9. Quartus II Integrated Synthesis

QII51008-9.0.0

Introduction

As programmable logic designs become more complex and require increased performance, advanced synthesis has become an important part of the design flow. The Quartus® II software includes advanced integrated synthesis that fully supports VHDL and Verilog HDL, as well as 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 documents the design flow and language support in the Quartus II software. It explains how you can use incremental compilation to reduce your compilation time, how you can improve synthesis results with Quartus II synthesis options, and how you can control the inference of architecture-specific megafunctions. This chapter also explains some of the node-naming conventions used during synthesis to help you better understand your synthesized design and the messages issued during synthesis to improve your HDL code. Scripting techniques for applying all the options and settings described are also provided.

This chapter contains the following sections:

This chapter provides examples of how to use attributes described within the chapter, but does not cover specific coding examples.

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 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 and Synthesis process includes Quartus II integrated synthesis, which fully supports Verilog HDL and VHDL languages as well as Altera-specific languages, and supports major features in the SystemVerilog language (refer to "Language Support" on page 9–4 for details). This stage of the compilation flow performs logic synthesis to optimize design logic and performs technology mapping to implement the design logic using device resources, such as logic elements (LEs) or adaptive logic modules (ALMs) and other dedicated logic blocks. This stage also generates the single project database that integrates all the design files in a project (including any netlists from third-party synthesis tools).

You can use the Analysis and Synthesis stage of the Quartus II compilation to perform any of the following levels of Analysis and Synthesis:

- Analyze Current File—Parse the current design source file to check for syntax errors. This command does not report on many semantic errors that require further design synthesis. On the Processing menu, click **Analyze Current File**.
- Analysis & Elaboration—Check a design for syntax and semantic errors and perform elaboration to identify the design hierarchy. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
- Analysis & Synthesis—Perform complete Analysis and Synthesis on a design, including technology mapping. On the Processing menu, point to **Start** and click **Start Analysis & Synthesis**. This is the most commonly used command and is part of the full compilation flow.

The Quartus II design and compilation flow using Quartus II integrated synthesis 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. On the File menu, click **New Project Wizard**.
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 the design during synthesis and fitting. For synthesis settings, refer to "Quartus II Synthesis Options" on page 9–22. Add timing constraints to specify the timing requirements.
5. Compile the design in the Quartus II software. To synthesize the design, on the Processing menu, point to **Start**, and click **Start Analysis & Synthesis**.

   ☞ On the Processing menu, click **Start Compilation** to run a complete compilation flow including placement, routing, creation of a programming file, and timing analysis.

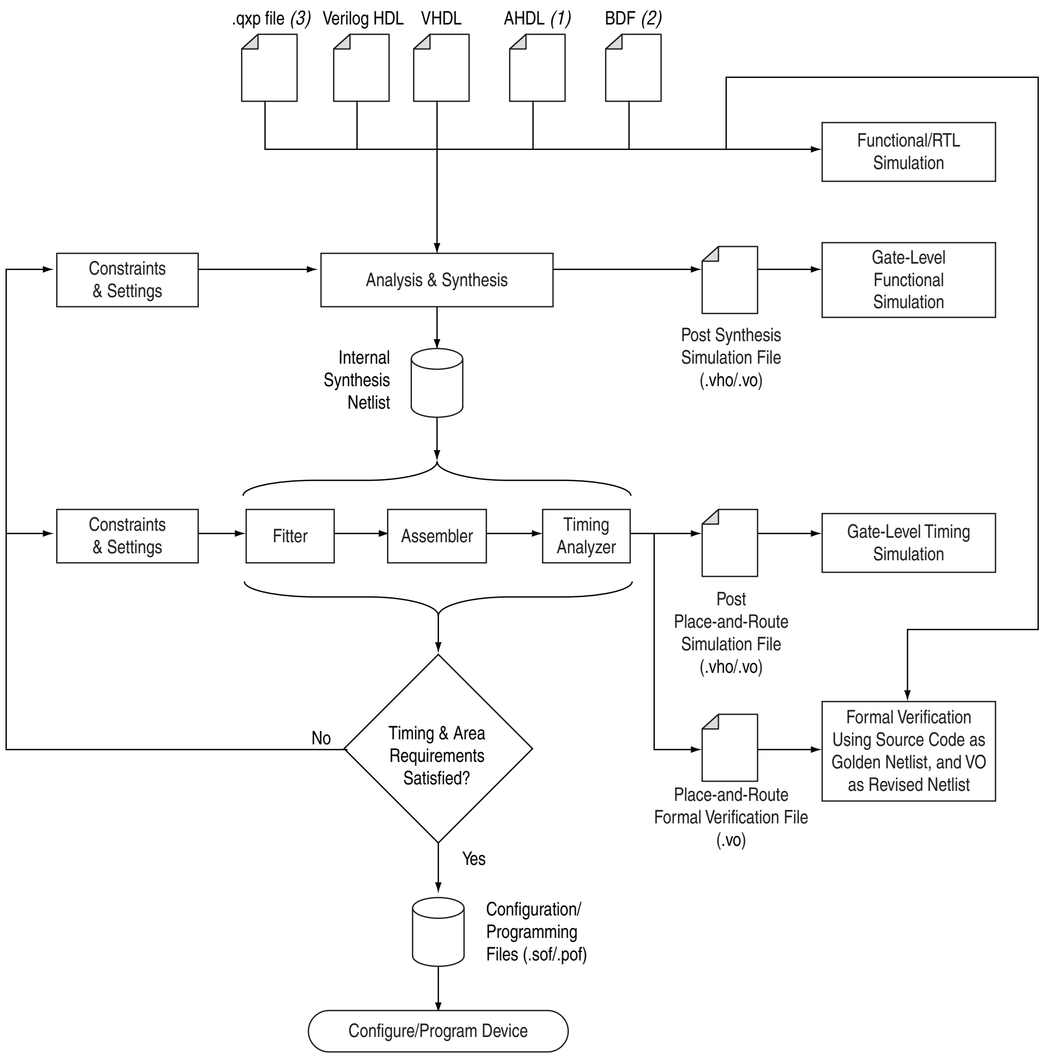
6. After obtaining synthesis and place-and-route results that meet your requirements, program or configure the Altera device.

The software provides netlists that allow you to perform functional simulation and gate-level timing simulation in the Quartus II simulator or a third-party simulator, perform timing analysis in a third-party timing analysis tool in addition to the TimeQuest Timing Analyzer or Classic Timing Analyzer, and/or perform formal verification in a third-party formal verification tool. The Quartus II software also provides many additional analysis and debugging features.

For more information about creating a project, compilation flow, and other features in the Quartus II software, refer to the Quartus II Help. For an overall summary of Quartus II features, refer to the *Introduction to the Quartus II Software* manual.

Figure 9–1 shows the basic design flow using Quartus II integrated synthesis.

Figure 9–1. Quartus II Design Flow Using Quartus II Integrated Synthesis


.qxp file (3)
Verilog HDL
VHDL
AHDL (1)
BDF (2)
Functional/RTL Simulation
Constraints & Settings
Analysis & Synthesis
Gate-Level Functional Simulation
Post Synthesis Simulation File (.vho/.vo)
Internal Synthesis Netlist
Constraints & Settings
Fitter
Assembler
Timing Analyzer
Gate-Level Timing Simulation
Post Place-and-Route Simulation File (.vho/.vo)
No
Timing & Area Requirements Satisfied?
Formal Verification Using Source Code as Golden Netlist, and VO as Revised Netlist
Place-and-Route Formal Verification File (.vo)
Yes
Configuration/ Programming Files (.sof/.pof)
Configure/Program Device

Notes to Figure 9–1:

(1) AHDL is the Altera Hardware Description Language.

(2) BDF is Altera's schematic Block Design File format (**.bdf**).

(3) The Quartus II eXported Partition (**.qxp**) file is a precompiled netlist that can be used as a design source file. For more information, refer to "Quartus II eXported Partition (.qxp) File as Source" on page 9–21.

Language Support

This section explains the Quartus II software's integrated synthesis support for HDL and schematic design entry, as well as graphical state machine entry, and explains how to specify the Verilog HDL or VHDL language version used in your design. It also documents language features such as Verilog HDL macros, initial constructs and memory system tasks, and VHDL libraries. "Design Libraries" on page 9–12 describes how to compile and reference design units in different custom libraries and "Using Parameters/Generics" on page 9–16 describes how to use parameters or generics and pass them between different languages.

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

Verilog HDL Support

The Quartus II compiler's Analysis and 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) (not all constructs are supported)

For complete information about specific Verilog HDL syntax features, and language constructs, refer to the Quartus II Help.

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**.

The Verilog HDL code samples provided in this document follow the Verilog-2001 standard unless otherwise specified.

To specify a default Verilog HDL version for all files, perform the following steps:

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

You can override the default Verilog HDL version for each Verilog HDL design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, select the appropriate file in the list and click the **Properties** button.

3. In the **HDL Version** list, select **SystemVerilog_2005**, **Verilog_2001**, or **Verilog_1995** and click **OK**.

You can also control the Verilog HDL version inside a design file using the `VERILOG_INPUT_VERSION` synthesis directive, as shown in Example 9–1. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 9–1. Controlling the Verilog HDL Input Version with a Synthesis Directive

```
// synthesis VERILOG_INPUT_VERSION <language version>
```

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

- `VERILOG_1995`
- `VERILOG_2001`
- `SYSTEMVERILOG_2005`

When the software reads a `VERILOG_INPUT_VERSION` synthesis directive, the current language version changes as specified until the end of the file, or until the next `VERILOG_INPUT_VERSION` directive is reached.

☞ You cannot change the language version in the middle of a Verilog HDL module.

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 9–27.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. Refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 9–80.

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 (relative to project root, user libraries, or current file location). When searching for a relative path, the Quartus II software initially searches relative to the project directory. If the software cannot find the file, it then searches relative to all user libraries, and finally relative to the directory location of the current file.

Verilog-2001 Support

The Quartus II software does not support Verilog-2001 libraries and configurations.

SystemVerilog Support

The Quartus II software supports the following SystemVerilog constructs:

- Parameterized interfaces, generic interfaces, and `modport` constructs
- Packages
- `Extern` module declarations
- Built-in data types `logic`, `bit`, `byte`, `shortint`, `longint`, `int`

- Unsized integer literals `'0`, `'1`, `'x`, `'z`, `'X`, and `'Z`
- Structure data types using `struct`
- Ports and parameters with unrestricted data types
- User-defined types using `typedef`
- Global declarations of task/functions/parameters/types (does not support global variables)
- Coding constructs `always_comb`, `always_latch`, `always_ff`
- Continuous assignments to nodes other than `nets`, and procedural assignments to nodes other than `reg`
- Enumeration methods `First`, `Last`, `Next(n)`, `Prev(n)`, `Num`, and `Name`
- Assignment operators +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, <<<=, and >>>=
- Increment ++ and decrement --
- Jump statements `return`, `break`, and `continue`
- Enhanced `for` loop (declare loop variables inside initial condition)
- `Do-while` loop and local loop constructs
- Assignment patterns
- Keywords `unique` and `priority` in case statements
- Default values for function/task arguments
- Closing labels
- Extensions to directives `'define` and `'include`
- Expression size system function `$bits`
- Array query system functions `$dimensions, $unpacked_dimensions`, `$left`, `$right`, `$high`, `$low`, `$increment`, `$size`
- Packed array (include multidimensional packed array)
- Unpacked array (include single-valued range dimension)
- Implicit port connections with **.name** and **.***

Quartus II integrated synthesis also parses, but otherwise ignores, SystemVerilog assertions.

☞ Designs written to comply with the Verilog-2001 standard might not compile successfully using the SystemVerilog setting because the SystemVerilog standard adds a number of new reserved keywords. For a list of reserved words in each language standard, refer to the Quartus II Help.

Initial Constructs and Memory System Tasks

The Quartus II software infers power-up conditions from Verilog HDL `initial` constructs. The software creates power-up settings for variables, including RAM blocks. If the Quartus II software encounters non-synthesizable constructs in an `initial` block, it generates an error. To avoid such errors, enclose non-synthesizable 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 9–62. Synthesis of initial constructs 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.

☞ Initial blocks do not infer power-up conditions in some third-party EDA synthesis tools. If you are converting between synthesis tools, ensure that your power-up conditions are set correctly.

Quartus II integrated synthesis supports the `$readmemb` and `$readmemh` system tasks to initialize memories. Example 9–2 shows an initial construct that initializes an inferred RAM with `$readmemb`.

Example 9–2. 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, then specify the memory word such as `110101` or `abcde` on the next line. Example 9–3shows a portion of a memory initialization file for the RAM in Example 9–2.

Example 9–3. 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 GUI or on the command line.

Setting a Verilog HDL Macro Default Value in the GUI

To specify a macro in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Verilog HDL Input**. Under **Verilog HDL macro**, type the macro name in the **Name** box, the value in the **Setting** box, and 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 9–4.

Example 9–4. Command Syntax for Specifying a Verilog HDL Macro

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

The command in Example 9–5 has the same effect as specifying `` `define a 2`` in the Verilog HDL source code.

Example 9–5. 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 9–6.

Example 9–6. 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 and Synthesis module supports the following VHDL standards:

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

For information about specific VHDL syntax features and language constructs, refer to the Quartus II Help.

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 document follow the VHDL 1993 standard.

To specify a default VHDL version for all files, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
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, then click **OK**.

You can override the default VHDL version for each VHDL design file by performing the following steps:

1. On the Project menu, click **Add/Remove Files in Project**. The **Settings** dialog box appears.
2. On the **Files** page, select the appropriate file in the list and click **Properties**.
3. In the HDL version list, select **VHDL93** or **VHDL87** and click **OK**.

You can also specify the VHDL version for each design file using the `VHDL_INPUT_VERSION` synthesis directive, as shown in Example 9–7. This directive overrides the default HDL version and any HDL version specified in the **File Properties** dialog box.

Example 9–7. Controlling the VHDL Input Version with a Synthesis Directive

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

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

- `VHDL87`
- `VHDL93`

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

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

For more information about specifying synthesis directives, refer to "Synthesis Directives" on page 9–27.

If you use scripts to add design files, you can use the `-HDL_VERSION` command to specify the HDL version for each design file. Refer to "Adding an HDL File to a Project and Setting the HDL Version" on page 9–80.

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.

VHDL Standard Libraries and Packages

The Quartus II software includes the standard IEEE libraries and a number of vendor-specific VHDL libraries. For information about organizing your own design units into custom libraries, refer to "Design Libraries" on page 9–12.

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.

For a complete listing of library and package support, refer to the Quartus II Help.

Beginning with the Quartus II software version 5.1, you should 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 only a single VHDL `wait until` statement in a process block. Other VHDL wait constructs, such as `wait for`, or `wait on` statements, or processes with multiple `wait` statements, are not synthesizable.

Example 9–8 is a VHDL code example of a supported `wait until` construct.

Example 9–8. 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;
```

Example 9–9 is a VHDL code example of unsupported `wait for` construct. The process block a with `wait for`, or `wait on` statement is not synthesizable.

Example 9–9. VHDL Code: Unsupported wait for Construct

```
process
begin
CLK <= '0';
wait for 20 ns;
CLK <= '1';
wait for 12 ns;
end process;
```

The process block with multiple `wait until` statements is not synthesizable. Example 9–10 shows an example of multiple `wait until` statements in a process block.

Example 9–10. Multiple wait until Statements in a Process Block

```
output: process begin
wait until (CLK'event and CLK='1');
Q <= D;
Qbar <= not D;

wait until (CLK'event and CLK='0');
Q <= 0;
Qbar <= 1;
end process output;
```

AHDL Support

The Quartus II compiler's Analysis and 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** file with an AHDL `include` statement. Altera provides **.inc** files for all megafunctions shipped with the Quartus II software.

For information about specific AHDL syntax features and language constructs, refer to the Quartus II Help.

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

Schematic Design Entry Support

The Quartus II compiler's Analysis and Synthesis module fully supports Block Design Files (**.bdf**) for schematic design entry.

You can use the Quartus II software's Block Editor to create and edit **.bdf** files and open Graphic Design Files (**.gdf**) imported from the MAX+PLUS II software. Use the Symbol Editor to create and edit Block Symbol Files (**.bsf**) and open MAX+PLUS II Symbol Files (**.sym**). You can read and edit these legacy MAX+PLUS II formats with the Quartus II Block and Symbol Editors; however, the Quartus II software saves them as **.bdf** or **.bsf** files.

For information about creating and editing schematic designs, refer to the Quartus II Help.

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

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 choose **State Machine File**.

In the editor, you can use the State Machine Wizard to step you through the state machine creation. Click the **State Machine Wizard** icon. Specify the reset information, define the input ports, states, and transitions, and then define the output ports and output conditions. Click **Finish** to create the state machine diagram.

You can also create the state machine diagram using the editor GUI. Use the icons or right-click menu options to insert new input and output signals and create states in the schematic display. To specify transitions, select the **Transition Tool** and click on the source state, then drag the mouse to the destination state. Double-click on a transition to specify the transition equation, using a syntax that conforms to Verilog HDL. Double-click on a state to open the **State Properties** dialog box, where you can change the state name, specify whether it acts as the reset state, and change the incoming and outgoing transition equations.

To view and edit state machine information in a table format, click the **State Machine Table** icon.

The state machine diagram is saved as a State Machine File (**.smf**). When you have finished defining the state machine logic, create a Verilog HDL or VHDL design file by clicking the **Generate HDL File** icon. You can then instantiate the state machine in your design using any design entry language.

For more information about creating and editing state machine diagrams, refer to the 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, or if a file refers to a library that does not exist, or if the library does not contain a referenced design unit, the software searches the work library. This behavior allows the Quartus II software to compile most designs with minimal setup. (Creating separate custom design libraries is optional.)

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"
- "Specifying a Destination Library Name in the Quartus II Settings File or Using Tcl"

When the Quartus II compiler analyzes the file, it stores the analyzed design units in the file's destination library.

Beginning with the Quartus II software version 6.1, a design can contain two or more entities with the same name if they are compiled 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 no other library is specified). If the entity definition is not found, the software searches for a unique entity definition in all design libraries. If more than one entity with the same name is found, 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, there are several ways to associate an instance with a particular entity, as described in "Mapping a VHDL Instance to an Entity in a Specific Library". In Verilog HDL, BDF schematic entry, AHDL, as well as VQM and EDIF netlists, 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, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Files**. The **Files** page appears.
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 desired library name in the **Library** field.
7. Click **OK**.

Specifying a Destination Library Name in the Quartus II Settings File or Using 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 **.qsf** file or Tcl assignments specify that the Quartus II software analyze **my_file.vhd** and store its contents (design units) in the VHDL library **my_lib**, and analyze the Verilog HDL file **my_header_file.h** and store its contents in a library called **another_lib**. Refer to Example 9–11.

Example 9–11. 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 9–80.

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 a single string argument: the name of the destination library. Specify the `library` directive in a VHDL comment prior to the context clause for a primary design unit (that is, a package declaration, an entity declaration, or a configuration), using 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 9–27.

The `library` directive overrides the default library destination **work**, the library setting specified for the current file through the **Settings** dialog box, any existing **.qsf** file 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 9–12 uses the `library` synthesis directive to create a library called **my_lib** that contains the design unit `my_entity`.

Example 9–12. 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 the design units in a given source file by specifying the library name in the **Settings** dialog box, editing the **.qsf** file, or using the Tcl interface. Using the `library` directive to change the destination VHDL library within a source file gives you the option of organizing the design units in a single file into different libraries, rather than just a single library.

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

Mapping a VHDL Instance to an Entity in a Specific Library

The VHDL language provides a number of 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 9–13.

Example 9–13. 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

There is more than one mechanism for binding a component to an entity. In an explicit binding indication, you bind a component instance to a specific entity, as shown in Example 9–14.

Example 9–14. 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, a component instance is bound to the nearest visible entity with the same name. If no such entity is visible in the current scope, the instance is bound to the entity in the library in which the component was declared. For example, if the component is declared in a package in library `MY_LIB`, an instance of the component is bound to the entity in library `MY_LIB`. The portions of code in Example 9–15 and Example 9–16 show this instantiation method.

Example 9–15. 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 9–16. 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 parameters (called generics in VHDL) are supported in the Quartus II software, and how you can pass these parameters between different design languages.

You can enter default parameter values for your design in the **Default Parameters** box in the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Default parameters allow you to specify the parameter overrides for your top-level entity. In AHDL, parameters are inherited, so any default parameters apply to all AHDL instances in the design. You can also specify parameters for instantiated modules in a **.bdf** file. To modify parameters in a **.bdf** instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab. For these GUI-based entry methods, information about how parameter values are interpreted, and recommendations about the format you should use, refer to “Setting Default Parameter Values and BDF Instance Parameter Values”.

You can specify parameters for instantiated modules in your design source files, using the syntax provided for that 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. In most cases, you do not have to do anything special to pass parameters from one language to another. However, in some cases you might have to specify the type of parameter you are passing. In those cases, you should follow certain guidelines to ensure that the parameter value is interpreted correctly. Refer to “Passing Parameters Between Two Design Languages” on page 9–18 for parameter type rules.

Setting Default Parameter Values and BDF Instance Parameter Values

Default parameter values and BDF instance parameter values do not have an explicitly declared type. In most cases, the Quartus II software can correctly infer the type from the value without ambiguity. For example, “`ABC`” is interpreted 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 in the instantiated entity to determine how to interpret the value, so that a value of `123` is interpreted as a string if the VHDL parameter is of 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 where the first or first and second character 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 simply 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 interpret the parameter value correctly. Table 9–1 provides a list of valid parameter strings and shows how they are interpreted within the Quartus II software. Altera recommends using the type-encoded format only when necessary to resolve ambiguity.

Table 9–1. Valid Parameter Strings and How They are Interpreted

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"`	Enum type, value name is apple
`P"1 unit"`	Physical literal, the value is (1, unit)
`A(...)`, `a(...)`	Array type or record type, whose content is determined by the string (...)

Beginning in Quartus II software version 8.1, you can select the parameter type using the pull-down list in the **Parameter** tab of the **Symbol Properties** dialog box. You can select the parameter types for global parameters or global constants. 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**

If you do not specify the parameter type, the Quartus II software interprets the parameter value and defines the parameter type. Altera recommends specifying parameter type by selecting from the pull-down list to avoid ambiguity.

☞ If you open a **.bdf** file in the Quartus II software version 8.1 and above, the software automatically updates the parameter types of old symbol blocks by interpreting the parameter value based on the language-independent format. If the parameter value type is not recognized, the parameter type is set as untyped. You can specify the parameter type as described in the preceding list.

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. It is essential that the parameter be correctly interpreted by the subdesign language (the design entity that is instantiated). 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), it is essential that the enumeration literal is spelled correctly in the higher-level design. The parameter value is passed as a string literal, and it is up to the language of the lower-level design to correctly convert the string literal into the correct enumeration literal.

If the lower-level language is SystemVerilog, it is essential that the `enum` value is used in the correct case. In SystemVerilog, it is recommended that two enumeration literals do not only differ in case. For example, `enum {item, ITEM}` is not a good choice of item names because these names can create confusion among users and it 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 9–17 shows a VHDL subdesign that is instantiated into a top-level Verilog HDL design in Example 9–18. Example 9–19 shows a Verilog HDL subdesign that is instantiated in a top-level VHDL design in Example 9–20.

Example 9–17. 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 9–18. Verilog HDL Top-Level Design Instantiating and Passing Parameters to VHDL Entity from Example 9–17

```
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 9–19. 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;
```

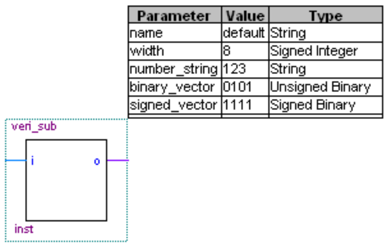
Example 9–20. VHDL Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–19

```
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 9–19 in a top-level BDF design, you must first generate a symbol for the HDL file, as shown in Figure 9–2. Open the HDL file in the Quartus II software, and then, on the File menu, point to **Create/Update** and click **Create Symbol Files for Current File**.

To modify parameters on a BDF instance, double-click on the parameter value box for the instance symbol, or right-click on the symbol and choose **Properties**, then click the **Parameters** tab.

Figure 9–2. BDF Top-Level Design Instantiating and Passing Parameters to the Verilog HDL Module from Example 9–19.



Parameter	Value	Type
name	default	String
width	8	Signed Integer
number_string	123	String
binary_vector	0101	Unsigned Binary
signed_vector	1111	Signed Binary

Incremental Compilation

The incremental compilation feature in the Quartus II software manages a design hierarchy for incremental design by allowing you to divide the design into multiple partitions. Incremental compilation ensures that when a design is compiled, only those partitions of the design that have been updated are resynthesized, reducing compilation time and runtime memory usage. This also means that node names are maintained 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 optionally 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.

Partitions for Preserving Hierarchical Boundaries

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

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

Beginning with Quartus II software version 9.0, if you want to preserve the **Optimization Technique** and **Restructure Multiplexers** logic options set in any entity, you must create new partitions for the particular 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. Similarly, if you are performing formal verification, you must use partitions with incremental compilation to ensure that no optimizations occur across specific design hierarchies.

Parallel Synthesis

The **Parallel Synthesis** option is one of the Analysis and Synthesis options that you can use to reduce compilation time for synthesis. The feature enables the Quartus II software to use multi-processors to synthesize multiple partitions in parallel.

This feature is available only if the following requirements are met:

- The number of processors allowed is greater than 1

  You can specify the maximum number of processors allowed under **Parallel Compilation** options in the **Compilation Process Settings** page of the **Settings** dialog box.

- Incremental compilation is enabled and your design has two or more partitions
- **Timing Driven Synthesis** is not enabled
- **Physical Synthesis** is not enabled
- **Parallel Synthesis** is enabled using one of the following procedures:

To enable parallel synthesis, perform the following steps:

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

You can also set the **Parallel Synthesis** option using the following Tcl command:

```
set_global_assignment -name parallel_synthesis on
```

You can view all messages generated during the parallel synthesis in the Message console. Messages from different partitions are interleaved at runtime, but the **Partition Column** displays the partition ID of the partition that has a message. After compilation, you can sort the messages by **Partition Column**—effectively grouping all the messages from a particular partition. To display the partition column, right click on the message console, point to **Message Column** and select **Show Partition Column**. You can also display the **Partition** column on the Tools menu, by clicking **Options** and selecting **Messages** in the **Category** list. In the Messages page, turn on **Show the Partition column**.

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 **Messages** page. This option shows the partition ID in parenthesis for each message.

Quartus II eXported Partition (.qxp) File as Source

You can use a Quartus II eXported Partition (**.qxp**) file as a source file beginning with Quartus II software version 8.1. The **.qxp** file contains the precompiled design netlist exported from another Quartus II project, or from a design partition within the project, and fully defines the entity. Project team members or IP providers can use a **.qxp** file to send their design to the project lead, instead of sending the original HDL source code. Using this file preserves the previous compilation results and instance-specific assignments. Not all global assignments can be used in a different Quartus II project. You can override the assignments for the entity in the **.qxp** file by applying assignments in the full top-level project.

A **.qxp** file instance that is not assigned as a design partition does not preserve placement and routing results. If you want to preserve the placement (and optionally routing) results from another project or compilation, you must import a post-fitting **.qxp** file into a design partition in your project using the bottom-up incremental compilation flow.

Perform the following steps to create a **.qxp** file:

1. On the Project menu, click **Export Design Partition**.
2. In the **Export file** box, type the name of the **.qxp** file. By default, the directory path and file name are the same as the current project.
3. You can also select the Partition hierarchy to export. By default, the Top partition (the entire project) is exported, but you can choose to export the compilation results of any partition hierarchy in the project.
4. Under **Netlist to export**, select either **Post-fit netlist** or **Post-synthesis netlist**. The default is **Post-fit netlist**. For post-fit netlists, turn on or off the **Export routing** option as required.
5. Click **OK**. The Quartus II software creates the **.qxp** file in the specified directory.

The Quartus II software adds the file into the project and **.qxp** file into a specific library. The design entity in the **.qxp** file can also be instantiated multiple times in the design.

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 a number of options to help you control the synthesis process and achieve optimal results for your design. "Setting Synthesis Options" on page 9–24 describes the **Analysis & Synthesis Settings** page of the **Settings** dialog box, where you can set the most common global settings and options, and defines the following three types of synthesis options: Quartus II logic options, synthesis attributes, and synthesis directives. The other subsections describe the following common synthesis options in the Quartus II software, and provide HDL examples of how to use each option, where applicable:

- Major Optimization Settings
    - "Optimization Technique" on page 9–28
    - "Speed Optimization Technique for Clock Domains" on page 9–28
    - "PowerPlay Power Optimization" on page 9–31
    - "Restructure Multiplexers" on page 9–34
    - "Synthesis Effort" on page 9–35

- Settings Related to Timing Constraints
  - "Optimization Technique" on page 9–28
  - "Speed Optimization Technique for Clock Domains" on page 9–28
  - "Auto Gated Clock Conversion" on page 9–29
  - "Timing-Driven Synthesis" on page 9–30
  - "SDC Constraint Protection" on page 9–31
- State Machine Settings and Enumerated Types
  - "State Machine Processing" on page 9–36
  - "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 9–37
  - "Manually Specifying Enumerated Types Using the enum_encoding Attribute" on page 9–39
  - "Safe State Machines" on page 9–41
- Register Power-Up Settings
  - "Power-Up Level" on page 9–42
  - "Power-Up Don't Care" on page 9–43
- Controlling, Preserving, Removing, and Duplicating Logic and Registers
  - "Limiting DSP Block Usage in Partitions" on page 9–32
  - "Remove Duplicate Registers" on page 9–44
  - "Remove Redundant Logic Cells" on page 9–44
  - "Preserve Registers" on page 9–44
  - "Disable Register Merging/Don't Merge Register" on page 9–45
  - "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–46
  - "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46
  - "Don't Retime, Disabling Synthesis Netlist Optimizations" on page 9–47
  - "Don't Replicate, Disabling Synthesis Netlist Optimizations" on page 9–48
  - "Maximum Fan-Out" on page 9–49
  - "Controlling Clock Enable Signals with Auto Clock Enable Replacement and direct_enable" on page 9–50
  - "Auto Gated Clock Conversion" on page 9–29
  - To preserve design hierarchy, refer to "Partitions for Preserving Hierarchical Boundaries" on page 9–20

- Megafunction Inference Options
  - "Megafunction Inference Control" on page 9–51
  - "RAM Style and ROM Style—for Inferred Memory" on page 9–53
  - "Turning Off Add Pass-Through Logic to Inferred RAMs/ no_rw_check Attribute Setting" on page 9–54
  - "RAM Initialization File—for Inferred Memory" on page 9–57
  - "Multiplier Style—for Inferred Multipliers" on page 9–58
- Controlling Synthesis with Other Synthesis Directives
  - "Full Case" on page 9–60
  - "Parallel Case" on page 9–61
  - "Translate Off and On / Synthesis Off and On" on page 9–62
  - "Ignore translate_off and synthesis_off Directives" on page 9–63
  - "Read Comments as HDL" on page 9–63
- Specifying I/O-Related Assignments
  - "Use I/O Flipflops" on page 9–64
  - "Specifying Pin Locations with chip_pin" on page 9–65
- Setting Quartus II Logic Options in Your HDL Source Code
  - "Using altera_attribute to Set Quartus II Logic Options" on page 9–66

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 within the HDL source code.

Analysis & Synthesis Settings Page of the Settings Dialog Box

On the Assignments menu, click **Settings**. The **Settings** dialog box appears. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page allows you to set global synthesis options that apply to the entire project. These options are described in later subsections.

Beginning with Quartus II software version 9.0, some of the advanced synthesis settings can be set in the **Physical Synthesis Optimization** 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

Quartus II logic options control many aspects of the synthesis and place-and-route process. To set logic options in the Quartus II GUI, on the Assignments menu, click **Assignment Editor**. You can also use a corresponding Tcl command. Quartus II logic options allow you to set instance or node-specific assignments without editing the source HDL code. Logic options can be used with all design entry languages supported by the Quartus II software.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*.

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 in a particular manner. Attributes always apply to a specific design element, and are applied in the HDL source code. Some synthesis attributes are also available as Quartus II logic options via the Quartus II GUI or with Tcl. Each attribute description in this chapter indicates whether there is a corresponding setting or logic option that can be set in the GUI; some attributes can be specified only with HDL synthesis attributes.

Attributes specified in your HDL code are not visible in the Assignment Editor or in the **.qsf** file. Assignments or settings made through the Quartus II GUI, the **.qsf** file, 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 invalid attributes are found, but does not generate an error or stop the compilation. This behavior is required because attributes are specific to various design tools, and attributes not recognized in the Quartus II software might be intended for a different EDA tool. The Quartus II software lists the attributes specified in your HDL code in the Source assignments table in 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 9–21 through Example 9–24, where <*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 are also case-sensitive.

Example 9–21. Synthesis Attributes in Verilog-1995

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

Verilog-1995 comment-embedded attributes, as shown in Example 9–21, must be used as a suffix to (that is, placed after) the declaration of an item and must appear before the semicolon when one is required.

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

To apply multiple attributes to the same instance in Verilog-1995, separate the attributes with spaces, as follows:

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

For example, to set the `maxfan` attribute to `16` (Refer to "Maximum Fan-Out" on page 9–49 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 9–44 for details) on a register called `my_reg`, use the following syntax:

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

In addition to the `synthesis` keyword shown above, the keywords `pragma`, `synopsys`, and `exemplar` are supported for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis attributes that will be recognized only by Quartus II integrated synthesis 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.

Example 9–22. Synthesis Attributes in Verilog-2001 and SystemVerilog

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

Verilog-2001 attributes, as shown in Example 9–22, must be used as a prefix to (that is, placed before) a declaration, module item, statement, or port connection, and used as a suffix to (that is, placed after) an operator or a Verilog HDL function name in an expression.

☞ Because formal verification tools do not recognize the syntax, the Verilog-2001 attribute syntax is not supported when using formal verification.

To apply multiple attributes to the same instance in Verilog-2001 or SystemVerilog, separate the attributes with commas, as shown in Example 9–23:

Example 9–23. Applying Multiple Attributes

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

For example, to set the `maxfan` attribute to `16` (refer to "Maximum Fan-Out" on page 9–49 for details) and set the `preserve` attribute (refer to "Preserve Registers" on page 9–44 for details) on a register called `my_reg`, use the following syntax:

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

Example 9–24. Synthesis Attributes in VHDL

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

VHDL attributes, as shown in Example 9–24, declare the attribute type and then apply it to a specific object. For VHDL designs, all supported synthesis attributes are declared 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 follows:

```
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 in a particular manner. Directives do not apply to a specific design node but change the behavior of the synthesis tool from the point where 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 shown in Example 9–25 and Example 9–26, where <*directive*> and <*value*> are variables, and the entry in brackets is optional. Notice that for synthesis directives there is no = sign before the value; this is different than the 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 9–25. Verilog HDL Code: Synthesis Directives

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

Example 9–26. VHDL Code: Synthesis Directives

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

In addition to the `synthesis` keyword shown above, the `pragma`, `synopsys`, and `exemplar` keywords are supported in both Verilog HDL and VHDL for compatibility with other synthesis tools. The keyword `altera` is also supported, which allows you to add synthesis directives that are recognized only by Quartus II integrated synthesis and not by other tools that recognize the same synthesis directives.

Because formal verification tools ignore keywords `exemplar`, `pragma`, and `altera`, avoid using these directive keywords when you are using 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. Table 9–2 lists the settings for this logic option, which you can apply only to a design entity. You can also set this logic option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box. If you want to set this logic option for an entity, you must create a design partition for the entity before setting the **Optimization Technique** logic option. Beginning in Quartus II version 9.0, this option is ignored when set on an entity that is not a design partition.

Table 9–2. Optimization Technique Settings

Setting	Description
Area	The compiler makes the design as small as possible to minimize resource usage.
Speed	The compiler chooses a design implementation that has the fastest f_{MAX}.
Balanced *(1)*	The compiler maps part of the design for area and part for speed, providing better area utilization than optimizing for speed, with only a slightly slower f_{MAX} than optimizing for speed.

Note to Table 9–2:

(1) The balanced optimization technique is not supported for all device families.

The default setting varies by device family and is generally optimized for the best area/speed trade-off. Results are design-dependent and vary depending on which device family you use.

Speed Optimization Technique for Clock Domains

The **Speed Optimization Technique for Clock Domains** logic option specifies that all combinational logic in or between the specified clock domain(s) is optimized for speed.

When this option is set on a particular clock signal, all the logic in this clock domain is optimized for speed during synthesis. The remainder of the design in other clock domains is synthesized with the project-wide **Optimization Technique** that is set in the **Analysis & Synthesis Settings** page. The option can also be set from one clock to another clock signal, in which case the logic in paths from registers in the first clock domain to registers in the second clock domain are synthesized for speed. The advantage of using this option over the project-wide setting to optimize for speed is that there is less penalty to the area of the design because a smaller part of the circuit is optimized for speed. This can also have a positive effect on clock speed. This option also has an advantage over setting the **Optimization Technique** on a design entity because that option forces the hierarchical blocks to be synthesized separately. Doing so can increase area and decrease performance due to the lack of optimizations across hierarchies. The **Speed Optimization Technique for Clock Domains** option does not treat hierarchical entities separately, and can optimize across hierarchical boundaries for logic within the same clock domain.

This option is useful if you have one or more clock domains that do not meet your timing requirements. When there are failing paths within a clock domain, the option can be set on the clock of that clock domain. When there are failing paths between clock domains, the option can be set from one clock domain to the other clock domain.

This option is available for the following device families: Arria® GX, Stratix® series, Cyclone® series, HardCopy® II, HardCopy Stratix, and MAX® II.

Auto Gated Clock Conversion

Clock gating is a common optimization technique used in ASIC designs to minimize power consumption. You can use the **Auto Gated Clock Conversion** 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. However, this feature should not be used when migrating FPGA designs to HardCopy ASICs. The **Auto Gated Clock Conversion** option automatically converts qualified gated clocks (base clocks as defined in the SDC assignments) to clock enables. To use **Auto Gated Clock Conversion**, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **Auto Gated Clock Conversion** and in the **Setting** list, select **On**.
5. Click **OK**.
6. Click **OK** again.

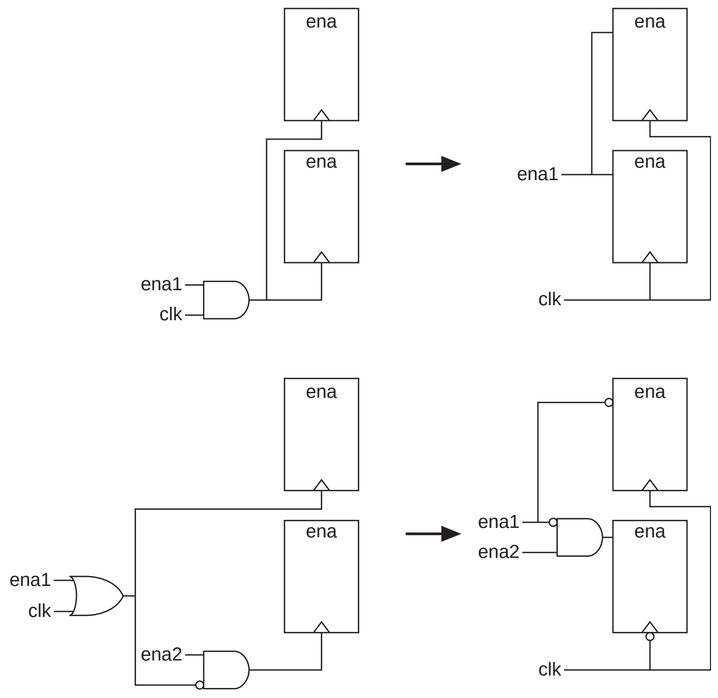
This feature is available only for the TimeQuest Timing Analyzer and supports the following device families: Arria II GX, Arria GX, Stratix series (except for Stratix) and Cyclone series (except for Cyclone), HardCopy II, and MAX II devices.

The gated clock conversion occurs when the following 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 feature supports combinational gates in clock gating network. Figure 9–3 shows examples of gated clock conversions.

Figure 9–3. Gated Clock Conversion



This feature does not support registers in RAM, DSP blocks, or I/O related WYSIWYG. The gated clock conversion does not support multiple design partitions from incremental compilation where the gated clock and base clock are not in the same hierarchical partition because the gated-clock conversion cannot trace the base clock from the gated clock. Thus, base clocks and gated clocks must be in the same hierarchical design partition. If a gated clock that is derived from a root gated clock of a multiple cascaded gated clock cannot be converted, the whole gated clock tree will not be converted, because each conversion is based on a gated clock tree instead of every gated clock.

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

Timing-Driven Synthesis

The **Timing-Driven Synthesis** option specifies whether synthesis should use the design's SDC timing constraints to better optimize the circuit. This feature enables synthesis to take into account the SDC timing constraints to focus on the truly critical parts of the design when optimizing for performance. Altera recommends using **Timing-Driven Synthesis** with **Optimization Technique Balanced**. When you turn on **Timing-Driven Synthesis**, Synthesis improves logic depth on parts of the design that require meeting performance requirements at the cost of area increase. Compared to **Optimization Technique** option for speed, **Timing-Driven Synthesis** gets similar performance but saves area.

To use the **Timing-Driven Synthesis** option, perform the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. In the **Analysis & Synthesis Settings** page, select **Timing-Driven Synthesis**.

The feature is available only for the TimeQuest Timing Analyzer and supports Arria II GX, Arria GX, Stratix series (except Stratix devices) and Cyclone series (except Cyclone devices), and HardCopy II devices. Altera recommends that you select a specific device for timing-driven synthesis to have the most accurate timing information. When auto device is selected, timing-driven synthesis uses the smallest device for the selected family to obtain timing information.

SDC Constraint Protection

The **SDC constraint protection** option allows you to preserve timing constraints when you use the TimeQuest Timing Analyzer. The feature checks on register merging and retiming. It prevents register merging on registers with incompatible SDC constraints and prevents register retiming on constrained registers. It helps maintain the validity of SDC constraints throughout compilation. To use the **SDC constraint protection** option, perform the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **SDC constraint protection** and in the **Setting** list, select **On**.

This feature is available only for the TimeQuest Timing Analyzer and supports the following device families: Arria GX, Stratix series (except Stratix devices), Cyclone series (except Cyclone devices), HardCopy II, and MAX II devices.

PowerPlay Power Optimization

This logic option controls the power-driven compilation setting of Analysis and Synthesis and determines how aggressively Analysis and Synthesis optimizes the design for power. On the Assignments menu, click **Settings**. In the **Category** list, select **Analysis & Synthesis Settings**. This displays the **Analysis & Synthesis Settings** page. The following three settings are available for the **PowerPlay Power Optimization** option:

- **Off**—Analysis and Synthesis does not perform any power optimizations.
- **Normal Compilation**—Analysis and Synthesis performs power optimizations, without reducing design performance.
- **Extra Effort**—Analysis and Synthesis performs additional power optimizations, which can reduce design performance.

This logic option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.

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 DSP Block Usage in Partitions

One important step of Analysis and Synthesis is resource balancing. In this step, Quartus II integrated synthesis looks at the digital signal processing (DSP) block used in the design and balances it against the resources available in the targeted device that are converting the DSP blocks that cannot fit in the device into equivalent logic. For incremental compilation, each partition has a separate balancing step.

By default, the Quartus II integrated synthesis looks at the targeted device information to find out the number of DSP blocks available for use. However, in incremental compilation, each partition looks at the device information independently and consequently assumes that it has all the DSP blocks in the device available for use. This can result in over-allocation of DSP blocks in the design, which means that the total number of DSP blocks used by all the partitions is greater than the number of DSP blocks available in the device. This can eventually lead to a no-fit error during the fitting process.

To avoid this, Altera recommends that you set the **Maximum DSP Block Usage** assignment on each partition to manually limit the number of DSP blocks used. You can set this assignment on a partition using the Assignment Editor by selecting the **Maximum DSP Block Usage** assignment, and setting it on the root of a partition. Set any positive integer as the value of this assignment. If this assignment is set on a name other than a partition root, the Quartus II integrated synthesis gives an error.

The **Maximum DSP Block Usage** assignment is available only for supported device families. Refer to the Quartus II Help for a list of the devices.

For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook.*

You can also set this assignment globally as a project-wide option by performing the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. From the pull-down menu, point to **Maximum DSP Block Usage**, and from the **Settings** pull-down menu, select your desired value.

The partition-specific assignment overrides the global assignment, if any. However, each partition that does not have a partition-specific **Maximum DSP Block Usage** assignment limits the number of the DSP blocks to the value set by the global assignment. This can also lead to over-allocation of DSP blocks. Therefore, Altera recommends that you always set this assignment on each partition when you use the incremental compilation.

Manually limiting the DSP blocks usage is also useful for HardCopy II device migration, where the number of DSP blocks that can be implemented in a HardCopy II device is more than the number of DSP blocks that can be implemented in its equivalent Stratix II device.

In Quartus II software version 8.1 and later, the floorplan aware synthesis feature enables you to use LogicLock regions to define resource allocation for DSPs and RAMs before setting the maximum resource allocation assignment.

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*, or refer to the Quartus II Help.

Altera recommends that you always use LogicLock assignments first before setting the maximum resource allocation assignments per partition. However, this recommendation might not affect the resource balancing if you manually assign nodes in a partition to different LogicLock regions and if there are some unassigned nodes which fall in the root LogicLock region where nodes are often from more than one partition. Thus, you can move the unassigned nodes to the defined LogicLock regions in the respective partitions and use the floorplan aware synthesis feature for better DSP and RAM balancing.

The floorplan aware synthesis feature is turned on by default. 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. Set the **Use LogicLock Constraints During Resource Balancing** option to **Off** in the **Analysis & Synthesis Settings** page by clicking **More Settings**.

DSP balancing converts extra DSP blocks in the design into equivalent logic to meet Fitter requirements where the number of DSP blocks in design is less than or equal to the number of DSP blocks available. RAM balancing converts RAMs from one RAM type to another to meet Fitter requirements where the RAM block utilization of each RAM type is within limits of the available blocks for each RAM type. The floorplan aware synthesis option also allows you to specify maximum resources for different RAM types, such as **Maximum Number of M4K/M9K Memory Blocks**, **Maximum Number of M512 Memory Blocks**, or **Maximum Number of M-RAM/M144K Memory Blocks**.

You can specify the maximum DSP and RAM resource allocation by performing the following steps:

1. On the Assignment menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Settings** dialog box appears.
4. Under **Option**, in the **Name** list, select **Maximum DSP Block Usage** or **Maximum Number** <*block type*> **Memory Blocks** and specify the resource count in the **Setting** list.
5. Click **OK**.
6. Click **OK**.

☞ HardCopy II devices have limited RAM blocks and there is no assignment to limit the RAM blocks usage in the HardCopy II device migration. Thus, Altera recommends that the maximum resource options are set to the default value of **-1 (UNLIMITED)** for the migration flow.

You can view the DSP and RAM block usage after balancing from the Compilation Report.

Restructure Multiplexers

This option specifies whether the Quartus II software should extract and optimize buses of multiplexers during synthesis.

This option is useful if your design contains buses of fragmented multiplexers. This option restructures multiplexers more efficiently for area, allowing the design to implement multiplexers with a reduced number of LEs or ALMs. This option is available for the following device families: Arria GX, Stratix series, Cyclone series, HardCopy II, and MAX II.

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

Table 9–3 lists the settings for the logic option, which you can apply only to a design entity individual node, or to an entity that is a design partition. Beginning in Quartus II version 9.0, this option is only valid when set on an entity that is a design partition. You can also specify this option on the **Analysis & Synthesis Settings** page in the **Settings** dialog box for your whole project by clicking **More Settings** and setting the option value.

Table 9–3. Restructure Multiplexer Settings

Setting	Description
On	Enables multiplexer restructuring to minimize your design area. This setting can reduce the f_{MAX}.
Off	Disables multiplexer restructuring to avoid possible reductions in f_{MAX}.
Auto (Default)	Allows the compiler to determine whether to enable the option based on your other Quartus II synthesis settings. The option is **On** when the **Optimization Technique** option is set to **Area**, **Balanced**, or **Speed**. When the **Optimization Technique** option is set to **Speed**, Quartus II integrated synthesis attempts to restructure the multiplexers selectively and makes a good trade-off between area and f_{MAX}.

After you have compiled your design, you can view multiplexer restructuring information in the **Multiplexer Restructuring Statistics** report in the **Multiplexer Statistics** folder under **Analysis & Synthesis Optimization Results** in the **Analysis & Synthesis** section of the Compilation Report. Table 9–4 describes the information that is listed in the **Multiplexer Restructuring Statistics** report table for each bus of multiplexers.

Table 9–4. Multiplexer Information in the Multiplexer Restructuring Statistics Report

Heading	Description
Multiplexer Inputs	The number of different choices that are multiplexed together.
Bus Width	The width of the bus in bits.
Baseline Area	An estimate of how many logic cells are required to implement the bus of multiplexers (before any multiplexer restructuring takes place). This estimate can be used to identify any large multiplexers in the design.
Area if Restructured	An estimate of how many logic cells are required to implement the bus of multiplexers if **Multiplexer Restructuring** is applied.
Saving if Restructured	An estimate of how many logic cells are saved if **Multiplexer Restructuring** is applied.
Registered	An indication of whether registers are present on the multiplexer outputs. **Multiplexer Restructuring** uses the secondary control signals of a register (such as synchronous clear and synchronous load) to further reduce the amount of logic required to implement the bus of multiplexers.
Example Multiplexer Output	The name of one of the multiplexers' outputs. This name can help determine where in the design the multiplexer bus originated.

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*.

Synthesis Effort

This option specifies the overall synthesis effort level in the Quartus II software. The level can be either **Fast** or **Auto**.

Auto is the default, which means synthesis goes through the normal flow and tries to optimize your design as much as possible.

When the effort level is set to **Fast**, Quartus II integrated synthesis skips a number of steps to make synthesis run much faster (at the cost of performance and resource utilization). This option is especially useful if you perform an early timing estimate. 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.

Altera recommends using the **Fast** synthesis effort level with the Fitter early timing estimate feature. When the **Fast** synthesis effort level is used with the full Fitter, the Fitter runtime might increase because fast synthesis produces a netlist that is slightly harder for the Fitter to route as compared to the netlist from a normal synthesis.

To set the **Synthesis Effort** option from the Quartus II GUI, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.

2. Under **Category**, click **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**.
4. Next to **Name**, from the pull-down menu, select **Synthesis Effort**.
5. Next to **Setting**, from the pull-down menu, select **Auto** or **Fast**. Click **OK**.
6. Click **OK** to close the **Settings** dialog box.

To set the **Synthesis Effort** option at the command line, use the `--effort` option, as shown in Example 9–27.

Example 9–27. Command Syntax for Specifying Synthesis Effort Option

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

If you want to run fast synthesis with the Fitter **Early Timing Estimate** option, use the command shown in Example 9–28. This command runs the full flow with Timing Analysis.

Example 9–28. Command Syntax for running fast synthesis with Early Timing Estimate Option

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

You can also run this flow from the Tasks pane in the Quartus II software. Select and expand **Compile Design**, then **Analysis & Synthesis**. Double-click **Early Timing Estimate** to start the flow.

State Machine Processing

This logic option specifies the processing style used to compile a state machine. Table 9–5 lists the settings for this logic option, which you can apply to a state machine name or to a design entity containing a state machine. You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box.

Table 9–5. State Machine Processing Settings

Setting	Description
Auto (Default)	Allows the compiler to choose what it determines to be the best encoding for the state machine
Minimal Bits	Uses the least number of bits to encode the state machine
One-Hot	Encodes the state machine in the one-hot style. See the example below for details.
User-Encoded	Encodes the state machine in the manner specified by the user
Sequential	Uses a binary encoding in which the first enumeration literal in the Enumeration Type has encoding `0`, the second `1`, and so on.
Gray	Uses 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	Uses an encoding similar to a gray code, in which each state only has one bit different from its neighboring states. Each state is generated by shifting the previous state’s bits to the right by 1; the most significant bit of each state is the negation of the least significant bit of the previous state. An N-bit Johnson code can represent at most 2N states but requires less logic than a gray encoding.

The default state machine encoding, which is **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 to ensure that your state machine is inferred and encoded correctly, 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 to zero. Quartus II integrated synthesis creates one-hot register encoding by using standard one-hot encoding and then inverting 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: each state can be recognized 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 following 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 the **State Machine Processing** logic option is set 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 the following example:

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

If the software infers states `S0, S1,...` it 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.

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. Refer to "Manually Specifying State Assignments Using the syn_encoding Attribute" for more information.

For information about the **Safe State Machine** option, refer to "Safe State Machines" on page 9–41.

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 9–36. 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 the different encoding styles shown in Table 9–6.

Table 9–6. syn_encoding Attribute Values

Attribute Value	Description
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the `"sequential"` encoding. If there 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, the second 1, and so on.
`"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 2N states but requires less logic than a gray encoding.
`"one-hot"`	The default encoding style requiring N bits, where N is the number of enumeration literals in the Enumeration Type.
`"compact"`	Use an encoding with the fewest bits.

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

In Example 9–29, 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 9–29. 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.

Example 9–30. Specifying User-Encoded States with the syn_encoding Attribute in Verilog-2001

```
(* 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
```

In Example 9–30, the states are encoded as follows:

```
init = "00"
last = "11"
next = "01"
later = "10"
```

Without the `syn_encoding` attribute, the Quartus II software would encode the state machine based on the current value of the **State Machine Processing** logic option.

If you are also specifying a safe state machine (as described in "Safe State Machines" on page 9–41), separate the encoding style value in the quotation marks with the safe value with a comma, as follows: "`safe, one-hot`" or "`safe, gray`".

Manually Specifying Enumerated Types Using the enum_encoding Attribute

By default, the Quartus II software one-hot encodes all user-defined Enumerated Types. 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 using the encoding specified by the attribute, and they are not listed as state machines in the Report window for the project. 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 must 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 there are enumeration literals in your Enumeration Type. In addition, the encodings must all 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 the code fragment of Example 9–31, the `enum_encoding` attribute specifies an arbitrary user encoding for the Enumeration Type `fruit`.

Example 9–31. 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";
```

In this example, the enumeration literals are encoded as:

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

You might want to 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 shown in Table 9–7.

Table 9–7. enum_encoding Attribute Values

Attribute Value	Description
`"default"`	Use an encoding based on the number of enumeration literals in the Enumeration Type. If there are fewer than five literals, use the `"sequential"` encoding. If there 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`, the second `1`, and so on.
`"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, where *N* is the number of enumeration literals in the Enumeration Type.

Observe that in Example 9–31, the `enum_encoding` attribute manually specified a gray encoding for the Enumeration Type `fruit`. This example could be written more concisely by specifying the `"gray"` encoding style instead of a manual encoding, as shown in Example 9–32.

Example 9–32. 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 Machines

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

It is possible for a finite state machine to 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. Use this option if you have asynchronous inputs to your state machine. The most common cause of this situation 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. An alternative is to add synchronizer registers to the inputs.

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 allow you to explicitly specify a behavior for all states in the state machine, including unreachable states. However, synthesis tools detect if state machine logic is unreachable and minimize or remove the logic. Any flag signals or logic used in the design to indicate such an illegal state are also removed. If the state machine is implemented as safe, the recovery logic forces its transition from an illegal state to the reset state.

The **Safe State Machine** option can be set globally, or on individual state machines. To set this option, perform the following steps:

1. On the Assignments menu, click **Settings**. The **Settings** dialog box appears.
2. In the **Category** list, select **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. The **More Analysis & Synthesis Settings** dialog box appears.
4. In the **Existing option settings** list, select **Safe State Machine**.
5. Under **Option**, in the **Setting** list, select **On**.
6. Click **OK**.
7. Click **OK** to close the **Settings** dialog box.

You can also use the Assignment Editor to turn on the **Safe State Machine** option for specific state machines.

You can set the `syn_encoding` `safe` attribute on a state machine in HDL, as shown in Example 9–33 through Example 9–35.

Example 9–33. Verilog HDL Code: a Safe State Machine Attribute

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

Example 9–34. Verilog-2001 Code: a Safe State Machine Attribute

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

Example 9–35. VHDL Code: a Safe State Machine Attribute

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

If you are also specifying an encoding style (as described in "Manually Specifying State Assignments Using the syn_encoding Attribute" on page 9–37), 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 the design. Therefore, Altera recommends that you set this option only on the critical state machines in the design where the safe mode is required, such as a state machine that uses inputs from asynchronous clock domains. You can also reduce the necessity of this option by correctly synchronizing inputs coming from other clock domains.

☞ If the `safe` state machine assignment is made on an instance that is not recognized 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 instance is recognized and properly inferred as a state machine.

For guidelines to ensure that your state machine is inferred correctly, 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). Registers in the device core hardware power up to 0 in all Altera devices. For the register to power up with a logic level High specified using this option, 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 low, but the register output is inverted so the signal arriving at all destinations is high. This option is available for all Altera devices supported by the Quartus II software except MAX® 3000A and MAX 7000S devices.

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 the design entity. If this option is assigned 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 it to take effect. You can also apply the option to a pin with the logic configurations described in the following list:

- If this option is turned on for an input pin, the option is transferred automatically to the register that is driven by the pin if the following conditions are present:
  - There is 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 this option is turned on for an output or bidirectional pin, it is transferred automatically to the register that feeds the pin, if the following conditions are present:
  - There is no logic, other than inversion, between the register and the pin
  - The register does not fan-out to any other logic

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 that are assigned values in Verilog HDL initial blocks into power-up conditions. Synthesis of these default and initial constructs enables the design's synthesized behavior to match, as closely as possible, the power-up state of the HDL code during a functional simulation.

For example, the following register declarations all set a power-up level of V_{CC} or a logic value "1":

```
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 the power-up level is affected by set and reset control signals, 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 the design that do not have a defined power-up condition. This option is turned on by default.

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 this option is enabled, 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 the design starts up on the first clock signal. Thus, the register is not stuck at V_{CC} and cannot be removed. Similarly, if the register has a clear signal, it is not removed because after the clear is asserted, 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 indicating 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**.

Remove Duplicate Registers

If you turn on this logic option, the compiler removes registers that are identical to another register. If two registers generate the same logic, the compiler removes the second one, and the first one fans out to the second one's destinations. Also, if the deleted register has different logic option assignments, the compiler ignores them. This option is turned on by default.

Typically, you should use this option only if you want to prevent the compiler from removing duplicate registers. That is, you should use this option only with the **Off** setting. You can apply this option to an individual register or a design entity that contains registers.

Remove Redundant Logic Cells

This logic option removes redundant `LCELL` primitives or WYSIWYG cells. The option is off by default to preserve logic cells that have been used intentionally. If you turn on this option, the compiler optimizes a circuit for area and speed. You can set this option globally or apply it to individual nodes and entities. If you turn on the option at the global level, you can use the `keep` attribute or **Implement as Output of Logic Cell** logic option to preserve specific wire signals or nodes (refer to "Keep Combinational Node/Implement as Output of Logic Cell" on page 9–46).

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 a register from being reduced to a constant or merged with a duplicate register. This option can preserve a register so you can observe it during simulation or with the SignalTap II Embedded Logic Analyzer. Additionally, it can preserve registers if you are creating a preliminary version of the design in which secondary signals are not specified. You can also use the attribute to preserve a duplicate of an I/O register so that one copy can be placed in an I/O cell and the second can be placed in the core. By default, the software might remove one of the two duplicate registers. In this case, the `preserve` attribute can be added to both registers to prevent this.

☞ 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 9–46.

The **Preserve Registers** option prevents a register from being inferred as a state machine.

You can set the **Preserve Registers** logic option in the Quartus II GUI or you can set the `preserve` attribute in your HDL code, as shown in Example 9–36 through Example 9–38. In these examples, the `my_reg` register is preserved.

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

Example 9–36. Verilog HDL Code: syn_preserve Attribute

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

Example 9–37. Verilog-2001 Code: syn_preserve Attribute

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

The `= 1` after the `preserve` in Example 9–36 and Example 9–37 is optional, because the assignment uses a default value of `1` when it is specified.

Example 9–38. VHDL Code: preserve Attribute

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

Disable Register Merging/Don’t Merge Register

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

You can use this option to instruct the compiler to correctly use your timing constraints for the register during synthesis. For example, if the register has a multicycle constraint, this option prevents the compiler from merging other registers into the specified register, avoiding unintended timing effects and functional differences.

This option differs from the **Preserve Register** option because it does not prevent a register with constant drivers or a redundant register from being removed. In addition, this option prevents other registers from merging with the specified register.

You can set the **Disable Register Merging** logic option in the Quartus II GUI, or you can set the `dont_merge` attribute in your HDL code, as shown in Example 9–39 through Example 9–41. In these examples, the `my_reg` register is prevented from merges.

Example 9–39. Verilog HDL Code: dont_merge Attribute

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

Example 9–40. Verilog-2001 Code: dont_merge Attribute

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

Example 9–41. VHDL Code: dont_merge Attribute

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

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 is different from the **Preserve Registers** option, which prevents a register from being reduced to a constant or merged 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 it in the Simulator or the SignalTap II Embedded Logic Analyzer. Additionally, it can retain registers if you are creating a preliminary version of the design in which the registers' fan-out logic is not specified. This option is supported for inferred registers in the Arria GX, Stratix series, Cyclone series, and MAX II device families.

You can set the **Preserve Fan-out Free Register Node** logic option in the Quartus II GUI, or you can set the `noprune` attribute in your HDL code, as shown in Example 9–42 though Example 9–44. In these examples, the `my_reg` register is preserved.

☞ 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, registers with no fan-out are removed (or "pruned") during Analysis and 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 within its module or entity, you can use the logic option to retain the register through compilation.

The attribute name `syn_noprune` is supported for compatibility with other synthesis tools.

Example 9–42. Verilog HDL Code: syn_noprune Attribute

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

Example 9–43. Verilog-2001 Code: noprune Attribute

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

Example 9–44. VHDL Code: noprune Attribute

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

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 will be the same as the name of the wire or node. You can use this directive to make combinational nodes visible to the SignalTap II Embedded Logic Analyzer.

The option cannot keep nodes that have no fan-out. Node names cannot be maintained for wires with tri-state drivers, or if the signal feeds a top-level pin of the same name (in this case, the node name is changed to a name such as <*net name*>`~buf0`).

You can set the **Implement as Output of Logic Cell** logic option in the Quartus II GUI, or you can set the `keep` attribute in your HDL code, as shown in Example 9–45 through Example 9–47. 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 9–45. Verilog HDL Code: keep Attribute

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

Example 9–46. Verilog-2001 Code: keep Attribute

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

Example 9–47. VHDL Code: syn_keep Attribute

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

Don't Retime, Disabling Synthesis Netlist Optimizations

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

You can use this option to turn off retiming optimizations 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 GUI 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 9–48 through Example 9–50. In these examples, the `my_reg` register is prevented from being retimed.

Example 9–48. Verilog HDL Code: dont_retime Attribute

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

Example 9–49. Verilog-2001 Code: dont_retime Attribute

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

Example 9–50. 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 set to `1` or `true`.

Don't Replicate, Disabling Synthesis Netlist Optimizations

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

You can use this option to turn off register replication (or duplication) optimizations so that the compiler can use your timing constraints for the register.

You can set the **Netlist Optimizations** logic option to **Never Allow** in the Quartus II GUI 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 9–51 through Example 9–53. In these examples, the `my_reg` register is prevented from being replicated.

Example 9–51. Verilog HDL Code: dont_replicate Attribute

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

Example 9–52. Verilog-2001 Code: dont_replicate Attribute

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

Example 9–53. 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 set to `1` or `true`.

Maximum Fan-Out

This attribute and logic option directs the compiler to control the number of destinations fed by a node. 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 (or replicate) a register that feeds nodes in different locations on the target device. Duplicating the register can allow the Fitter to place these new registers closer to their destination logic, minimizing routing delay.

This option is available for all devices supported in the Quartus II software except MAX 3000, MAX 7000, FLEX 10K®, and ACEX® 1K devices. To turn off the option for a given node if the option is set 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 maximum fan-out constraint is honored 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/or pins through data, address, clock enable, and so on, but not through any asynchronous control ports (such as asynchronous clear)

The software does not create duplicate nodes in these cases, either because there is no clear way to duplicate the node, or to avoid the possible situation in which small differences in timing could produce functional differences in the implementation (in the third condition above where asynchronous control signals are involved). If the constraint cannot be applied because one of these conditions is not met, the Quartus II software issues a message indicating that it ignored maximum fan-out assignment. To instruct the software not to check the node's destinations for possible problems like 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 registers are not affected by any of the netlist optimization algorithms, such as register retiming.

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 GUI; this option supports wildcard characters. You can also set the `maxfan` attribute in your HDL code, as shown in Example 9–54 through Example 9–56. 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 name for compatibility with other synthesis tools.

Example 9–54. Verilog HDL Code: syn_maxfan Attribute

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

Example 9–55. Verilog-2001 Code: maxfan Attribute

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

Example 9–56. VHDL Code: maxfan Attribute

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

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. The option is on by default. You can set this option to **Off** for individual registers or design entities to solve fitting or performance issues with designs that have many clock enables. 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 specific logic is not automatically moved 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. Applying the `direct_enable` attribute to a specific signal instructs the software to use the clock enable port of a register to implement the signal. The attribute ensures that the clock enable port is driven directly by the signal, and the signal is not optimized or combined with any other logic.

Example 9–57 through Example 9–59 show how to set this attribute to ensure that the signal is preserved and used directly 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 9–57. Verilog HDL Code: direct_enable attribute

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

Example 9–58. Verilog-2001 Code: syn_direct_enable attribute

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

Example 9–59. VHDL Code: direct_enable attribute

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

Megafunction Inference Control

The Quartus II compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction. The software uses the Altera megafunction code when compiling your design, even when you do not specifically instantiate the megafunction. The software infers megafunctions to take advantage of logic that is optimized for Altera devices. The area and performance of such logic can be better than the results obtained by 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 generally 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

Use the **Auto DSP Block Replacement** logic option to control DSP block inference for multiply-accumulations and multiply-adders. This option is turned on by default. To disable inference, turn off this option for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box, or disable the option for a specific block with the Assignment Editor.

Any registers that the software maps to the ALTMULT_ACCUM and ALTMULT_ADD megafunctions and places in DSP blocks are not available in the Simulator because their node names do not exist after synthesis.

Shift Registers

Use the **Auto Shift Register Replacement** logic option to control shift register inference. This option is turned on by default. To disable inference, turn off this option for your whole 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. The software might not infer small shift registers because small shift registers typically 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 considered too small.

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 **Auto Shift Register Replacement** logic option is turned off automatically when a formal verification tool is selected on the **EDA Tool Settings** page. The software issues a warning and lists shift registers that would have been inferred if no formal verification tool was selected in the compilation report. To allow the use of a megafunction for the shift register in the formal verification flow, you can either instantiate a shift register explicitly using the MegaWizard™ Plug-In Manager or make the shift register into a black box in a separate entity/module.

RAM and ROM

Use the **Auto RAM Replacement** and **Auto ROM Replacement** logic options to control RAM and ROM inference, respectively. These options are turned on by default. To disable inference, turn off the appropriate option for your whole 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 inferred shift registers are implemented 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").

The software might not infer very small RAM or ROM blocks because very small memory blocks can typically be implemented more efficiently by using 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 considered too small.

☞ The **Auto ROM Replacement** logic option is automatically turned off when a formal verification tool is selected in the **EDA Tool Settings** page. A warning is issued and a report panel lists ROMs that would have been inferred if no formal verification tool was selected. To allow the use of 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/module.

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

RAM to Logic Cell Conversion

The **Auto RAM to Logic Cell Conversion** option allows Quartus II integrated synthesis to convert RAM blocks that are small in size to logic cells if the logic cell implementation is deemed to give better quality of results. Only single-port or simple-dual port RAMs with no initialization files can be converted to logic cells. This option is off by default. 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 your whole project in the **More Analysis & Synthesis Settings** dialog box.

For the FLEX 10K, APEX series, Arria GX, and the Stratix series of devices, the software uses the following rules to determine whether a RAM should be placed 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 series 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

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 to be used, or specify the use of standard logic cells (LEs or ALMs). The attributes are supported only for device families with TriMatrix embedded memory blocks.

The `ramstyle` and `romstyle` attributes take a single string value. The values "`M512`", "`M4K`", "`M-RAM`", "`MLAB`", "`M9K`", and "`M144K`" (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 value `logic` indicates that the RAM or ROM should be implemented 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 value of `logic`, the memory still appears as a RAM or ROM block in the RTL Viewer, but it is converted to regular logic during a later synthesis step.

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

Example 9–60 through Example 9–62 specify that all memory in the module or entity `my_memory_blocks` should be implemented using a specific type of block.

Example 9–60. Verilog-1995 Code: Applying a romstyle Attribute to a Module Declaration

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

Example 9–61. Verilog-2001 Code: Applying a ramstyle Attribute to a Module Declaration

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

Example 9–62. 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 9–63 through Example 9–65 specify that the inferred memory `my_ram` or `my_rom` should be implemented using regular logic instead of a TriMatrix memory block.

Example 9–63. Verilog-1995 Code: Applying a syn_ramstyle Attribute to a Variable Declaration

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

Example 9–64. Verilog-2001 Code: Applying a romstyle Attribute to a Variable Declaration

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

Example 9–65. 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";
```

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

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.

In some cases, an inferred RAM must be mapped into regular logic cells because it has a read-during-write behavior that is not supported by the TriMatrix memory blocks in your target device. In other cases, the Quartus II software must insert extra logic to mimic read-during-write behavior of the HDL source, increasing the area of your design and potentially reducing its performance. In 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 through the Quartus II GUI, click **More Settings** on the **Analysis & Synthesis Settings** page of the **Settings** dialog box. Example 9–66 and Example 9–67 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 you don't require a defined read-during-write condition in your design, this extra logic is not required. With the `no_rw_check` attribute, Quartus II integrated synthesis won't generate the extra logic.

Example 9–66. 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 9–67. 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.

This attribute is also useful in cases where some asynchronous RAM blocks might be coded with read-during-write behavior that does not match the Stratix III architecture. Thus, the device behavior would not exactly match the behavior described in the code. 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 this attribute is set, Quartus II integrated synthesis allows the behavior of the output to be different when the asynchronous read occurs on an address that had a write on the most recent clock edge. That is, functional HDL simulation results will not match the hardware behavior if you write to an address that is being read.

To include both attributes, set the value of the ramstyle attribute to "`MLAB, no_rw_check`".

Example 9–68 and Example 9–69 show the method of setting two values to the ramstyle attribute using a small asyncronous RAM block, with the ramstyle synthesis attribute set so that the memory can be implemented in the MLAB memory block and 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 8 memory ALUTs and just 15 registers.

Example 9–68. 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 9–69. 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;
```

RAM Initialization File—for Inferred Memory

The `ram_init_file` attribute specifies the initial contents of an inferred memory in the form of a Memory Initialization File (**.mif**). The attribute takes a string value containing the name of the RAM initialization file.

Example 9–70. Verilog-1995 Code: Applying a ram_init_file Attribute

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

Example 9–71. Verilog-2001 Code: Applying a ram_init_file Attribute

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

Example 9–72. 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** file for the inferred RAM.

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 containing 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 dedicated multiplication hardware.

Example 9–73. Verilog-1995 Code: Applying a multstyle Attribute to a Module Declaration

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

Example 9–74. 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 to be used for a multiplication operator whose result is directly assigned to the variable. It overrides the `multstyle` attribute associated with the enclosing module, if present. In Example 9–75 and Example 9–76, the `multstyle` attribute applied to variable `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 9–75. 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 9–76. 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 containing the * operator, the attribute specifies the implementation style for that specific operator alone and overrides any `multstyle` attribute associated with the target variable or enclosing module. In Example 9–77, the `multstyle` attribute indicates that `a * b` should be implemented in dedicated hardware.

Example 9–77. 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 9–78, 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 9–78. 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 to be used for all instances of the `*` operator whose result is directly assigned to the signal or variable. It overrides the `multstyle` attribute associated with the enclosing entity or architecture, if present. In Example 9–79, the `multstyle` attribute associated with signal `result` directs the Quartus II software to implement `a * b` in general logic rather than dedicated hardware.

Example 9–79. 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

A Verilog HDL case statement is considered 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 the unspecified states to be treated 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 avoids the latch inference problems discussed in the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Latches have limited support in formal verification tools. It is important to ensure that you do not infer latches unintentionally; for example, through an incomplete case statement when using formal verification. Formal verification tools do support the `full_case` synthesis attribute (with limited support for attribute syntax, as described in “Synthesis Attributes” on page 9–25).

When you use the `full_case` attribute, there is a potential cause for a simulation mismatch between Verilog HDL functional and post-Quartus II simulation because unknown case statement cases can still function like latches during functional simulation. For example, a simulation mismatch can occur with the code in Example 9–80 when `sel` is `2'b11` because a functional HDL simulation output behaves like a latch while the Quartus II simulation output behaves like “don’t care.”

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

The case statement in Example 9–80 is not full because not all binary values for `sel` are specified. Because the `full_case` attribute is used, synthesis treats the output as “don’t care” when the `sel` input is `2'b11`.

Example 9–80. 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 9–81 in the `case` header instead of the comment form shown in Example 9–80.

Example 9–81. Verilog-2001 Syntax for the full_case Attribute

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

Parallel Case

The `parallel_case` attribute indicates that a Verilog HDL case statement should be considered parallel; that is, only one case item can be matched 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 relationship 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 required to satisfy this priority relationship.

Attaching a `parallel_case` attribute to a case statement's 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 reduce the complexity of 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.

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

☞ Altera recommends that you avoid using the `parallel_case` attribute, due to the possibility of introducing mismatches between Verilog HDL functional and 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 9–82 shows a `casez` statement with overlapping case items. In functional HDL simulation, the three case items have a priority order that depends on 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 order. 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 9–82. 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 shown in Example 9–83 in the `case` (or `casez`) header instead of the comment form, as shown in Example 9–82.

Example 9–83. 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.

A common use of these directives is to indicate a portion of code that is intended 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 9–84 and Example 9–85 show these directives.

Example 9–84. Verilog HDL Code: Translate Off and On

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

Example 9–85. VHDL Code: Translate Off and On

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

If you wish to ignore a portion of code in Quartus II integrated synthesis only, 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 is intended 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. This allows you to compile code that was previously intended to be ignored by third-party synthesis tools; for example, megafunction declarations that were treated as black boxes in other tools but can be compiled in the Quartus II software. 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.

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 is 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` marks 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 both a megafunction instantiation for synthesis and a behavioral description for simulation.

Because formal verification tools do not recognize the `read_comments_as_HDL` directive, it is not supported when you are using formal verification.

In Example 9–86 and Example 9–87, the commented code enclosed by `read_comments_as_HDL` is visible to the Quartus II compiler and is synthesized.

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

Example 9–86. 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 9–87. 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
```

Use I/O Flipflops

This 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. Applying the `useioff` synthesis attribute can improve I/O performance by minimizing setup, clock-to-output, and clock-to-output enable times. This synthesis attribute is supported using the **Fast Input Register**, **Fast Output Register**, and **Fast Output Enable Register** logic options that can also be set in the Assignment Editor.

For more information about which device families support fast input, output, and output enable registers, refer to the device family data sheet, device handbook, or the Quartus II Help.

The `useioff` synthesis attribute takes a Boolean value and can only be applied 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 9–88 and Example 9–89, the `useioff` synthesis attribute directs the Quartus II software to implement the registers `a_reg`, `b_reg`, and `o_reg` in the I/O cells corresponding to the ports `a`, `b`, and `o`, respectively.

Example 9–88. 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
```

Verilog-2001 syntax also accepts the type of statements shown in Example 9–89 and Example 9–90 instead of the comment form shown in Example 9–88.

Example 9–89. Verilog-2001 Code: the useioff Attribute

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

Example 9–90. 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

This attribute enables you to assign pin locations in your HDL source. The attribute can be used only on the ports of the top-level entity or module in the design, and cannot be used to assign pin locations from entities at lower levels of the design hierarchy. 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 device's pin table.

In addition to `chip_pin`, 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 9–91 through Example 9–93 show different ways of assigning input pin `my_pin1` to Pin C1 and `my_pin2` to Pin 4 on a different target device.

Example 9–91. 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 9–92. 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 9–93. 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 port's range determines the mapping of assignments to individual bits in the port. To leave a particular bit unassigned, simply leave its corresponding pin assignment blank.

Example 9–94 assigns `my_pin[2]` to `Pin_4`, `my_pin[1]` to `Pin_5`, and `my_pin[0]` to `Pin_6`.

Example 9–94. Verilog-1995 Code: Applying Chip Pin to a Bus of Pins

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

Example 9–95 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 9–95. Verilog-1995 Code: Applying Chip Pin to Part of a Bus

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

Example 9–96 assigns `my_pin[2]` to Pin 4 and `my_pin[0]` to Pin 6, but leaves `my_pin[1]` unassigned.

Example 9–96. 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";
```

Using altera_attribute to Set Quartus II Logic Options

This attribute enables you to apply Quartus II 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 beyond Analysis and Synthesis, such as Fitting.

Assignments or settings made through the Quartus II GUI, the **.qsf** file, 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 9–25.

The attribute value is a single string containing a list of **.qsf** file variable assignments separated by semicolons, as shown in Example 9–97.

Example 9–97. 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/or section tag, use the syntax in Example 9–98 for each **.qsf** file variable assignment.

Example 9–98. Syntax for Each .qsf File Variable Assignment

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

The syntax for the full attribute value, including the optional target, source, and section tags for two different **.qsf** file assignments, is shown in Example 9–99.

Example 9–99. Syntax for Fill 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 a variable's assigned value is a string of text, you must use escaped quotes around the value in Verilog HDL, or double-quotes in VHDL, as in the following examples (using non-existent variable and value terms):

Verilog HDL

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

VHDL

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

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

Example 9–100 through Example 9–102 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, so you should apply the attribute to one of the instance's output nets. The Quartus II software moves the attribute to the inferred instance automatically.

Example 9–100. Verilog-1995 Code: Applying Altera Attribute to an Instance

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

Example 9–101. Verilog-2001 Code: Applying Altera Attribute to an Instance

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

Example 9–102. 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 9–103 through Example 9–105 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 9–103. Verilog-1995 Code: Applying Altera Attribute to an Entity

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

Example 9–104. Verilog-2001 Code: Applying Altera Attribute to an Entity

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

Example 9–105. 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 involving more than one instance. In Example 9–106 through Example 9–108, the `altera_attribute` is used to cut all timing paths from `reg1` to `reg2`, equivalent to this Tcl or QSF command:

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

Example 9–106. Verilog-1995 Code: Applying Altera Attribute with -to

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

Example 9–107. Verilog-2001 Code: Applying Altera Attribute with -to

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

Example 9–108. VHDL Code: Applying Altera Attribute with -to

```
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.

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

Analyzing Synthesis Results

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

Analysis and Synthesis Section of the Compilation Report

The Compilation Report, which provides a summary of results for the project, appears after a successful compilation, or you can choose it from the Processing menu. After Analysis and Synthesis, before the Fitter begins, the Summary information provides a summary of utilization based on synthesis data, before Fitter optimizations have occurred. Synthesis-specific information is listed in the **Analysis & Synthesis** section.

There are various report sections under Analysis and Synthesis, 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 Quartus II Help.

Project Navigator

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

If you hold your mouse pointer over one of the entities in the **Hierarchy** tab, a tooltip appears that shows parameter information for each instance.

Analyzing and Controlling Synthesis Messages

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

Quartus II Messages

The messages that appear during Analysis and Synthesis describe many of the optimizations that the software performs during the synthesis stage, and provide information about how the design is interpreted. You should always check the messages to analyze **Critical Warnings** and **Warnings**, because these messages can relate to important design problems. It is also useful to read the information messages **Info** and **Extra Info** to get more information about how the software processes your design.

The **Info**, **Extra Info**, **Warning**, **Critical Warning**, and **Error** tabs display messages grouped by type.

You can right-click on a message in the Messages window and get help on the message, locate the source of the message in your design, and manage messages.

You can use message suppression to reduce the number of messages listed after a compilation by preventing individual messages and entire categories of messages from being displayed. For example, if you review a particular message and determine that it is not caused by something in your design that should be changed or fixed, you can suppress the message so it is not displayed during subsequent compilations. This saves time because you see only new messages during subsequent compilations.

You can right-click on an individual message in the Messages window and choose commands in the **Suppress** submenu. Another way to achieve the same goal is to open the Message Suppression Manager. To do this, right-click in the Messages window, point to **Suppress**, and click **Message Suppression Manager**.

For more information about messages and how to suppress them, refer to the *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook.*

Beginning with Quartus II software version 8.1, you can specify the type of Analysis and 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**. The **Settings** dialog box appears.
2. In the **Category** list, click **Analysis & Synthesis Settings**. The **Analysis & Synthesis Settings** page appears.
3. Click **More Settings**. Select the level for the **Analysis & Synthesis Message Level** option.

For more information about the **Analysis & Synthesis Message Level** option, refer to the Quartus II Help File.

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 three 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, you should always investigate 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. Consult the Help associated with any HDL error messages for assistance in removing the error from your design.

In Example 9–109, 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 typo: Variable `j` should be listed on the sensitivity list to avoid a possible simulation/synthesis mismatch.

Example 9–109. 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 this HDL code, the Quartus II software generates the following warning message:

```
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 9–110 are two different variables. However, declaring variables whose names only differ in case might confuse some users, especially those users who use VHDL, where variables are not case-sensitive.

Example 9–110. Generating HDL Info Messages

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

When processing this HDL code, the Quartus II software generates the following informational message:

```
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 neither `my_reg` or `MY_REG` are used in this small design:

```
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 see during the Analysis and 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 9–27.

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 9–8 details the information about the HDL message levels.

Table 9–8. HDL Info Message Level

Level	Purpose	Description
Level1	Displays high-severity messages only	If you want to see only those HDL messages that identify likely problems with your design, select Level1. When Level1 is selected, the Quartus II software issues a message only if there is a high probability that it points to an actual problem with your design.
Level2	Displays high-severity and medium-severity messages	If you want to see additional HDL messages that identify possible problems with your design, select Level2. This is the default setting.
Level3	Displays all messages, including low-severity messages	If you want to see 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 or make it easier to understand.

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

To set the HDL Message Level in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, click **Analysis & Synthesis Settings**. Set the desired message level from the pull-down menu in the **HDL Message Level** list, and 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 9–111 and Example 9–112.

Example 9–111. Verilog HDL Examples of message_level Directive

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

Example 9–112. 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

You can enable or disable a specific HDL info or warning message with its Message ID, which is displayed in parentheses at the beginning of the 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 use this method to disable messages for a specific module or entity. This method applies only the HDL messages, and if you disable a message with this method, the message is listed as a Suppressed message in the Quartus II GUI.

To disable specific HDL messages in the GUI, on the Assignments menu, click **Settings**. In the **Category** list, expand **Analysis & Synthesis Settings** and select **Advanced**. In the **Advanced Message Settings** dialog box, add the Message IDs you wish to enable or disable.

To enable or disable specific HDL messages in your HDL, use the `message_on` and `message_off` synthesis directives. Both directives take 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 in the middle of 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 its status is changed by another `message_on` or `message_off` directive.

Example 9–113. Verilog HDL message_off Directive for Message with ID 10000

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

Example 9–114. VHDL message_off Directive for Message with ID 10000

```
-- altera message_off 10000
```

Node-Naming Conventions in Quartus II Integrated Synthesis

Being able to find the logic node names after synthesis can be useful during verification or while debugging a design. This section provides an overview of the conventions used by the Quartus II software when it names the nodes created from your HDL design. The section focuses on the conventions for Verilog HDL and VHDL code, but AHDL and BDFs are discussed 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 typically 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 discusses the following topics:

- "Hierarchical Node-Naming Conventions"
- "Node-Naming Conventions for Registers (DFF or D Flipflop Atoms)"
- "Register Changes During Synthesis" on page 9–76
- "Preserving Register Names" on page 9–78
- "Node-Naming Conventions for Combinational Logic Cells" on page 9–78
- "Preserving Combinational Logic Names" on page 9–79

Hierarchical Node-Naming Conventions

To make each name in the design unique, the Quartus II software adds the hierarchy path to the beginning of each name. The "|" separator is used to indicate a level of hierarchy. For each instance in the hierarchy, the software adds the entity name and the instance name of that entity, using the ":" separator between each entity name and its instance name. For example, if a design instantiates entity A with the name `my_A_inst`, the hierarchy path of that entity would be `A:my_A_inst`. The full name of any node is obtained by starting with the hierarchical instance path, followed by a "|", and ending with the node name inside that entity, using the following convention:

<entity 0>`:`*<instance_name 0>*`|`*<entity 1>*:
<instance_name 1>`|...|`*<instance_name n>*

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`.

On the **Compilation Process Settings** page of the **Settings** dialog box, click **More Settings** and turn off **Display entity name for node name** to instruct the compiler to generate node names that do not contain the name for each level of the hierarchy. With this option off, the node names use the following convention:

<instance_name 0>`|`*<instance_name 1>*`|...|`*<instance_name n>*

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

In Verilog HDL and VHDL, inferred registers are named after the `reg` or `signal` connected to the output.

Example 9–115 is a description of a register in Verilog HDL that creates a DFF primitive called `my_dff_out`:

Example 9–115. Verilog HDL Register

```
wire dff_in, my_dff_out, clk;

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

Similarly, Example 9–116 is a description of a register in VHDL that creates a DFF primitive called `my_dff_out`.

Example 9–116. 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;
```

In AHDL designs, DFF registers are declared explicitly rather than inferred, so the software uses the user-declared name for the register.

For schematic designs using a **.bdf** file, all elements are given a name when they are instantiated in the design, so the software uses the user-defined name 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. In this case, the Quartus II integrated synthesis appends `~reg0` to the register name.

For example, the Verilog HDL code in Example 9–117 produces a register called `q~reg0`:

Example 9–117. 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 port is removed during hierarchy flattening and the register retains its original name, in this case, `q`.

Register Changes During Synthesis

On some occasions, you might not be able to find registers that you expect to see in the synthesis netlist. Registers might be removed by logic optimization, or their names might be changed due to synthesis optimization. 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 registers are packed 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 9–77
- "Packed Input and Output Registers of RAM and DSP Blocks" on page 9–77
- "Preserving Register Names" on page 9–78
- "Preserving Combinational Logic Names" on page 9–79

Synthesis and Fitting Optimizations

Registers might be removed by synthesis logic optimization if they are not connected to inputs or outputs in the design, or if the logic can be simplified due to constant signal values. Register names might also be changed due to synthesis optimizations, such as when duplicate registers are merged together 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 registers are used 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 registers can be combined or duplicated to optimize the 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 are removed during synthesis optimizations, and a brief reason for the removal. In the **Analysis & Synthesis** folder, open **Optimization Results**, and then open **Register Statistics**, and click on the **Registers Removed During Synthesis** report, and the **Removed Registers Triggering Further Register Optimizations** report. The second report contains a list of registers that are the cause of other registers being removed in the design. It provides a brief reason for the removal, and a list of registers that were removed due to the removal of the initial register.

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 logic is modified by physical synthesis). 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 9–78.

State Machines

If a state machine is inferred from your HDL code, the registers that represent the states are mapped into a new set of registers that implement the state machine. Most commonly, the software converts the state machine into a one-hot form where each state is represented by one register. In this case, for Verilog HDL or VHDL designs, the registers are named according to the name of the state register and the states, where possible.

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

In AHDL, state machines are explicitly specified with a machine name. State machine registers are given synthesized names based on the state machine name but not the state names. For example, if a state machine is called `my_fsm` and has four state bits, they might be synthesized 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 can be 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 registers and logic are typically implemented 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

Registers can be packed 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

You might want to preserve certain register names for verification or debugging, or to ensure that timing assignments are applied correctly. Quartus II integrated synthesis preserves certain nodes automatically if they are likely to be used 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. Refer to "Preserve Registers" on page 9–44 for details.

Use the `noprune` attribute to preserve a fan-out-free register through the entire compilation flow. Refer to "Noprune Synthesis Attribute/Preserve Fan-out Free Register Node" on page 9–46 for details.

Use the synthesis attribute `syn_dont_merge` to make sure registers are not merged with other registers, and other registers are not merged with them. Refer to "Disable Register Merging/Don't Merge Register" on page 9–45 for details.

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 9–118. Quartus II integrated synthesis uses the names `c`, `d`, `e`, and `f` for the combinational logic cells that are produced.

Example 9–118. 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** file, all elements are given a name when they are instantiated in the design and the software uses the user-defined name when possible.

☞ Node naming conventions for schematic buses in the Quartus II software version 7.2 and later are different than the MAX+PLUS II software and older versions of the Quartus II software. In most cases, the Quartus II software uses the appropriate naming convention for the design source file. Designs created using the Quartus II software version 7.1 or earlier use the MAX+PLUS II naming convention. Designs created in the Quartus II software version 7.2 and later use the Quartus II naming convention that matches the behavior of standard HDLs. In some cases, however, a design might contain files created in various versions. To set an assignment for a particular instance in the Assignment Editor, enter the instance name in the **To** field, choose **Block Design Naming** from the **Assignment Name** list, and set the value to **MaxPlusII** or **QuartusII**.

If logic cells, such as those created in Example 9–118, 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. In some cases, 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`, `w~2`, and so on. Sometimes the original wire name `w` is removed, and an arbitrary name such as `rtl~123` is created. It is a goal of Quartus II integrated synthesis 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 can help you understand your post-synthesis results and make it easier to debug your design or make assignments.

The software maintains combinational clock logic by making sure nodes that are likely to be a clock are not changed during synthesis. The software also maintains (or “protects”) multiplexers in clock trees so that the TimeQuest Timing 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 design selects between different clocks. To help 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 look-up 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. This option applies to Arria GX devices, the Stratix and Cyclone series, and MAX II devices.

Preserving Combinational Logic Names

You might want to 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. Refer to “Keep Combinational Node/Implement as Output of Logic Cell” on page 9–46 for details.

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 on combinational logic involved in `cut` assignments and `-through` assignments.

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

Scripting Support

You can run procedures and make settings described in this chapter 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 following command at the command prompt:

```
quartus_sh --qhelp ↵
```

The *Quartus II Scripting Reference Manual* includes the same information in PDF form.

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

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

Use the following Tcl command to make a global assignment:

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

Use the following Tcl command to make an instance assignment:

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

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

Use the following Tcl assignments to add an HDL or schematic entry design file to your project:

```
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 following option at the end of the `VERILOG_FILE` or `VHDL_FILE` command:

`-HDL_VERSION` <*language version*>

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

- `VERILOG_1995`
- `VERILOG_2001`

- `SYSTEMVERILOG_2005`
- `VHDL87`
- `VHDL93`

For example, to add a Verilog HDL file called **my_file** that is written in Verilog-1995, use the following command:

```
set_global_assignment -name VERILOG_FILE my_file.v -HDL_VERSION
VERILOG_1995
```

Quartus II Synthesis Options

Table 9–9 lists the **.qsf** file variable names and applicable values for the settings discussed in this chapter. The **.qsf** file variable name is used in the Tcl assignment to make the setting along with the appropriate value.

Table 9–9. Quartus II Synthesis Options (Part 1 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Add Pass-Through Logic to Inferred RAMs	`ADD_PASS_THROUGH_LOGIC_TO_INFERRED_ RAMS`	On/Off
Allow Any RAM Size for Recognition	`ALLOW_ANY_RAM_SIZE_FOR_RECOGNITION`	On/Off
Allow Any ROM Size for Recognition	`ALLOW_ANY_ROM_SIZE_FOR_RECOGNITION`	On/Off
Allow Any Shift Register Size for Recognition	`ALLOW_ANY_SHIFT_REGISTER_SIZE_FOR_ RECOGNITION`	On/Off
Allow Asynchronous Clear Usage For Shift Register Replacement	`ALLOW_ACLR_FOR_SHIFT_REGISTER_ RECOGNITION`	On/Off
Allow Synchronous Control Signals	`ALLOW_SYNCH_CTRL_USAGE`	On/Off
Analysis & Synthesis Message Level	`SYNTH_MESSAGE_LEVEL`	Low/Medium/High
Auto Carry Chains	`AUTO_CARRY_CHAINS`	On/Off
Auto Clock Enable Replacement	`AUTO_CLOCK_ENABLE_RECOGNITION`	On/Off
Auto DSP Block Replacement	`AUTO_DSP_RECOGNITION`	On/Off
Auto Gated Clock Conversion	`SYNTH_GATED_CLOCK_CONVERSION`	On/Off
Auto Open-Drain Pins	`AUTO_OPEN_DRAIN_PINS`	On/Off
Auto RAM Block Balancing	`AUTO_RAM_BLOCK_BALANCING`	On/Off
Auto RAM to Logic Cell Conversion	`AUTO_RAM_TO_LCELL_CONVERSION`	On/Off
Auto RAM Replacement	`AUTO_RAM_RECOGNITION`	On/Off
Auto Resource Sharing	`AUTO_RESOURCE_SHARING`	On/Off
Auto ROM Replacement	`AUTO_ROM_RECOGNITION`	On/Off
Auto Shift-Register Replacement	`AUTO_SHIFT_REGISTER_RECOGNITION`	Always/Auto/Off
Block Design Naming	`BLOCK_DESIGN_NAMING`	Auto/Max+Plus II/ Quartus II
Carry Chain Length	*`<device name>`*`_CARRY_CHAIN_LENGTH`	<*Maximum allowable length of a chain*>

Table 9–9. Quartus II Synthesis Options (Part 2 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Clock MUX Protection	`SYNTH_CLOCK_MUX_PROTECTION`	On/Off
Create Debugging Nodes for IP Cores	`ENABLE_IP_DEBUG`	On/Off
DSP Block Balancing	`DSP_BLOCK_BALANCING`	Auto/DSP Blocks/ Logic Elements/ Off/Simple 18-bit Multipliers/ Simple Multipliers/Width 18-bit Multipliers
Extract Verilog State Machines	`EXTRACT_VERILOG_STATE_MACHINES`	On/Off
Extract VHDL State Machines	`EXTRACT_VHDL_STATE_MACHINES`	On/Off
Force Use of Synchronous Clear Signals	`FORCE_SYNCH_CLEAR`	On/Off
HDL Message Level	`HDL_MESSAGE_LEVEL`	Level1/Level2/ Level3
Ignore CARRY Buffers	`IGNORE_CARRY_BUFFERS`	On/Off
Ignore CASCADE Buffers	`IGNORE_CASCADE_BUFFERS`	On/Off
Ignore GLOBAL Buffers	`IGNORE_GLOBAL_BUFFERS`	On/Off
Ignore LCELL Buffers	`IGNORE_LCELL_BUFFERS`	On/Off
Ignore Maximum Fan-Out Assignments	`IGNORE_MAX_FANOUT_ASSIGNMENTS`	On/Off
Ignore ROW GLOBAL Buffers	`IGNORE_ROW_GLOBAL_BUFFERS`	On/Off
Ignore SOFT Buffers	`IGNORE_SOFT_BUFFERS`	On/Off
Ignore translate_off and synthesis_off directives	`IGNORE_TRANSLATE_OFF_AND_SYNTHESIS_OFF`	On/Off
Ignore Verilog Initial Constructs	`IGNORE_VERILOG_INITIAL_CONSTRUCTS`	On/Off
Iteration limit for constant Verilog loops	`VERILOG_CONSTANT_LOOP_LIMIT`	*<Maximum limit to infinite loops before exhaustion of memory>*
Iteration limit for non-constant Verilog loops	`VERILOG_NON_CONSTANT_LOOP_LIMIT`	*<Maximum limit to infinite loops before exhaustion of memory>*
Limit AHDL Integers to 32 Bits	`LIMIT_AHDL_INTEGERS_TO_32_BITS`	On/Off
Maximum DSP Block Usage *(2)*	`MAX_BALANCING_DSP_BLOCKS`	*<Maximum DSP Block Usage Value>*
Maximum Number of M4K/ M9K Memory Blocks	`MAX_RAM_BLOCKS_M4K`	*<Maximum memory blocks usage>*
Maximum Number of M512 Memory Blocks	`MAX_RAM_BLOCKS_M512`	*<Maximum memory blocks usage>*
Maximum Number of M-RAM/ M144K Memory Blocks	`MAX_RAM_BLOCKS_MRAM`	*<Maximum memory blocks usage>*
NOT Gate Push-Back	`NOT_GATE_PUSH_BACK`	On/Off
Number of Inverted Registers Reported in Synthesis Report	`NUMBER_OF_INVERTED_REGISTERS_REPORTED`	*<Maximum number of inverted registers>*
Number of Removed Registers Reported in Synthesis Report	`NUMBER_OF_REMOVED_REGISTERS_REPORTED`	*<Maximum number of inverted registers>*

Table 9–9. Quartus II Synthesis Options (Part 3 of 3) *(Note 1)*

Setting Name	Quartus II Settings File Variable	Values
Optimization Technique	`<device family>_OPTIMIZATION_TECHNIQUE`	Area/Speed/ Balanced
Parallel Synthesis	`PARALLEL_SYNTHESIS`	On/Off
Perform WYSIWYG Primitive Resynthesis	`ADV_NETLIST_OPT_SYNTH_WYSIWYG_REMAP`	On/Off
PowerPlay Power Optimization	`OPTIMIZE_POWER_DURING_SYNTHESIS`	Normal compilation/ Extra effort/Off
Power-Up Don't Care *(2)*	`ALLOW_POWER_UP_DONT_CARE`	On/Off
Remove Duplicate Registers	`REMOVE_DUPLICATE_REGISTERS`	On/Off
Remove Redundant Logic Cells *(2)*	`REMOVE_REDUNDANT_LOGIC_CELLS`	On/Off
Restructure Multiplexers	`MUX_RESTRUCTURE`	On/Off/Auto
Safe State Machine	`SAFE_STATE_MACHINE`	On/Off
SDC Constraint Protection	`SYNTH_PROTECT_SDC_CONSTRAINT`	On/Off
Show Parameter Settings Tables in Synthesis Report	`SHOW_PARAMETER_SETTINGS_TABLES_IN_ SYNTHESIS_REPORT`	On/Off
State Machine Processing	`STATE_MACHINE_PROCESSING`	Auto/One-Hot/ Gray/Johnson/ Minimal Bits/ Sequential/ User-Encoded
Strict RAM Replacement	`STRICT_RAM_RECOGNITION`	On/Off
Synthesis Effort *(2)*	`SYNTHESIS_EFFORT`	Auto/Fast
Timing Driven Synthesis	`SYNTH_TIMING_DRIVEN_SYNTHESIS`	On/Off
Use LogicLock Constraints during Resource Balancing	`USE_LOGICLOCK_CONSTRAINTS_IN_BALANCING`	On/Off

Notes to Table 9–9:

(1) These settings are supported as Global and Instance settings, unless specified.

(2) This setting is only a Global setting.

Assigning a Pin

Use the following Tcl command to assign a signal to a pin or device location:

```
set_location_assignment -to <signal name> <location>
```

For example: `set_location_assignment -to data_input Pin_A3`

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 particular device.

Creating Design Partitions for Incremental Compilation

To create a partition, use the following command:

```
set_instance_assignment -name PARTITION_HIERARCHY \
<file name> -to <destination> -section_id <partition name>
```

The <*destination*> should be the entity's short hierarchy path. 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 9–74.

The <*partition name*> is the user-designated partition name, which must be unique and less than 1024 characters long. The name can consist only of alphanumeric characters, as well as pipe (`|`), colon (`:`), and underscore (`_`) characters. Altera recommends enclosing the name in double quotation marks (" ").

The <*file name*> is the name used for internally generated netlist files during incremental compilation. Netlists are named automatically by the Quartus II software based on the instance name if you create the partition in the GUI. If you are using 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 must 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`. For simplicity, Altera recommends that you base each file name on the corresponding instance name for the partition.

The software stores all netlists in the **db** compilation database directory.

Conclusion

The Quartus II software includes complete Verilog HDL and VHDL language support, as well as support for Altera-specific languages, making it an easy-to-use, standalone solution for Altera designs. You can use the synthesis options available in the software to help you improve your synthesis results, giving you more control over the way your design is synthesized. Use Quartus II reports and messages to analyze your compilation results.

Referenced Documents

This chapter references the following documents:

- *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*
- *Command-Line Scripting* chapter in volume 2 of the *Quartus II Handbook*
- *Designing With Low-Level Primitives User Guide*
- *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*
- *Introduction to the Quartus II Software*
- *Managing Quartus II Projects* chapter in volume 2 of the *Quartus II Handbook*
- *Netlist Optimizations and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analysis* chapter in volume 3 of the *Quartus II Handbook*

- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Scripting Reference Manual*
- *Quartus II Settings File Reference Manual*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*
- *Tcl Scripting* chapter in volume 2 of the *Quartus II Handbook*

Document Revision History

Table 9–10 shows the revision history for this chapter.

Table 9–10. Document Revision History (Part 1 of 2)

Date and Document Version	Changes Made	Summary of Changes
March 2009 v9.0.0	■ 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	Updated for Quartus II software version 9.0 release.

Table 9–10. Document Revision History (Part 2 of 2)

November 2008 v8.1.0	■ Changed page size to 8.5” × 11” ■ Restructured chapter by rearranging sections ■ Updated Figure 8–1 ■ Updated Table 8–9 ■ Added Example 8–23 and Example 8–28 ■ Updated the following sections: →“Setting Default Parameter Values and BDF Instance Parameter Values” →“Incremental Compilation” →“Quartus II Synthesis Options” →“Limiting DSP Block Usage in Partitions” →“Synthesis Effort” →“Using altera_attribute to Set Quartus II Logic Options” →“Quartus II Messages” ■ Added the following sections: →“Quartus II eXported Partition (.qxp) File as Source” →“Auto Gated Clock Conversion” →“Timing-Driven Synthesis” →“SDC Constraint Protection”	Updated for Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Adjusted the items listed in “System Verilog Support” ■ Added the section “VHDL wait Constructs and associated Examples” ■ Added the section “Limiting DSP Block Usage in Partitions” ■ Added the section “Synthesis Effort” ■ Added hyperlinks to referenced documents throughout the chapter ■ Minor editorial updates	Updated for Quartus II software version 8.0 release.

For previous versions of the *Quartus II Handbook*, refer to the Quartus II Handbook Archive.