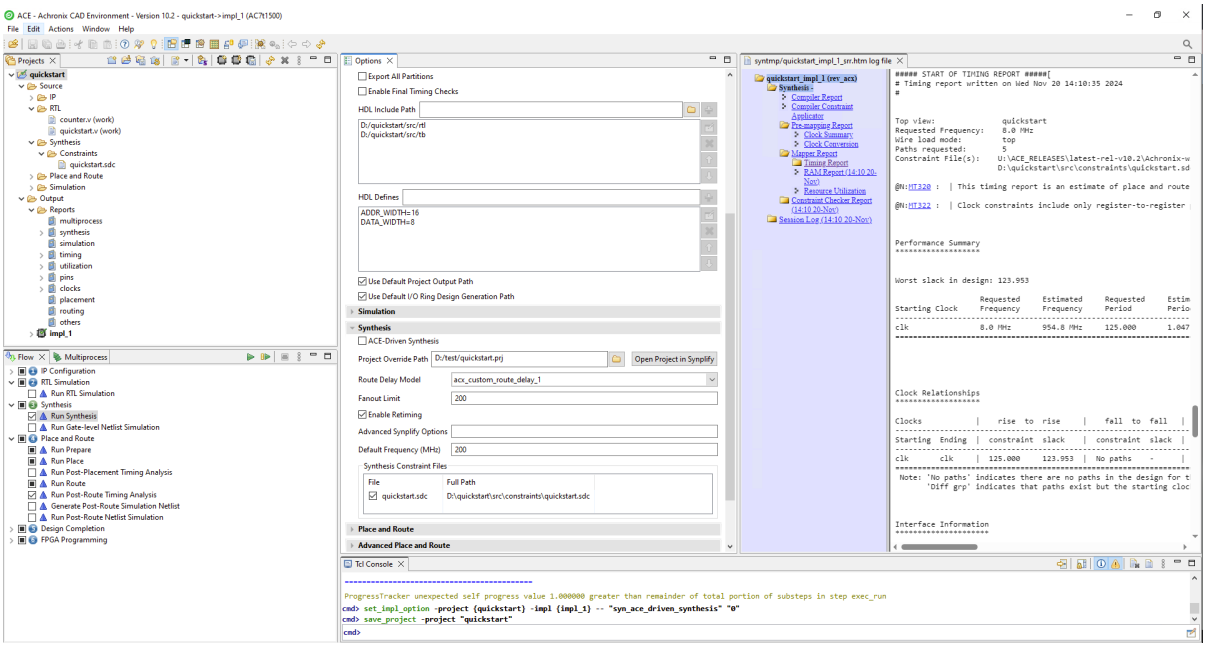


Figure 17 • Synthesis Reports and Messages

These reports can be found later on in the ACE Projects View under the Project → Output → Reports → synthesis virtual folder. ACE automatically organizes all reports in a central location for easy access.

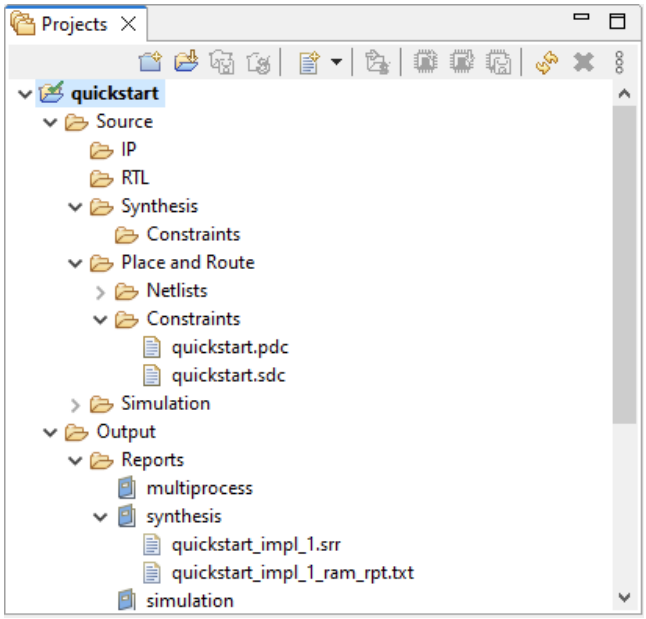


Figure 18 • ACE Project Reports Virtual Folders

Opening Synplify Project File in Synplify Pro

As of ACE 10.1, users can open the path specified as **Project Override Path** (implementation option: `syn_project_override_path`) by clicking the **Open Project in Synplify** button next to the path text field in the Options view. For more details, refer to the section Opening Synplify in ACE in the *ACE Users Guide* (UG070)[12].

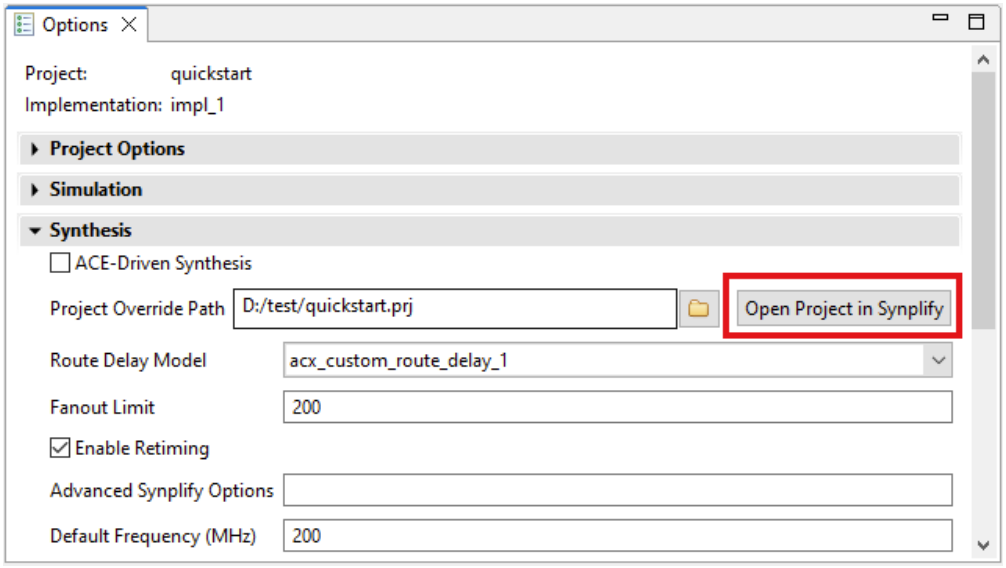


Figure 19 • Open Project in Synplify for Synplify-Pro-Driven Integrated Synthesis

12 https://www.achronix.com/documentation/ace-user-guide-ug070

Chapter 4 : Stand-Alone Synthesis in Synplify Pro

In this flow, synthesis is run outside of ACE in Synplify Pro, and the generated gate-level synthesized netlist is added to the ACE project as a source file. In this flow, the Run Synthesis flow step in ACE is disabled (unchecked).

Configuring the Synthesis Project in Synplify Pro

The first step is to create a new synthesis project and configure the synthesis options as documented in the section, Managing Projects in Synplify Pro (page 27).

Running Synthesis

After selecting all the options according to the users design, click **OK**. The user is returned to the Synplify Pro main window to run the synthesis. From this main window, click **RUN** button to start synthesis.

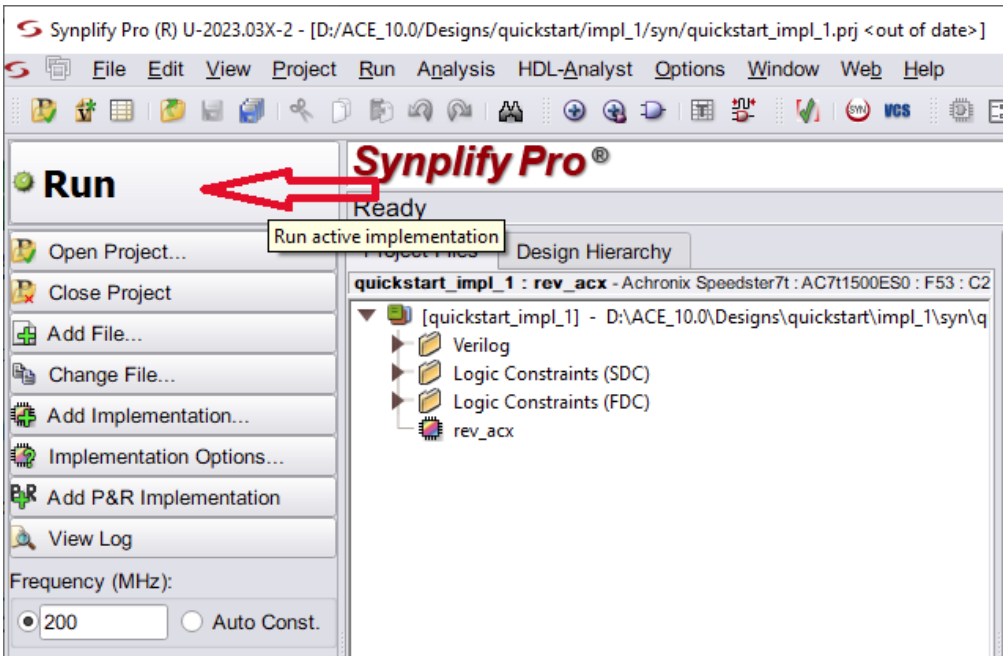


Figure 20 • Running Synthesis in Synplify Pro

Adding the Synthesized Netlist to ACE for Place and Route

Once synthesis has successfully completed, add the generated synthesized netlist to the project in ACE. In the Projects View, Click the ( ) **Add Source Files** toolbar button and select **Add Place and Route Netlist Files**. Browse to the Synplify-generated synthesized netlist file and click **Open**.

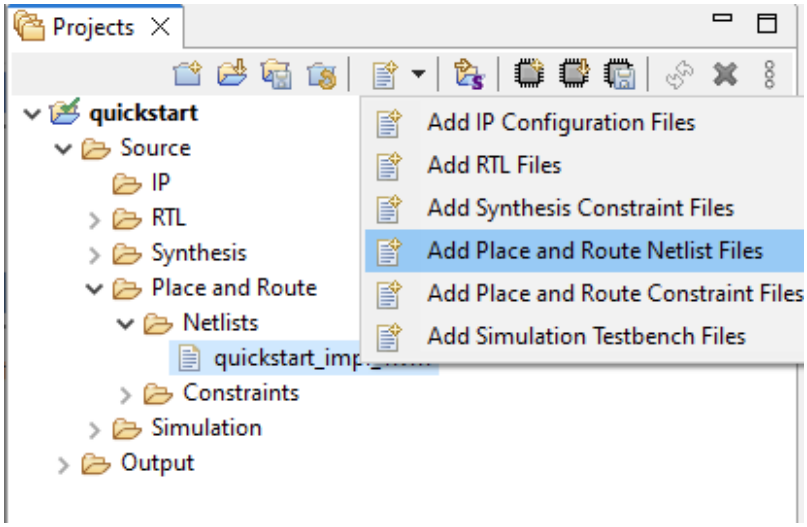


Figure 21 • Adding the Synthesized Netlist to the ACE Project

In this flow, RTL files or synthesis constraints files do not need to be added to the ACE project since the synthesis project outside of ACE. Also, the HDL Include Path, HDL Defines nor any of the synthesis implementation options in ACE need to be configured. only the synthesized gate- level netlist needs to be added to the ACE project.

When running the ACE flow steps, ensure that the option the **ACE-Driven Synthesis** is unchecked; otherwise, ACE will error as the project is not configured to run synthesis. If this happens, simply uncheck the **Run Synthesis flow** step and try running the ACE flow again.

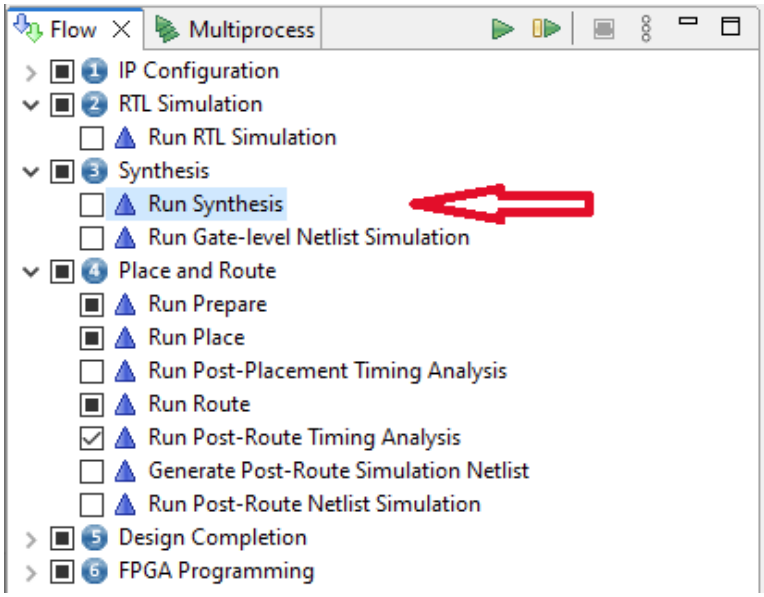


Figure 22 • Run Synthesis Flow Step Disabled

Chapter 5 : Synthesis Integration with Multiprocess Option Exploration

When using the ACE-Driven Integrated Synthesis (page 4) flow or the Synplify-Pro-Driven Integrated Synthesis (page 14) flow, users can take advantage of the automated design option exploration features built in to the ACE multiprocess tool. This tool can generate implementation option sets which sweep over both synthesis and place-and-route options to explore f_{MAX} performance variations.

The following items are required to enable synthesis implementation options exploration:

1. Enable the **Run Synthesis** flow step (checked in the Flow View)

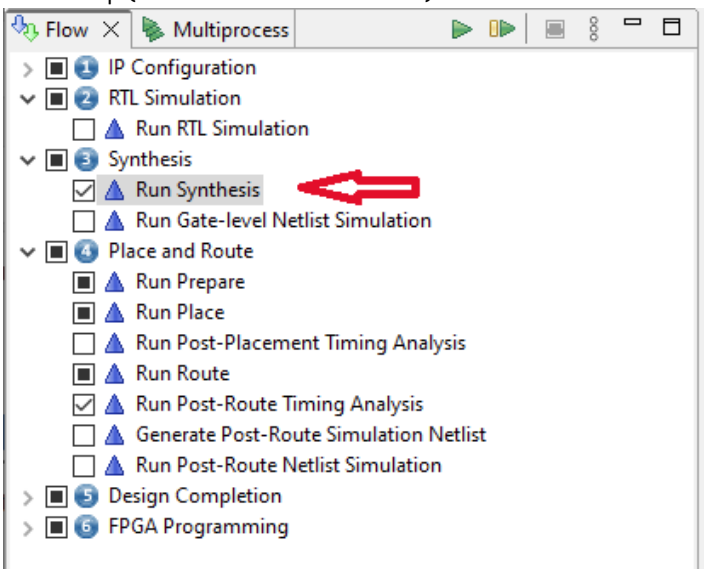


Figure 23 • Run Synthesis Flow Step Enabled

> **Note**
>
> This option is only supported when using ACE-Driven Integrated Synthesis (page 4) or the Synplify-Pro-Driven Integrated Synthesis (page 14) flows.

2. Uncheck the **Exclude Synthesis Option** in the Multiprocess View must be unchecked

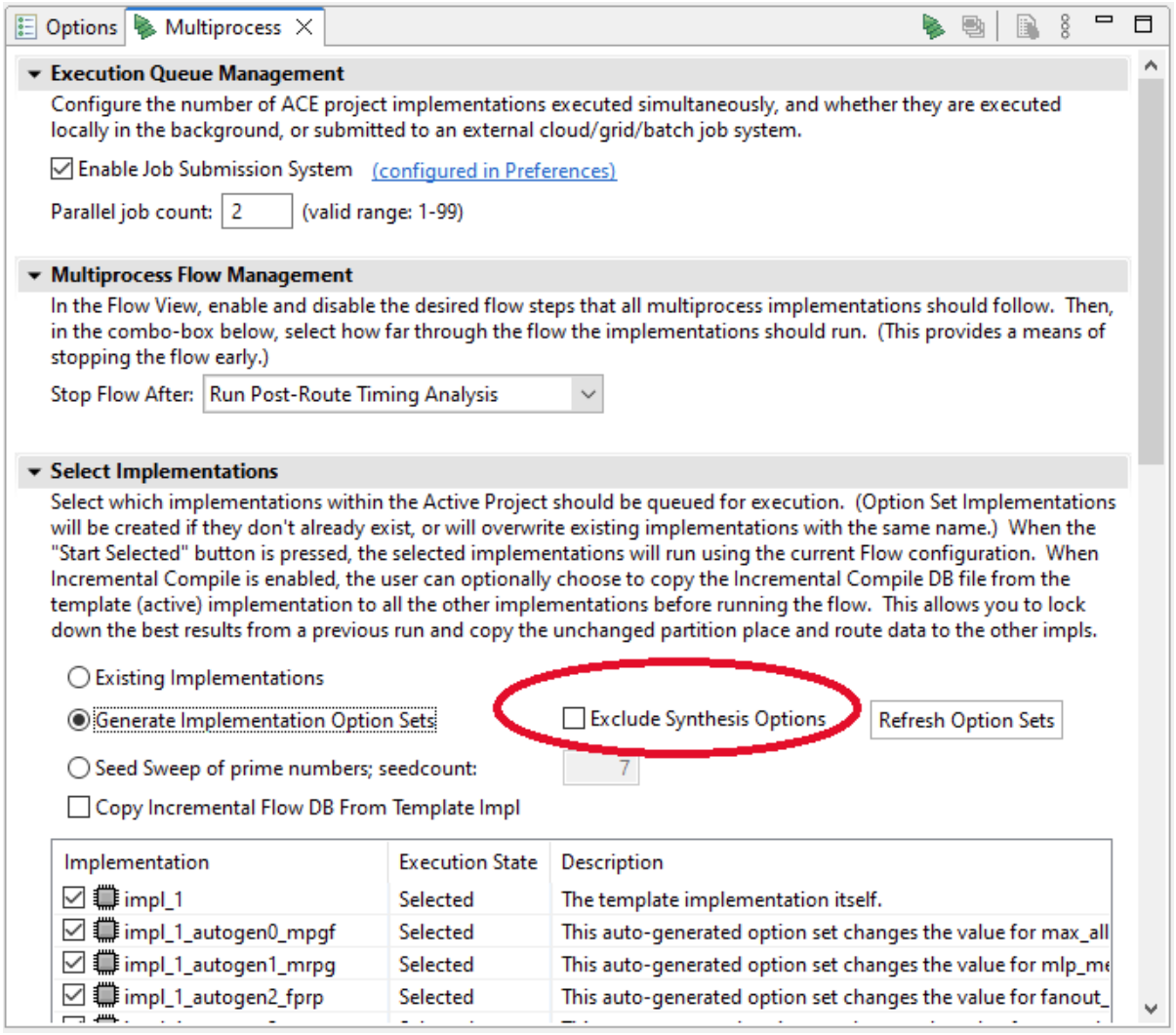


Figure 24 • Multiprocess View

When these requirements are met, ACE will sweep over synthesis implementation options in addition to the place-and-route implementation options, which can create a wider range of performance variation and help hone in on the best options to achieve that last 5% to 10% of f_{MAX} performance boost. Refer to the "Running Multiple Flows in Parallel" section of the *ACE Users Guide* (UG070)[13]. for more details.

13 https://www.achronix.com/documentation/ace-user-guide-ug070

Chapter 6 : Managing Projects in Synplify Pro

This chapter is only applicable to the following synthesis flows:

- Synplify-Pro-Driven Integrated Synthesis (page 14)
- Stand-Alone Synthesis in Synplify Pro (page 23)

> **Note**
>
> If using the ACE-Driven Integrated Synthesis (page 4) flow, Synplify Pro does not need to be launched outside of ACE. ACE will manage all aspects of synthesis automatically, including Synplify Pro project creation.

This guide assumes that Synplify Pro is installed with the **synplify_pro** command added to the $PATH variable.

Creating and Setting up a Project

1. In a Linux command shell type `synplify_pro` to invoke Synplify Pro synthesis. When invoked, the following window will be displayed:

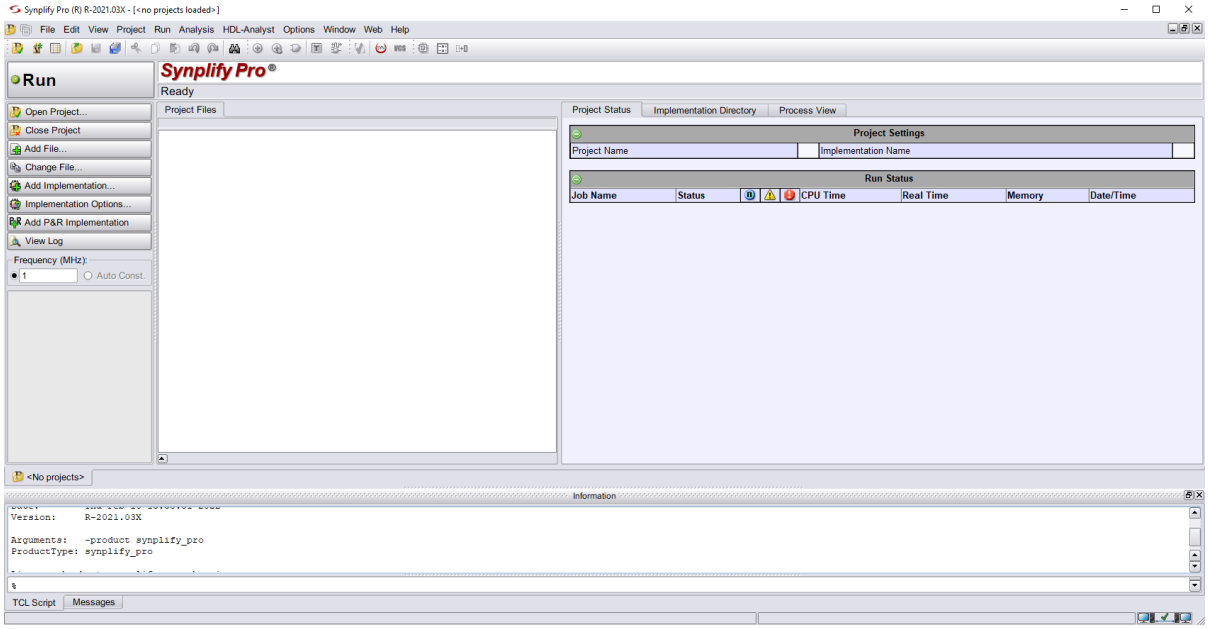


Figure 25 • Synplify Pro Invoked from the Command Shell

2. Click the **Open Project** button on the left side to open the open project dialog-box:.

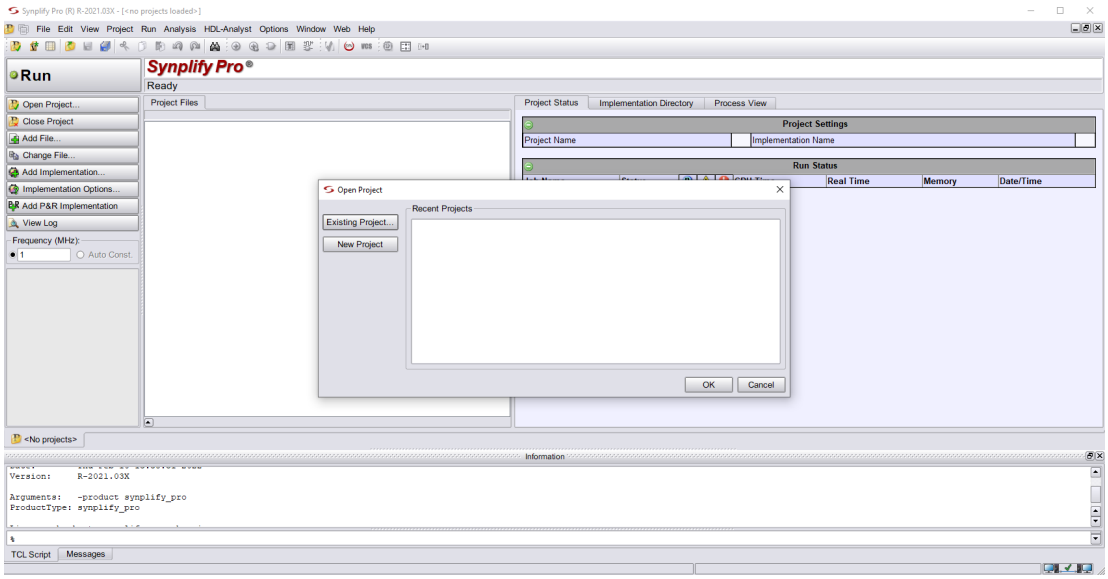


Figure 26 • Dialog Box to Select the New Project

3. Click the **New Project** button to open the following window:

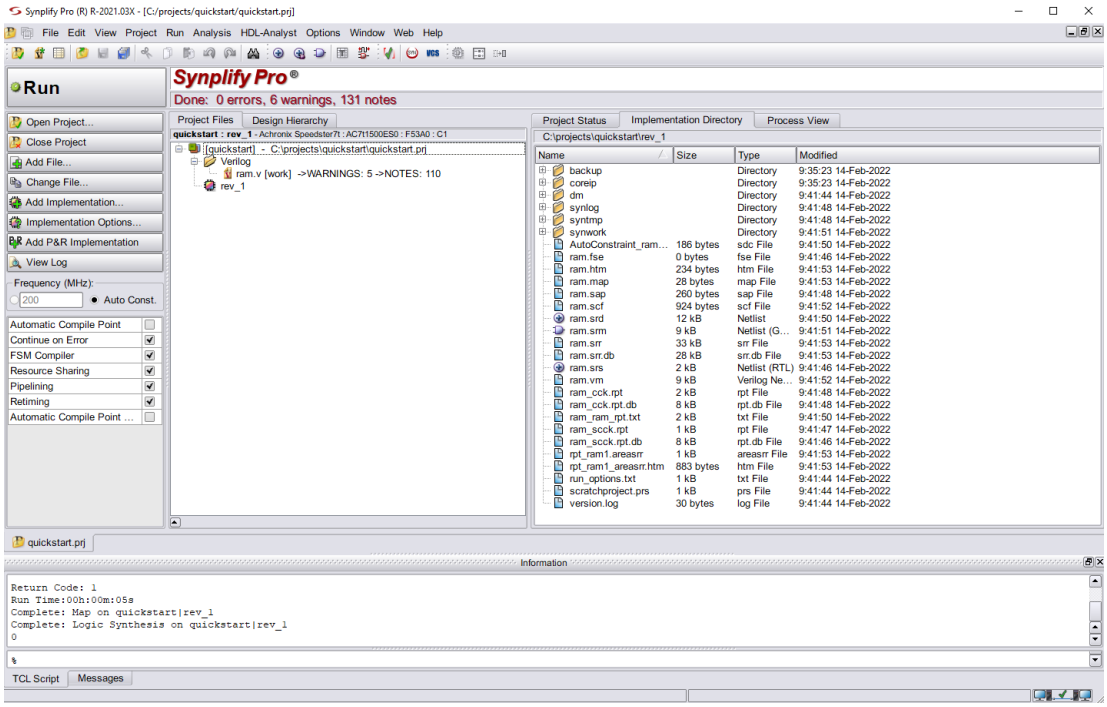


Figure 27 • Starting a New Project

Notes

1. Synplify Pro can open multiple projects at once; however only one can be run at time.
2. A single project supports multiple implementations with each having different:
   a. Device settings
   b. Optimization settings
   c. RTL define for different code builds

Adding the Synthesis Library Include File

After selecting and saving the project file inside the desired directory path, add the appropriate synthesis library include file and device specific synthesis constraints file:

- `<ACE_INSTALL_DIR>/libraries/device_models/<DEVICE>_synplify.sv`
- `<ACE_INSTALL_DIR>/libraries/device_models/<DEVICE>_synplify.fdc` (page 27)

The first file in the project file list should be the relevant ACE library file.

For the path to ACE libraries, the **ACE_INSTALL_DIR** environment variable can be used. By manually editing the Synplify Pro `.prj` file, a TCL variable that stores the value of an environment variable can be defined. Then, each time the TCL variable is used, ensure the full string is enclosed in { } rather than " ". For example:

```
#--  Synopsys, Inc.
#--  Version S-2021.09X-3
#--  Project file /views3/kevinhine/main/hls/PandA-Bambu/designs/pcie_mnist/syn/
pcie_mnist.prj
#--  Written on Thu Aug 31 10:01:41 2023

# Custom TCL source
syn_source {
    set ACE_INSTALL_DIR $::env(ACE_INSTALL_DIR)
}

...

add_file -verilog -vlog_std sysv {$ACE_INSTALL_DIR/libraries/device_models/
AC7t1500_synplify.sv}
set_option -include_path {../src/shell/include/;../hls/;$ACE_INSTALL_DIR/libraries/}

...
```

When the `.prj` is saved, the entire "syn_source" command written is preserved, as well as any places with the variable is enclosed with { }.

Warning!

If the variable is enclosed with "" instead of { }, the *value of the variable* will be written into the `.prj` on the next save.

Adding Source Files to the Project

There are two ways to add RTL source files. One is using the **Add File** button in the left menu bar, and the other one is to right-click on the project file and select **Add Source File**. Selecting either option directs the user to a dialog box listing available RTL files (see the figure below). The same procedure is followed for adding both source and constraint files.

In the examples that follow, the Speedster 7t technology has been selected, so the file `AC7t1500ES0_synplify.sv` is used. From this dialog box, select the desired RTL file(s) and then click **Add** followed by **OK**. The Verilog/VHDL file(s) will now be added to the project for synthesis.

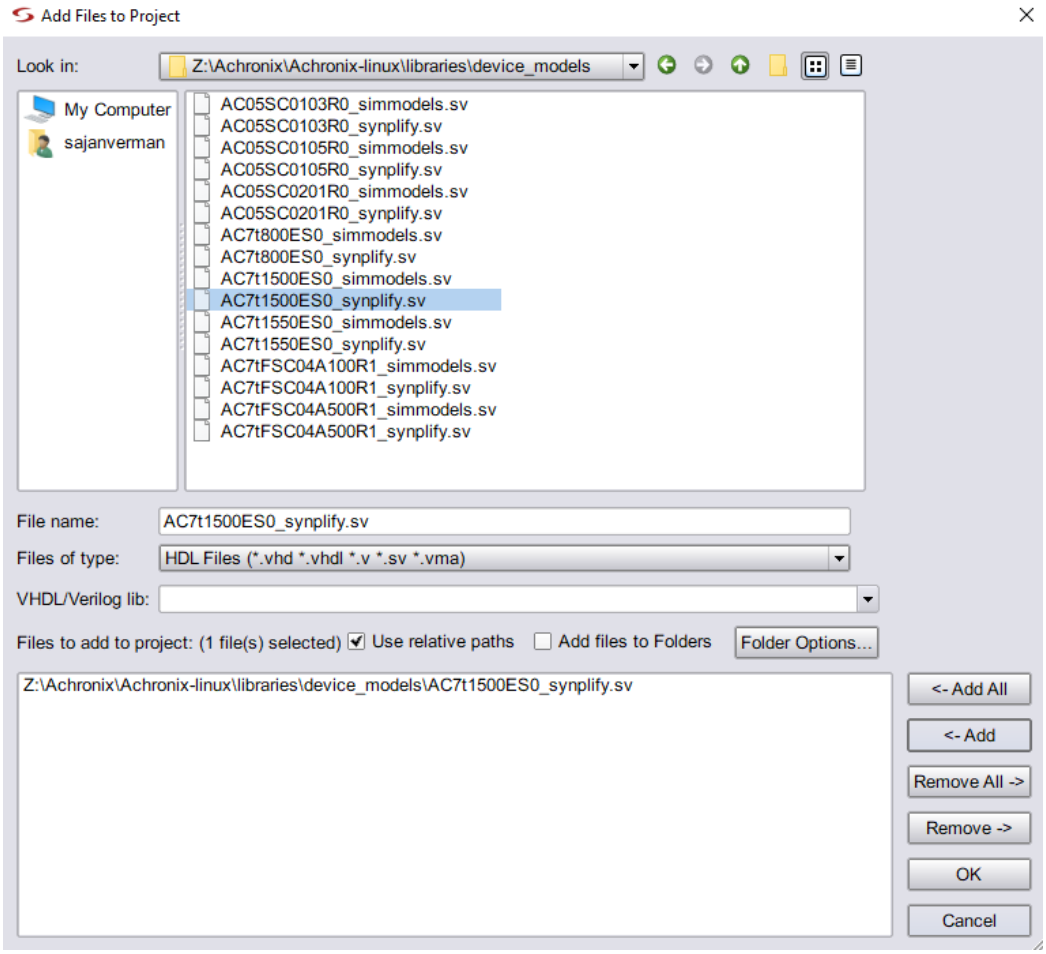


Figure 28 • Add Files to Project

Implementation Options

After adding the RTL files and constraint files, the next step is to set the implementation options. Click **Implementation Options** to open the window. shown below. This dialog box shows the default options. For example the "Fanout Guide" defaults to 10,000, but can be overwritten by the user for tuning QoR.

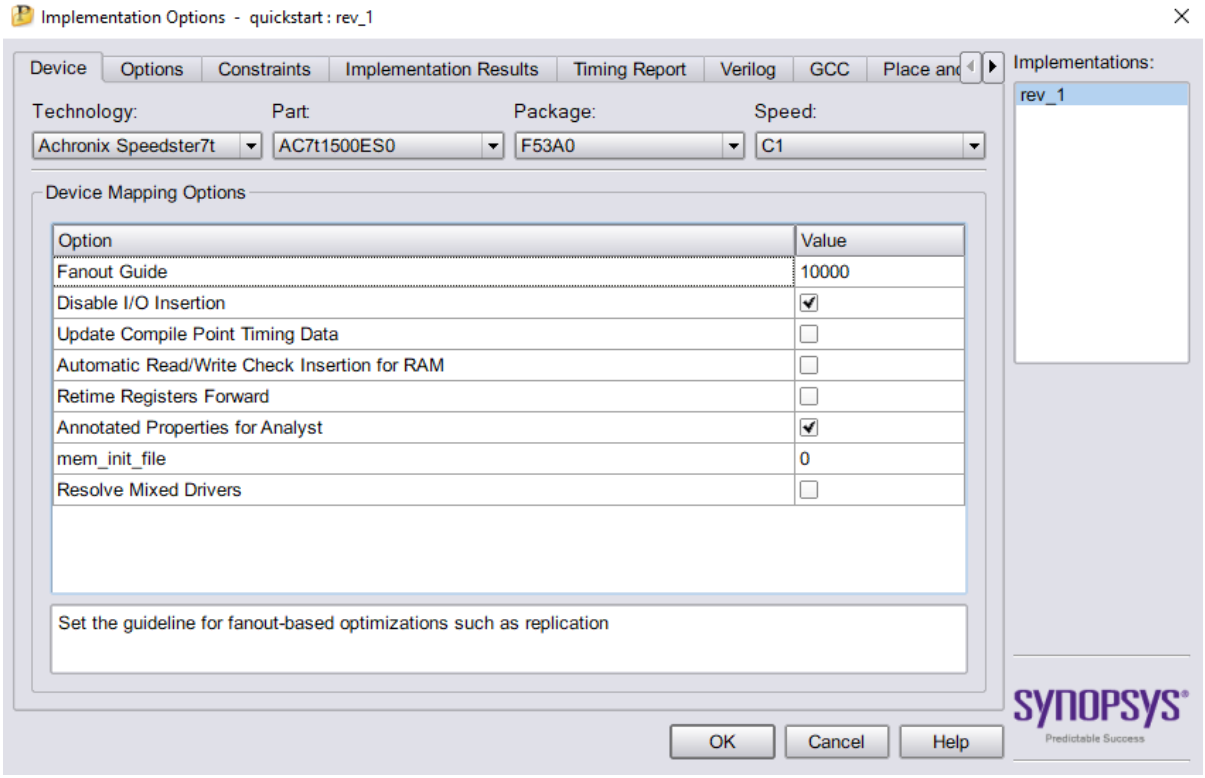


Figure 29 • Implementation Options

> **Note**
>
> For Achronix devices, ensure the **Disable I/O Insertion** option is checked as shown.

In the "Implementation Options" dialog box, the "Device" tab is selected by default. Each tab presentation additional options that can be set according to user's needs. Below are some guidelines for these options.

Verilog

Under this tab, the user may designate the top-level design module name. The user can also provide the names of any parameters existing in the design along with associated values. If parameters are defined in this manner, Synplify Pro propagates this value throughout the design. In this tab, the user must include the path to needed libraries under "Include Path Order." Click on the **+ file** icon to add the directory path and select from the ACE_installation path as shown below.

Note

"Library Directories or Files" box can be left empty.

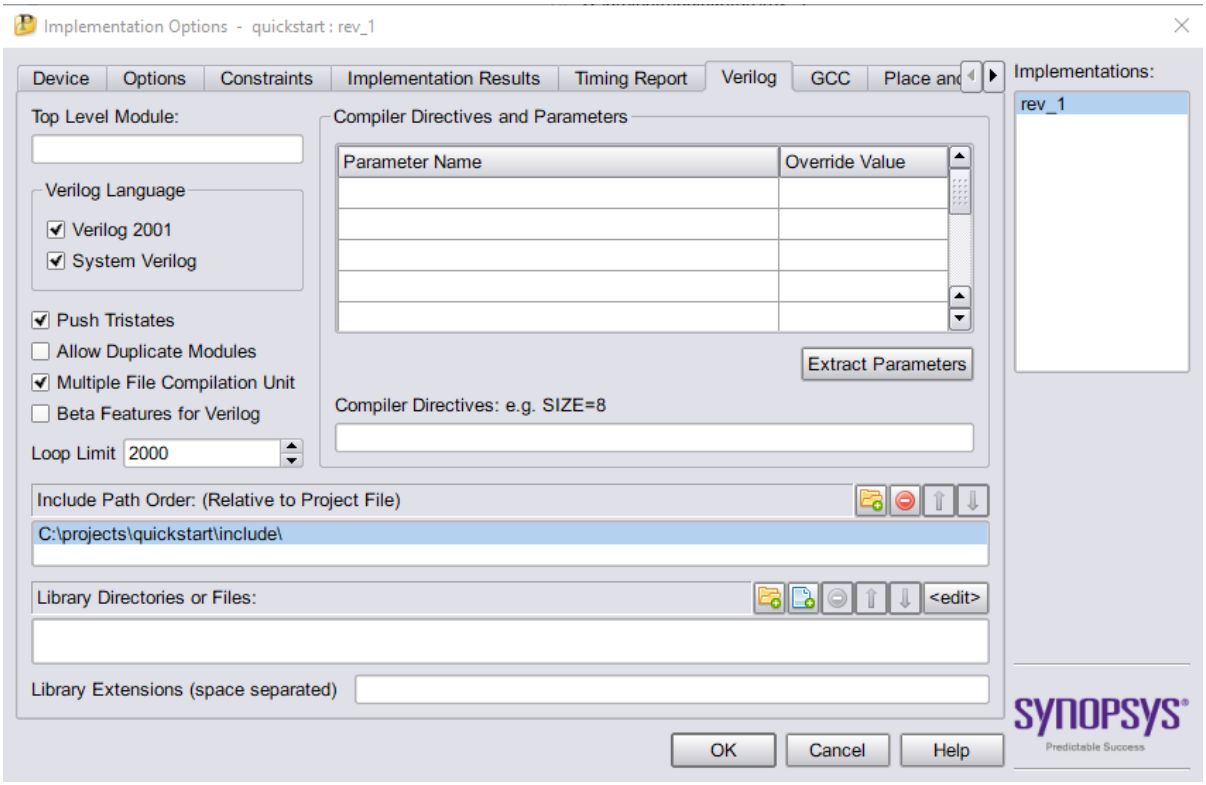


Figure 30 • Implementation Options: Include Path Order.

Place and Route

This tab is not presently utilized by ACE.

Timing Report

In the Timing report tab, the number of critical paths and number of start and end points can be specified to appear in the timing report. Default timing report is available in the synthesis report (`.srr`) file. The two available options are:

- Number of Critical paths – sets the number of critical paths for the tool to report.
- Number of Start/End points – specifies the number of start and end points to see reported in the critical path sections.

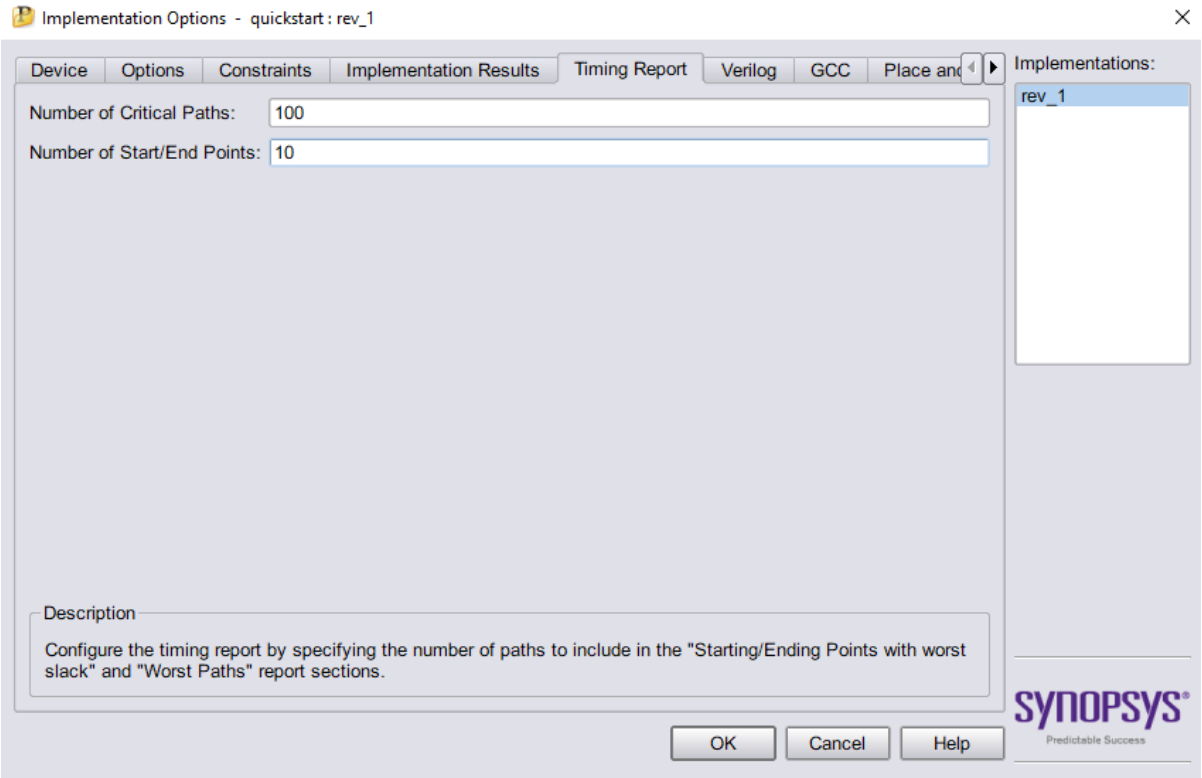


Figure 31 • Implementation Options: Timing Report

Implementation Results

Users may set their own implementation name in this tab; the default name is rev_1. The next box is the "Results Directory," specifying where users want to save the synthesized netlist file. The third box is "Results File Name," which sets the synthesized netlist file name.

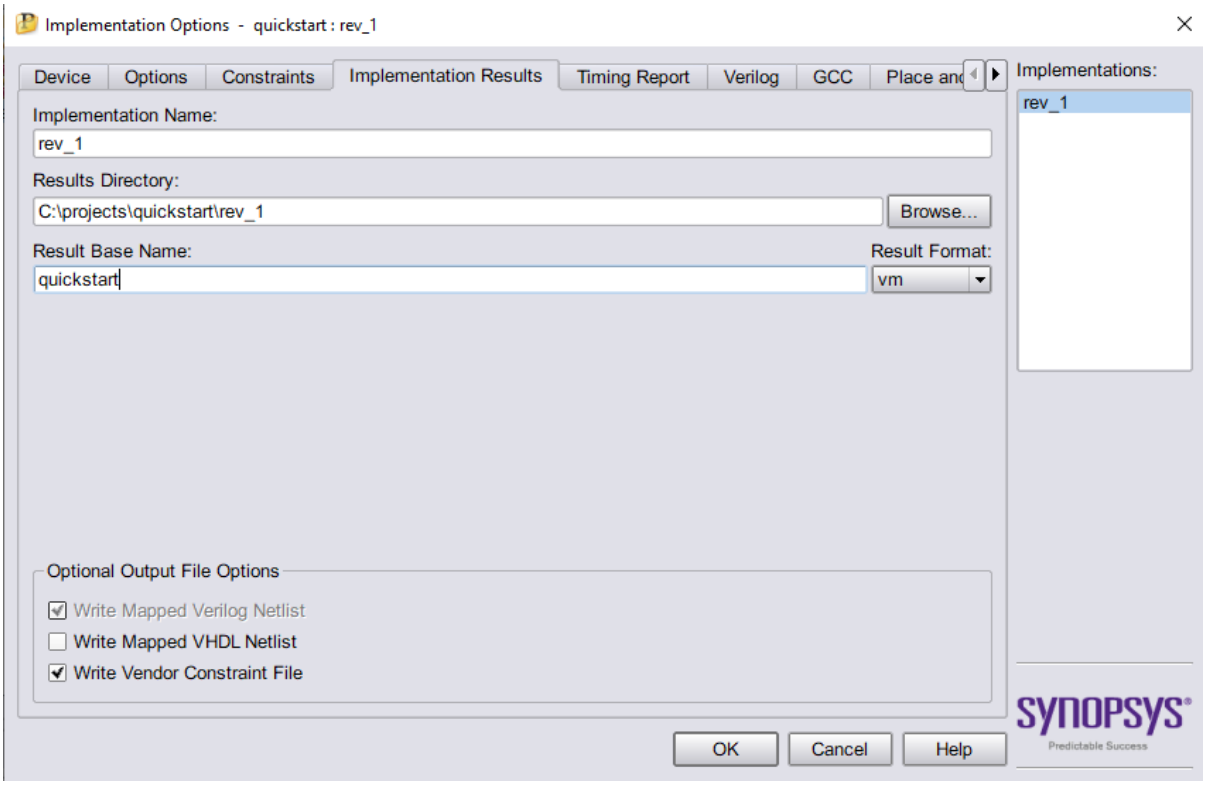


Figure 32 • Implementation Options: Implementation Results

Constraints

The Constraints tab is used to add synthesis constraint files if they were not added after adding source RTL files. This tab is also used to set the default clock speed of the design. Achronix highly recommends that a suitable constraint file be created for the synthesis project, specifying all of the clocks in the design. For details of how to add constraint files and their syntax see Synthesis Constraints (page 44).

In addition the default frequency should be set to the match the most common system clock frequency (by default it is set to 200 MHz).

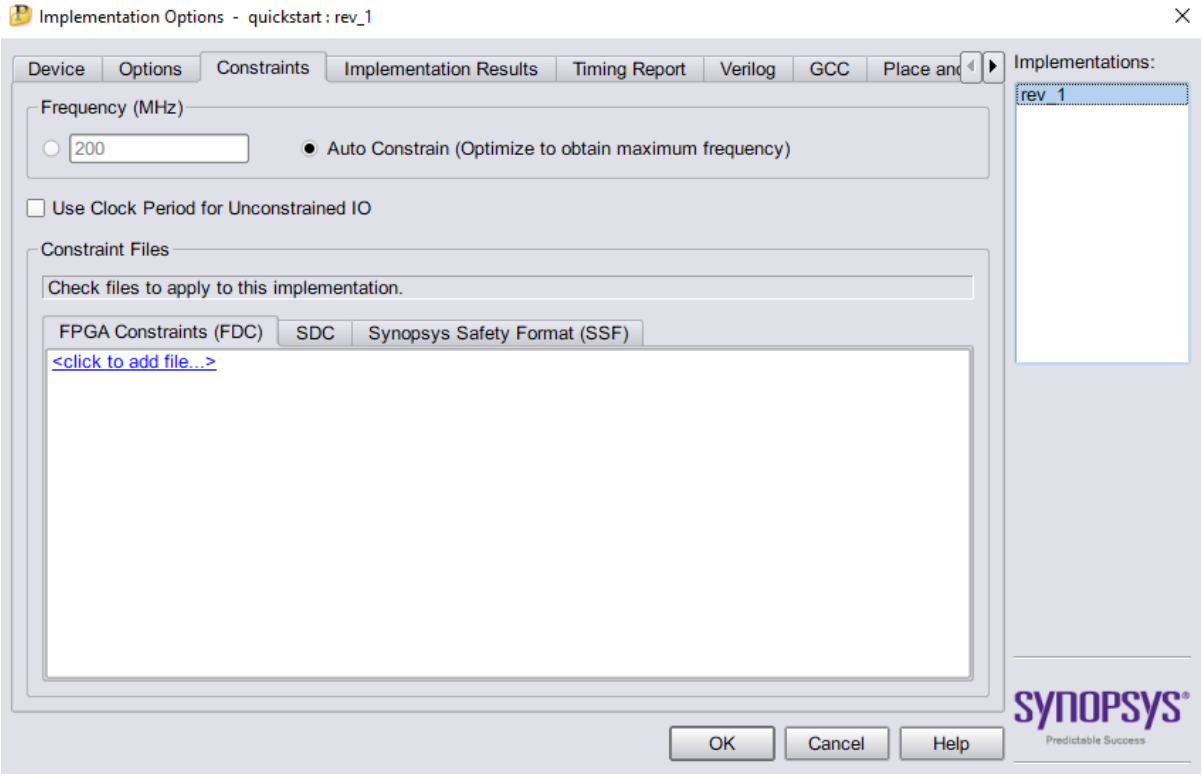


Figure 33 • Implementation Options: Constraints

Options

The Options tab sets the following optimization switches: **FSM Compiler, Resource Sharing, Pipelining** and **Retiming** — all are enabled by default. Users may change these optimization options according to design needs. For example, with resource sharing enabled, the software uses the same arithmetic operators for mutually exclusive statements as in branches of a case statement and hence area is optimized. Conversely, timing can be improved by disabling resource sharing, but at the expense of increased area.

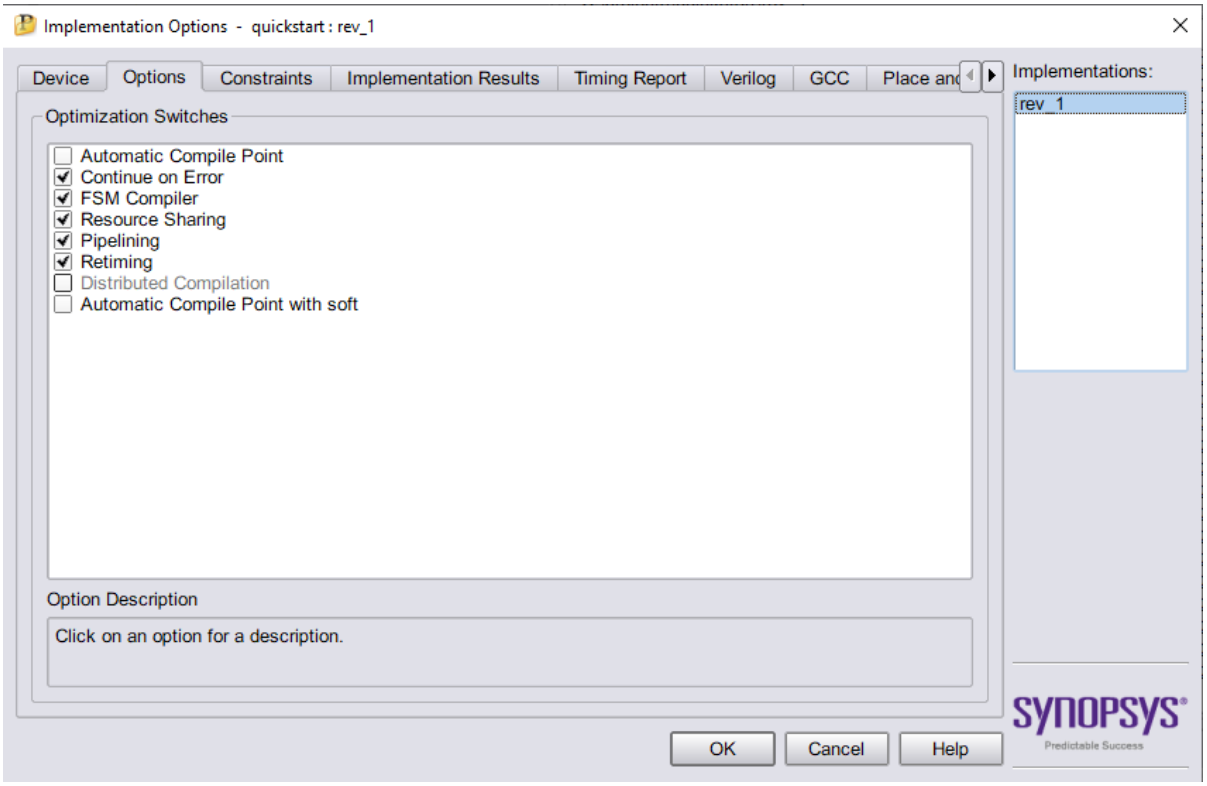


Figure 34 • Implementation Options: Options

Chapter 7 : Synplify Pro Features

There are several features in Synplify Pro which can be very useful. This section covers recommendations for:

- Synplify Warnings
- Synthesis Hierarchical Report
- HDL Analyst Schematics
- Watch Window
- Validating Constraints
- Using Help

Synplify Warnings

Users can make use of strong linting and checking capabilities provided by Synplify Pro.

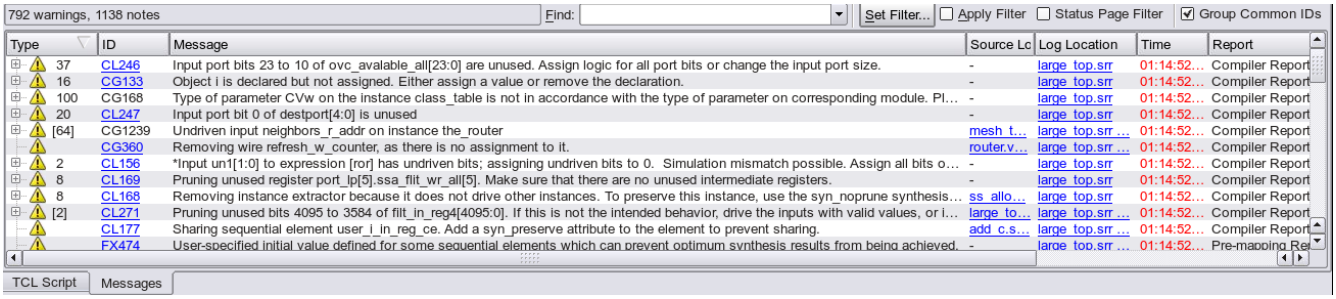
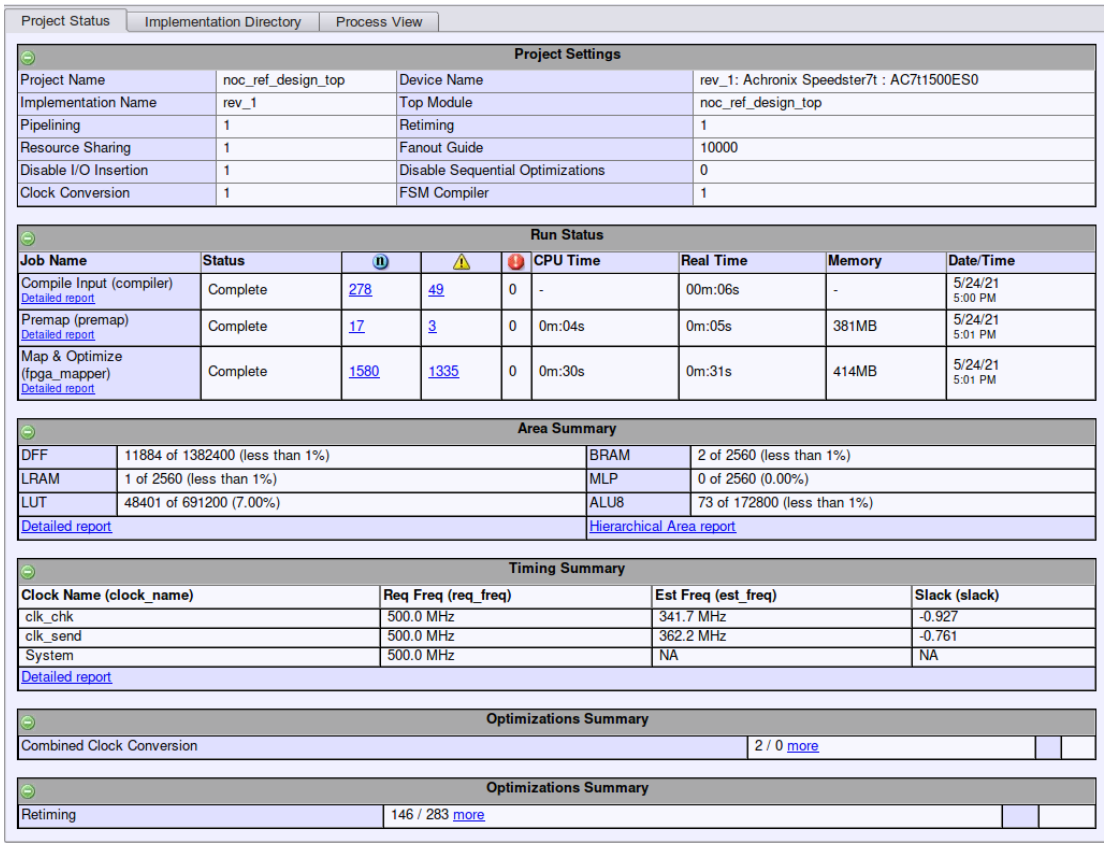


Figure 35 • Warning Messages

Synthesis Hierarchical Report

Synplify Pro has a hierarchical report to show different design statistics. The right-hand pane also shows, Project build status, Predicted timing, Resource Utilization:



Project Status | Implementation Directory | Process View

Project Settings

Project Name	noc_ref_design_top	Device Name	rev_1: Achronix Speedster7t : AC7t1500ES0
Implementation Name	rev_1	Top Module	noc_ref_design_top
Pipelining	1	Retiming	1
Resource Sharing	1	Fanout Guide	10000
Disable I/O Insertion	1	Disable Sequential Optimizations	0
Clock Conversion	1	FSM Compiler	1

Run Status

Job Name	Status				CPU Time	Real Time	Memory	Date/Time
Compile Input (compiler) Detailed report	Complete	278	49	0	-	00m:06s	-	5/24/21 5:00 PM
Premap (premap) Detailed report	Complete	17	3	0	0m:04s	0m:05s	381MB	5/24/21 5:01 PM
Map & Optimize (fpga_mapper) Detailed report	Complete	1580	1335	0	0m:30s	0m:31s	414MB	5/24/21 5:01 PM

Area Summary

DFF	11884 of 1382400 (less than 1%)	BRAM	2 of 2560 (less than 1%)
LRAM	1 of 2560 (less than 1%)	MLP	0 of 2560 (0.00%)
LUT	48401 of 691200 (7.00%)	ALU8	73 of 172800 (less than 1%)
Detailed report		Hierarchical Area report	

Timing Summary

Clock Name (clock_name)	Req Freq (req_freq)	Est Freq (est_freq)	Slack (slack)
clk_chk	500.0 MHz	341.7 MHz	-0.927
clk_send	500.0 MHz	362.2 MHz	-0.761
System	500.0 MHz	NA	NA

Detailed report

Optimizations Summary

Combined Clock Conversion	2 / 0 more

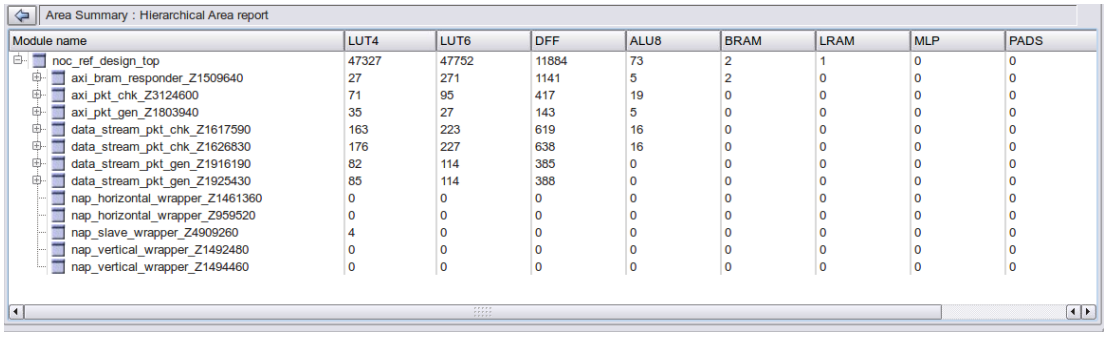
Optimizations Summary

Retiming	146 / 283 more

Figure 36 • Synthesis Hierarchical Report

Hierarchical Area Report

This report is useful to understand utilization of elements in the design, as well as, total sequential utilization for specific modules. The report is really helpful to understand the utilization hotspots in the design.



Area Summary : Hierarchical Area report

Module name	LUT4	LUT6	DFF	ALU8	BRAM	LRAM	MLP	PADS
noc_ref_design_top	47327	47752	11884	73	2	1	0	0
axi_bram_responder_Z1509640	27	271	1141	5	2	0	0	0
axi_pkt_chk_Z3124600	71	95	417	19	0	0	0	0
axi_pkt_gen_Z1803940	35	27	143	5	0	0	0	0
data_stream_pkt_chk_Z1617590	163	223	619	16	0	0	0	0
data_stream_pkt_chk_Z1626830	176	227	638	16	0	0	0	0
data_stream_pkt_gen_Z1916190	82	114	385	0	0	0	0	0
data_stream_pkt_gen_Z1925430	85	114	388	0	0	0	0	0
nap_horizontal_wrapper_Z1461360	0	0	0	0	0	0	0	0
nap_horizontal_wrapper_Z959520	0	0	0	0	0	0	0	0
nap_slave_wrapper_Z4909260	4	0	0	0	0	0	0	0
nap_vertical_wrapper_Z1492480	0	0	0	0	0	0	0	0
nap_vertical_wrapper_Z1494460	0	0	0	0	0	0	0	0

Figure 37 • Hierarchical Area Report

HDL Analyst Schematics

The Synplify Pro HDL Analyst features enable the user to visualize the end user design in several useful schematic views, including the hierarchical RTL view and flattened gate level netlist view. There are a variety of features to help filter and explore the design which can be accessed by the HDL Analyst top level menus or by right click menus within the schematic.

Browsing back and forth between the RTL view and the Technology (gate-level netlist) view enables users to visualize how the design RTL was mapped to FPGA primitives such as LUTs and registers.

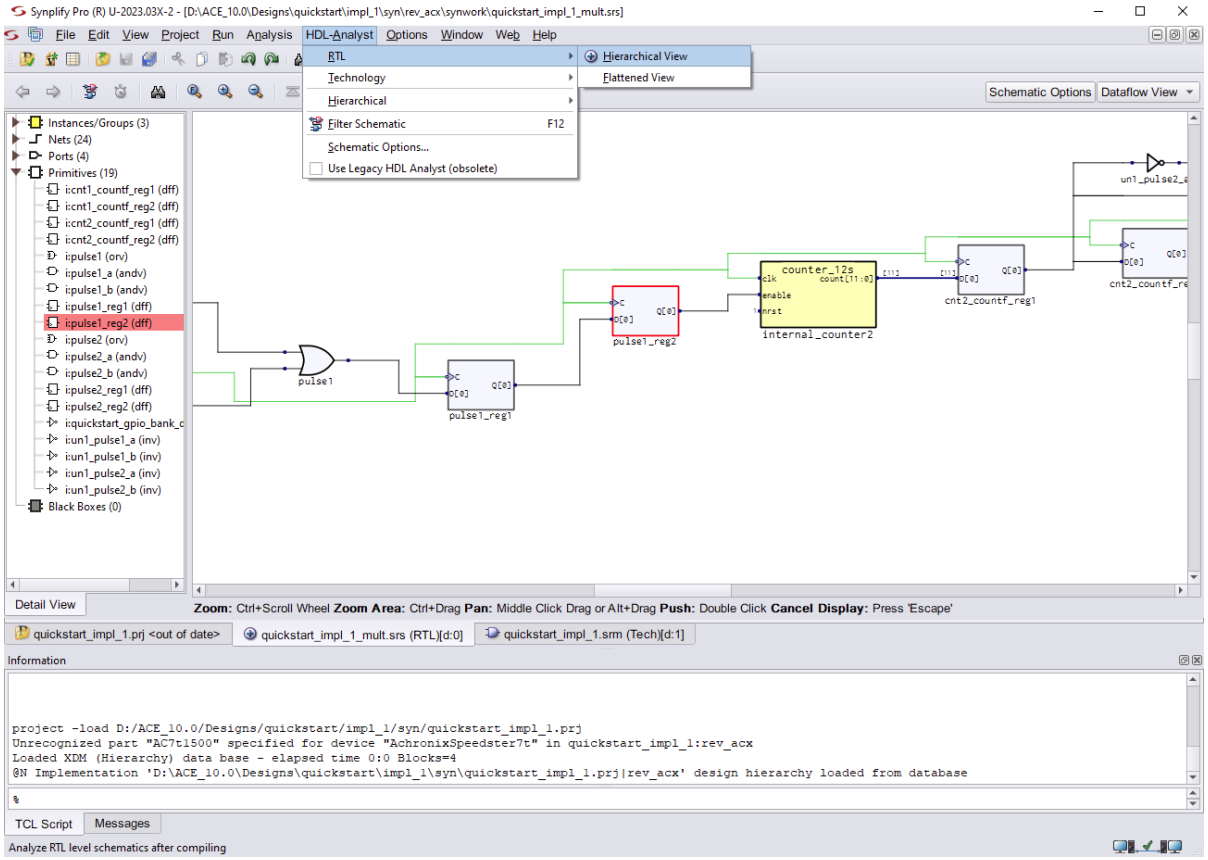


Figure 38 • HDL Analyst Hierarchical RTL View

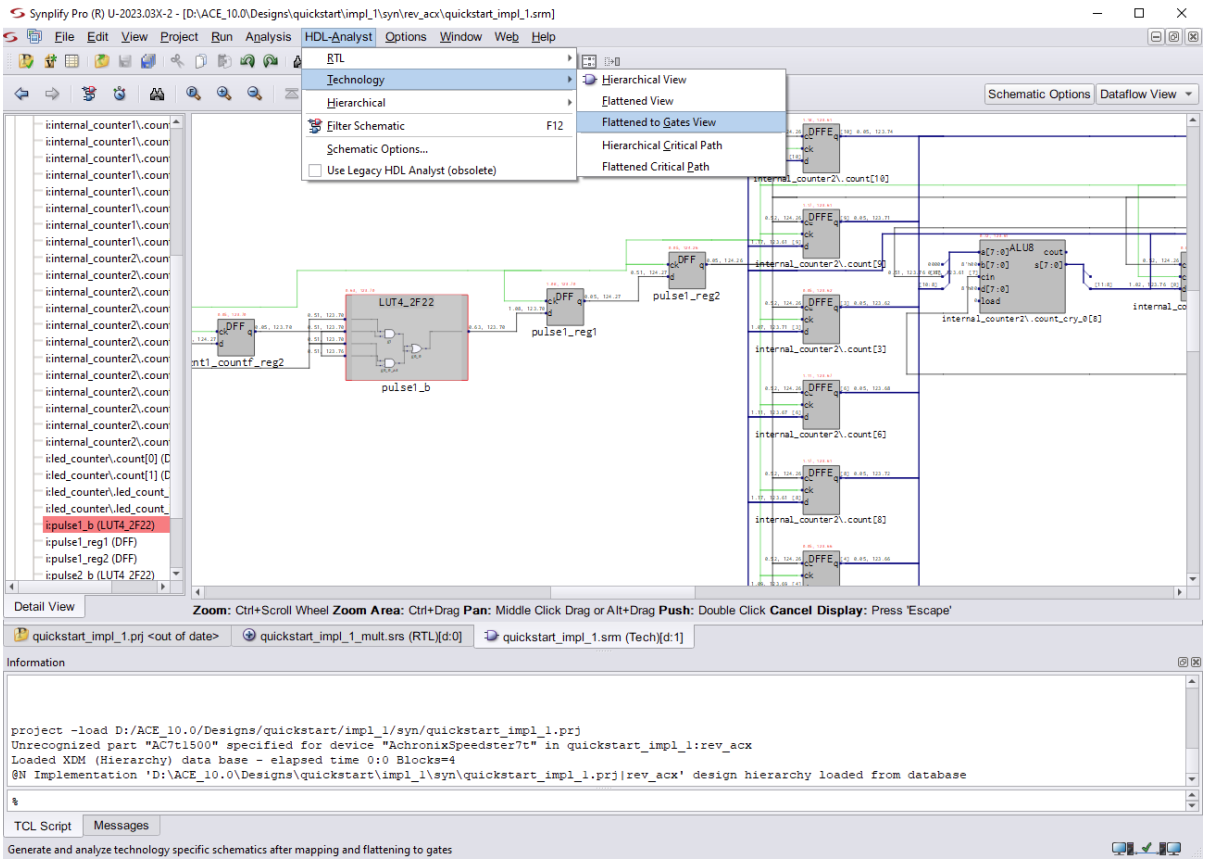


Figure 39 • HDL Analyst Flattened Gate-Level Netlist View

Watch Window

Watch window is useful to view and compare results of multiple implementations. Watch window can be enabled by the **View → Watch Window** command. Click in the **Log Parameter** section of the window and then click the pull-down arrows to display the parameter choices.

> **Note**
>
> Only a limited set of design parameters are supported for display.

To choose the implementations to watch, use the "Configure Watch" dialog box (right-click on **"Log Parameter**" section of the window) and select the implementations to watch.

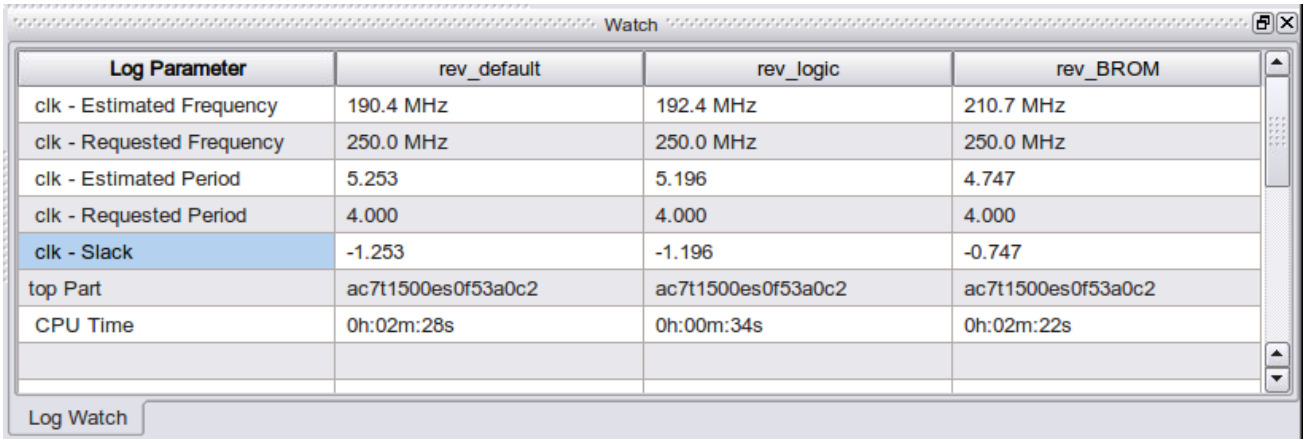


Figure 40 • Watch Window

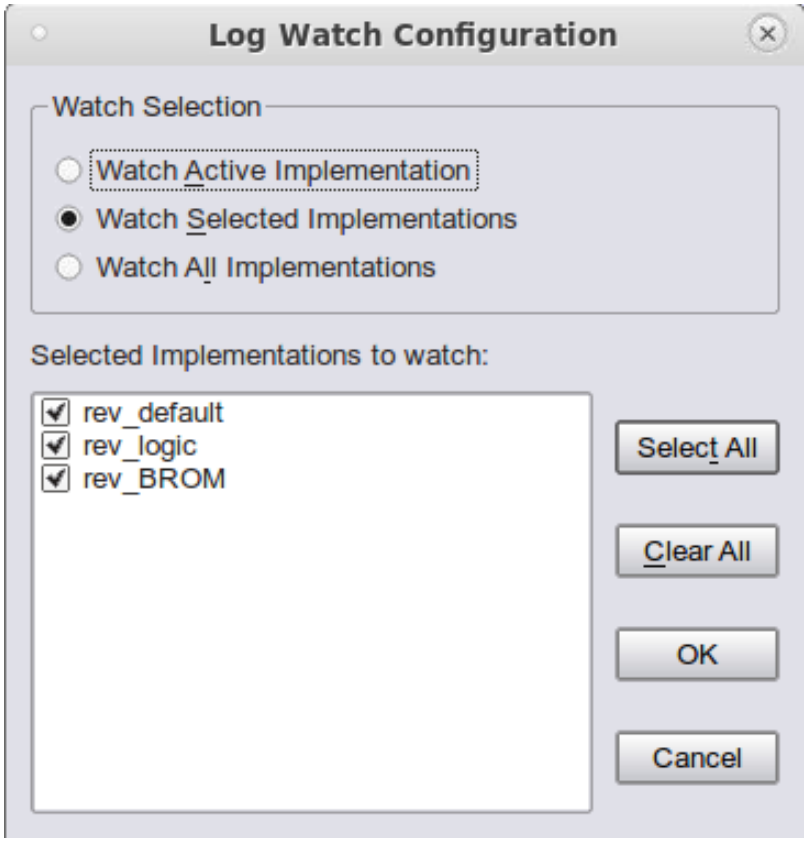


Figure 41 • Log Watch Configuration

Validating Constraints

Synplify Pro provides a constraint checker, which runs the preliminary stages of synthesis, and then checks the project constraint files against the objects in the design. It will report if any constraints cannot be successfully applied. It is highly recommended that constraint check is run to ensure that all constraints the user requires to be applied to the design are in fact being applied.

Select **Run** → **Constraint Check** to validate a project's constraints.

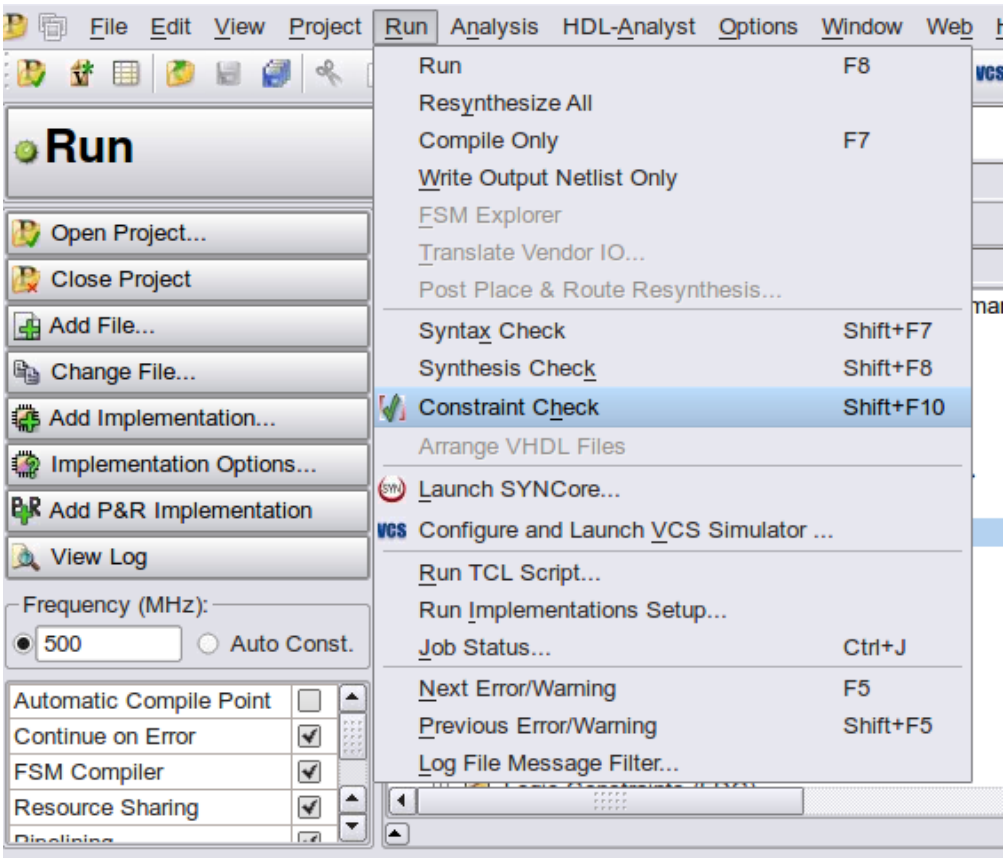


Figure 42 • Validating Constraints

Using Help

For getting help quickly, Synplify Pro provides very useful context sensitive help. For example, to access more information about the "**Attributes**" tab of the Scope editor, click **F1** key.

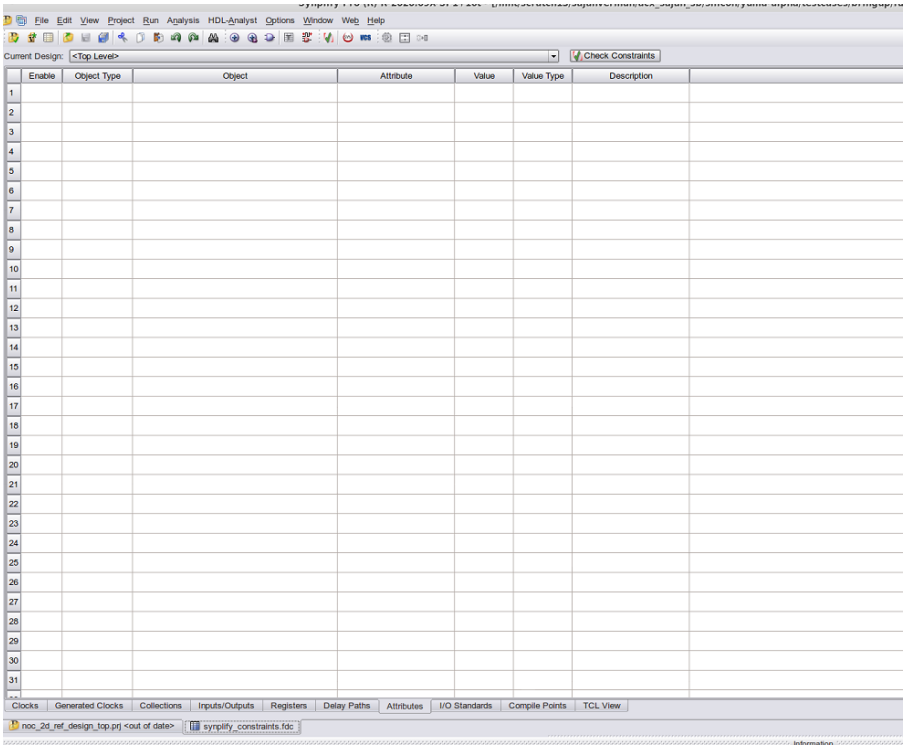


Figure 43 • Attributes Tab Within the Scope Editor

On clicking the **F1** key, help will automatically direct to relevant section of the help.

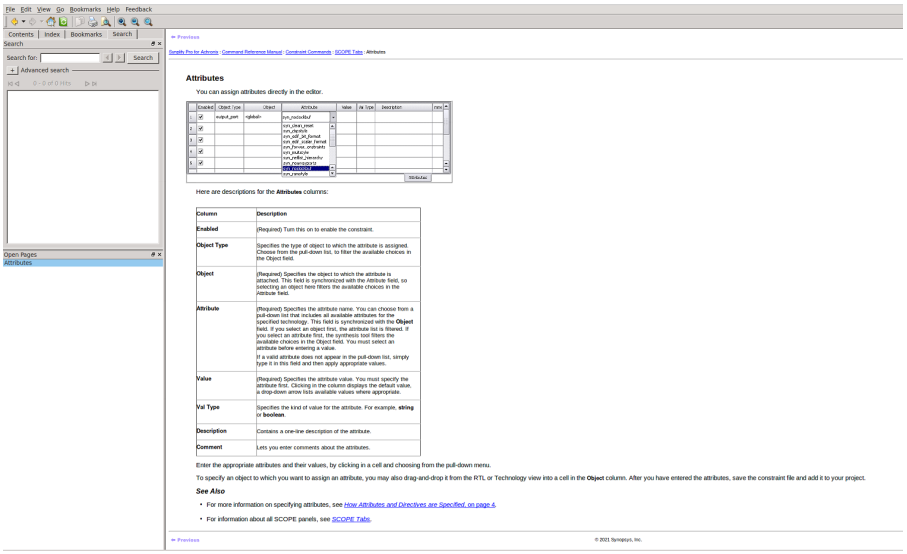


Figure 44 • Sample Help Screen

Chapter 8 : Synthesis Constraints

Synplify Pro constraints can be specified in two file types:

- Synopsys design constraints (SDC) – normally used for timing (clock) constraints. A second SDC file would be required for any non-timing constraints.
- FPGA design constraints (FDC) – usually used for non-timing constraints; however, can contain timing constraints as well.

SDC files are usually edited using a text editor, either as part of Synplify Pro or an external editor. FDC files can be edited in either a text editor or using the Scope editor within Synplify Pro. When using Synplify Pro to edit FDC files, an assistant tab is available which provides details of available FDC commands and their format.

Timing Constraints

It is highly recommended that the user defines all clocks in the design using an SDC file. If the design has multiple clocks, clock constraints should be set accordingly, defining either appropriate clock groups or false paths between asynchronous clocks. In addition, if required, the user can specify specific duty cycles for any particular clock.

Use the `create_clock` timing constraint to define each input clock signal and the `create_generated_clock` timing constraint to define a clock signal output from clock divider logic. The clock name (set with the `-name` option) will be applied to the output signal name of the source register instance. When constraining a differential clock, the user only needs to constrain the positive input.

For any clock signal that is not defined, Synplify Pro uses a default global frequency, which can be set with the `set_option -frequency` Tcl command in the Synplify Pro project file. However, Achronix recommends defining each clock in the design rather than relying on using this default frequency for undefined clocks.

A list of SDC commands are given below with examples. Refer to
`fpga_reference.pdf`
available in **Synplify Pro Tool → Help → PDF documents** for the description of the various options of the remaining SDC commands listed here.

create_clock

This command creates a clock object and defines its waveform in the current design. The options for `create_clock` are described in the table following.

Syntax

```
create_clock -name clockName [-add] {objectList} | -period {Value} [-waveform {riseValue fallValue}] [-disable] [-comment commentString]
```

Command Examples

```
create_clock -name inclk     -period 10 [get_ports {inclk1}]
create_clock -name divclk    -period 20 [get_nets {divclk}]
create_clock -name inclkfast -period 5 -add [get_ports {inclk1}]
create_clock -name inclk     -period 20 [get_ports {inclk1 inclk2 inclk3}] -waveform {
10 15 }
```

Table 1 • Option Description for create_clock

Option	Descriptions
-name clockName	Specifies the name for the clock being created, enclosed in quotation marks or curly braces. If this option is not used, the clock is given the name of the first clock source specified in the objectList option. If the objectList option is not specified, the -name option must also be used, which creates a virtual clock not associated with a port, pin, or net. Both the -name and objectList options can be used to give the clock a more descriptive name than the first source pin, port, or net. If specifying the -add option, the -name option must be used, and clocks with the same source must have different names.
-add	Specifies whether to add this clock to the existing clock or to overwrite it. Use this option when multiple clocks must be specified on the same source for simultaneous analysis with different waveforms. When this option is specified, the -name option must also be used.
-period Value	Specifies the clock period in nanoseconds (ns). The value type must be greater than zero.
-waveform riseValue fallValue	Specifies the rise and fall times for the clock in nanoseconds with respect to the clock period. The first value is a rising transition, typically the first rising transition after time zero. There must be two edges, and they are assumed to be a rise followed by a fall. The edges must be monotonically increasing. If this option is not specified, a default timing is assumed which has a rising edge of 0.0 and a falling edge of periodValue/2.
objectList	Clocks can be defined on the following objects: pins, ports, and nets.
-disable	Disables the constraint.
-comment textString	Allows the command to accept a comment string.

create_generated_clock

This command creates a generated clock object.

Syntax

```
create_generated_clock -name {clockName} [-add] -source {masterPin} -divide_by integer
```

Command Examples

```
create_generated_clock -name divclk -source [get_ports {inclk}] -divide_by 2 [get_nets
{divclk}]

create_generated_clock -name clk_div2 -source [get_pins {iPLL.ddr3_pll.iACX_PLL/
ogg_gm_clk[0]}] \
                                       -divide_by 2 \
                                       [get_pins
{i_ddr3xN_phy_w_ctrl_core.ddr3_inst\.i_ddr3_macro.x_ddr3.i_ddr3xN_phy_w_controller.i_ddr
3xN_phy.i_phy_sd_clkdiv/clkout}]
```

The period (.) is used as a separator between levels of hierarchy and instances. The backslash (\) is only used when referencing what is inside a generate block name. For example, the RTL appears as follows:

```
generate
 begin: ddr3_inst
   ddr3_macro i_ddr3_macro (...)
```

set_clock_groups

Specifies clock groups that are mutually exclusive or asynchronous with each other in a design.

Syntax

```
set_clock_groups -asynchronous -name clockGroupname -group{clockList}
```

Command Example

```
set_clock_groups -asynchronous -group {clk1 clk2} -group {clk3 clk4} -name clkgroup
```

set_false_path

This command removes timing constraints from particular paths.

Syntax

```
set_false_path [-setup]  [-from | -rise_from | -fall_from]  [-through]  [-to | -rise_to
| -fall_to]  value {objectList}
```

Command Examples

```
set_false_path -from [get_clocks inclk1] -to [get_clocks inclk2]
set_false_path -from temp2 -to out                    #(where temp2 is a register and out
is an output port)
set_false_path -from in                               #(where in is an input port )
set_false_path -from temp1 -to temp2                  #(where temp1 and temp2 are
registers)
set_false_path -from in -to temp1                     #(where in is an input port and
temp1 is a register)
set_false_path -from {i:temp2[*]} -to {mem_mem_0_0}  #(where temp is register bus and
mem_mem_0_0 is a RAM
```

set_input_delay

Sets input delay on pins or input ports relative to a clock signal.

Syntax

```
set_input_delay [-clock {clockName}] [-clock_fall] [-rise] [-fall] [-min] [-max] [-
add_delay] {delayValue} {portPinList}
```

Command Examples

```
set_input_delay 1.00 -clock clk {at} -max
set_input_delay {1.00} -clock [get_clocks {clk}] -max [get_ports {at}]
set_input_delay 2.00 -clock clk {bt} -min
set_input_delay 1.00 -clock clk -min -add_delay {bt}
set_input_delay 3.00 -clock clk {st}
set_input_delay 4.00 -clock clk -add_delay {st}
set_input_delay 1.00 -clock clk {din2} -clock_fall
set_input_delay 1.50 -clock clk {din1 din2}
set_input_delay 2.00 -clock clk [all_inputs]
```

set_output_delay

Sets output delay on pins or output ports relative to a clock signal.

Syntax

```
set_output_delay [-clock clockName [-clock_fall]] [-rise|[-fall] [-min|-max] [-
add_delay] delayValue {portPinList} [-disable] [-comment commentString]
```

Command Examples

```
set_output_delay 1.00 -clock clk {o1} -max
set_output_delay 3.00 -clock clk -max -add_delay {o1}
set_output_delay 2.00 -clock clk {o2} -min
```

set_max_delay

Specifies a maximum delay target for paths in the current design.

Syntax

```
set_max_delay [-from |-rise_from | -fall_from] [-through] [-to | -rise_to | -fall_to]
{delay_value}
```

Command Examples

```
set_max_delay 2 -from {a b } -to {o1}
set_max_delay  -rise_from {clk} {1}
set_max_delay  -through {{n:dout1}} {1}
set_max_delay 1  -fall_to {clk1}
```

set_multicycle_path

Modifies single-cycle timing relationship of a constrained path.

Syntax

```
set_multicycle_path [-start |-end] [-from {objectList}] [-through {objectList} [-through
{objectList} ...] ] [-to {objectList}] pathMultiplier [-disable] [-comment
commentString]
```

Command Examples

```
set_multicycle_path 2 -from [get_clocks inclk1] -to [get_clocks inclk2]
set_multicycle_path 4 -from temp2 -to out
```

set_clock_latency

Specifies clock network latency.

Syntax

```
set_clock_latency -source [-clock {clockList}] delayValue {objectList}
```

Command Example

```
set_clock_latency 0.2 -source [get_ports clk] -clock [get_clocks {clk}]
```

set_clock_uncertainty

Specifies the uncertainty or skew of the specified clock networks.

Syntax

```
set_clock_uncertainty {objectList} -from fromClock |-rise_from riseFromClock |
-fall_from fallFromClock -to toClock |-rise_to riseToClock | -fall_to fallToClock value
```

Command Example

```
set_clock_uncertainty 0.4 [get_clocks clk]
```

Below is an example of clock constraint commands for a multiple clock domain design.

> **Note**
>
> Most timing engines only use up to three decimal places of accuracy; therefore, it is normal to truncate non-rational values to this level.

```
# Clock definitions
create_clock -period 10    [ get_ports
{pll_refclk_p}                                                          ] -name
pll_refclk_p
create_clock -period 100   [ get_ports
{tck}                                                                   ] -name
tck
create_clock -period 1.527 [ get_pins
{i_clock_generator.i_PLL_EN.SW_APLL_0_pll_en_clk_APLL.iACX_PLL/ogg_gm_clk[0]}   ] -name
en_mac_ref_clk
create_clock -period 3.175 [ get_pins
{i_clock_generator.i_PLL_FF.SW_APLL_1_pll_ff_clk_APLL.iACX_PLL/ogg_gm_clk[0]}   ] -name
ff_clk
create_clock -period 3.448 [ get_pins
{i_clock_generator.i_PLL_SYS.SW_APLL_2_pll_sys_clk_APLL.iACX_PLL/ogg_gm_clk[0]} ] -name
sys_clk
create_clock -period 62.5  [ get_pins
{i_clock_generator.i_PLL_DCC.SW_APLL_3_pll_dcc_clk_APLL.iACX_PLL/ogg_gm_clk[0]} ] -name
sbus_clk

# By specifying clock group, each of the above clocks will be determined to be
asynchronous to all other clocks
set_clock_groups -asynchronous -name clk_grp1 -group {sbus_clk}       \
                                              -group {en_mac_ref_clk} \
                                              -group {pll_refclk_p}   \
                                              -group {sys_clk}        \
                                              -group {ff_clk}         \
                                              -group {tck}
```

Non-timing Constraints

An FDC file is used to specify non-timing constraints, which can be either attributes on an object (global or local), using the `define_attribute` statement, or compile points.

Compile Points

To implement compile points, they are specified in the FDC file as follows.

> **Note**
>
> For a detailed explanation of compile points how and when to use them, see **Compile Points** (page 60).

To set a single compile point, enter:

```
define_compile_point  {v:work.pac_ddr3_ip} -type {locked}
```

To find every instance of a module and set as a compile point, enter:

Compile Point syntax

```
foreach inst [c_list [find -hier -view pac_ddr3_ip*]] {
    define_compile_point $inst -type {locked}
}
```

Attributes

Attributes provides a mechanism to control how a design is mapped by Synplify Pro. Attributes can be defined both globally and also applied to individual instances. Attributes can be entered both in HDLs or in the SCOPE attributes tab, FDC files for project-wide entities. Attributes with **syn_*** do not affect synthesis and passed to the netlist.

Here is summary and examples of some of these attributes:

Attribute	Description
syn_allow_retiming	Controls retiming of registers across combinatorial logic on a global level or to specific register.
syn_dspstyle	Controls the mapping of objects to technology-specific DSP components. Options are "dsp" and "logic" for DSP64 or LUT/FF, respectively.
syn_ramstyle	Controls the implementation of an inferred RAM. Options are "block_ram", "logic_ram", and "registers" for BRAM, LRAM, and registers, respectively.
syn_romstyle	Controls the implementation of an inferred ROM. Options are "block_rom", "logic_rom" for BRAM and LRAM, respectively.
syn_keep	To preserve net in synthesis during optimization.
syn_preserve	To prevent sequential optimizations.
syn_noprune	To prevent optimization on instances and black boxes when output is not used.
syn_maxfan	To override global fanout guide for an individual port, net, register.

To override the number of available resources in a device, enter the following command. This command can be used to limit the mapping to certain resources.

```
define_global_attribute syn_allowed_resources {blockrams=1000}
```

To synthesize all ROMs using logic, enter:

```
define_global_attribute  {syn_romstyle} {logic}
```

To ensure that RAMs are only inferred for sufficiently large register sets, enter:

```
define_global_attribute  {syn_max_memsize_reg} {2048}
```

For more detailed information on all the supported attributes, refer to Synplify Pro online help "Attribute Reference Manual"

Chapter 9 : Synthesis Optimizations

There are several optimizations that can be performed by the user during Synplify Pro synthesis. This sections covers recommendations for:

Preventing Objects from Being Optimized Away

Dangling Nets

Synplify Pro always performs optimization on redundant or feed-through nets. At times, the user may want to preserve these nets. In order for these nets not to be optimized away (removed), add the following directive to the RTL, In this example, synthesis will not optimize away (remove) the logic. Instead, it infers a buffer between the two wire statements. If it is not specified, the user may not see the buffer insertion by the tool.

```
wire net1 /* synthesis syn_keep = 1 */ ;
wire net2 ;

assign net2 = net1 ;
```

Dangling Sequential Logic

For sequential logic the `syn_preserve` attribute is used.

```
reg net_reg1 /* synthesis syn_preserve = 1 */ ;

always @ (posedge clk)
    net_reg1 <= some_net;
```

Unconnected Instances

For input instances when their output pins are unconnected, the `syn_noprune` attribute is used. The following examples show how to apply this attribute to both Speedster I/O pads and Speedcore boundary pins.

Speedster Output Pad

```
PADIN ipad ( .padin(in[0]) ) /* synthesis syn_noprune = 1 */;
```

Speedcore Output Pin

```
IPIN ipin ( .din(in[0]) ) /* synthesis syn_noprune = 1 */;
```

Prevent ACE Optimizing Objects Away

In the above examples, Synplify Pro does not remove the unconnected entity, ensuring that the Synplify Pro netlist retains these entities. However, when the netlist is read into ACE, ACE performs netlist optimization and resynthesis. If the objects retained by synthesis are still unconnected, then ACE will remove these entities from the final place-and-route netlist. To prevent ACE from optimizing these entities, use the ACE `must_keep` directive in conjunction with the above attributes. Using the preceding sequential logic example, the `must_keep` attribute is passed through Synplify and included in the synthesized netlist. ACE will then recognize this attribute and keep the instance.

> **Note**
>
> The attribute `must_keep` can be applied to both sequential elements and wires.

```
(* must_keep=1 *) reg net_reg1 /* synthesis syn_preserve = 1 */ ;

always @ (posedge clk)
    net_reg1 <= some_net;
```

Pipelining

Pipelining is the process of splitting logic into stages so that the first stage can begin processing new inputs while the last stage is finishing the previous inputs. Pipelining ensures better throughput and faster circuit performance. If using selected technologies which use pipelining, also use the related technique of retiming to improve performance.

When this switch is enabled in a project file, synthesis uses register balancing and pipeline registers on multipliers and ROMs.This option is equivalent to enabling the Pipelining option on the Options panel of the Implementation Options dialog box.

Retiming

The retiming process moves storage devices (flip-flops) across computational elements with no memory (only gates/LUTs) to improve the performance of the circuit.

When this switch is enabled, synthesis tries to improve the timing performance of sequential circuits. This option is equivalent to enabling the Retiming option on the Options panel of the Implementation Options dialog box. Use the `syn_allow_retiming` attribute to enable or disable retiming for individual flip-flops. This option also adds a retiming report to the log file.

> **Note**
> Pipelining is automatically enabled when retiming is enabled.

Forward Annotation of RTL Attributes to the Netlist

Synplify Pro supports forward annotation of RTL attributes to the netlist. These user-defined attributes propagate to the netlist to be used by ACE place and route for optimization. This feature requires the usage of various directives available in Synplify Pro such as `syn_noprune`,`syn_keep`, `syn_hier`,`syn_preserve`, etc., to propagate user-define attributes to the netlist. The table below lists the directives to be set on the mentioned objects in order to forward annotate the RTL attribute.

Object	Directive	Result
Module	syn_hier=”hard”	Attribute applied on the module will propagate to the netlist
Instantiated Components	syn_noprune	Attribute applied on the instantiated component will propagate to the netlist
Input/Output ports	syn_hier="hard" on the module containing the ports	Attribute applied on ports will propagate to the input/output port in the netlist
Registers	syn_preserve	Attribute applied on the registers will propagate to the netlist
Wire	syn_keep	Attribute applied on nets/wires will propagate to the netlist

Below are some examples:

Example 1

The attribute `weight="3.0"` propagates to my_reg in the netlist. The syntax used is Verilog 2001 style parenthetical comments.

```
(* syn_preserve=1, weight="3.0" *) reg  my_reg;
```

Example 2

The syntax used is C-style comment.

```
reg my_reg /* synthesis syn_preserve=1 weight=4 */;
```

Note

When using C-style comment, a comma is *not* required after `syn_preserve=1.` When using Verilog 2001 style, a comma *is required* after `syn_preserve=1`.

Example 3

This example illustrate attribute propagation on nets.

```
(* syn_keep = 1, weight = 3 *) wire n2;
```

Example 4

This feature of attribute propagation is utilized in flop pushing to boundary pins or I/O pads via the ACE attribute `syn_useioff`. The `syn_useioff` is applied to the input and output ports in the below example.

```
module flop_push_test1 (
    ina, inb, sel, clk, z0
);

input  wire [3:0] ina  /* synthesis syn_useioff=1 */;
input  wire [3:0] inb  /* synthesis syn_useioff=0 */;
input  wire       sel  /* synthesis syn_useioff=1 */;
input  wire       clk;
output reg        z0   /* synthesis syn_useioff=1 */;

    reg       sel_r0=1'b0, sel_r1=1'b0;
    reg [3:0] ina_r0=4'h0, ina_r1=4'h0, inb_r0=4'h0, inb_r1=4'h0;

    always @(posedge clk)
    begin
      sel_r0 <= sel;
      sel_r1 <= sel_r0;
      ina_r0 <= ina;
      ina_r1 <= ina_r0;
      inb_r0 <= inb;
      inb_r1 <= inb_r0;

      z0 <= sel_r1 ? & inb_r1 : |ina_r1;
    end

endmodule
```

Note

In example 4, the module flop_push_test1 is a top module; therefore, `syn_hier="hard"` is not specified on the module. If it were a sub module, `syn_hier="hard"` is required for the attribute on ports to propagate to the netlist; for example:

```
module flop_push_test1 (ina, inb, sel, clk, z0) /* synthesis syn_hier="hard" */;
```

Note

In example 4, the `syn_useioff` attribute could also be specified in the Verilog 2001 comment style. For example:

```
(* syn_useioff=1 *) input [3:0] ina;
```

However, that style only works correctly when the attribute has a non-zero value. Synplify Pro cannot distinguish between the value zero and and an attribute that is not present. In that case it will not forward annotate the attribute to the netlist used by ACE. Therefore, it is recommended to always use the C-style comment used in example 4.

Example 5

This example illustrates attribute propagation on instantiated components:

```
module att_propagate_instcomp (
    d1, d2, d3, clk, out1
);

input  wire d1,d2, d3, clk;
output reg  out1;

reg q1,q2;

//Instantiate 2 instances U1 and U2 of module test2
    (* must_keep = 1, syn_noprune = 1 *) test2 U1 (d1,d2,d3, clk,out2);
    (* syn_noprune = 1, must_keep = 1 *) test2 U2 (d1,d2,d3, clk,out2);

    always @(posedge clk)
        q1 <= d1;

    assign combo1 = q1 & d2 & d3;

    always @(posedge clk)
        q2 <= combo1;
```

```
    assign combo2 = q2 | combo1;

    always @(posedge clk)
        out1 <= combo2;

endmodule

// ------------------------------------------------------

module test2 (
    d1, d2, d3, clk, out1
) /*synthesis syn_hier = hard */;

input  wire  d1, d2, d3, clk;
output reg   out1;

reg q1,q2;

    always @(posedge clk)
        q1 <= d1;

    assign combo1 = q1 | d2 | d3;

    always @(posedge clk)
        q2 <= combo1;

    assign combo2 = q2 & combo1;

    always @(posedge clk)
        out1 <= combo2;

endmodule
```

Example 6

This example shows attribute propagation on modules:

```
(* att0=1 *) module top (
    d1, d2, d3, clk, out1, out2
);

input  wire d1, d2, d3, clk;
output wire out2;
output wire out1,

    // Instantiate test1
```

```
    test1 U1 (d1, d2, d3, clk, out1);

endmodule

// ------------------------------------------------------

(* must_keep=1 *) module test1 (
    d1, d2, d3, clk, out1
) /* synthesis syn_hier="hard" */;

input  wire d1, d2, d3, clk;
output reg  out1;

reg q1,q2;

    always @(posedge clk)
        q1 <= d1;

    assign combo1 = q1 & d2 & d3;

    always @(posedge clk)
        q2 <= combo1;

    assign combo2 = q2 | combo1;

    always @(posedge clk)
        out1 <= combo2;

endmodule
```

Example 7

As shown above, flop pushing can take advantage of attribute propagation to control specific I/O pads or boundary pins. The examples below shows how to control flop pushing from within the RTL, applying the attribute to both Speedster I/O pads and Speedcore device boundary pins.

This example illustrates the application of the `syn_useioff` attribute with a value of `0` on, respectively:

- A wire
- A black-box PAD instance, an IPIN instance, the IPIN input net, the IPIN output net (Speedcore only)
- An PADIN instance (Speedster only)
- A pair of DFF instances

All of the above are valid instances to which to apply this property:

```
(* syn_keep=1 *) wire ipad_dout          /* synthesis syn_useioff = 0 */;
(* syn_keep=1 *) wire ipin_dout          /* synthesis syn_useioff = 0 */;
                 wire dff1_q, dff2_q;
```

```
BB_PADIN i_bb_padin  ( .padin(sc_in) ,  .dout(bb_pad_dout)    )  /* synthesis
syn_useioff = 0 */;
PADIN    i_padin     ( .padin(sp_in) ,  .dout(padin_dout)     )  /* synthesis
syn_useioff = 0 */;
IPIN     i_ipin      ( .din(ipad_dout), .dout(ipin_dout)      )  /* synthesis
syn_useioff = 0 */;

ACX_DFF  i_dff1      ( .d(ipin_dout) ,  .ck(clk) . .q(dff1_q) )  /* synthesis
syn_useioff = 0 */;
ACX_DFF  i_dff2      ( .d(ipin_dout) ,  .ck(clk) . .q(dff2_q) )  /* synthesis
syn_useioff = 0 */;
```

For full details on all the options for flop pushing, see the section "Automatic Flop Pushing into I/O Pins" in the ***ACE Users Guide*** (UG070)[14].

> **Note**
>
> As in Example 7, the `syn_useioff` attribute must be specified with a `synthesis` directive in a C-style comment because it has a value of zero. However, the `syn_keep=1` attribute on the wire can be specified in either style.

Compile Points

Compile points are RTL partitions of the design which are defined before synthesizing a design. The advantages of using compile points is design preservation, runtime savings and improves efficiency of top-down and traditional bottom-up design flows.

Synplify Pro supports both automatic and manual compile points. The automatic compile-point feature can be selected from "Implementation Options" dialog box as shown below. When automatic compile points are enabled, the tool automatically identifies compile points based on various parameters such as size of the design, hierarchical modules, boundary logic, etc. Refer the

`fpga_user_guide.pdf`

available with Synplify Pro for details on compile points.

14 https://www.achronix.com/documentation/ace-user-guide-ug070

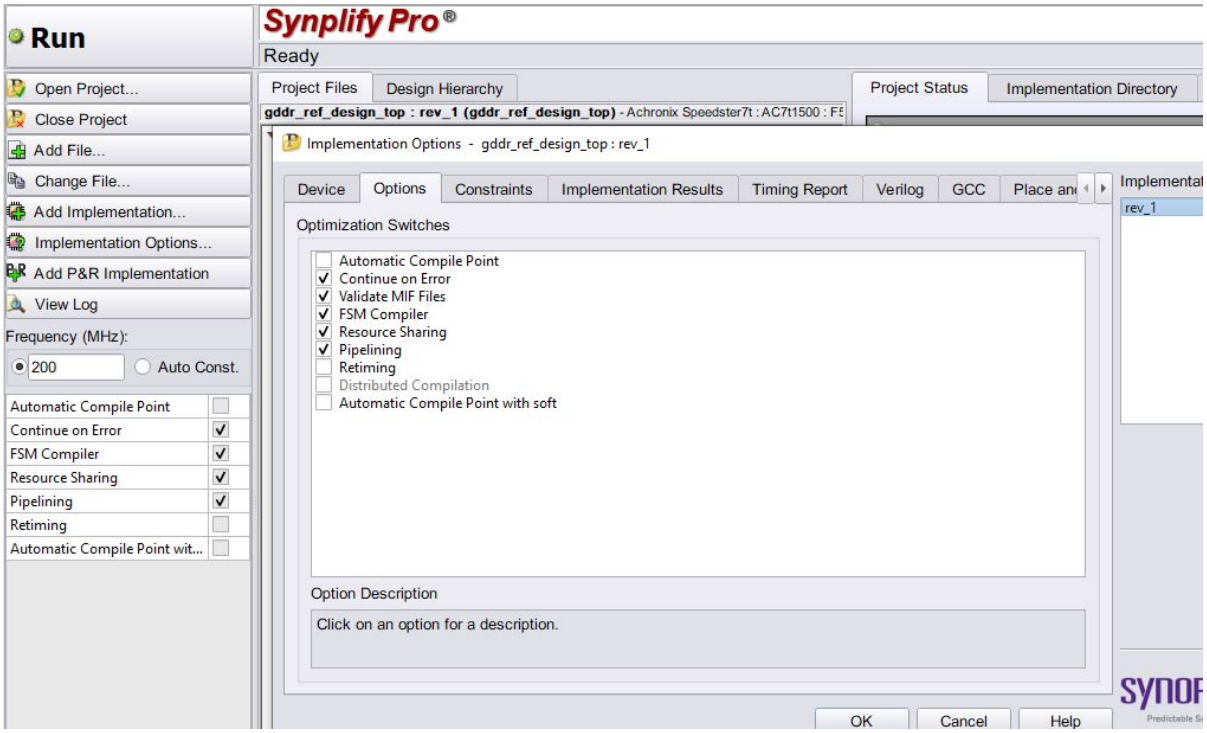


Figure 45 • Setting Compile Points

Although compile points can deliver significant runtime savings, users should be aware that they can have a detrimental effect on quality of results (QoR) if not used with care. Compile points identify blocks of code that are repeated, guiding Synplify Pro to only synthesize that block once. The level of optimization between a compile point and it's enclosing module is defined by the compile point type:

- **Locked** – No optimizations across compile point boundary. Locked compile points are used for the Achronix incremental compile flow
- **Hard** – Signals can be optimized across the compile point boundary (i.e., back-to-back inverters removed). However, the actual interface is not optimized — all signals remain. All automatic compile points are set to hard.
- **Soft** – Signals can be optimized across the compile point boundary, and the signals themselves may be removed, or renamed. Therefore, almost full optimization can occur as though the design did not have compile points.

The three modes above result in increasing runtimes; however, they also generally result in increased QoR as greater optimizations can be performed. Users should determine which configuration of compile points, if any, best meet their needs with regards to performance versus runtime.

> **Caution!**
>
> If automatic compile points are enabled, users must be aware that all automatic compile points are set to **hard**. Therefore, it may not be possible to achieve the highest QoR.

> **Note**
>
> Compile points will only have a significant effect on runtime either when used as locked to enable incremental synthesis (and place and route), or else in designs with a large number of repeating structures.

Finite State Machines

The FSM compiler is an automatic tool for encoding state machines. FSM coding style in the RTL design will directly impact performance. By default Synplify Pro implements the following FSM encoding:

- 0-4 states is binary encoded
- 5-40 states is one-hot encoded
- >40 states is Gray encoded

FSM compiler is used to generate better results and to debug state machines.

Generating Better Results

The software uses optimization techniques that are specifically tuned for FSMs such as reachability analysis. The FSM compiler examines the design for state machines, converting them to a symbolic form that provides a better starting point for logic optimization. The FSM compiler may convert an encoded state machine into a different encoding style (to improve speed and area utilization) without changing the source. This optimization can be overridden by choosing a particular encoding style through appropriate synthesis attributes in the RTL design.

Debugging the State Machines

State machine description errors can result in unreachable states. The user can also use the FSM viewer to see a high-level bubble diagrams and cross-probe from the diagram with respect to RTL. The user can then check whether the source code describes the state(s) correctly.

FSM Encoding

There are two choices to define the encoding via attributes in the RTL code:

- Use `syn_encoding` attribute and enable the FSM compiler.
- Use `syn_enum_encoding` to define the states (sequential, one-hot, gray, and safe) and disable the FSM compiler. If the user does not disable the FSM compiler, the `syn_enum_encoding` values are not implemented. This behavior is because the FSM compiler, which is a mapper operation, overrides any user attributes for the FSM encoding. The FSM compiler can be disabled via the GUI or the from the Synplify Pro project file with the following syntax:

```
set_option -symbolic_fsm_compiler 0
```

The user may also direct the synthesis process to deploy a user-defined FSM encoding, for example:

```
attribute syn_enum_encoding of state_type: type is "001 010 101" ;
```

There is a synthesis attribute to turn on/off FSM extraction. By using this attribute the user can see how state machines are extracted. The attributes is set in the source code as follows:

- Specify a state machine for extraction and optimization – `syn_state_machine=1`
- Prevent state machines from being extracted and optimized – `syn_state_machine=0`

In VHDL

```
------ Attribute ----

attribute syn_state_machine : boolean;
attribute syn_state_machine of tx_training_cstate : signal is true;
```

In Verliog

If user does not want to optimize the state machine, add the `syn_state_machine` directive to the registers in the Verilog code. Set the value to 0. When synthesized, these registers are not extracted as state machines.

```
reg [39:0] curstate /* synthesis syn_state_machine=0 */ ;
```

For greater than 40 states, Synplify Pro performs Gray encoding. For one-hot encoding, specify the `syn_encoding = "onehot"` as shown below.

```
reg [39:0] state /* synthesis syn_encoding = "onehot" */ ;
```

Replication of States with High Fan-ins

Large and complex state machines present another unique challenge in state machine design. Complex state machines can be made to run faster by actually making them larger by adding more states. This technique can be counter intuitive as the number of levels of logic between the states and not the number of states typically limits state machine performance. The performance of a state machine is limited by both the number of fan-ins into a given state and the decisions made in that state. For example, idle-type states can have a large number of inputs plus increased computational load. With the 6-input LUT architecture of Achronix devices, once the number of fan-ins exceeds six, another level of logic is needed. An easy method to reduce the number of fan-ins is to replicate these states. The duplicated high fan-in states reduce the number of inputs, thus reducing the number of levels of logic.

Both state machines in the figure below are equivalent in function, but State A is duplicated in Version II so that A and A1 have two or less return inputs. As a result, if each state has to deal with four additional inputs, they can now be contained in one 6-input LUT. Although this example is simplistic, the methodology can be applied to larger and more complex state machines.

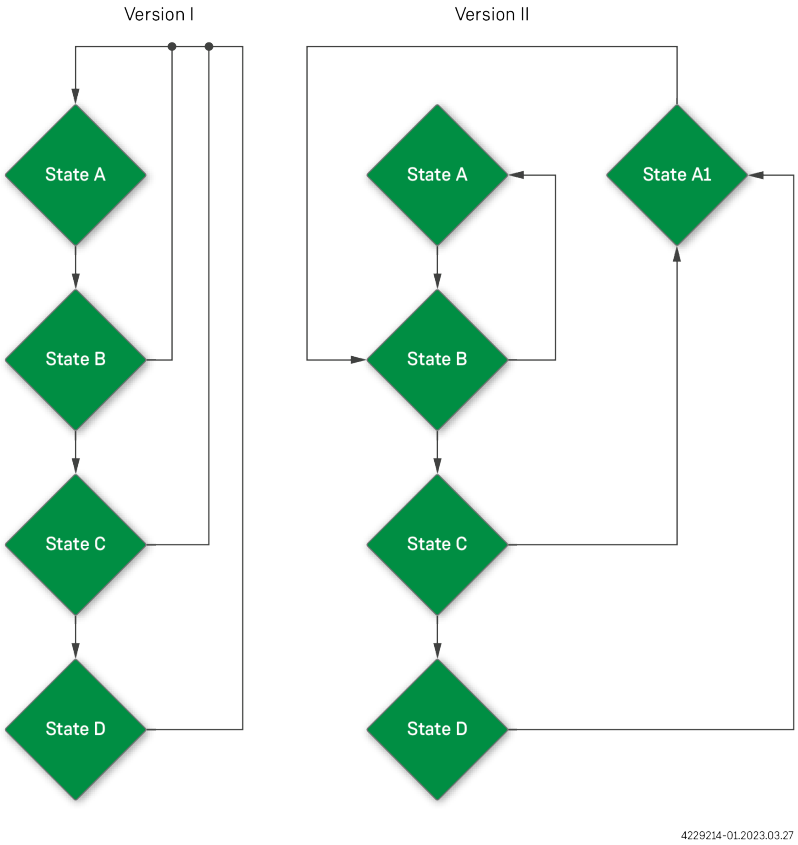


Figure 46 • Replicated High Fan-in State Example

Fanout Limit

This fanout limit can also be controller through RTL design. In this case if the user knows about a net with high fanout and wants to replicate the cell after a certain fanout is reached, the following coding style is needed:

```
wire net1 /* synthesis syn_maxfan = 8 */ ;
```

Here Synplify Pro will infer a buffer/logic if the fanout limit on net1 exceeds 8.

Chapter 10 : Synthesis User Guide Revision History

Version	Date	Description
1.0	17 Jul 2016	• Initial revision. Ported document to Confluence and made it Speedcore specific.
1.1	31 Oct 2016	• Fix for minor type and additional clock constraint example. • Updated document template to include confidentiality note.
1.2	31 Mar 2017	• Corrected one of the create_generated_clock examples in the code block.
1.3	01 Oct 2018	• **Synthesis Optimizations** (page 53): ◦ Corrected the syn_keep attribute in **Example 7** (page 59). ◦ Removed the instantiation templates, referred the user to the *Speedcore IP Component Library User Guide* (UG065). ◦ Added details on **Compile Points.** (page 60) ◦ Updated **DSP64** (page 0) . ◦ Updated **Block RAM** (page 0) . • **Managing Projects in Synplify Pro** (page 27): Removed references to version L-2016 limitations. • Example Synplify-Pro Project File: Removed internal paths from file names.
1.4	10 Jun 2019	• **Synthesis Optimizations** (page 53) : ◦ Removed technology specific entries to make the guide suitable for all technologies. Technology specific parts moved to their appropriate IP Component Library User Guide ◦ Specifically removed inference templates for Speedster16t parts, (DSP64, BRAMTDP & BRAMSDP). • **Managing Projects in Synplify Pro** (page 27): ◦ Combined Speedster and Speedcore differing library files into single Synthesis library include files table. • Example Synplify-Pro Project File: ◦ Added ACE_INSTALL_DIR environment variable to example project file

Version	Date	Description
2.0	20 Jun 2024	• Overview (page 1): Minor correction. • Added major new content for integrated synthesis flows with ACE 10.0 and beyond: ◦ ACE-Driven Integrated Synthesis (page 4) ◦ Synplify-Pro-Driven Integrated Synthesis (page 14) ◦ Stand-Alone Synthesis in Synplify Pro (page 23) ◦ Managing Projects in Synplify Pro (page 27) • Added chapter Synthesis Integration with Multiprocess Option Exploration (page 25)
2.1	20 Aug 2024	• Updated screenshots for ACE 10.1. • Added information on new ACE feature to open Synplify Pro projects in the Synplify Pro GUI from within ACE.
2.2	05 Dec 2024	• Updated screenshots for ACE 10.2 • Updated information on Synplify Pro installation and environment path setup • Updated content for the name change of synthesis project option "Generate Project File" to "ACE-Driven Synthesis"