5. Design Recommendations for Altera Devices and the Quartus II Design Assistant

QII51006-9.1.0

Introduction

Current FPGA applications have reached the complexity and performance requirements of ASICs. In the development of such complex system designs, good design practices have an enormous impact on your device's timing performance, logic utilization, and system reliability. Well-coded designs behave in a predictable and reliable manner even when re-targeted to different families or speed grades. Good design practices also aid in successful design migration between FPGA and HardCopy® or ASIC implementations for prototyping and production.

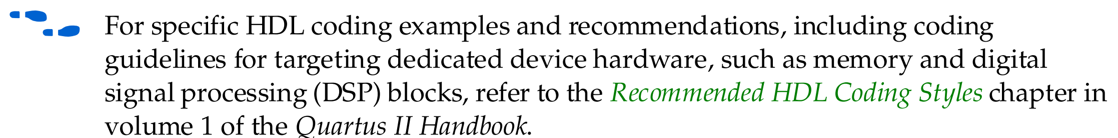
For optimal performance, reliability, and faster time-to-market when designing with Altera® devices, adhere to the following guidelines:

- Understand the impact of synchronous design practices
- Follow recommended design techniques including hierarchical design partitioning
- Take advantage of the architectural features in the targeted device

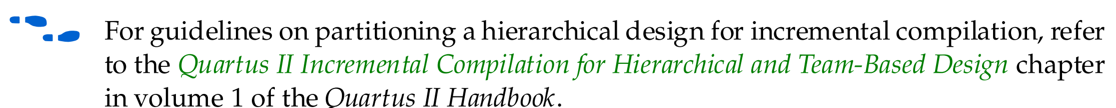
This chapter presents design recommendations in these areas and describes the Quartus® II Design Assistant that can help you check your design for violations of design recommendations.

This chapter contains the following sections:

- "Synchronous FPGA Design Practices" on page 5–2
- "Design Guidelines" on page 5–4
- "Checking Design Violations Using the Design Assistant" on page 5–13
- "Targeting Clock and Register-Control Architectural Features" on page 5–39
- "Targeting Embedded RAM Architectural Features" on page 5–41


For specific HDL coding examples and recommendations, including coding guidelines for targeting dedicated device hardware, such as memory and digital signal processing (DSP) blocks, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

For information about migrating designs to HardCopy devices, refer to the *Design Guidelines for HardCopy Series Devices* chapter in volume 1 of the *HardCopy Series Handbook*.


For guidelines on partitioning a hierarchical design 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*.

Synchronous FPGA Design Practices

The first step in good design methodology is to understand the implications of your design practices and techniques. This section outlines some of the benefits of optimal synchronous design practices and the hazards involved in other techniques. Good synchronous design practices can help you meet your design goals consistently. Problems with other design techniques can include reliance on propagation delays in a device, incomplete timing analysis, and possible glitches.

In a synchronous design, a clock signal triggers all events. As long as all the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. You can easily target synchronous designs to different device families or speed grades. In addition, synchronous design practices help ensure successful migration if you plan to migrate your design to a high-volume solution such as an Altera HardCopy device or if you are prototyping an ASIC.

Fundamentals of Synchronous Design

In a synchronous design, everything is related to the clock signal. On every active edge of the clock (usually the rising edge), the data inputs of registers are sampled and transferred to outputs. Following an active clock edge, the outputs of combinational logic feeding the data inputs of registers change values. This change triggers a period of instability due to propagation delays through the logic as the signals go through a number of transitions and finally settle to new values. Changes happening on data inputs of registers do not affect the values of their outputs until the next active clock edge.

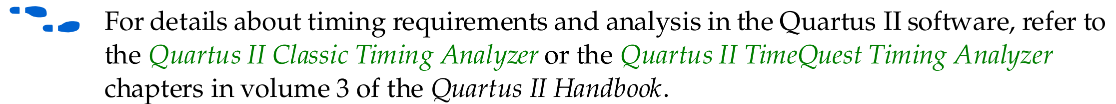
Because the internal circuitry of registers isolates data outputs from inputs, instability in the combinational logic does not affect the operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input has been stable for at least the setup time of the register
- After an active clock edge, the data input remains stable for at least the hold time of the register

When you specify all of your clock frequencies and other timing requirements, the Quartus II Classic Timing Analyzer reports actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin of your design. By meeting these external pin requirements and following synchronous design techniques, you ensure that you satisfy the setup and hold times for all registers within the Altera device.

To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feeds a register should have a synchronous relationship with the clock of the register. If signals are asynchronous, you can register the signals at the input of the Altera device to help prevent a violation of the required setup and hold times.

When the setup or hold time of a register is violated, the output can be set to an intermediate voltage level between the high and low levels, called a metastable state. In this unstable state, small perturbations such as noise in power rails can cause the register to assume either the high or low voltage level, resulting in an unpredictable valid state. Various undesirable effects can occur, including increased propagation delays and incorrect output states. In some cases, the output can even oscillate between the two valid states for a relatively long period of time.



For details about timing requirements and analysis in the Quartus II software, refer to the *Quartus II Classic Timing Analyzer* or the *Quartus II TimeQuest Timing Analyzer* chapters in volume 3 of the *Quartus II Handbook*.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in programmable logic device (PLD) designs, enabling them to take “short cuts” to save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays in a device, which can result in incomplete timing constraints and possible glitches and spikes. Because current FPGAs provide many high-performance logic gates, registers, and memory, resource and performance trade-offs have changed. Now it is more important to focus on design practices that help you meet design goals consistently than to save device resources using problematic asynchronous techniques.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. In these cases, race conditions can arise where the order of signal changes can affect the output of the logic. PLD designs can have varying timing delays, depending on how the design is placed and routed in the device with each compilation. Therefore, it is almost impossible to determine the timing delay associated with a particular block of logic ahead of time. As devices become faster because of device process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Specific examples are provided in “Design Guidelines” on page 5–4. Relying on a particular delay also makes asynchronous designs very difficult to migrate to different architectures, devices, or speed grades.

The timing of asynchronous design structures is often difficult or impossible to model with timing assignments and constraints. If you do not have complete or accurate timing constraints, the timing-driven algorithms used by your synthesis and place-and-route tools may not be able to perform the best optimizations and the reported results may not be complete.

Some asynchronous design structures can generate harmful glitches, which are pulses that are very short compared with clock periods. Most glitches are generated by combinational logic. When the inputs of combinational logic change, the outputs exhibit a number of glitches before they settle to their new values. These glitches can propagate through the combinational logic, leading to incorrect values on the outputs in asynchronous designs. In a synchronous design, glitches on the data inputs of registers are normal events that have no negative consequences because the data is not processed until the clock edge.

Design Guidelines

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL design techniques and what results to expect. Your design techniques can affect logic utilization and timing performance, as well as the design's reliability. This section describes some basic design techniques that ensure optimal synthesis results for designs targeted to Altera devices while avoiding several common causes of unreliability and instability. Design your combinational logic carefully to avoid potential problems and pay attention to your clocking schemes so you can maintain synchronous functionality and avoid timing problems.

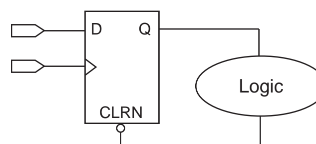
Combinational Logic Structures

Combinational logic structures consist of logic functions that depend only on the current state of the inputs. In Altera FPGAs, these functions are implemented in the look-up tables (LUTs) of the device's architecture, using either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can also be used to implement part of the logic function to save LUT resources. By following the recommendations in this section, you can improve the reliability of your combinational design.

Combinational Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs. They should be avoided whenever possible. In a synchronous design, feedback loops should include registers. Combinational loops generally violate synchronous design principles by establishing a direct feedback loop that contains no registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side in HDL code. A combinational loop also occurs when you feed back the output of a register to an asynchronous pin of the same register through combinational logic, as shown in Figure 5–1.

Figure 5–1. Combinational Loop through Asynchronous Control Pin



Use recovery and removal analysis to perform timing analysis on asynchronous ports such as `clear` or `reset` in the Quartus II software.

- If you are using the Classic Timing Analyzer, on the Assignments menu, click **Settings**. In the **Settings** dialog box, under **Timing Analysis Settings**, select **Classic Timing Analyzer Settings**. On the **Classic Timing Analyzer Settings** page, click **More Settings** and turn on the **Enable Recovery/Removal Analysis** option.
- If you are using the TimeQuest Timing Analyzer, refer to the "Recovery and Removal" section in the *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* for details about how the TimeQuest Timing Analyzer performs recovery and removal analysis.

 © November 2009 Altera Corporation

Combinational loops are inherently high-risk design structures for the following reasons:

- Combinational loop behavior generally depends on relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change, which means the behavior of the loop is unpredictable.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinational loops to process the design. The various tools used in the design flow may open a given loop in a different manner, processing it in a way that is inconsistent with the original design intent.

Latches

A latch is a small circuit with combinational feedback that holds a value until a new value is assigned. Latches can be implemented directly with primitives, using `LPM_LATCH`, or inferred from HDL code. It is common for mistakes in HDL code to cause unintended latch inference. Quartus II Synthesis issues a warning message if this occurs.

Unlike other technologies, a latch in an FPGA architecture is not significantly smaller than a register. The architecture is not optimized for latch implementation and latches generally have slower timing performance compared to equivalent registered circuitry.

Latches have a transparent mode in which data flows continuously from input to output. A positive latch is in transparent mode when the enable signal is high (low for negative latch). In transparent mode, glitches on the input can pass through the output because of the direct path created. This presents significant complexity for timing analysis. Typical latch schemes use multiple enable phases to prevent long transparent paths from occurring. However, timing analysis is generally not able to identify these safe applications.

The Quartus II software setting **Analyze latches as Synchronous Elements** allows you to treat latches as having nontransparent start and end points. Bear in mind that even an instantaneous transition through transparent mode can lead to glitch propagation. The Quartus II software does not perform cycle-borrowing analysis, such as that performed by third-party timing analysis tools (such as the Synopsys PrimeTime software).

Due to various timing complexities, latches have limited support in formal verification tools. Therefore, it is very important that you do not use latches when using formal verification.

Altera recommends you avoid using latches to ensure that you can completely analyze the timing performance and reliability of your design.

Delay Chains

Delay chains occur when two or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Inverters are often chained together to add delay. Delay chains are sometimes used to resolve race conditions created by other asynchronous design practices.

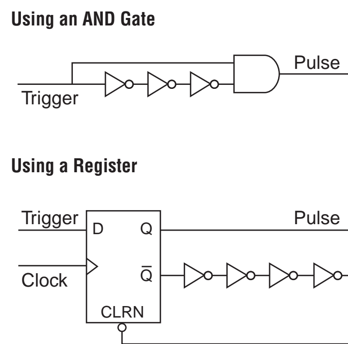
Delays in PLD designs can change with each place-and-route cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Refer to "Hazards of Asynchronous Design" on page 5–3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kind of problems.

In some ASIC designs, delays are used for buffering signals as they are routed around the device. This functionality is not required in FPGA devices because the routing structure provides buffers throughout the device.

Pulse Generators and Multivibrators

Delay chains are sometimes used to generate either one pulse (pulse generators) or a series of pulses (multivibrators). There are two common methods for pulse generation, as shown in Figure 5–2. These techniques are purely asynchronous and need to be avoided.

Figure 5–2. Asynchronous Pulse Generators



In "Using an AND Gate" (Figure 5–2), a trigger signal feeds both inputs of a 2-input AND gate, but the design inverts or adds a delay chain to one of the inputs. The width of the pulse depends on the relative delays of the path that feeds the gate directly and the one that goes through the delay. This is the same mechanism responsible for the generation of glitches in combinational logic following a change of input values. This technique artificially increases the width of the glitch by using a delay chain.

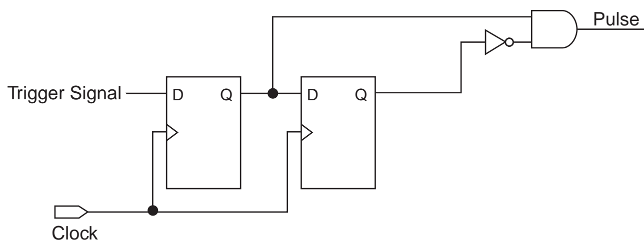
In "Using a Register" (Figure 5–2), a register's output drives the same register's asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

The width of pulses generated in this way are difficult for synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined after placement and routing, when routing and propagation delays are known. You cannot reliably determine the width of the pulse when creating HDL code and it cannot be set by EDA tools. The pulse may not be wide enough for the application under all PVT conditions. Also, the pulse width changes if you change to a different device. In addition, static timing analysis cannot be used to verify the pulse width, so verification is very difficult.

Multivibrators use a glitch generator to create pulses, together with a combinational loop that turns the circuit into an oscillator. This creates additional problems because of the number of pulses involved. In addition, when the structures generate multiple pulses, they also create a new artificial clock in the design that has to be analyzed by the design tools.

When you must use a pulse generator, use synchronous techniques, as shown in Figure 5–3.

Figure 5–3. Recommended Pulse-Generation Technique



In this design, the pulse width is always equal to the clock period. This pulse generator is predictable, can be verified with timing analysis, and is easily moved to other architectures, devices, or speed grades.

Clocking Schemes

Like combinational logic, clocking schemes have a large effect on your design's performance and reliability. Avoid using internally generated clocks wherever possible because they can cause functional and timing problems in the design. Clocks generated with combinational logic can introduce glitches that create functional problems and the delay inherent in combinational logic can lead to timing problems.

Specify all clock relationships in the Quartus II software to allow for the best timing-driven optimizations during fitting and to allow correct timing analysis. Use clock setting assignments on any derived or internal clocks to specify their relationship to the base clock.

Altera recommends using global device-wide, low-skew dedicated routing for all internally-generated clocks, instead of routing clocks on regular routing lines. For a detailed explanation, refer to "Clock Network Resources" on page 5–40.

Avoid data transfers between different clocks wherever possible. If you require a data transfer between different clocks, use FIFO circuitry. You can use the clock uncertainty features in the Quartus II software to compensate for the variable delays between clock domains. Consider setting a Clock Setup Uncertainty and Clock Hold Uncertainty value of 10% to 15% of the clock delay.

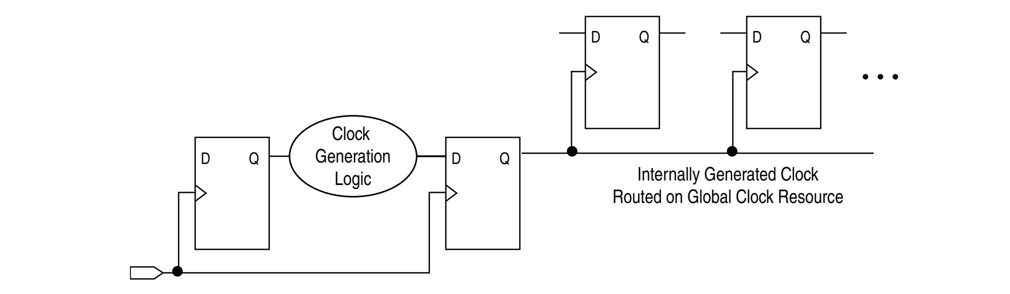
The following sections provide some specific examples and recommendations for avoiding these problems.

Internally Generated Clocks

If you use the output from combinational logic as a clock signal or as an asynchronous reset signal, expect to see glitches in your design. In a synchronous design, glitches on data inputs of registers are normal events that have no consequences. However, a glitch or a spike on the clock input (or an asynchronous input) to a register can have significant consequences. Narrow glitches can violate the register's minimum pulse width requirements. Setup and hold times may also be violated if the data input of the register is changing when a glitch reaches the clock input. Even if the design does not violate timing requirements, the register output can change value unexpectedly and cause functional hazards elsewhere in the design.

Because of these problems, Altera recommends that you always register the output of combinational logic before you use it as a clock signal (Figure 5–4).

Figure 5–4. Recommended Clock-Generation Technique



Registering the output of combinational logic ensures that the glitches generated by the combinational logic are blocked at the data input of the register.

Divided Clocks

Designs often require clocks created by dividing a master clock. Most Altera FPGAs provide dedicated phase-locked loop (PLL) circuitry for clock division. Using dedicated PLL circuitry can help you to avoid many of the problems that can be introduced by asynchronous clock division logic.

When you must use logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design so that registers always directly generate divided clock signals, as described in "Internally Generated Clocks" on page 5–8, and route the clock on global clock resources. To avoid glitches, do not decode the outputs of a counter or a state machine to generate clock signals.

Ripple Counters

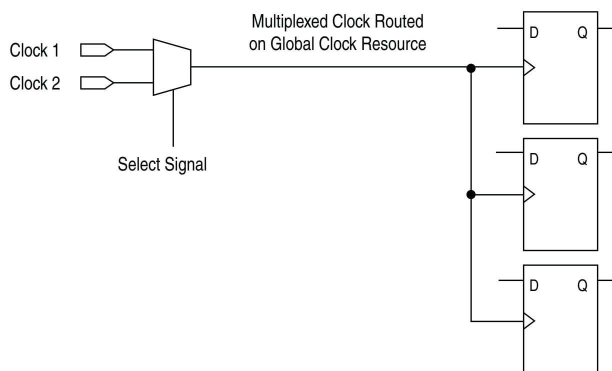
To simplify verification, Altera recommends avoiding ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which the output pin of each register feeds the clock pin of the register in the next stage. This cascading can cause problems because the counter creates a ripple clock at each stage. These ripple clocks have to be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and place-and-route tools.

Ripple clock structures are often used to make ripple counters out of the smallest amount of logic possible. However, in all Altera devices supported by the Quartus II software, using a ripple clock structure to reduce the amount of logic used for a counter is unnecessary because the device allows you to construct a counter using one logic element per counter bit. Altera recommends that you avoid using ripple counters under any circumstances.

Multiplexed Clocks

Use clock multiplexing to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source, as in Figure 5–5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 5–5. Multiplexing Logic and Clock Sources



Adding multiplexing logic to the clock signal can create the problems addressed in the previous sections, but requirements for multiplexed clocks vary widely, depending on the application. Clock multiplexing is acceptable when the clock signal uses global clock routing resources and if the following criteria are met:

- The clock multiplexing logic does not change after initial configuration
- The design uses multiplexing logic to select a clock for testing purposes
- Registers are always reset when the clock switches
- A temporarily incorrect response following clock switching has no negative consequences

If the design switches clocks in real time with no reset signal, and your design cannot tolerate a temporarily incorrect response, you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems. By default, the Quartus II software optimizes and analyzes all possible paths through the multiplexer and between both internal clocks that may come from the multiplexer. This may lead to more restrictive analysis than required if the multiplexer is always selecting one particular clock. If you do not require the more complete analysis, you can assign the output of the multiplexer as a base clock in the Quartus II software, so that all register-register paths are analyzed using that clock.

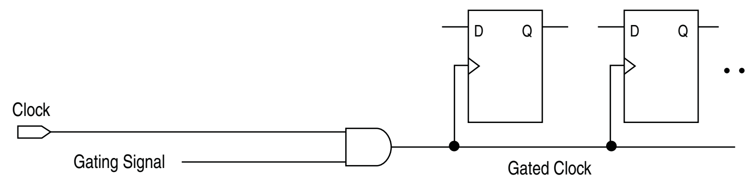
Altera recommends using dedicated hardware to perform clock multiplexing when it is available, instead of using multiplexing logic. For example, you can use the Clock Switchover feature or Clock Control Block available in certain Altera devices. These dedicated hardware blocks ensure that you use global low-skew routing lines and avoid any possible hold time problems on the device due to logic delay on the clock line.

Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry, as shown in Figure 5–6. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 5–6. Gated Clock



You can use gated clocks to reduce power consumption in some device architectures by effectively shutting down portions of a digital circuit when they are not in use. When a clock is gated, both the clock network and the registers driven by it stop toggling, thereby eliminating their contributions to power consumption. However, gated clocks are not part of a synchronous scheme and therefore can significantly increase the effort required for design implementation and verification. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

Altera recommends that you use dedicated hardware to perform clock gating rather than using an AND or OR gate. For example, you can use the clock control block in newer Altera devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew and avoid any possible hold time problems on the device due to logic delay on the clock line.

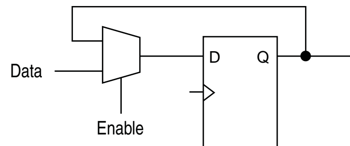
Refer to the appropriate device data sheet or handbook for device-specific information about clocking structures.

From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable signal. However, when using a synchronous clock enable scheme, the clock network continues toggling. This practice does not reduce power consumption as much as gating the clock at the source does. In most cases, use a synchronous scheme such as those described in "Synchronous Clock Enables". For improved power reduction when gating clocks with logic, refer to "Recommended Clock-Gating Methods" on page 5–11.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use a synchronous clock enable signal. FPGAs efficiently support clock enable signals because there is a dedicated clock enable signal available on all device registers. This scheme does not reduce power consumption as much as gating the clock at the source because the clock network keeps toggling, but it performs the same function as a gated clock by disabling a set of registers. Insert a multiplexer in front of the data input of every register to either load new data or copy the output of the register (Figure 5–7).

Figure 5–7. Synchronous Clock Enable

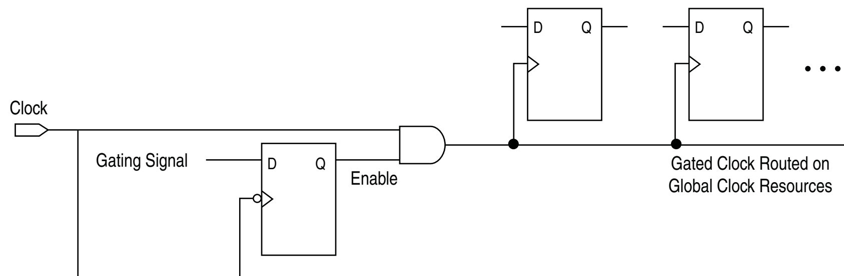


Recommended Clock-Gating Methods

Use gated clocks only when your target application requires power reduction and when gated clocks are able to provide the required reduction in your device architecture. If you must use clocks gated by logic, implement these clocks using the robust clock-gating technique shown in Figure 5–8 and ensure that the gated clock signal uses dedicated global clock routing.

You can gate a clock signal at the source of the clock network, at each register, or somewhere in between. Because the clock network contributes to switching power consumption, gate the clock at the source whenever possible, so you can shut down the entire clock network instead of gating it further along the clock network at the registers.

Figure 5–8. Recommended Clock-Gating Technique



In the technique shown in Figure 5–8, a register generates the enable signal to ensure that the signal is free of glitches and spikes. The register that generates the enable signal is triggered on the inactive edge of the clock to be gated (use the falling edge when gating a clock that is active on the rising edge, as shown in Figure 5–8). Using this technique, only one input of the gate that turns the clock on and off changes at a time. This prevents any glitches or spikes on the output. Use an AND gate to gate a clock that is active on the rising edge. For a clock that is active on the falling edge, use an OR gate to gate the clock and register the enable command with a positive edge-triggered register.

When using this technique, pay attention to the duty cycle of the clock and the delay through the logic that generates the enable signal because the enable command must be generated in one-half the clock cycle. This situation might cause problems if the logic that generates the enable command is particularly complex, or if the duty cycle of the clock is severely unbalanced. However, careful management of the duty cycle and logic delay may be an acceptable solution when compared with problems created by other methods of gating clocks.

Ensure that you apply a clock setting to the gated clock in the Quartus II software. As shown in Figure 5–8 on page 5–11, apply a clock setting to the output of the AND gate. Otherwise, the timing analyzer may analyze the circuit using the clock path through the register as the longest clock path and the path that skips the register as the shortest clock path, resulting in artificial clock skew.

In certain cases, converting the gated clocks to clock enables may help to reduce glitch and clock skew, and eventually produce a more accurate timing analysis. You can set the Quartus II software to automatically convert gated clocks to clock enables by turning on the **Auto Gated Clock Conversion** option. The conversion applies to two types of gated clocking schemes: single-gated clock and cascaded-gated clock. This option is only available for devices that are supported by the TimeQuest Timing Analyzer, except for Stratix® and Cyclone® devices.

For information about the settings and limitations of this option, refer to the “Auto Gated Clock Conversion” section of the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*.

Design Techniques to Save Power

The total FPGA power consumption is comprised of I/O power, core static power, and core dynamic power. Knowledge of the relationship between these components is fundamental in calculating the overall total power consumption. You can use various optimization techniques and tools to minimize power consumption when applied during FPGA design implementation. The Quartus II software offers power-driven compilation features to fully optimize device power consumption. Power-driven compilation focuses on reducing your design’s total power consumption using power-driven synthesis and power-driven place-and-route.

For information about power-driven compilation flow and low-power design guidelines, refer to the *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*.

For information about power optimization techniques available for Stratix III devices, refer to *AN 437: Power Optimization in Stratix III FPGAs*. For information about power optimization techniques available for Stratix IV devices, refer to *AN 514: Power Optimization in Stratix IV FPGAs*.

In addition, you can use the Quartus II PowerPlay Power Analyzer to aid you during the design process by delivering fast and accurate estimations of power consumption. For information about the PowerPlay Power Analyzer, refer to the *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

Checking Design Violations Using the Design Assistant

To improve the reliability, timing performance, and logic utilization of your design, practicing good design methodology and understanding how to avoid design rule violations are important. The Quartus II software provides a tool that automatically checks for design rule violations and reports their location.

The Design Assistant is a design rule checking tool that allows you to check for design issues early in the design flow. The Design Assistant checks your design for adherence to Altera-recommended design guidelines. You can specify which rules you want the Design Assistant to apply to your design. This is useful if you know that your design violates particular rules that are not critical, so you can allow these rule violations. The Design Assistant generates design violation reports with clear details about each violation, based on the settings you specified.

The first parts in this section provide an introduction to the Quartus II design flow with Design Assistant, message severity levels, and an explanation about how to set up the Design Assistant. The last parts of the section describe the design rules and the reports generated by the Design Assistant.

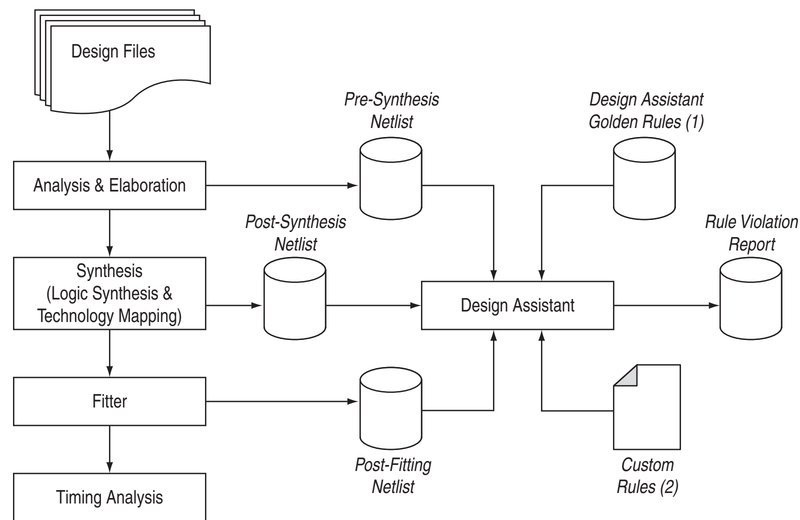
Quartus II Design Flow with the Design Assistant

You can run the Design Assistant after Analysis and Elaboration, Analysis and Synthesis, fitting, or a full compilation. To run the Design Assistant, on the Processing menu, point to **Start**, and click **Start Design Assistant**.

To set the Design Assistant to run automatically during compilation, on the Assignments menu, click **Settings**. In the **Category** list, select **Design Assistant**. Turn on **Run Design Assistant during compilation**. This enables the Design Assistant to perform a post-fitting netlist analysis of your design. The default is to apply all of the rules to your project. But if there are some rules that are unimportant to your design, you can turn off the rules that you do not want the Design Assistant to use. Refer to “The Design Assistant Settings Page” on page 5–15.

Figure 5–9 shows the Quartus II software design flow with the Design Assistant.

Figure 5–9. Quartus II Design Flow with the Design Assistant



Notes to Figure 5–9:

(1) Database of the default rules for the Design Assistant.

(2) A file that contains the **.xml** codes of the custom rules for the Design Assistant. For more details about how to create this file, refer to "Custom Rules" on page 5–33.

The Design Assistant analyzes your design netlist at different stages of the compilation flow and may yield different warnings or errors, even though the netlists are functionally the same. Your pre-synthesis, post-synthesis, and post-fitting netlists may be different due to optimizations performed by the Quartus II software. For example, a warning message in a pre-synthesis netlist may be removed after the netlist has been synthesized into a post-synthesis or post-fitting netlist.

- When you run the Design Assistant after running a full compilation or fitting, the Design Assistant performs a post-fitting analysis on the design.
- When you start the Design Assistant after performing Analysis and Synthesis, the Design Assistant performs post-synthesis analysis on the design.
- When you start the Design Assistant after performing Analysis and Elaboration, the Design Assistant performs a pre-synthesis analysis on the design. You can also perform pre-synthesis analysis with the Design Assistant using the command-line. You can use the `-rtl` option with the `quartus_drc` executable, as shown in the following example:

  quartus_drc <*project_name*> --rtl=on ↵

The Design Assistant generates warning messages when your design violates design rules and generates information messages to provide information regarding the rules. The Design Assistant supports all Altera devices supported by the Quartus II software.

The Design Assistant Settings Page

To apply design rules in the Design Assistant, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Settings** dialog box, in the **Category** list, select **Design Assistant**.
3. In the **Design Assistant** page, turn on the rules that you want the Design Assistant to apply during analysis. By default, all of the rules except the finite state machine rules are turned on.

To specify the file path to the custom rule file of the user-defined rules, refer to "Specifying the Path to the Custom Rules File" on page 5–35.

In the **Timing Closure** category, if **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out** are turned on, you can use the **High Fan-Out Net Settings** dialog box to specify the number of fan-out a node must have to be reported by the Design Assistant. To open the **High Fan-Out Net Settings** dialog box, in the **Design Assistant** page, in the **Timing Closure** category, select **Nodes with more than specified number of fan-outs** or **Top nodes with highest fan-out**. Click **High Fan-Out Net Settings**.

In the **Clock** category, if you turn on **Clock signal should be a global signal**, you can use the **Global Clock Threshold Settings** dialog box to specify the number of nodes with the highest fan-out that you want the Design Assistant to report. To open the **Global Clock Threshold Settings** dialog box, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings**.

To specify the maximum number of messages reported by the Design Assistant, on the **Design Assistant** page, click **Report Settings** and enter the maximum number of violation messages and detail messages to be reported.

Message Severity Levels

The Design Assistant classifies messages and rules using the four severity levels described in Table 5–1. Following Altera guidelines is very important for designs that are migrated to the HardCopy series of devices; therefore, the table highlights the impact of a rule violation on a HardCopy migration. Designs that adhere to Altera-recommended design guidelines do not produce any messages with critical-, high-, or medium-level severity.

Table 5–1. Design Assistant Message Severity Levels

Severity Level	Explanation
Critical	A violation of the rule critically affects the reliability of the design. Altera may not be able to implement the design successfully without closely reviewing the violations with the designer for HardCopy device conversions.
High	A violation of the rule affects the reliability of the design. Altera must review the violation before implementing the design for HardCopy device conversions.
Medium	The rule violation may result in implementation complexity that may have an impact for HardCopy device conversions.
Information Only	The rule provides information regarding the design.

Design Assistant Rules

This section describes the Design Assistant rules and details some of the reasons that Altera recommends following certain guidelines. Many of the Design Assistant rules enforce the design guidelines described in previous sections of this chapter.

Every rule is represented by a rule ID and has its own severity level. The rule ID is normally used in Tcl commands for rule suppression. The letter in each rule ID corresponds to the group of rules based on the following scheme:

- **A**—Asynchronous design structure rules
- **C**—Clock rules
- **R**—Reset rules
- **S**—Signal race rules
- **T**—Timing closure rules
- **D**—Asynchronous clock domain rules
- **H**—HardCopy rules
- **M**—Finite state machine rules

For example, the rule "Design Should Not Contain Combinational Loops" is the first rule in the asynchronous design structure rules; therefore, it is represented by rule ID A101.

Finite state machine rules are applicable only to register transfer level (RTL) check.

Summary of Rules and IDs

Table 5–2 lists the rules, their rule IDs, and their severity level.

Table 5–2. Summary of Rules and IDs (Part 1 of 2)

Rule ID	Rule Name	Severity Level
A101	Design Should Not Contain Combinational Loops	Critical
A102	Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic	Critical
A103	Design Should Not Contain Delay Chains	High
A104	Design Should Not Contain Ripple Clock Structures	Medium
A105	Pulses Should Not Be Implemented Asynchronously	Critical
A106	Multiple Pulses Should Not Be Generated in the Design	Critical
A107	Design Should Not Contain SR Latches	High
A108	Design Should Not Contain Latches	High
C101	Gated Clocks Should Be Implemented According to Altera Standard Scheme	Critical
C102	Logic Cell Should Not Be Used to Generate Inverted Clock	High
C103	Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>	Medium
C104	Clock Signal Source Should Drive Only Input Clock Ports	Medium
C105	Clock Signal Should Be a Global Signal	High
C106	Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges	Medium

Table 5–2. Summary of Rules and IDs (Part 2 of 2)

Rule ID	Rule Name	Severity Level
R101	Combinational Logic Used as a Reset Signal Should Be Synchronized	High
R102	External Reset Should Be Synchronized Using Two Cascaded Registers	Medium
R103	External Reset Should Be Synchronized Correctly	High
R104	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly	High
R105	Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized	Medium
S101	Output Enable and Input of the Same Tri-State Nodes Should Not Be Driven by the Same Signal Source	High
S102	Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source	High
S103	More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source	High
S104	Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source	High
T101	Nodes with More Than Specified Number of Fan-outs: <n>	Information Only
T102	Top Nodes with Highest Fan-out: <n>	Information Only
D101	Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains	High
D102	Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain	Medium
D103	Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains	High
M101	Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains	High
M102	No Reset Signal Defined to Initialize the State Machine	Medium
M103	State Machine Should Not Contain Unreachable State	Medium
M104	State Machine Should Not Contain a Deadlock State	Medium
M105	State Machine Should Not Contain a Dead Transition	Medium

Design Should Not Contain Combinational Loops

Severity Level: Critical
Rule ID: A101

A combinational loop is created by establishing a direct feedback loop on combinational logic that is not synchronized by a register. A combinational loop also occurs when the output of a register is fed back to an asynchronous pin of the same register through combinational logic. Combinational loops are among the most common causes of instability and reliability in your designs and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Register Output Should Not Drive Its Own Control Signal Directly or through Combinational Logic

Severity Level: Critical
Rule ID: A102

A combinational loop occurs when you feed back the output of a register to an asynchronous pin of the same register (for example, the register's preset or asynchronous load signal), or the register drives combinational logic that drives one of the control signals on the same register. Combinational loops are among the most common causes of instability and reliability in your designs and should be avoided whenever possible. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Design Should Not Contain Delay Chains

Severity Level: High
Rule ID: A103

Delay chains are created when one or more consecutive nodes with a single fan-in and a single fan-out are used to cause delay. Delay chains are sometimes used to create intentional delay to resolve race conditions. Delay chains may cause significant problems because they affect the rise and fall time differences in your design.

This rule applies only for delay chains implemented in logic cells and is limited to the clock and reset path of your design. This rule does not apply to delay chains in the data path. Altera recommends that you do not instantiate a cell that does not benefit the design and is used only to delay the signal. Refer to "Delay Chains" on page 5–5 for examples of the kinds of problems that delay chains can cause.

Design Should Not Contain Ripple Clock Structures

Severity Level: Medium
Rule ID: A104

Designs should not contain ripple clock structures. These structures use two or more cascaded registers in which the output of each register feeds the clock pin of the register in the next stage. Cascading structures cause large skew in the output signal because each stage of the structure causes a new clock domain to be defined. The additional clock domains from each stage of the ripple clock are difficult for static timing analysis tools to analyze. Refer to "Ripple Counters" on page 5–8 for examples of the kinds of problems that ripple clock structures can cause.

Pulses Should Not Be Implemented Asynchronously

Severity Level: Critical
Rule ID: A105

There are two common methods for pulse generation:

- Increasing the width of a glitch using a 2-input AND, NAND, OR, or NOR gate, where the source for the two gate inputs are the same, but one of the gate inputs is inverted
- Using a register where the register output drives the register's own asynchronous reset signal through a delay chain (refer to "Delay Chains" on page 5–5 for more details).

These techniques are purely asynchronous and therefore need to be avoided. Refer to "Pulse Generators and Multivibrators" on page 5–6 for recommended pulse generation guidelines.

Multiple Pulses Should Not Be Generated in the Design

Severity Level: Critical
Rule ID: A106

A common asynchronous multiple-pulse-generation technique consists of a combinational logic gate in which the inverted output feeds back to one of the inputs of the same gate. This feedback path causes a combinational loop that forces the output to change state and therefore, oscillate. Sometimes multiple pulse generators or multivibrator circuits are built out of a series of cascaded inverters in a structure called a "ring oscillator." Oscillation creates a new artificial clock in your design that is difficult for the Quartus II software to determine, set, or verify.

Structures that generate multiple pulses cause more problems than pulse generators because of the number of pulses involved. In addition, multiple pulse generators increase the frequency of the design. Refer to "Pulse Generators and Multivibrators" on page 5–6 for recommended pulse generation guidelines.

Design Should Not Contain SR Latches

Severity Level: High
Rule ID: A107

A latch is a combinational loop that holds the value of a signal until a new value is assigned. Combinational loops are hazardous to your design and are the most common causes of instability and unreliability. Refer to "Combinational Loops" on page 5–4 for examples of the kinds of problems that combinational loops can cause.

Rule A107 triggers only when your design contains SR latches. An SR latch can cause glitches and ambiguous timing, which complicates the timing analysis of your design. Refer to "Latches" on page 5–5 for details about latches and for more examples of the kinds of problems that latches can cause.

Design Should Not Contain Latches

Severity Level: High
Rule ID: A108

The Design Assistant generates warnings when it identifies one or more structures as latches.

Refer to "Latches" on page 5–5 for details about latches and for examples of the kinds of problems that latches can cause.

The difference between A107 ("Design Should Not Contain SR Latches") and A108 is that A107 triggers only when an SR latch is detected. A108 triggers when an unidentified latch exists in your design.

Gated Clocks Should Be Implemented According to Altera Standard Scheme

Severity Level: Critical
Rule ID: C101

Clock gating is sometimes used to turn parts of a circuit on and off to reduce the total power consumption of a device. Clock gating is implemented using an enable signal that controls some sort of gating circuitry. The gated clock signal prevents any of the logic driven by it from switching so the logic does not consume any power. For example, when a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive. However, the disadvantage of using this type of circuit is that it can lead to unexpected glitches on the resultant gated clock signal if certain rules are not followed.

Refer to "Gated Clocks" on page 5–10 for examples of the kinds of problems gated clocks can cause. Refer to "Recommended Clock-Gating Methods" on page 5–11 for a recommended clock gating technique. However, when following the recommended clock gating techniques, your design must have a minimum number of fan-outs to meet rule C103, "Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>."

Logic Cell Should Not Be Used to Generate Inverted Clock

Severity Level: High
Rule ID: C102

Your design may require both positive and negative edges of a clock to operate. However, do not implement an inverter to drive the clock input of a register in your design with a logic cell. Implementing the inverter with a logic cell can lead to clock insertion delay and skew, which is hazardous to your design and can cause problems with the timing closure of the design.

In addition, using a logic cell to implement an inverter is unnecessary. Use the programmable clock inversion featured in the register to generate the inverted clock signal. Refer to "Clocking Schemes" on page 5–7 for details about different types of clocking methods.

Gated Clock Is Not Feeding At Least A Pre-Defined Number Of Clock Ports to Effectively Save Power: <n>

Severity Level: Medium
Rule ID: C103

Your design can contain an input clock pin that fans out to more than one gated clock. However, to effectively reduce power consumption, Altera recommends that the gated clock feed at least a pre-defined number of clock ports (*n* ports). The default value for *n* is 30. You can set the number of clock ports (*n*) by performing the following steps:

1. Click **Settings** on the Assignments menu.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, expand the **Clock** category and turn on **Gated clock is not feeding at least a pre-defined number of clock port to effectively save power: <n>**.
4. Click on the **Gated Clock Settings** button, and in the **Gated Clock Settings** dialog box, set the number of clock ports a gated clock should feed.

Refer to “Clocking Schemes” on page 5–7, and “Recommended Clock-Gating Methods” on page 5–11 for proper clock-gating techniques.

Clock Signal Source Should Drive Only Input Clock Ports

Severity Level: Medium
Rule ID: C104

Clock signal sources in a design should drive only input clock ports of registers. Rule C104 triggers when a design contains a clock signal source of a register that connects to the port rather than the clock port of another register. Note that if the clock signal source and ports are of the same register, rule S104 “Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source” is triggered instead. Such a design is considered asynchronous and should be avoided. Asynchronous design structures can be hazardous to your design because some of them rely on the relative propagation delays of signals to function correctly, which can result in incomplete timing constraints and possible glitches and spikes.

Refer to “Hazards of Asynchronous Design” on page 5–3 for examples of the kinds of problems that asynchronous design structures can cause. Also refer to “Clocking Schemes” on page 5–7 for proper clocking techniques.

This rule does not apply in the following conditions:

- When the clock signal source drives combinational logic that is used as a clock signal and the combinational logic is implemented according to the Altera standard scheme
- When the clock signal source drives only a clock multiplexer that selects one clock source from a number of different clock sources

This type of multiplexer adds complexity to the timing analysis of a design. Avoid using the multiplexer in the design.

- Using a clock multiplexer causes the “Gated Clocks Should Be Implemented According to Altera Standard Scheme” rule (C101) to be applied; refer to “Multiplexed Clocks” on page 5–9 for recommended clock multiplexing techniques

Clock Signal Should Be a Global Signal

Severity Level: High
Rule ID: C105

Ensure that all clock signals in your design use the global clock networks that exist in the target FPGA. Mapping clock signals to use non-dedicated clock networks can negatively affect the performance of your design. A non-global signal can be slower and have larger skew than a global signal because the clock must be distributed using regular FPGA routing resources.

To specify the number of minimum fan-outs that you want the Design Assistant to report, on the **Design Assistant** page, in the **Clock** category, select **Clock signal should be a global signal**. Click **Global Clock Threshold Settings** and enter the number in the dialog box.

If a design contains more clock signals than are available in the target device, consider reducing the number of clock signals in the design, such that only dedicated clock resources are used for clock distribution. However, if the design must use more clock signals than you can specify as global signals, implement the clock signals with the lowest fan-out using regular routing resources. Also, implement the fastest clock signals as global signals. Refer to "Clock Network Resources" on page 5–40 for detailed information about clock resources.

Clock Signal Source Should Not Drive Registers that Are Triggered by Different Clock Edges

Severity Level: Medium
Rule ID: C106

This rule triggers an error message if your design contains a clock signal source that drives the clock inputs of both positive and negative edge-sensitive registers. This error also triggers if your design contains an inverted clock signal that drives the clock inputs of either positive or negative edge-sensitive registers.

These two scenarios can cause an increase in timing requirement complexity and difficulties in design optimization. Also, synchronous resetting may not be possible because registers are not clocked on the same edge in the design. Refer to "Clocking Schemes" on page 5–7 for some specific examples and recommended clocking methods.

Combinational Logic Used as a Reset Signal Should Be Synchronized

Severity Level: High
Rule ID: R101

All combinational logic used to drive reset signals in your design needs to be synchronized. This means that a register is required between the combinational logic that drives the reset signal and input reset pin. Unsynchronized combinational logic can cause glitches and spikes that lead to unintentional reset signals. Synchronizing the combinational logic that drives the reset signal delays the resulting reset signal by an extra clock cycle and avoids unintentional reset. You must consider the extra clock cycle delay when using this method in your design.

Rule R101 does not trigger if the combinational logic used is either a 2-input AND or NOR that feeds active low reset port, or either a 2-input OR or NAND that feeds active high reset port.

External Reset Should Be Synchronized Using Two Cascaded Registers

Severity Level: Medium
Rule ID: R102

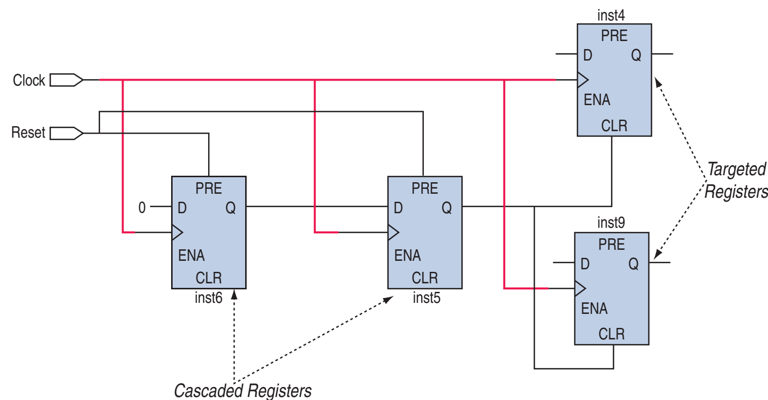
The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, an asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using a double-buffer circuit, which consists of two cascaded registers triggered on the same clock edge and on the same clock domain as the targeted registers.

This rule does not apply in the following two conditions:

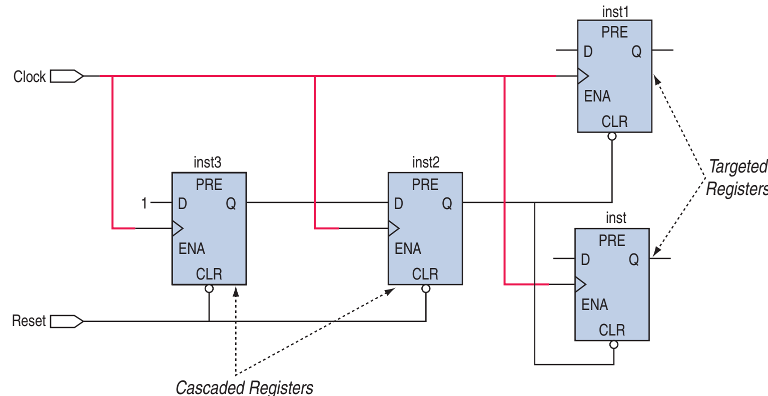
- When the targeted registers use active-high reset ports and the external reset signal drives the PRE ports on the cascaded registers with the input port of the first cascaded registers is fed to GND. Refer to Figure 5–10.

Figure 5–10. Active-High Reset Ports



- When the targeted registers use active-low reset ports and the external reset signal drives the CLR ports on the cascaded registers with the input port of the first cascaded registers is fed to V_{CC}. Refer to Figure 5–11.

Figure 5–11. Active-Low Reset Ports



External Reset Should Be Synchronized Correctly

Severity Level: High
Rule ID: R103

The only way to put your design into a reset state in the absence of a clock signal is to use an asynchronous reset or external reset. However, asynchronous reset can affect the recovery time of a register, cause design stability problems, and unintentionally reset the state machines in your design to incorrect states.

As a guideline, you can synchronize an external reset signal by using two cascaded registers. The registers need to be triggered on the same clock edge and should be in the same clock domain as the targeted registers.

This rule applies when an asynchronous reset or external reset signal is synchronized but fails to follow the recommended guidelines, as described in rule R102 (“External Reset Should Be Synchronized Using Two Cascaded Registers”). This violation happens when the external reset is synchronized with only one register or the cascaded synchronization registers are triggered on different clock edges.

R102 triggers when you don’t use two cascaded registers to synchronize the external reset. R103 triggers when the external reset is synchronized but fails to follow the recommended guidelines.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly

Severity Level: High
Rule ID: R104

If your design uses an internally generated reset signal generated in one clock domain and used in one or more other asynchronous clock domains, the reset signal needs to be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, use the following guidelines:

- The reset signal needs to be synchronized with two or more cascading registers in the receiving asynchronous clock domain.
- The cascading registers needs to be triggered on the same clock edge.
- There must be no logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. The synchronization registers may sample unintended data due to the glitches caused by the logic.

This rule applies when the internal reset signal is synchronized but fails to follow the recommended guidelines. This happens when the external reset is only synchronized with one register, when the cascaded synchronization registers are triggered on different clock edges, or when there is logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain. Synchronizing the reset signal delays the signal by an extra clock cycle. Consider this delay when using the reset signal in a design.

Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized

Severity Level: Medium
Rule ID: R105

If your design uses an internally generated reset signal that is generated in one clock domain and used in one or more other asynchronous clock domain, the reset signal needs to be synchronized. An unsynchronized reset signal can cause metastability issues. To synchronize reset signals across clock domains, follow the guidelines described in Rule R104 (“Reset Signal Generated in One Clock Domain and Used in Other Asynchronous Clock Domains Should Be Synchronized Correctly”).

This rule applies when the internally generated reset signal is not being synchronized.

 © November 2009 Altera Corporation

Output Enable and Input of the Same Tri-State Nodes Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S101

This rule applies when your design contains a tri-state node in which the input and output enable are driven by the same signal source. Signal race occurs between the input and output enable signals of the tri-state when they are propagated simultaneously. Race conditions lead to incorrect design function and unpredictable results. To avoid violation of this rule, the input and output enable of the tri-state should be driven by separate signal sources.

Synchronous Port and Asynchronous Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S102

A purely synchronous design is free of signal race conditions as long as the clock signal is properly distributed and the timing requirements of the registers are met. However, race conditions can occur typically when the synchronous and asynchronous input pins of the register are driven by the same signal source. Race conditions can cause incorrect design function and unpredictable results. Rule S102 triggers when the synchronous and asynchronous pins of a register are driven by the same signal source. Rule S102 does not trigger if the signal source is from a negative-edge sensitive register of the same clock and if the source register is directly feeding the reset port, provided there is no combinational logic in-between the signal and register.

More Than One Asynchronous Signal Source of the Same Register Should Not Be Driven by the Same Source

Severity Level: High
Rule ID: S103

To avoid race conditions in your design, Altera recommends that you avoid using the same signal source to drive more than one port on a register. The following ports are affected: `ALOAD`, `ADATA`, `Preset`, and `Clear`.

Clock Port and Any Other Signal Port of the Same Register Should Not Be Driven by the Same Signal Source

Severity Level: High
Rule ID: S104

Any clock signal source in your design needs to drive only input clock ports of registers. Rule S104 triggers only when your design contains clock signal sources that connect to ports other than the clock ports of the same register. Rule S104 is a subset of C104, “Clock Signal Source Should Drive Only Input Clock Ports” on page 5–21. Such a design is considered asynchronous and should be avoided.

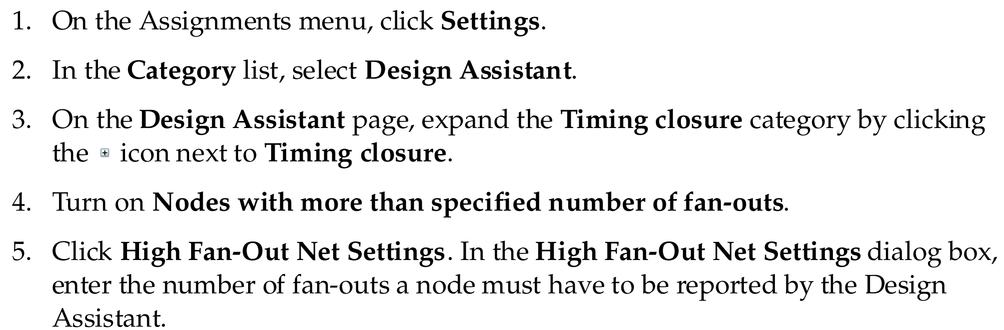
Refer to “Hazards of Asynchronous Design” for examples of the kinds of problems that asynchronous design structures can cause. Refer to “Clocking Schemes” for proper clocking techniques.

Nodes with More Than Specified Number of Fan-outs: <n>

Severity Level: Information Only
Rule ID: T101

This rule reports nodes that have more than a specified number of fan-outs, which can create timing challenges for your design.

To specify the number of fan-outs, follow these steps:

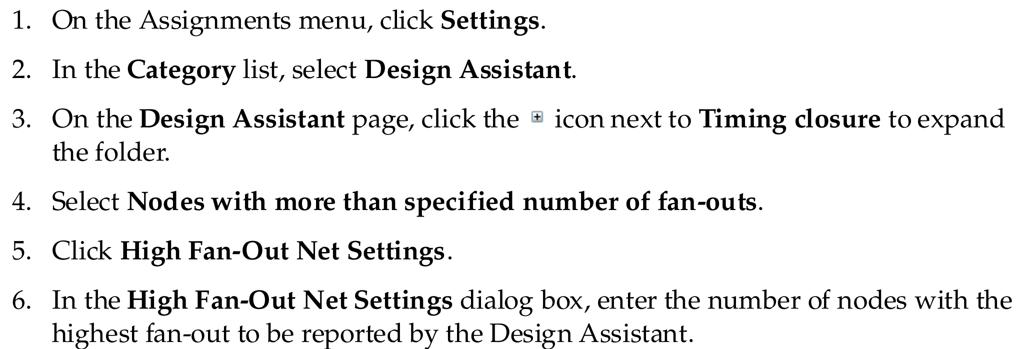
1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, expand the **Timing closure** category by clicking the  icon next to **Timing closure**.
4. Turn on **Nodes with more than specified number of fan-outs**.
5. Click **High Fan-Out Net Settings**. In the **High Fan-Out Net Settings** dialog box, enter the number of fan-outs a node must have to be reported by the Design Assistant.

Top Nodes with Highest Fan-out: <n>

Severity Level: Information Only
Rule ID: T102

This rule reports the specified number of nodes with the highest fan-out, which can create timing challenges for your design.

To specify the number of fan-outs, follow these steps:

1. On the Assignments menu, click **Settings**.
2. In the **Category** list, select **Design Assistant**.
3. On the **Design Assistant** page, click the  icon next to **Timing closure** to expand the folder.
4. Select **Nodes with more than specified number of fan-outs**.
5. Click **High Fan-Out Net Settings**.
6. In the **High Fan-Out Net Settings** dialog box, enter the number of nodes with the highest fan-out to be reported by the Design Assistant.

Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains

Severity Level: High
Rule ID: D101

The data bits transferred between asynchronous clock domains in a design need to be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit needs to be synchronized with two cascading registers in the receiving asynchronous clock domain, in which the cascaded registers are triggered on the same clock edge. Do not put any logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

If the data bits belong to multiple-bit data, use a handshake protocol to guarantee that all bits of the data bus are stable when the receiving clock domain samples the data. If you use a handshake protocol, only the data bits that act as `REQ` (request) and `ACK` (acknowledge) signals must be synchronized. The data bits that belong to multiple-bit data do not need to be synchronized. Ignore the violation on the data bits that use a handshake protocol.

Multiple Data Bits Transferred Across Asynchronous Clock Domains Are Synchronized, But Not All Bits May Be Aligned in the Receiving Clock Domain

Severity Level: Medium
Rule ID: D102

This rule applies when data bits from a multiple-bit data that are transferred between asynchronous clock domains are synchronized. However, not all data bits may be aligned in the receiving clock domain. Propagation delays may cause skew when the data reaches the receiving clock domain.

If the data bits belong to multiple-bit data and you use a handshake protocol, only the data bits that act as `REQ`, `ACK`, or both signals for the transfer need to be synchronized with two or more cascading registers in the receiving asynchronous clock domain.

If all of the data bits belong to single-bit data, the synchronization of the data bits does not cause problems in the design.

Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains

Severity Level: High
Rule ID: D103

The data bits that are transferred between asynchronous clock domains in a design need to be synchronized to avoid metastability problems.

If the data bits belong to single-bit data, each data bit needs to be synchronized with two cascading registers in the receiving asynchronous clock domain. In this case, the cascaded registers are triggered on the same clock edge. Do not put any logic between the output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain.

This rule only applies when the data bits across asynchronous clock domains are synchronized but fail to follow the guidelines.

Data Bits Are Not Synchronized When Transferred to the State Machine of Asynchronous Clock Domains

Severity Level: High
Rule ID: M101

Data bits that are transferred between asynchronous clock domains in your design need to be synchronized to avoid metastability problems. Rule M101 is a state-machine-specific rule that triggers when input signals of a state machine across asynchronous clock domains are not synchronized or improperly synchronized. Rule M101 applies to state machines only, while the “Data Bits Are Not Synchronized When Transferred between Asynchronous Clock Domains” rule (D101) and the “Data Bits Are Not Correctly Synchronized When Transferred Between Asynchronous Clock Domains” rule (D103) apply only to data synchronization between registers.

No Reset Signal Defined to Initialize the State Machine

Severity Level: Medium
Rule ID: M102

A finite state machine (FSM) needs to have a reset signal that initializes the state machine to its initial state. A finite state machine without a proper initialization state is susceptible to functional problems and can introduce extra difficulty in analysis, verification, and maintenance.

State Machine Should Not Contain Unreachable State

Severity Level: Medium
Rule ID: M103

An unreachable state is a state that can never be reached from the initial state. Having an unreachable state in your design causes logic redundancy and affects your design functionality. Rule M103 triggers when the initial state cannot traverse to a state through any of the reachable states and transitions.

State Machine Should Not Contain a Deadlock State

Severity Level: Medium
Rule ID: M104

A deadlock state is a state that does not have any transitions to another state except to loop to itself. When the state machine enters a deadlock state, it stays in that state until the state machine is reset. Your design may have a single state, or a few states forming a deadlock structure. Having a deadlock state in your design leads to design functionality problems, and theoretically may consume more power.

You can change the maximum number of states to be detected as a deadlock structure by clicking **Settings** on the Assignments menu, and in the **Settings** dialog box, in the **Category** list, select **Design Assistant**. In the **Design Assistant** page, click **Finite State Machine Deadlock Settings**. In the **Finite State Machine Deadlock Settings** dialog box, specify the maximum number of states to be reported as a deadlock structure. The default setting is `2`.

State Machine Should Not Contain a Dead Transition

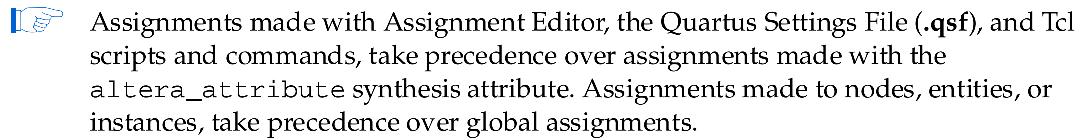
Severity Level: Medium
Rule ID: M105

A dead transition is a redundant transition that never occurs regardless of the event sequence input to the state machine. A dead transition indicates logic redundancy in your design, although it may not affect your design functionality. Rule M105 triggers when the condition required to trigger a transition is not possible.

Enabling and Disabling Design Assistant Rules

You can selectively enable or disable Design Assistant rules on individual nodes by making an assignment in the Assignment Editor or by using the `altera_attribute` synthesis attribute in Verilog HDL or VHDL, or using a Tcl command.

For a list of the types of nodes that allow Design Assistant rule suppression, refer to *Node Types Eligible for Rule Suppression* in the Quartus II Help.


Assignments made with Assignment Editor, the Quartus Settings File (**.qsf**), and Tcl scripts and commands, take precedence over assignments made with the `altera_attribute` synthesis attribute. Assignments made to nodes, entities, or instances, take precedence over global assignments.

Using the Assignment Editor

You can enable or disable a Design Assistant rule on selected nodes in your design by using the Assignments Editor. You must first compile your design if you have not already done so.

To enable or disable a Design Assistant rule, follow these steps:

1. On the Assignments menu, click **Assignment Editor**.
2. In the spreadsheet, for the desired node, entity, or instance, double-click the cell in the **Assignment Name** column and select **Enable Design Assistant Rule** or **Disable Design Assistant Rule** in the pull-down menu.
3. Double-click the **Value** cell and type in the Rule ID.

   or

   Click **Browse** to open the **Design Assistant Rules** dialog box. In the **Design Assistant Rules** dialog box, select the rule you want to enable or disable for that particular node.

You can enable or disable multiple rules by typing more than one Rule ID in the cell and separating each Rule ID with a comma.

Using Verilog HDL

You can use the `altera_attributes` synthesis attribute in your Verilog HDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, the syntax is shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="enable_da_rule=<ruleID>" */
```

You can enable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="enable_da_rule=\"<ruleID>,
<ruleID>, <ruleID>\""*/
```

To disable the rule on the selected node, the syntax is shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="disable_da_rule=<ruleID>" */
```

You can disable more than one rule on a selected node as shown in the following example:

```
<entity class> <object> /* synthesis altera_attribute="disable_da_rule=\"<ruleID>,
<ruleID>, <ruleID>\""*/
```

When enabling or disabling multiple rules in Verilog HDL, you must separate the Rule ID strings with commas and spaces only and they must be enclosed within the `\"` and `\"` characters.

Using VHDL

You can use the `altera_attributes` synthesis attribute in your VHDL code to enable or disable a Design Assistant rule on the selected nodes in your design.

To enable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "enable_da_rule=<ruleID>"
```

You can enable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "enable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```

To disable the rule on the selected node, use the following syntax:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "disable_da_rule=<ruleID>"
```

You can disable more than one rule on a selected node as shown in the following example:

```
attribute altera_attribute : string;attribute altera_attribute of <object>: <entity
class> is "disable_da_rule=""<ruleID>, <ruleID>, <ruleID>"""
```

When enabling or disabling multiple rules in VHDL, you must separate the Rule ID strings with commas and spaces only and they must be enclosed with double quotation mark (" ") characters.

Using TCL Commands

To enable a Design Assistant rule on the selected node in your design using Tcl in a script or at a Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>" -to <design element> ↵
```

To enable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name enable_da_rule "<rule ID>, <rule ID>"
-to <design element> ↵
```

To disable a Design Assistant rule on a selected node in your design using Tcl in a script, or at a command or Tcl prompt, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>" -to <design element> ↵
```

To disable more than one rule on a selected node, use the following Tcl command:

```
set_instance_assignment -name disable_da_rule "<rule ID>, <rule ID>"
-to <design element> ↵
```

Viewing Design Assistant Results

If your design violates a design rule, the Design Assistant generates warning messages and information messages about the violated design rule. The Design Assistant displays these messages in the Messages window, in the Design Assistant Messages report, and in the Design Assistant report files. You can find the Design Assistant report files called <*project_name*>**.drc.rpt** in the <*project_name*> subdirectory of the project directory.

The Design Assistant generates the following reports based on the settings specified in the Design Assistant page:

- Summary Report
- Settings Report
- Detailed Results Report
- Messages Report
- Rule Suppression Assignments Report
- Ignored Design Assistant Assignments Report
- Custom Rules Report

Summary Report

The Design Assistant Summary report contains a summary of the Design Assistant process on a particular project. The Design Assistant Summary report provides the following information:

- **Design Assistant Status**—the status, end date, and end time of the Design Assistant operation
- **Revision Name**—the revision name specified in the **Revisions** dialog box
- **Top-level Entity Name**—the top-level entity of your design
- **Family**—the device family name specified in the **Device** page of the **Settings** dialog box
- **Total Critical Violations**, **Total High Violations**, **Total Medium Violations**, and **Total Information Only Violations**—the total violations of the rules organized by level, some of which might affect the reliability of the design

Review the violations closely before converting your design for HardCopy devices to achieve a successful conversion.

Settings Report

The Design Assistant Settings report contains a list of enabled Design Assistant rules and options that you specified in the **Design Assistant Settings** page, as shown in Figure 5–12.

Figure 5–12. The Design Assistant Settings Report

Compilation Report - Design Assistant Settings

- Compilation Report
  - Legal Notice
  - Flow Summary
  - Flow Settings
  - Flow Non-Default Global Se
  - Flow Elapsed Time
  - Flow Log
  - Analysis & Synthesis
  - Fitter
  - Assembler
  - Timing Analyzer
  - Design Assistant
    - Summary
    - Settings
    - Medium Violations
    - Information only Violat
    - Messages
  - EDA Netlist Writer

Design Assistant Settings

	Option	Setting	To
1	Design Assistant mode	Post-Fitting	
2	Threshold value for clock net not mapped to clock spines rule	25	
3	Minimum number of node fan-out	30	
4	Maximum number of nodes to report	50	
5	Rule C101: Gated clock should be implemented according to Altera standard scheme	On	
6	Rule C102: Logic cell should not be used to generate inverted clock	On	
7	Rule C103: Input clock pin should fan out to only one set of clock gating logic	On	
8	Rule C104: Clock signal source should drive only input clock ports	On	
9	Rule C105: Clock signal should be a global signal (Rule applies during post-fitting analysis. This rule applies during both post-fitting analysis and post-synthesis analysis if the design targets a MAX 3000 or MAX 7000 device. For more information, see the Help for the rule.)	On	
10	Rule C106: Clock signal source should not drive registers that are triggered by different clock edges	On	
11	Rule R101: Combinational logic used as reset signal should be synchronized	On	
12	Rule R102: External reset should be synchronized using two cascaded registers	On	
13	Rule R103: External reset should be correctly synchronized	On	
14	Rule R104: Reset signal that is generated in one clock domain and used in other, asynchronous clock domains should be correctly synchronized	On	
15	Rule R105: Reset signal that is generated in one clock domain and used in other, asynchronous clock domains should be synchronized	On	
16	Rule T101: Nodes with more than specified number of fan-outs	On	

Detailed Results Report

The Detailed Results report contains detailed information of every rule violation including the rule name, node name, and fan-out. This report appears only if you specify settings in the **Design Assistant Settings** page. For more information about how to specify the settings, refer to "The Design Assistant Settings Page" on page 5–15.

Separate Detailed Results reports are generated for critical, high, medium, and information only results. Figure 5–13 shows the **Information Only Violations** report.

Figure 5–13. The Design Detailed Results Report, Information Only

Compilation Report - Information only Violations

- Compilation Report
  - Legal Notice
  - Flow Summary
  - Flow Settings
  - Flow Non-Default Global Settings
  - Flow Elapsed Time
  - Flow Log
  - Analysis & Synthesis
  - Fitter
  - Assembler
  - Timing Analyzer
  - Design Assistant
    - Summary
    - Settings
    - Medium Violations
    - Information only Violations
    - Messages
  - EDA Netlist Writer

Information only Violations

	Rule name	Name
1	Rule T102: Top nodes with highest fan-out	clock
2	Rule T102: Top nodes with highest fan-out	clken
3	Rule T102: Top nodes with highest fan-out	aclr
4	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
5	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
6	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
7	Rule T102: Top nodes with highest fan-out	denom[0]
8	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
9	Rule T102: Top nodes with highest fan-out	denom[1]
10	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
11	Rule T102: Top nodes with highest fan-out	denom[3]
12	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
13	Rule T102: Top nodes with highest fan-out	denom[2]
14	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
15	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
16	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
17	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
18	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul
19	Rule T102: Top nodes with highest fan-out	my_divider:inst|lpm_divide:lpm_divide_component|lpm_divide_6is:aul

Messages Report

The Messages report contains current information, warning, and error messages generated during the Design Assistant process. You can right-click a message in the Messages report and click **Help** to display the Quartus II software Help with details about the selected message, or click **Locate** to trace or cross-probe the selected node and locate the source of the violation.

Rule Suppression Assignments Report

The Rule Suppression Assignments report contains detailed information about which Design Assistant rules are enabled or disabled, as explained in the "Enabling and Disabling Design Assistant Rules" on page 5–28. The report shows the following information:

- **Assignment**—shows the name of the assignment
- **Value**—identifies the rule
- **To**—shows the name of the node where the rule is being applied

Ignored Design Assistant Assignments Report

The Ignored Design Assistant Assignments report lists detailed information about the invalid and conflicting rule assignments reported by the Design Assistant. This report is generated only if you specify an invalid rule ID in the rule suppression or a conflicting rule assignment. The following information appears in the report:

- **Assignment**—shows the name of the assignment
- **Value**—identifies the rule
- **To**—shows the name of the node where the rule is being applied
- **Comment**—shows why the assignment is being ignored

Custom Rules Report

The Design Assistant Custom Rules report contains the names of the custom rules used in the design checking, the path to the custom rules files from which the custom rules are read, and the list of ignored custom rules.

Custom Rules

In addition to the existing design rules that the Design Assistant offers, you can also create your own rules and specify your own reporting format in a text file (with any file extension) using the XML format. You then specify the path to that file in the Design Assistant settings page and run the Design Assistant for violations checking.

The file that contains the default rules for the Design Assistant is located at <*Quartus II install path*>**\quartus\libraries\design-assistant\da_golden_rule.xml**.

For details about how to set the file path to your custom rules, refer to "Specifying the Path to the Custom Rules File" on page 5–35.

This section explains the basics of writing a custom rule, the Design Assistant settings, and provides coding examples on how to check for clock relationship and node relationship in a design.

XML File Format for Custom Rules

All XML commands in custom rules file must be written within the `<ROOT>` and `</ROOT>` tags. Every user-defined rule consists of three main sections:

- Rule Attribute
- Rule Definition
- Reporting

The Rule Definition and Reporting sections must be defined inside the Rule Attribute section. Example 5–1 shows all three sections in a pre-defined custom rule file.

XML commands and attributes are case sensitive. However, attribute values are *not* case sensitive.

Example 5–1. Predefined XML File Format for a Custom Rule

```
<ROOT>
<!-Start creating a rule here -->

<!--Define rule attribute for a rule here -->
<DA_RULE ID=<rule id> NAME=<rule name> SEVERITY=<rule severity> DEFAULT_RUN=<default run> >

<RULE_DEFINITION>
      <!--Define rule definition here -->
   </RULE_DEFINITION>

   <REPORTING>
      <!--Define report settings here -->
   </REPORTING>

</DA_RULE>

</ROOT>
```

The Rule Attribute section contains the name, ID, severity level, and enable value of a rule. The order of these attributes is not important. This section is enclosed within `<DA_RULE>` and `</DA_RULE>` tags. Table 5–3 describes the attributes of the Rule Attribute section.

Table 5–3. Attributes for the Rule Attribute Section

Attribute	Description
ID	The value for this attribute is string type and must be unique. This attribute is required. For the list of IDs of the default rules, refer to Table 5–2 on page 5–16.
NAME	The value for this attribute is string type. This attribute is optional.
SEVERITY	This attribute presents the severity level of the rule. The value is string type and can be `CRITICAL`, `HIGH`, `MEDIUM`, or `INFORMATION`. This attribute is required. For details about rule severity level, refer to "Message Severity Levels" on page 5–15.
DEFAULT_RUN	The value is string type and can only be `YES`, or `NO`. If the value is `YES`, the rule is included in the design rule check, and vice versa. By default, the value is `YES`. This attribute is optional.

All string-type values must be enclosed within double quotes.

Command lines that begin with a single XML tag must end with the "/>" sign before another command begins.

The Rule Definition section is where you declare the node properties and rule triggering conditions, enclosed by <RULE_DEFINITION> and </RULE_DEFINITION> tags.

There are four subsections within the Rule Definition section that you can use to declare the properties and conditions:

- <DECLARE>—Global nodes that are used in the file are declared in this subsection. Every node name must be unique.

  A node declared outside of the <DECLARE> subsection is considered a local node. You can perform local node declaration at any place in the <BASIC>, <REQUIRED>, and <FORBID> subsections, and can be performed using the node declaration command directly without being enclosed within the <DECLARE> tag.

- <BASIC>—This subsection contains the condition that acts like a trigger point which the Design Assistant continuously checks for a match. If the condition is fulfilled, the Design Assistant checks the remaining conditions in the <REQUIRED> and <FORBID> subsections.
- <REQUIRED>—This subsection contains the acceptable conditions that your design must meet. If the condition is not fulfilled, the Design Assistant reports a rule violation.
- <FORBID>—This subsection contains the undesirable condition for a design. If the condition is fulfilled, the Design Assistant highlights a rule violation. This subsection may be optional, depending on your rule situation.

The Reporting section is where you describe the settings for rule violation reporting, enclosed by <REPORTING> and </REPORTING> tags. This section is optional. If there is no Reporting section defined, the violated rule is not reported. If the Reporting section is defined, the Design Assistant reports the name of the violated rule and the nodes that violated the rule according to the reporting format that you defined.

Specifying the Path to the Custom Rules File

To specify the path to the custom rule file, follow these steps:

1. To specify the path, on the Assignments menu, click **Settings**.
2. In the Category list, click **Design Assistant** and select **Custom Rules Settings**.
3. In the **Custom Rules Settings** dialog box, in the **Project custom rules file name** field, specify the path to your custom rules file.
4. Click **OK**.

Your rules are now included in the list of default Design Assistant rules.

The default file extension for a Design Assistant custom rules file is **.dacr**, but the file can have any file name or extension.

To specify the rules that you want the Design Assistant to check for violations, refer to "The Design Assistant Settings Page" on page 5–15.

Custom Rules Coding Examples

The following examples of custom rules show how to check node relationships and clock relationships in a design.

Checking SR Latch Structures In a Design

Example 5–2 shows the XML codes for checking SR latch structures in a design.

Example 5–2. Detecting SR Latches in a Design

```
<DA_RULE ID="EX01" SEVERITY="CRITICAL" NAME="Checking Design for SR Latch"
DEFAULT_RUN="YES">
<RULE_DEFINITION>
   <FORBID>
   <OR>
      <NODE NAME="NODE_1" TYPE="SRLATCH" />
      <HAS_NODE NODE_LIST="NODE_1" />
      <NODE NAME="NODE_1" TOTAL_FANIN="EQ2" />
      <NODE NAME="NODE_2" TOTAL_FANIN="EQ2" />
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
      </AND>
      <AND>
         <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR" TO_NAME="NODE_2"
TO_TYPE="NOR" />
         <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR" TO_NAME="NODE_1"
TO_TYPE="NOR" />
      </AND>
      </OR>
   </FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
   <MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
      <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
      <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   </MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

In Example 5–2, the possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the block is true, the block is fulfilled (OR gate similarity). If no `<AND></AND>` or `<OR></OR>` block is specified, the default is `<AND></AND>`.

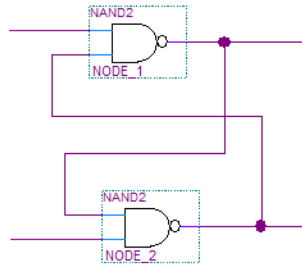
The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the SR latch structures. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples are the undesired conditions from Example 5–2 with their equivalent block diagrams (Figure 5–14 and Figure 5–15):

```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
```

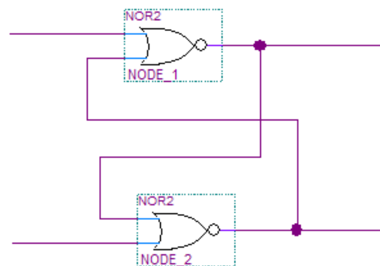
```
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
</AND>
```

Figure 5–14. Undesired Condition 1




```
<AND>
 <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NOR" TO_NAME="NODE_2" TO_TYPE="NOR" />
 <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NOR" TO_NAME="NODE_1" TO_TYPE="NOR" />
</AND>
```

Figure 5–15. Undesired Condition 2



Relating Nodes to a Clock Domain

Example 5–3 shows how to use the `CLOCK_RELATIONSHIP` attribute to relate nodes to clock domains. This example checks for correct synchronization in data transfer between asynchronous clock domains. Synchronization is done using cascaded registers, also called synchronizers, at the receiving clock domain. The code in Example 5–3 checks for the synchronizer configuration based on the following guidelines:

- The cascading registers need to be triggered on the same clock edge
- Do not put any logic between the register output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain

Example 5–3. Detecting Incorrect Synchronizer Configuration

```
<DA_RULE ID="EX02" SEVERITY="HIGH" NAME="Data Transfer Not Synch Correctly"
DEFAULT_RUN="YES">

<RULE_DEFINITION>
<DECLARE>
   <NODE NAME="NODE_1" TYPE="REG" />
   <NODE NAME="NODE_2" TYPE="REG" />
   <NODE NAME="NODE_3" TYPE="REG" />
</DECLARE>
<FORBID>
   <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />
   <NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
   <OR>
      <NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
      <CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
   </OR>
</FORBID>
</RULE_DEFINITION>

<REPORTING_ROOT>
<MESSAGE NAME="Rule %ARG1%: Found %ARG2% node(s) related to this rule.">
   <MESSAGE_ARGUMENT NAME="ARG1" TYPE="ATTRIBUTE" VALUE="ID" />
   <MESSAGE_ARGUMENT NAME="ARG2" TYPE="TOTAL_NODE" VALUE="NODE_1" />
   <MESSAGE NAME="Source node(s): %ARG3%, Destination node(s): %ARG4%">
      <MESSAGE_ARGUMENT NAME="ARG3" TYPE="NODE" VALUE="NODE_1" />
      <MESSAGE_ARGUMENT NAME="ARG4" TYPE="NODE" VALUE="NODE_2" />
   </MESSAGE>
</MESSAGE>
</REPORTING_ROOT>
</DA_RULE>
```

The codes differentiate the clock domains. `ASYN` means asynchronous, and `!ASYN` means non-asynchronous. This notation is useful for describing nodes that are in different clock domains. The following lines from Example 5–3 state that `NODE_2` and `NODE_3` are in the same clock domain, but `NODE_1` is not.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />
```

The next line of code states that `NODE_2` and `NODE_3` have a clock relationship of either sequential edge or asynchronous.

```
 <CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

The `<FORBID></FORBID>` section contains the undesirable condition for the design, which in this case is the undesired configuration of the synchronizer. If the condition is fulfilled, the Design Assistant highlights a rule violation.

The following examples are the undesired conditions from Example 5–3 with their equivalent block diagrams (Figure 5–16 and Figure 5–17):

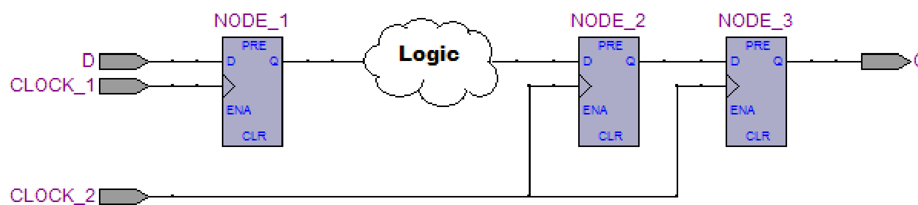
Example 5–4.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
REQUIRED_THROUGH="YES" THROUGH_TYPE="COMB" CLOCK_RELATIONSHIP="ASYN" />
```

Figure 5–16. Undesired Condition 3



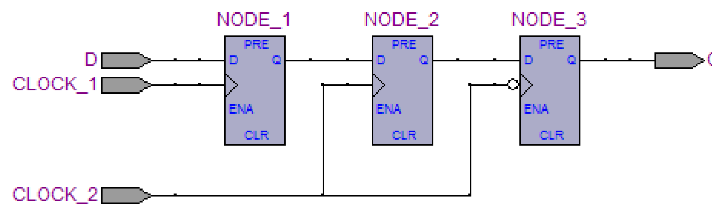
Example 5–5.

```
<NODE_RELATIONSHIP FROM_NAME="NODE_1" TO_NAME="NODE_2" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="ASYN" />

<NODE_RELATIONSHIP FROM_NAME="NODE_2" TO_NAME="NODE_3" TO_PORT="D_PORT"
CLOCK_RELATIONSHIP="!ASYN" />

<CLOCK_RELATIONSHIP NAME="SEQ_EDGE|ASYN" NODE_LIST="NODE_2, NODE_3" />
```

Figure 5–17. Undesired Condition 4



Targeting Clock and Register-Control Architectural Features

In addition to following general design guidelines, it is important to code your design with the device architecture in mind. FPGAs provide device-wide clocks and register control signals that can improve performance.

Clock Network Resources

Altera FPGAs provide device-wide global clock routing resources and dedicated inputs. Use the FPGA's low-skew, high fan-out dedicated routing where available. By assigning a clock input to one of these dedicated clock pins or using a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

In ASIC design, balancing clock delay as it is distributed across the device is important. Because Altera FPGAs provides device-wide global clock routing resources and dedicated inputs, there is no need to manually balance delays on the clock network.

Altera recommends limiting the number of clocks in your design to the number of dedicated global clock resources available in your FPGA. Clocks feeding multiple locations that do not use global routing may exhibit clock skew across the device that could lead to timing problems. In addition, when you use combinational logic to generate an internal clock, it adds delays on the clock line. In some cases, delay on a clock line can result in a clock skew greater than the data path length between two registers. If the clock skew is greater than the data delay, the timing parameters of the register (such as hold time requirements) are violated and the design will not function correctly.

Current FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. Many large FPGA devices provide dedicated global clock networks, regional clock networks, and dedicated fast regional clock networks. These clocks are typically organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically a number of dedicated clock pins to drive either the global or regional clock networks and both PLL outputs and internal clocks can drive various clock networks.

To reduce clock skew within a given clock domain and ensure that hold times are met within that clock domain, assign each clock signal to one of the global high fan-out, low-skew clock networks in the FPGA device. The Quartus II software automatically uses global routing for high fan-out control signals, PLL outputs, and signals feeding the global clock pins on the device. You can make explicit Global Signal logic option settings by turning on the **Global Signal** option settings. On the Assignments menu, click **Assignment Editor**. Use this option when it is necessary to force the software to use the global routing for particular signals.

To take full advantage of these routing resources, the sources of clock signals in a design (input clock pins or internally-generated clocks) need to drive only the clock input ports of registers. In older Altera device families (such as FLEX® 10K and ACEX® 1K), if a clock signal feeds the data ports of a register, the signal may not be able to use dedicated routing, which can lead to decreased performance and clock skew problems. In general, allowing clock signals to drive the data ports of registers is not considered synchronous design and can complicate timing analysis. Altera does not recommend this practice.

Reset Resources

ASIC designs may use local resets to avoid long routing delays on the signal. Take advantage of the device-wide asynchronous reset pin available on most FPGAs to eliminate these problems. This reset signal provides low-skew routing across the device.

 © November 2009 Altera Corporation

Register Control Signals

Avoid using an asynchronous load signal if the design target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture provides only one of those control signals. Stratix III devices, for example, directly support an asynchronous clear function, but not a preset or load function. When the target device does not directly support the signals, the synthesis or place-and-route software must use combinational logic to implement the same functionality. In addition, if you use signals in a priority other than the inherent priority in the device architecture, combinational logic may be required to implement the desired control signals. Combinational logic is less efficient and can cause glitches and other problems; it is best to avoid these implementations.

For Verilog HDL and VHDL examples of registers with various control signals, and information about the inherent priority order of register control signals in Altera device architecture, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

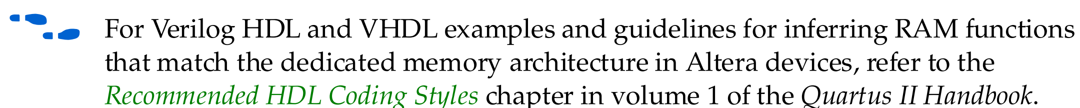
Targeting Embedded RAM Architectural Features

Altera's dedicated memory architecture offers many advanced features that you can target easily using the MegaWizard™ Plug-In Manager or using the recommended HDL coding styles that infer the appropriate RAM megafunction (ALTSYNCRAM or ALTDPRAM). Altera recommends using synchronous memory blocks for your design, so the blocks can be mapped directly into the device dedicated memory blocks. You can choose to use single-port, dual-port, or three-port RAM with a single- or dual-clocking method. Asynchronous memory logic is not inferred as a memory block or placed in the dedicated memory block, but is implemented in regular logic cells.

Altera memory blocks have differing read-during-write behaviors, depending on the targeted device family, memory mode, and block type. Read-during-write behavior refers to read and write from the same memory address in the same clock cycle; for example, you read from the same address to which you write in the same clock cycle.

It is important to check how you specify the memory in your HDL code when you use read-during-write behavior. The HDL code describes that the read returns either the old data at the memory location, or the new data being written to the memory location. The old data refers to the data stored in the memory location. The new data refers to the data that is being written to the memory location.

In some cases, when the device architecture cannot implement the memory behavior described in your HDL code, the memory block is not mapped to the dedicated RAM blocks, or the memory block is implemented using extra logic in addition to the dedicated RAM block. Altera recommends that you implement the read-during-write behavior using single-port RAM in Arria® GX devices and the Stratix and Cyclone series of devices to avoid this extra logic implementation.



For Verilog HDL and VHDL examples and guidelines for inferring RAM functions that match the dedicated memory architecture in Altera devices, refer to the *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*.

In many synthesis tools, you can specify that the read-during-write behavior is not important to your design; if, for example, you never read and write from the same address in the same clock cycle. For Quartus II integrated synthesis, add the synthesis attribute `ramstyle="no_rw_check"` to allow the software to choose the read-during-write behavior of a RAM, rather than using the read-during-write behavior specified in your HDL code. Using this type of attribute prevents the synthesis tool from using extra logic to implement the memory block and, in some cases, can allow memory inference when it would otherwise be impossible.

For details about using the `ramstyle` attribute, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For information about the synthesis attributes in other synthesis tools, refer to your synthesis tool documentation, or to the appropriate chapter in the *Synthesis* section in volume 1 of the *Quartus II Handbook*.

Conclusion

Following the design practices described in this chapter can help you to consistently meet your design goals. Asynchronous design techniques may result in incomplete timing analysis, may cause glitches on data signals, and may rely on propagation delays in a device leading to race conditions and unpredictable results. Taking advantage of the architectural features in your FPGA device can also improve the quality of your results.

Referenced Documents

This chapter references the following documents:

- *AN 437: Power Optimization in Stratix III FPGAs*
- *AN 514: Power Optimization in Stratix IV FPGAs*
- *Design Guidelines for HardCopy Series Devices* chapter in the *HardCopy Series Handbook*
- *Power Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *PowerPlay Power Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Quartus II Classic Timing Analyzer* 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 Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*
- *Recommended HDL Coding Styles* chapter in volume 1 of the *Quartus II Handbook*

Document Revision History

Table 5–4 shows the revision history for this chapter.

Table 5–4. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2009 v9.1.0	Removed documentation of obsolete rules.	Updated for the Quartus II software version 9.1 release.
March 2009 v9.0.0	No change to content.	Updated for the Quartus II software version 9.0 release.
November 2008 v8.1.0	■ Changed to 8-1/2 x 11 page size. ■ Added new section "Custom Rules Coding Examples" on page 5–36 ■ Added paragraph to "Recommended Clock-Gating Methods" on page 5–11. ■ Added new section: "Design Techniques to Save Power" on page 5–12.	Updated for the Quartus II software version 8.1 release.
May 2008 v8.0.0	■ Updated Figure 5–9 on page 5–13; added custom rules file to the flow ■ Added notes to Figure 5–9 on page 5–13 ■ Added new section: "Custom Rules Report" on page 5–34 ■ Added new section: "Custom Rules" on page 5–34 ■ Added new section: "Targeting Embedded RAM Architectural Features" on page 5–38 ■ Minor editorial updates throughout the chapter ■ Added hyperlinks to referenced documents throughout the chapter	Updated for the Quartus II software version 8.0 release.

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