Synthesis User Guide (UG018)

All Achronix Devices

Copyrights, Trademarks and Disclaimers

Copyright © 2019 Achronix Semiconductor Corporation. All rights reserved. Achronix, Speedcore, Speedster, and ACE are trademarks of Achronix Semiconductor Corporation in the U.S. and/or other countries All other trademarks are the property of their respective owners. All specifications subject to change without notice.

NOTICE of DISCLAIMER: The information given in this document is believed to be accurate and reliable. However, Achronix Semiconductor Corporation does not give any representations or warranties as to the completeness or accuracy of such information and shall have no liability for the use of the information contained herein. Achronix Semiconductor Corporation reserves the right to make changes to this document and the information contained herein at any time and without notice. All Achronix trademarks, registered trademarks, disclaimers and patents are listed at http://www.achronix.com/legal.

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 6
Chapter - 2: Synplify Pro Introduction 7
Creating and Setting up a Project 7
Adding the Synthesis Library Include File 8
Adding Source Files to the Project 9
Implementation Options 10
Verilog 11
Place and Route 12
Timing Report 12
Implementation Results 13
Constraints 14
Options 15
Running Synthesis 16
Chapter - 3: Synthesis Constraints 17
Timing Constraints 17
create_clock 17
create_generated_clock 18
set_clock_groups 19
set_false_path 19
set_input_delay 19
set_output_delay 20
set_max_delay 20
set_multicycle_path 20
set_clock_latency 21
set_clock_uncertainty 21
Non-timing Constraints 22
Compile Points 22
Attributes 22
Constraint Check 23
Chapter - 4: Synthesis Optimizations 24
Preventing Objects from Being Optimized Away 24
Dangling Nets 24

Dangling Sequential Logic 24
Unconnected Instances 24
Prevent ACE Optimizing Objects Away 25
Pipelining 25
Retiming 25
Forward Annotation of RTL Attributes to Netlist 26
Example 1 26
Example 2 26
Example 3 27
Example 4 27
Example 5 28
Example 6 29
Example 7 30
Compile Points 30
Finite State Machines 32
Generating Better Results 32
Debugging the State Machines 32
FSM Encoding 32
Replication of States with High Fan-ins 33
Chapter - 5: Example Synplify-Pro Project File 35
Revision History 37

Chapter - 1: Overview

This user guide describes how to use Synplify Pro from Synopsys to synthesize a design and generate a netlist for implementation in an Achronix Speedcore instance. Suggested optimization techniques are also included.

Synplify-Pro reads in standard RTL and outputs a mapped netlist (`.vm`) which is used by the ACE tool suite. A high-level overview of the Achronix design flow is shown in figure below.

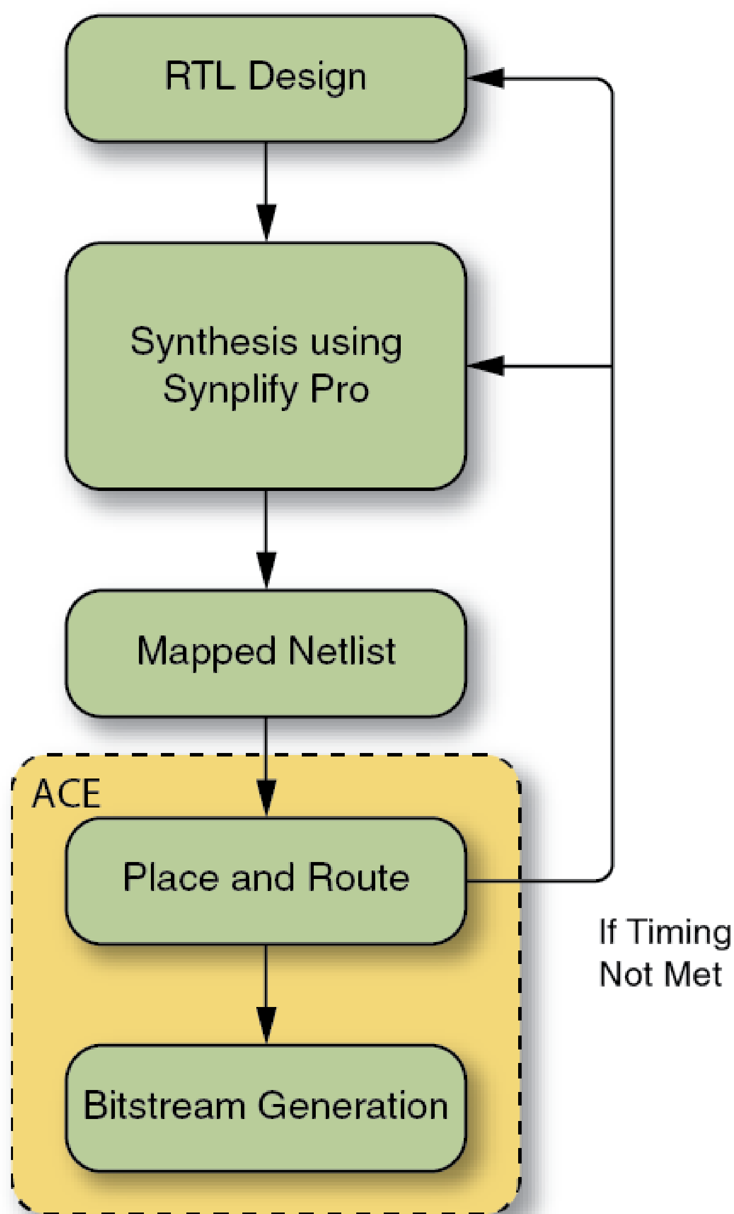

Figure 1: ***Achronix Synthesis Design Flow***

Chapter - 2: Synplify Pro Introduction

This guide assumes that Synplify Pro is installed with the **synplify_pro** command added to the $PATH. The examples in this guide uses the Linux version of the software; the Windows version of Synplify Pro has the same options.

Creating and Setting up a Project

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

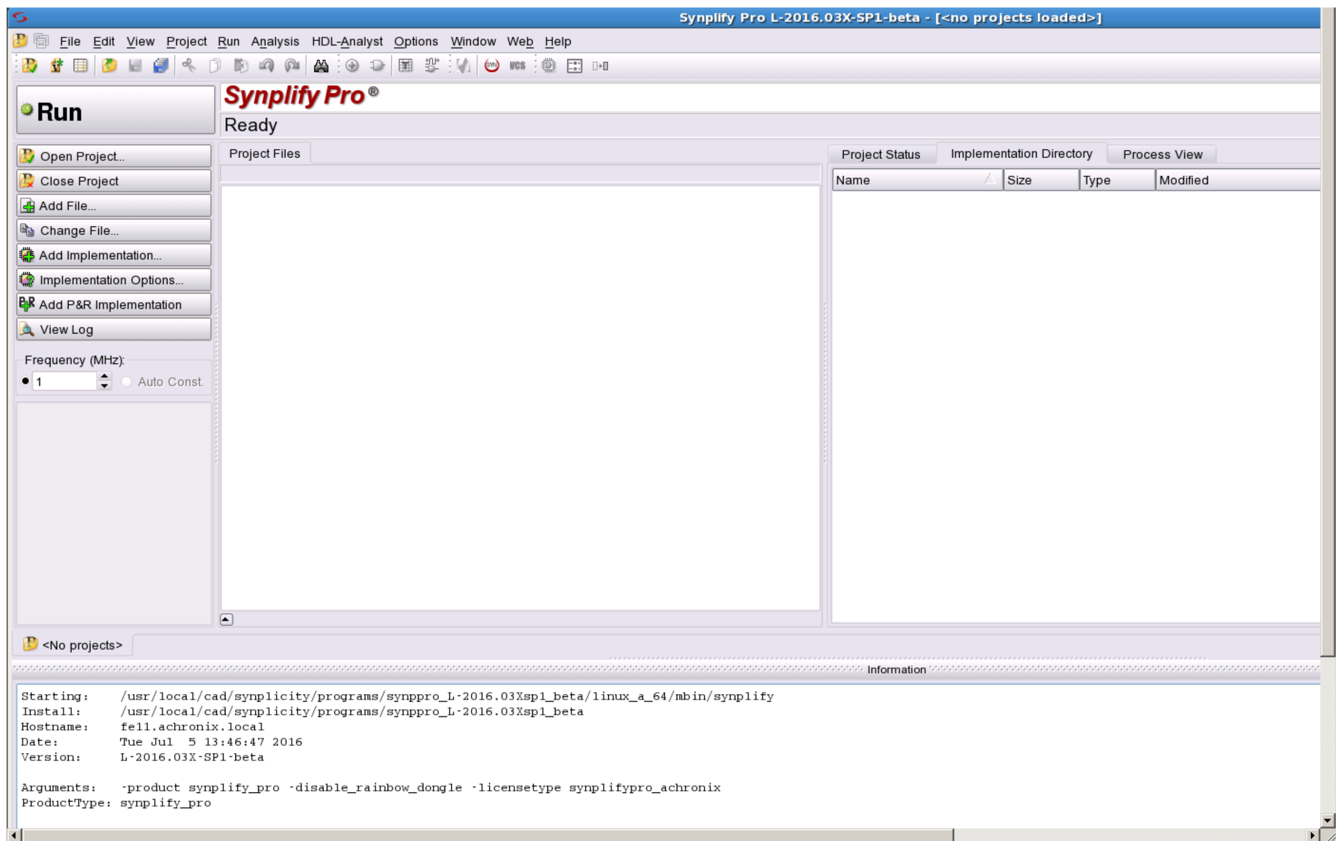

Figure 2: ***Synplify Pro Invoked from the Command Shell***

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

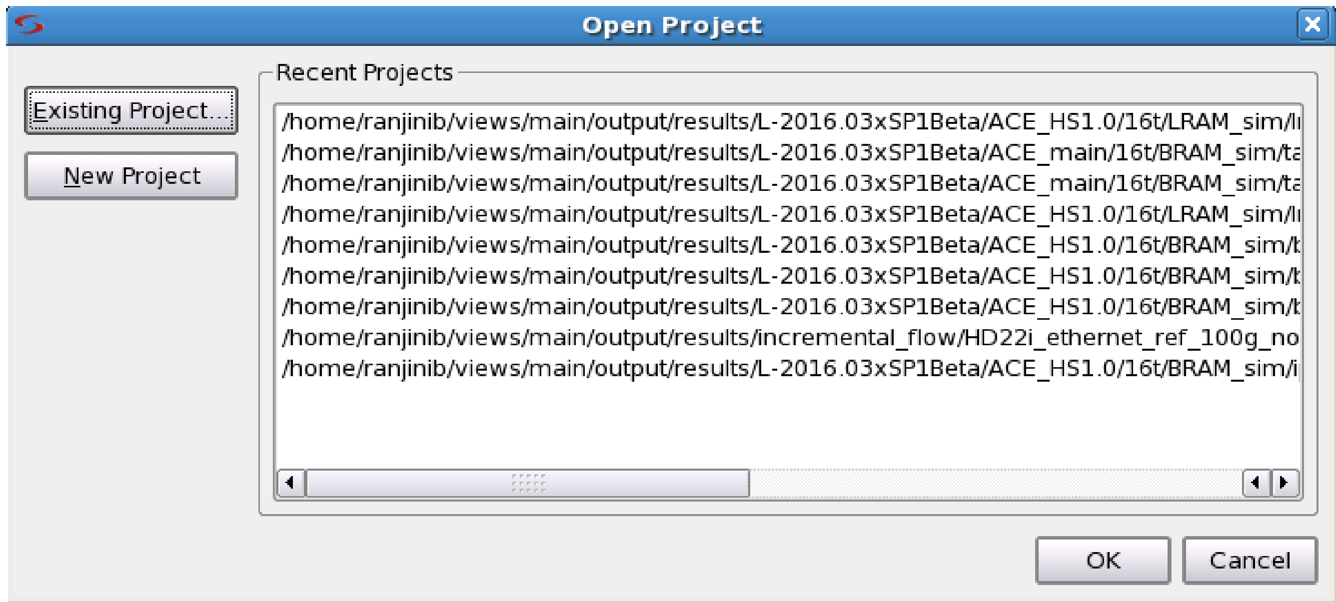

Figure 3: ***Dialog Box to Select the New Project***

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

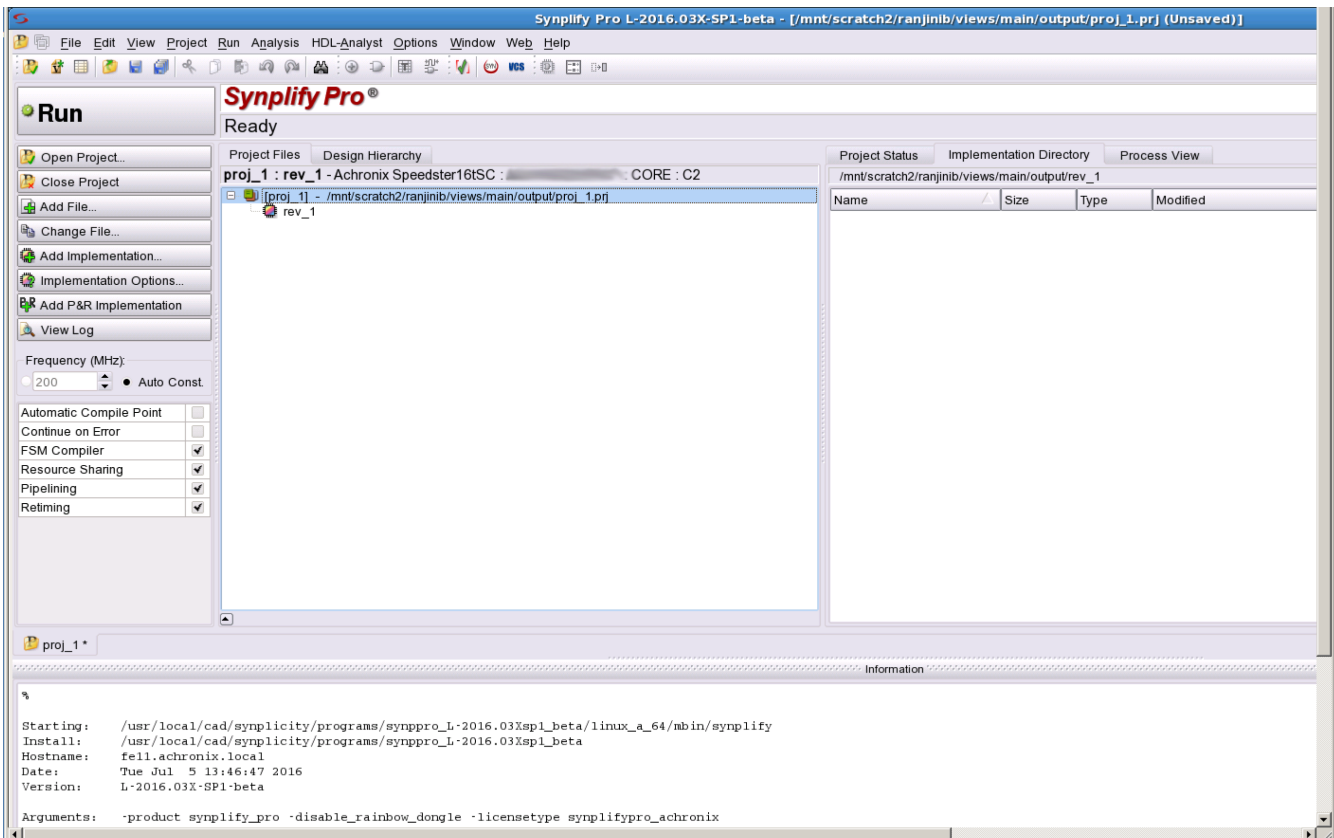

Figure 4: ***Starting a New Project***

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. The file to be included varies according to the target device technology; the respective files are listed in Synthesis library include files table (see page 9) below. All the synthesis library includes files are located in the directory `<ACE_INSTALL_DIR>/libraries/device_models/`.

Table 1: *Synthesis Library Include Files*

Technology	Library file
HD1000 22nm	22i_synplify.v
Speedcore 16t	16t_synplify.v
Speedster 7t	7t_synplify.v
Speedcore 7t	7t_synplify.v

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 Speedcore 16t technology has been selected, so the file `16t_synplify.v` 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.

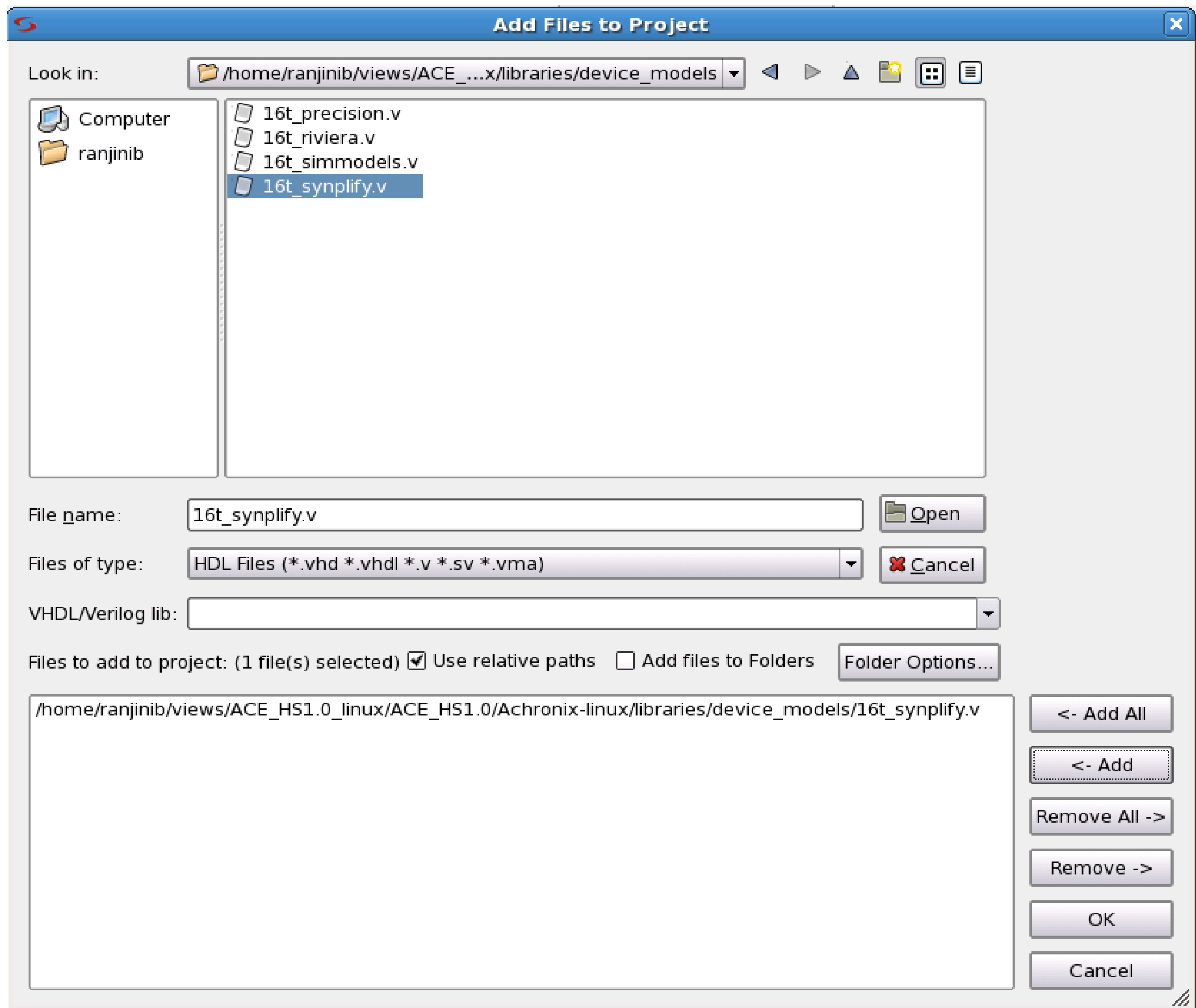

Figure 5: *Add Files to Project*

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.

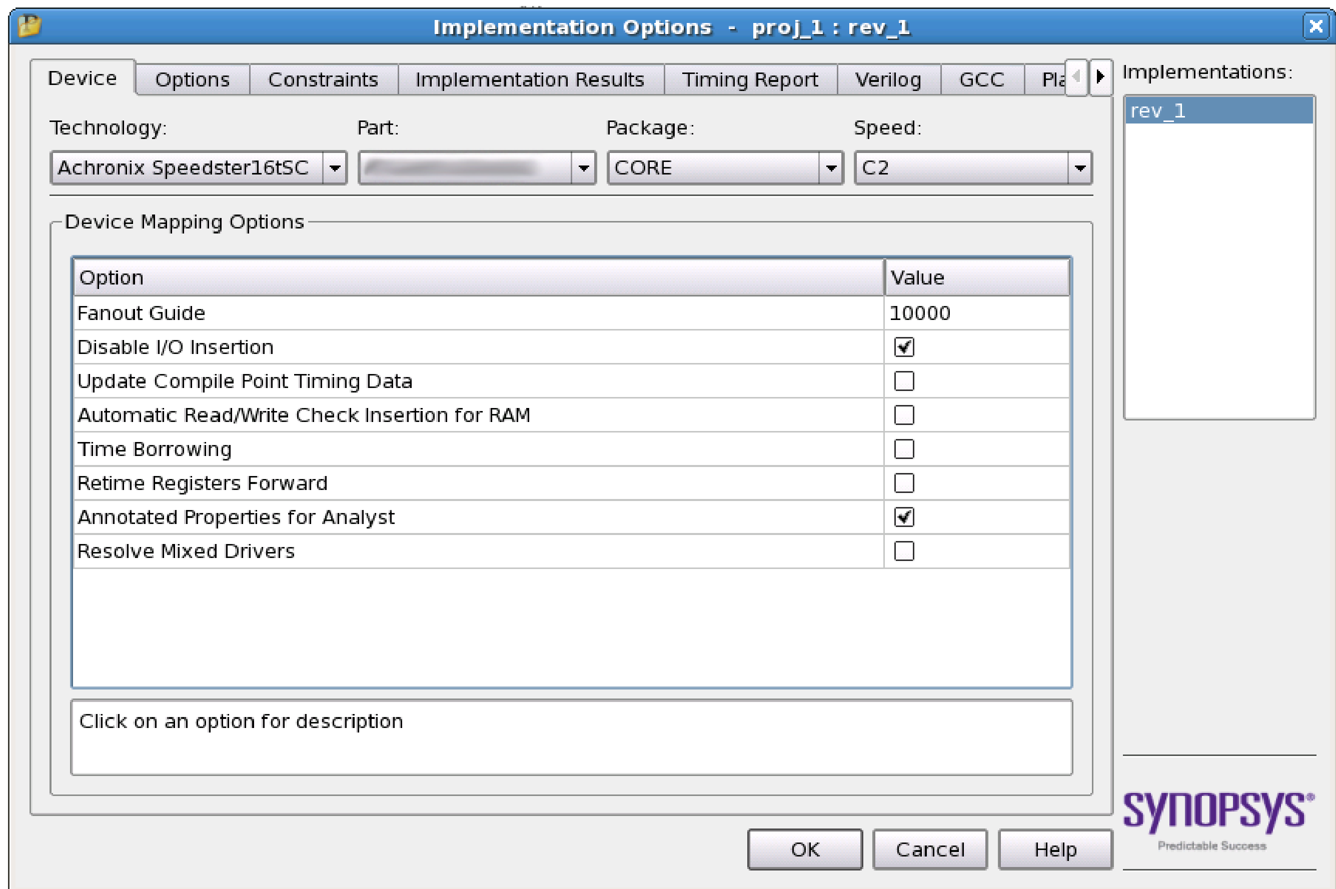

Figure 6: ***Implementation Options***

> **Note**
>
> If using a Speedcore device, ensure the **Disable I/O Insertion** option is checked as shown. If using a Speedster device, then this option must be disabled..

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.

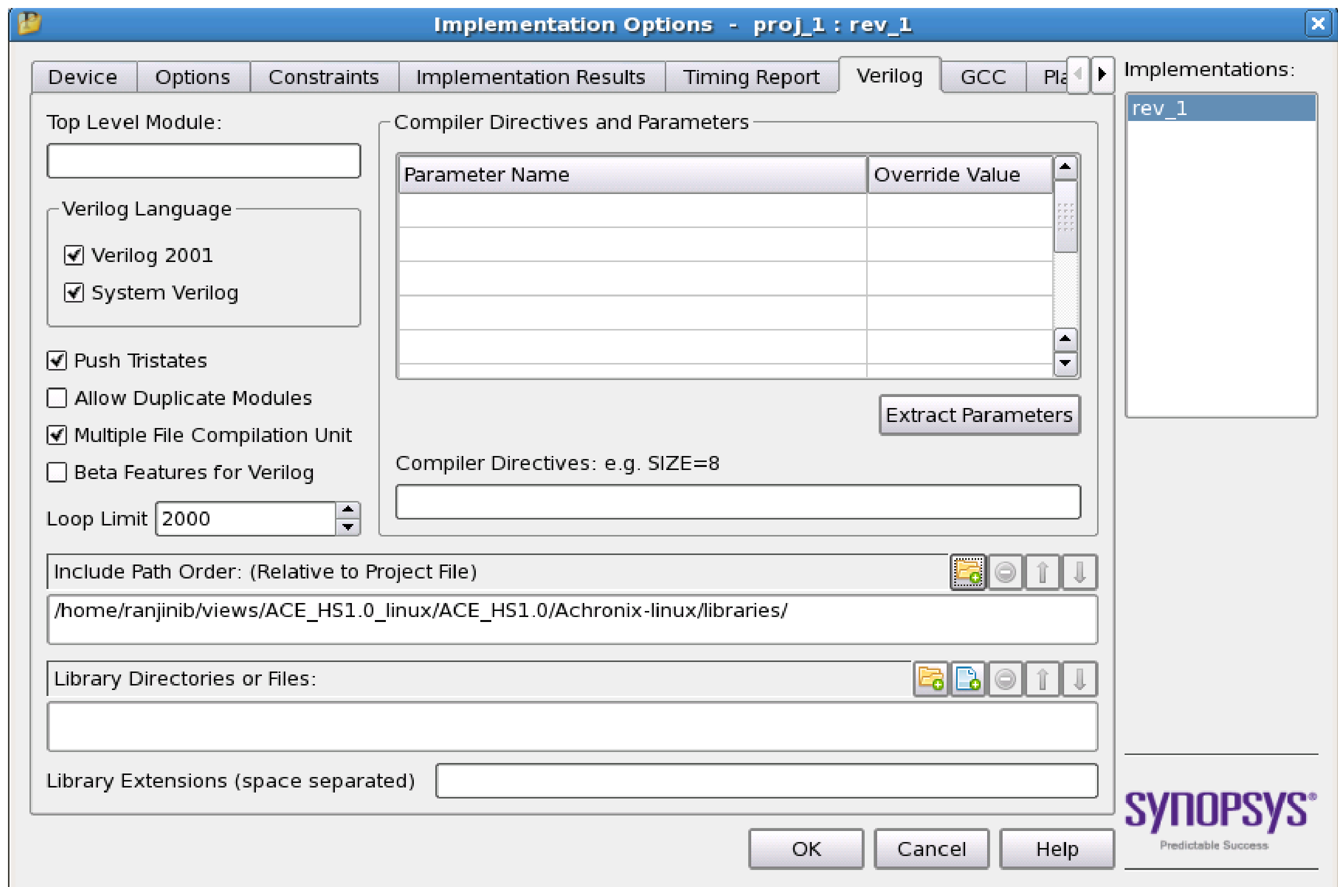

Figure 7: *Implementation Options: Include Path Order.*

Place and Route

This tab is not presently utilized by the Achronix back-end tool (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.

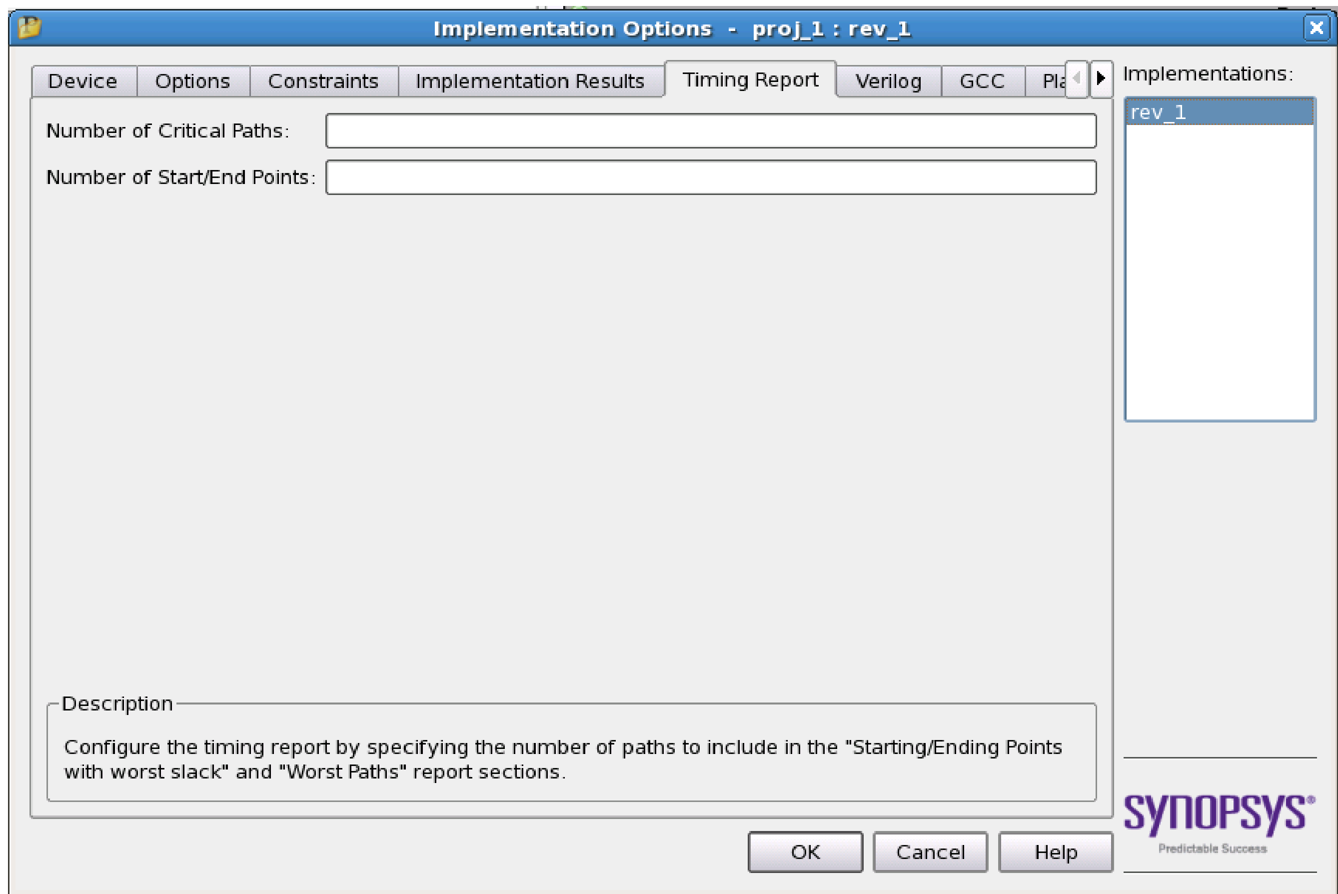

Figure 8: ***Implementation Options: Timing Report***

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.

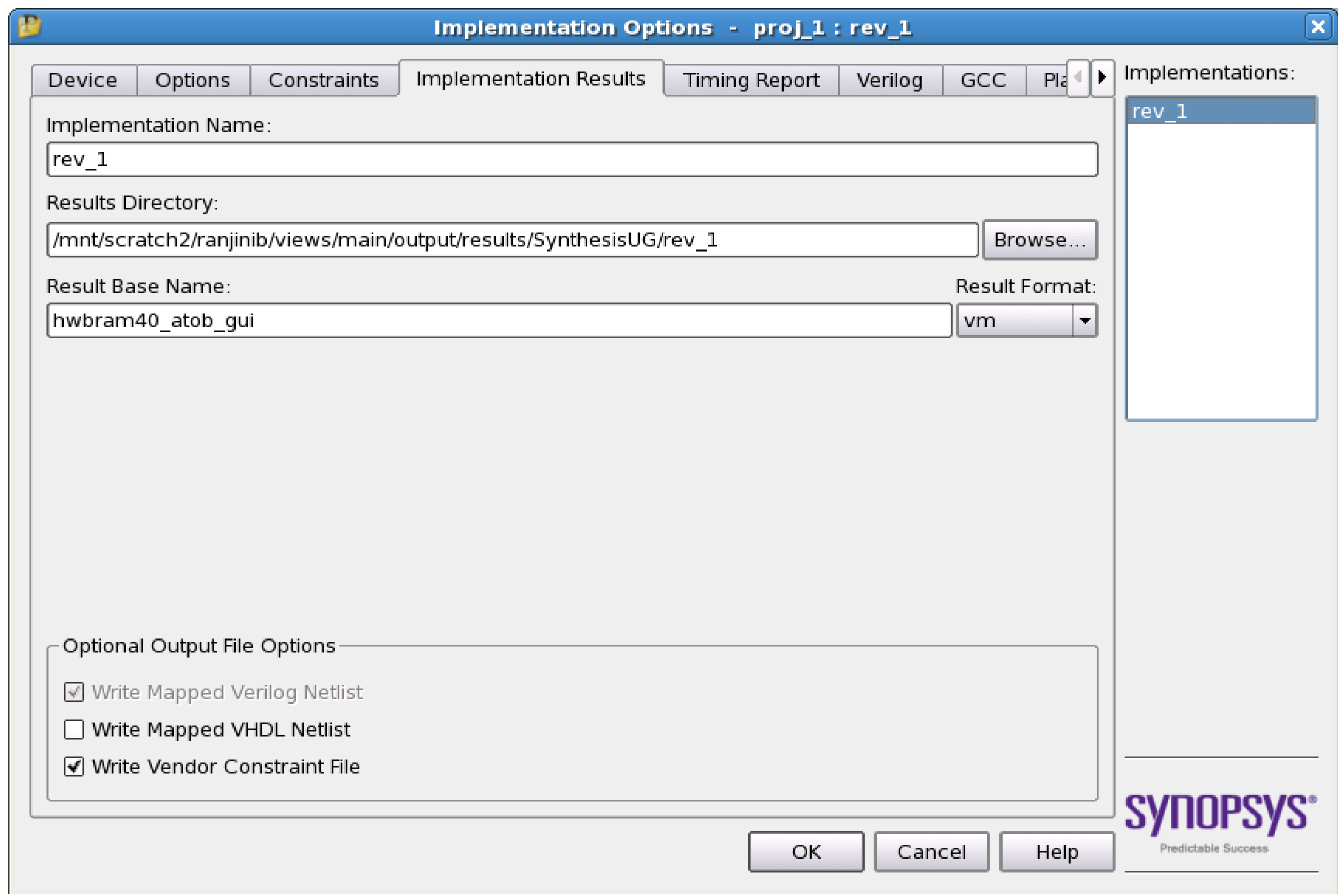

Figure 9: ***Implementation Options: Implementation Results***

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 (see page 17).

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).

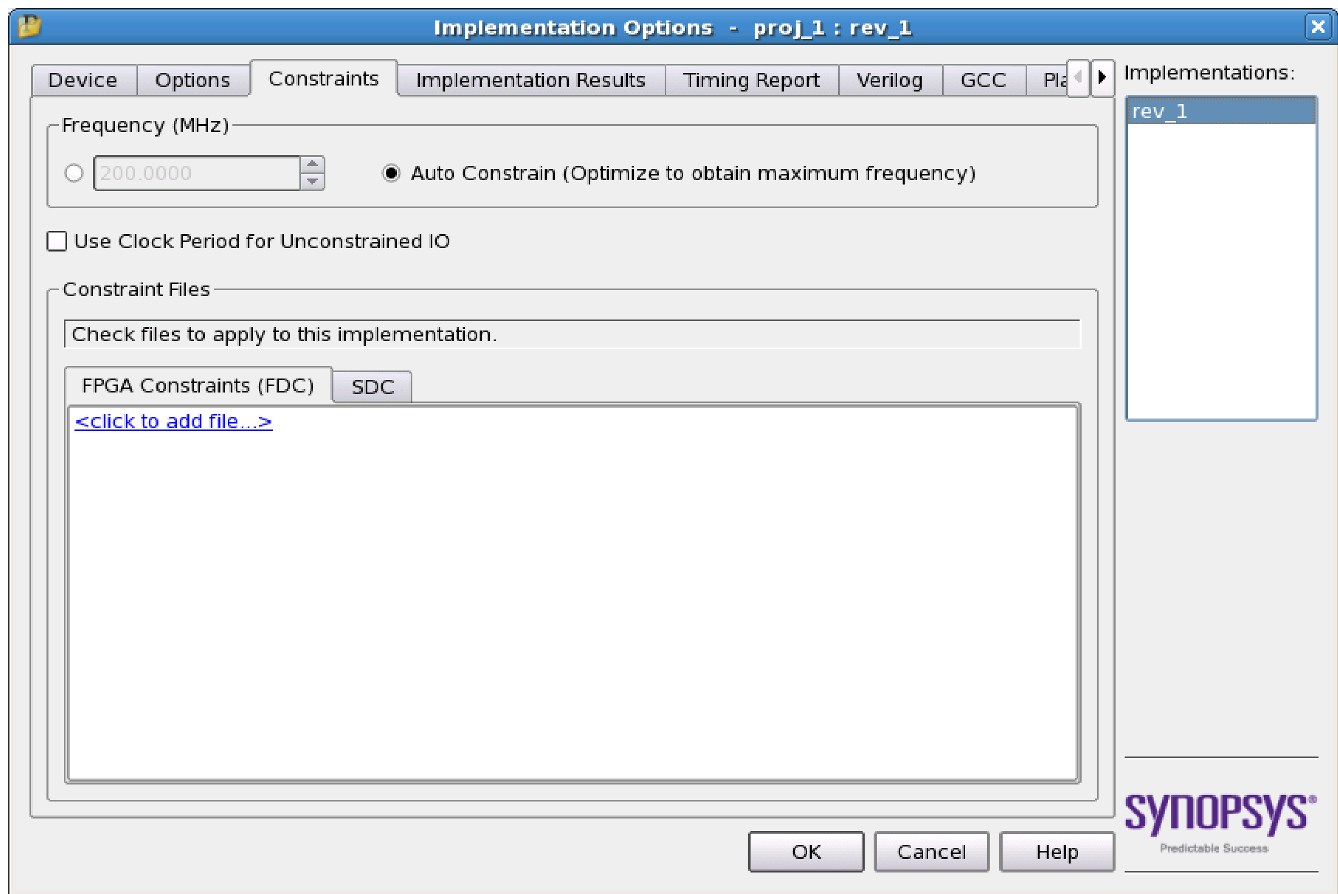

Figure 10: ***Implementation Options: Constraints***

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.

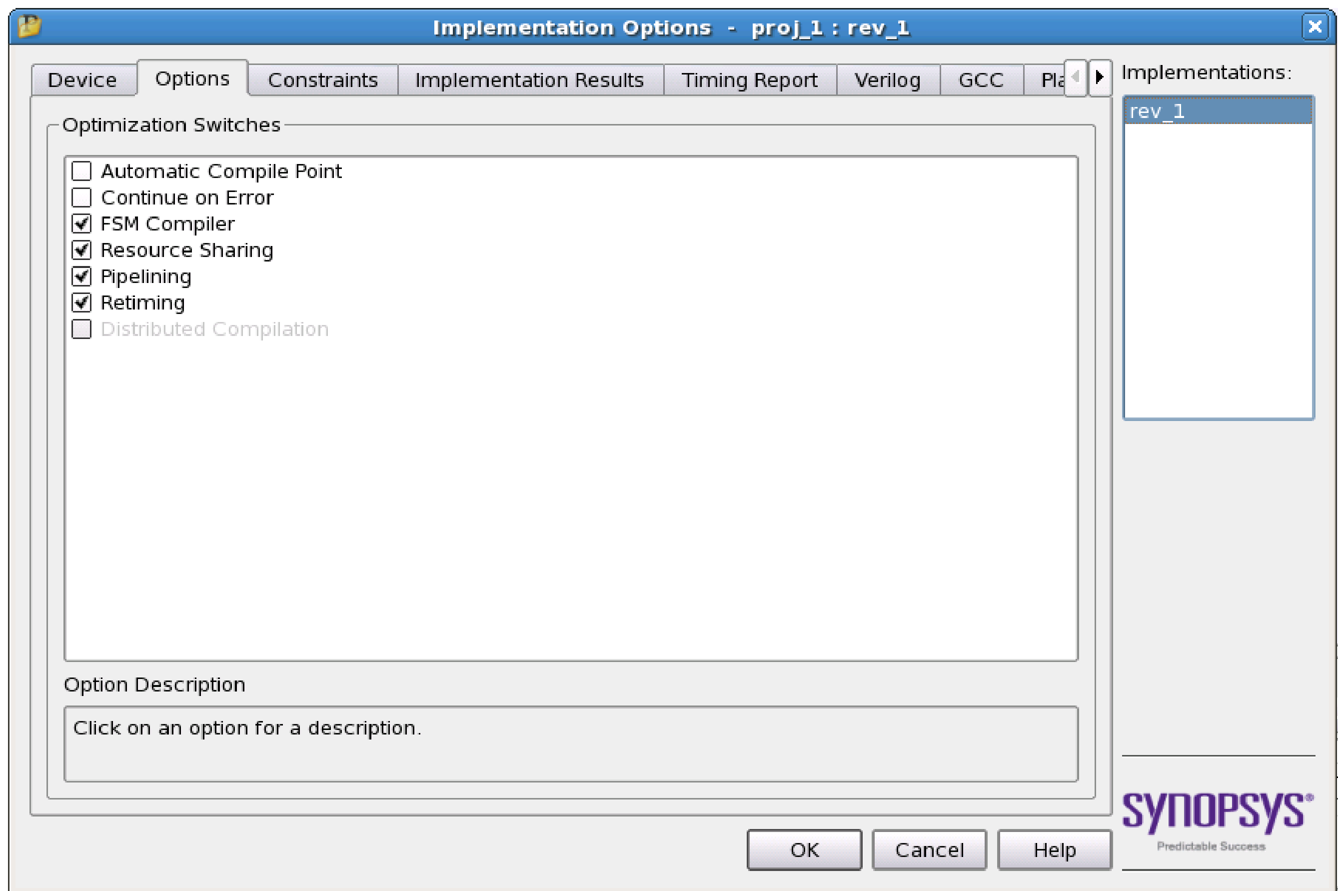

Figure 11: ***Implementation Options: Options***

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.

Chapter - 3: Synthesis Constraints

Synplify 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 both 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. Use 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 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 2: ***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_ddr3xN_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 (see page 30).

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 can be defined both globally and also applied to individual instances.

Enable a wide MUX option in Speedster16t technology, enter:

```
define_global_attribute  {syn_acx_mux41_opt} {1}
```

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}
```

Constraint Check

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.

Constraint Check is launched using **Run** → **Constraint Check.**

Chapter - 4: Synthesis Optimizations

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

- Hanging nets
- Pipelining
- Retiming
- Forward annotation of RTL attributes to netlist
- Compile points
- Finite state machines

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 keep these nets. In order for these nets not to get optimized away (removed), add the following directive to the RTL, In this example, the synthesis tool does 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

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

Retiming

When this switch is enabled, the synthesis tool tries to improve the timing performance of sequential circuits. The retiming process moves storage devices (flip-flops) across computational elements with no memory (only gates /LUTs) to improve the performance of the circuit. This option also adds a retiming report to the log file. 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. Pipelining is automatically enabled when retiming is enabled.

Forward Annotation of RTL Attributes to 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 tool 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 get propagated to the netlist
Instantiated Components	syn_noprune	Attribute applied on the instantiated component will get propagated to the netlist
Input / Output ports	syn_hier="hard" on the module containing the ports	Attribute applied on ports get propagated to the inpt /output port in the netlist
Registers	syn_preserve	Attribute applied on the registers will get propagated to the netlist
wire	syn_keep	Attribute applied on nets/wires will get propagated to the netlist

Below are some examples:

Example 1

The attribute weight="3.0" gets propagated 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, comma is not required after syn_preserve=1. When using Verilog 2001 style 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 `ace_useioff`. The `ace_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 ace_useioff=1 */;
input  wire [3:0] inb  /* synthesis ace_useioff=0 */;
input  wire       sel  /* synthesis ace_useioff=1 */;
input  wire       clk;
output reg        z0   /* synthesis ace_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 so 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 ace_useioff attribute could also be specified in the Verilog 2001 comment style. For example:

```
(* ace_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, so in that case it will not forward annotate the attribute into 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 a Speedcore device boundary pins.

This example illustrates the application of the `ace_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 ace_useioff = 0 */;
(* syn_keep=1 *) wire ipin_dout           /* synthesis ace_useioff = 0 */;
                 wire dff1_q, dff2_q;

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

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

For full details on all the options for flop pushing, see the section Automatic Flop Pushing into I/O Pads in the *ACE User Guide* (UG001).

Note

As in Example 7, the `ace_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 in the Synplify Pro tool for details on compile points.

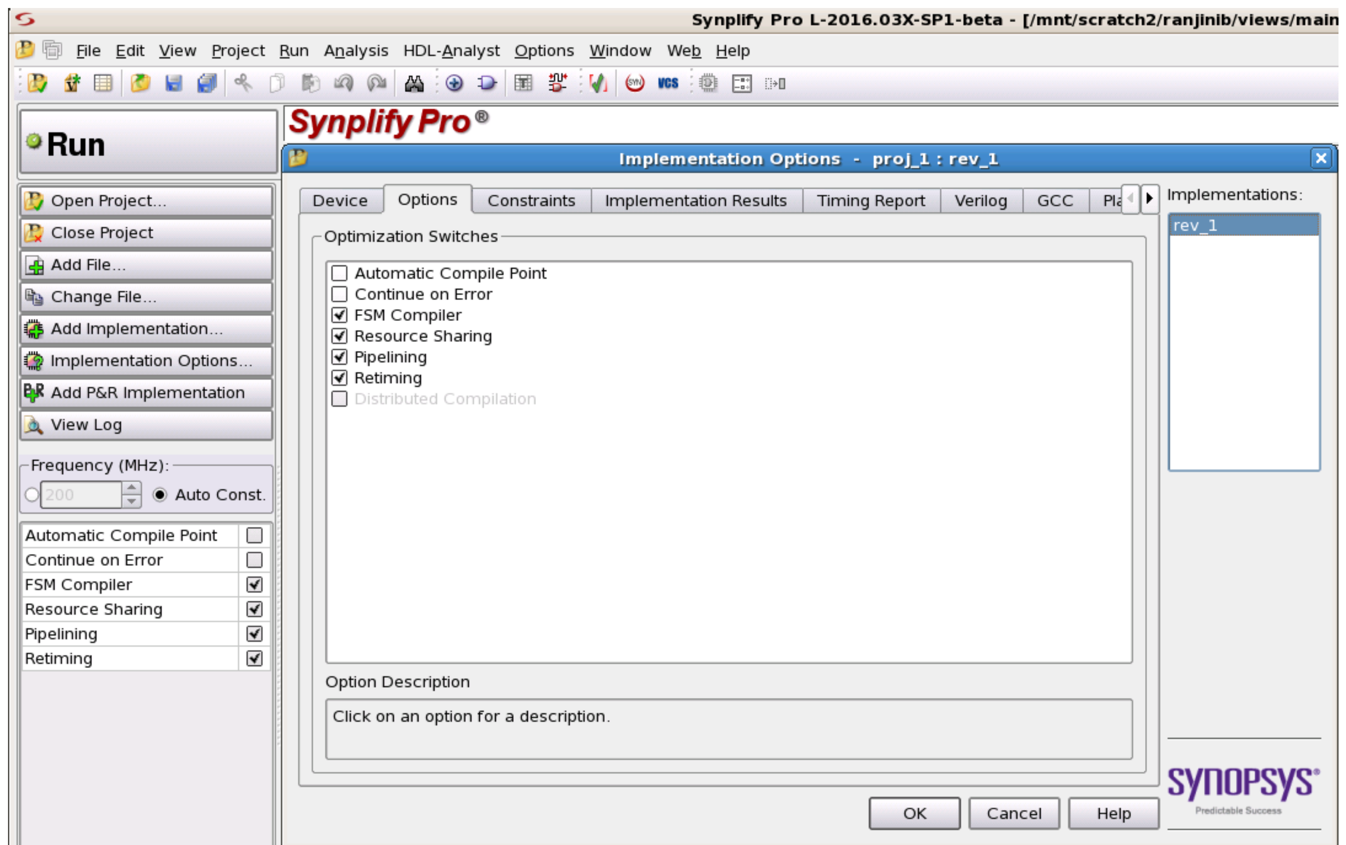

Figure 12: *Setting Compile Points*

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 synthesise that block once. The level of optimisation 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 what 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 runtimes 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 overriden 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 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 fanins 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 4input LUT architecture of Speedcore devices, once the number of fanins exceeds four, another level of logic is needed. An easy method to reduce the number of fanins is to replicate these states. The duplicated high fanin 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 two additional inputs, they can now be contained in one 4input LUT. Although this example is simplistic, the methodology can be applied to larger and more complex state machines.

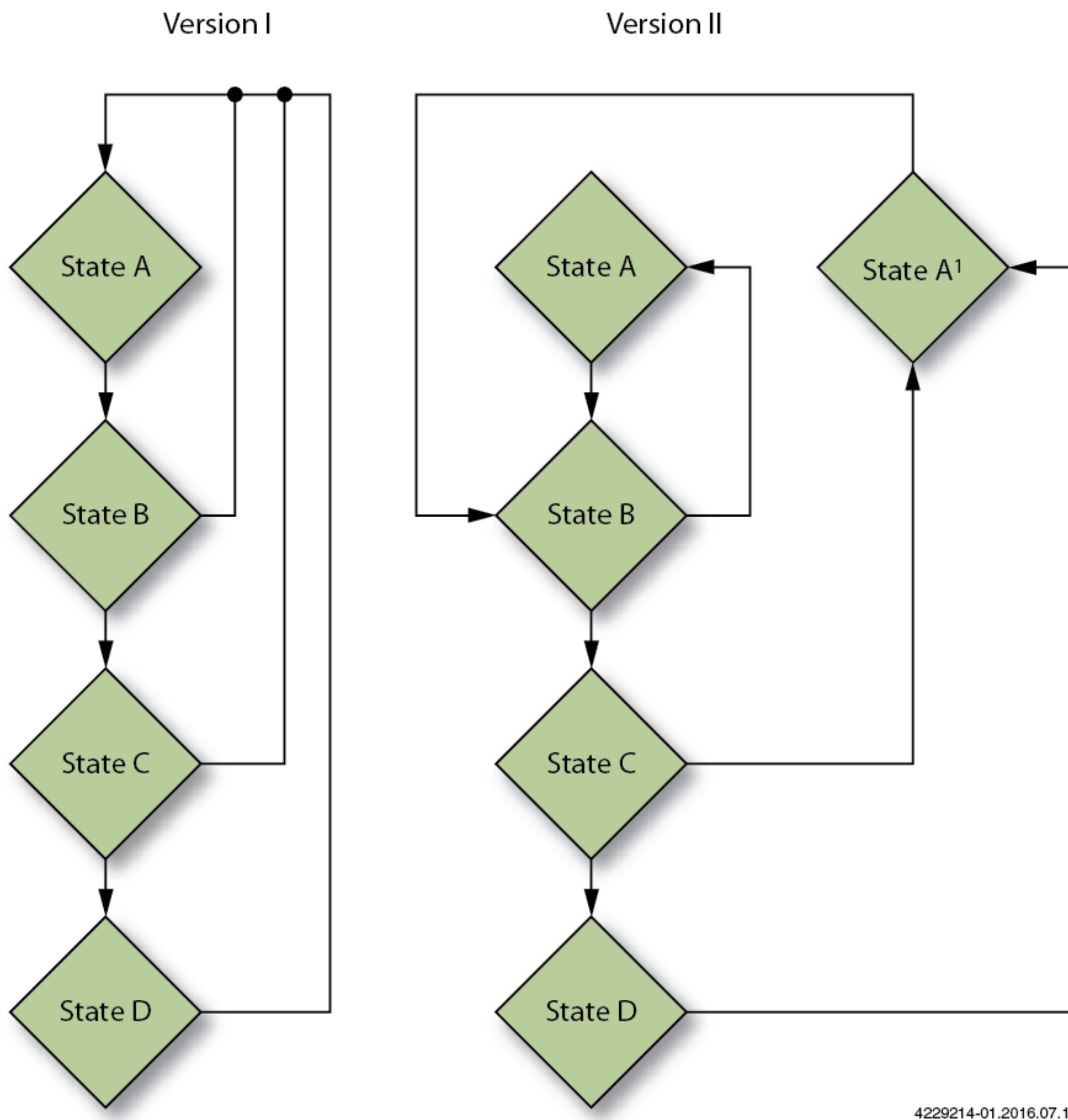

Figure 13: ***Replicated High Fan-in State Example***

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 - 5: Example Synplify-Pro Project File

```
#--  Synopsys, Inc.
#--  Version O-2019.09X
#--  Project file output/rev_1/hwbram40_atob_gui.prj
#--  Written on Tue Jul  5 17:20:52 2018

# Get the ACE installation directory from the environment variable
# This allows for portable projects
# Note : If Synplify is used to save the project file, this line will be removed
# When saving with synplify, copy this line first, then repaste in the project file
# and update the XX_synpliy.v path.
set ACE_INSTALL_DIR $::env(ACE_INSTALL_DIR)

#project files
# Synplify include file for ACE
add_file -verilog "$ACE_INSTALL_DIR/Achronix-linux/libraries/device_models/XX_synplify.v"
# Verilog file
add_file -verilog "../src/rtl/bram40_atob.v"
# SystemVerilog file
add_file -verilog -vlog_std sysv "../src/rtl/hwbram40_atob_gui.sv"
# Constraint file
add_file -constraint "../src/constraints/synplify_constraints.sdc"


#implementation: "rev_1"
impl -add rev_1 -type fpga

#implementation attributes
set_option -vlog_std sysv
set_option -project_relative_includes 1
set_option -include_path {$ACE_INSTALL_DIR/libraries/;../src/rtl/include}

#device options
set_option -technology <Technology Family>
set_option -part <Device_Name>
set_option -package <Device Package>
set_option -speed_grade <Device Speed Grade>
set_option -part_companion ""

#compilation/mapping options

# hdl_compiler_options
set_option -distributed_compile 0

# mapper_without_write_options
set_option -frequency auto
set_option -srs_instrumentation 1

# mapper_options
set_option -write_verilog 1
set_option -write_vhdl 0
set_option -maxfan 10000
set_option -rw_check_on_ram 0
```

```
#---------------------------------------------------
# IO pad insertion, varies according to target device
#---------------------------------------------------
# For Speedcore
set_option -disable_io_insertion 1
# For Speedster
set_option -disable_io_insertion 0

set_option -retime_registers_forward 0
set_option -pipe 1
set_option -retiming 1
set_option -update_models_cp 0
set_option -run_prop_extract 1
set_option -fix_gated_and_generated_clocks 1

# NFilter
set_option -no_sequential_opt 0

# sequential_optimization_options
set_option -symbolic_fsm_compiler 1

# Compiler Options
set_option -compiler_compatible 0
set_option -resource_sharing 1
set_option -multi_file_compilation_unit 1

# Compiler Options
set_option -auto_infer_blackbox 0

#automatic place and route (vendor) options
set_option -write_apr_constraint 1

#set result format/file last
project -result_file "rev_1/hwbram40_atob_gui.vm"

impl -active "rev_1"
```

Note

The device specific values are set from the part selected from the drop-down menu of the Implementation Options Dialog Box.

ACE_INSTALL_DIR is a local environment variable that is the path to the local ACE installation.

Revision History

Version	Date	Description
1.0	July 17, 2016	• Initial revision. Ported document to Confluence and made it Speedcore specific.
1.1	October 31, 2016	• Fix for minor type and additional clock constraint example. • Updated document template to include confidentiality note.
1.2	March 31, 2017	• Corrected one of the create_generated_clock examples in the code block.

1.3	October 1, 2018	• Synthesis Optimizations (see page 24):
		  ◦ Corrected the syn_keep attribute in Example 7 (see page 30).
		  ◦ Removed the instantiation templates, referred the user to the *Spe edcore IP Component Library User Guide* (UG065).
		  ◦ Added details on Compile Points. (see page 30)
		  ◦ Updated DSP64 (see page).
		  ◦ Updated Block RAM (see page).
		• Synplify Pro Introduction (see page 7): Removed references to version L-2016 limitations.
		• Example Synplify-Pro Project File (see page 35): Removed internal paths from file names.

1.4	June 10, 2019	• Synthesis Optimizations (see page 24):
		  ◦ 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).
		• Synplify Pro Introduction (see page 7):
		  ◦ Combined Speedster and Speedcore differing library files into single Synthesis library include files table (see page 9).
		• Example Synplify-Pro Project File (see page 35):
		  ◦ Added ACE_INSTALL_DIR environment variable to example project file