

This chapter documents support for the Synopsys Synplify software in the Quartus® II software, as well as key design flows, methodologies, and techniques for achieving good results in Altera® devices.

## Introduction

This chapter includes the following topics:

- General design flow with the Synplify and Quartus II software
- Synplify software optimization strategies, including timing-driven compilation settings, optimization options, and Altera-specific attributes
- Exporting designs and constraints to the Quartus II software using NativeLink integration
- Guidelines for Altera megafunctions and library of parameterized module (LPM) functions, instantiating them with the MegaWizard™ Plug-In Manager, and tips for inferring them from hardware description language (HDL) code
- Incremental compilation and block-based design, including the MultiPoint flow in the Synplify Pro and Synplify Premier software

The content in this chapter applies to the Synplify, Synplify Pro, and Synplify Premier software unless otherwise specified. This chapter includes the following sections:

- “Altera Device Family Support”
- “Design Flow” on page 10–2
- “Synplify Optimization Strategies” on page 10–6
- “Exporting Designs to the Quartus II Software Using NativeLink Integration” on page 10–14
- “Guidelines for Altera Megafunctions and Architecture-Specific Features” on page 10–25
- “Incremental Compilation and Block-Based Design” on page 10–37

This chapter assumes that you have set up, licensed, and are familiar with the Synplify software.

## Altera Device Family Support

The Synplify software maps synthesis results to Altera device families. The following list shows the Altera device families supported by the Synplify software version C-2009.06 SP1, with the Quartus II software version 9.1:

- Arria® series
- Cyclone®, series
- HardCopy® series

- MAX® series
- Stratix® series

The Synplify software also supports the FLEX 8000 and MAX 9000 legacy devices that are supported only in the Altera MAX+PLUS® II software, as well as ACEX® 1K, APEX™ II, APEX 20K, APEX 20KC, APEX 20KE, FLEX® 10K, and FLEX 6000 legacy devices that are supported by the Quartus II software version 9.0 and earlier.



To learn about new device support for a specific Synplify version, refer to the release notes at [www.synopsys.com](http://www.synopsys.com). Support for newly released device families may require an overlay.

## Design Flow

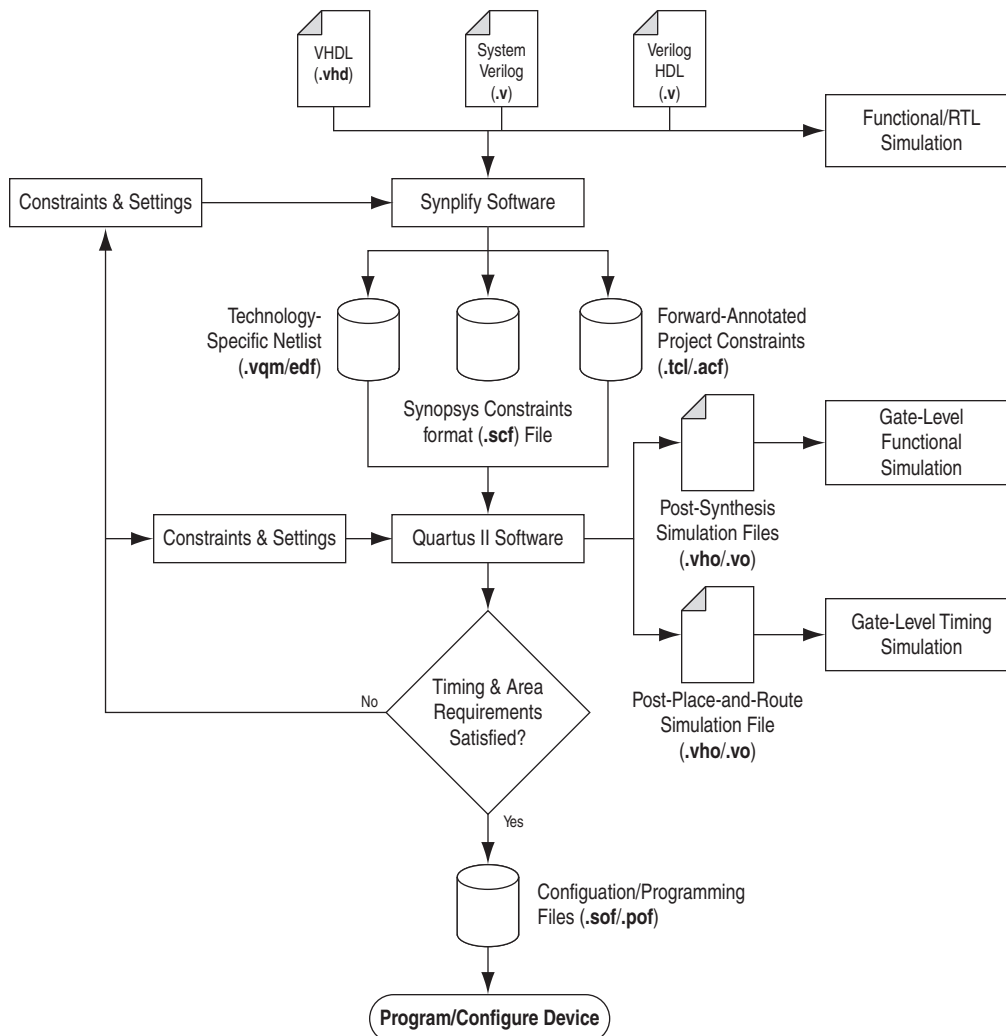
A Quartus II software design flow using the Synplify software consists of the following steps:

1. Create Verilog HDL or VHDL design files.
2. Set up a project in the Synplify software and add the HDL design files for synthesis.
3. Select a target device and add timing constraints and compiler directives in the Synplify software to optimize the design during synthesis.
4. Run synthesis in the Synplify software.
5. Create a Quartus II project and import these files generated by the Synplify software into the Quartus II software. These files are used for placement and routing, and for performance evaluation.
  - The technology-specific Verilog Quartus Mapping File (**.vqm**) netlist or EDIF (**.edf**) netlist for legacy devices also supported by the MAX+PLUS II software
  - The Synopsys Constraints Format (**.scf**) file for TimeQuest timing constraints
  - The Tcl constraints file (**.tcl**)

Alternatively, you can run the Quartus II software from within the Synplify software. For more detailed information, refer to “[Running the Quartus II Software from within the Synplify Software](#)” on page 10-14.
6. After obtaining place-and-route results that meet your requirements, configure or program the Altera device.

Figure 10-1 shows the recommended design flow when using the Synplify and the Quartus II software.


Figure 10-1. Recommended Design Flow



The Synplify software supports VHDL, Verilog HDL, and SystemVerilog source files. However, only the Synplify Pro and Premier software supports mixed synthesis, allowing a combination of VHDL and Verilog HDL or SystemVerilog format source files.

Specify timing constraints and attributes for a design in a SCOPE Design Constraints File (.sdc) with the SCOPE window in the Synplify software using standard Synopsys Design Constraint (SDC) format, or directly in the HDL source file. Compiler directives can also be defined in the HDL source file. Many of these constraints are forward-annotated for use by the Quartus II software. (See Table 10-1 for a list of the files generated by Synplify.)

The HDL Analyst that is included in the Synplify software is a graphical tool for generating schematic views of the technology-independent RTL view netlist (.srs) and technology-view netlist (.srm) files. You can use the Synplify HDL Analyst to analyze and debug your design visually. The HDL Analyst supports cross probing between the RTL and Technology views, the HDL source code, and the Finite State Machine (FSM) viewer. The HDL Analyst also supports cross-probing between the technology view and the timing report file in the Quartus II software.


 A separate license file is required to enable the HDL Analyst in the Synplify software. The Synplify Pro and Premier software include the HDL Analyst.

After synthesis is completed, import the .vqm or .edf netlist to the Quartus II software for place-and-route. Use the .tcl file generated by the Synplify software to forward-annotate your constraints (including device selection) and optionally to set up your project in the Quartus II software.


If you select a Stratix III, Cyclone III, Arria GX, or newer device, the Quartus II software uses the SDC-format timing constraints from the .scf file with the TimeQuest Timing Analyzer by default. If you select a Stratix II or Stratix II GX device, you have the option to switch from the Classic Timing Analyzer to the TimeQuest Timing Analyzer by turning on the **Use TimeQuest Timing Analyzer** option in the **Device** tab in the **Implementation Options** dialog box in the Synplify software. For older devices, the Quartus II software uses the Tcl-format timing constraints from the Quartus Setting File (.qsf) with the Classic Timing Analyzer. Refer to [“Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File”](#) on page 10-16 for information about manually changing from the TimeQuest Timing Analyzer to the Classic Timing Analyzer in the Quartus II software.


If the area and timing requirements are satisfied, use the files generated by the Quartus II software to program or configure the Altera device. As shown in [Figure 10-1](#), if your area or timing requirements are not met, you can change the constraints in the Synplify software or the Quartus II software and repeat the synthesis. Altera recommends that you provide the timing constraints in the Synplify software and the placement constraints in the Quartus II software. Repeat the process until the area and timing requirements are met.

While you can perform simulation at various points in the process, you can also perform final timing analysis after placement and routing is complete. You can also perform formal verification at various stages of the design process.

 For more information about how the Synplify software supports formal verification, refer to [Section III. Formal Verification](#) in volume 3 of the *Quartus II Handbook*.

You can also use other options and techniques in the Quartus II software to meet area and timing requirements. One such option is called WYSIWYG Primitive Resynthesis, which can perform optimizations on your .vqm netlist within the Quartus II software.

 For information about netlist optimizations, refer to the [Netlist Optimizations and Physical Synthesis](#) chapter in volume 2 of the *Quartus II Handbook*.

 In some cases, you might be required to modify the source code if the area and timing requirements cannot be met using options in the Synplify and Quartus II software.

After synthesis, the Synplify software produces several intermediate and output files. Table 10–1 lists these file types.

**Table 10–1.** Synplify Intermediate and Output Files

File Extensions	File Description
<b>.srs</b>	Technology-independent RTL netlist that can be read only by the Synplify software.
<b>.srm</b>	Technology view netlist.
<b>.srr (1)</b>	Synthesis Report file.
<b>.vqm/.edf</b>	Technology-specific netlist in <b>.vqm</b> or <b>.edf</b> file format. An <b>.edf</b> file is created for devices supported by the MAX+PLUS II software. A <b>.vqm</b> file is created for all other Altera device families.
<b>.tcl</b>	Forward-annotated constraints file containing constraints and assignments. A <b>.tcl</b> file for the Quartus II software is created for all devices. The <b>.tcl</b> file contains the appropriate Tcl commands to create and set up a Quartus II project and pass placement constraints.
<b>.acf</b>	Assignment and Configurations file for backward compatibility with the MAX+PLUS II software. For devices supported by the MAX+PLUS II software, the MAX+PLUS II assignments are imported from the MAX+PLUS II <b>.acf</b> file.
<b>.scf</b>	Synopsys Constraint Format file containing timing constraints for the TimeQuest Timing Analyzer.

**Note to Table 10–1:**

- (1) This report file includes performance estimates that are often based on pre-place-and-route information. Use the  $f_{MAX}$  reported by the Quartus II software after place-and-route—it is the only reliable source of timing information. This report file includes post-synthesis device resource utilization statistics that might inaccurately predict resource usage after place-and-route. The Synplify software does not account for black box functions nor for logic usage reduction achieved through register packing performed by the Quartus II software. Register packing combines a single register and look-up table (LUT) into a single logic cell, reducing logic cell utilization below the Synplify software estimate. Use the device utilization reported by the Quartus II software after place-and-route.

## Output Netlist File Name and Result Format

To specify the output netlist directory location and name for the Synplify software, perform the following steps:

1. On the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab.
3. In the **Results Directory** box, type your output netlist file directory location.
4. In the **Result File Name** box, type your output netlist file name.

By default, the directory and file name are set to the project implementation directory and the top-level design module or entity name.

The **Result Format** and **Quartus Version** options are also available on the **Implementation Results** tab. The **Result Format** list specifies an **.edf** or **.vqm** netlist, depending on your device family. The software creates an **.edf** output netlist file only for devices supported by the MAX+PLUS II software. For current Altera devices, the software generates a **.vqm**-formatted netlist.

Select the version of the Quartus II software that you are using in the **Quartus Version** list. This option ensures that the netlist is compatible with the software version and supports the newest features. Altera recommends using the latest version of the Quartus II software whenever possible. If your Quartus II software is newer than the versions available in the **Quartus Version** list, check if there is a newer version of the Synplify software available that supports the current Quartus II software version. Otherwise, choose the latest version in the list for the best compatibility.



The **Quartus Version** list is available only after selecting an Altera device.

To set the Quartus II software version used in the Synplify software, perform the following steps:

1. In the Synplify software, on the Project menu, click **Implementation Options**.
2. Click the **Implementation Results** tab, then click **Quartus Version**.
3. Choose the correct version number in the list.

Alternatively, use the following command from the command line:

```
set_option -quartus_version <version number> ↵
```

## Synplify Optimization Strategies

As designs become more complex and require increased performance, using different optimization strategies has become important. Combining Synplify software constraints with VHDL and Verilog HDL coding techniques and Quartus II software options can help you obtain the required results.



For additional design and optimization techniques, refer to the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 and the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

The Synplify software offers many constraints and optimization techniques to improve your design's performance. The Synplify Pro and Premier software add additional techniques that are not supported by the basic Synplify software. This section provides an overview of some of the techniques you can use to help improve the quality of your results.



For more information about applying the attributes discussed in this section, refer to the *Altera Constraints, Attributes, and Options* chapter of the *Synopsys FPGA Synthesis Reference Manual*.

## Using Synplify Premier to Optimize Your Design

Synplify Premier offers additional physical synthesis optimizations than the other Synplify products. After regular logic synthesis, Synplify Premier places and routes the design and attempts to restructure the netlist based on the physical location of the logic in the Altera device. Synplify Premier forward-annotates the design netlist to the Quartus II software to perform the final placement and routing. In the default flow, Synplify Premier also forward-annotates placement information for the critical path(s) in the design, which can improve the compilation time in the Quartus II software.

The physical location annotation file is called `<design name>_plc.tcl`. If you call the Quartus II software from the Synplify Premier user interface, the Quartus II software automatically uses this file for the placement information.

The Physical Analyst allows you to examine the placed netlist from Synplify Premier, similar to the HDL Analyst for a logical netlist. You can use this display to analyze and diagnose possible problems.

## Implementations in Synplify Pro or Premier

To create different synthesis results without overwriting the other results, in the Synplify Pro or Premier software, on the Project menu, click **New Implementation**. For each implementation, specify the target device, synthesis options, and constraint files. Each implementation generates its own subdirectory that contains all the resulting files, including `.vqm/.edf`, `.scf`, and `.tcl` files, from a compilation of the particular implementation. You can then compare the results of the different implementations to find the optimal set of synthesis options and constraints for a design.

## Timing-Driven Synthesis Settings

The Synplify software supports timing-driven synthesis with user-assigned timing constraints to optimize the performance of the design. The Synplify software optimizes the design to attempt to meet these constraints.

The Quartus II NativeLink feature allows timing constraints that are applied in the Synplify software to be forward-annotated for the Quartus II software using either a `.tcl` script file or a `.scf` file for timing-driven place and route. Refer to [“Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File”](#) on page 10-16 or [“Passing Constraints to the Quartus II Software using Tcl Commands”](#) on page 10-17 for more details about how constraints such as clock frequencies, false paths, and multicycle paths are forward-annotated. This section explains some of the important timing constraints in the Synplify software.



The Synplify Synthesis Report File (`.srr`) contains timing reports of estimated place-and-route delays. The Quartus II software can perform further optimizations on a post-synthesis netlist from third-party synthesis tools. In addition, designs might contain black boxes or intellectual property (IP) functions that have not been optimized by the third-party synthesis software. Actual timing results are obtained only after the design has gone through full placement and routing in the Quartus II software. For these reasons, the Quartus II post place-and-route timing reports provide a more accurate representation of the design. Use the statistics in these reports to evaluate design performance.

## Clock Frequencies

For single-clock designs, specify a global frequency when using the push-button flow. While this flow is simple and provides good results, often it does not meet the performance requirements for more advanced designs. You can use timing constraints, compiler directives, and other attributes to help optimize the performance of a design. You can enter these attributes and directives directly in the HDL code. Alternatively, you can enter attributes (not directives) into an `.sdc` file with the SCOPE window in the Synplify software.

Use the SCOPE window to set global frequency requirements for the entire design and individual clock settings. Use the **Clocks** tab in the SCOPE window to specify frequency (or period), rise times, fall times, duty cycle, and other settings. Assigning individual clock settings, rather than over-constraining the global frequency, helps the Quartus II software and the Synplify software achieve the fastest clock frequency for the overall design. The `define_clock` attribute assigns clock constraints.

## Multiple Clock Domains

The Synplify software can perform timing analysis on unrelated clock domains. Each clock group is a different clock domain and is treated as unrelated to the clocks in all other clock groups. All the clocks in a single clock group are assumed to be related and the Synplify software automatically calculates the relationship between the clocks. You can assign clocks to a new clock group or put related clocks in the same clock group by using the **Clocks** tab in the SCOPE window or with the `define_clock` attribute.

## Input and Output Delays

Specify the input and output delays for the ports of a design in the **Input/Output** tab of the SCOPE window or with the `define_input_delay` and `define_output_delay` attributes. The Synplify software does not allow you to assign the  $t_{CO}$  and  $t_{SU}$  values directly to inputs and outputs. However, a  $t_{CO}$  value can be inferred by setting an external output delay; a  $t_{SU}$  value can be inferred by setting an external input delay.

Equation 10-1 illustrates the relationship between  $t_{CO}$  and the output delay:

### Equation 10-1.

---

$$t_{CO} = \text{clock period} - \text{external output delay}$$

---

Equation 10-2 illustrates the relationship between  $t_{SU}$  and the input delay:

### Equation 10-2.

---

$$t_{SU} = \text{clock period} - \text{external input delay}$$

---

When the `syn_forward_io_constraints` attribute is set to 1, the Synplify software passes the external input and output delays to the Quartus II software using NativeLink integration. The Quartus II software then uses the external delays to calculate the maximum system frequency.

## Multicycle Paths

Specify any multicycle paths in the design in the **Multi-Cycle Paths** tab of the SCOPE window or with the `define_multicycle_path` attribute. A multicycle path is a path that requires more than one clock cycle to propagate. You must specify which paths are multicycle to avoid having the Quartus II and the Synplify compilers work excessively on a non-critical path. Not specifying these paths can also result in an inaccurate critical path being reported during timing analysis.

## False Paths

False paths are paths that should not be considered during timing analysis or which should be assigned low (or no) priority during optimization. Some examples of false paths are slow asynchronous resets and test logic added to the design. Set these paths in the **False Paths** tab of the SCOPE window. Use the `define_false_path` attribute.

## FSM Compiler

If the FSM Compiler is turned on, the compiler automatically detects state machines in a design. The compiler can then extract and optimize the state machine. The FSM Compiler analyzes the state machine and decides to implement sequential, gray, or one-hot encoding based on the number of states. It also performs unused-state analysis, optimization of unreachable states, and minimization of transition logic.

If the FSM Compiler is turned off, the compiler does not optimize logic as state machines. The state machines are implemented as coded in the HDL code. Thus, if the coding style for the state machine was sequential, the implementation is also sequential. If the FSM Compiler is turned on, the compiler infers and optimizes the state machines. The implementation is based on the number of states regardless of the coding style in the HDL code.

Use the `syn_state_machine` compiler directive to specify or prevent a state machine from being extracted and optimized. To override the default encoding of the FSM Compiler, use the `syn_encoding` directive.

The values for the `syn_encoding` directive are shown in [Table 10-2](#).

**Table 10-2.** `syn_encoding` Directive Values

Value	Description
Sequential	Generates state machines with the fewest possible flipflops. Sequential, also called binary, state machines are useful for area-critical designs when timing is not the primary concern.
Gray	Generates state machines where only one flipflop changes during each transition. Gray-encoded state machines tend to be free of glitches.
One-hot	Generates state machines containing one flipflop for each state. One-hot state machines typically provide the best performance and shortest clock-to-output delays. However, one-hot implementations are usually larger than sequential implementations.
Safe	Generates extra control logic to force the state machine to the reset state if an invalid state is reached. You can use the safe value in conjunction with the other three values, which results in the state machine being implemented with the requested encoding scheme and the generation of the reset logic.

[Example 10-1](#) shows sample VHDL code for applying the `syn_encoding` directive.

**Example 10-1.** Sample VHDL Code for `syn_encoding`

```
SIGNAL current_state : STD_LOGIC_VECTOR(7 DOWNT0 0);  
ATTRIBUTE syn_encoding : STRING;  
ATTRIBUTE syn_encoding OF current_state : SIGNAL IS "sequential";
```

The default is to optimize state machine logic for speed and area, but this is potentially undesirable for critical systems. The safe value generates extra control logic to force the state machine to the reset state if an invalid state is reached.

**FSM Explorer in Synplify Pro and Premier**

The Synplify Pro and Premier software can use FSM Explorer to explore different encoding styles for a state machine automatically, and then implement the best encoding based on the overall design constraints. FSM Explorer uses the FSM Compiler to identify and extract state machines from a design. However, unlike the FSM Compiler that chooses the encoding style based on the number of states, the FSM Explorer tries several different encoding styles before choosing a specific one. The trade-off is that the compilation requires more time to perform the analysis of the state machine, but finds an optimal encoding scheme for the state machine.

**Optimization Attributes and Options**

The following sections describe other attributes and options that you can modify in the Synplify software to improve your design performance.

**Retiming in Synplify Pro and Premier**

The Synplify Pro and Premier software can retime a design, which can improve the timing performance of sequential circuits by moving registers (register balancing) across combinational elements. Be aware that retimed registers incur name changes. To retime your design, turn on the **Retiming** option in the **Device** tab in the **Implementation Options** section, or use the `syn_allow_retiming` attribute.

**Maximum Fan-Out**

When your design has critical path nets with high fan-out, use the `syn_maxfan` attribute to control the fan-out of the net. Setting this attribute for a specific net results in the replication of the driver of the net to reduce overall fan-out. The `syn_maxfan` attribute takes an integer value and applies it to inputs or registers. (The `syn_maxfan` attribute cannot be used to duplicate control signals. The minimum allowed value of the attribute is 4.) Using this attribute might result in increased logic resource utilization, thus putting a strain on routing resources and leading to long compile times and difficult fitting.

If you must duplicate an output register or output enable register, you can create a register for each output pin by using the `syn_useioff` attribute (refer to [“Register Packing”](#)).

## Preserving Nets

During synthesis, the compiler maintains ports, registers, and instantiated components. However, some nets cannot be maintained to create an optimized circuit. Applying the `syn_keep` directive overrides the optimization of the compiler and preserves the net during synthesis. The `syn_keep` directive takes a Boolean value and can be applied to wires (Verilog HDL) and signals (VHDL). Setting the value to `true` preserves the net through synthesis.

## Register Packing

Altera devices allow for the packing of registers into I/O cells. Altera recommends allowing the Quartus II software to make the I/O register assignments. However, you can control register packing with the `syn_useioff` attribute. The `syn_useioff` attribute takes a Boolean value and can be applied to ports or entire modules. Setting the value to `1` instructs the compiler to pack the register into an I/O cell. Setting the value to `0` prevents register packing in both the Synplify and Quartus II software.

## Resource Sharing

The Synplify software uses resource sharing techniques during synthesis by default to reduce area. Turning off the **Resource Sharing** option on the **Options** tab of the **Implementation Options** dialog box improves performance results for some designs. You can also turn off the option for a specific module with the `syn_sharing` attribute. If you turn off this option, be sure to check the results to determine if it helps the timing performance. If it does not help, leave **Resource Sharing** turned on.

## Preserving Hierarchy

The Synplify software performs cross-boundary optimization by default. This results in the flattening of the design to allow optimization. Use the `syn_hier` attribute to over-ride the default compiler settings. The `syn_hier` attribute takes a string value and applies it to modules, architectures, or both. Setting the value to `hard` maintains the boundaries of a module, architecture, or both, but allows constant propagation. Setting the value to `locked` prevents all cross-boundary optimizations. Use the `locked` setting with the partition setting to create separate design blocks and multiple output netlists for incremental compilation, as described in [“Using MultiPoint Synthesis with Incremental Compilation”](#) on page 10-39.

By default, the Synplify software generates a hierarchical `.vqm` file. To flatten the file, set the `syn_netlist_hierarchy` attribute to `0`.

## Register Input and Output Delays

The advanced options called `define_reg_input_delay` and `define_reg_output_delay` can speed up paths feeding a register or coming from a register by a specific number of nanoseconds. The Synplify software attempts to meet the global clock frequency goals for a design as well as the individual clock frequency goals (set with `define_clock`). You can use these attributes to add delay to paths feeding into or out of registers to further constrain critical paths. The setting also works with negative numbers, so you can slow down a path that is too highly optimized.

These options are useful to close timing when your design does not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. Rerun synthesis using these options, specifying the actual routing delay (from place-and-route results) so that the tool can meet the required clock frequency. Synopsys recommends that for best results, do not make these assignments too aggressively. For example, increase the routing delay value but don't use the full routing delay from the last compilation.

In the SCOPE constraint window, use the registers panel with the following entries:

- **Register**—Specifies the name of the register. If you have initialized a compiled design, choose the name from the list.
- **Type**—Specifies whether the delay is an input or output delay.
- **Route**—Shrinks the effective period for the constrained registers by the specified value without affecting the clock period that is forward-annotated to the Quartus II software.

Use the following Tcl command syntax to specify an input or output register delay in nanoseconds.

---

**Example 10-2.** Specifying an Input or Output Register Delay Using Tcl Command Syntax

---

```
define_reg_input_delay {<register>} -route <delay in ns>
define_reg_output_delay {<register>} -route <delay in ns>
```

---

### syn\_direct\_enable

This attribute controls the assignment of a clock-enable net to the dedicated enable pin of a register. Using this attribute, you can direct the Synplify mapper to use a particular net as the only clock enable when the design has multiple clock enable candidates.

You can also use this attribute as a compiler directive to infer registers with clock enables. To do so, enter the `syn_direct_enable` directive in your source code, not the SCOPE spreadsheet.

The `syn_direct_enable` data type is Boolean. A value of **1** or **true** enables net assignment to the clock-enable pin. The following is the syntax for Verilog HDL:

```
object /* synthesis syn_direct_enable = 1 */ ;
```

### I/O Standard

For certain Altera devices, specify the I/O standard type to use for an I/O pad in the design using the **I/O Standard** panel in the Synplify SCOPE window.

**Example 10-3** shows the Synplify SDC syntax for the `define_io_standard` constraint, in which the `delay_type` must be either `input_delay` or `output_delay`.

---

**Example 10-3.** Synplify SDC Syntax for the `define_io_standard` Constraint

---

```
define_io_standard [-disable|-enable] {<objectName>} -delay_type \
[input_delay|output_delay] <columnTclName>{<value>} [<columnTclName>{<value>}...]
```

---



For details about supported I/O standards, refer to *Altera I/O Standards* in the *Synopsys FPGA Synthesis Reference Manual*.

## Altera-Specific Attributes

The following attributes are for use with specific Altera device features. These attributes are forward-annotated to the Quartus II project and are used during the place-and-route process.

### **altera\_chip\_pin\_lc**

Use this attribute to make pin assignments. This attribute takes a string value and applies it to inputs and outputs. Use the attribute only on the ports of the top-level entity in the design. Do not use this attribute to assign pin locations from entities at lower levels of the design hierarchy.



This attribute is not supported for any of the MAX series devices.

In the SCOPE window, select the attribute **altera\_chip\_pin\_lc** and set the value to a pin number or a list of pin numbers.

**Example 10-4** shows VHDL code for making location assignments for supported Altera devices. Pin location assignments for these devices are written to the output Tcl script.

#### **Example 10-4. Making Location Assignments in VHDL**

```
ENTITY sample (data_in : IN STD_LOGIC_VECTOR (3 DOWNTO 0);  
              data_out: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));  
ATTRIBUTE altera_chip_pin_lc : STRING;  
ATTRIBUTE altera_chip_pin_lc OF data_out : SIGNAL IS "14, 5, 16, 15";
```



The `data_out` signal is a 4-bit signal; `data_out[3]` is assigned to pin 14 and `data_out[0]` is assigned to pin 15.

### **altera\_implement\_in\_esb or altera\_implement\_in\_eab**

Use these attributes to implement logic in either embedded system blocks (ESBs) or embedded array blocks (EABs) rather than in logic resources to improve area utilization. The modules selected for such implementation cannot have feedback paths, and either all or none of the I/Os must be registered. This attribute takes a boolean value and can be applied to instances. (This option is applicable for devices with ESBs/EABs only. For example, the Stratix series is not supported by this option. This attribute is ignored for designs targeting devices that do not have ESBs or EABs.)

### **altera\_io\_powerup**

Use this attribute to define the power-up value of an I/O register that has no set or reset. This attribute takes a string value (**high** | **low**) and applies it to ports that have I/O registers. By default, the power-up value of the I/O is set to **low**.

### **altera\_io\_opendrain**

Use this attribute to specify open-drain mode I/O ports. This attribute takes a boolean value and applies it to outputs or bidirectional ports for devices that support open-drain mode.

## Exporting Designs to the Quartus II Software Using NativeLink Integration

The NativeLink feature in the Quartus II software facilitates the seamless transfer of information between the Quartus II software and EDA tools, and allows you to run other EDA design entry or synthesis, simulation, and timing analysis tools automatically from within the Quartus II software. After a design is synthesized in the Synplify software, a **.vqm** or **.edf** netlist file, an **.scf** file for TimeQuest Timing Analyzer timing constraints, and **.tcl** files are used to import the design into the Quartus II software for place-and-route. You can run the Quartus II software from within the Synplify software or as a stand-alone application. After you have imported the design into the Quartus II software, you can specify different options to further optimize the design.



When you are using NativeLink integration, the path to your project must not contain white space. The Synplify software uses Tcl scripts to communicate with the Quartus II software, and the Tcl language does not accept arguments with white space in the path.

Use NativeLink integration to integrate the Synplify software and Quartus II software with a single GUI for both synthesis and place-and-route operations. NativeLink integration allows you to run the Quartus II software from within the Synplify software GUI or to run the Synplify software from within the Quartus II software GUI.

This section explains the different NativeLink flows and provides details about how constraints are passed to the Quartus II software. This section describes the following topics:

- [“Running the Quartus II Software from within the Synplify Software” on page 10-14](#)
- [“Using the Quartus II Software to Run the Synplify Software” on page 10-15](#)
- [“Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script” on page 10-16](#)
- [“Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File” on page 10-16](#)
- [“Passing Constraints to the Quartus II Software using Tcl Commands” on page 10-17](#)

### Running the Quartus II Software from within the Synplify Software

To use the Quartus II software from within the Synplify software, you must first verify that the `QUARTUS_ROOTDIR` environment variable contains the Quartus II software installation directory located at `<Altera Design Suite Installation Directory>\quartus`. This environment variable is required to use the Synplify and Quartus II software together.

In the Windows operating system, the `QUARTUS_ROOTDIR` variable is set when you open the Quartus II user interface, so it is automatically set to the most recent version you opened in the user interface. If your software installation is located on another machine, ensure that you set this variable correctly. You can change the variable manually using the Control Panel, System icon.

On UNIX and Linux operating systems, the variable is not set automatically, so you must create an environment variable `QUARTUS_ROOTDIR` that points to the *<Altera Design Suite Installation Directory>/quartus* location.

Under each implementation in the Synplify Pro software, create a place-and-route implementation called `pr_<number> Altera Place and Route`. You can create new place and route implementations using the **New P&R** button in the GUI. To run the Quartus II software in command-line mode after each synthesis run, use the text box to turn on the place-and-route implementation. The results of the place and route are written to a log file in the `pr_<number>` directory under the current implementation directory.

You can also use the commands in the Quartus II menu to run the Quartus II software at any time following a successful completion of synthesis. In the Synplify software, on the Options menu, click **Quartus II** and then choose one of the following commands:

- **Launch Quartus**—Opens the Quartus II software GUI and creates a Quartus II project with the synthesized output file, forward-annotated timing constraints, and pin assignments. Use this command to configure options for the project and execute any Quartus II commands.
- **Run Background Compile**—Runs the Quartus II software in command-line mode with the project settings from the synthesis run. The results of the place-and-route are written to a log file.

The *<project\_name>\_cons.tcl* file is used to set up the Quartus II project and calls the *<project\_name>.tcl* file to pass constraints from the Synplify software to the Quartus II software. By default, the *<project\_name>.tcl* file contains device, timing, and location assignments. If the project is set up to use the TimeQuest Timing Analyzer, the *<project\_name>.tcl* file contains the command to use the Synplify-generated `.scf` constraints file with TimeQuest instead of using the Tcl constraints with the Classic Timing Analyzer.

## Using the Quartus II Software to Run the Synplify Software

You can set up the Quartus II software to run the Synplify software for synthesis using NativeLink integration. This feature allows you to use the Synplify software to quickly synthesize a design as part of a normal compilation in the Quartus II software. When you use this feature, the Synplify software does not use any timing constraints or assignments such as incremental compilation partitions that you have set in the Quartus II software.



For best results, Synopsys recommends that you set constraints in the Synplify software and use the Tcl script to pass these constraints to the Quartus II software, instead of calling Synplify from within the Quartus II software.

To set up Synplify in the Quartus II software, on the Tools menu, click **Options**. In the **Options** dialog box, click **EDA Tool Options** and specify the path of Synplify or Synplify Pro software.



For detailed information about using NativeLink integration with the Synplify software, refer to the Quartus II Help.

Beginning with the Quartus II software version 7.1, running the Synplify software with NativeLink integration is supported on both floating network and node-locked single-PC licenses. Both types of licenses support batch mode compilation.

## Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script

You can also use the Quartus II software separately from the Synplify software. To run the Tcl script generated by the Synplify software to set up your project and set up assignments such as the device selection, perform the following steps:

1. Ensure the **.vqm**, **.edf**, **.scf** (if you are using the TimeQuest Timing Analyzer timing constraints), and **.tcl** files are located in the same directory (they are located in the implementation directory by default).
2. In the Quartus II software, on the View menu, point to **Utility Windows** and click **Tcl Console**. The Quartus II Tcl Console opens.
3. At the Tcl Console command prompt, type the following:

```
source <path>/<project name>_cons.tcl ←
```


## Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File

The TimeQuest Timing Analyzer is a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry standard constraints format, Synopsys Design Constraints (SDC). This section explains how timing constraints set in the Synplify software are passed to the Quartus II software for use with the TimeQuest Timing Analyzer.

The Synplify-generated **.tcl** file contains constraints for the Quartus II software, such as the device specification and any location constraints. The timing constraints are forward-annotated using the **.tcl** file for the Quartus II Classic Timing Analyzer, as described in [“Passing Constraints to the Quartus II Software using Tcl Commands”](#) on page 10-17. For the TimeQuest Timing Analyzer, the timing constraints are forward-annotated in the Synopsys Constraints Format (**.scf**) file.

Altera recommends that you use the TimeQuest Timing Analyzer, as specified in the Synplify **.tcl** file that sets up the Quartus II project for the newest devices. However, you can use the Tcl commands for the Classic Timing Analyzer if required. You can manually change from the TimeQuest Timing Analyzer to the Classic Timing Analyzer in the Quartus II software by performing the following steps:

1. From the Assignments menu, click **Settings**.
2. In the **Category** list, select **Timing Analysis Settings**.
3. Under **Timing analysis processing**, select **Use Classic Timing Analyzer during compilation**.
4. Click **OK**.


 For additional information about the TimeQuest Timing Analyzer, refer to the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Synopsys recommends that you modify constraints using the SCOPE constraint editor window and not through the generated `.sdc`, `.scf`, or `.tcl` file.

The following list of Synplify constraints are converted to the equivalent Quartus II SDC commands and are forward-annotated to the Quartus II software in the `.scf` file:

- `define_clock`
- `define_input_delay`
- `define_output_delay`
- `define_multicycle_path`
- `define_false_path`

All Synplify constraints described in the following sections use the same Synplify commands as described in “[Passing Constraints to the Quartus II Software using Tcl Commands](#)” on page 10-17; however, the constraints are mapped to SDC commands for the TimeQuest Timing Analyzer.

 For syntax and arguments for these commands, refer to the applicable subsection or refer to Synplify Help. For a list of corresponding commands in the Quartus II software, refer to the Quartus II Help.

### Individual Clocks and Frequencies

Specify clock frequencies for individual clocks in the Synplify software with the command `define_clock`. This command is passed to the Quartus II software with `create_clock`.

### Input and Output Delay

Specify input delay and output delay constraints in the Synplify software with the commands `define_input_delay` and `define_output_delay`, respectively. These commands are passed to the Quartus II software with `set_input_delay` and `set_output_delay`.

### Multicycle Path

Specify a multicycle path constraint in the Synplify software with the command `define_multicycle_path`. This command is passed to the Quartus II software with `set_multicycle_path`.

### False Path

Specify a false path constraint in the Synplify software with the command `define_false_path`. This command is passed to the Quartus II software with `set_false_path`.

## Passing Constraints to the Quartus II Software using Tcl Commands

This section describes how Synplify constraints are converted to the equivalent Quartus II assignments and are forward-annotated to the Quartus II software with Tcl commands.

This section also describes timing constraints for the Quartus II Classic Timing Analyzer. If you are using the TimeQuest Timing Analyzer, the Quartus II timing constraints described in this section do not apply. Refer to “[Passing TimeQuest SDC Timing Constraints to the Quartus II Software in the .scf File](#)” on page 10-16 for information about timing constraints supported by TimeQuest.

### Global Signals

The Synplify software automatically promotes clock signals to global routing lines and passes Global Signal assignments to the Quartus II software. The assignments ensure that the same global routing constraints are applied during placement and routing.



The signals promoted to global routing can be different than the ones that the Quartus II software promotes to global routing by default. The Synplify software promotes only clock signals and not other control signals such as reset or enable. By default, without constraints from the Synplify software, the Quartus II software promotes control signals to global routing if they have high fan-out.

### Default or Global Clock Frequency

Use the following Synplify command to set the Synplify default or global clock frequency that applies to the entire project:

```
set_option -frequency <frequency>
```

The *<frequency>* is specified in MHz. If a global frequency is not specified, the software uses the default global clock frequency of 1 MHz.

The `set_option` constraint is passed to the Quartus II software with the following command:

```
set_global_assignment -name FMAX_REQUIREMENT <frequency> MHz
```

If a frequency is not specified in the Quartus II software, the software uses the default global clock frequency of 1 GHz.

### Individual Clocks and Frequencies

Specify clock frequencies for individual clocks with the following Synplify commands as shown in [Example 10-5](#).

#### Example 10-5. Specifying Clock Frequencies for Individual Clocks

```
define_clock -name {<clock_name>} -freq <frequency> -clockgroup <clock_group> -rise <rise_time>\
-fall <fall_time>
define_clock -name {<clock_name>} -period <period> -clockgroup <clock_group> -rise <rise_time>\
-fall <fall_time>
```

[Table 10-3](#) shows the command arguments.

**Table 10-3.** Command Arguments (Part 1 of 2)

Argument	Description
-name	The <i>&lt;clock_name&gt;</i> specifies a design port name or register output signal name and, after synthesis, corresponds to a <i>&lt;mapped_clock_name&gt;</i> .
-freq (1)	The <i>&lt;frequency&gt;</i> is specified in MHz.

**Table 10-3.** Command Arguments (Part 2 of 2)

Argument	Description
-period (2)	The <period> is specified in ns.
-clockgroup	If the <clock_group> is not specified, it defaults to default_clkgroup. The Synplify software assumes all clocks belonging to the same clock group are related. If you do not specify a clock group, the clock belongs to the default clock group. Therefore, if you do not specify any clock groups, all the clocks are considered related by default in the software.
-rise -fall	The <rise_time> and <fall_time> specify a non-default duty cycle. By default, the Synplify synthesis tool assumes that the clock is a 50% duty cycle clock, with the rising edge at 0 and the falling edge at period/2. If you have another duty clock cycle, you can specify the appropriate <b>Rise At</b> and <b>Fall At</b> values.

**Notes to Table 10-3:**

- (1) When the <frequency> is specified, the Synplify software uses <fall\_time> and <frequency> to calculate the duty\_cycle with the following formula:  $duty\_cycle = (<fall\_time> - <rise\_time>) \times <frequency> / 100$ .
- (2) When the <period> is specified, the Synplify software uses <fall\_time> and <period> to calculate the duty\_cycle with the following formula:  $duty\_cycle = 100 \times (<fall\_time> - <rise\_time>) / <period>$ .

The equivalent Quartus II Classic Timing Analyzer commands depend on how the clock groups are defined. In the Quartus II software, clocks that belong to the same or related clock settings are considered related clocks. Clocks assigned to unrelated clock settings are unrelated clocks. There is a one-to-one correspondence between each Quartus II clock setting and a Synplify clock group.



The following sections describe only the frequency constraints. Use the corresponding constraints for the period.

### Virtual Clocks

The Quartus II software supports virtual clocks. If you use the virtual clock setting in the Synplify software, the setting is mapped to a constraint in the Quartus II software.

### Route Delay Option

The -route option in the Synplify software clock constraints is designed for use in synthesis only if you do not meet timing goals because the routing delay after placement and routing exceeds the delay predicted by the Synplify software. This constraint does not have to be forward-annotated to the Quartus II software.

### Multiple Clocks in Different Clock Groups

You can specify clock frequencies for multiple clocks with the Synplify commands shown in [Example 10-6](#).

**Example 10-6.** Specifying Clock Frequencies for Multiple Clocks

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group2> \
-rise <rise_time2> -fall <fall_time2>
```

<clock\_group1> and <clock\_group2> are unique names defined in the Synplify software for base clock settings in the Quartus II Classic Timing Analyzer.

If the clock *<rise\_time>* is zero ("0"), multiple separate clocks are passed to the Quartus II software with the commands shown in [Example 10-7](#).

---

**Example 10-7.** Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target mapped_clock_name1 <base_clock_setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty_cycle2> \
-target mapped_clock_name2 <base_clock_setting2>
```

---

If the clock *<rise\_time>* is non-zero, multiple separate clocks are passed to the Quartus II software with the following commands shown in [Example 10-8](#).

---

**Example 10-8.** Quartus II Assignments for Multiple Clocks if the Clock Rise Time is Not Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> -no_target <base clock setting1>

create_base_clock -fmax <frequency2>MHz -duty_cycle <duty_cycle2> -no_target <base clock setting2>

create_relative_clock -base_clock <base clock setting1> -offset <rise time1>ns \
-duty_cycle <duty cycle1> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name1> <derived clock setting1>

create_relative_clock -base_clock <base clock setting2> -offset <rise time2>ns \
-duty_cycle <duty cycle2> -multiply <multiply by> -divide <divide by> \
-target <mapped clock name2> <derived clock setting2>
```

---

### Multiple Clocks with Different Frequencies in the Same Clock Group

In the Synplify software, you can specify multiple clocks with relative clock settings in the same clock group with different frequencies, with the commands shown in [Example 10-9](#).

---

**Example 10-9.** Specifying Multiple Clocks with Different Frequencies in the Same Clock Group

```
define_clock -name {<clock_name1>} -freq <frequency1> -clockgroup <clock_group1> \
-rise <rise_time1> -fall <fall_time1>

define_clock -name {<clock_name2>} -freq <frequency2> -clockgroup <clock_group2> \
-rise <rise_time2> -fall <fall_time2>
```

---



When you specify clocks with different frequencies in the same clock group, the software calculates the *<multiply\_by>* and the *<divide\_by>* factors for relative clock settings from *<frequency1>* and *<frequency2>* in the clock group settings.

If the clock *<rise\_time>* is zero, multiple clocks with relative clock settings in the same clock group with different frequencies are passed to the Quartus II software with the commands shown in [Example 10-10](#).

---

**Example 10-10.** Quartus II Assignments for Multiple Clocks with Different Frequencies in the Same Clock Group, if the Clock Rise Time is Zero

```
create_base_clock -fmax <frequency1>MHz -duty_cycle <duty_cycle1> \
-target <mapped_clock_name1> <base_clock_setting1>

create_relative_clock -base_clock <base_clock_setting1> -duty_cycle <duty_cycle2> \
-multiply <multiply_by> -divide <divide_by> -target <mapped_clock_name2> <derived_clock_setting2>
```

---

## Inter-Clock Relationships—Delays and False Paths between Clocks

Set a clock-to-clock delay constraint in Synplify with the commands in [Example 10-11](#).

### Example 10-11. Specifying Clock-to-Clock Delay Constraints

```
define_clock_delay -fall <clock_name1> -rise <clock_name2> <delay_value>  
define_clock_delay -rise <clock_name1> -fall <clock_name2> <delay_value>  
define_clock_delay -rise <clock_name1> -rise <clock_name2> <delay_value>  
define_clock_delay -fall <clock_name1> -fall <clock_name2> <delay_value>
```

If *<delay\_value>* is set to `false`, these constraints in Synplify indicate a false path between the two clocks. If all four rise/fall clock-edge pairs are specified in the Synplify software, the Synplify constraints are mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <clock_name1> -to <clock_name2>
```

If all four clock-edge pairs are not specified in Synplify, the constraint cannot be mapped to a constraint for the Quartus II Classic Timing Analyzer.

If *<delay\_value>* is set to a value other than `false`, these constraints in Synplify are not mapped to constraints in the Quartus II software. The Quartus II Classic Timing Analyzer does not support clock-edge to clock-edge delay constraints.

### False Paths

Specify the false path constraint in the Synplify software with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2>
```

The signals *<sig\_name1>* and *<sig\_name2>* can be design port names or register instance names.

The `define_false_path` constraint in the Synplify software is mapped to the constraint in the Quartus II software, as shown in the following command:

```
set_timing_cut_assignment -from <sig_name2> -to <sig_name2>
```

The Synplify software can identify pairs of signal sets such that every member of the cross-product of these two sets is a valid false path constraint. Signal groups can be defined in the Quartus II Classic Timing Analyzer with the following commands:

```
timegroup -add_member sig_name1_i <sig_group1>  
(for every signal in <sig_group1>)  
timegroup -add_member sig_name2_i <sig_group2>  
(for every signal in <sig_group2>)  
set_timing_cut_assignment -from <sig_group1> -to <sig_group2>
```

If the signals *<sig\_name1>* or *<sig\_name2>* represent multiple signals such as a wildcard, group, or bus, the constraints can be appropriately expanded for representation in the Quartus II software. The Quartus II software supports wildcard signal names, and signal groups for timing assignments. The Quartus II software does not support bus notation, such as `A[7:4]`.

### False Path from a Signal

Specify a false path constraint from a signal in the Synplify software with the following command:

```
define_false_path -from <sig_name>
```

The Quartus II Classic Timing Analyzer does not support “from-only” path specifications. You must also include a “to-path” specification. However, you can specify a wildcard for the `-to` signal. This constraint in Synplify is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from <sig_name> -to {*}
```

### False Path to a Signal

Specify a false path constraint to a signal in the Synplify software with the following command:

```
define_false_path -to <sig_name>
```

The Quartus II Classic Timing Analyzer does not support “to-only” path specifications. You must include a “from-path” specification. However, you can specify a wildcard for the `-from` signal. This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_timing_cut_assignment -from {*} -to <sig_name>
```

### False Path through a Signal

Specify a false path constraint through a signal in the Synplify software with the following command:

```
define_false_path -from <sig_name1> -to <sig_name2> \
-through <sig_name3>
```

The Quartus II Classic Timing Analyzer does not support false paths with a “through path” specification. Any constraint in the Synplify software with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

### Multicycle Paths

Specify a multicycle path constraint in the Synplify software with the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> <clock_cycles>
```

This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_multicycle_assignment -from <sig_name1> \
-to <sig_name2> <clock_cycles>
```

If the signals `<sig_name1>` or `<sig_name2>` represent multiple signals such as a wildcard, group, or bus, the constraints can be appropriately expanded for representation in the Quartus II software as described in [“False Paths” on page 10-9](#).



`<clock_cycles>` is the number of clock cycles for the multicycle path.

### Multicycle Path from a Signal

Specify a multicycle path constraint from a signal in the Synplify software with the following command:

```
define_multicycle_path -from <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from <sig_name> -to {*} <clock_cycles>
```

### Multicycle Path to a Signal

Specify a multicycle path constraint to a signal in the Synplify software with the following command:

```
define_multicycle_path -to <sig_name> <clock_cycles>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints:

```
set_multicycle_assignment -from {*} -to <sig_name> <clock_cycles>
```

### Multicycle Path through a Signal

Specify a multicycle path constraint through a signal in the Synplify software using the following command:

```
define_multicycle_path -from <sig_name1> -to <sig_name2> \  
-through <sig_name3> <clock_cycles>
```

The Quartus II Classic Timing Analyzer does not support multicycle paths with a “through path” specification. Any constraint in the Synplify software with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

### Maximum Path Delays

Specify the maximum path delay relationships between signals in the Synplify software with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> -max <delay_value>
```

This constraint in the Synplify software is mapped to the following constraint in the Quartus II software:

```
set_instance_assignment -from <sig_name1> \  
-to <sig_name2> -name SETUP_RELATIONSHIP <delay_value>ns
```

The Quartus II Classic Timing Analyzer does not support signal groups or bus notation. It supports only register names for this constraint.

### Maximum Path Delay from a Signal

Specify the maximum path delay constraint from a signal in the Synplify software with the following command:

```
define_path_delay -from <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-to` value in the Quartus II Classic Timing Analyzer, similar to false path constraints:

```
set_instance_assignment -from <sig_name> -to {*} \  
-name SETUP_RELATIONSHIP <delay_value>ns
```

### Maximum Path Delay to a Signal

Specify the maximum path delay constraint to a signal in the Synplify software with the following command:

```
define_path_delay -to <sig_name> -max <delay_value>
```

This constraint is mapped using a wildcard for the `-from` value in the Quartus II Classic Timing Analyzer, similar to the false path constraints.

```
set_instance_assignment -from {*}<sig_name> \  
-to <sig_name> -name SETUP_RELATIONSHIP <delay_value>ns
```

### Maximum Path Delay through a Signal

Specify the maximum path delay constraint through a signal in the Synplify software with the following command:

```
define_path_delay -from <sig_name1> -to <sig_name2> \  
-through <sig_name3> -max <delay_value>
```

The Quartus II Classic Timing Analyzer does not support maximum path delay constraints with a “through path” specification. Any constraint in Synplify with a `-through` specification is not mapped to a constraint for the Quartus II Classic Timing Analyzer.

### Register Input and Output Delays

These register input delay and register output delay constraints in the Synplify software are for use in synthesis only, and therefore are not forward-annotated to the Quartus II software.

#### Default External Input Delay

Specify the default input delay constraint in the Synplify software with the following command:

```
define_input_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock {*} <delay_value> {*}
```

#### Port-Specific External Input Delay

Specify a port-specific input delay constraint in the Synplify software with the following command:

```
define_input_delay <input_port_name> <delay_value> \  
-ref <clock_name>:<clock_edge>
```

The `<clock_edge>` can be set to **r** (rising edge) or **f** (falling edge).

When the clock edge is **r** (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_input_delay -clock <clock_name> <delay_value> <input_port_name>
```

When the `<clock_edge>` is **f** (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of input delays with respect to the falling edge of the clock.

### Default External Output Delay

Specify the default output delay constraint in the Synplify software with the following command:

```
define_output_delay -default <delay_value>
```

This constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock {*} <delay_value> {*}
```

### Port-Specific External Output Delay

Specify a port-specific input delay constraint in the Synplify software with the following command:

```
define_output_delay <output_port_name> <delay_value> \  
-ref <clock_name>:<clock_edge>
```

The *<clock\_edge>* can be set to **r** (rising edge) or **f** (falling edge). When the clock edge is **r** (rising edge), this constraint is mapped to the following constraint in the Quartus II software:

```
set_output_delay -clock <clock_name> <delay_value> <output_port_name>
```

When the clock\_edge is **f** (falling edge), this constraint is not mapped to a constraint in the Quartus II software. The Quartus II Classic Timing Analyzer does not support the specification of output delays with respect to the falling edge of the clock.

## Guidelines for Altera Megafunctions and Architecture-Specific Features


Altera provides parameterizable megafunctions including LPMs, device-specific Altera megafunctions, IP available as Altera MegaCore® functions, and IP available through the Altera Megafunction Partners Program (AMPP<sup>SM</sup>). You can use megafunctions and IP functions by instantiating them in your HDL code, or you can infer certain megafunctions from generic HDL code.

If you want to instantiate a megafunction in your HDL code, you can do so with the MegaWizard Plug-In Manager to parameterize the function or by instantiating the function using the port and parameter definition. The MegaWizard Plug-In Manager provides a graphical interface within the Quartus II software for customizing and parameterizing any available megafunction for the design. For more information about the MegaWizard Plug-In Manager flow with the Synplify software, refer to [“Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager” on page 10–26](#) and [“Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench” on page 10–28](#).



For more information about specific Altera megafunctions, refer to the Quartus II Help. For more information about IP functions, refer to the appropriate IP documentation.

The Synplify software also automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. The Synplify software provides options to control inference of certain types of megafunctions, as described in [“Inferring Altera Megafunctions from HDL Code” on page 10–31](#).


 For a detailed discussion about instantiating versus inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. This chapter also provides details about using the MegaWizard Plug-In Manager in the Quartus II software and explains the files generated by the wizard, as well as providing coding style recommendations and HDL examples for inferring megafunctions in Altera devices.

## Instantiating Altera Megafunctions Using the MegaWizard Plug-In Manager

This section describes how to instantiate Altera megafunctions using the MegaWizard Plug-In Manager.

When you use the MegaWizard Plug-In Manager to set up and parameterize a megafunction, the MegaWizard Plug-In Manager creates a VHDL or Verilog HDL wrapper file `<output file>.v | vhd` that instantiates the megafunction.

The Synplify software makes use of the Quartus II timing and resource estimation netlist feature to report more accurate resource utilization and timing performance estimates, and take better advantage of timing-driven optimization than treating the megafunction as a “black box”. Include the MegaWizard-generated megafunction variation wrapper file in your Synplify project so the Synplify software has all the information about the megafunction.

 There is an option in the MegaWizard Plug-In Manager to generate a netlist for resource and timing estimation. This option is not recommended for the Synplify software because the software generates this information in the background without a separate netlist. If you do create a separate netlist `<output file>_syn.v` and use that file in your synthesis project, you must also include the `<output file>.v | vhd` file in your Quartus II project.

Make sure to set the correct Quartus II version in the Synplify software before compiling the MegaWizard-generated file so the software uses the correct library definitions for the megafunction. The **Quartus Version** setting must match the version of the Quartus II software used to generate the customized megafunction in the MegaWizard Plug-In Manager.

For details about how to set the Quartus II version in the Synplify software, refer to “[Output Netlist File Name and Result Format](#)” on page 10-5.

In addition, ensure that the `QUARTUS_ROOTDIR` environment variable is set to the installation directory location of the correct Quartus II version. The Synplify software uses this information to launch the Quartus II software in the background. The environment variable setting must match the version of the Quartus II software used to generate the customized megafunction in the MegaWizard Plug-In Manager. Refer to “[Using the Quartus II Software to Run the Synplify Software](#)” on page 10-15 for details.

### Using MegaWizard Plug-In Manager-Generated Verilog HDL Files for Megafunction Instantiation

If you check the `<output file>_inst.v` option on the last page of the wizard, the MegaWizard Plug-In Manager generates a Verilog HDL instantiation template file for use in your Synplify design. The instantiation template file, `<output file>_inst.v`, helps to instantiate the megafunction variation wrapper file, `<output file>.v`, in your top-level design. Include the megafunction variation wrapper file `<output file>.v` in your Synplify project. The Synplify software includes the megafunction information in the output `.vqm` netlist file. There is no need to include the MegaWizard-generated megafunction variation wrapper file in your Quartus II project.

### Using MegaWizard Plug-In Manager-Generated VHDL Files for Megafunction Instantiation

If you check the `<output file>.cmp` and `<output file>_inst.vhd` options on the last page of the wizard, the MegaWizard Plug-In Manager generates a VHDL component declaration file and a VHDL instantiation template file for use in your Synplify design. These files can help you instantiate the megafunction variation wrapper file, `<output file>.vhd`, in your top-level design. Include `<output file>.vhd` in your Synplify project. The Synplify software includes the megafunction information in the output `.vqm` netlist file. There is no need to include the MegaWizard-generated megafunction variation wrapper file in your Quartus II project.

### Changing Synplify's Default Behavior for Instantiated Altera Megafunctions

By default, the Synplify software automatically calls the Quartus II software in the background to generate a resource and timing estimation netlist for megafunctions, as described in the previous sections.

You might want to change this behavior to reduce run times in the Synplify software (because generating the netlist files can take several minutes for large designs), or if the Synplify software cannot access your Quartus II software installation to generate the files. Changing this behavior might speed up the compilation time in the Synplify software, but the Quality of Results (QoR) might be reduced.

The Synplify software calls the Quartus II software to generate information in two ways:

- Some megafunctions provide a “clear box” model—Synplify software can fully synthesize this model and include the device architecture-specific primitives in the output `.vqm` netlist file.
- Other megafunctions provide a “grey box” model—Synplify can read the resource information but the netlist does not contain all the logic functionality.

For these functions, the Synplify software uses the logic information for resource and timing estimation and optimization, and then instantiates the megafunction in the output `.vqm` netlist file so the Quartus II software can implement the appropriate device primitives. By default, the Synplify software uses the clear box model when available, and otherwise uses the grey box model. To change this behavior, click **Implementation Options**, and on the **Device** tab, change the Altera Models setting. The default is **on**. To enable clear box models but not grey box, select **clearbox\_only**, or to turn off the feature entirely, choose **off**.

## Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench

Many Altera IP functions include a resource and timing estimation netlist that the Synplify software uses to report more accurate resource utilization and timing performance estimates, and take better advantage of timing-driven optimization than a black box function.

To create this netlist file, first select the IP function in the MegaWizard Plug-In Manager and click **Next** to open the IP Toolbench. Click **Step 2: Set Up Simulation**, which sets up all the EDA options. Enable the **Generate netlist** option to generate a netlist for resource and timing estimation. The netlist file is generated when you click **Step 3: Generate**.

The Quartus II software generates a file *<output file>\_syn.v*. This netlist contains the “grey box” information for resource and timing estimation, but does not contain the actual implementation. Include this netlist file in your Synplify project. Next, include the megafunction variation wrapper file *<output file>.v|vhd* in the Quartus II project along with your Synplify *.vqm* output netlist.

If your IP function does not include a resource and timing estimation netlist, the Synplify software must treat the IP function as a black box. In this case, refer to the following subsections for details about creating black boxes.

For information about including Quartus II-specific files in your Synplify project so they are automatically passed to the Quartus II software along with the output *.vqm* file, refer to [“Including Files for Quartus II Placement and Routing Only” on page 10-30](#).

### Using Generated Verilog HDL Files for Black Box IP Function Instantiation

Use the `syn_black_box` compiler directive to declare a module as a black box. The top-level design files must contain the IP port mapping and a hollow-body module declaration. Apply the `syn_black_box` directive to the module declaration in the top-level file or a separate file included in the project to instruct the Synplify software that this is a black box. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives, as discussed in [“Other Synplify Software Attributes for Creating Black Boxes” on page 10-29](#).

[Example 10-12](#) shows a sample top-level file that instantiates `my_verilogIP.v`, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

#### Example 10-12. Sample Top-Level Verilog HDL Code with Black Box Instantiation of IP

---

```

module top (clk, count);
    input clk;
    output[7:0] count;
    my_verilogIP verilogIP_inst (.clock (clk), .q (count));
endmodule
// Module declaration
// The following attribute is added to create a
// black box for this module.
module my_verilogIP (clock, q) /* synthesis syn_black_box */;
    input clock;
    output[7:0] q;
endmodule

```

---

## Using Generated VHDL Files for Black Box IP Function Instantiation

Use the `syn_black_box` compiler directive to declare a component as a black box. The top-level design files must contain the megafunction variation component declaration and port mapping. Apply the `syn_black_box` directive to the component declaration in the top-level file. The software compiles successfully without this directive, but reports an additional warning message. Using this directive allows you to add other directives, such as the ones in the “[Other Synplify Software Attributes for Creating Black Boxes](#)” section.

[Example 10-13](#) shows a sample top-level file that instantiates `my_vhdlIP.vhd`, which is a simplified customized variation generated by the MegaWizard Plug-In Manager and IP Toolbench.

### Example 10-13. Sample Top-Level VHDL Code with Black Box Instantiation of IP

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY top IS
  PORT (
    clk: IN STD_LOGIC ;
    count: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
END top;

ARCHITECTURE rtl OF top IS
  COMPONENT my_vhdlIP
  PORT (
    clock: IN STD_LOGIC ;
    q: OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
end COMPONENT;
attribute syn_black_box : boolean;
attribute syn_black_box of my_vhdlIP: component is true;
BEGIN
  vhdlIP_inst : my_vhdlIP PORT MAP (
    clock => clk,
    q => count
  );
END rtl;
```

## Other Synplify Software Attributes for Creating Black Boxes

Instantiating a function as a black box methodology does not provide the synthesis tool any visibility into the function module. Thus, it does not take full advantage of the synthesis tool's timing-driven optimization. For better timing optimization, especially if the black box does not have registered inputs and outputs, add timing models to black boxes. This can be done by adding the `syn_tpd`, `syn_tsu`, and `syn_tco` attributes. Refer to [Example 10-14](#) for a Verilog HDL example.

**Example 10-14.** Adding Timing Models to Black Boxes in Verilog HDL

---

```

module ram32x4 (z,d,addr,we,clk);
  /* synthesis syn_black_box syn_tcol="clk->z[3:0]=4.0"
     syn_tpd1="addr[3:0]->z[3:0]=8.0"
     syn_tsu1="addr[3:0]->clk=2.0"
     syn_tsu2="we->clk=3.0" */
  output[3:0]z;
  input[3:0]d;
  input[3:0]addr;
  input we;
  input clk;
endmodule

```

---

The following additional attributes are supported by the Synplify software to communicate details about the characteristics of the black box module within the HDL code:

- `syn_resources`—Specifies the resources used in a particular black box
- `black_box_pad_pin`—Prevents mapping to I/O cells
- `black_box_tri_pin`—Indicates a tri-stated signal



For more information about applying these attributes, refer to the *Altera Constraints, Attributes, and Options* chapter of the *Synopsys FPGA Synthesis Reference Manual*.

## Including Files for Quartus II Placement and Routing Only

In the Synplify software, you can add files to your project that are used only during placement and routing in the Quartus II software. This can be useful if you have grey boxes or black boxes for Synplify synthesis that require the full design files to be compiled in the Quartus II software.

Add the files to the Synplify project like other source files. Then right-click on the file and click **File options**. Enable the **Use for Place and Route Only** option. You can also set the option in a script using the `-job_owner par` option.

For example, the commands in [Example 10-15](#) define files for a Synplify project that includes a top-level design file, a grey box netlist file, an IP wrapper file, and an encrypted IP file. With these files, the Synplify software writes an empty instantiation of “core” in the `.vqm` file and uses the gray box netlist for resource and timing estimation. The files `core.v` and `core_enc8b10b.v` are not compiled by Synplify and are copied into the place-and-route directory. The Quartus II software compiles these files to implement the “core” IP block.

**Example 10-15.** Commands to Define Files for a Synplify Project

---

```

add_file -verilog -job_owner par "core_enc8b10b.v"
add_file -verilog -job_owner par "core.v"
add_file -verilog "core_gb.v"
add_file -verilog "top.v"

```

---



### Controlling the Inferring of DSP Blocks

You can implement multipliers in DSP blocks or in logic in Altera devices that contain DSP blocks. You can control this implementation through attribute settings in the Synplify software.

#### Signal Level Attribute

You can control the implementation of individual multipliers by using the `syn_multstyle` attribute as shown in the following Verilog HDL code:

```
<signal_name> /* synthesis syn_multstyle = "logic" */;
```

where `signal_name` is the name of the signal.



This setting applies to wires only; it cannot be applied to registers.

Table 10-4 shows the values for the signal level attribute in the Synplify software that controls the implementation of the multipliers in the DSP blocks or LEs.

**Table 10-4.** Attribute Settings for DSP Blocks in the Synplify Software

Attribute Name	Value	Description
<code>syn_multstyle</code>	<code>lpm_mult</code>	LPM function inferred and multipliers implemented in DSP blocks
<code>syn_multstyle</code>	<code>logic</code>	LPM function not inferred and multipliers implemented LEs by the Synplify software
<code>syn_multstyle</code>	<code>block_mult</code>	DSP megafunction is inferred and multipliers are mapped directly to DSP block device primitives (for supported devices)

Example 10-16 and Example 10-17 show simple Verilog HDL and VHDL code using the `syn_multstyle` attribute.

#### Example 10-16. Signal Attributes for Controlling DSP Block Inference in Verilog HDL Code

```
module mult(a,b,c,r,en);
  input [7:0] a,b;
  output [15:0] r;
  input [15:0] c;
  input en;
  wire [15:0] temp /* synthesis syn_multstyle="logic" */;

  assign temp = a*b;
  assign r = en ? temp : c;
endmodule
```

---

**Example 10–17.** Signal Attributes for Controlling DSP Block Inference in VHDL Code

---

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity onereg is port (
    r : out std_logic_vector(15 downto 0);
    en : in std_logic;
    a : in std_logic_vector(7 downto 0);
    b : in std_logic_vector(7 downto 0);
    c : in std_logic_vector(15 downto 0)
);
end onereg;

architecture beh of onereg is
    signal temp : std_logic_vector(15 downto 0);
    attribute syn_multstyle : string;
    attribute syn_multstyle of temp : signal is "logic";

begin
    temp <= a * b;
    r <= temp when en='1' else c;
end beh;
```

---

### Inferring RAM

When a RAM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For certain devices, the software maps directly to memory block device primitives instead of instantiating a megafunction in the **.vqm** file.

Follow these guidelines for the Synplify software to successfully infer RAM in a design:

- The address line must be at least two bits wide.
- Resets on the memory are not supported. Refer to the device family documentation for information about whether read and write ports must be synchronous.
- Some Verilog HDL statements with blocking assignments might not be mapped to RAM blocks, so avoid blocking statements when modeling RAMs in Verilog HDL.

For certain device families, the `syn_ramstyle` attribute specifies the implementation to use for an inferred RAM. You can apply `syn_ramstyle` globally, to a module, or to a RAM instance, to specify `registers` or `block_ram` values. To turn off RAM inference, set the attribute value to `registers`.

When inferring RAM for certain Altera device families, the Synplify software generates additional bypass logic. This logic is generated to resolve a half-cycle read/write behavior difference between the RTL and post-synthesis simulations. The RTL simulation shows the memory being updated on the positive edge of the clock; the post-synthesis simulation shows the memory being updated on the negative edge of the clock. To eliminate bypass logic, the output of the RAM must be registered. By adding this register, the output of the RAM is seen after a full clock cycle, by which time the update has occurred, thus eliminating the need for bypass logic.

For devices with TriMatrix memory blocks, disable the creation of glue logic by setting the `syn_ramstyle` value to `no_rw_check`. Use `syn_ramstyle` with a value of `no_rw_check` to disable the creation of glue logic in dual-port mode.

[Example 10-18](#) shows sample VHDL code for inferring dual-port RAM.

---

**Example 10-18.** VHDL Code for Inferred Dual-Port RAM

---

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
      data_in: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      wr_addr, rd_addr: IN STD_LOGIC_VECTOR (6 DOWNTO 0);
      we: IN STD_LOGIC;
      clk: IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem: Mem_Type;
SIGNAL addr_reg: STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN
  data_out <= mem (CONV_INTEGER(rd_addr));
  PROCESS (clk, we, data_in) BEGIN
    IF (clk='1' AND clk'EVENT) THEN
      IF (we='1') THEN
        mem(CONV_INTEGER(wr_addr)) <= data_in;
      END IF;
    END IF;
  END PROCESS;
END ram_infer;
```

---

**Example 10–19** shows an example of the VHDL code preventing bypass logic for inferring dual-port RAM. The extra latency behavior stems from the inferring methodology and is not required when instantiating a megafunction.

---

**Example 10–19.** VHDL Code for Inferred Dual-Port RAM Preventing Bypass Logic

---

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;

ENTITY dualport_ram IS
PORT ( data_out: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
      data_in : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      wr_addr, rd_addr : IN STD_LOGIC_VECTOR (6 DOWNTO 0);
      we : IN STD_LOGIC;
      clk : IN STD_LOGIC);
END dualport_ram;

ARCHITECTURE ram_infer OF dualport_ram IS
TYPE Mem_Type IS ARRAY (127 DOWNTO 0) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL mem : Mem_Type;
SIGNAL addr_reg : STD_LOGIC_VECTOR (6 DOWNTO 0);
SIGNAL tmp_out : STD_LOGIC_VECTOR(7 DOWNTO 0); --output register

BEGIN
  tmp_out <= mem (CONV_INTEGER(rd_addr));
  PROCESS (clk, we, data_in) BEGIN
    IF (clk='1' AND clk'EVENT) THEN
      IF (we='1') THEN
        mem(CONV_INTEGER(wr_addr)) <= data_in;
      END IF;
      data_out <= tmp_out; --registers output preventing
                          -- bypass logic generation.
    END IF;
  END PROCESS;
END ram_infer;
```

---

### RAM Initialization

Use Verilog HDL system tasks \$readmemb or \$readmemh in your HDL code to initialize RAM memories. The Synplify compiler forward-annotates the initialization values in the **.srs** (technology-independent RTL netlist) file and the mapper generates a corresponding hexadecimal memory initialization (**.hex**) file. One **.hex** file is created for each of the **altsyncram** megafunctions that are inferred in the design. The **.hex** file is associated with the **altsyncram** instance in the **.vqm** file using the **init\_file** attribute.

[Example 10-20](#) and [Example 10-21](#) illustrate how RAM memories can be initialized through HDL code and how the corresponding `.hex` file is generated using Verilog HDL.

---

**Example 10-20.** Using `$readmemb` System Task to Initialize an Inferred RAM in Verilog HDL Code

---

```
initial
begin
    $readmemb("mem.ini", mem);
end

always @(posedge clk)
begin
    raddr_reg <= raddr;
    if(we)
        mem[waddr] <= data;
end
```

---

**Example 10-21.** Sample of `.vqm` Instance Containing Memory Initialization File from [Example 10-20](#)

---

```
altsyncram mem_hex( .wren_a(we), .wren_b(GND), ... );

defparam mem_hex.lpm_type = "altsyncram";
defparam mem_hex.operation_mode = "Dual_Port";
...
defparam mem_hex.init_file = "mem_hex.hex";
```

---

### Inferring ROM

When a ROM block is inferred from an HDL design, the software uses an Altera megafunction to target the device memory architecture. For certain devices, the software maps directly to memory block device atoms instead of instantiating a megafunction in the `.vqm` file. Follow these guidelines for the Synplify software to successfully infer ROM in a design:

- The address line must be at least two bits wide.
- The ROM must be at least half full.
- A CASE or IF statement must make 16 or more assignments using constant values of the same width.

### Inferring Shift Registers

The software infers shift registers for sequential shift components so that they can be placed in dedicated memory blocks in supported device architectures using the `ALTSHIFT_TAPS` megafunction.

If required, set the implementation style with the `syn_sr1style` attribute. If you do not want the components automatically mapped to shift registers, set the value to `registers`. You can set the value globally or on individual modules or registers.

For some designs, turning off shift register inference improves the design performance.

## Incremental Compilation and Block-Based Design

As designs become more complex and designers work in teams, a block-based incremental design flow is often an effective design approach. In an incremental compilation flow, you can make changes to part of the design while maintaining the placement and performance of unchanged parts of the design. Design iterations are made dramatically faster by focusing new compilations on particular design partitions and merging results with previous compilation results of other partitions. You can perform optimization on individual subblocks and then preserve the results before you integrate the blocks into a final design and optimize it at the top level.

MultiPoint synthesis, which is available for certain device technologies in the Synplify Pro and Premier software, provides an automated block-based incremental synthesis flow. The MultiPoint feature manages a design hierarchy to let you design incrementally and synthesize designs that take too long for top-down synthesis of the entire project. MultiPoint synthesis allows different netlist files to be created for different sections of a design hierarchy and supports the Quartus II incremental compilation methodology. It also ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections of the design.


You can also partition your design and create different netlist files manually with the Synplify software by creating a separate project for the logic in each partition of the design. Creating different netlist files for each partition of the design also means that each partition can be independent of the others.

Hierarchical design methodologies can improve the efficiency of your design process, providing better design reuse opportunities and fewer integration problems when working in a team environment. When you use these incremental synthesis methodologies, you can take advantage of incremental compilation in the Quartus II software. You can perform placement and routing on only the changed partitions of the design, reducing place-and-route time and preserving your fitting results. Follow the guidelines in this section to help you achieve good results with these methodologies.

The following list shows the general top-down compilation flow when using these features of the Quartus II software:

1. Create Verilog HDL or VHDL design files as in the regular design flow.
2. Determine which hierarchical blocks are to be treated as separate partitions in your design.
3. Set up your design using the MultiPoint feature or separate projects so that a separate netlist file is created for each partition of the design.
4. If using separate projects, disable I/O pad insertion in the implementations for lower-level partitions.
5. Compile and map each partition in the Synplify software, making constraints as you would in the regular design flow.
6. Import the **.vqm** netlist and **.tcl** file for each partition into the Quartus II software and set up the Quartus II project(s) to use incremental compilation.

7. Compile your design in the Quartus II software and preserve the compilation results using the post-fit netlist in incremental compilation.
8. When you make design or synthesis optimization changes to part of your design, resynthesize only the changed partition to generate a new netlist and `.tcl` file. Do not regenerate netlist files for the unchanged partitions.
9. Import the new netlist and `.tcl` file into the Quartus II software and recompile the design in the Quartus II software using incremental compilation.

 For more information about creating partitions and using the incremental compilation in the Quartus II software, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.


## Creating a Design with Separate Netlist Files for Incremental Compilation

The first stage of a hierarchical or incremental design flow is to ensure that different parts of your design do not affect each other. Ensure that you have separate netlists for each partition in your design so you can take advantage of incremental compilation in the Quartus II software. If the entire design is in one netlist file, changes in one partition might affect other partitions because of possible node name changes when you resynthesize the design.

To ensure the proper functioning of the synthesis flow, create separate netlist files only for modules and entities. In addition, each module or entity requires its own design file. If two different modules are in the same design file but are defined as being part of different partitions, you cannot maintain incremental compilation since both partitions would have to be recompiled when you change one of the modules.

Altera recommends that you register all inputs and outputs of each partition. This makes logic synchronous and avoids any delay penalty on signals that cross partition boundaries.

If you use boundary tri-states in a lower-level block, the Synplify software pushes (or “bubbles”) the tri-states through the hierarchy to the top level to make use of the tri-state drivers on output pins of Altera devices. Because bubbling tri-states requires optimizing through hierarchies, lower-level tri-states are not supported with a block-based compilation methodology. Use tri-state drivers only at the external output pins of the device and in the top-level block in the hierarchy.

 For more detailed recommendations about designing your hierarchy and creating partitions, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.

You can generate multiple `.vqm` netlist files with the MultiPoint synthesis flow in the Synplify Pro and Premier software, or by manually creating separate Synplify projects and creating a black box for each block that you want to be considered as a separate design partition.

In the MultiPoint synthesis flow (Synplify Pro and Premier only), you create multiple `.vqm` netlist files from one easy-to-manage, top-level synthesis project. By using the manual black box method, you have multiple synthesis projects, which might be required for certain team-based or bottom-up designs where a single top-level project is not desired.

After you have created multiple **.vqm** files using one of these two methods, you must create the appropriate Quartus II projects to place-and-route the design.

## Using MultiPoint Synthesis with Incremental Compilation

This section describes how to generate multiple **.vqm** files using the Synplify Pro and Premier MultiPoint synthesis flow. You must first set up your constraint file and Synplify options, then apply the appropriate Compile Point settings to write multiple **.vqm** files and create design partition assignments for incremental compilation.

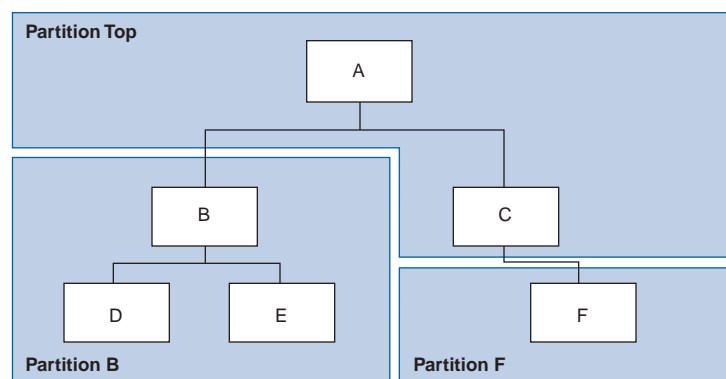
### Set Compile Points and Create Constraint Files

The MultiPoint flow lets you segment a design into smaller synthesis units, called “Compile Points.” The synthesis software treats each Compile Point as a partition for incremental mapping, which allows you to isolate and work on individual Compile Point modules as independent segments of the larger design without impacting other design modules. A design can have any number of Compile Points, and Compile Points can be nested. The top-level module is always treated as a Compile Point.

Compile Points are optimized in isolation from their parent, which can be another Compile Point or a top-level design. Each block created with a Compile Point is unaffected by critical paths or constraints on its parent or other blocks. A Compile Point is independent, with its own individual constraints. During synthesis, any Compile Points that have not yet been synthesized are synthesized before the top level. Nested Compile Points are synthesized before the parent Compile Points in which they are contained. When you apply the appropriate setting for the Compile Point, a separate netlist is created for that Compile Point, isolating that logic from any other logic in the design.

Figure 10-3 shows an example of a design hierarchy that is split into multiple partitions. The top-level block of each partition can be synthesized as a separate Compile Point.

**Figure 10-3.** Partitions in a Hierarchical Design



In this case, modules A, B, and F are Compile Points. The top-level Compile Point consists of the top-level block in the design (that is, block A in this example), including the logic that is not defined under another Compile Point. In this example, the design for top-level Compile Point A also includes the logic in one of its subblocks, C. Because block F is defined as its own Compile Point, it is not treated as part of the top-level Compile Point A. Another separate Compile Point B contains the logic in blocks B, D, and E. One netlist is created for the top-level module A and submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

Apply Compile Points to the module or architecture in the Synplify Pro SCOPE spreadsheet or in the .sdc file. You cannot set a Compile Point in the Verilog HDL or VHDL source code. You can set the constraints manually using Tcl or by editing the .sdc file, or you can use one of two methods in the GUI, as described in the following subsections.

### Defining Compile Points Using .tcl or .sdc Files

To set Compile Points using a .tcl or .sdc file, use the `define_compile_point` command, as shown in [Example 10-22](#).

#### Example 10-22. The `define_compile_point` Command

---

```
define_compile_point [-disable] {<objname>} -type {locked, partition}
```

---

In [Example 10-22](#), `objname` represents any module in the design. The Compile Point type `{locked, partition}` indicates that the Compile Point represents a partition for the Quartus II incremental compilation flow.

Each Compile Point has a set of constraint files that begin with the `define_current_design` command to set up the SCOPE environment, as follows:

```
define_current_design {<my_module>}
```

### Defining Compile Points in the Top-Level SCOPE Window

The following method requires you to separately create constraint files for the top-level and lower-level Compile Points:

1. In the top-level SCOPE window, select the **Compile Points** tab.
2. Select the modules that you want to define as Compile Points and set **Type** to **locked, partition**.
3. Manually create a constraint file for each module to set constraints for each Compile Point.

### Defining Compile Points by Creating a New SCOPE File

When you use the following process, the lower-level constraint file is created automatically:

1. On the File menu, click **New** and choose to create a new **Constraint File**. Or, click the **SCOPE** icon in the tool bar.
2. From the **Select File Type** tab of the **Create a New SCOPE File** dialog box, select **Compile Point**.

3. Select the module you want to designate as a Compile Point and click **OK**. The software automatically sets the Compile Points in the top-level constraint file and creates a lower-level constraint file for each Compile Point.


### Additional Considerations for Compile Points

To ensure that changes to a Compile Point do not affect the top-level parent module, disable the **Update Compile Point Timing Data** option in the **Implementation Options** dialog box. If this option is enabled, updates to a child module can impact the top-level module.

You can apply the `syn_allowed_resources` attribute to any Compile Point view to restrict the number of resources for a particular module.

When using Compile Points with incremental compilation, keep the following restrictions in mind:

- To use Compile Points effectively, you must provide timing constraints (timing budgeting) for each Compile Point; the more accurate the constraints, the better your results are. Constraints are not automatically budgeted, so manual time budgeting is essential. Altera recommends that you register all inputs and outputs of each partition. This avoids any logic delay penalty on signals that cross partition boundaries.
- When using the Synplify attribute `syn_useioff` to pack registers in the I/O Elements (IOEs) of Altera devices, these registers must be in the top-level module, not a lower level. Otherwise, you must allow the Quartus II software to perform I/O register packing instead of the `syn_useioff` attribute. You can use the **Fast Input Register** or **Fast Output Register** options, or set I/O timing constraints and turn on **Optimize I/O cell register placement for timing** on the Fitter Settings page of the **Settings** dialog box in the Quartus II software.
- There is no incremental synthesis support for top-level logic; any logic in the top-level is resynthesized during every compilation in the Synplify software.

 For more information about using Compile Points and setting Synplify attributes and constraints for both top-level and lower-level Compile Points, refer to the *Synopsys FPGA Synthesis User Guide* and the *Synopsys FPGA Synthesis Reference Manual* in the Synplify software.

### Creating a Quartus II Project for Compile Points and Multiple .vqm Files

During compilation, the Synplify Pro and Premier software creates a *<top-level project>.tcl* file that provides the Quartus II software with the appropriate constraints and design partition assignments, creating a partition for each **.vqm** file along with the information to set up a Quartus II project. For details about using the Tcl script generated by the Synplify software to set up your Quartus II project and pass your constraints, refer to [“Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script”](#) on page 10-16.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and optionally LogicLock™ floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design.

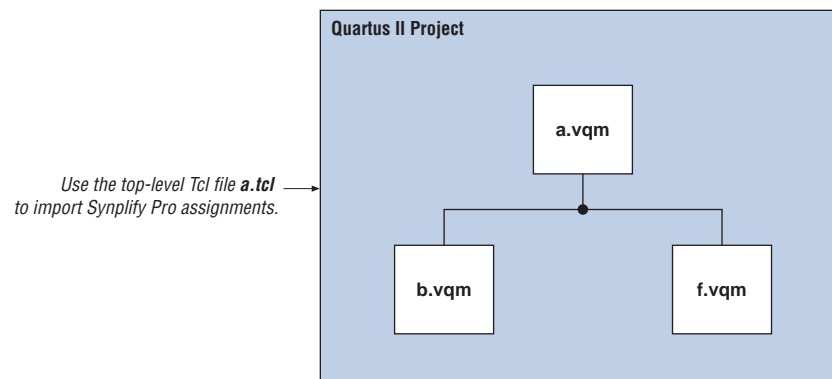
You might require a bottom-up design flow if each partition must be optimized separately, such as in certain team-based design flows. If you use this flow, Altera recommends you create a design floorplan to avoid placement conflicts between each partition. To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain placement results.

The following sections describe how to create the Quartus II projects for these two design flows.

### Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the `<top-level project>.tcl` file that contains the Synplify assignments for all partitions within the project. This method allows you to import all the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation-time reduction incremental compilation offers. Figure 10-4 shows a visual representation of the design flow for the example design in Figure 10-3 on page 10-39.

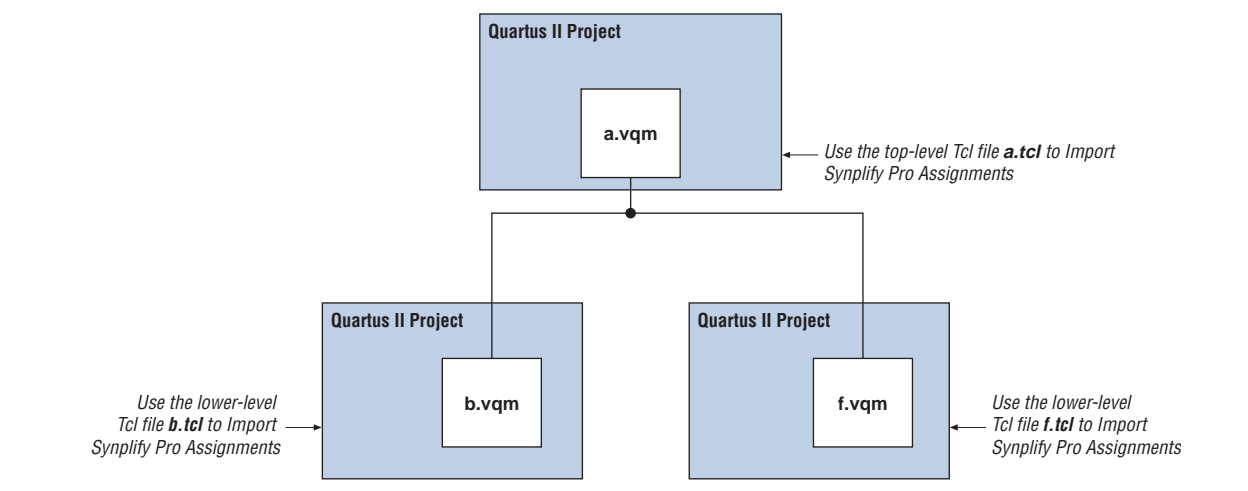
**Figure 10-4.** Design Flow Using Multiple .vqm Files with One Quartus II Project



### Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the `<lower-level compile point>.tcl` files that contain the Synplify assignments for each Compile Point. Generate multiple Quartus II projects, one for each partition and netlist in the design. The designers in the project can optimize their own partitions separately within the Quartus II software and export the results for their own partitions. Figure 10-5 shows a visual representation of the design flow for the example design in Figure 10-3 on page 10-39. You can export the optimized sub-designs and then import them into one top-level Quartus II project using incremental compilation to complete the design.

**Figure 10-5.** Design Flow Using Multiple .vqm Files with Multiple Quartus II Projects



## Creating Multiple .vqm Files for Incremental Compilation Using Separate Synplify Projects

This section describes how to manually generate multiple .vqm files for incremental compilation using black boxes and separate Synplify projects for each design partition. This manual flow is supported by versions of the Synplify software that do not include the MultiPoint Synthesis feature.

### Manually Creating Multiple .vqm Files Using Black Boxes

To create multiple .vqm files manually in the Synplify software, create a separate project for each low-level module and top-level design that you want to maintain as a separate .vqm file for an incremental compilation partition. Implement black box instantiations of lower-level partitions in your top-level project.

When synthesizing the projects for the lower-level modules, perform the following steps:

1. In the **Implementation Options** dialog box, turn on **Disable I/O Insertion** for the target technology.
2. Read the HDL files for the modules.



Modules might include black box instantiations of lower-level modules that are also maintained as separate .vqm files.

3. Add constraints with the SCOPE constraint window.
4. Enter the clock frequency to ensure that the sub-design is correctly optimized.
5. In the **Attributes** tab, set **syn\_netlist\_hierarchy** to 0.

When synthesizing the top-level design project, perform the following steps:

1. Turn off **Disable I/O Insertion** for the target technology.
2. Read the HDL files for top-level designs.
3. Create black boxes using lower-level modules in the top-level design.

4. Add constraints with the SCOPE constraint window.
5. Enter the clock frequency to ensure that the design is correctly optimized.
6. In the **Attributes** tab, set `syn_netlist_hierarchy` to 0.

The following sections describe an example of black box implementation to create separate `.vqm` files. [Figure 10-3 on page 10-39](#) shows an example of a design hierarchy that is split into multiple partitions.

The partition top contains the top-level block in the design (block A) and the logic that is not defined as part of another partition. In this example, the partition for top-level block A also includes the logic in one of its sub-blocks, C. Because block F is contained in its own partition, it is not treated as part of the top-level partition A. Another separate partition, B, contains the logic in blocks B, D, and E. In a team-based design, different engineers can work on the logic in different partitions. One netlist is created for the top-level module A and its submodule C, another netlist is created for B and its submodules D and E, while a third netlist is created for F.

To create multiple `.vqm` files for this design, follow these steps:

1. Generate a `.vqm` file for module B. Use `B.v.vhd`, `D.v.vhd`, and `E.v.vhd` as the source files.
2. Generate a `.vqm` file for module F. Use `F.v.vhd` as the source files.
3. Generate a top-level `.vqm` file for module A. Use `A.v.vhd` and `C.v.vhd` as the source files. Ensure that you use black box modules B and F, which were optimized separately in the previous steps.

### Creating Black Boxes in Verilog HDL

Any design block that is not defined in the project or included in the list of files to be read for a project are treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to create a black box for the given module. In Verilog HDL, you must provide an empty module declaration for the module that is treated as a black box.

[Example 10-23](#) shows an example of the `A.v` top-level file. Follow the same procedure for lower-level files that also contain a black box for any module beneath the current level hierarchy.

**Example 10-23.** Verilog HDL Black Box for Top-Level File A.v

---

```

module A (data_in, clk, e, ld, data_out);
    input data_in, clk, e, ld;
    output [15:0] data_out;

    wire [15:0] cnt_out;

    B U1 (.data_in (data_in), .clk(clk), .ld (ld), .data_out(cnt_out));
    F U2 (.d(cnt_out), .clk(clk), .e(e), .q(data_out));

    // Any other code in A.v goes here.
endmodule

// Empty Module Declarations of Sub-Blocks B and F follow here.
// These module declarations (including ports) are required for black
// boxes.

module B (data_in, clk, ld, data_out) /* synthesis syn_black_box */ ;
    input data_in, clk, ld;
    output [15:0] data_out;
endmodule

module F (d, clk, e, q) /* synthesis syn_black_box */ ;
    input [15:0] d;
    input clk, e;
    output [15:0] q;
endmodule

```

---

**Creating Black Boxes in VHDL**

Any design block that is not defined in the project or included in the list of files to be read for a project is treated as a black box by the software. Use the `syn_black_box` attribute to indicate that you intend to treat the given component as a black box. In VHDL, you must have a component declaration for the black box just like any other block in the design.



Although VHDL is not case-sensitive, a `.vqm` (a subset of Verilog HDL) file is case-sensitive. Entity names and their port declarations are forwarded to the `.vqm` file. Black box names and port declarations are also passed to the `.vqm` file. To prevent case-based mismatches, use the same capitalization for black box and entity declarations in VHDL designs.

**Example 10-24** shows an example of the `A.vhd` top-level file. Follow this same procedure for any lower-level files that contain a black box for any block beneath the current level of hierarchy.

**Example 10-24.** VHDL Black Box for Top-Level File A.vhd

---

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY synplify;
USE synplify.attributes.all;

ENTITY A IS
PORT ( data_in : IN INTEGER RANGE 0 TO 15;
      clk, e, ld : IN STD_LOGIC;
      data_out : OUT INTEGER RANGE 0 TO 15 );
END A;

ARCHITECTURE a_arch OF A IS

COMPONENT B PORT(
  data_in : IN INTEGER RANGE 0 TO 15;
  clk, ld : IN STD_LOGIC;
  d_out : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

COMPONENT F PORT(
  d : IN INTEGER RANGE 0 TO 15;
  clk, e: IN STD_LOGIC;
  q : OUT INTEGER RANGE 0 TO 15 );
END COMPONENT;

attribute syn_black_box of B: component is true;
attribute syn_black_box of F: component is true;

-- Other component declarations in A.vhd go here
signal cnt_out : INTEGER RANGE 0 TO 15;

BEGIN

U1 : B
PORT MAP (
  data_in => data_in,
  clk => clk,
  ld => ld,
  d_out => cnt_out );

U2 : F
PORT MAP (
  d => cnt_out,
  clk => clk,
  e => e,
  q => data_out );

-- Any other code in A.vhd goes here

END a_arch;

```

---

After you have completed the steps described in this section, you have a netlist file for each partition of the design. These files are ready for use with incremental compilation in the Quartus II software.

## Creating a Quartus II Project for Multiple .vqm Files

The Synplify software creates a .tcl file for each .vqm file that provides the Quartus II software with the appropriate constraints and information to set up a project. For details about using the Tcl script generated by the Synplify software to set up your Quartus II project and pass your constraints, refer to “Running the Quartus II Software Manually Using the Synplify-Generated Tcl Script” on page 10-16.

Depending on your design methodology, you can create one Quartus II project for all netlists (a top-down placement and routing flow) or a separate Quartus II project for each netlist (a bottom-up placement and routing flow). In a top-down incremental compilation design flow, you create design partition assignments and optional LogicLock floorplan location assignments for each partition in the design within a single Quartus II project. This methodology allows for the best quality of results and performance preservation during incremental changes to your design. You might require a bottom-up design flow where each partition must be optimized separately, such as in certain team-based design flows.

To perform a bottom-up compilation in the Quartus II software, create separate Quartus II projects and import each design partition into a top-level design using the incremental compilation export and import features to maintain the results.

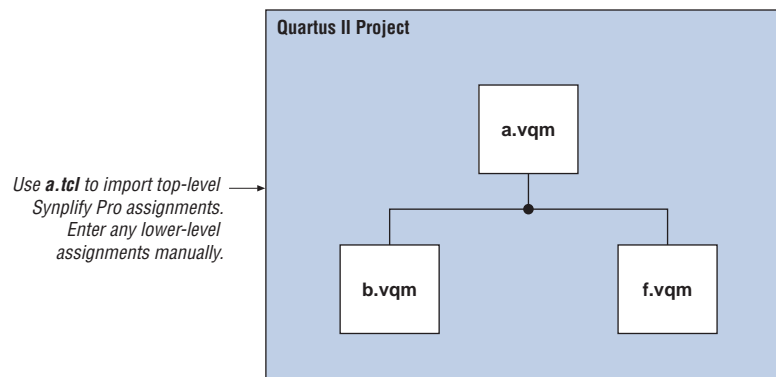
The following sections describe how to create the Quartus II projects for these two design flows.

### Creating a Single Quartus II Project for a Top-Down Incremental Compilation Flow

Use the <top-level project>.tcl file that contains the Synplify assignments for the top-level design. This method allows you to import all of the partitions into one Quartus II project and optimize all modules within the project at once, taking advantage of the performance preservation and compilation time reduction offered by incremental compilation. Figure 10-6 shows a visual representation of the design flow for the example design in Figure 10-3 on page 10-39.

All of the constraints from the top-level project are passed to the Quartus II software in the top-level .tcl file, but any constraints made in the lower-level projects within the Synplify software is not forward-annotated. Enter these constraints manually in your Quartus II project.

**Figure 10-6.** Design Flow Using Multiple .vqm Files with One Quartus II Project

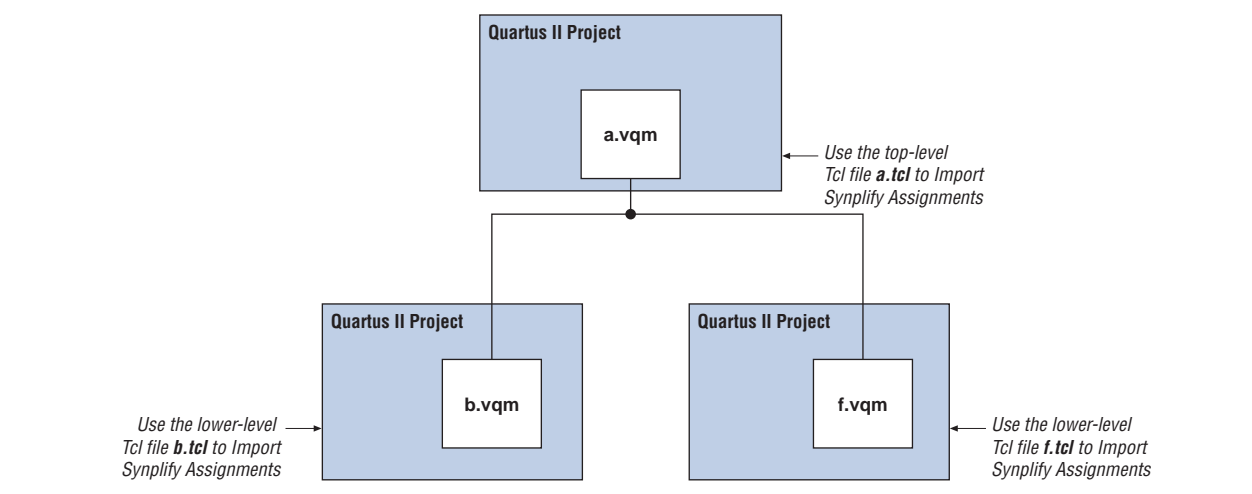


### Creating Multiple Quartus II Projects for a Bottom-Up Incremental Compilation Flow

Use the `.tcl` file that is created for each `.vqm` file by the Synplify software for each Synplify project. This method generates multiple Quartus II projects, one for each block in the design. The designers in the project can optimize their own blocks separately within the Quartus II software and export the placement of their own blocks. Figure 10-7 shows a visual representation of the design flow for the example in Figure 10-3 on page 10-39.

Designers should create a LogicLock region to create a design floorplan for each block to avoid conflicts between partitions. The top-level designer then imports all the blocks and assignments into the top-level project. This method allows each block in the design to be optimized separately and then imported into one top-level project.

**Figure 10-7.** Design Flow Using Multiple Synplify Projects and Multiple Quartus II Projects



### Performing Incremental Compilation in the Quartus II Software

In a top-down design flow using Multipoint Synthesis, the Synplify software uses the Quartus II top-level `.tcl` file to ensure that the two tools databases stay synchronized. The Tcl creates, changes, or deletes partition assignments in the Quartus II software for Compile Points that you create, change, or delete in Synplify. However, if you create, change, or delete a partition in the Quartus II software, the Synplify software does not change your Compile Point settings. Make any corresponding change in your Synplify project so that you create the correct `.vqm` files.




If you use the NativeLink integration feature described in “Using the Quartus II Software to Run the Synplify Software” on page 10-15, the Synplify software does not use any information about design partition assignments that you have set in the Quartus II software.

If you are creating netlist files using multiple Synplify projects, or if you don’t use the Synplify Pro or Premier-generated `.tcl` files to update constraints in your Quartus II project, you must ensure that your Synplify `.vqm` netlists align with your Quartus II partition settings.

After you have set up your Quartus II project with `.vqm` netlist files as separate design partitions, set the appropriate Quartus II options to preserve your compilation results. On the Assignments menu, click **Design Partitions Window**. Change the Netlist Type to **Post-Fit** to preserve the previous compilation's post-fit placement results. To preserve routing results as well, set the **Fitter Preservation Level to Placement and Routing**. If you do not make these settings, the Quartus II software does not reuse the placement or routing results from the previous compilation.

You can take advantage of incremental compilation with your Synplify design to reduce compilation time in the Quartus II software and preserve the results for unchanged design blocks.

 For more information about using Quartus II incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

## Conclusion

Advanced synthesis is an important part of the design flow. Taking advantage of the Synopsys Synplify and Altera Quartus II design flows allow you to control how your design files are prepared for the Quartus II place-and-route process, as well as improve performance and optimize a design for use with Altera devices. The methodologies outlined in this chapter can help optimize a design to achieve performance goals and save design time.

## Referenced Documents


This chapter references the following documents:

- *Altera Constraints, Attributes, and Options* chapter in the *Synopsys FPGA Synthesis Reference Manual*
- *Altera I/O Standards* in the *Synopsys FPGA Synthesis Reference Manual*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*
- *Design Recommendations for Altera Devices* chapter in volume 1 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Section III. Formal Verification* in volume 3 of the *Quartus II Handbook*

## Document Revision History

**Table 10-5.** Document Revision History

Date and Version	Changes Made	Summary of Changes
November 2009 v9.1.0	<ul style="list-style-type: none"> <li>■ Minor updates for the Quartus II software version 9.1 release</li> </ul>	Updated for the Quartus II software version 9.1 release
March 2009 v9.0.0	<ul style="list-style-type: none"> <li>■ Added new section “<a href="#">Exporting Designs to the Quartus II Software Using NativeLink Integration</a>” on page 10–14</li> <li>■ Minor updates for the Quartus II software version 9.0 release</li> <li>■ Chapter 10 was previously Chapter 9 in software version 8.1</li> </ul>	Updated for the Quartus II software version 9.0 release
November 2008 v8.1.0	<ul style="list-style-type: none"> <li>■ Changed to 8-1/2 x 11 page size</li> <li>■ Changed the chapter title from “Synplicity Synplify &amp; Synplify Pro Support” to “Synopsys Synplify Support”</li> <li>■ Replaced references to Synplicity with references to Synopsys</li> <li>■ Added information about Synplify Premier</li> <li>■ Updated supported device list</li> <li>■ Added SystemVerilog information to <a href="#">Figure 10-1</a></li> </ul>	Updated for the Quartus II software version 8.1 release and the Synplify software version 9.6.2 release.
May 2008 v8.0.0	<ul style="list-style-type: none"> <li>■ Updated supported device list</li> <li>■ Updated constraint annotation information for the TimeQuest Timing Analyzer</li> <li>■ Updated RAM and MAC constraint limitations</li> <li>■ Revised Table 9-1</li> <li>■ Added new section “Changing Synplify’s Default Behavior for Instantiated Altera Megafunctions”</li> <li>■ Added new section “Instantiating Intellectual Property Using the MegaWizard Plug-In Manager and IP Toolbench”</li> <li>■ Added new section “Including Files for Quartus II Placement and Routing Only”</li> <li>■ Added new section “Additional Considerations for Compile Points”</li> <li>■ Removed section “Apply the LogicLock Attributes”</li> <li>■ Modified <a href="#">Figure 9-4</a>, <a href="#">9-43</a>, <a href="#">9-47</a>, and <a href="#">9-48</a></li> <li>■ Added new section “Performing Incremental Compilation in the Quartus II Software”</li> <li>■ Numerous text changes and additions throughout the chapter</li> <li>■ Renamed several sections</li> <li>■ Updated “Referenced Documents” section</li> </ul>	Updated for Quartus II software release, version 8.0, and the Synplify software release, version 9.4.

 For previous versions of the *Quartus II Handbook*, refer to the [Quartus II Handbook Archive](#).