

This chapter provides design recommendations for Altera® devices and describes the Quartus® II Design Assistant, which helps you check your design for violations of Altera's design recommendations. Current FPGA applications have reached the complexity and performance requirements of ASICs. In the development of complex system designs, good design practices have an enormous impact on the timing performance, logic utilization, and system reliability of a device. Well-coded designs behave in a predictable and reliable manner even when retargeted 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, you should 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 contains the following sections:

- “Synchronous FPGA Design Practices” on page 9–2
- “Design Guidelines” on page 9–4
- “Checking Design Violations With the Design Assistant” on page 9–13
- “Targeting Clock and Register-Control Architectural Features” on page 9–19
- “Targeting Embedded RAM Architectural Features” on page 9–24

 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 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*.

 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*.

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 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 you ensure that all the timing requirements of the registers 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 a HardCopy device or if you are prototyping an ASIC design.

Fundamentals of Synchronous Design

In a synchronous design, the clock signal controls the activities of all inputs and outputs. 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 several transitions and finally settle to new values. Changes that occur 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 you meet the following timing requirements:

- Before an active clock edge, you must ensure that the data input has been stable for at least the setup time of the register.
- After an active clock edge, you must ensure that 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 TimeQuest Timing Analyzer reports actual hardware requirements for the setup times (t_{SU}) and hold times (t_H) for every pin in 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 in your device.



To meet setup and hold time requirements on all input pins, any inputs to combinational logic that feed 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 device to help prevent a violation of the required setup and hold times.

When you violate the setup or hold time of a register, 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.

- ② For information about timing requirements and analysis in the Quartus II software, refer *About TimeQuest Timing Analysis* in Quartus II Help.

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.

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 due to 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 9-4*. Relying on a particular delay also makes asynchronous designs 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 several 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, you should 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 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.

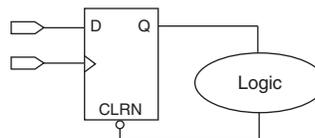
Combinational Logic Structures

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, with either logic elements (LEs) or adaptive logic modules (ALMs). For some cases in which combinational logic feeds registers, the register control signals can 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 9-1](#).

Figure 9-1. 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 TimeQuest Timing Analyzer, refer to *Specifying Timing Constraints and Exceptions (TimeQuest Timing Analyzer)* in Quartus II Help for details about how the TimeQuest analyzer performs recovery and removal analysis.

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. You can implement latches with the Quartus II Text Editor or Block Editor. 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 cannot identify these safe applications.

The TimeQuest analyzer analyzes latches as synchronous elements by default, and allows you to treat latches as having nontransparent start and end points. Be aware that even an instantaneous transition through transparent mode can lead to glitch propagation. The TimeQuest analyzer cannot perform cycle-borrowing analysis; this is performed instead by the Synopsys PrimeTime third-party timing analysis tool.

Due to various timing complexities, latches have limited support in formal verification tools. Therefore, you should not rely on formal verification for a design that includes latches.



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

Delay Chains

Delay chains occur when you use two or more consecutive nodes with a single fan-in and a single fan-out 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.

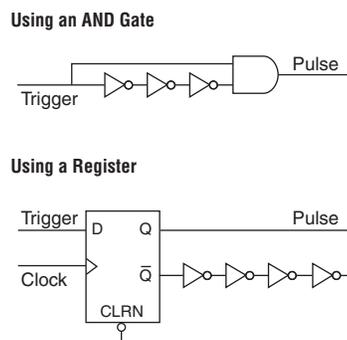
Delays in PLD designs can change with each placement and routing cycle. Effects such as rise and fall time differences and on-chip variation mean that delay chains, especially those placed on clock paths, can cause significant problems in your design. Refer to “[Hazards of Asynchronous Design](#)” on page 9-3 for examples of the kinds of problems that delay chains can cause. Avoid using delay chains to prevent these kinds 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

You can use delay chains 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 9-2](#). These techniques are purely asynchronous and must be avoided.

Figure 9-2. Asynchronous Pulse Generators



In [Figure 9-2](#), 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 feed the gate directly and the path 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.

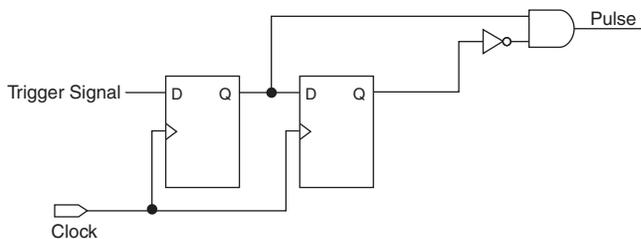
As also shown in [Figure 9-2](#), a register’s output drives the same register’s asynchronous reset signal through a delay chain. The register resets itself asynchronously after a certain delay.

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. Additionally, static timing analysis cannot be used to verify the pulse width, so verification is 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. Additionally, 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 9-3](#).

Figure 9-3. 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 the performance and reliability of a design. 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.

You use global device-wide, low-skew dedicated routing for all internally-generated clocks, instead of routing clocks on regular routing lines. For more information, refer to [“Clock Network Resources” on page 9-20](#).

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.

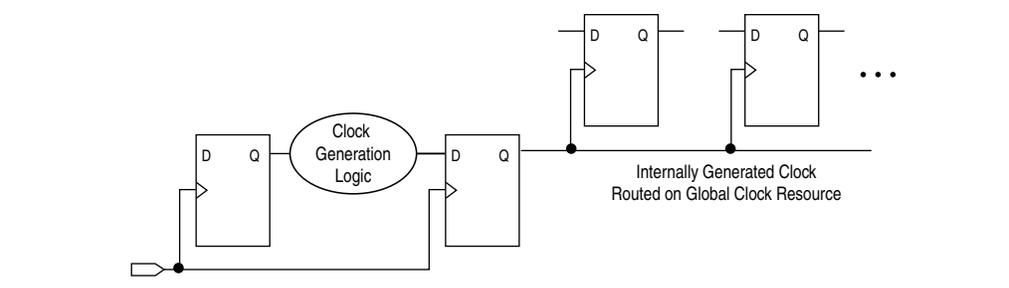
The following sections provide some specific examples and recommendations for avoiding clocking scheme problems.

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 might 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.

To avoid these problems, you should always register the output of combinational logic before you use it as a clock signal (Figure 9-4).

Figure 9-4. 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. Additionally, create your design so that registers always directly generate divided clock signals, as described in “Internally Generated Clocks”, 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

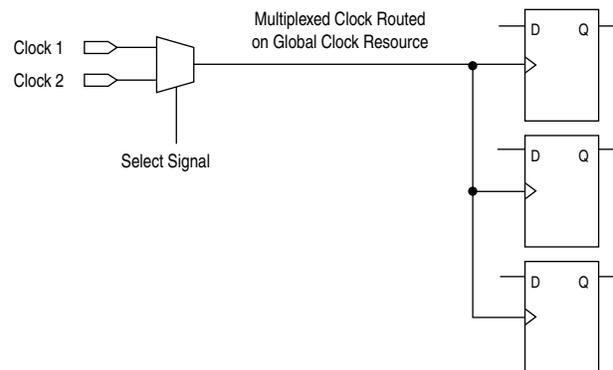
To simplify verification, avoid ripple counters in your design. In the past, FPGA designers implemented ripple counters to divide clocks by a power of two because the counters are easy to design and may use fewer gates than their synchronous counterparts. Ripple counters use cascaded registers, in which 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 must be handled properly during timing analysis, which can be difficult and may require you to make complicated timing assignments in your synthesis and placement and routing 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. You should avoid using ripple counters completely.

Multiplexed Clocks

Use clock multiplexing to operate the same logic function with different clock sources. In these designs, multiplexing selects a clock source, as shown in Figure 9-5. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Figure 9-5. 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

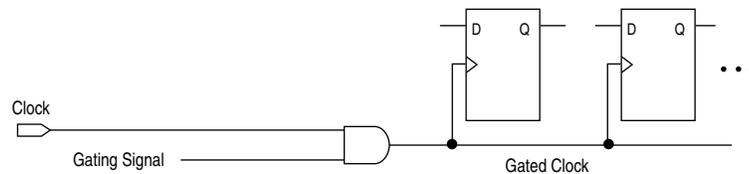
If the design switches clocks in real time with no reset signal, and your design cannot tolerate a temporarily incorrect response, you must use a synchronous design so that there are no timing violations on the registers, no glitches on clock signals, and no race conditions or other logical problems. 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-to-register paths are analyzed using that clock.

-  Use 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.
-  For device-specific information about clocking structures, refer to the appropriate device data sheet or handbook on the [Literature](#) page of the Altera website.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls gating circuitry, as shown in [Figure 9-6](#). When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive.

Figure 9-6. 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.

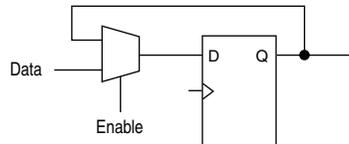
Use dedicated hardware to perform clock gating rather than an AND or OR gate. For example, you can use the clock control block in newer Altera devices to shut down an entire clock network. Dedicated hardware blocks ensure that you use global routing with low skew and avoid any possible hold time problems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

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 9-11.

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 clock enable signal 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 9-7).

Figure 9-7. Synchronous Clock Enable

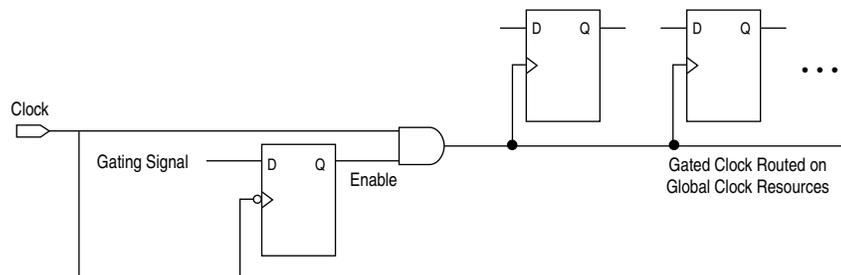


Recommended Clock-Gating Methods

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 9-8 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 9-8. Recommended Clock-Gating Technique



In the technique shown in Figure 9-8, 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 9-8. 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 9-8 on page 9-11](#), apply a clock setting to the output of the AND gate. Otherwise, the timing analyzer might 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 available for all devices that are supported by the TimeQuest analyzer (Arria® II, Arria II GX, Cyclone® II, Cyclone III, Cyclone IV, HardCopy series, Stratix® II, Stratix II GX, Stratix III, Stratix IV, and Stratix V 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*.

Power Optimization

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 placement and routing.

-  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](#).
-  Additionally, you can use the Quartus II PowerPlay suite of power analysis and optimization tools to help you during the design process by delivering fast and accurate estimations of power consumption. For information about the Quartus II PowerPlay suite of power analysis and optimization tools, refer to [About Power Estimation and Analysis](#) in Quartus II Help.

Metastability

Metastability in Altera designs can be caused by the synchronization of asynchronous signals. You can use the Quartus II software to analyze the mean time between failures (MTBF) due to metastability, thus optimizing the design to improve the metastability MTBF. A high metastability MTBF indicates a more robust design.

-  For more information about how to ensure complete and accurate metastability analysis, refer to the *Managing Metastability With the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.
-  For more information about viewing metastability reports, refer to *Viewing Metastability Reports* in Quartus II Help.

Incremental Compilation

The incremental compilation feature in the Quartus II software allows you to partition your design, separately compile partitions, and reuse the results for unchanged partitions. Incremental compilation flows require more up-front planning than flat compilations, and generally require you to be more rigorous about following good design practices than flat compilations.

-  For more information about incremental compilation and floorplan assignments, refer to the *Best Practices for Incremental Compilation Partitions and Floorplan Assignments* chapter in volume 1 of the *Quartus II Handbook*.
-  For more information about incremental compilation, refer to *About Incremental Compilation* in Quartus II Help.

Checking Design Violations With 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 the Design Assistant 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 that you specified.

This section provides an introduction to the Quartus II design flow with the 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. The Design Assistant supports all Altera devices supported by the Quartus II software.

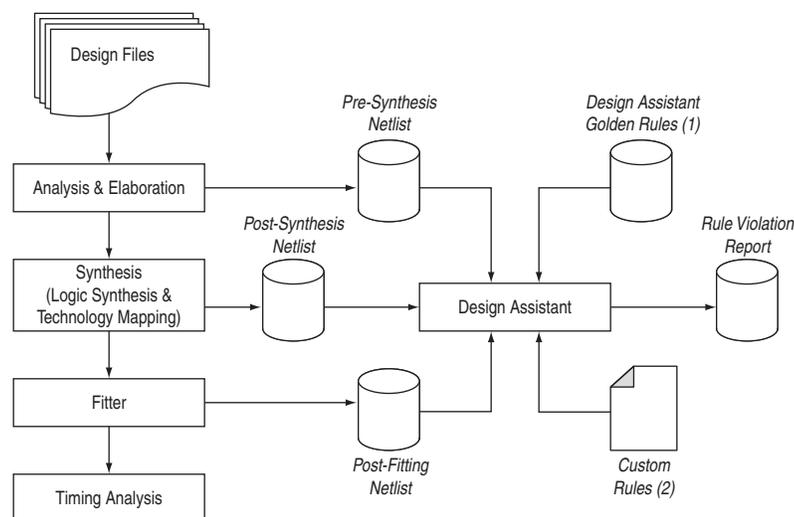
Quartus II Design Flow with the Design Assistant

You can run the Design Assistant after Analysis and Elaboration, Analysis and Synthesis, fitting, or a full compilation. If you set the Design Assistant to run automatically during compilation, the Design Assistant performs a post-fitting netlist analysis of your design. The default is to apply all of the rules to your project. 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.

- ② For more information about running the Design Assistant, refer to *About the Design Assistant* in Quartus II Help.

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

Figure 9-9. Quartus II Design Flow with the Design Assistant



Notes to Figure 9-9:

- (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 9-15.

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 might 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.

The exact operation of the Design Assistant depends on when you run it:

- 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 ←
```

- ④ For more information about Design Assistant settings, refer to *About the Design Assistant* and *Design Assistant Page (Settings Dialog Box)* in Quartus II Help.

Enabling and Disabling Design Assistant Rules

- ④ For more information about enabling or disabling Design Assistant rules on individual nodes by making an assignment in the Assignment Editor, in the Quartus II Settings File (`.qsf`), with the `altera_attribute` synthesis attribute in Verilog HDL or VHDL, or with a Tcl command, refer to *Enabling Design Assistant Rules on Nodes, Entities, or Instances*, or *Disabling Design Assistant Rules on Nodes, Entities, or Instances* in Quartus II Help.

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.

- ④ For information about the contents of the reports generated by the Design Assistant, refer to *Design Assistant Reports* in Quartus II Help.

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) with the XML format. You then specify the path to that file in the Design Assistant settings page and run the Design Assistant for violation 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`.

- ④ For more information about how to set the file path to your custom rules, refer to *Custom Rules Settings Dialog Box* in Quartus II Help. For more information about the basics of writing custom rules, the Design Assistant settings, and coding examples on how to check for clock relationship and node relationship in a design, refer to *Creating Custom Design Assistant Rules* in Quartus II Help. To specify the rules that you want the Design Assistant to use when checking for violations, refer to *Design Assistant Page (Settings Dialog Box)* in Quartus II Help.

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 9-1](#) shows the XML codes for checking SR latch structures in a design.

Example 9-1. 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>
```

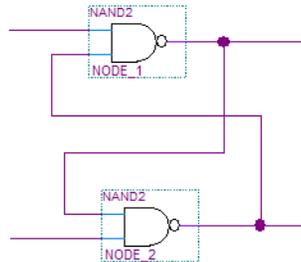
In [Example 9-1](#), the possible SR latch structures are specified in the rule definition section. Codes defined in the `<AND></AND>` block are tied together, meaning that each statement in the block must be true for the block to be fulfilled (AND gate similarity). In the `<OR></OR>` block, as long as one statement in the block is true, the block is fulfilled (OR gate similarity). If no `<AND></AND>` or `<OR></OR>` block are 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 9-1](#) with their equivalent block diagrams ([Figure 9-10](#) and [Figure 9-11](#)):

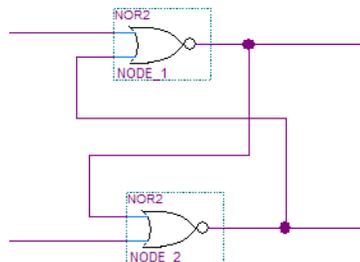
```
<AND>
  <NODE_RELATIONSHIP FROM_NAME="NODE_1" FROM_TYPE="NAND" TO_NAME="NODE_2"
TO_TYPE="NAND" />
  <NODE_RELATIONSHIP FROM_NAME="NODE_2" FROM_TYPE="NAND" TO_NAME="NODE_1"
TO_TYPE="NAND" />
</AND>
```

Figure 9-10. 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 9-11. Undesired Condition 2



Relating Nodes to a Clock Domain

[Example 9-2](#) 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 with cascaded registers, also called synchronizers, at the receiving clock domain. The code in [Example 9-2](#) checks for the synchronizer configuration based on the following guidelines:

- The cascading registers need to be triggered on the same clock edge

- There is no logic between the register output of the transmitting clock domain and the cascaded registers in the receiving asynchronous clock domain

Example 9-2. 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 9-2](#) 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 9-2](#) with their equivalent block diagrams ([Figure 9-12](#) and [Figure 9-13](#)):

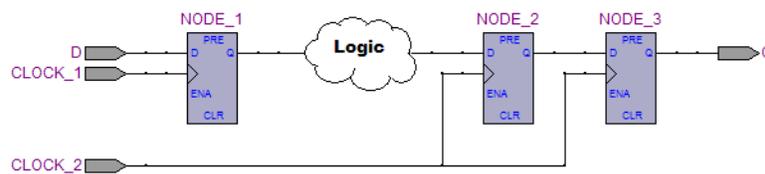
Example 9-3.

```
<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 9-12. Undesired Condition 3



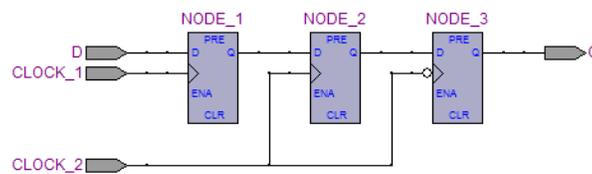
Example 9-4.

```
<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" />

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

Figure 9-13. Undesired Condition 4



Targeting Clock and Register-Control Architectural Features

In addition to following general design guidelines, you must 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 with a Quartus II logic option to assign global routing, you can take advantage of the dedicated routing available for clock signals.

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

You should limit 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 does not function correctly.

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 organized into a hierarchical clock structure that allows many clocks in each device region with low skew and delay. There are typically several dedicated clock pins to drive either global or regional clock networks, and both PLL outputs and internal clocks can drive various clock networks.

To reduce clock skew in a given clock domain and ensure that hold times are met in 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. 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.

Reset Resources

ASIC designs may use local resets to avoid long routing delays. 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.

Three types of resets are used in synchronous circuits:

- Synchronous Reset
- Asynchronous Reset
- Synchronized Asynchronous Reset—this type is preferred when designing an FPGA circuit

Synchronous Reset

The synchronous reset ensures that the circuit is fully synchronous. You can easily time it with static timing analyzer tools, such as the Quartus II TimeQuest analyzer. The synchronous reset is easier to use with cycle-based simulators.

However, the synchronous reset might require pulse stretchers to guarantee a reset pulse width wide enough to ensure that reset is present during an active edge of the clock. The synchronous reset requires a clock to reset a circuit. If the clock fails to launch, the resulting circuit is not reset.

Asynchronous Reset

The asynchronous reset is the most common form of reset used in circuit designs. Typically, you can insert the asynchronous reset into the device, turn on the global buffer, and connect to the asynchronous reset pin of every register in the device. This method is only advantageous under certain circumstances—you do not need to always reset the register. Unlike the synchronous reset, the asynchronous reset is not inserted in the data path, and does not negatively impact the data arrival times between registers. Reset takes effect immediately, and as soon as the registers receive the reset pulse, the registers are reset. The asynchronous reset is not dependent on the clock.

However, when the reset is deasserted and does not pass the recovery (μt_{SU}) or removal (μt_{H}) time check (both times are checked by the TimeQuest analyzer Recovery and Removal Analysis), the edge is said to have fallen in the metastability zone. Additional time is required to determine the correct state, and the delay can cause the setup time to fail to register downstream, leading to system failure.

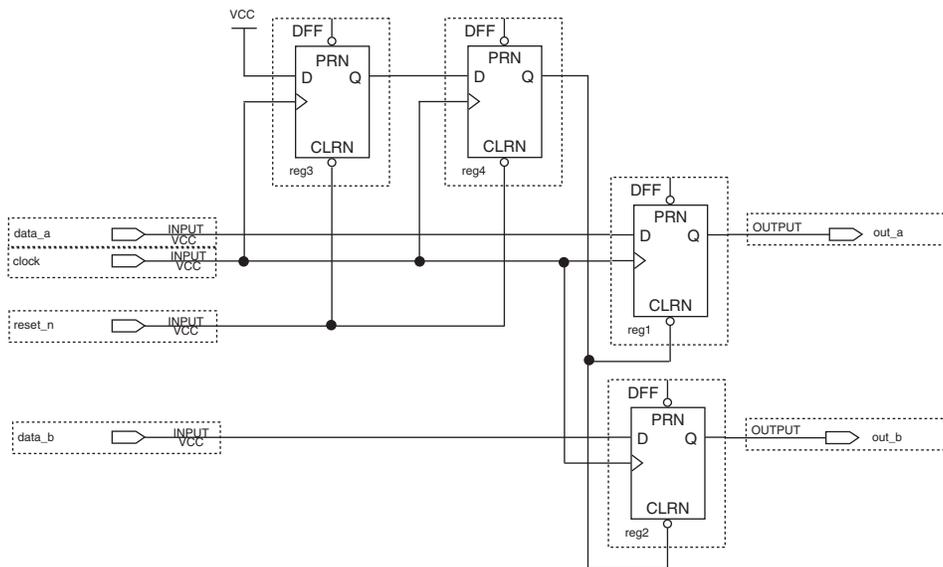
The asynchronous reset is susceptible to noise, and a noisy asynchronous reset can cause a spurious reset. You must ensure that the asynchronous reset is debounced and filtered. You can easily enter into a reset asynchronously, but releasing a reset asynchronously can lead to potential problems (also referred to as “reset removal”) with metastability, including the hazards of unwanted situations with synchronous circuits involving feedback.

Synchronized Asynchronous Reset

To avoid potential problems associated with purely synchronous resets and purely asynchronous resets, you can use synchronized asynchronous resets. Synchronized asynchronous resets combine the advantages of synchronous and asynchronous resets. These resets are asynchronously asserted and synchronously deasserted. This takes effect almost instantaneously, and ensures that no data path for speed is involved, and that the circuit is synchronous for timing analysis and is resistant to noise.

Figure 9-14 shows a method for implementing the synchronized asynchronous reset. You should use synchronizer registers in a similar manner as synchronous resets. However, the asynchronous reset input is gated directly to the CLRN pin of the synchronizer registers and immediately asserts the resulting reset. When the reset is deasserted, logic "1" is clocked through the synchronizers to synchronously deassert the resulting reset.

Figure 9-14. Schematic of Synchronized Asynchronous Reset



Example 9–5 shows the equivalent Verilog code. The active edge of the reset is used in the sensitivity list for the blocks in Figure 9–14.

Example 9–5. Verilog Code for Synchronized Asynchronous Reset

```
module sync_async_reset (
    input    clock,
    input    reset_n,
    input    data_a,
    input    data_b,
    output   out_a,
    output   out_b
);

reg    reg1, reg2;
reg    reg3, reg4;

assign out_a    = reg1;
assign out_b    = reg2;
assign rst_n    = reg4;

always @ (posedge clock, negedge reset_n)
begin
    if (!reset_n)
    begin
        reg3    <= 1'b0;
        reg4    <= 1'b0;
    end
    else
    begin
        reg3    <= 1'b1;
        reg4    <= reg3;
    end
end

always @ (posedge clock, negedge rst_n)
begin
    if (!rst_n)
    begin
        reg1    <= 1'b0;
        reg2    <= 1'b0;
    end
    else
    begin
        reg1    <= data_a;
        reg2    <= data_b;
    end
end

endmodule // sync_async_reset
```

To minimize the metastability effect between the two synchronization registers, and to increase the MTBF, the registers should be located as close as possible in the device to minimize routing delay. If possible, locate the registers in the same logic array block (LAB). The input reset signal (`reset_n`) must be cut with a `set_false_path` command, so the reset that comes from the synchronization register (`rst_n`) can be timed in the TimeQuest analyzer with Recovery and Removal Analysis.



For more information about specifying the minimum routing delay, refer to the *Best Practices for the Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook*.

The circuit in [Figure 9-14 on page 9-22](#) ensures that the synchronized asynchronous reset is at least one full clock period in length. To extend this time to n clock periods, you must increase the number of synchronizer registers to $n + 1$. You must connect the asynchronous input reset (`reset_n`) to the `CLRN` pin of all the synchronizer registers to maintain the asynchronous assertion of the synchronized asynchronous reset.

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 these 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 placement and routing 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 necessary 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 with the MegaWizard™ Plug-In Manager or with the recommended HDL coding styles that infer the appropriate RAM megafunction (ALTSYNCRAM or ALTDPRAM). Use synchronous memory blocks for your design, so the blocks can be mapped directly into the device dedicated memory blocks. You can 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 different 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 that describes the read returns either the old data stored at the memory location, or the new data 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. Implement the read-during-write behavior using single-port RAM in Arria GX devices and the Cyclone and Stratix 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.

Document Revision History

Table 9-1 shows the revision history for this chapter.

Table 9-1. Document Revision History

Date	Version	Changes
December 2010	10.1.0	<ul style="list-style-type: none"> ■ Title changed from Design Recommendations for Altera Devices and the Quartus II Design Assistant. ■ Updated to new template. ■ Added references to Quartus II Help for “Metastability” on page 9-13 and “Incremental Compilation” on page 9-13. ■ Removed duplicated content and added references to Quartus II Help for “Custom Rules” on page 9-15.
July 2010	10.0.0	<ul style="list-style-type: none"> ■ Removed duplicated content and added references to Quartus II Help for Design Assistant settings, Design Assistant rules, Enabling and Disabling Design Assistant Rules, and Viewing Design Assistant reports. ■ Removed information from “Combinational Logic Structures” on page 5-4 ■ Changed heading from “Design Techniques to Save Power” to “Power Optimization” on page 5-12 ■ Added new “Metastability” section ■ Added new “Incremental Compilation” section ■ Added information to “Reset Resources” on page 5-23 ■ Removed “Referenced Documents” section
November 2009	9.1.0	<ul style="list-style-type: none"> ■ Removed documentation of obsolete rules.
March 2009	9.0.0	<ul style="list-style-type: none"> ■ No change to content.
November 2008	8.1.0	<ul style="list-style-type: none"> ■ Changed to 8-1/2 x 11 page size ■ Added new section “Custom Rules Coding Examples” on page 5-18 ■ Added paragraph to “Recommended Clock-Gating Methods” on page 5-11 ■ Added new section: “Design Techniques to Save Power” on page 5-12
May 2008	8.0.0	<ul style="list-style-type: none"> ■ 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

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

 Take an [online survey](#) to provide feedback about this handbook chapter.