

Introduction

As programmable logic designs (PLD) become more complex and require increased performance, advanced synthesis has become an important part of the design flow.

The Quartus® II software includes improved integrated synthesis that fully supports the Verilog hardware description language (HDL) and VHDL languages and provides options to control the synthesis process. With this synthesis support, the Quartus II software provides a complete, easy-to-use, standalone solution for system-on-a-programmable-chip (SOPC) designs.

This chapter also explains how to improve and control your Quartus II synthesis results by using Quartus II synthesis options; set other Quartus II options in your HDL source code; and follow Altera-recommended guidelines for writing state machines.

Verilog HDL & VHDL Support

The Quartus II software's integrated synthesis fully supports Verilog HDL and VHDL synthesizable language features, as well as some compiler directives and attributes.



For information on specific syntax features and language constructs, see *Quartus II Verilog HDL Support* and *Quartus II VHDL Support* in Quartus II Help.

Verilog HDL

The Quartus II Compiler's analysis and synthesis module supports the Verilog-1995 standard (IEEE Std. 1364-1995) and the Verilog-2001 standard (IEEE Std. 1364-2001) constructs. You can select which standard to use in the **Verilog version** section of the **Verilog HDL Input** page under the **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu). The Quartus II Compiler uses the Verilog-2001 standard by default.



The code samples provided in this document follow the Verilog-2001 standard.

Supported Verilog-2001 standard constructs include:

- Generate statements: `generate` and `genvar`
- `localparam` constants
- Pre-processor statements such as ``elsif`, ``line`, ``ifdef`, ``file`, and ``default_nettype`
- Signed declarations for all variables
- Operators such as `**`, `<<<`, and `>>>`
- Attributes using the syntax `(* name = value *)`
- Indexed part selects using `+` and `-`:
- Combinational logic sensitivity wild card token `@*`
- Combined port and data type declarations
- ANSI-style port lists
- In-line parameter passing by name (explicit re-definition using `#`)

Unsupported Verilog-2001 standard constructs include the following:

- Multi-dimensional arrays
- Libraries and configurations



See Quartus II Help for a complete listing of supported constructs.

The Quartus II software supports case-sensitive Verilog HDL code, in accordance with the Verilog HDL standard.

The Quartus II software supports the ``include` construct to include files with absolute paths (with either `/` or `\` as the separator), or relative paths (relative to project root or current file location). When searching for a relative path, the Quartus II software first searches relative to the project directory. If the software cannot find the file, it will search relative to the directory location of the file.

VHDL

The Quartus II Compiler's analysis and synthesis module supports the VHDL 1987 (IEEE Std. 1076-1987) and VHDL 1993 (IEEE Std. 1076-1993) standards. You can select which standard to use in the **VHDL version** section of the **VHDL Input** page under the **Analysis & Synthesis Settings** in the **Settings** dialog box (Assignments menu). The Quartus II Compiler uses the VHDL 1993 standard by default.



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

The Quartus II software supports VHDL libraries differently from the MAX+PLUS® II software or versions of the Quartus II software released before version 2.1. In the Quartus II software version 2.1 and later, standard IEEE and vendor VHDL libraries and packages can be called from VHDL code in the Quartus II software.

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

- 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 packages such as `maxplus2`, `altera_mf_components`, and `lpm_components` in the ALTERA library



For a complete listing of library and package support, see *Using Quartus II Packages* in the Quartus II Help.

The Quartus II software does not support user-defined precompiled libraries.

To call a user-defined VHDL package in the Quartus II software, indicate the library and package name using the `LIBRARY` and `USE` commands. You can use any name for your library, including `work`; therefore, you can use current software versions for projects developed with older versions of Altera software that used precompiled libraries without the need to modify any code. To compile using a VHDL package projects, include the VHDL package in your Quartus II project on the **Files** page of the **Settings** dialog box (Assignments menu). The package must be listed before other files that use the package because it must be analyzed by the Quartus II Compiler first.

Synthesis Options

The Quartus II software provides a number of options to guide the synthesis process and achieve optimal results. You can use compiler directives, attributes, and Quartus II logic options to control synthesis.



Versions of Quartus II software earlier than 2.1 did not support compiler directives or attributes; the software treated these options as comments. Quartus II behavior is different if designs compiled in earlier versions of the software included these synthesis options. You may need to change older code to take into account that the software recognizes these options.

This section defines three types of synthesis options: compiler directives, attributes, and Quartus II logic options. It also describes each of the following synthesis options:

- Translate off and on
- Read comments as HDL
- Full case
- Parallel case
- Keep combinational node/implement as output of logic cell
- Preserve registers
- Maximum fan-out
- Optimization technique
- State machine processing
- Preserve hierarchical boundary
- Power-up level
- Power-up don't care
- Remove duplicate logic
- Remove duplicate registers
- Remove redundant logic cells

Compiler Directives

The Quartus II software supports compiler directives, also called pragmas. You include compiler directives in Verilog HDL or VHDL code as comments. These directives are not Verilog HDL or VHDL commands; however, synthesis tools use them to drive the synthesis process in a particular manner. Other tools such as simulators ignore these directives and treat them as comments.

You can enter compiler directives in your code using the following syntax, where *directive* and *value* are variables, and the entry in brackets is optional.

Verilog HDL

```
// synthesis <directive> { <value> }  
or  
/* synthesis <directive> { <value> } */
```

VHDL

```
-- synthesis <directive> { <value> }
```

In addition to the `synthesis` keyword shown above, the keywords `pragma`, `synopsys`, and `exemplar` are supported in both Verilog HDL and VHDL for compatibility with other synthesis tools.

Attributes

The Quartus II software supports attributes, also known as pragmas or directives. Attributes are similar to compiler directives in that they drive the synthesis process. However, attributes always apply to a specific design element. Some attributes are also available as Quartus II logic options.

The Verilog-2001 and VHDL language definitions provide specific syntax for specifying attributes. However in Verilog-1995 HDL, you must use comments similar to compiler directives. You can enter attributes in your code using the following syntax, where *attribute*, *attribute type*, *value*, *object*, and *object type* are variables, and the entry in brackets is optional.

Verilog-1995 HDL

```
// synthesis <attribute> { <value> }
or
/* synthesis <attribute> { <value> } */
```



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.

In addition to the `synthesis` keyword as shown above, the keywords `pragma`, `synopsys`, and `exemplar` are supported for compatibility with other synthesis tools.

Verilog-2001 HDL

```
(* <attribute> { = <value> } *)
```

VHDL

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

In this chapter, the examples demonstrate each syntax form.

Quartus II Logic Options

Quartus II logic options control many aspects of the synthesis and place-and-route process. You can set logic options in the Quartus II graphical user interface (GUI) through the **Assignment Editor** (Assignments menu). Quartus II logic options allow you to set the associated attributes without editing the source HDL code.

Quartus II Synthesis Options

Table 8–1 lists the Quartus II synthesis options discussed in this section. Some options are simply compiler directives, some are only available as either attributes or logic options, and some are available as both attributes and logic options.



For information on using another Quartus II attribute to set options in your HDL (available as logic options), see *“Setting Other Quartus II Options in Your HDL Source Code”* on page 8–17.

Synthesis Option	Compiler Directive	Attribute	Quartus II Logic Option
Translate off and on	translate_off translate_on	–	–
Read comments as HDL	read_comments_as_HDL	–	–
Full case	–	full_case	–
Parallel case	–	parallel_case	–
Keep combinational node/implement as output of logic cell	–	keep syn_keep	Implement as output of logic cell
Preserve registers	–	preserve syn_preserve	Preserve registers
Maximum fan-out	–	maxfan syn_maxfan	Maximum fan-out
Optimization technique	–	–	Optimization technique
State machine processing	–	–	State machine processing
Preserve hierarchical boundary	–	–	Preserve hierarchical boundary
Power-up level	–	–	Power-up level
Power-up don't care	–	–	Power-up don't care
Remove duplicate logic	–	–	Remove duplicate logic

Table 8–1. Synthesis Options as Compiler Directives, Attributes, & Logic Options (Part 2 of 2)

Synthesis Option	Compiler Directive	Attribute	Quartus II Logic Option
Remove duplicate registers	–	–	Remove duplicate registers
Remove redundant logic cells	–	–	Remove redundant logic cells



Because Verilog HDL is case-sensitive, compiler directives and attributes are also case sensitive.

The following sections provide more information on each option shown in [Table 8–1](#).

Translate Off & On

The `translate_off` and `translate_on` compiler 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. 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. The following are examples of these directives.

Verilog HDL Example of Translate Off & On

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

#tpd;
// synthesis translate_on
```

VHDL Example of Translate Off & On

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

Read Comments as HDL

The `read_comments_as_HDL` compiler 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 the 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.

In the following examples, the commented code enclosed by `read_comments_as_HDL` is visible to the Quartus II Compiler and is synthesized.



Because compiler directives are case-sensitive in Verilog HDL, you must match the case of the directive, as shown below.

Verilog HDL Example of Read Comments as HDL

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

VHDL Example of 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
```

Full Case

A Verilog HDL case statement is considered full when all possible binary combinations of cases are specified or a default case is present. A `full_case` attribute attached to a case statement header that is not full forces the unspecified states to be treated as logic “don't care” values. Using this attribute on a case statement that is not full avoids the latch inference problems discussed in the *Design Guidelines* section in Volume 1 of the *Quartus II Handbook*. VHDL case statements must be full, so the attribute does not apply.

When using the `full_case` attribute, there is a potential cause for simulation-mismatch between Verilog HDL functional and post-Quartus II simulation because unknown case statement cases may still function like latches during functional simulation. For example, a

simulation mismatch may occur with the code in figures 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 the following example 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`.

Sample Verilog HDL Code with a `full_case` Attribute

```
endmodule 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
```

Verilog-2001 syntax also accepts the following statements in the case header instead of the comment form shown in the example above.

```
(* 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. A `parallel_case` attribute attached to a case statement header forces the synthesis tool to generate multiplexer logic instead of a priority encoder. VHDL case statements have no overlap in any of the case items, so they are always parallel and this attribute does not apply.

Use this attribute only when the case statement is truly parallel and you want to use one-hot style encoding. If you use the attribute in any other situation, the generated logic will not match the functional HDL simulation.



Altera recommends that you avoid use of the `parallel_case` attribute, due to the possibility of introducing mismatches between Verilog HDL functional and post-Quartus II simulation.

The following example shows a `casez` statement with non-parallel cases. Functional HDL simulation behaves as a priority encoder with bits of `sel` having high-to-low priority order `sel[2]`, `sel[1]`, and `sel[0]`. However, the synthesized version of this design may simulate differently because it is implemented with multiplexer logic.

Sample Verilog HDL Code with 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
end
```

Verilog-2001 syntax also accepts the following statements in the `case` header instead of the comment form shown in the example above.

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

Keep Combinational Node/Implement as Output of Logic Cell

This attribute and corresponding logic option direct the Compiler to keep a wire or net configuration intact during logic synthesis minimizations and netlist optimizations. When this attribute is applied, the Compiler will insert LCELL buffers in the design to maintain the node name. A wire that has a `keep` attribute or **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. You can use this directive to make combinational nodes visible to the SignalTap® II logic analyzer.

The option cannot keep nodes that have no fan-out.

You 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 below. In this example, the Compiler maintains the node name `my_wire` or `my_signal`.



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

Verilog HDL

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

Verilog-2001

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

VHDL

```
signal my_signal: bit;
```

```
attribute syn_keep: boolean;
attribute syn_keep of my_signal: signal is true;
```

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 can preserve a register so you can observe it during simulation or with the SignalTap II 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 will remove one of the two duplicate registers in this case; the `preserve` attribute can be added to both registers to prevent this.

The option cannot preserve registers that have no fan-out.

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 below. In this example, 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.

Verilog HDL

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

Verilog-2001

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

VHDL

```
signal my_reg : stdlogic;
```

```
attribute preserve : boolean;
```

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



Setting the **Preserve Registers** logic option does not affect registers that are removed during the analysis and elaboration stage of compilation (before logic synthesis). To fully preserve the register throughout compilation, use the HDL attribute instead of the logic option.

Maximum Fan-Out

This attribute and logic option directs the Compiler to control the number of destinations fed by a node. The fan-out count of the node will not exceed the value specified for the maximum number of fan-out. You can apply this option to a register or a logic cell buffer. This option is useful for reducing the load of critical signals, which can improve performance. Additionally, you can use this option to instruct the Compiler to duplicate or replicate a register that feeds nodes in different locations on the target device. Duplicating the register may allow the PowerFit™ 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®, ACEX® 1K, and Mercury™ devices.



If you have enabled any of the Quartus II netlist optimizations that affect registers, add a `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 re-timing.



For details on netlist optimizations, see the *Netlist Optimization & 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, or you can set the `maxfan` attribute in your HDL code as shown below. In this example, 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.

Verilog HDL

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

Verilog-2001

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

VHDL

```
signal clk_gen : stdlogic;

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

Optimization Technique

This logic option specifies the goal for logic optimization, i.e., whether to attempt to achieve maximum speed performance or minimum area usage, or a balance between the two, during compilation. [Table 8-2](#) lists the settings for this option, which you can only apply to a design entity.

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	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.

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


You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box (Assignments menu).

State Machine Processing

This logic option specifies the processing style used to compile a state machine. [Table 8-3](#) lists the settings for this option, which you can apply to a state machine name or to a design entity containing a state machine.

Table 8-3. State Machine Processing Settings	
Setting	Description
Auto (Default)	Allows the Compiler to choose 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.
User-Encoded	Encodes the state machine in the manner specified by the user.

The default state machine encoding, Auto, is one-hot encoding for FPGA devices and minimal-bits encoding for complex programmable logic devices (CPLDs).

 See **“State Machine Guidelines”** on [page 8-21](#) for guidelines to ensure that your state machine is inferred and encoded correctly.

You can also set this option for your whole project on the **Analysis & Synthesis Settings** page in the **Settings** dialog box (Assignments menu).

Preserve Hierarchical Boundary

This logic option determines how strictly the hierarchical boundaries between design entities should be maintained during logic synthesis. [Table 8-4](#) lists the settings for the option, which you can only apply to a design entity. Lower-level entities do not inherit their parent entity's setting for this option.

Table 8-4. Preserve Hierarchical Boundary Settings	
Setting	Description
Off	Completely ignores boundaries and therefore allows unlimited optimization. This setting provides the greatest logic minimization.

Table 8–4. Preserve Hierarchical Boundary Settings

Setting	Description
Relaxed	Allows only partial cross-boundary optimization, which may reduce the compilation time. Non-trivial inputs and outputs of the entity are visible during simulation and timing analysis.
Firm	Strictly maintains hierarchical boundaries. This setting may increase compilation time, increase logic cell count, and negatively affect design performance.

The **Relaxed** setting means that the Compiler preserves hierarchical boundaries. However, certain signals such as VCC and GND are propagated and optimized through the boundaries. The **Firm** setting does not allow optimization across boundaries, and keeps each hierarchical block separate.

Power-Up Level

This logic option causes a register (flipflop) to power up with the specified logic level, either High (1) or Low (0). You can apply this option to any register or to a pin with the logic configurations described as follows:

- 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 intervening 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 intervening logic, other than inversion, between the register and the pin
 - The register does not fan out to any other logic

For the register to power up to with the specified logic level, the Compiler may perform NOT Gate Push-Back on the register.

Power-Up Don't Care

This logic option causes registers to power up with a “don't care” logic level (X), or the logic level most appropriate for the design. You might use this option to allow the Compiler to change the power-up condition of a register to minimize your design's area usage. This option is turned ON by default.

For example, a register may have its D input tied to VCC. If you turned this option off, the register powers up low even though it goes high at the first clock signal. If you turned this option on, the Compiler sets the power-up value of the register to high and, therefore, can eliminate the register and connect the output of the register to VCC. If the Compiler makes this type of optimization, it produces 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.



Versions of the Quartus II software earlier than version 2.1 did not include this option. If you compile an older design that relies on registers to power-up to a specific level, the Compiler may synthesize the design differently. Turn off the **Power-Up Don't Care** option if you want your design to use the power-up behavior of older versions of Quartus II software.

Remove Duplicate Logic

If you turn on this option, the Compiler removes logic, if it is identical to other logic in the design. If two functions generate the same logic, the Compiler removes the second one, and the first one fans out to the second one's destinations. Additionally, if the deleted logic function has different logic option assignments, the Compiler ignores them. This option is turned on by default.

When turned on, this option also removes all duplicate registers, such as the **Remove Duplicate Registers** option. If you do not want the Compiler to remove certain registers when this option is turned on, turn off the **Remove Duplicate Registers** option for those registers. See [Table 8-5](#) for more details.

Even if you turn this option on, the Compiler does not remove duplicate logic that you inserted deliberately. If a function's output feeds an LCELL buffer, the Compiler always treats it as a unique signal and the **Remove Duplicate Logic** option does not apply (i.e., the Compiler does not remove an LCELL buffer if you turn on this option).

Remove Duplicate Registers

If you turn on this logic option, it removes a register if it is 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.

The Compiler only recognizes this option if you turned on the **Remove Duplicate Logic** option. When turned on, the **Remove Duplicate Logic** option also removes duplicate registers. Therefore, you should use the this option only if you want to prevent the Compiler from removing duplicate registers that you have used deliberately. That is, you should use this option only with the **Off** setting. See [Table 8-5](#). You can apply this option to an individual register or a design entity that contains registers.

Table 8-5. Settings for Remove Duplicate Logic & Remove Duplicate Registers

Remove Duplicate Logic Setting	Remove Duplicate Registers Setting	Description
On (Default)	On (Default)	Removes logic (including registers) if it is identical to other logic in the design.
On	Off	Preserves all registers for which the Remove Duplicate Registers option is turned off. Removes logic (including any other registers) if it is identical to other logic in the design.
Off	On or Off	Preserves duplicate logic and registers.

Remove Redundant Logic Cells

This logic option removes redundant LCELL primitives or WYSIWYG cells. If you turn this option **On**, the Compiler optimizes a circuit for area and speed. The project-wide option is turned **Off** by default.

Setting Other Quartus II Options in Your HDL Source Code

This section describes two Quartus II attributes--Altera attribute and chip pin--that can be used to set other Quartus II options and settings in your HDL source code.

Altera Attribute

This attribute enables you to make any number of Quartus II options and assignments on an object (entity, instance, or net) in your HDL source code. With `altera_attribute`, you can control synthesis options from your HDL source even when the options lack a specific HDL attribute (such as the logic options presented earlier in this chapter). You can also

use this attribute to pass option settings and assignments to phases of the Compiler flow beyond Analysis & Synthesis, such as Fitting. The syntax for setting this attribute is the syntax defined in the section [“Attributes” on page 8–5](#) for HDL attributes (examples are provided below).

Assignments made with the Altera Attribute take precedence over assignments made through the Quartus II user interface, the Quartus Settings File (.qsf), and the Tcl interface.

The attribute takes a single string argument containing a list of QSF assignments separated by semicolons, as follows:

```
altera_attribute =  
"<variable_1>=<value_1>;<variable_2>=<value_2>;..."
```

If a variable's assigned value is a string of text, you must use escaped quotes around the value, as in the following example (using non-existent variable and value terms):

```
altera_attribute = "VARIABLE_NAME=\"STRING_VALUE\""
```

To find the QSF variable name, you can make the setting in the user interface and then note the changes in the QSF file.

The following examples use `altera_attribute` to set the power-up level of an inferred register. Note that 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 automatically moves the attribute to the inferred instance.

Verilog-1995 Example of Applying Altera Attribute on an Instance

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

Verilog-2001 Example of Applying Altera Attribute on an Instance

```
reg my_reg (* altera_attribute =  
"POWER_UP_LEVEL=HIGH" *)
```

VHDL Example of Altera Attribute

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

The following examples use the `altera_attribute` to disable the **Auto Shift Register Replacement** synthesis option on an entity.

Verilog-1995 Example of Applying Altera Attribute on an Entity

```
module my_entity(...) /* synthesis altera_attribute =
"AUTO_SHIFT_REGISTER_RECOGNITION=OFF" */;
```

Verilog-2001 Example of Applying Altera Attribute on an Entity

```
module my_entity(...) (*altera_attribute =
"AUTO_SHIFT_REGISTER_RECOGNITION=OFF" *);
```

VHDL Example of Applying Altera Attribute on 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
"AUTO_SHIFT_REGISTER_RECOGNITION=OFF";

begin
    -- The architecture body
end rtl;
```

To apply the Altera Attribute to a VHDL entity, you must set the attribute on its architecture rather than on the entity itself.

Chip Pin

This attribute enables you to assign pins to the ports of an entity or module in your HDL source. You may only assign pins 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.

The following examples show different ways of assigning input pin `my_pin1` and `my_pin2` to Pin 4 on a target device.

Verilog-1995 Example of Applying Chip Pin to a Single Pin

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

Verilog-2001 Example of Applying Chip Pin to a Single Pin

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

VHDL Example of 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.

The following examples assign my_pin[2] to Pin_4, my_pin[1] to Pin_5, and my_pin[0] to Pin_6.

Verilog-1995 Example of Applying Chip Pin to a Bus of Pins

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

Verilog-2001 Example of Applying Chip Pin to Part of a Bus of Pins

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

The following example 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.

VHDL Example of Applying Chip Pin to 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, 5, 6";
```

Megafunction Inference Control

The Quartus II Compiler automatically recognizes certain types of HDL code and infers the appropriate megafunction when a megafunction provides optimal results. That is, the software uses the Altera megafunction code when compiling your design even though you did not specifically instantiate the megafunction. The software uses inference because the megafunctions are optimized for Altera devices, so the area and/or performance may be better than generic HDL code. Additionally, you must use megafunctions to access certain architecture-specific

features, such as RAM, digital signal processing (DSP) blocks, and shift registers, that generally provide improved performance compared with basic logic elements.



For details on coding style recommendations when targeting megafunctions in Altera devices, see 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 paragraphs.

Multiply-Accumulators & 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 (Assignment menu), or disable the option for a specific block using the **Assignment Editor** (Assignments menu).

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 (Assignment menu), or disable the option for a specific block using the **Assignment Editor** (Assignments menu).

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 (Assignments menu), or for a specific block using the **Assignment Editor**.



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.

State Machine Guidelines

The Quartus II software can recognize and encode Verilog HDL and VHDL state machines during synthesis. This section explains the Quartus II support for state machines and presents guidelines to ensure the best results.

The default state machine encoding in the Quartus II software is one-hot encoding for FPGA devices and minimal-bits encoding for CPLD devices. While these settings achieve the best results on average, another encoding style might be more appropriate for your design. You can modify the encoding style by setting the **State Machine Processing** logic option to another value. This option can be set for your whole project on the **Analysis & Synthesis Settings** page of the **Settings** dialog box (Assignments menu), or using the logic option described in the [“State Machine Processing” on page 8–14](#). The available encoding styles for state machines are: “Auto”, “One-Hot”, “Minimal-Bits”, and “User-Encoded”.

To ensure proper recognition and inference of state machines and to improve performance, Altera recommends that you observe the following guidelines (which apply to both Verilog HDL and VHDL):

- Assign default values to outputs derived from the state machine to avoid generation of unwanted latches during synthesis.
- Assign a default clause to direct the state machine in case it accidentally reaches an unused state.
- Separate the state machine logic from all arithmetic functions and data paths, including assigning output values.
- If your design contains an operation that is used by more than one state, define the operation outside the state machine and make the output logic of the state machine use this value.
- Use a simple asynchronous or synchronous reset to ensure a defined power-up state. If your state machine design contains more elaborate reset logic, such as an asynchronous reset and an asynchronous load at the same time, the Quartus II software will generate regular logic rather than infer a state machine.

See the following sections for additional guidelines and coding examples using [“Verilog HDL State Machines” on page 8–22](#) and [“VHDL State Machines” on page 8–25](#).

Verilog HDL State Machines

To ensure proper recognition and inference of Verilog HDL state machines, observe the following additional Verilog-specific guidelines:

- Represent the states in a state machine by the `parameter` data types and use the parameters to make state assignments. This implementation makes the state machine easier to read and reduces the risks of errors during coding.



Altera recommends against the direct use of integer values for state variables such as `next_state <= 0` (but integer use does not prevent inference).

- No state machine is inferred if the state transition logic uses arithmetic as follows:

```
case (state)
  0: begin
    if (ena) next_state <= state + 2;
    else next_state <= state + 1;
  end
  1: begin
    ...
  end
endcase
```

- No state machine is inferred if the state variable is used to create an output as follows:

```
output out1
case (state)
  state_0: begin
    if (ena) out1 <= state_1;
    else out1 <= state_2;
    next_state <= state_2;
  end
  state_1: begin
    ...
  end
endcase
```

Verilog HDL State Machine Coding Example

The following module `verilog_fsm` is an example of a typical Verilog HDL state machine implementation.

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in_1` and `in_2` is used as an output of the state machine in the states `state_1` and `state_2`. The difference of `in_1` and `in_2` is used in the states `state_1` and `state_3`. The temporary variables `tmp_out_0` and `tmp_out_1` are used to store the sum and the difference of `in_1` and `in_2`. The use of these temporary variables in the various states of the state machine ensures proper resource sharing between these mutually exclusive states.

Example State Machine in Verilog HDL

```
module verilog_fsm (clk, reset, in_1, in_2, out);
```

```
input clk;
input reset;
input [3:0] in_1;
input [3:0] in_2;

parameter state_0 = 3'b000;
parameter state_1 = 3'b001;
parameter state_2 = 3'b010;
parameter state_3 = 3'b011;
parameter state_4 = 3'b100;

reg [4:0] tmp_out_0, tmp_out_1, tmp_out_2;
reg [2:0] state, next_state;

always @(posedge clk or posedge reset)
begin
    if (reset)
        state <= state_0;
    else
        state <= next_state;
end

always @(state or in_1 or in_2)
begin
    tmp_out_0 <= in_1 + in_2;
    tmp_out_1 <= in_1 - in_2;

    case (state)
        state_0:begin
            tmp_out_2 <= in_1 + 5'b00001;
            next_state <= state_1;
        end
        state_1:begin
            if (in_1 < in_2) begin
                next_state <= state_2;
                tmp_out_2 <= tmp_out_0;
            end
            else begin
                next_state <= state_3;
                tmp_out_2 <= tmp_out_1;
            end
        end
        state_2:begin
            tmp_out_2 <= tmp_out_0 - 5'b00001;
            next_state <= state_3;
        end
        state_3:begin
            tmp_out_2 <= tmp_out_1 + 5'b00001;
            next_state <= state_4;
        end
        state_4:begin
```

```

        tmp_out_2 <= in_2 + 5'b00001;
        next_state <= state_0;
    end
    default:begin
        tmp_out_2 <= 5'b00000;
        next_state <= state_0;
    end
endcase
end
assign out = tmp_out_2;
endmodule

```

An equivalent implementation of this state machine could be achieved by using ``define` instead of the parameter data type, as follows:

```

`define state_0 3'b000
`define state_1 3'b001
`define state_2 3'b010
`define state_3 3'b011
`define state_4 3'b100

```

In this case, the state and `next_state` assignments are assigned a ``state_0` instead of a `state_0`, as follows:

```
next_state <= `state_3;
```

Although the ``define` construct is supported, Altera strongly recommends the use of the parameter data type as it conserves the state names throughout synthesis.

VHDL State Machines

To ensure proper recognition and inference of VHDL state machines, represent the states in a state machine by enumerated types and use the corresponding types to make state assignments. This implementation makes the state machine easier to read and reduces the risks of errors during coding. If the state is not represented by an enumerated type, the Quartus II software will not recognize the state machine. Instead, it will be implemented as regular logic gates and registers, and it will not be listed as a state machine in the Analysis & Synthesis report.

The state assignments created automatically by the Quartus II software can be overwritten by using specific state assignments with the `ENUM_ENCODING` attribute. The `ENUM_ENCODING` attribute must follow the associated type declaration and precede any associated signal declarations. To use the `ENUM_ENCODING` attribute during compilation, set the **State Machine Processing** logic option to User-Encoded on the **Analysis & Synthesis Settings** page of the **Settings** dialog box

(Assignments menu), or by using the **Assignment Editor** (Assignments menu). For more information, see the *Manually Specifying State Assignments* topic in the Quartus II Help.

VHDL State Machine Coding Example

The following entity `vhdl_fsm` is an example of a typical VHDL state machine implementation.

This state machine has five states. The asynchronous reset sets the variable `state` to `state_0`. The sum of `in_1` and `in_2` is used as an output of the state machine in the states `state_1` and `state_2`. The difference (`in1-in2`) is also used in the states `state_1` and `state_2`. The temporary variables `tmp_out_0` and `tmp_out_1` are used to store the sum and the difference of `in_1` and `in_2`. The use of these temporary variables in the various states of the state machine ensures the proper resource sharing between these mutually exclusive states.

Example State Machine in VHDL

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

ENTITY vhdl_fsm IS
  PORT (
    clk: IN STD_LOGIC;
    reset: IN STD_LOGIC;
    in1: IN STD_LOGIC_VECTOR(4 downto 0);
    in2: IN STD_LOGIC_VECTOR(4 downto 0);
    out_1: OUT STD_LOGIC_VECTOR(4 downto 0)
  );
END vhdl_fsm;

ARCHITECTURE rtl OF vhdl_fsm IS
  TYPE Tstate IS (state_0, state_1, state_2, state_3, state_4);

  SIGNAL tmp_out_0: STD_LOGIC_VECTOR(4 downto 0);
  SIGNAL tmp_out_1: STD_LOGIC_VECTOR(4 downto 0);
  SIGNAL state: Tstate;
  SIGNAL next_state: Tstate;

BEGIN
  PROCESS(clk, reset)
  BEGIN
    IF reset = '1' THEN
      state <= state_0;
    ELSIF rising_edge(clk) THEN
      state <= next_state;
    END IF;
  END PROCESS;
END rtl;
```

```

PROCESS (state, in1, in2, tmp_out_0, tmp_out_1)
BEGIN
    tmp_out_0 <= STD_LOGIC_VECTOR'(UNSIGNED(in1)+UNSIGNED(in2));
    tmp_out_1 <= STD_LOGIC_VECTOR'(UNSIGNED(in1)-UNSIGNED(in2));

    CASE state IS
        WHEN state_0 =>
            out_1 <= in1;
            next_state <= state_1;
        WHEN state_1 =>
            IF (in1 < in2) then
                next_state <= state_2;
                out_1 <= tmp_out_0;
            ELSE
                next_state <= state_3;
                out_1 <= tmp_out_1;
            END IF;
        WHEN state_2 =>
            IF (in1 < "0100") then
                out_1 <= tmp_out_0;
            ELSE
                out_1 <= tmp_out_1;
            END IF;
            next_state <= state_3;
        WHEN state_3 =>
            out_1 <= "11111";
            next_state <= state_4;
        WHEN state_4 =>
            out_1 <= in2;
            next_state <= state_0;
        WHEN OTHERS =>
            out_1 <= "00000";
            next_state <= state_0;
    END CASE;
END PROCESS;

END rtl;

```

Conclusion

The Quartus II software includes complete Verilog HDL and VHDL language support, making it an easy-to-use, standalone solution for SOPC designs. This document describes methodologies and guidelines that you can use to improve synthesis results and obtain optimum performance in your target Altera device.

