16. Quartus II Integrated Synthesis

QII51008-13.1.0

This chapter describes the Integrated Synthesis design flow and provides scripting techniques for applying all the options and settings described in this chapter.

As programmable logic designs become more complex and require increased performance, advanced synthesis becomes an important part of a design flow. The Altera® Quartus® II software includes advanced Integrated Synthesis that fully supports VHDL, Verilog HDL, and Altera-specific design entry languages, and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use solution.

This chapter contains the following sections:

- "Design Flow" on page 16–1
- "Language Support" on page 16–4
- "Incremental Compilation" on page 16–21
- "Quartus II Synthesis Options" on page 16–23
- "Analyzing Synthesis Results" on page 16–73
- "Analyzing and Controlling Synthesis Messages" on page 16–74
- "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 16–78
- "Scripting Support" on page 16–84

For examples of Verilog HDL and VHDL code synthesized for specific logic functions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*. For more information about coding with primitives that describe specific low-level functions in Altera devices, refer to the *Designing With Low-Level Primitives User Guide*.

Design Flow

The Quartus II Analysis & Synthesis stage of the compilation flow runs Integrated Synthesis, which fully supports Verilog HDL, VHDL, and Altera-specific languages, and major features of the SystemVerilog language. For more information, refer to "Language Support" on page 16–4.

In the synthesis stage of the compilation flow, the Quartus II software performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic in device resources such as logic elements (LEs) or adaptive logic modules (ALMs), and other dedicated logic blocks. The synthesis stage generates a single project database that integrates all your design files in a project (including any netlists from third-party synthesis tools).

© 2013 Altera Corporation. All rights reserved. ALTERA, ARRIA, CYCLONE, HARDCOPY, MAX, MEGACORE, NIOS, QUARTUS and STRATIX words and logos are trademarks of Altera Corporation and registered in the U.S. Patent and Trademark Office and in other countries. All other words and logos identified as trademarks or service marks are the property of their respective holders as described at www.altera.com/common/legal.html. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

You can use Analysis & Synthesis to perform the following compilation processes:

- **Analyze Current File—**parses your current design source file to check for syntax errors. This command does not report many semantic errors that require further design synthesis. To perform this analysis, on the Processing menu, click **Analyze Current File**.
- **Analysis & Elaboration—**checks your design for syntax and semantic errors and performs elaboration to identify your design hierarchy. To perform Analysis & Elaboration, on the Processing menu, point to **Start,** and then click **Start Analysis & Elaboration**.
- **Hierarchy Elaboration—**parses HDL designs and generates a skeleton of hierarchies. Hierarchy Elaboration is similar to the Analysis & Elaboration flow, but without any elaborated logic, thus making it much faster to generate.

  For more information about the Hierarchy Elaboration flow, refer to *Start Hierarchy Elaboration Command (Processing Menu)* in Quartus II Help.

- **Analysis & Synthesis—**performs complete Analysis & Synthesis on a design, including technology mapping. To perform Analysis & Synthesis, on the Processing menu, point to **Start,** and then click **Start Analysis & Synthesis**.

The Quartus II Integrated Synthesis design and compilation flow consists of the following steps:

1. Create a project in the Quartus II software and specify the general project information, including the top-level design entity name.
2. Create design files in the Quartus II software or with a text editor.
3. On the Project menu, click **Add/Remove Files in Project** and add all design files to your Quartus II project using the **Files** page of the **Settings** dialog box.
4. Specify Compiler settings that control the compilation and optimization of your design during synthesis and fitting. For synthesis settings, refer to "Quartus II Synthesis Options" on page 16–23.
5. Add timing constraints to specify the timing requirements.

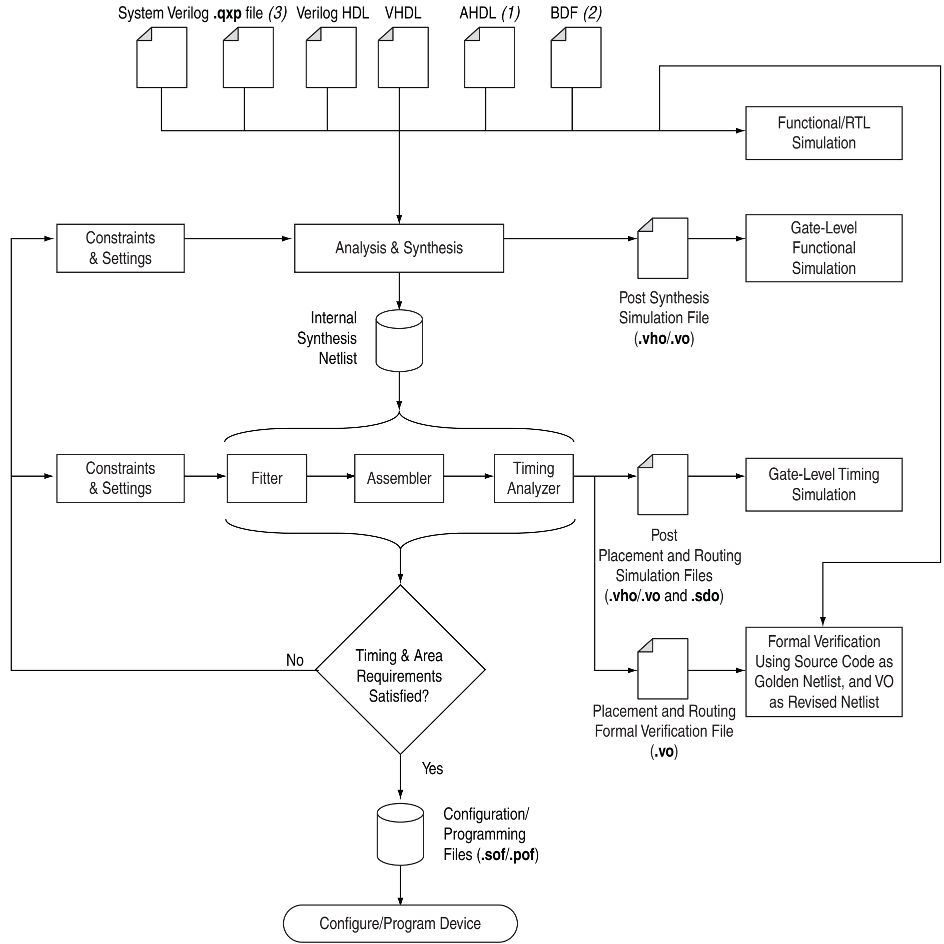
   To partition your design to reduce compilation time, refer to "Incremental Compilation" on page 16–21.

6. Compile your design. To synthesize your design, on the Processing menu, point to **Start**, and then click **Start Analysis & Synthesis**. To run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis, click **Start Compilation** on the Processing menu.
7. After obtaining synthesis and placement and routing results that meet your requirements, program or configure your Altera device.

Integrated Synthesis generates netlists that enable you to perform functional simulation or gate-level timing simulation, timing analysis, and formal verification.

Figure 16–1 shows the basic design flow using Quartus II Integrated Synthesis.

Figure 16–1. Quartus II Design Flow Using Quartus II Integrated Synthesis



Notes to Figure 16–1:

(1) AHDL stands for the Altera Hardware Description Language.

(2) BDF stands for the Altera schematic Block Design File (**.bdf**).

(3) The Quartus II Exported Partition File (**.qxp**) is a precompiled netlist that you can use as a design source file. For more information about using **.qxp** as a design source file, refer to “Quartus II Exported Partition File as Source” on page 16–22.

For an overall summary of features in the Quartus II software, refer to the *Introduction to the Quartus II Software* manual.

For more information about Quartus II projects and the compilation flow, refer to *Managing Files in a Project* and *About Compilation Flows* in Quartus II Help.

Language Support

This section describes Quartus II Integrated Synthesis support for HDL, schematic design entry, graphical state machine entry, and how to specify the Verilog HDL or VHDL language version in your design. This section also describes language features such as Verilog HDL macros, initial constructs and memory system tasks, and VHDL libraries. "Design Libraries" on page 16–12 describes how to compile and reference design units in custom libraries, and "Using Parameters/Generics" on page 16–16 describes how to use parameters or generics and pass them between languages.

To ensure that the Quartus II software reads all associated project files, add each file to your Quartus II project by clicking **Add/Remove Files in Project** on the Project menu. You can add design files to your project. You can mix all supported languages and netlists generated by third-party synthesis tools in a single Quartus II project.

You can also use the available templates in the Quartus II Text Editor for various Verilog and VHDL features. For more information, refer to *Insert Template Dialog Box* in Quartus II Help.

Verilog HDL Support

The Quartus II Compiler's Analysis & Synthesis module supports the following Verilog HDL standards:

- Verilog-1995 (IEEE Standard 1364-1995)
- Verilog-2001 (IEEE Standard 1364-2001)
- SystemVerilog-2005 (IEEE Standard 1800-2005) (the Compiler does not support all constructs)

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified. The Quartus II Compiler uses the Verilog-2001 standard by default for files that have the extension **.v**, and the SystemVerilog standard for files that have the extension **.sv**.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. For more information, refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 16–85.

The Quartus II software support for Verilog HDL is case sensitive in accordance with the Verilog HDL standard. The Quartus II software supports the compiler directive `` `define``, in accordance with the Verilog HDL standard.

The Quartus II software supports the `include` compiler directive to include files with absolute paths (with either "/" or "\" as the separator), or relative paths. When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the Quartus II software cannot find the file, the software then searches relative to all user libraries and then relative to the directory location of the current file.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 16–27.

For more information about Verilog HDL, refer to *About Verilog HDL* in Quartus II Help.

For more information about Quartus II Verilog HDL support, refer to *Quartus II Verilog HDL Support* in Quartus II Help.

For more information about specifying a default Verilog HDL version for all files, refer to *Specifying Verilog Input Settings* in Quartus II Help.

For more information about controlling the Verilog HDL version that compiles your design in a design file with the `VERILOG_INPUT_VERSION` synthesis directive, refer to *verilog_input_version Synthesis Directive* in Quartus II Help.

For more information about Verilog HDL synthesis attributes and directives, refer to *Verilog HDL Synthesis Attributes and Directives* in Quartus II Help.

Verilog HDL Configuration

Verilog HDL configuration is a set of rules that specify the source code for particular instances.

Verilog HDL configuration allows you to perform the following tasks:

- Specify a library search order for resolving cell instances (as does a library mapping file)
- Specify overrides to the logical library search order for specified instances
- Specify overrides to the logical library search order for all instances of specified cells

For more information about these tasks, refer to Table 16–1.

Configuration Syntax

A configuration contains the following statements:

Example 16–1. Verilog HDL Configuration Statement

```
config config_identifier;
design [library_identifier.]cell_identifier;
config_rule_statement;
endconfig
```

Where:

- `config`—the keyword that begins the configuration.
- `config_identifier`—the name you enter for the configuration.
- `design`—the keyword that starts a design statement for specifying the top of the design.
- `[library_identifier.]cell_identifier`—specifies the top-level module (or top-level modules) in the design and the logical library for this module (modules).
- `config_rule_statement`—one or more of the following clauses: default, instance, or cell. For more information, refer to Table 16–1.
- `endconfig`—the keyword that ends a configuration.

Table 16–1 lists the type of clauses for the `config_rule_statement` keyword:

Table 16–1. Type of Clauses for the config_rule_statement Keyword

Clause Type	Description
`default`	Specifies the logical libraries to search to resolve a default cell instance. A default cell instance is an instance in the design that is not specified in a subsequent instance or cell clause in the configuration. You specify these libraries with the `liblist` keyword. The following is an example of a default clause: `default liblist lib1 lib2;` Also specifies resolving default instances in the logical libraries (`lib1` and `lib2`). Because libraries are inherited, some simulators (for example, VCS) also search the default (or current) library as well after the searching the logical libraries (`lib1` and `lib2`).

`instance`	Specifies a specific instance. The specified instance clause depends on the use of the following keywords:
	■ `liblist`—specifies the logical libraries to search to resolve the instance.
	■ `use`—specifies that the instance is an instance of the specified cell in the specified logical library.
	The following are examples of `instance` clauses:
	`instance top.dev1 liblist lib1 lib2;`
	This instance clause specifies to resolve `instance top.dev1` with the cells assigned to logical libraries `lib1` and `lib2`;
	`instance top.dev1.gm1 use lib2.gizmult;`
	This instance clause specifies that `top.dev1.gm1` is an instance of the cell named `gizmult` in logical library `lib2`.

`cell`	A `cell` clause is similar to an `instance` clause, except that the `cell` clause specifies all instances of a cell definition instead of specifying a particular instance. What it specifies depends on the use of the `liblist` or `use` keywords: ■ `liblist`—specifies the logical libraries to search to resolve all instances of the cell. ■ `use`—the specified cell's definition is in the specified library.

Hierarchical Configurations

A design can have more than one configuration. For example, you can define a configuration that specifies the source code you use in particular instances in a sub hierarchy, then define a configuration for a higher level of the design.

Suppose, for example, a sub hierarchy of a design is an eight-bit adder and the RTL Verilog code describes the adder in a logical library named `rtllib` and the gate-level code describes the adder in a logical library named `gatelib`. If you want to use the gate-level code for the 0 (zero) bit of the adder and the RTL level code for the other seven bits, the configuration might appear as shown in Example 16–2:

Example 16–2.

```
config cfg1;
design aLib.eight_adder;
default liblist rtllib;
instance adder.fulladd0 liblist gatelib;
endconfig
```

If you are instantiating this eight-bit adder eight times to create a 64-bit adder, use configuration `cfg1` for the first instance of the eight-bit adder, but not in any other instance. A configuration that would perform this function is shown in Example 16–3:

Example 16–3.

```
config cfg2;
design bLib.64_adder;
default liblist bLib;
instance top.64add0 use work.cfg1:config;
endconfig
```

The name of the unbound module may be different than the name of the cell that is bounded to the instance.

Suffix :config

To distinguish between a module by the same name, use the optional extension `:config` to refer to configuration names. For example, you can always refer to a `cfg2` configuration as `cfg2:config` (even if the `cfg2` module does not exist).

SystemVerilog Support

The Quartus II software supports the SystemVerilog constructs.

Designs written to support the Verilog-2001 standard might not compile with the SystemVerilog setting because the SystemVerilog standard has several new reserved keywords.

For more information about the supported SystemVerilog constructs and the supported Verilog-2001 features, refer to *Quartus II Support for SystemVerilog* and *Quartus II Support for Verilog 2001* in Quartus II Help.

Initial Constructs and Memory System Tasks

The Quartus II software infers power-up conditions from Verilog HDL `initial` constructs. The Quartus II software also creates power-up settings for variables, including RAM blocks. If the Quartus II software encounters nonsynthesizable constructs in an `initial` block, it generates an error. To avoid such errors, enclose nonsynthesizable constructs (such as those intended only for simulation) in `translate_off` and `translate_on` synthesis directives, as described in "Translate Off and On / Synthesis Off and On" on page 16–64. Synthesis of initial constructs enables the power-up state of the synthesized design to match the power-up state of the original HDL code in simulation. For more information, refer to "Power-Up Level" on page 16–40.

Initial blocks do not infer power-up conditions in some third-party EDA synthesis tools. If you convert between synthesis tools, you must set your power-up conditions correctly.

Quartus II Integrated Synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories. Example 16–4 shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 16–4. Verilog HDL Code: Initializing RAM with the readmemb Command

```
reg [7:0] ram[0:15];
initial
begin
$readmemb("ram.txt", ram);
end
```

When creating a text file to use for memory initialization, specify the address using the format @<*location*> on a new line, and then specify the memory word such as `110101` or `abcde` on the next line. Example 16–5 shows a portion of a Memory Initialization File (**.mif**) for the RAM in Example 16–4.

Example 16–5. Text File Format: Initializing RAM with the readmemb Command

```
@0
00000000
@1
00000001
@2
00000010
…
@e
00001110
@f
00001111
```

Verilog HDL Macros

The Quartus II software fully supports Verilog HDL macros, which you can define with the `'define` compiler directive in your source code. You can also define macros in the Quartus II software or on the command line.

Setting a Verilog HDL Macro Default Value in the Quartus II Software

To specify a macro in the Quartus II software, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Verilog HDL Input**.
3. Under **Verilog HDL macro**, type the macro name in the **Name** box and the value in the **Setting** box.
4. Click **Add**.

Setting a Verilog HDL Macro Default Value on the Command Line

To set a default value for a Verilog HDL macro on the command line, use the `--verilog_macro` option, as shown in Example 16–6.

Example 16–6. Command Syntax for Specifying a Verilog HDL Macro

```
quartus_map <Design name> --verilog_macro= "<Macro name>=<Macro setting>" ↵
```

The command in Example 16–7 has the same effect as specifying `` `define a 2 `` in the Verilog HDL source code.

Example 16–7. Specifying a Verilog HDL Macro a = 2

```
quartus_map my_design --verilog_macro="a=2" ↵
```

To specify multiple macros, you can repeat the option more than once, as in Example 16–8.

Example 16–8. Specifying Verilog HDL Macros a = 2 and b = 3

```
quartus_map my_design --verilog_macro="a=2" --verilog_macro="b=3" ↵
```

VHDL Support

The Quartus II Compiler's Analysis & Synthesis module supports the following VHDL standards:

- VHDL 1987 (IEEE Standard 1076-1987)
- VHDL 1993 (IEEE Standard 1076-1993)
- VHDL 2008 (IEEE Standard 1076-2008)

The Quartus II Compiler uses the VHDL 1993 standard by default for files that have the extension **.vhdl** or **.vhd**.

The VHDL code samples provided in this chapter follow the VHDL 1993 standard.

To specify a default VHDL version for all files, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **VHDL Input**.
3. On the **VHDL Input** page, under **VHDL version**, select the appropriate version, and then click **OK**.

To override the default VHDL version for each VHDL design file, follow these steps:

1. On the Project menu, click **Add/Remove Files in Project**.
2. On the **Files** page, select the appropriate file in the list, and then click **Properties**.
3. In the HDL version list, select **VHDL_2008**, **VHDL_1993**, or **VHDL_1987,** and then click **OK**.

You can also specify the VHDL version that compiles your design for each design file with the `VHDL_INPUT_VERSION` synthesis directive, as shown in Example 16–9. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 16–9. Controlling the VHDL Input Version with a Synthesis Directive

```
--synthesis VHDL_INPUT_VERSION <language version>
```

Example 16–10. VHDL 2008—Controlling the VHDL Input Version with a Synthesis Directive

```
/* synthesis VHDL_INPUT_VERSION <language version> */
```

The variable <*language version*> requires one of the following values:

- `VHDL_1987`
- `VHDL_1993`
- `VHDL_2008`

When the Quartus II software reads a `VHDL_INPUT_VERSION` synthesis directive, it changes the current language version as specified until after the file or until it reaches the next `VHDL_INPUT_VERSION` directive.

You cannot change the language version in a VHDL design unit.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 16–27.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. For more information, refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 16–85.

The Quartus II software reads default values for registered signals defined in the VHDL code and converts the default values into power-up level settings. This enables the power-up state of the synthesized design to match, as closely as possible, the power-up state of the original HDL code in simulation. For more information, refer to "Power-Up Level" on page 16–40.

VHDL-2008 Support

The Quartus II software contains support for VHDL 2008 with constructs defined in the IEEE Standard 1076-2008 version of the *IEEE Standard VHDL Language Reference Manual*.

For more information, refer to *Quartus II Support for VHDL 2008* in Quartus II Help.

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and several vendor-specific VHDL libraries. For information about organizing your own design units into custom libraries, refer to “Design Libraries” on page 16–12.

The IEEE library includes the standard VHDL packages `std_logic_1164`, `numeric_std`, `numeric_bit`, and `math_real`. The STD library is part of the VHDL language standard and includes the packages `standard` (included in every project by default) and `textio`. For compatibility with older designs, the Quartus II software also supports the following vendor-specific packages and libraries:

- Synopsys packages such as `std_logic_arith` and `std_logic_unsigned` in the IEEE library
- Mentor Graphics® packages such as `std_logic_arith` in the ARITHMETIC library
- Altera primitive packages `altera_primitives_components` (for primitives such as `GLOBAL` and `DFFE`) and `maxplus2` (for legacy support of MAX+PLUS® II primitives) in the ALTERA library
- Altera megafunction packages `altera_mf_components` and `stratixgx_mf_components` in the ALTERA_MF library (for Altera-specific megafunctions including LCELL), and `lpm_components` in the LPM library for library of parameterized modules (LPM) functions.

Altera recommends that you import component declarations for Altera primitives such as `GLOBAL` and `DFFE` from the `altera_primitives_components` package and not the `altera_mf_components` package.

VHDL wait Constructs

The Quartus II software supports one VHDL `wait until` statement per process block. However, the Quartus II software does not support other VHDL wait constructs, such as `wait for` and `wait on` statements, or processes with multiple `wait` statements.

Example 16–11 is a VHDL code example of a supported `wait until` construct.

Example 16–11. VHDL Code: Supported wait until Construct

```
architecture dff_arch of ls_dff is
begin
output: process begin
wait until (CLK'event and CLK='1');
Q <= D;
Qbar <= not D;
end process output;
end dff_arch;
```

AHDL Support

The Quartus II Compiler's Analysis & Synthesis module fully supports the Altera Hardware Description Language (AHDL).

AHDL designs use Text Design Files (**.tdf**). You can import AHDL Include Files (**.inc**) into a **.tdf** with an AHDL `include` statement. Altera provides **.inc** files for all megafunctions shipped with the Quartus II software.

The AHDL language does not support the synthesis directives or attributes in this chapter.

For more information about AHDL, refer to *About AHDL* in the Quartus II Help.

Schematic Design Entry Support

The Quartus II Compiler's Analysis & Synthesis module fully supports **.bdf** for schematic design entry.

Schematic entry methods do not support the synthesis directives or attributes in this chapter.

For information about creating and editing schematic designs, refer to *About Schematic Design Entry* in Quartus II Help.

State Machine Editor

The Quartus II software supports graphical state machine entry. To create a new finite state machine (FSM) design, on the File menu, click **New**. In the **New** dialog box, expand the **Design Files** list, and then select **State Machine File**.

For more information about the State Machine Editor, refer to *About the State Machine Editor* in Quartus II Help.

Design Libraries

By default, the Quartus II software compiles all design files into the work library. If you do not specify a design library, if a file refers to a library that does not exist, or if the referenced library does not contain a referenced design unit, the Quartus II software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup, but you have the option of creating separate custom design libraries.

To compile your design files into specific libraries (for example, when you have two or more functionally different design entities that share the same name), you can specify a destination library for each design file in various ways, as described in the following subsections:

- "Specifying a Destination Library Name in the Settings Dialog Box" on page 16–13
- "Specifying a Destination Library Name in the Quartus II Settings File or with Tcl" on page 16–13

When the Quartus II Compiler analyzes the file, it stores the analyzed design units in the destination library of the file.

A design can contain two or more entities with the same name if the Quartus II software compiles the entities into separate libraries.

When compiling a design instance, the Quartus II software initially searches for the entity in the library associated with the instance (which is the work library if you do not specify any library). If the Quartus II software could not locate the entity definition, the software searches for a unique entity definition in all design libraries. If the Quartus II software finds more than one entity with the same name, the software generates an error. If your design uses multiple entities with the same name, you must compile the entities into separate libraries.

In VHDL, you can associate an instance with an entity in several ways, as described in "Mapping a VHDL Instance to an Entity in a Specific Library" on page 16–14. In Verilog HDL, BDF schematic entry, AHDL, VQM and EDIF netlists, you can use different libraries for each of the entities that have the same name, and compile the instantiation into the same library as the appropriate entity.

Specifying a Destination Library Name in the Settings Dialog Box

To specify a library name for one of your design files, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Files**.
3. Select the file in the **File Name** list.
4. Click **Properties**.
5. In the **File Properties** dialog box, select the type of design file from the **Type** list.
6. Type the library name in the **Library** field.
7. Click **OK**.

Specifying a Destination Library Name in the Quartus II Settings File or with Tcl

You can specify the library name with the `-library` option to the <*language type*>`_FILE` assignment in the Quartus II Settings File (**.qsf**) or with Tcl commands.

For example, the following assignments specify that the Quartus II software analyzes the **my_file.vhd** and stores its contents (design units) in the VHDL library **my_lib**, and then analyzes the Verilog HDL file **my_header_file.h** and stores its contents in a library called **another_lib**. Refer to Example 16–12.

Example 16–12. Specifying a Destination Library Name

```
set_global_assignment -name VHDL_FILE my_file.vhd -library my_lib
set_global_assignment -name VERILOG_FILE my_header_file.h -library another_lib
```

For more information about Tcl scripting, refer to "Scripting Support" on page 16–84.

Specifying a Destination Library Name in a VHDL File

You can use the `library` synthesis directive to specify a library name in your VHDL source file. This directive takes the name of the destination library as a single string argument. Specify the `library` directive in a VHDL comment before the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), with one of the supported keywords for synthesis directives, that is, `altera`, `synthesis`, `pragma`, `synopsys`, or `exemplar`.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 16–27.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file in the **Settings** dialog box, any existing **.qsf** setting, any setting made through the Tcl interface, or any prior `library` directive in the current file. The directive remains effective until the end of the file or the next `library` synthesis directive.

Example 16–13 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 16–13. Using the Library Synthesis Directive

```
-- synthesis library my_lib
library ieee;
use ieee.std_logic_1164.all;
entity my_entity(...)
end entity my_entity;
```

You can specify a single destination library for all your design units in a given source file by specifying the library name in the **Settings** dialog box, editing the **.qsf**, or using the Tcl interface. To organize your design units in a single file into different libraries rather than just a single library, you can use the `library` directive to change the destination VHDL library in a source file.

The Quartus II software generates an error if you use the library directive in a design unit.

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides several ways to map or bind an instance to an entity in a specific library, as described in the following subsections.

Direct Entity Instantiation

In the direct entity instantiation method, the instantiation refers to an entity in a specific library, as shown in Example 16–14.

Example 16–14. VHDL Code: Direct Entity Instantiation

```
entity entity1 is
port(...);
end entity entity1;

architecture arch of entity1 is
begin
inst: entity lib1.foo
port map(...);
end architecture arch;
```

Component Instantiation—Explicit Binding Instantiation

You can bind a component to an entity in several mechanisms. In an explicit binding indication, you bind a component instance to a specific entity, as shown in Example 16–15.

Example 16–15. VHDL Code: Binding Instantiation

```
entity entity1 is
port(...);
end entity entity1;

package components is
component entity1 is
port map (...);
end component entity1;
end package components;

entity top_entity is
port(...);
end entity top_entity;

use lib1.components.all;
architecture arch of top_entity is
-- Explicitly bind instance I1 to entity1 from lib1
for I1: entity1 use entity lib1.entity1
port map(...);
end for;
begin
I1: entity1 port map(...);
end architecture arch;
```

Component Instantiation—Default Binding

If you do not provide an explicit binding indication, the Quartus II software binds a component instance to the nearest visible entity with the same name. If no such entity is visible in the current scope, the Quartus II software binds the instance to the entity in the library in which you declare the component. For example, if you declare the component in a package in the MY_LIB library, an instance of the component binds to the entity in the MY_LIB library. The code examples in Example 16–16 and Example 16–17 show this instantiation method:

Example 16–16. VHDL Code: Default Binding to the Entity in the Same Library as the Component Declaration

```
use mylib.pkg.foo; -- import component declaration from package “pkg” in
                   -- library “mylib”
architecture rtl of top
...
begin
-- This instance will be bound to entity “foo” in library “mylib”
inst: foo
port map(...);
end architecture rtl;
```

Example 16–17. VHDL Code: Default Binding to the Directly Visible Entity

```
use mylib.foo; -- make entity “foo” in library “mylib” directly visible
architecture rtl of top
component foo is
generic (...)
port (...);
end component;
begin
-- This instance will be bound to entity “foo” in library “mylib”
inst: foo
port map(...);
end architecture rtl;
```

Using Parameters/Generics

This section describes how the Quartus II software supports parameters (known as generics in VHDL) and how you can pass these parameters between design languages.

You can enter default parameter values for your design in the **Default Parameters** page under the **Analysis & Synthesis Settings** page in the **Settings** dialog box. Default parameters enable you to add, change, and delete global parameters for the current assignment. In AHDL, the Quartus II software inherits parameters, so any default parameters apply to all AHDL instances in your design. You can also specify parameters for instantiated modules in a **.bdf**. To specify parameters in a **.bdf** instance, double-click the parameter value box for the instance symbol, or right-click the symbol and click **Properties**, and then click the **Parameters** tab. For more information about the GUI-based entry methods, the interpretation of parameter values, and format recommendations, refer to “Setting Default Parameter Values and BDF Instance Parameter Values” on page 16–17.

You can specify parameters for instantiated modules in your design source files with the provided syntax for your chosen language. Some designs instantiate entities in a different language; for example, they might instantiate a VHDL entity from a Verilog HDL design file. You can pass parameters or generics between VHDL, Verilog HDL, AHDL, and BDF schematic entry, and from EDIF or VQM to any of these languages. You do not require an additional procedure to pass parameters from one language to another. However, sometimes you must specify the type of parameter you are passing. In those cases, you must follow certain guidelines to ensure that the Quartus II software correctly interprets the parameter value. For more information about parameter type rules, refer to "Passing Parameters Between Two Design Languages" on page 16–19.

Setting Default Parameter Values and BDF Instance Parameter Values

Default parameter values and BDF instance parameter values do not have an explicitly declared type. Usually, the Quartus II software can correctly infer the type from the value without ambiguity. For example, the Quartus II software interprets "`ABC`" as a string, `123` as an integer, and `15.4` as a floating-point value. In other cases, such as when the instantiated subdesign language is VHDL, the Quartus II software uses the type of the parameter, generic, or both in the instantiated entity to determine how to interpret the value, so that the Quartus II software interprets a value of `123` as a string if the VHDL parameter is of a type string. In addition, you can set the parameter value in a format that is legal in the language of the instantiated entity. For example, to pass an unsized bit literal value from **.bdf** to Verilog HDL, you can use `'1` as the parameter value, and to pass a 4-bit binary vector from **.bdf** to Verilog HDL, you can use `4'b1111` as the parameter value.

In a few cases, the Quartus II software cannot infer the correct type of parameter value. To avoid ambiguity, specify the parameter value in a type-encoded format in which the first or first and second characters of the parameter indicate the type of the parameter, and the rest of the string indicates the value in a quoted sub-string. For example, to pass a binary string `1001` from **.bdf** to Verilog HDL, you cannot use the value `1001`, because the Quartus II software interprets it as a decimal value. You also cannot use the string `"1001"` because the Quartus II software interprets it as an ASCII string. You must use the type-encoded string `B"1001"` for the Quartus II software to correctly interpret the parameter value.

Table 16–2 lists valid parameter strings and how the Quartus II software interprets the parameter strings. Use the type-encoded format only when necessary to resolve ambiguity.

Table 16–2. Valid Parameter Strings and Interpretations

Parameter String	Quartus II Parameter Type, Format, and Value
`S"abc"`, `s"abc"`	String value abc
`"abc123"`, `"123abc"`	String value abc123 or 123abc
`F"12.3"`, `f"12.3"`	Floating point number 12.3
`-5.4`	Floating point number -5.4
`D"123"`, `d"123"`	Decimal number 123
`123`, `-123`	Decimal number 123, -123
`X"ff"`, `H"ff"`	Hexadecimal value FF
`Q"77"`, `O"77"`	Octal value 77
`B"1010"`, `b"1010"`	Unsigned binary value 1010
`SB"1010"`, `sb"1010"`	Signed binary value 1010
`R"1"`, `R"0"`, `R"X"`, `R"Z"`, `r"1"`, `r"0"`, `r"X"`, `r"Z"`	Unsized bit literal
`E"apple"`, `e"apple"`	Enumeration type, value name is apple
`P"1 unit"`	Physical literal, the value is (1, unit)
`A(...)`, `a(...)`	Array type or record type. The string (...) determines the array type or record type content

You can select the parameter type for global parameters or global constants with the pull-down list in the **Parameter** tab of the **Symbol Properties** dialog box. If you do not specify the parameter type, the Quartus II software interprets the parameter value and defines the parameter type. You must specify parameter type with the pull-down list to avoid ambiguity.

If you open a **.bdf** in the Quartus II software, the software automatically updates the parameter types of old symbol blocks by interpreting the parameter value based on the language-independent format. If the Quartus II software does not recognize the parameter value type, the software sets the parameter type as **untyped**.

The Quartus II software supports the following parameter types:

- **Unsigned Integer**
- **Signed Integer**
- **Unsigned Binary**
- **Signed Binary**
- **Octal**
- **Hexadecimal**
- **Float**
- **Enum**
- **String**

- **Boolean**
- **Char**
- **Untyped/Auto**

Passing Parameters Between Two Design Languages

When passing a parameter between two different languages, a design block that is higher in the design hierarchy instantiates a lower-level subdesign block and provides parameter information. The subdesign language (the design entity that you instantiate) must correctly interpret the parameter. Based on the information provided by the higher-level design and the value format, and sometimes by the parameter type of the subdesign entity, the Quartus II software interprets the type and value of the passed parameter.

When passing a parameter whose value is an enumerated type value or literal from a language that does not support enumerated types to one that does (for example, from Verilog HDL to VHDL), you must ensure that the enumeration literal is in the correct spelling in the language of the higher-level design block (block that is higher in the hierarchy). The Quartus II software passes the parameter value as a string literal, and the language of the lower-level design correctly convert the string literal into the correct enumeration literal.

If the language of the lower-level entity is SystemVerilog, you must ensure that the `enum` value is in the correct case. In SystemVerilog, two enumeration literals differ in more than just case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion and is more difficult to pass parameters from case-insensitive HDLs, such as VHDL.

Arrays have different support in different design languages. For details about the array parameter format, refer to the **Parameter** section in the Analysis & Synthesis Report of a design that contains array parameters or generics.

The following code shows examples of passing parameters from one design entry language to a subdesign written in another language. Example 16–18 shows a VHDL subdesign that you instantiate in a top-level Verilog HDL design in Example 16–19. Example 16–20 shows a Verilog HDL subdesign that you instantiate in a top-level VHDL design in Example 16–21.

Example 16–18. VHDL Parameterized Subdesign Entity

```
type fruit is (apple, orange, grape);
entity vhdl_sub is
generic (
name : string := "default",
width : integer := 8,
number_string : string := "123",
f : fruit := apple,
binary_vector : std_logic_vector(3 downto 0) := "0101",
signed_vector : signed (3 downto 0) := "1111");
```

Example 16–19. Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity from Example 16–18

```
vhdl_sub inst (...);
defparam inst.name = "lower";
defparam inst.width = 3;
defparam inst.num_string = "321";
defparam inst.f = "grape"; // Must exactly match enum value
defparam inst.binary_vector = 4'b1010;
      defparam inst.signed_vector = 4'sb1010;
```

Example 16–20. Verilog HDL Parameterized Subdesign Module

```
module veri_sub (...)
parameter name = "default";
parameter width = 8;
parameter number_string = "123";
parameter binary_vector = 4'b0101;
parameter signed_vector = 4'sb1111;
```

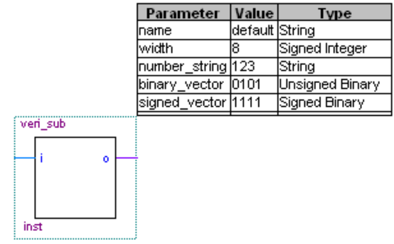
Example 16–21. VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 16–20

```
inst:veri_sub
generic map (
name => "lower",
width => 3,
number_string => "321"
binary_vector = "1010"
signed_vector = "1010")
```

To use an HDL subdesign such as the one shown in Example 16–20 in a top-level **.bdf** design, you must generate a symbol for the HDL file, as shown in Figure 16–2. Open the HDL file in the Quartus II software, and then, on the File menu, point to **Create/Update,** and then click **Create Symbol Files for Current File**.

To specify parameters on a **.bdf** instance, double-click the parameter value box for the instance symbol, or right-click the symbol and click **Properties**, and then click the **Parameters** tab. Right-click the symbol and click **Update Design File from Selected Block** to pass the updated parameter to the HDL file.

Figure 16–2. BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 16–20



Incremental Compilation

Incremental compilation manages a design hierarchy for incremental design by allowing you to divide your design into multiple partitions. Incremental compilation ensures that the Quartus II software resynthesizes only the updated partitions of your design during compilation, to reduce the compilation time and the runtime memory usage. The feature maintains node names during synthesis for all registered and combinational nodes in unchanged partitions. You can perform incremental synthesis by setting the netlist type for all design partitions to **Post-Synthesis**.

You can also preserve the placement and routing information for unchanged partitions. This feature allows you to preserve performance of unchanged blocks in your design and reduces the time required for placement and routing, which significantly reduces your design compilation time.

For more information about incremental compilation, refer to *About Incremental Compilation* in Quartus II Help.

For more information about incremental compilation, refer to *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* and *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapters in volume 1 of the *Quartus II Handbook*.

Partitions for Preserving Hierarchical Boundaries

A design partition represents a portion of your design that you want to synthesize and fit incrementally.

If you want to preserve the **Optimization Technique** and **Restructure Multiplexers** logic options in any entity, you must create new partitions for the entity instead of using the **Preserve Hierarchical Boundary** logic option. If you have settings applied to specific existing design hierarchies, particularly those created in the Quartus II software versions before 9.0, you must create a design partition for the design hierarchy so that synthesis can optimize the design instance independently and preserve the hierarchical boundaries.

The **Preserve Hierarchical Boundary** logic option is available only in Quartus II software versions 8.1 and earlier. Altera recommends using design partitions if you want to preserve hierarchical boundaries through the synthesis and fitting process, because incremental compilation maintains the hierarchical boundaries of design partitions.

Parallel Synthesis

The **Parallel Synthesis** logic option reduces compilation time for synthesis. The option enables the Quartus II software to use multiple processors to synthesize multiple partitions in parallel.

This option is available when you perform the following tasks:

- Specifying the maximum number of processors allowed under **Parallel Compilation** options in the **Compilation Process Settings** page of the **Settings** dialog box.

- Enabling the incremental compilation feature.
- Using two or more partitions in your design.
- Turning on the **Parallel Synthesis** option.

By default, the Quartus II software enables the **Parallel Synthesis** option. To disable parallel synthesis, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, click **Analysis & Synthesis Settings,** and then click **More Settings** to select **Parallel Synthesis**.

You can also set the **Parallel Synthesis** option with the following Tcl command, as shown in Example 16–22:

Example 16–22. Setting the Parallel Synthesis Option with Tcl Command

```
set_global_assignment -name parallel_synthesis off
```

If you use the command line, you can differentiate among the interleaved messages by turning on the **Show partition that generated the message** option in the Messages page. This option shows the partition ID in parenthesis for each message.

You can view all the interleaved messages from different partitions in the Messages window. The **Partition** column in the Messages window displays the partition ID of the partition referred to in the message. After compilation, you can sort the messages by partition.

For more information about displaying the **Partition** column, refer to *About the Messages Window* in Quartus II Help.

Quartus II Exported Partition File as Source

You can use a **.qxp** as a source file in incremental compilation. The **.qxp** contains the precompiled design netlist exported as a partition from another Quartus II project, and fully defines the entity. Project team members or intellectual property (IP) providers can use a **.qxp** to send their design to the project lead, instead of sending the original HDL source code. The **.qxp** preserves the compilation results and instance-specific assignments. Not all global assignments can function in a different Quartus II project. You can override the assignments for the entity in the **.qxp** by applying assignments in the top-level design.

For more information about **.qxp**, refer to *Quartus II Exported Partition File (.qxp)* in Quartus II Help.

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

Quartus II Synthesis Options

The Quartus II software offers several options to help you control the synthesis process and achieve optimal results for your design. "Setting Synthesis Options" on page 16–25 describes the **Analysis & Synthesis Settings** page of the **Settings** dialog box, in which you can set the most common global settings and options, and defines the following types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives.

When you apply a Quartus II Synthesis option globally or to an entity, the option affects all lower-level entities in the hierarchy path, including entities instantiated with Altera and third-party IP.

The following subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples on how to use each option:

- Major Optimization Settings:
  - "Optimization Technique" on page 16–28
  - "Auto Gated Clock Conversion" on page 16–29
  - "PowerPlay Power Optimization" on page 16–31
  - "Restructure Multiplexers" on page 16–34
- Settings Related to Timing Constraints:
  - "Timing-Driven Synthesis" on page 16–30
  - "Optimization Technique" on page 16–28
  - "Auto Gated Clock Conversion" on page 16–29
  - "SDC Constraint Protection" on page 16–31
- State Machine Settings and Enumerated Types:
  - "State Machine Processing" on page 16–34
  - "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 16–36
  - "Manually Specifying Enumerated Types Using the enum_encoding Attribute" on page 16–37
  - "Safe State Machine" on page 16–38
- Register Power-Up Settings:
  - "Power-Up Level" on page 16–40
  - "Power-Up Don't Care" on page 16–41

- Controlling, Preserving, Removing, and Duplicating Logic and Registers:

  - "Limiting Resource Usage in Partitions" on page 16–32
  - "Remove Duplicate Registers" on page 16–41
  - "Preserve Registers" on page 16–42
  - "Disable Register Merging/Don't Merge Register" on page 16–43
  - "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 16–43
  - "Keep Combinational Node/Implement as Output of Logic Cell" on page 16–44
  - "Disabling Synthesis Netlist Optimizations with dont_retime Attribute" on page 16–45
  - "Disabling Synthesis Netlist Optimizations with dont_replicate Attribute" on page 16–46
  - "Maximum Fan-Out" on page 16–47
  - "Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable" on page 16–48
  - "Auto Gated Clock Conversion" on page 16–29
  - "Partitions for Preserving Hierarchical Boundaries" on page 16–21

- Megafunction Inference Options:

  - "Inferring Multiplier, DSP, and Memory Functions from HDL Code" on page 16–49
  - "RAM Style and ROM Style—for Inferred Memory" on page 16–53
  - "Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute" on page 16–56
  - "RAM Initialization File—for Inferred Memory" on page 16–59
  - "Multiplier Style—for Inferred Multipliers" on page 16–60

- Controlling Synthesis with Other Synthesis Directives:

  - "Full Case Attribute" on page 16–62
  - "Parallel Case" on page 16–63
  - "Translate Off and On / Synthesis Off and On" on page 16–64
  - "Ignore translate_off and synthesis_off Directives" on page 16–65
  - "Read Comments as HDL" on page 16–66

- Specifying I/O-Related Assignments:

  - "Use I/O Flipflops" on page 16–67
  - "Specifying Pin Locations with chip_pin" on page 16–68

- Setting Quartus II Logic Options in Your HDL Source Code:

  - "Using altera_attribute to Set Quartus II Logic Options" on page 16–70

- Other Settings:
  - "Synthesis Effort" on page 16–34
  - "Synthesis Seed" on page 16–34

Setting Synthesis Options

You can set synthesis options in the **Settings** dialog box, or with logic options in the Quartus II software, or you can use synthesis attributes and directives in your HDL source code.

The **Analysis & Synthesis Settings** page of the **Settings** dialog box allows you to set global synthesis options that apply to the entire project. You can also use a corresponding Tcl command.

You can set some of the advanced synthesis settings in the **Physical Synthesis Optimizations** page under **Compilation Process Settings**.

For more information about Physical Synthesis options, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Quartus II Logic Options

The Quartus II logic options control many aspects of the synthesis and placement and routing process. To set logic options in the Quartus II software, on the Assignments menu, click **Assignment Editor**. You can also use a corresponding Tcl command to set global assignments. The Quartus II logic options enable you to set instance or node-specific assignments without editing the source HDL code.

For more information about using the Assignment Editor, refer to the *About the Assignment Editor* in Quartus II Help.

Synthesis Attributes

The Quartus II software supports synthesis attributes for Verilog HDL and VHDL, also commonly called pragmas. These attributes are not standard Verilog HDL or VHDL commands. Synthesis tools use attributes to control the synthesis process. The Quartus II software applies the attributes in the HDL source code, and attributes always apply to a specific design element. Some synthesis attributes are also available as Quartus II logic options via the Quartus II software or scripting. Each attribute description in this chapter indicates a corresponding setting or a logic option that you can set in the Quartus II software. You can specify only some attributes with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the Assignment Editor or in the **.qsf**. Assignments or settings made with the Quartus II software, the **.qsf**, or the Tcl interface take precedence over assignments or settings made with synthesis attributes in your HDL code. The Quartus II software generates warning messages if the software finds invalid attributes, but does not generate an error or stop the compilation. This behavior is necessary because attributes are specific to various design tools, and attributes not recognized in the Quartus II software might be for a different EDA tool. The Quartus II software lists the attributes specified in your HDL code in the Source assignments table of the Analysis & Synthesis report.

The Verilog-2001, SystemVerilog, and VHDL language definitions provide specific syntax for specifying attributes, but in Verilog-1995, you must embed attribute assignments in comments. You can enter attributes in your code using the syntax in Example 16–23 through Example 16–29, in which *<attribute>*, *<attribute type>*, *<value>*, *<object>*, and *<object type>* are variables, and the entry in brackets is optional. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case sensitive; therefore, synthesis attributes in Verilog HDL files are also case sensitive.

Example 16–23. Specifying Synthesis Attributes in Verilog-1995

```
// synthesis <attribute> [ = <value> ]
or
/* synthesis <attribute> [ = <value> ] */
```

You must use Verilog-1995 comment-embedded attributes as a suffix to the declaration of an item and must appear before a semicolon, when a semicolon is necessary (refer to Example 16–23).

You cannot use the open one-line comment in Verilog HDL when a semicolon is necessary after the line, because it is not clear to which HDL element that the attribute applies. For example, you cannot make an attribute assignment such as `reg r; // synthesis <attribute>` because the Quartus II software could read the attribute as part of the next line.

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces, as shown in Example 16–24:

Example 16–24. Applying Multiple Attributes to the Same Instance in Verilog-1996

```
//synthesis <attribute1> [ = <value> ] <attribute2> [ = <value> ]
```

For example, to set the `maxfan` attribute to `16` (for details, refer to “Maximum Fan-Out” on page 16–47) and set the `preserve` attribute (for details, refer to “Preserve Registers” on page 16–42) on a register called `my_reg`, use the following syntax as shown in Example 16–25:

Example 16–25. Setting maxfan and preserve Attribute on a Register

```
reg my_reg /* synthesis maxfan = 16 preserve */;
```

In addition to the `synthesis` keyword shown above, the Quartus II software supports the `pragma`, `synopsys`, and `exemplar` keywords for compatibility with other synthesis tools. The software also supports the `altera` keyword, which allows you to add synthesis attributes that the Quartus II Integrated Synthesis feature recognizes and not by other tools that recognize the same synthesis attribute.

Because formal verification tools do not recognize the `exemplar`, `pragma`, and `altera` keywords, avoid using these attribute keywords when using formal verification.

You must use Verilog-2001 attributes as a prefix to a declaration, module item, statement, or port connection, and as a suffix to an operator or a Verilog HDL function name in an expression (refer to Example 16–26).

Example 16–26. Specifying Synthesis Attributes in Verilog-2001 and SystemVerilog

```
(* <attribute> [ = <value> ] *)
```

Formal verification does not support the Verilog-2001 attribute syntax because the tools do not recognize the syntax.

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas, as shown in Example 16–27:

Example 16–27. Applying Multiple Attributes

```
(* <attribute1> [ = <value1>], <attribute2> [ = <value2> ] *)
```

For example, to set the `maxfan` attribute to `16` (refer to "Maximum Fan-Out" on page 16–47 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 16–42 for details) on a register called `my_reg`, use the following syntax as shown in Example 16–28:

Example 16–28. Setting Attribute

```
(* maxfan = 16, preserve *) reg my_reg;
```

VHDL attributes, as shown in Example 16–29, declare and apply the attribute type to the object you specify.

Example 16–29. Synthesis Attributes in VHDL

```
attribute <attribute> : <attribute type> ;
attribute <attribute> of <object> : <object type> is <value>;
```

The Quartus II software defines and applies each attribute separately to a given node. For VHDL designs, the software declares all supported synthesis attributes in the `altera_syn_attributes` package in the Altera library. You can call this library from your VHDL code to declare the synthesis attributes, as shown in Example 16–30:

Example 16–30.

```
LIBRARY altera;
USE altera.altera_syn_attributes.all;
```

Synthesis Directives

The Quartus II software supports synthesis directives, also commonly called compiler directives or pragmas. You can include synthesis directives in Verilog HDL or VHDL code as comments. These directives are not standard Verilog HDL or VHDL commands. Synthesis tools use directives to control the synthesis process. Directives do not apply to a specific design node, but change the behavior of the synthesis tool from the point in which they occur in the HDL source code. Other tools, such as simulators, ignore these directives and treat them as comments.

You can enter synthesis directives in your code using the syntax in Example 16–31, Example 16–32, and Example 16–33, in which <*directive*> and <*value*> are variables, and the entry in brackets are optional. For synthesis directives, no equal sign before the value is necessary; this is different than the Verilog syntax for synthesis attributes. The examples in this chapter demonstrate each syntax form.

Verilog HDL is case sensitive; therefore, all synthesis directives are also case sensitive.

Example 16–31. Specifying Synthesis Directives with Verilog HDL

```
// synthesis <directive> [ <value> ]
or
/* synthesis <directive> [ <value> ] */
```

Example 16–32. Specifying Synthesis Directives with VHDL

```
-- synthesis <directive> [ <value> ]
```

Example 16–33. Specifying Synthesis Directives with VHDL-2008

```
/* synthesis <directive> [<value>] */
```

In addition to the `synthesis` keyword shown above, the software supports the `pragma`, `synopsys`, and `exemplar` keywords in Verilog HDL and VHDL for compatibility with other synthesis tools. The Quartus II software also supports the keyword `altera`, which allows you to add synthesis directives that only Quartus II Integrated Synthesis feature recognizes, and not by other tools that recognize the same synthesis directives.

Because formal verification tools ignore the `exemplar`, `pragma`, and `altera` keywords, Altera recommends that you avoid using these directive keywords when you use formal verification to prevent mismatches with the Quartus II results.

Optimization Technique

The **Optimization Technique** logic option specifies the goal for logic optimization during compilation; that is, whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two.

For more information about the **Optimization Technique** logic option, refer to *Optimization Technique logic option* in Quartus II Help.

Auto Gated Clock Conversion

Clock gating is a common optimization technique in ASIC designs to minimize power consumption. You can use the **Auto Gated Clock Conversion** logic option to optimize your prototype ASIC designs by converting gated clocks into clock enables when you use FPGAs in your ASIC prototyping. The automatic conversion of gated clocks to clock enables is more efficient than manually modifying source code. The **Auto Gated Clock Conversion** logic option automatically converts qualified gated clocks (base clocks as defined in the Synopsys Design Constraints [SDC]) to clock enables. To use **Auto Gated Clock Conversion**, you must select the option from the **More Analysis & Synthesis Settings** dialog box, in the **Analysis & Synthesis Settings** page.

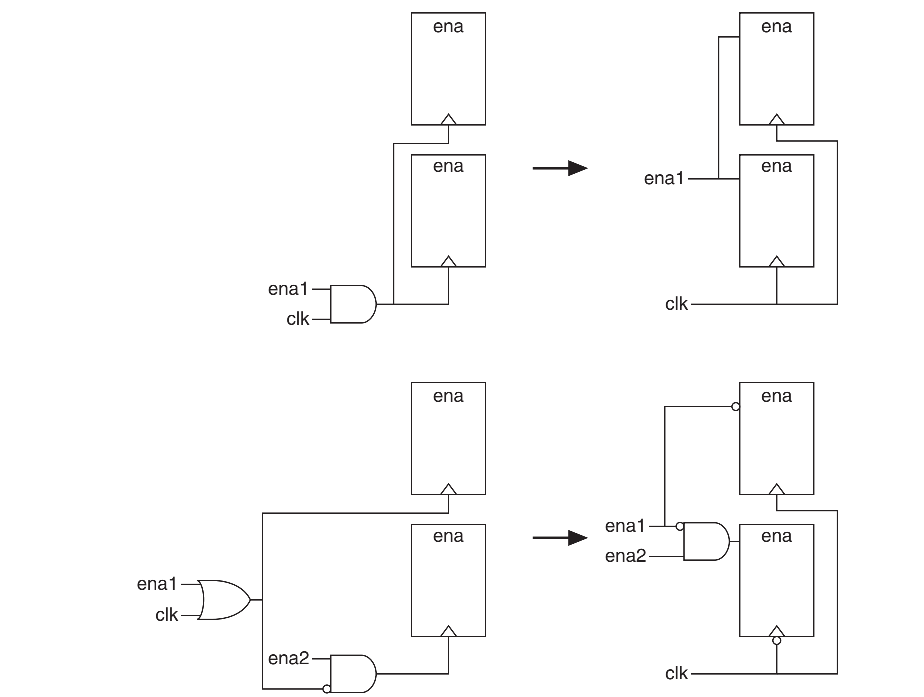
The gated clock conversion occurs when all these conditions are met:

- Only one base clock drives a gated-clock
- For one set of gating input values, the value output of the gated clock remains constant and does not change as the base clock changes
- For one value of the base clock, changes in the gating inputs do not change the value output for the gated clock

The option supports combinational gates in clock gating network.

Figure 16–3 shows example of gated clock conversions.

Figure 16–3. Example Gated Clock Conversion



This option does not support registers in RAM, DSP blocks, or I/O related WYSIWYG primitives. Because the gated-clock conversion cannot trace the base clock from the gated clock, the gated clock conversion does not support multiple design partitions from incremental compilation in which the gated clock and base clock are not in the same hierarchical partition. A gated clock tree, instead of every gated clock, is the basis of each conversion. Therefore, if you cannot convert a gated clock from a root gated clock of a multiple cascaded gated clock, the conversion of the entire gated clock tree fails.

The **Info** tab in the Messages window lists all the converted gated clocks. You can view a list of converted and nonconverted gated clocks from the Compilation Report under the **Optimization Results** of the Analysis & Synthesis Report. The **Gated Clock Conversion Details** table lists the reasons for nonconverted gated clocks.

For more information about **Auto Gated Clock Conversion** logic option and a list of supported devices, refer to *Auto Gated Clock Conversion logic option* in Quartus II Help.

Timing-Driven Synthesis

The **Timing-Driven Synthesis** logic option specifies whether Analysis & Synthesis should use the SDC timing constraints of your design to better optimize the circuit. When you turn on this option, Analysis & Synthesis runs timing analysis to obtain timing information about the netlist, and then considers the SDC timing constraints to focus on critical portions of your design when optimizing for performance, while optimizing noncritical portions for area. When you turn on this option, Analysis & Synthesis also protects SDC constraints by not merging duplicate registers that have incompatible timing constraints. For more information, refer to "SDC Constraint Protection" on page 16–31.

When you turn on the **Timing-Driven Synthesis** logic option, Analysis & Synthesis increases performance by improving logic depth on critical portions of your design, and improving area on noncritical portions of your design. The increased performance affects the amount of area used, specifically adaptive look-up tables (ALUTs) and registers in your design. Depending on how much of your design is timing critical, overall area can increase or decrease when you turn on the **Timing-Driven Synthesis** logic option. Runtime and peak memory use increases slightly if you turn on the **Timing-Driven Synthesis** logic option.

When you turn on the **Timing-Driven Synthesis** logic option, the **Optimization Technique** logic option has the following effect. With **Optimization Technique Speed**, Timing-Driven Synthesis optimizes timing-critical portions of your design for performance at the cost of increasing area (logic and register utilization). With an **Optimization Technique** of **Balanced**, Timing-Driven Synthesis also optimizes the timing-critical portions of your design for performance, but the option allows only limited area increase. With **Optimization Technique Area**, Timing-Driven Synthesis optimizes your design only for area. **Timing-Driven Synthesis** prevents registers with incompatible timing constraints from merging for any **Optimization Technique** setting. If your design contains multiple partitions, you can select **Timing-Driven Synthesis** unique options for each partition. If you use a **.qxp** as a source file, or if your design uses partitions developed in separate Quartus II projects, the software cannot properly compute timing of paths that cross the partition boundaries.

Even with the **Optimization Technique** logic option set to **Speed**, the **Timing-Driven Synthesis** option still considers the resource usage in your design when increasing area to improve timing. For example, the **Timing-Driven Synthesis** option checks if a device has enough registers before deciding to implement the shift registers in logic cells instead of RAM for better timing performance.

When using incremental compilation, Integrated Synthesis allows each partition to use up all the registers in a device. You can use the **Maximum Number of LABs** settings to specify the number of LABs that every partition can use. If your design has only one partition, you can also use the **Maximum Number of LABs** settings to limit the number of resources that your design can use. This limitation is useful when you add more logic to your design.

To turn on or turn off the **Timing-Driven Synthesis** logic option, follow these steps:

1. On the Assignment menu, click **Settings**.
2. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, turn on or turn off **Timing-Driven Synthesis**.

Altera recommends that you select a specific device for timing-driven synthesis to have the most accurate timing information. When you select auto device, timing-driven synthesis uses the smallest device for the selected family to obtain timing information.

For more information about **Timing-Driven Synthesis** logic option and a list of supported devices, refer to *Timing-Driven Synthesis logic option* in Quartus II Help.

SDC Constraint Protection

The **SDC Constraint Protection** option specifies whether Analysis & Synthesis should protect registers from merging when they have incompatible timing constraints. For example, when you turn on this option, the software does not merge two registers that are duplicates of each other but have different multicycle constraints on them. When you turn on the **Timing-Driven Synthesis** option, the software detects registers with incompatible constraints, and you do not need to turn on **SDC Constraint Protection**. To use the **SDC constraint protection** option, you must turn on the option in the **More Analysis & Synthesis Settings** dialog box in the **Analysis & Synthesis Settings** page.

PowerPlay Power Optimization

The **PowerPlay Power Optimization** logic option controls the power-driven compilation setting of Analysis & Synthesis and determines how aggressively Analysis & Synthesis optimizes your design for power.

For more information about the available settings for the **PowerPlay power optimization** logic option and a list of supported devices, refer to *PowerPlay Power Optimization logic option* in Quartus II Help.

For more information about optimizing your design for power utilization, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*. For information about analyzing your power results, refer to the *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*.

Limiting Resource Usage in Partitions

Resource balancing is important when performing Analysis & Synthesis. During resource balancing, Quartus II Integrated Synthesis considers the amount of used and available DSP and RAM blocks in the device, and tries to balance these resources to prevent no-fit errors.

For DSP blocks, resource balancing converts the remaining DSP blocks to equivalent logic if there are more DSP blocks in your design that the software can place in the device. For RAM blocks, resource balancing converts RAM blocks to different types of RAM blocks if there are not enough blocks of a certain type available in the device; however, Quartus II Integrated Synthesis does not convert RAM blocks to logic.

The RAM balancing feature does not support Stratix V devices because Stratix V has only M20K memory blocks.

By default, Quartus II Integrated Synthesis considers the information in the targeted device to identify the number of available DSP or RAM blocks. However, in incremental compilation, each partition considers the information in the device independently and consequently assumes that the partition has all the DSP and RAM blocks in the device available for use, resulting in over allocation of DSP or RAM blocks in your design, which means that the total number of DSP or RAM blocks used by all the partitions is greater than the number of DSP or RAM blocks available in the device, leading to a no-fit error during the fitting process.

The following sections describe the methods to prevent a no-fit error during the fitting process:

- “Creating LogicLock Regions” on page 16–32
- “Using Assignments to Limit the Number of RAM and DSP Blocks” on page 16–33

Creating LogicLock Regions

The floorplan-aware synthesis feature allows you to use LogicLock regions to define resource allocation for DSP blocks and RAM blocks. For example, if you assign a certain partition to a certain LogicLock region, resource balancing takes into account that all the DSP and RAM blocks in that partition need to fit in this LogicLock region. Resource balancing then balances the DSP and RAM blocks accordingly.

Because floorplan-aware balancing step considers only one partition at a time, it does not know that nodes from another partition may be using the same resources. When using this feature, Altera recommends that you do not manually assign nodes from different partitions to the same LogicLock region.

If you do not want the software to consider the LogicLock floorplan constraints when performing DSP and RAM balancing, you can turn off the floorplan-aware synthesis feature. You can turn off the **Use LogicLock Constraints During Resource Balancing** option in the **More Analysis & Synthesis Settings** dialog box in the **Analysis & Synthesis Settings** page.

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

Using Assignments to Limit the Number of RAM and DSP Blocks

For DSP and RAM block balancing, you can use assignments to limit the maximum number of blocks that the balancer allows. You can set these assignments globally or on individual partitions. For DSP block balancing, the **Maximum DSP Block Usage** logic option allows you to specify the maximum number of DSP blocks that the DSP block balancer assumes are available for the current partition. For RAM blocks, the floorplan-aware logic option allows you to specify maximum resources for different RAM types, such as **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks**, **Maximum Number of M512 Memory Blocks**, **Maximum Number of M-RAM/M144K Memory Blocks**, or **Maximum Number of LABs**.

The partition-specific assignment overrides the global assignment, if any. However, each partition that does not have a partition-specific assignment uses the value set by the global assignment, or the value derived from the device size if no global assignment exists. This action can also lead to over allocation. Therefore, Altera recommends that you always set the assignment on each partition individually.

To select the **Maximum Number** <*block type*> **Memory Blocks** option or the **Maximum DSP Block Usage** option globally, follow these steps:

1. On the Assignment menu, click **Settings**.
2. Under **Category**, click **Analysis & Synthesis Settings**.
3. In the **Analysis & Synthesis Settings** dialog box, click **More Settings**.
4. In the **Name** list, select the required option and set the necessary value.

You can use the Assignment Editor to set this assignment on a partition by selecting the assignment, and setting it on the root entity of a partition. You can set any positive integer as the value of this assignment. If you set this assignment on a name other than a partition root, Analysis & Synthesis gives an error.

For more information about the logic options, including a list of supported device families, refer to *Maximum DSP Block Usage logic option*, *Maximum Number of M4K/M9K/M20K/M10K Memory Blocks logic option*, *Maximum Number of M512 Memory Blocks logic option*, *Maximum Number of M-RAM/144K Memory Blocks logic option*, and *Maximum Number of LABs logic option* in Quartus II Help.

Restructure Multiplexers

The **Restructure Multiplexers** logic option restructures multiplexers to create more efficient use of area, allowing you to implement multiplexers with a reduced number of LEs or ALMs.

When multiplexers from one part of your design feed multiplexers in another part of your design, trees of multiplexers form. Multiplexers may arise in different parts of your design through Verilog HDL or VHDL constructs such as the "`if`," "`case`," or "`?:`" statements. Multiplexer buses occur most often as a result of multiplexing together arrays in Verilog HDL, or `STD_LOGIC_VECTOR` signals in VHDL. The **Restructure Multiplexers** logic option identifies buses of multiplexer trees that have a similar structure. This logic option optimizes the structure of each multiplexer bus for the target device to reduce the overall amount of logic in your design.

Results of the multiplexer optimizations are design dependent, but area reductions as high as 20% are possible. The option can negatively affect your design's f_{MAX}.

For more information about optimizing for multiplexers, refer to the "Multiplexers" section of the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For more information about the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers, refer to *Analysis & Synthesis Optimization Results Reports* in Quartus II Help.

For more information about the **Restructure Multiplexers** logic option, including the settings and a list of supported device families, refer to *Restructure Multiplexers logic option* in Quartus II Help.

Synthesis Effort

The **Synthesis Effort** logic option specifies the overall synthesis effort level in the Quartus II software.

For more information about **Synthesis Effort** logic option, including a list of supported device families, refer to *Synthesis Effort logic option* in Quartus II Help.

Synthesis Seed

The **Synthesis Seed** option specifies the seed that Synthesis uses to randomly run synthesis in a slightly different way. You can use this seed when your design is close to meeting requirements, to get a slightly different result. The seeds that produce the best result for a design might change if your design changes.

To set the **Synthesis Seed** option from the Quartus II software, on the **Analysis & Synthesis Settings** page, click **More Settings**. The default value is **1**. You can specify a positive integer value.

State Machine Processing

The **State Machine Processing** logic option specifies the processing style to synthesize a state machine.

The default state machine encoding, **Auto**, uses one-hot encoding for FPGA devices and minimal-bits encoding for CPLDs. These settings achieve the best results on average, but another encoding style might be more appropriate for your design, so this option allows you to control the state machine encoding.

For guidelines on how to correctly infer and encode your state machine, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For one-hot encoding, the Quartus II software does not guarantee that each state has one bit set to one and all other bits set to zero. Quartus II Integrated Synthesis creates one-hot register encoding with standard one-hot encoding and then inverts the first bit. This results in an initial state with all zero values, and the remaining states have two 1 values. Quartus II Integrated Synthesis encodes the initial state with all zeros for the state machine power-up because all device registers power up to a low value. This encoding has the same properties as true one-hot encoding: the software recognizes each state by the value of one bit. For example, in a one-hot-encoded state machine with five states, including an initial or reset state, the software uses the register encoding shown in Example 16–34:

Example 16–34. Register Encoding

```
State 0     0 0 0 0 0
State 1     0 0 0 1 1
State 2     0 0 1 0 1
State 3     0 1 0 0 1
State 4     1 0 0 0 1
```

If you set the **State Machine Processing** logic option to **User-Encoded** in a Verilog HDL design, the software starts with the original design values for the state constants. For example, a Verilog HDL design can contain a declaration such as shown in Example 16–35:

Example 16–35.

```
parameter S0 = 4'b1010, S1 = 4'b0101, ...
```

If the software infers the states `S0, S1,...` the software uses the encoding `4'b1010, 4'b0101,...`. If necessary, the software inverts bits in a user-encoded state machine to ensure that all bits of the reset state of the state machine are zero.

You can view the state machine encoding from the Compilation Report under the State Machines of the Analysis & Synthesis Report. The State Machine Viewer displays only a graphical representation of the state machines as interpreted from your design.

For more information about the State Machine Viewer, refer to the State Machine Viewer section of the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

To assign your own state encoding with the **User-Encoded** setting of the **State Machine Processing** option in a VHDL design, you must apply specific binary encoding to the elements of an enumerated type because enumeration literals have no numeric values in VHDL. Use the `syn_encoding` synthesis attribute to apply your encoding values. For more information, refer to "Manually Specifying State Assignments Using the syn_encoding Attribute".

For information about the **State Machine Processing** logic option, including the settings and supported devices, refer to *State Machine Processing logic option* in Quartus II Help.

Manually Specifying State Assignments Using the syn_encoding Attribute

The Quartus II software infers state machines from enumerated types and automatically assigns state encoding based on "State Machine Processing" on page 16–34. With this logic option, you can choose the value **User-Encoded** to use the encoding from your HDL code. However, in standard VHDL code, you cannot specify user encoding in the state machine description because enumeration literals have no numeric values in VHDL.

To assign your own state encoding for the **User-Encoded State Machine Processing** setting, use the `syn_encoding` synthesis attribute to apply specific binary encodings to the elements of an enumerated type or to specify an encoding style. The Quartus II software can implement Enumeration Types with different encoding styles, as shown in Table 16–3.

Table 16–3. syn_encoding Attribute Values

Attribute Value	Enumeration Types
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If the number of literals is less than five, use the `"sequential"` encoding. If the number of literals is more than five, but fewer than 50, use a `"one-hot"` encoding. Otherwise, use a `"gray"` encoding.
`"sequential"`	Use a binary encoding in which the first enumeration literal in the Enumeration Type has encoding 0 and the second 1.
`"gray"`	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
`"johnson"`	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2^N states, but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring N bits, in which N is the number of enumeration literals in the Enumeration Type.
`"compact"`	Use an encoding with the fewest bits.
`"user"`	Encode each state using its value in the Verilog source. By changing the values of your state constants, you can change the encoding of your state machine.

The `syn_encoding` attribute must follow the enumeration type definition, but precede its use.

Manually Specifying Enumerated Types Using the enum_encoding Attribute

By default, the Quartus II software one-hot encodes all enumerated types you defined. With the `enum_encoding` attribute, you can specify the logic encoding for an enumerated type and override the default `one-hot` encoding to improve the logic efficiency.

If an enumerated type represents the states of a state machine, using the `enum_encoding` attribute to specify a manual state encoding prevents the Compiler from recognizing state machines based on the enumerated type. Instead, the Compiler processes these state machines as regular logic with the encoding specified by the attribute, and the Report window for your project does not list these states machines as state machines. If you want to control the encoding for a recognized state machine, use the **State Machine Processing** logic option and the `syn_encoding` synthesis attribute.

To use the `enum_encoding` attribute in a VHDL design file, associate the attribute with the enumeration type whose encoding you want to control. The `enum_encoding` attribute must follow the enumeration type definition, but precede its use. In addition, the attribute value should be a string literal that specifies either an arbitrary user encoding or an encoding style of `"default"`, `"sequential"`, `"gray"`, `"johnson"`, or `"one-hot"`.

An arbitrary user encoding consists of a space-delimited list of encodings. The list must contain as many encodings as the number of enumeration literals in your enumeration type. In addition, the encodings should have the same length, and each encoding must consist solely of values from the `std_ulogic` type declared by the `std_logic_1164` package in the IEEE library. In Example 16–36, the `enum_encoding` attribute specifies an arbitrary user encoding for the enumeration type `fruit`.

Example 16–36. Specifying an Arbitrary User Encoding for Enumerated Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "11 01 10 00";
```

Example 16–37 shows the encoded enumeration literals:

Example 16–37. Encoded Enumeration Literals

```
apple   = "11"
orange  = "01"
pear    = "10"
mango   = "00"
```

Altera recommends that you specify an encoding style, rather than a manual user encoding, especially when the enumeration type has a large number of enumeration literals. The Quartus II software can implement Enumeration Types with the different encoding styles, as shown in Table 16–4.

Table 16–4. enum_encoding Attribute Values

Attribute Value	Enumeration Types
"default"	Use an encoding based on the number of enumeration literals in the enumeration type. If the number of literals are fewer than five, use the "sequential" encoding. If the number of literals are more than five, but fewer than 50 literals, use a "one-hot" encoding. Otherwise, use a "gray" encoding.
"sequential"	Use a binary encoding in which the first enumeration literal in the enumeration type has encoding 0 and the second 1.
"gray"	Use an encoding in which the encodings for adjacent enumeration literals differ by exactly one bit. An N-bit gray code can represent 2^N values.
"johnson"	Use an encoding similar to a gray code. An N-bit Johnson code can represent at most 2^N states, but requires less logic than a gray encoding.
"one-hot"	The default encoding style requiring *N* bits, in which *N* is the number of enumeration literals in the enumeration type.

In Example 16–36, the `enum_encoding` attribute manually specified a gray encoding for the enumeration type `fruit`. You can concisely write this example by specifying the `"gray"` encoding style instead of a manual encoding, as shown in Example 16–38.

Example 16–38. Specifying the “gray” Encoding Style or Enumeration Type

```
type fruit is (apple, orange, pear, mango);
attribute enum_encoding : string;
attribute enum_encoding of fruit : type is "gray";
```

Safe State Machine

The **Safe State Machine** logic option and corresponding `syn_encoding` attribute value `safe` specify that the software must insert extra logic to detect an illegal state, and force the transition of the state machine to the reset state.

A finite state machine can enter an illegal state—meaning the state registers contain a value that does not correspond to any defined state. By default, the behavior of the state machine that enters an illegal state is undefined. However, you can set the `syn_encoding` attribute to `safe` or use the **Safe State Machine** logic option if you want the state machine to recover deterministically from an illegal state. The software inserts extra logic to detect an illegal state, and forces the transition of the state machine to the reset state. You can use this logic option when the state machine enters an illegal state. The most common cause of an illegal state is a state machine that has control inputs that come from another clock domain, such as the control logic for a clock-crossing FIFO, because the state machine must have inputs from another clock domain. This option protects only state machines (and not other registers) by forcing them into the reset state. You can use this option if your design has asynchronous inputs. However, Altera recommends using a synchronization register chain instead of relying on the safe state machine option.

The `safe` state machine value does not use any user-defined default logic from your HDL code that corresponds to unreachable states. Verilog HDL and VHDL enable you to specify a behavior for all states in the state machine explicitly, including unreachable states. However, synthesis tools detect if state machine logic is unreachable and minimize or remove the logic. Synthesis tools also remove any flag signals or logic that indicate such an illegal state. If the software implements the state machine as safe, the recovery logic added by Quartus II Integrated Synthesis forces its transition from an illegal state to the reset state.

You can set the **Safe State Machine** logic option globally, or on individual state machines. To set this logic option, on the **Analysis & Synthesis Settings** page, select **More Settings**. In the **Existing option settings** list, select **Safe State Machine**, and turn on this option in the **Setting** list.

You can set the `syn_encoding safe` attribute on a state machine in HDL, as shown in Example 16–39 through Example 16–41.

Example 16–39. Verilog HDL Code: a Safe State Machine Attribute

```
reg [2:0] my_fsm /* synthesis syn_encoding = "safe" */;
```

Example 16–40. Verilog-2001 and SystemVerilog Code: a Safe State Machine Attribute

```
(* syn_encoding = "safe" *) reg [2:0] my_fsm;
```

Example 16–41. VHDL Code: a Safe State Machine Attribute

```
ATTRIBUTE syn_encoding OF my_fsm : TYPE IS "safe";
```

If you specify an encoding style (refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 16–36), separate the encoding style value in the quotation marks with the `safe` value with a comma, as follows: `"safe, one-hot"` or `"safe, gray"`.

Safe state machine implementation can result in a noticeable area increase for your design. Therefore, Altera recommends that you set this option only on the critical state machines in your design in which the safe mode is necessary, such as a state machine that uses inputs from asynchronous clock domains. You may not need to use this option if you correctly synchronize inputs coming from other clock domains.

If you create the `safe` state machine assignment on an instance that the software fails to recognize as a state machine, or an entity that contains a state machine, the software takes no action. You must restructure the code, so that the software recognizes and infers the instance as a state machine.

For more information about the **Safe State Machine** logic option, refer to *Safe State Machine logic option* in Quartus II Help.

For guidelines to ensure that the software correctly infers your state machine, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Level

This logic option causes a register (flipflop) to power up with the specified logic level, either high (1) or low (0). The registers in the core hardware power up to 0 in all Altera devices. For the register to power up with a logic level high, the Compiler performs an optimization referred to as NOT-gate push back on the register. NOT-gate push back adds an inverter to the input and the output of the register, so that the reset and power-up conditions appear to be high and the device operates as expected. The register itself still powers up to 0, but the register output inverts so the signal arriving at all destinations is 1.

The **Power-Up Level** option supports wildcard characters, and you can apply this option to any register, registered logic cell WYSIWYG primitive, or to a design entity containing registers, if you want to set the power level for all registers in your design entity. If you assign this option to a registered logic cell WYSIWYG primitive, such as an atom primitive from a third-party synthesis tool, you must turn on the **Perform WYSIWYG Primitive Resynthesis** logic option for the option to take effect. You can also apply the option to a pin with the logic configurations described in the following list:

- If you turn on this option for an input pin, the option transfers to the register that the pin drives, if all these conditions are present:
  - No logic, other than inversion, between the pin and the register.
  - The input pin drives the data input of the register.
  - The input pin does not fan-out to any other logic.
- If you turn on this option for an output or bidirectional pin, the option transfers to the register that feeds the pin, if all these conditions are present:
  - No logic, other than inversion, between the register and the pin.
  - The register does not fan out to any other logic.

For more information about the **Power-Up Level** logic option, including information on the supported device families, refer to *Power-Up Level logic option* in Quartus II Help.

Inferred Power-Up Levels

Quartus II Integrated Synthesis reads default values for registered signals defined in Verilog HDL and VHDL code, and converts the default values into **Power-Up Level** settings. The software also synthesizes variables with assigned values in Verilog HDL initial blocks into power-up conditions. Synthesis of these default and initial constructs allows synthesized behavior of your design to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

The following register declarations all set a power-up level of V_{CC} or a logic value “1”, as shown in Example 16–42:

Example 16–42.

```
signal q : std_logic = '1';  -- power-up to VCC

reg q = 1'b1;  // power-up to VCC

reg q;
initial begin q = 1'b1; end  // power-up to VCC
```

For more information about NOT-gate push back, the power-up states for Altera devices, and how set and reset control signals affect the power-up level, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Power-Up Don’t Care

This logic option allows the Compiler to optimize registers in your design that do not have a defined power-up condition.

For example, your design might have a register with its D input tied to V_{CC}, and with no clear signal or other secondary signals. If you turn on this option, the Compiler can choose for the register to power up to V_{CC}. Therefore, the output of the register is always V_{CC}. The Compiler can remove the register and connect its output to V_{CC}. If you turn this option off or if you set a **Power-Up Level** assignment of **Low** for this register, the register transitions from GND to V_{CC} when your design starts up on the first clock signal. Thus, the register is at V_{CC} and you cannot remove the register. Similarly, if the register has a clear signal, the Compiler cannot remove the register because after asserting the clear signal, the register transitions again to GND and back to V_{CC}.

If the Compiler performs a **Power-Up Don’t Care** optimization that allows it to remove a register, it issues a message to indicate that it is doing so.

This project-wide option does not apply to registers that have the **Power-Up Level** logic option set to either **High** or **Low**.

For more information about **Power-Up Don’t Care** logic option and a list of supported devices, refer to *Power-Up Don’t Care logic option* in Quartus II Help.

Remove Duplicate Registers

The **Remove Duplicate Registers** logic option removes registers that are identical to other registers.

For more information about **Remove Duplicate Registers** logic option and the supported devices, refer to *Remove Duplicate Registers logic option* in Quartus II Help.

Preserve Registers

This attribute and logic option directs the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. Optimizations can eliminate redundant registers and registers with constant drivers; this option prevents the software from reducing a register to a constant or merging with a duplicate register. This option can preserve a register so you can observe the register during simulation or with the SignalTap® II Logic Analyzer. Additionally, this option can preserve registers if you create a preliminary version of your design in which you have not specified the secondary signals. You can also use the attribute to preserve a duplicate of an I/O register so that you can place one copy of the I/O register in an I/O cell and the second in the core.

This option cannot preserve registers that have no fan-out. To prevent the removal of registers with no fan-out, refer to “Noprune Synthesis Attribute/Preserve Fan-out Free Register Node” on page 16–43.

The **Preserve Registers** logic option prevents the software from inferring a register as a state machine.

You can set the **Preserve Registers** logic option in the Quartus II software, or you can set the `preserve` attribute in your HDL code, as shown in Example 16–43 through Example 16–45. In these examples, the Quartus II software preserves the `my_reg` register.

In addition to `preserve`, the Quartus II software supports the `syn_preserve` attribute name for compatibility with other synthesis tools.

Example 16–43. Verilog HDL Code: syn_preserve Attribute

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

Example 16–44. Verilog-2001 Code: syn_preserve Attribute

```
(* syn_preserve = 1 *) reg my_reg;
```

The `= 1` after the `preserve` in Example 16–43 and Example 16–44 is optional, because the assignment uses a default value of `1` when you specify the assignment.

Example 16–45. VHDL Code: preserve Attribute

```
signal my_reg : stdlogic;
attribute preserve : boolean;
attribute preserve of my_reg : signal is true;
```

For more information about the **Preserve Registers** logic option and the supported devices, refer to *Preserve Registers logic option* in Quartus II Help.

Disable Register Merging/Don't Merge Register

This logic option and attribute prevents the specified register from merging with other registers and prevents other registers from merging with the specified register. When applied to a design entity, it applies to all registers in the entity.

You can set the **Disable Register Merging** logic option in the Quartus II software, or you can set the dont_merge attribute in your HDL code, as shown in Example 16–46 through Example 16–48. In these examples, the logic option or the attribute prevents the my_reg register from merging.

Example 16–46. Verilog HDL Code: dont_merge Attribute

```
reg my_reg /* synthesis dont_merge */;
```

Example 16–47. Verilog-2001 and SystemVerilog Code: dont_merge Attribute

```
(* dont_merge *) reg my_reg;
```

Example 16–48. VHDL Code: dont_merge Attribute

```
signal my_reg : stdlogic;
attribute dont_merge : boolean;
attribute dont_merge of my_reg : signal is true;
```

For more information about the **Disable Register Merging** logic option and the supported devices, refer to *Disable Register Merging logic option* in Quartus II Help.

Noprune Synthesis Attribute/Preserve Fan-out Free Register Node

This synthesis attribute and corresponding logic option direct the Compiler to preserve a fan-out-free register through the entire compilation flow. This option is different from the **Preserve Registers** option, which prevents the Quartus II software from reducing a register to a constant or merging with a duplicate register. Standard synthesis optimizations remove nodes that do not directly or indirectly feed a top-level output pin. This option can retain a register so you can observe the register in the Simulator or the SignalTap II Logic Analyzer. Additionally, this option can retain registers if you create a preliminary version of your design in which you have not specified the fan-out logic of the register.

You can set the **Preserve Fan-out Free Register Node** logic option in the Quartus II software, or you can set the noprune attribute in your HDL code, as shown in Example 16–49 though Example 16–51. In these examples, the logic option or the attribute preserves the my_reg register.

You must use the noprune attribute instead of the logic option if the register has no immediate fan-out in its module or entity. If you do not use the synthesis attribute, the software removes (or "prunes") registers with no fan-out during Analysis & Elaboration before the logic synthesis stage applies any logic options. If the register has no fan-out in the full design, but has fan-out in its module or entity, you can use the logic option to retain the register through compilation.

The software supports the attribute name syn_noprune for compatibility with other synthesis tools.

Example 16–49. Verilog HDL Code: syn_noprune Attribute

```
reg my_reg /* synthesis syn_noprune */;
```

Example 16–50. Verilog-2001 and SystemVerilog Code: noprune Attribute

```
(* noprune *) reg my_reg;
```

Example 16–51. VHDL Code: noprune Attribute

```
signal my_reg : stdlogic;
attribute noprune: boolean;
attribute noprune of my_reg : signal is true;
```

For more information about **Preserve Fan-out Free Register Node** logic option and a list of supported devices, refer to *Preserve Fan-out Free Register logic option* in Quartus II Help.

Keep Combinational Node/Implement as Output of Logic Cell

This synthesis attribute and corresponding logic option direct the Compiler to keep a wire or combinational node through logic synthesis minimizations and netlist optimizations. A wire that has a keep attribute or a node that has the **Implement as Output of Logic Cell** logic option applied becomes the output of a logic cell in the final synthesis netlist, and the name of the logic cell remains the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II Logic Analyzer.

The option cannot keep nodes that have no fan-out. You cannot maintain node names for wires with tri-state drivers, or if the signal feeds a top-level pin of the same name (the software changes the node name to a name such as <*net name*>~buf0).

You can use the **Ignore LCELL Buffers** logic option to direct Analysis & Synthesis to ignore logic cell buffers that the **Implement as Output of Logic Cell** logic option or the LCELL primitive created. If you apply this logic option to an entity, it affects all lower-level entities in the hierarchy path.

To avoid unintended design optimizations, ensure that any entity instantiated with Altera or third-party IP that relies on logic cell buffers for correct behavior does not inherit the **Ignore LCELL Buffers** logic option. For example, if an IP core uses logic cell buffers to manage high fan-out signals and inherits the **Ignore LCELL Buffers** logic option, the target device may no longer function properly.

You can turn off the **Ignore LCELL Buffers** logic option for a specific entity to override any assignments inherited from higher-level entities in the hierarchy path if logic cell buffers created by the **Implement as Output of Logic Cell** logic option or the LCELL primitive are required for correct behavior.

You can set the **Implement as Output of Logic Cell** logic option in the Quartus II software, or you can set the `keep` attribute in your HDL code, as shown in Example 16–52 through Example 16–54. In these examples, the Compiler maintains the node name `my_wire`.

In addition to `keep`, the Quartus II software supports the `syn_keep` attribute name for compatibility with other synthesis tools.

Example 16–52. Verilog HDL Code: keep Attribute

```
wire my_wire /* synthesis keep = 1 */;
```

Example 16–53. Verilog-2001 Code: keep Attribute

```
(* keep = 1 *) wire my_wire;
```

Example 16–54. VHDL Code: syn_keep Attribute

```
signal my_wire: bit;
attribute syn_keep: boolean;
attribute syn_keep of my_wire: signal is true;
```

For more information about the **Implement as Output of Logic Cell** logic option and the supported devices, refer to *Implement as Output of Logic Cell logic option* in Quartus II Help.

Disabling Synthesis Netlist Optimizations with dont_retime Attribute

This attribute disables synthesis retiming optimizations on the register you specify. When applied to a design entity, it applies to all registers in the entity.

You can turn off retiming optimizations with this option and prevent node name changes, so that the Compiler can correctly use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II software to disable retiming along with other synthesis netlist optimizations, or you can set the `dont_retime` attribute in your HDL code, as shown in Example 16–55 through Example 16–57. In these examples, the code prevents `my_reg` register from being retimed.

Example 16–55. Verilog HDL Code: dont_retime Attribute

```
reg my_reg /* synthesis dont_retime */;
```

Example 16–56. Verilog-2001 and SystemVerilog Code: dont_retime Attribute

```
(* dont_retime *) reg my_reg;
```

Example 16–57. VHDL Code: dont_retime Attribute

```
signal my_reg : std_logic;
attribute dont_retime : boolean;
attribute dont_retime of my_reg : signal is true;
```

For compatibility with third-party synthesis tools, Quartus II Integrated Synthesis also supports the attribute `syn_allow_retiming`. To disable retiming, set `syn_allow_retiming` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when you set the attribute to `1` or `true`.

Disabling Synthesis Netlist Optimizations with dont_replicate Attribute

This attribute disables synthesis replication optimizations on the register you specify. When applied to a design entity, it applies to all registers in the entity.

You can turn off register replication (or duplication) optimizations with this option, so that the Compiler uses your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II software to disable replication along with other synthesis netlist optimizations, or you can set the `dont_replicate` attribute in your HDL code, as shown in Example 16–58 through Example 16–60. In these examples, the code prevents the replication of the `my_reg` register.

Example 16–58. Verilog HDL Code: dont_replicate Attribute

```
reg my_reg /* synthesis dont_replicate  */;
```

Example 16–59. Verilog-2001 and SystemVerilog Code: dont_replicate Attribute

```
(* dont_replicate *) reg my_reg;
```

Example 16–60. VHDL Code: dont_replicate Attribute

```
signal my_reg : std_logic;
attribute dont_replicate : boolean;
attribute dont_replicate of my_reg : signal is true;
```

For compatibility with third-party synthesis tools, Quartus II Integrated Synthesis also supports the attribute `syn_replicate`. To disable replication, set `syn_replicate` to `0` (Verilog HDL) or `false` (VHDL). This attribute does not have any effect when you set the attribute to `1` or `true`.

Maximum Fan-Out

This **Maximum Fan-Out** attribute and logic option direct the Compiler to control the number of destinations that a node feeds. The Compiler duplicates a node and splits its fan-out until the individual fan-out of each copy falls below the maximum fan-out restriction. You can apply this option to a register or a logic cell buffer, or to a design entity that contains these elements. You can use this option to reduce the load of critical signals, which can improve performance. You can use the option to instruct the Compiler to duplicate a register that feeds nodes in different locations on the target device. Duplicating the register can enable the Fitter to place these new registers closer to their destination logic to minimize routing delay.

To turn off the option for a given node if you set the option at a higher level of the design hierarchy, in the **Netlist Optimizations** logic option, select **Never Allow**. If not disabled by the **Netlist Optimizations** option, the Compiler acknowledges the maximum fan-out constraint as long as the following conditions are met:

- The node is not part of a cascade, carry, or register cascade chain.
- The node does not feed itself.
- The node feeds other logic cells, DSP blocks, RAM blocks, and pins through data, address, clock enable, and other ports, but not through any asynchronous control ports (such as asynchronous clear).

The Compiler does not create duplicate nodes in these cases, because there is no clear way to duplicate the node, or to avoid the small differences in timing which could produce functional differences in the implementation (in the third condition above in which asynchronous control signals are involved). If you cannot apply the constraint because you do not meet one of these conditions, the Compiler issues a message to indicate that the Compiler ignores the maximum fan-out assignment. To instruct the Compiler not to check node destinations for possible problems such as the third condition, you can set the **Netlist Optimizations** logic option to **Always Allow** for a given node.

If you have enabled any of the Quartus II netlist optimizations that affect registers, add the `preserve` attribute to any registers to which you have set a `maxfan` attribute. The `preserve` attribute ensures that the netlist optimization algorithms, such as register retiming, do not affect the registers.

For details about netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

You can set the **Maximum Fan-Out** logic option in the Quartus II software. This option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code, as shown in Example 16–61 through Example 16–63. In these examples, the Compiler duplicates the `clk_gen` register, so its fan-out is not greater than 50.

In addition to `maxfan`, the Quartus II software supports the `syn_maxfan` attribute for compatibility with other synthesis tools.

Example 16–61. Verilog HDL Code: syn_maxfan Attribute

```
reg clk_gen /* synthesis syn_maxfan = 50 */;
```

Example 16–62. Verilog-2001 Code: maxfan Attribute

```
(* maxfan = 50 *) reg clk_gen;
```

Example 16–63. VHDL Code: maxfan Attribute

```
signal clk_gen : stdlogic;
attribute maxfan : signal ;
attribute maxfan of clk_gen : signal is 50;
```

For more information about the **Maximum Fan-Out** logic option and the supported devices, refer to *Maximum Fan-Out logic option* in Quartus II Help.

Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable

The **Auto Clock Enable Replacement** logic option allows the software to find logic that feeds a register and move the logic to the register's clock enable input port. To solve fitting or performance issues with designs that have many clock enables, you can turn off this option for individual registers or design entities. Turning the option off prevents the software from using the register's clock enable port. The software implements the clock enable functionality using multiplexers in logic cells.

If the software does not move the specific logic to a clock enable input with the **Auto Clock Enable Replacement** logic option, you can instruct the software to use a direct clock enable signal. The attribute ensures that the signal drives the clock enable port, and the software does not optimize or combine the signal with other logic.

Example 16–64 through Example 16–66 show how to set this attribute to ensure that the attribute preserves the signal and uses the signal as a clock enable.

In addition to `direct_enable`, the Quartus II software supports the `syn_direct_enable` attribute name for compatibility with other synthesis tools.

Example 16–64. Verilog HDL Code: direct_enable attribute

```
wire my_enable /* synthesis direct_enable = 1 */ ;
```

Example 16–65. Verilog-2001 and SystemVerilog Code: syn_direct_enable attribute

```
(* syn_direct_enable *) wire my_enable;
```

Example 16–66. VHDL Code: direct_enable attribute

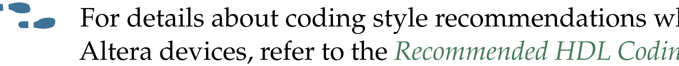
```
attribute direct_enable: boolean;
attribute direct_enable of my_enable: signal is true;
```

For more information about the **Auto Clock Enable Replacement** logic option and the supported devices, refer to *Auto Clock Enable Replacement logic option* in Quartus II Help.

Inferring Multiplier, DSP, and Memory Functions from HDL Code

The Quartus II Compiler automatically recognizes multipliers, multiply-accumulators, multiply-adders, or memory functions described in HDL code, and either converts the HDL code into respective megafunction or maps them directly to device atoms or memory atoms. If the software converts the HDL code into a megafunction, the software uses the Altera megafunction code when you compile your design, even when you do not specifically instantiate the megafunction. The software infers megafunctions to take advantage of logic that you optimize for Altera devices. The area and performance of such logic can be better than the results from inferring generic logic from the same HDL code.

Additionally, you must use megafunctions to access certain architecture-specific features, such as RAM, DSP blocks, and shift registers that provide improved performance compared with basic logic cells.



For details about coding style recommendations when targeting megafunctions in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

The Quartus II software provides options to control the inference of certain types of megafunctions, as described in the following subsections:

- "Multiply-Accumulators and Multiply-Adders"
- "Shift Registers" on page 16–50
- "RAM and ROM" on page 16–51
- "Resource Aware RAM, ROM, and Shift-Register Inference" on page 16–51
- "Auto RAM to Logic Cell Conversion" on page 16–52

Multiply-Accumulators and Multiply-Adders

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. To disable inference, turn off this option for the entire project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or turn off the option for a specific block with the Assignment Editor. By default, the software enables this logic option for Stratix V devices.

For more information about the **Auto DSP Block Replacement** logic option and the supported devices, refer to *Auto DSP Block Replacement logic option* in Quartus II Help.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option has three settings: **Off**, **Auto** and **Always**. **Auto** is the default setting in which Quartus II Integrated Synthesis decides which shift registers to replace or leave in registers. Placing shift registers in memory saves logic area, but can have a negative effect on f_{max}. Quartus II Integrated Synthesis uses the optimization technique setting, logic and RAM utilization of your design, and timing information from **Timing-Driven Synthesis** to determine which shift registers are located in memory and which are located in registers. To disable inference, turn off this option for the entire project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box by clicking **More Settings** and setting the option to **Off**. You can also disable the option for a specific block with the Assignment Editor. Even if you set the logic option to **On** or **Auto**, the software might not infer small shift registers because small shift registers do not benefit from implementation in dedicated memory. However, you can use the **Allow Any Shift Register Size for Recognition** logic option to instruct synthesis to infer a shift register even when its size is too small.

You can use the **Allow Shift Register Merging across Hierarchies** option to prevent the Compiler from merging shift registers in different hierarchies into one larger shift register. The option has three settings: **On**, **Off**, and **Auto**. The **Auto** setting is the default setting, and the Compiler decides whether or not to merge shift registers across hierarchies. When you turn on this option, the Compiler allows all shift registers to merge across hierarchies, and when you turn off this option, the Compiler does not allow any shift registers to merge across hierarchies. You can set this option globally or on entities or individual nodes.

The registers that the software maps to the ALTSHIFT_TAPS megafunction and places in RAM are not available in the Simulator because their node names do not exist after synthesis.

The Compiler turns off the **Auto Shift Register Replacement** logic option when you select a formal verification tool on the **EDA Tool Settings** page. If you do not select a formal verification tool, the Compiler issues a warning and the compilation report lists shift registers that the logic option might infer. To enable a megafunction for the shift register in the formal verification flow, you can either instantiate a shift register explicitly with the MegaWizard™ Plug-In Manager or make the shift register into a black box in a separate entity or module.

For more information about the **Auto Shift Register Replacement** logic option and the supported devices, refer to *Auto Shift Register Replacement logic option* in Quartus II Help.

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. To disable inference, turn off the appropriate option for the entire project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box by clicking **More Settings** and setting the option to **Off**. You can also disable the option for a specific block with the Assignment Editor.

Although the software implements inferred shift registers in RAM blocks, you cannot turn off the **Auto RAM Replacement** option to disable shift register replacement. Use the **Auto Shift Register Replacement** option (refer to "Shift Registers" on page 16–50).

The software might not infer very small RAM or ROM blocks because you can implement very small memory blocks with the registers in the logic. However, you can use the **Allow Any RAM Size for Recognition** and **Allow Any ROM Size for Recognition** logic options to instruct synthesis to infer a memory block even when its size is too small.

The software turns off the **Auto ROM Replacement** logic option when you select a formal verification tool in the **EDA Tool Settings** page. If you do not select a formal verification tool, the software issues a warning and a report panel provides a list of ROMs that the logic option might infer. To enable a megafunction for the shift register in the formal verification flow, you can either instantiate a ROM explicitly using the MegaWizard Plug-In Manager or create a black box for the ROM in a separate entity or in a separate module.

Although formal verification tools do not support inferred RAM blocks, due to the importance of inferring RAM in many designs, the software turns on the **Auto RAM Replacement** logic option when you select a formal verification tool in the **EDA Tool Settings** page. The software automatically performs black box instance for any module or entity that contains an inferred RAM block. The software issues a warning and lists the black box created in the compilation report. This black box allows formal verification tools to proceed; however, the formal verification tool cannot verify the entire module or entire entity that contains the RAM. Altera recommends that you explicitly instantiate RAM blocks in separate modules or in separate entities so that the formal verification tool can verify as much logic as possible.

For more information about the **Auto RAM Replacement** and **Auto ROM Replacement** logic options and their supported devices, refer to *Auto RAM Replacement logic option* and *Auto ROM Replacement logic option* in Quartus II Help.

Resource Aware RAM, ROM, and Shift-Register Inference

The Quartus II Integrated Synthesis considers resource usage when inferring RAM, ROM, and shift registers. During RAM, ROM, and shift register inferencing, synthesis looks at the number of memories available in the current device and does not infer more memory than is available to avoid a no-fit error. Synthesis tries to select the memories that are not inferred in a way that aims at the smallest increase in logic and registers.

Resource aware RAM, ROM and shift register inference is controlled by the **Resource Aware Inference for Block RAM** option. You can disable this option for the entire project in the **More Analysis & Synthesis Settings** dialog box, or per partition in the Assignment Editor.

When you select the **Auto** setting, resource aware RAM, ROM, and shift register inference use the resource counts from the largest device.

For designs with multiple partitions, Quartus II Integrated Synthesis considers one partition at a time. Therefore, for each partition, it assumes that all RAM blocks are available to that partition. If this causes a no-fit error, you can limit the number of RAM blocks available per partition with the **Maximum Number of M512 Memory Blocks**, **Maximum Number of M4K/M9K/M20K/M10K Memory Blocks**, **Maximum Number of M-RAM/M144K Memory Blocks and Maximum Number of LABs** settings in the Assignment Editor. The balancer also uses these options. For more information, refer to "Limiting Resource Usage in Partitions" on page 16–32.

Auto RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** logic option allows Quartus II Integrated Synthesis to convert small RAM blocks to logic cells if the logic cell implementation gives better quality of results. The software converts only single-port or simple-dual port RAMs with no initialization files to logic cells. You can set this option globally or apply it to individual RAM nodes. You can enable this option by turning on the appropriate option for the entire project in the **More Analysis & Synthesis Settings** dialog box.

For Arria GX and Stratix family of devices, the software uses the following rules to determine the placement of a RAM, either in logic cells or a dedicated RAM block:

- If the number of words is less than 16, use a RAM block if the total number of bits is greater than or equal to 64.
- If the number of words is greater than or equal to 16, use a RAM block if the total number of bits is greater than or equal to 32.
- Otherwise, implement the RAM in logic cells.

For the Cyclone family of devices, the software uses the following rules:

- If the number of words is greater than or equal to 64, use a RAM block.
- If the number of words is greater than or equal to 16 and less than 64, use a RAM block if the total number of bits is greater than or equal to 128.
- Otherwise, implement the RAM in logic cells.

For more information about the **Auto RAM to Logic Cell Conversion** logic options and the supported devices, refer to *Auto RAM to Logic Cell Conversion logic option* in Quartus II Help.

RAM Style and ROM Style—for Inferred Memory

These attributes specify the implementation for an inferred RAM or ROM block. You can specify the type of TriMatrix embedded memory block, or specify the use of standard logic cells (LEs or ALMs). The Quartus II software supports the attributes only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The `M512`, `M4K`, `M-RAM`, `MLAB`, `M9K`, `M144K`, `M20K`, and `M10K` values (as applicable for the target device family) indicate the type of memory block to use for the inferred RAM or ROM. If you set the attribute to a block type that does not exist in the target device family, the software generates a warning and ignores the assignment. The `logic` value indicates that the Quartus II software implements the RAM or ROM in regular logic rather than dedicated memory blocks. You can set the attribute on a module or entity, in which case it specifies the default implementation style for all inferred memory blocks in the immediate hierarchy. You can also set the attribute on a specific signal (VHDL) or variable (Verilog HDL) declaration, in which case it specifies the preferred implementation style for that specific memory, overriding the default implementation style.

If you specify a `logic` value, the memory appears as a RAM or ROM block in the RTL Viewer, but Integrated Synthesis converts the memory to regular logic during synthesis.

In addition to `ramstyle` and `romstyle`, the Quartus II software supports the `syn_ramstyle` attribute name for compatibility with other synthesis tools.

Example 16–67 through Example 16–69 specify that you must implement all memory in the module or the `my_memory_blocks` entity with a specific type of block.

Example 16–67. Verilog-1995 Code: Applying a romstyle Attribute to a Module Declaration

```
module my_memory_blocks (...) /* synthesis romstyle = "M4K" */;
```

Example 16–68. Verilog-2001 and SystemVerilog Code: Applying a ramstyle Attribute to a Module Declaration

```
 (* ramstyle = "M512" *) module my_memory_blocks (...);
```

Example 16–69. VHDL Code: Applying a romstyle Attribute to an Architecture

```
architecture rtl of my_ my_memory_blocks is
attribute romstyle : string;
attribute romstyle of rtl : architecture is "M-RAM";
begin
```

Example 16–70 through Example 16–72 specify that you must implement the inferred my_ram or my_rom memory with regular logic instead of a TriMatrix memory block.

Example 16–70. Verilog-1995 Code: Applying a syn_ramstyle Attribute to a Variable Declaration

```
reg [0:7] my_ram[0:63] /* synthesis syn_ramstyle = "logic" */;
```

Example 16–71. Verilog-2001 and SystemVerilog Code: Applying a romstyle Attribute to a Variable Declaration

```
(* romstyle = "logic" *) reg [0:7] my_rom[0:63];
```

Example 16–72. VHDL Code: Applying a ramstyle Attribute to a Signal Declaration

```
type memory_t is array (0 to 63) of std_logic_vector (0 to 7);
signal my_ram : memory_t;
attribute ramstyle : string;
attribute ramstyle of my_ram : signal is "logic";
```

You can control the depth of an inferred memory block and optimize its usage with the max_depth attribute. You can also optimize the usage of the memory block with this attribute. Example 16–73 through Example 16–75 specify the depth of the inferred memory mem using the max_depth synthesis attribute.

Example 16–73. Verilog-1995 Code: Applying a max_depth Attribute to a Variable Declaration

```
reg [7:0] mem [127:0] /* synthesis max_depth = 2048 */
```

Example 16–74. Verilog-2001 and SystemVerilog Code: Applying a max_depth Attribute to a Variable Declaration

```
(* max_depth = 2048*) reg [7:0] mem [127:0];
```

Example 16–75. VHDL Code: Applying a max_depth Attribute to a Variable Declaration

```
type ram_block is array (0 to 31) of std_logic_vector (2 downto 0);
signal mem : ram_block;
attribute max_depth : natural;
attribute max_depth OF mem : signal is 2048;
```

The syntax for setting these attributes in HDL is the same as the syntax for other synthesis attributes, as shown in "Synthesis Attributes" on page 16–25.

RAM Style Attribute—For Shift Registers Inference

The RAM style attribute for shift register allows you to use the RAM style attribute for shift registers, just as you use them for RAM or ROMs. The Quartus II Synthesis uses the RAM style attribute during shift register inference. If synthesis infers the shift register to RAM, it will be sent to the requested RAM block type. Shift registers are merged only if the RAM style attributes are compatible. If the RAM style is set to logic, a shift register does not get inferred to RAM.

Example 16–76 shows the code to set the RAM style attribute for shift registers in Verilog.

Example 16–76. Verilog Code: Setting the RAM Style Attribute for Shift Registers

```
(* ramstyle = "mlab" *)reg [N-1:0] sr;
```

Example 16–77 shows the code to set the RAM style attribute for shift registers in VHDL.

Example 16–77. VHDL Code: Setting the RAM Style Attribute for Shift Registers

```
attribute ramstyle : string;attribute ramstyle of sr : signal is "M20K";
```

You can also assign the RAM style attribute for shift registers globally, which will affect all shift registers.

Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute

Setting the `no_rw_check` value for the `ramstyle` attribute, or turning off the corresponding global **Add Pass-Through Logic to Inferred RAMs** logic option indicates that your design does not depend on the behavior of the inferred RAM when there are reads and writes to the same address in the same clock cycle. If you specify the attribute or turn off the logic option, the Quartus II software can choose a read-during-write behavior instead of using the read-during-write behavior of your HDL source code.

Sometimes, you must map an inferred RAM into regular logic cells because the inferred RAM has a read-during-write behavior that the TriMatrix memory blocks in your target device do not support. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source to increase the area of your design and potentially reduce its performance. In some of these cases, you can use the attribute to specify that the software can implement the RAM directly in a TriMatrix memory block without using logic. You can also use the attribute to prevent a warning message for dual-clock RAMs in the case that the inferred behavior in the device does not exactly match the read-during-write conditions described in the HDL code.

For more information about recommended styles for inferring RAM and some of the issues involved with different read-during-write conditions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

To set the **Add Pass-Through Logic to Inferred RAMs** logic option with the Quartus II software, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Example 16–78 and Example 16–79 use two addresses and normally require extra logic after the RAM to ensure that the read-during-write conditions in the device match the HDL code. If your design does not require a defined read-during-write condition, the extra logic is not necessary. With the `no_rw_check` attribute, Quartus II Integrated Synthesis does not generate the extra logic.

Example 16–78. Verilog HDL Inferred RAM Using no_rw_check Attribute

```
module ram_infer (q, wa, ra, d, we, clk);
    output [7:0] q;
    input [7:0] d;
    input [6:0] wa;
    input [6:0] ra;
    input we, clk;
    reg [6:0] read_add;
    (* ramstyle = "no_rw_check" *) reg [7:0] mem [127:0];
    always @ (posedge clk) begin
        if (we)
            mem[wa] <= d;
        read_add <= ra;
    end
    assign q = mem[read_add];
endmodule
```

Example 16–79. VHDL Inferred RAM Using no_rw_check Attribute

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY ram IS
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        write_address: IN INTEGER RANGE 0 to 31;
        read_address: IN INTEGER RANGE 0 to 31;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END ram;

ARCHITECTURE rtl OF ram IS
    TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL ram_block: MEM;
    ATTRIBUTE ramstyle : string;
    ATTRIBUTE ramstyle of ram_block : signal is "no_rw_check";
    SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_address) <= data;
            END IF;
            read_address_reg <= read_address;
        END IF;
    END PROCESS;
    q <= ram_block(read_address_reg);
END rtl;
```

You can use a ramstyle attribute with the MLAB value, so that the Quartus II software can infer a small RAM block and place it in an MLAB.

You can use this attribute in cases in which some asynchronous RAM blocks might be coded with read-during-write behavior that does not match the Stratix III, Stratix IV, and Stratix V architectures. Thus, the device behavior would not exactly match the behavior that the code describes. If the difference in behavior is acceptable in your design, use the `ramstyle` attribute with the `no_rw_check` value to specify that the software should not check the read-during-write behavior when inferring the RAM. When you set this attribute, Quartus II Integrated Synthesis allows the behavior of the output to differ when the asynchronous read occurs on an address that had a write on the most recent clock edge. That is, the functional HDL simulation results do not match the hardware behavior if you write to an address that is being read. To include these attributes, set the value of the ramstyle attribute to `MLAB`, `no_rw_check`.

Example 16–80 and Example 16–81 show the method of setting two values to the `ramstyle` attribute with a small asynchronous RAM block, with the ramstyle synthesis attribute set, so that the software can implement the memory in the MLAB memory block and so that the read-during-write behavior is not important. Without the attribute, this design requires 512 registers and 240 ALUTs. With the attribute, the design requires eight memory ALUTs and only 15 registers.

Example 16–80. Verilog HDL Inferred RAM Using no_rw_check and MLAB Attributes

```
module async_ram (
    input   [5:0] addr,
    input   [7:0] data_in,
    input         clk,
    input         write,
    output  [7:0] data_out );

   (* ramstyle = "MLAB, no_rw_check" *) reg [7:0] mem[0:63];

   assign  data_out = mem[addr];

   always @ (posedge clk)
   begin
      if (write)
         mem[addr] = data_in;
   end
endmodule
```

Example 16–81. VHDL Inferred RAM Using no_rw_check and MLAB Attributes

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY ram IS
   PORT (
      clock: IN STD_LOGIC;
      data: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
      write_address: IN INTEGER RANGE 0 to 31;
      read_address: IN INTEGER RANGE 0 to 31;
      we: IN STD_LOGIC;
      q: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END ram;

ARCHITECTURE rtl OF ram IS
   TYPE MEM IS ARRAY(0 TO 31) OF STD_LOGIC_VECTOR(2 DOWNTO 0);
   SIGNAL ram_block: MEM;
   ATTRIBUTE ramstyle : string;
   ATTRIBUTE ramstyle of ram_block : signal is "MLAB , no_rw_check";
   SIGNAL read_address_reg: INTEGER RANGE 0 to 31;
BEGIN
   PROCESS (clock)
   BEGIN
      IF (clock'event AND clock = '1') THEN
         IF (we = '1') THEN
            ram_block(write_address) <= data;
         END IF;
         read_address_reg <= read_address;
      END IF;
   END PROCESS;
   q <= ram_block(read_address_reg);
END rtl;
```

For more information about the **Add Pass-Through Logic to Inferred RAMs** logic option and the supported devices, refer to *Add Pass-Through Logic to Inferred RAMs logic option* in Quartus II Help.

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory with a **.mif**. The attribute takes a string value containing the name of the RAM initialization file.

Example 16–82. Verilog-1995 Code: Applying a ram_init_file Attribute

```
reg [7:0] mem[0:255] /* synthesis ram_init_file
= " my_init_file.mif" */;
```

Example 16–83. Verilog-2001 Code: Applying a ram_init_file Attribute

```
(* ram_init_file = "my_init_file.mif" *) reg [7:0] mem[0:255];
```

Example 16–84. VHDL Code: Applying a ram_init_file Attribute

```
type mem_t is array(0 to 255) of unsigned(7 downto 0);
signal ram : mem_t;
attribute ram_init_file : string;
attribute ram_init_file of ram :
signal is "my_init_file.mif";
```

In VHDL, you can also initialize the contents of an inferred memory by specifying a default value for the corresponding signal. In Verilog HDL, you can use an initial block to specify the memory contents. Quartus II Integrated Synthesis automatically converts the default value into a **.mif** for the inferred RAM.

The `ram_init_file` attribute is supported for ROM too. For more information, refer to *Inferring ROM Functions from HDL Code* section in the *Recommended HDL Coding Styles* chapter of the *Quartus II Handbook*.

Multiplier Style—for Inferred Multipliers

The `multstyle` attribute specifies the implementation style for multiplication operations (`*`) in your HDL source code. You can use this attribute to specify whether you prefer the Compiler to implement a multiplication operation in general logic or dedicated hardware, if available in the target device.

The `multstyle` attribute takes a string value of `"logic"` or `"dsp"`, indicating a preferred implementation in logic or in dedicated hardware, respectively. In Verilog HDL, apply the attribute to a module declaration, a variable declaration, or a specific binary expression that contains the `*` operator. In VHDL, apply the synthesis attribute to a signal, variable, entity, or architecture.

Specifying a `multstyle` of `"dsp"` does not guarantee that the Quartus II software can implement a multiplication in dedicated DSP hardware. The final implementation depends on several conditions, including the availability of dedicated hardware in the target device, the size of the operands, and whether or not one or both operands are constant.

In addition to `multstyle`, the Quartus II software supports the `syn_multstyle` attribute name for compatibility with other synthesis tools.

When applied to a Verilog HDL module declaration, the attribute specifies the default implementation style for all instances of the `*` operator in the module. For example, in the following code examples, the `multstyle` attribute directs the Quartus II software to implement all multiplications inside module `my_module` in the dedicated multiplication hardware.

Example 16–85. Verilog-1995 Code: Applying a multstyle Attribute to a Module Declaration

```
module my_module (...) /* synthesis multstyle = "dsp" */;
```

Example 16–86. Verilog-2001 Code: Applying a multstyle Attribute to a Module Declaration

```
(* multstyle = "dsp" *) module my_module(...);
```

When applied to a Verilog HDL variable declaration, the attribute specifies the implementation style for a multiplication operator, which has a result directly assigned to the variable. The attribute overrides the `multstyle` attribute with the enclosing module, if present. In Example 16–87 and Example 16–88, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in logic rather than the dedicated hardware.

Example 16–87. Verilog-2001 Code: Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
(* multstyle = "logic" *) wire [17:0] result;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

Example 16–88. Verilog-1995 Code: Applying a multstyle Attribute to a Variable Declaration

```
wire [8:0] a, b;
wire [17:0] result /* synthesis multstyle = "logic" */;
assign result = a * b;  //Multiplication must be
                        //directly assigned to result
```

When applied directly to a binary expression that contains the `*` operator, the attribute specifies the implementation style for that specific operator alone and overrides any `multstyle` attribute with the target variable or enclosing module. In Example 16–89, the `multstyle` attribute indicates that you must implement `a * b` in the dedicated hardware.

Example 16–89. Verilog-2001 Code: Applying a multstyle Attribute to a Binary Expression

```
wire [8:0] a, b;
wire [17:0] result;
assign result = a * (* multstyle = "dsp" *) b;
```

You cannot use Verilog-1995 attribute syntax to apply the `multstyle` attribute to a binary expression.

When applied to a VHDL entity or architecture, the attribute specifies the default implementation style for all instances of the `*` operator in the entity or architecture. In Example 16–90, the `multstyle` attribute directs the Quartus II software to use dedicated hardware, if possible, for all multiplications inside architecture `rtl` of entity `my_entity`.

Example 16–90. VHDL Code: Applying a multstyle Attribute to an Architecture

```
architecture rtl of my_entity is
   attribute multstyle : string;
   attribute multstyle of rtl : architecture is "dsp";
begin
```

When applied to a VHDL signal or variable, the attribute specifies the implementation style for all instances of the `*` operator, which has a result directly assigned to the signal or variable. The attribute overrides the `multstyle` attribute with the enclosing entity or architecture, if present. In Example 16–91, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in logic rather than the dedicated hardware.

Example 16–91. VHDL Code: Applying a multstyle Attribute to a Signal or Variable

```
signal a, b : unsigned(8 downto 0);
signal result : unsigned(17 downto 0);

attribute multstyle : string;
attribute multstyle of result : signal is "logic";
result <= a * b;
```

Full Case Attribute

A Verilog HDL case statement is full when its case items cover all possible binary values of the case expression or when a default case statement is present. A `full_case` attribute attached to a case statement header that is not full forces synthesis to treat the unspecified states as a don't care value. VHDL case statements must be full, so the attribute does not apply to VHDL.

Using this attribute on a case statement that is not full allows you to avoid the latch inference problems discussed in the *Recommended Design Practices* chapter in volume 1 of the *Quartus II Handbook*.

Latches have limited support in formal verification tools. Do not infer latches unintentionally, for example, through an incomplete case statement when using formal verification. Formal verification tools support the `full_case` synthesis attribute (with limited support for attribute syntax, as described in "Synthesis Attributes" on page 16–25).

Using the `full_case` attribute might cause a simulation mismatch between the Verilog HDL functional and the post-Quartus II simulation because unknown case statement cases can still function as latches during functional simulation. For example, a simulation mismatch can occur with the code in Example 16–92 when `sel` is `2'b11` because a functional HDL simulation output behaves as a latch and the Quartus II simulation output behaves as a don't care value.

Altera recommends making the case statement "full" in your regular HDL code, instead of using the `full_case` attribute.

The case statement in Example 16–92 is not full because you do not specify some `sel` binary values. Because you use the `full_case` attribute, synthesis treats the output as “don’t care” when the `sel` input is `2'b11`.

Example 16–92. Verilog HDL Code: a full_case Attribute

```
module full_case (a, sel, y);
   input [3:0] a;
   input [1:0] sel;
   output y;
   reg y;
   always @ (a or sel)
   case (sel) // synthesis full_case
      2'b00: y=a[0];
      2'b01: y=a[1];
      2'b10: y=a[2];
   endcase
endmodule
```

Verilog-2001 syntax also accepts the statements in Example 16–93 in the `case` header instead of the comment form as shown in Example 16–92.

Example 16–93. Verilog-2001 Syntax for the full_case Attribute

```
(* full_case *) case (sel)
```

Parallel Case

The `parallel_case` attribute indicates that you must consider a Verilog HDL case statement as parallel; that is, you can match only one case item at a time. Case items in Verilog HDL case statements might overlap. To resolve multiple matching case items, the Verilog HDL language defines a priority among case items in which the case statement always executes the first case item that matches the case expression value. By default, the Quartus II software implements the extra logic necessary to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement header allows the Quartus II software to consider its case items as inherently parallel; that is, at most one case item matches the case expression value. Parallel case items simplify the generated logic.

In VHDL, the individual choices in a case statement might not overlap, so they are always parallel and this attribute does not apply.

Altera recommends that you use this attribute only when the `case` statement is truly parallel. If you use the attribute in any other situation, the generated logic does not match the functional simulation behavior of the Verilog HDL.

Altera recommends that you avoid using the `parallel_case` attribute, because you may mismatch the Verilog HDL functional and the post-Quartus II simulation.

If you specify SystemVerilog-2005 as the supported Verilog HDL version for your design, you can use the SystemVerilog keyword `unique` to achieve the same result as the `parallel_case` directive without causing simulation mismatches.

Example 16–94 shows a casez statement with overlapping case items. In functional HDL simulation, the software prioritizes the three case items by the bits in sel. For example, sel[2] takes priority over sel[1], which takes priority over sel[0]. However, the synthesized design can simulate differently because the parallel_case attribute eliminates this priority. If more than one bit of sel is high, more than one output (a, b, or c) is high as well, a situation that cannot occur in functional HDL simulation.

Example 16–94. Verilog HDL Code: a parallel_case Attribute

```
module parallel_case (sel, a, b, c);
   input [2:0] sel;
   output a, b, c;
   reg a, b, c;
   always @ (sel)
   begin
      {a, b, c} = 3'b0;
      casez (sel) // synthesis parallel_case
         3'b1??: a = 1'b1;
         3'b?1?: b = 1'b1;
         3'b??1: c = 1'b1;
      endcase
   end
endmodule
```

Verilog-2001 syntax also accepts the statements as shown in Example 16–95 in the case (or casez) header instead of the comment form, as shown in Example 16–94.

Example 16–95. Verilog-2001 Syntax

```
(* parallel_case *) casez (sel)
```

Translate Off and On / Synthesis Off and On

The translate_off and translate_on synthesis directives indicate whether the Quartus II software or a third-party synthesis tool should compile a portion of HDL code that is not relevant for synthesis. The translate_off directive marks the beginning of code that the synthesis tool should ignore; the translate_on directive indicates that synthesis should resume. You can also use the synthesis_on and synthesis_off directives as a synonym for translate on and off.

You can use these directives to indicate a portion of code for simulation only. The synthesis tool reads synthesis-specific directives and processes them during synthesis; however, third-party simulation tools read the directives as comments and ignore them. Example 16–96, Example 16–97, and Example 16–98 show these directives.

Example 16–96. Verilog HDL Code: Translate Off and On

```
// synthesis translate_off
parameter tpd = 2;    // Delay for simulation
#tpd;
// synthesis translate_on
```

Example 16–97. VHDL Code: Translate Off and On

```
-- synthesis translate_off
use std.textio.all;
-- synthesis translate_on
```

Example 16–98. VHDL 2008 Code: Translate Off and On

```
/* synthesis translate_off */
use std.textio.all;
/* synthesis translate_on */
```

If you want to ignore only a portion of code in Quartus II Integrated Synthesis, you can use the Altera-specific attribute keyword `altera`. For example, use the `// altera translate_off` and `// altera translate_on` directives to direct Quartus II Integrated Synthesis to ignore a portion of code that you intend only for other synthesis tools.

Ignore translate_off and synthesis_off Directives

The **Ignore translate_off and synthesis_off Directives** logic option directs Quartus II Integrated Synthesis to ignore the `translate_off` and `synthesis_off` directives described in the previous section. Turning on this logic option allows you to compile code that you want the third-party synthesis tools to ignore; for example, megafunction declarations that the other tools treat as black boxes but the Quartus II software can compile. To set the **Ignore translate_off and synthesis_off Directives** logic option, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

For more information about the **Ignore translate_off and synthesis_off Directives** logic option and the supported devices, refer to *Ignore translate_off and synthesis_off Directives logic option* in Quartus II Help.

Read Comments as HDL

The `read_comments_as_HDL` synthesis directive indicates that the Quartus II software should compile a portion of HDL code that you commented out. This directive allows you to comment out portions of HDL source code that are not relevant for simulation, while instructing the Quartus II software to read and synthesize that same source code. Setting the `read_comments_as_HDL` directive to `on` indicates the beginning of commented code that the synthesis tool should read; setting the `read_comments_as_HDL` directive to `off` indicates the end of the code.

You can use this directive with `translate_off` and `translate_on` to create one HDL source file that includes a megafunction instantiation for synthesis and a behavioral description for simulation.

Formal verification tools do not support the `read_comments_as_HDL` directive because the tools do not recognize the directive.

In Example 16–99, Example 16–100, and Example 16–101, the Compiler synthesizes the commented code enclosed by `read_comments_as_HDL` because the directive is visible to the Quartus II Compiler. VHDL 2008 allows block comments, which comments are also supported for synthesis directives.

Because synthesis directives are case sensitive in Verilog HDL, you must match the case of the directive, as shown in the following examples.

Example 16–99. Verilog HDL Code: Read Comments as HDL

```
// synthesis read_comments_as_HDL on
// my_rom lpm_rom (.address (address),
//                  .data    (data));
// synthesis read_comments_as_HDL off
```

Example 16–100. VHDL Code: Read Comments as HDL

```
-- synthesis read_comments_as_HDL on
-- my_rom : entity lpm_rom
--   port map (
--     address => address,
--     data    => data,      );
-- synthesis read_comments_as_HDL off
```

Example 16–101. VHDL 2008 Code: Read Block Comments as HDL

```
/* synthesis read_comments_as_HDL on */
/* my_rom : entity lpm_rom
    port map (
    address => address,
    data => data, ); */
    synthesis read_comments_as_HDL off */
```

Use I/O Flipflops

The `useioff` attribute directs the Quartus II software to implement input, output, and output enable flipflops (or registers) in I/O cells that have fast, direct connections to an I/O pin, when possible. To improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times, you can apply the `useioff` synthesis attribute. The **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options support this synthesis attribute. You can also set this synthesis attribute in the Assignment Editor.

The `useioff` synthesis attribute takes a boolean value. You can apply the value only to the port declarations of a top-level Verilog HDL module or VHDL entity (it is ignored if applied elsewhere). Setting the value to `1` (Verilog HDL) or `TRUE` (VHDL) instructs the Quartus II software to pack registers into I/O cells. Setting the value to `0` (Verilog HDL) or `FALSE` (VHDL) prevents register packing into I/O cells.

In Example 16–102 and Example 16–103, the `useioff` synthesis attribute directs the Quartus II software to implement the `a_reg`, `b_reg`, and `o_reg` registers in the I/O cells corresponding to the `a`, `b`, and `o` ports, respectively.

Example 16–102. Verilog HDL Code: the useioff Attribute

```
module top_level(clk, a, b, o);
       input clk;
       input [1:0] a, b /* synthesis useioff = 1 */;
       output [2:0] o /* synthesis useioff = 1 */;
       reg [1:0] a_reg, b_reg;
       reg [2:0] o_reg;
       always @ (posedge clk)
       begin
           a_reg <= a;
           b_reg <= b;
           o_reg <= a_reg + b_reg;
       end
       assign o = o_reg;
endmodule
```

Example 16–103 and Example 16–104 show that the Verilog-2001 syntax also accepts the type of statements instead of the comment form in Example 16–102.

Example 16–103. Verilog-2001 Code: the useioff Attribute

```
(* useioff = 1 *)   input [1:0] a, b;
(* useioff = 1 *)   output [2:0] o;
```

Example 16–104. VHDL Code: the useioff Attribute

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity useioff_example is
   port (
      clk  : in  std_logic;
      a, b : in  unsigned(1 downto 0);
      o    : out unsigned(1 downto 0));
   attribute useioff : boolean;
   attribute useioff of a : signal is true;
   attribute useioff of b : signal is true;
   attribute useioff of o : signal is true;
end useioff_example;
architecture rtl of useioff_example is
   signal o_reg, a_reg, b_reg : unsigned(1 downto 0);
begin
   process(clk)
   begin
      if (clk = '1' AND clk'event) then
          a_reg <= a;
          b_reg <= b;
          o_reg <= a_reg + b_reg;
      end if;
   end process;
o <= o_reg;
end rtl;
```

Specifying Pin Locations with chip_pin

The `chip_pin` attribute allows you to assign pin locations in your HDL source. You can use the attribute only on the ports of the top-level entity or module in your design. You can assign pins only to single-bit or one-dimensional bus ports in your design.

For single-bit ports, the value of the `chip_pin` attribute is the name of the pin on the target device, as specified by the pin table of the device.

In addition to the `chip_pin` attribute, the Quartus II software supports the `altera_chip_pin_lc` attribute name for compatibility with other synthesis tools. When using this attribute in other synthesis tools, some older device families require an “@” symbol in front of each pin assignment. In the Quartus II software, the “@” is optional.

Example 16–105 through Example 16–107 show different ways of assigning `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

Example 16–105. Verilog-1995 Code: Applying Chip Pin to a Single Pin

```
input my_pin1 /* synthesis chip_pin = "C1" */;
input my_pin2 /* synthesis altera_chip_pin_lc = "@4" */;
```

Example 16–106. Verilog-2001 Code: Applying Chip Pin to a Single Pin

```
(* chip_pin = "C1" *) input my_pin1;
(* altera_chip_pin_lc = "@4" *) input my_pin2;
```

Example 16–107. VHDL Code: Applying Chip Pin to a Single Pin

```
entity my_entity is
port(my_pin1: in std_logic; my_pin2: in std_logic;…);
end my_entity;
attribute chip_pin : string;
attribute altera_chip_pin_lc : string;
attribute chip_pin of my_pin1 : signal is "C1";
attribute altera_chip_pin_lc of my_pin2 : signal is "@4";
```

For bus I/O ports, the value of the chip pin attribute is a comma-delimited list of pin assignments. The order in which you declare the range of the port determines the mapping of assignments to individual bits in the port. To leave a bit unassigned, leave its corresponding pin assignment blank.

Example 16–108 assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

Example 16–108. Verilog-1995 Code: Applying Chip Pin to a Bus of Pins

```
input [2:0]  my_pin /* synthesis chip_pin = "4, 5, 6" */;
```

Example 16–109 reverses the order of the signals in the bus, assigning `my_pin[0]` to `Pin_4` and `my_pin[2]` to `Pin_6` but leaves `my_pin[1]` unassigned.

Example 16–109. Verilog-1995 Code: Applying Chip Pin to Part of a Bus

```
input [0:2]  my_pin /* synthesis chip_pin = "4, ,6" */;
```

Example 16–110 assigns `my_pin[2]` to Pin `4` and `my_pin[0]` to Pin `6`, but leaves `my_pin[1]` unassigned.

Example 16–110. VHDL Code: Applying Chip Pin to Part of a Bus of Pins

```
entity my_entity is
port(my_pin: in std_logic_vector(2 downto 0);…);
end my_entity;

attribute chip_pin of my_pin: signal is "4, , 6";
```

Example 16–111 shows a VHDL example on how to assign pin location and I/O standard.

Example 16–111. VHDL Code: Assigning Pin Location and I/O Standard

```
attribute altera_chip_pin_lc: string;
attribute  altera_attribute: string;
attribute altera_chip_pin_lc of clk: signal is "B13";
attribute altera_attribute of clk:signal is "-name IO_STANDARD ""3.3-V
LVCMOS""";
```

Example 16–112 shows a Verilog-2001 example on how to assign pin location and I/O standard.

Example 16–112. Verilog-2001 Code: Assigning Pin Location and I/O Standard

```
(* altera_attribute = "-name IO_STANDARD \"3.3-V LVCMOS\"" *)(* chip_pin
= "L5" *)input clk;
(* altera_attribute = "-name IO_STANDARD LVDS" *)(* chip_pin = "L4"
*)input sel;
output [3:0] data_o, input [3:0] data_i);
```

Using altera_attribute to Set Quartus II Logic Options

The `altera_attribute` attribute allows you to apply Quartus II logic options and assignments to an object in your HDL source code. You can set this attribute on an entity, architecture, instance, register, RAM block, or I/O pin. You cannot set it on an arbitrary combinational node such as a net. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL synthesis attribute (such as many of the logic options presented earlier in this chapter). You can also use this attribute to pass entity-level settings and assignments to phases of the Compiler flow that follow Analysis & Synthesis, such as Fitting.

Assignments or settings made through the Quartus II software, the **.qsf**, or the Tcl interface take precedence over assignments or settings made with the `altera_attribute` synthesis attribute in your HDL code.

The syntax for setting this attribute in HDL is the same as the syntax for other synthesis attributes, as shown in “Synthesis Attributes” on page 16–25.

The attribute value is a single string containing a list of **.qsf** variable assignments separated by semicolons, as shown in Example 16–113.

Example 16–113. Variable Assignments Separated by Semicolons

```
-name <variable_1> <value_1>;-name <variable_2> <value_2>[;…]
```

If the Quartus II option or assignment includes a target, source, and section tag, you must use the syntax in Example 16–114 for each **.qsf** variable assignment:

Example 16–114. Syntax for Each .qsf Variable Assignment

```
-name <variable> <value>
-from <source> -to <target> -section_id <section>
```

Example 16–115 shows the syntax for the full attribute value, including the optional target, source, and section tags for two different **.qsf** assignments.

Example 16–115. Syntax for Full Attribute Value

```
" -name <variable_1> <value_1> [-from <source_1>] [-to <target_1>] [-section_id \
<section_1>]; -name <variable_2> <value_2> [-from <source_2>] [-to <target_2>] \
[-section_id <section_2>] "
```

If the assigned value of a variable is a string of text, you must use escaped quotes around the value in Verilog HDL (as shown in Example 16–116), or double-quotes in VHDL (as shown in Example 16–117):

Example 16–116. Assigned Value of a Variable in Verilog HDL (With Nonexistent Variable and Value Terms)

```
"VARIABLE_NAME \"STRING_VALUE\""
```

Example 16–117. Assigned Value of a Variable in VHDL (With Nonexistent Variable and Value Terms)

```
"VARIABLE_NAME ""STRING_VALUE"""
```

To find the **.qsf** variable name or value corresponding to a specific Quartus II option or assignment, you can set the option setting or assignment in the Quartus II software, and then make the changes in the **.qsf**. You can also refer to the *Quartus II Settings File Manual*, which documents all variable names.

Example 16–118 through Example 16–120 use `altera_attribute` to set the power-up level of an inferred register.

For inferred instances, you cannot apply the attribute to the instance directly. Therefore, you must apply the attribute to one of the output nets of the instance. The Quartus II software automatically moves the attribute to the inferred instance.

Example 16–118. Verilog-1995 Code: Applying altera_attribute to an Instance

```
reg my_reg /* synthesis altera_attribute = "-name POWER_UP_LEVEL HIGH"
*/;
```

Example 16–119. Verilog-2001 Code: Applying altera_attribute to an Instance

```
(* altera_attribute = "-name POWER_UP_LEVEL HIGH" *) reg my_reg;
```

Example 16–120. VHDL Code: Applying altera_attribute to an Instance

```
signal my_reg : std_logic;
attribute altera_attribute : string;
attribute altera_attribute of my_reg: signal is "-name POWER_UP_LEVEL
HIGH";
```

Example 16–121 through Example 16–123 use the `altera_attribute` to disable the **Auto Shift Register Replacement** synthesis option for an entity. To apply the Altera Attribute to a VHDL entity, you must set the attribute on its architecture rather than on the entity itself.

Example 16–121. Verilog-1995 Code: Applying altera_attribute to an Entity

```
module my_entity(…) /* synthesis altera_attribute = "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF" */;
```

Example 16–122. Verilog-2001 Code: Applying altera_attribute to an Entity

```
(* altera_attribute = "-name AUTO_SHIFT_REGISTER_RECOGNITION OFF" *)
module my_entity(…) ;
```

Example 16–123. VHDL Code: Applying altera_attribute to an Entity

```
entity my_entity is
-- Declare generics and ports
end my_entity;
architecture rtl of my_entity is
attribute altera_attribute : string;
-- Attribute set on architecture, not entity
attribute altera_attribute of rtl: architecture is "-name
AUTO_SHIFT_REGISTER_RECOGNITION OFF";
begin
-- The architecture body
end rtl;
```

You can also use `altera_attribute` for more complex assignments that have more than one instance. In Example 16–125 through Example 16–127, the `altera_attribute` cuts all timing paths from `reg1` to `reg2`, equivalent to this Tcl or **.qsf** command, as shown in Example 16–124:

Example 16–124.

```
set_instance_assignment -name CUT ON -from reg1 -to reg2 ↵
```

Example 16–125. Verilog-1995 Code: Applying altera_attribute with the -to Option

```
reg reg2;
reg reg1 /* synthesis altera_attribute = "-name CUT ON -to reg2" */;
```

Example 16–126. Verilog-2001 and SystemVerilog Code: Applying altera_attribute with the -to Option

```
reg reg2;
(* altera_attribute = "-name CUT ON -to reg2" *) reg reg1;
```

Example 16–127. VHDL Code: Applying altera_attribute with the -to Option

```
signal reg1, reg2 : std_logic;
attribute altera_attribute: string;
attribute altera_attribute of reg1 : signal is "-name CUT ON -to reg2";
```

You can specify either the `-to` option or the `-from` option in a single `altera_attribute`; Integrated Synthesis automatically sets the remaining option to the target of the `altera_attribute`. You can also specify wildcards for either option. For example, if you specify "`*`" for the `-to` option instead of `reg2` in these examples, the Quartus II software cuts all timing paths from `reg1` to every other register in this design entity.

You can use the `altera_attribute` only for entity-level settings, and the assignments (including wildcards) apply only to the current entity.

Analyzing Synthesis Results

After performing synthesis, you can check your synthesis results in the **Analysis & Synthesis** section of the Compilation Report and the Project Navigator.

Analysis & Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation. After Analysis & Synthesis, the Summary section of the Compilation Report provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred. The **Analysis & Synthesis** section lists synthesis-specific information.

Analysis & Synthesis includes various report sections, including a list of the source files read for the project, the resource utilization by entity after synthesis, and information about state machines, latches, optimization results, and parameter settings.

For more information about each report section, refer to the *Analysis & Synthesis Summary Reports* in Quartus II Help.

Project Navigator

The **Hierarchy** tab of the Project Navigator provides a view of the project hierarchy and a summary of resource and device information about the current project. After Analysis & Synthesis, before the Fitter begins, the Project Navigator provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred.

If an entity in the Hierarchy tab contains parameter settings, a tooltip displays the settings when you hold the pointer over the entity.

Upgrade IP Components Dialog Box

In the Quartus II software version 12.1 SP1 and later, the **Upgrade IP Components** dialog box allows you to upgrade all outdated IP in your project after you move to a newer version of the Quartus II software.

For more information about the **Upgrade IP Components** dialog box, refer to *Upgrade IP Components dialog box* in Quartus II Help.

Analyzing and Controlling Synthesis Messages

This section provides information about the generated messages during synthesis, and how you can control which messages appear during compilation.

Quartus II Messages

The messages that appear during Analysis & Synthesis describe many of the optimizations during the synthesis stage, and provide information about how the software interprets your design. Altera recommends checking the messages to analyze **Critical Warnings** and **Warnings**, because these messages can relate to important design problems. Read the **Info** messages to get more information about how the software processes your design.

The software groups the messages by following types: **Info**, **Warning**, **Critical Warning**, and **Error**.

For more information about the Messages window and message suppression, refer to *About the Messages Window* and *About Message Suppression* in Quartus II Help.

For more information about the Messages, refer to *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook.*

You can specify the type of Analysis & Synthesis messages that you want to view by selecting the **Analysis & Synthesis Message Level** option. You can specify the display level by performing the following steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, click **Analysis & Synthesis Settings**.
3. Click **More Settings**. Select the level for the **Analysis & Synthesis Message Level** option.

VHDL and Verilog HDL Messages

The Quartus II software issues a variety of messages when it is analyzing and elaborating the Verilog HDL and VHDL files in your design. These HDL messages are a subset of all Quartus II messages that help you identify potential problems early in the design process.

HDL messages fall into the following categories:

- **Info message—**lists a property of your design.
- **Warning message—**indicates a potential problem in your design. Potential problems come from a variety of sources, including typos, inappropriate design practices, or the functional limitations of your target device. Though HDL warning messages do not always identify actual problems, Altera recommends investigating code that generates an HDL warning. Otherwise, the synthesized behavior of your design might not match your original intent or its simulated behavior.
- **Error message—**indicates an actual problem with your design. Your HDL code can be invalid due to a syntax or semantic error, or it might not be synthesizable as written.

In Example 16–128, the sensitivity list contains multiple copies of the variable `i`. While the Verilog HDL language does not prohibit duplicate entries in a sensitivity list, it is clear that this design has a typing error: Variable `j` should be listed on the sensitivity list to avoid a possible simulation or synthesis mismatch.

Example 16–128. Generating an HDL Warning Message

```
//dup.v
module dup(input i, input j, output reg o);
always @ (i or i)
   o = i & j;
endmodule
```

When processing the HDL code, the Quartus II software generates the warning message shown in Example 16–129:

Example 16–129.

```
Warning: (10276) Verilog HDL sensitivity list warning at dup.v(2):
sensitivity list contains multiple entries for "i".
```

In Verilog HDL, variable names are case sensitive, so the variables `my_reg` and `MY_REG` in Example 16–130 are two different variables. However, declaring variables that have names in different cases is confusing, especially if you use VHDL, in which variables are not case sensitive.

Example 16–130. Generating HDL Info Messages

```
// namecase.v
module namecase (input i, output o);
   reg my_reg;
   reg MY_REG;
   assign o = i;
endmodule
```

When processing the HDL code, the Quartus II software generates the following informational message, as shown in Example 16–131:

Example 16–131.

```
Info: (10281) Verilog HDL information at namecase.v(3): variable name
"MY_REG" and variable name "my_reg" should not differ only in case.
```

In addition, the Quartus II software generates additional HDL info messages to inform you that this small design does not use neither `my_reg` nor `MY_REG` , as shown in Example 16–132:

Example 16–132.

```
Info: (10035) Verilog HDL or VHDL information at namecase.v(3): object
"my_reg" declared but not used
Info: (10035) Verilog HDL or VHDL information at namecase.v(4): object
"MY_REG" declared but not used
```

The Quartus II software allows you to control how many HDL messages you can view during the Analysis & Elaboration of your design files. You can set the HDL Message Level to enable or disable groups of HDL messages, or you can enable or disable specific messages, as described in the following sections.

For more information about synthesis directives and their syntax, refer to “Synthesis Directives” on page 16–27.

Setting the HDL Message Level

The HDL Message Level specifies the types of messages that the Quartus II software displays when it is analyzing and elaborating your design files. Table 16–5 describes the HDL message levels.

Table 16–5. HDL Info Message Level

Level	Purpose	Description
Level1	High-severity messages only	If you want to view only the HDL messages that identify likely problems with your design, select Level1. When you select Level1, the Quartus II software issues a message only if there is an actual problem with your design.
Level2	High-severity and medium-severity messages	If you want to view additional HDL messages that identify possible problems with your design, select Level2. Level2 is the default setting.
Level3	All messages, including low-severity messages	If you want to view all HDL info and warning messages, select Level3. This level includes extra “LINT” messages that suggest changes to improve the style of your HDL code.

You must address all issues reported at the **Level1** setting. The default HDL message level is **Level2**.

To set the HDL Message Level in the Quartus II software, follow these steps:

1. On the Assignments menu, click **Settings.**
2. In the **Category** list, click **Analysis & Synthesis Settings.**
3. Set the necessary message level from the pull-down menu in the **HDL Message Level** list, and then click **OK.**

You can override this default setting in a source file with the `message_level synthesis` directive, which takes the values `level1`, `level2`, and `level3`, as shown in Example 16–133 and Example 16–134.

Example 16–133. Verilog HDL Examples of message_level Directive

```
// altera message_level level1
or
/* altera message_level level3 */
```

Example 16–134. VHDL Code: message_level Directive

```
-- altera message_level level2
```

A message_level synthesis directive remains effective until the end of a file or until the next message_level directive. In VHDL, you can use the message_level synthesis directive to set the HDL Message Level for entities and architectures, but not for other design units. An HDL Message Level for an entity applies to its architectures, unless overridden by another message_level directive. In Verilog HDL, you can use the message_level directive to set the HDL Message Level for a module.

Enabling or Disabling Specific HDL Messages by Module/Entity

Message ID is in parentheses at the beginning of the message. Use the Message ID to enable or disable a specific HDL info or warning message. Enabling or disabling a specific message overrides its HDL Message Level. This method is different from the message suppression in the Messages window because you can disable messages for a specific module or a specific entity. This method applies only to the HDL messages, and if you disable a message with this method, the Quartus II software lists the message as a suppressed message.

To disable specific HDL messages in the Quartus II software, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Advanced**.
3. In the **Advanced Message Settings** dialog box, add the Message IDs you want to enable or disable.

To enable or disable specific HDL messages in your HDL, use the message_on and message_off synthesis directives. These directives require a space-separated list of Message IDs. You can enable or disable messages with these synthesis directives immediately before Verilog HDL modules, VHDL entities, or VHDL architectures. You cannot enable or disable a message during an HDL construct.

A message enabled or disabled via a message_on or message_off synthesis directive overrides its HDL Message Level or any message_level synthesis directive. The message remains disabled until the end of the source file or until you use another message_on or message_off directive to change the status of the message.

Example 16–135. Verilog HDL message_off Directive for Message with ID 10000

```
// altera message_off 10000
or
/* altera message_off 10000 */
```

Example 16–136. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

This section provides an overview of the conventions that the Quartus II software uses during synthesis to name the nodes created from your HDL design. This information is useful for finding logic node names during verification and debugging of a design. This section focuses on the conventions for Verilog HDL and VHDL code, but discusses AHDL and **.bdf** files when appropriate.

Whenever possible, Quartus II Integrated Synthesis uses wire or signal names from your source code to name nodes such as LEs or ALMs. Some nodes, such as registers, have predictable names that do not change when a design is resynthesized, although certain optimizations can affect register names. The names of other nodes, particularly LEs or ALMs that contain only combinational logic, can change due to logic optimizations that the software performs.

This section describes the following topics:

- "Hierarchical Node-Naming Conventions"
- "Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)"
- "Register Changes During Synthesis" on page 16–80
- "Preserving Register Names" on page 16–82
- "Node-Naming Conventions for Combinational Logic Cells" on page 16–82
- "Preserving Combinational Logic Names" on page 16–83

Hierarchical Node-Naming Conventions

To make each name in your design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator indicates a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, with the ":" separator between each entity name and its instance name. For example, if a design defines entity `A` with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. You can obtain the full name of any node by starting with the hierarchical instance path, followed by a "|", and ending with the node name inside that entity. Example 16–137 shows you the convention:

Example 16–137.

<entity 0>:<instance_name 0>|<entity 1>:<instance_name 1>|...|<instance_name n>|<node_name>

For example, if entity A contains a register (DFF atom) called `my_dff`, its full hierarchy name would be `A:my_A_inst|my_dff`.

To instruct the Compiler to generate node names that do not contain entity names, on the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings**, and then turn off D**isplay entity name for node name**. With this option turned off, the node names use the convention in shown in Example 16–138:

Example 16–138.

<instance_name 0>|<instance_name 1>|...|<instance_name n> |<node_name>

Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)

In Verilog HDL and VHDL, inferred registers use the names of the `reg` or `signal` connected to the output.

Example 16–139 is an example of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 16–139. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

always @ (posedge clk)
my_dff_out <= dff_in;
```

Similarly, Example 16–140 is an example of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 16–140. VHDL Register

```
signal dff_in, my_dff_out, clk;
process (clk)
begin
if (rising_edge(clk)) then
my_dff_out <= dff_in;
end if;
end process;
```

AHDL designs explicitly declare DFF registers rather than infer, so the software uses the user-declared name for the register.

For schematic designs using a **.bdf**, your design names all elements when you instantiate the elements in your design, so the software uses the name you defined for the register or DFF.

In the special case that a wire or signal (such as `my_dff_out` in the preceding examples) is also an output pin of your top-level design, the Quartus II software cannot use that name for the register (for example, cannot use `my_dff_out`) because the software requires that all logic and I/O cells have unique names. Here, Quartus II Integrated Synthesis appends `~reg0` to the register name.

For example, the Verilog HDL code in Example 16–141 generates a register called `q~reg0`:

Example 16–141. Verilog HDL Register Feeding Output Pin

```
module my_dff (input clk, input d, output q);
always @ (posedge clk)
q <= d;
endmodule
```

This situation occurs only for registers driving top-level pins. If a register drives a port of a lower level of the hierarchy, the software removes the port during hierarchy flattening and the register retains its original name, in this case, `q`.

Register Changes During Synthesis

On some occasions, you might not find registers that you expect to view in the synthesis netlist. Logic optimization might remove registers and synthesis optimizations might change the names of the registers. Common optimizations include inference of a state machine, counter, adder-subtractor, or shift register from registers and surrounding logic. Other common register changes occur when the software packs these registers into dedicated hardware on the FPGA, such as a DSP block or a RAM block.

This section describes the following factors that can affect register names:

- "Synthesis and Fitting Optimizations"
- "State Machines"
- "Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions" on page 16–81
- "Packed Input and Output Registers of RAM and DSP Blocks" on page 16–82
- "Preserving Register Names" on page 16–82
- "Preserving Combinational Logic Names" on page 16–83

Synthesis and Fitting Optimizations

Logic optimization during synthesis might remove registers if you do not connect the registers to inputs or outputs in your design, or if you can simplify the logic due to constant signal values. Synthesis optimizations might change register names, such as when the software merges duplicate registers to reduce resource utilization.

NOT-gate push back optimizations can affect registers that use preset signals. This type of optimization can impact your timing assignments when the software uses registers as clock dividers. If this situation occurs in your design, change the clock settings to work on the new register name.

Synthesis netlist optimizations often change node names because the software can combine or duplicate registers to optimize your design.

For more information about the type of optimizations performed by synthesis netlist optimizations, refer to the *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

The Quartus II Compilation Report provides a list of registers that synthesis optimizations remove, and a brief reason for the removal. To generate the Quartus II Compilation Report, follow these steps:

1. In the **Analysis & Synthesis** folder, open **Optimization Results.**
2. Open **Register Statistics,** and then click the **Registers Removed During Synthesis** report.
3. Click **Removed Registers Triggering Further Register Optimizations**.

The second report contains a list of registers that causes synthesis optimizations to remove other registers from your design. The report provides a brief reason for the removal, and a list of registers that synthesis optimizations remove due to the removal of the initial register.

Quartus II Integrated Synthesis creates synonyms for registers duplicated with the **Maximum Fan-Out** option (or `maxfan` attribute). Therefore, timing assignments applied to nodes that are duplicated with this option are applied to the new nodes as well.

The Quartus II Fitter can also change node names after synthesis (for example, when the Fitter uses register packing to pack a register into an I/O element, or when physical synthesis modifies logic). The Fitter creates synonyms for duplicated registers so timing analysis can use the existing node name when applying assignments.

You can instruct the Quartus II software to preserve certain nodes throughout compilation so you can use them for verification or making assignments. For more information, refer to "Preserving Register Names" on page 16–82.

State Machines

If your HDL code infers a state machine, the software maps the registers that represent the states into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form in which one register represents each state. In this case, for Verilog HDL or VHDL designs, the registers take the name of the state register and the states.

For example, consider a Verilog HDL state machine in which the states are `parameter state0 = 1`, `state1 = 2`, `state2 = 3`, and in which the software declares the state machine register as `reg [1:0] my_fsm`. In this example, the three one-hot state registers are `my_fsm.state0`, `my_fsm.state1`, and `my_fsm.state2`.

An AHDL design explicitly specifies state machines with a state machine name. Your design names state machine registers with synthesized names based on the state machine name, but not the state names. For example, if a `my_fsm` state machine has four state bits, The software might synthesize these state bits with names such as `my_fsm~12`, `my_fsm~13`, `my_fsm~14`, and `my_fsm~15`.

Inferred Adder-Subtractors, Shift Registers, Memory, and DSP Functions

The Quartus II software infers megafunctions from Verilog HDL and VHDL code for logic that forms adder-subtractors, shift registers, RAM, ROM, and arithmetic functions that are placed in DSP blocks.

For information about inferring megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Because adder-subtractors are part of a megafunction instead of generic logic, the combinational logic exists in the design with different names. For shift registers, memory, and DSP functions, the software implements the registers and logic inside the dedicated RAM or DSP blocks in the device. Thus, the registers are not visible as separate LEs or ALMs.

Packed Input and Output Registers of RAM and DSP Blocks

The software packs registers into the input registers and output registers of RAM and DSP blocks, so that they are not visible as separate registers in LEs or ALMs.

For information about packing registers into RAM and DSP megafunctions, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

Preserving Register Names

Altera recommends that you preserve certain register names for verification or debugging, or to ensure that you applied timing assignments correctly. Quartus II Integrated Synthesis preserves certain nodes automatically if the software uses the nodes in a timing constraint.

Use the `preserve` attribute to instruct the Compiler not to minimize or remove a specified register during synthesis optimizations or register netlist optimizations. For details, refer to "Preserve Registers" on page 16–42.

Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow. For details, refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 16–43.

Use the synthesis attribute `syn_dont_merge` to ensure that the Compiler does not merge registers with other registers. For more information, refer to "Disable Register Merging/Don't Merge Register" on page 16–43.

Node-Naming Conventions for Combinational Logic Cells

Whenever possible for Verilog HDL, VHDL, and AHDL code, the Quartus II software uses wire names that are the targets of assignments, but can change the node names due to synthesis optimizations.

For example, consider the Verilog HDL code in Example 16–142. Quartus II Integrated Synthesis uses the names `c`, `d`, `e`, and `f` for the generated combinational logic cells.

Example 16–142. Naming Nodes for Combinational Logic Cells in Verilog HDL

```
wire c;
reg d, e, f;

assign c = a | b;
always @ (a or b)
d = a & b;
always @ (a or b) begin : my_label
e = a ^ b;
end

always @ (a or b)
f = ~(a | b);
```

For schematic designs using a **.bdf**, your design names all elements when you instantiate the elements in your design and the software uses the name you defined when possible.

If logic cells, such as those created in Example 16–142, are packed with registers in device architectures such as the Stratix and Cyclone device families, those names might not appear in the netlist after fitting. In other devices, such as newer families in the Stratix and Cyclone series device families, the register and combinational nodes are kept separate throughout the compilation, so these names are more often maintained through fitting.

When logic optimizations occur during synthesis, it is not always possible to retain the initial names as described. Sometimes, synthesized names are used, which are the wire names with a tilde (~) and a number appended. For example, if a complex expression is assigned to wire `w` and that expression generates several logic cells, those cells can have names such as `w`, `w~1`, and `w~2`. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. Quartus II Integrated Synthesis attempts to retain user names whenever possible. Any node name ending with ~<*number*> is a name created during synthesis, which can change if the design is changed and re-synthesized. Knowing these naming conventions helps you understand your post-synthesis results, helping you to debug your design or create assignments.

During synthesis, the software maintains combinational clock logic by not changing nodes that might be clocks. The software also maintains or protects multiplexers in clock trees, so that the TimeQuest analyzer has information about which paths are unate, to allow complete and correct analysis of combinational clocks. Multiplexers often occur in clock trees when the software selects between different clocks. To help with the analysis of clock trees, the software ensures that each multiplexer encountered in a clock tree is broken into 2:1 multiplexers, and each of those 2:1 multiplexers is mapped into one lookup table (independent of the device family). This optimization might result in a slight increase in area, and for some designs a decrease in timing performance. You can turn off this multiplexer protection with the option **Clock MUX Protection** under **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box.

For more information about **Clock MUX Protection** logic option and a list of supported devices, refer to *Clock MUX Protection logic option* in Quartus II Help.

Preserving Combinational Logic Names

You can preserve certain combinational logic node names for verification or debugging, or to ensure that timing assignments are applied correctly.

Use the `keep` attribute to keep a wire name or combinational node name through logic synthesis minimizations and netlist optimizations. For details, refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 16–44.

For any internal node in your design clock network, use `keep` to protect the name so that you can apply correct clock settings. Also, set the attribute for combinational logic involved in `cut` and `-through` assignments.

Setting the `keep` attribute for combinational logic can increase the area utilization and increase the delay of the final mapped logic because the attribute requires the insertion of extra combinational logic. Use the attribute only when necessary.

Scripting Support

You can run procedures and make settings in a Tcl script. You can also run some procedures at a command prompt. For detailed information about scripting command options, refer to the Quartus II Command-Line and Tcl API Help browser.

To run the Help browser, type the command at the command prompt shown in Example 16–143:

Example 16–143.

```
quartus_sh --qhelp ↵
```

For more information about Tcl scripting, refer to the *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook.* For more information about all settings and constraints in the Quartus II software, refer to the *Quartus II Settings File Manual*. For more information about command-line scripting, refer to the *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*.

For more information about Tcl scripting, refer to *API Functions for Tcl* in Quartus II Help.

You can specify many of the options described in this section either on an instance, at the global level, or both.

To make a global assignment, use the Tcl command shown in Example 16–144:

Example 16–144.

```
set_global_assignment -name <QSF Variable Name> <Value> ↵
```

To make an instance assignment, use the Tcl command shown in Example 16–145:

Example 16–145.

```
set_instance_assignment -name <QSF Variable Name> <Value>\ -to
<Instance Name> ↵
```

To set the **Synthesis Effort** option at the command line, use the `--effort` option with the `quartus_map` executable, as shown in Example 16–146.

Example 16–146. Command Syntax for Specifying Synthesis Effort Option

```
quartus_map <Design name> --effort= "auto | fast" ↵
```

The early timing estimate feature gives you preliminary timing estimates before running a full compilation, which results in a quicker iteration time; therefore, you can save significant compilation time to get a good estimation of the final timing of your design.

If you want to run fast synthesis with the Fitter **Early Timing Estimate** option, use the command shown in Example 16–147. This command runs the full flow with timing analysis:

Example 16–147. Command Syntax for Running Fast Synthesis with Early Timing Estimate Option

```
quartus_sh --flow early_timing_estimate_with_synthesis <Design name> ↵
```

For more information, refer to "Synthesis Effort" on page 16–34.

Adding an HDL File to a Project and Setting the HDL Version

To add an HDL or schematic entry design file to your project, use the Tcl assignments shown in Example 16–148:

Example 16–148.

```
set_global_assignment -name VERILOG_FILE <file name>.<v|sv>
set_global_assignment -name SYSTEMVERILOG_FILE <file name>.sv
set_global_assignment -name VHDL_FILE <file name>.<vhd|vhdl>
set_global_assignment -name AHDL_FILE <file name>.tdf
set_global_assignment -name BDF_FILE <file name>.bdf
```

You can use any file extension for design files, as long as you specify the correct language when adding the design file. For example, you can use **.h** for Verilog HDL header files.

To specify the Verilog HDL or VHDL version, use the option shown in Example 16–149, at the end of the `VERILOG_FILE` or `VHDL_FILE` command:

Example 16–149.

```
-   HDL_VERSION <language version> ↵
```

The variable <*language version*> takes one of the following values:

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`
- `VHDL_1987`
- `VHDL_1993`
- `VHDL_2008`

For example, to add a Verilog HDL file called **my_file.v** written in Verilog-1995, use the command shown in Example 16–150:

Example 16–150.

```
set_global_assignment -name VERILOG_FILE my_file.v -HDL_VERSION \
VERILOG_1995
```

In Example 16–151, the syn_encoding attribute associates a binary encoding with the states in the enumerated type count_state. In this example, the states are encoded with the following values: zero = "11", one = "01", two = "10", three = "00".

Example 16–151. Specifying User-Encoded States with the syn_encoding Attribute in VHDL

```
ARCHITECTURE rtl OF my_fsm IS
   TYPE count_state is (zero, one, two, three);
   ATTRIBUTE syn_encoding : STRING;
   ATTRIBUTE syn_encoding OF count_state : TYPE IS "11 01 10 00";
   SIGNAL present_state, next_state : count_state;
BEGIN
```

You can also use the syn_encoding attribute in Verilog HDL to direct the synthesis tool to use the encoding from your HDL code, instead of using the **State Machine Processing** option.

The syn_encoding value "user" instructs the Quartus II software to encode each state with its corresponding value from the Verilog HDL source code. By changing the values of your state constants, you can change the encoding of your state machine.

In Example 16–152, the states are encoded as follows:

```
init = "00"
last = "11"
next = "01"
later = "10"
```

Example 16–152. Verilog-2001 and SystemVerilog Code: Specifying User-Encoded States with the syn_encoding Attribute

```
(* syn_encoding = "user" *) reg [1:0] state;
parameter init = 0, last = 3, next = 1, later = 2;
always @ (state) begin
case (state)
init:
out = 2'b01;
next:
out = 2'b10;
later:
out = 2'b11;
last:
out = 2'b00;
endcase
end
```

Without the syn_encoding attribute, the Quartus II software encodes the state machine based on the current value of the **State Machine Processing** logic option.

If you also specify a safe state machine (as described in "Safe State Machine" on page 16–38), separate the encoding style value in the quotation marks from the safe value with a comma, as follows: "safe, one-hot" or "safe, gray".

For more information, refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 16–36.

Assigning a Pin

To assign a signal to a pin or device location, use the Tcl command shown in Example 16–153:

Example 16–153.

```
set_location_assignment -to <signal name> <location>
```

Valid locations are pin location names. Some device families also support edge and I/O bank locations. Edge locations are `EDGE_BOTTOM`, `EDGE_LEFT`, `EDGE_TOP`, and `EDGE_RIGHT`. I/O bank locations include `IOBANK_1` to `IOBANK_n`, where `n` is the number of I/O banks in a device.

Creating Design Partitions for Incremental Compilation

To create a partition, use the command shown in Example 16–154:

Example 16–154.

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The <*file name*> variable is the name used for internally generated netlist files during incremental compilation. If you create the partition in the Quartus II software, netlist files are named automatically by the Quartus II software based on the instance name. If you use Tcl to create your partitions, you must assign a custom file name that is unique across all partitions. For the top-level partition, the specified file name is ignored, and you can use any dummy value. To ensure the names are safe and platform independent, file names should be unique, regardless of case. For example, if a partition uses the file name `my_file`, no other partition can use the file name `MY_FILE`. To make file naming simple, Altera recommends that you base each file name on the corresponding instance name for the partition.

The <*destination*> is the short hierarchy path of the entity. A *short* hierarchy path is the full hierarchy path without the top-level name, for example: `"ram:ram_unit|altsyncram:altsyncram_component"` (with quotation marks). For the top-level partition, you can use the pipe (|) symbol to represent the top-level entity.

For more information about hierarchical naming conventions, refer to "Node-Naming Conventions in Quartus II Integrated Synthesis" on page 16–78.

The <*partition name*> is the partition name you designate, which should be unique and less than 1024 characters long. The name may only consist of alphanumeric characters, as well as pipe (|), colon (:), and underscore (_) characters. Altera recommends enclosing the name in double quotation marks (" ").

Quartus II Synthesis Options

For more information about the **.qsf** variable names and applicable values for the settings discussed in this chapter, refer to *Logic options* in Quartus II Help.

Conclusion

The Quartus II Integrated Synthesis supports Verilog HDL, SystemVerilog, VHDL, and Altera-specific languages, making the synthesis feature an easy-to-use, standalone solution for Altera designs. You can use the synthesis options in the software or in your HDL code to better control the way your design is synthesized, helping you improve your synthesis results. Use Quartus II reports and messages to analyze your compilation results.

Document Revision History

Table 16–6 lists the revision history for this document.

Table 16–6. Document Revision History (Part 1 of 3)

Date	Version	Changes
November 2013	13.1.0	■ Added a note regarding ROM inference using the `ram_init_file` in "RAM Initialization File—for Inferred Memory" on page 16–59.
May 2013	13.0.0	■ Added "Verilog HDL Configuration" on page 16–5. ■ Added "RAM Style Attribute—For Shift Registers Inference" on page 16–55. ■ Added "Upgrade IP Components Dialog Box" on page 16–73.
June 2012	12.0.0	■ Updated "Design Flow" on page 16–1.
November 2011	11.1.0	■ Updated "Language Support" on page 16–4, "Incremental Compilation" on page 16–21, "Quartus II Synthesis Options" on page 16–23.
May 2011	11.0.0	■ Updated "Specifying Pin Locations with chip_pin" on page 14–65, and "Shift Registers" on page 14–48. ■ Added a link to Quartus II Help in "SystemVerilog Support" on page 14–5. ■ Added Example 14–106 and Example 14–107 on page 14–67.
December 2010	10.1.0	■ Updated "Verilog HDL Support" on page 13–4 to include Verilog-2001 support. ■ Updated "VHDL-2008 Support" on page 13–9 to include the condition operator (explicit and implicit) support. ■ Rewrote "Limiting Resource Usage in Partitions" on page 13–32. ■ Added "Creating LogicLock Regions" on page 13–32 and "Using Assignments to Limit the Number of RAM and DSP Blocks" on page 13–33. ■ Updated "Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute" on page 13–55. ■ Updated "Auto Gated Clock Conversion" on page 13–28. ■ Added links to Quartus II Help.

Table 16–6. Document Revision History (Part 2 of 3)

Date	Version	Changes

July 2010	10.0.0	■ Removed Referenced Documents section.
		■ Added “Synthesis Seed” on page 9–36 section.
		■ Updated the following sections:
		  ■ “SystemVerilog Support” on page 9–5
		  ■ “VHDL-2008 Support” on page 9–10
		  ■ “Using Parameters/Generics” on page 9–16
		  ■ “Parallel Synthesis” on page 9–21
		  ■ “Limiting Resource Usage in Partitions” on page 9–32
		  ■ “Synthesis Effort” on page 9–35
		  ■ “Synthesis Attributes” on page 9–25
		  ■ “Synthesis Directives” on page 9–27
		  ■ “Auto Gated Clock Conversion” on page 9–29
		  ■ “State Machine Processing” on page 9–36
		  ■ “Multiply-Accumulators and Multiply-Adders” on page 9–50
		  ■ “Resource Aware RAM, ROM, and Shift-Register Inference” on page 9–52
		  ■ “RAM Style and ROM Style—for Inferred Memory” on page 9–53
		  ■ “Turning Off the Add Pass-Through Logic to Inferred RAMs no_rw_check Attribute” on page 9–55
		  ■ “Using altera_attribute to Set Quartus II Logic Options” on page 9–68
		  ■ “Adding an HDL File to a Project and Setting the HDL Version” on page 9–83
		  ■ “Creating Design Partitions for Incremental Compilation” on page 9–85
		  ■ “Inferring Multiplier, DSP, and Memory Functions from HDL Code” on page 9–50
		■ Updated Table 9–9 on page 9–86.

December 2009	9.1.1	■ Added information clarifying inheritance of Synthesis settings by lower-level entities, including Altera and third-party IP ■ Updated “Keep Combinational Node/Implement as Output of Logic Cell” on page 9–46

Table 16–6. Document Revision History (Part 3 of 3)

Date	Version	Changes
November 2009	9.1.0	■ Updated the following sections: ■ "Initial Constructs and Memory System Tasks" on page 9–7 ■ "VHDL Support" on page 9–9 ■ "Parallel Synthesis" on page 9–21 ■ "Synthesis Directives" on page 9–27 ■ "Timing-Driven Synthesis" on page 9–31 ■ "Safe State Machines" on page 9–40 ■ "RAM Style and ROM Style—for Inferred Memory" on page 9–53 ■ "Translate Off and On / Synthesis Off and On" on page 9–62 ■ "Read Comments as HDL" on page 9–63 ■ "Adding an HDL File to a Project and Setting the HDL Version" on page 9–81 ■ Removed "Remove Redundant Logic Cells" section ■ Added "Resource Aware RAM, ROM, and Shift-Register Inference" section ■ Updated Table 9–9 on page 9–83
March 2009	9.0.0	■ Updated Table 9–9. ■ Updated the following sections: ■ "Partitions for Preserving Hierarchical Boundaries" on page 9–20 ■ "Analysis & Synthesis Settings Page of the Settings Dialog Box" on page 9–24 ■ "Timing-Driven Synthesis" on page 9–30 ■ "Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting" on page 9–54 ■ Added "Parallel Synthesis" on page 9–21 ■ Chapter 9 was previously Chapter 8 in software version 8.1

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.