
Design Guidelines for Optimal Results in High-Density FPGAs

Introduction

Today's FPGA applications are approaching the complexity and performance requirements of ASICs. In some cases, FPGAs are used to prototype large ASIC designs. In these complex designs, good design practices have an enormous impact on your FPGA's timing performance, logic utilization, and system reliability. Designs coded optimally will behave in a predictable and reliable manner, even when re-targeted to different device families or speed grades. Good design practices also aid in successful design migration between FPGA and ASIC implementations for both prototyping and production. For optimal performance when designing FPGAs, understand the impact of synchronous design practices, follow recommended design techniques, and target the advanced architectural features of your FPGA.

Synchronous FPGA Design Practices

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

Synchronous Design

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

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, combinational logic (feeding the data inputs of registers) changes 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 inputs from outputs, instability in the combinational logic does not affect the intended operation of the design as long as the following timing requirements are met:

- Before an active clock edge, the data input is settled 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 the setup or hold 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. Such a state is not fully stable and small perturbations, like noise in power rails, can return the register to a 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 time.

Hazards of Asynchronous Design

In the past, designers have often used asynchronous techniques such as ripple counters or pulse generators in FPGA designs, enabling designers to take “short cuts” and save device resources. Asynchronous design techniques have inherent problems such as relying on propagation delays within a device, leading to incomplete timing constraints, and generating possible glitches and spikes. Because FPGAs provide large amounts of high-performance logic gates, registers, and memory, resource and performance trade-offs have changed, and you should focus on design practices that help you meet your design goals consistently.

Some asynchronous design structures rely on the relative propagation delays of signals to function correctly. Design structures in FPGAs can have varying timing delays, depending on how the design is placed-and-routed in the device with each compile. 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 process improvements, the delays in an asynchronous design may decrease, resulting in a design that does not function as expected. Relying on a particular delay makes asynchronous designs very difficult to migrate to different devices or speed grades, including HardCopy devices or ASICs.

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, your synthesis and place-and-route tools’ timing-driven algorithms may not be able to perform the best optimizations, and reported results will not be complete.

Some asynchronous design structures can generate harmful glitches, pulses that are very short compared to 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. Therefore, incorrect values can be propagated 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.

Recommended Design Techniques

When designing with HDL code, it is important to understand how a synthesis tool interprets different HDL coding styles and the results to expect. Your coding style can affect logic utilization and timing performance. This section discusses some basic design techniques to ensure optimal synthesis results for FPGA designs 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 to maintain synchronous functionality.

Combinational Logic Structures

Combination Loops

Combinational loops are among the most common causes of instability and unreliability in digital designs. In a synchronous design, all feedback loops should include registers. Combinational loops violate synchronous design principles by establishing a direct feedback with no registers. For example, a combinational loop occurs when the left-hand side of an arithmetic expression also appears on the right-hand side. 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 1.

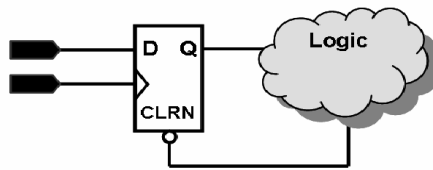


Figure 1. Combinational Loop through Asynchronous Control Pin

Combinational loops are inherently high-risk design structures because:

- Combinational loop behavior generally depends on the relative propagation delays through the logic involved in the loop. As discussed, propagation delays can change and the behavior of the loop may change.
- Combinational loops can cause endless computation loops in many design tools. Most tools break open combinatorial loops in order to proceed. The various tools used in the design flow may open a given loop a different manner, processing it in a way that may not be consistent with the original design intent.

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. Often inverters are chained together to add delay. Delay chains generally result from asynchronous design practices, and are sometimes used to resolve race conditions created by other combinational logic. As discussed above, FPGA delays can change with each place-and-route. Delay chains can cause various design problems, including an increase in a design's sensitivity to operating conditions, a decrease in a design's reliability, and difficulties when migrating to another device architecture.

In some ASIC designs, delays may be used for buffering signals as they are routed around the chip. This functionality isn't needed in FPGAs because the local routing provides buffers throughout the device.

Avoid using delay chains in your design; rely on synchronous practices instead.

Pulse Generators and Multi-Vibrators

Designers sometimes use delay chains to generate either one pulse (pulse generators) or a series of pulses (multi-vibrators). There are two common methods for pulse generation as shown in Figure 2 a) and b); these techniques are purely asynchronous and should be avoided:

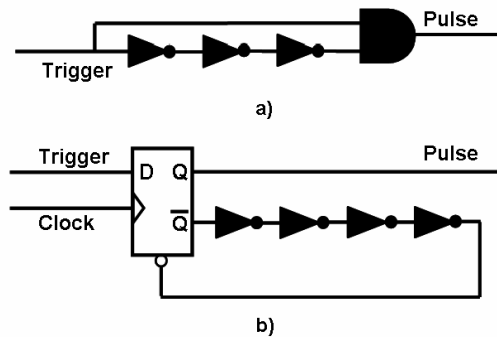


Figure 2. Asynchronous Pulse Generators

- a. A trigger signal feeds both inputs of a 2-input AND or OR 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 inputs. This technique artificially increases the width of the glitch by using a delay chain.
- b. A register's output drives the same register's asynchronous reset signal through a delay chain. The register essentially resets itself asynchronously after a certain delay.

The main problem with these designs is that the pulse widths are difficult for your synthesis and place-and-route software to determine, set, or verify. The actual pulse width can only be determined when routing and propagation delays are known, after placement and routing. So it is difficult to reliably determine the width of the pulse when creating HDL code and it can't be set by your EDA tools. The pulse may not be wide enough for the application in all PVT conditions, and the pulse width will change if you migrate to a different device. In addition, static timing analysis can not be used to verify the pulse width so verification is very difficult.

Multi-vibrators use the principle of the “glitch generator” to create pulses, in addition to a combinational loop that turns the circuit into an oscillator. Structures that generate multiple pulses cause even more problems than pulse generators because of the number of pulses involved. In addition, when the structures generate multiples pulses, they also create a new artificial clock in the design.

When you need to use a pulse generator, it should be implemented based on purely synchronous techniques as shown in Figure 3.

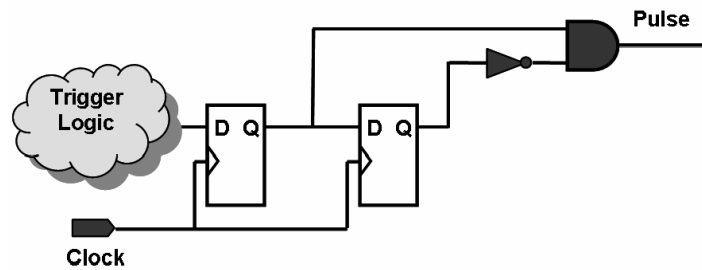


Figure 3. Recommended Pulse-Generation Technique

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 migrated to other architectures, devices or speed grades.

Latches

In digital logic, latches hold the value of a signal until a new value is assigned. Latches can also be inferred from HDL code to hold a value when the designer did not intend to use a latch. FPGAs are register-intensive; therefore, designing with latches uses more logic and leads to lower performance than designing with registers.

Latches can cause various difficulties in the design. Although latches are memory elements like registers, they are fundamentally different. When a latch is in a feed-through or transparent mode, there is a direct path between the data input and the output. Glitches on the data input can pass to the output. The timing for latches is also inherently ambiguous. When analyzing a design with a D latch, for example, the software cannot determine whether you intended to transfer data to the output on the leading edge of the clock or on the trailing edge. In many cases, only the original designer knows the full intent of the design, meaning another designer cannot easily migrate the design or reuse the code.

When designing combinational logic, certain coding styles can create an unintentional latch. For example, when CASE or IF statements do not cover all possible input conditions, latches may be required to hold the output if a new output value is not assigned. Omitting the final ELSE clause or WHEN OTHERS clause in an IF or CASE statement can also generate a latch. To avoid creating unintentional latches, assign the default CASE or final ELSE statement to a “don’t care” value.

Clocking Schemes

Internally Generated Clocks

Avoid using internally generated clocks where 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 due to the combinational logic can lead to timing problems.

If you use the output of combinational logic as a clock signal or as an asynchronous reset signal, you should 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 of 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, always register the output of combinational logic before you use it as a clock signal (see Figure 4).

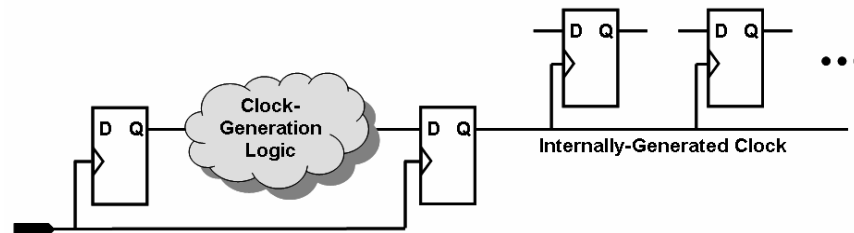


Figure 4. Recommended Clock-Generation Technique

This registering ensures that the glitches generated by the combinational logic are blocked on the data input of the register. The combinational logic used to generate an internal clock also adds delays on the clock line. In some cases, logic 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 will be violated and the design will not function correctly. To reduce the clock skew within the clock domain, assign the generated clock signal to one of the high-fan-out and low-skew clock trees in the FPGA device (if available). Using a low-skew clock tree such as Altera's global signals can help reduce the overall clock skew for the signal. In Altera's Quartus® II software, you can assign a node as a global signal using the Assignment Editor.

Divided Clocks

Many designs require clocks created by dividing a master clock.

Many FPGAs provide dedicated circuitry for clock division, such as Altera's phase-locked loops (PLLs). Using PLL circuitry will avoid many of the problems that can be introduced by asynchronous clock division logic.

When using logic to divide a master clock, always use synchronous counters or state machines. In addition, create your design such that registers always directly generate divided clock signals, as detailed above. Your design should never decode the outputs of a counter or a state machine to generate clock signals; this type of implementation often causes glitches.

Ripple Counters

FPGA designers have often 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 as such in timing analysis, which can be difficult and may require you to at least enter appropriate timing assignments in your synthesis and place-and-route tools. You should try to avoid these types of structures to ease verification effort.

Multiplexed Clocks

Clock multiplexing can be used to operate the same logic function with different clock sources. Multiplexing logic of some kind selects a clock source. For example, telecommunications applications that deal with multiple frequency standards often use multiplexed clocks.

Adding multiplexing logic to the clock signal can lead to some of the problems discussed in the previous sections, but requirements for multiplexed clocks vary widely depending on the application. Clock multiplexing is acceptable if:

- The clock multiplexing logic does not change after initial configuration.
- The design uses multiplexing logic to select a clock for testing purposes.
- You always apply a reset when switching clocks.
- A temporarily incorrect response of the chip following clock switching has no consequences.

If the design switches clocks on the fly with no reset and your design cannot tolerate a temporarily incorrect response of the chip, then 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.

Gated Clocks

Gated clocks turn a clock signal on and off using an enable signal that controls some sort of gating circuitry. When a clock is turned off, the corresponding clock domain is shut down and becomes functionally inactive. Gated clocks can be a powerful technique to reduce power consumption. When gating a clock, both the clock tree and registers no longer toggle, eliminating their contributions to switching power.

Gated clocks are not part of a synchronous scheme and therefore can significantly increase your design implementation and verification effort, posing challenges in some cases. Gated clocks contribute to clock skew and make device migration difficult. These clocks are also sensitive to glitches, which can cause design failure.

From a functional point of view, you can shut down a clock domain in a purely synchronous manner using a synchronous clock enable. However, when using a synchronous clock enable scheme, the clock tree keeps toggling and the internal circuitry of each register remains active (although outputs do not change values), which does not reduce power consumption. Therefore, you should use a synchronous scheme outlined in the next section in most cases, but for major power reduction, see the clock-gating recommendation below.

Synchronous Clock Enables

To turn off a clock domain in a synchronous manner, use synchronous clock enables. Clock enable signals are efficiently supported in most FPGAs with a clock enable signal on the device registers. This scheme does not reduce power consumption because the clock tree and register internal circuitry keep toggling, but it will perform 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 (see Figure 5).

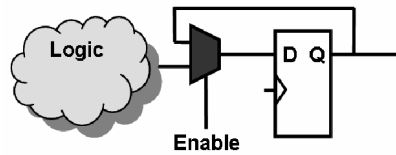


Figure 5. Synchronous Clock Enable

Recommended Clock-Gating Method

Only use gated clocks when your target application requires substantial power reduction. If you must use gated clocks, implement them using the robust clock-gating technique shown in Figure 6.

You can gate a clock signal at the root of the clock tree, at the leaves, or somewhere in between. Because the clock tree contributes to switching power, always generate the clock at the root so that you can shut down the entire clock tree instead of gating it further along the clock tree at the leaves.

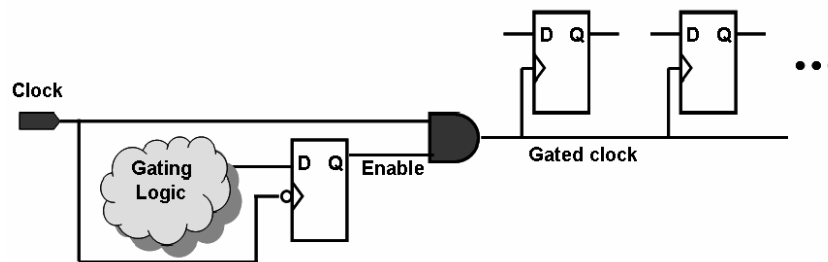


Figure 6. Recommended Clock Gating Technique

As shown in Figure 6, a register generates the enable command to ensure that it is free of glitches and spikes. The design clocks the register that generates the enable command 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 in Figure 6. With this implementation, only one input of the gate that turns the clock on and off changes at a time, which does not generate 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-edged register.

When using this scheme, you should pay attention to the duty cycle of the clock because only the on-time can generate the enable command. 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, the duty cycle is a minor issue compared to the problems created by other methods of gating clocks.

Targeting FPGA Architectural Features

As well as following general FPGA design guidelines, it is important to code your design with the target technology in mind. FPGAs provide device-wide control signals that can improve performance. FPGA devices contain high-performance logic gates, registers, memory elements, and advanced features like dedicated DSP circuitry. Take advantage of your FPGA architecture by following recommended

guidelines. You should understand how your synthesis tool will interpret different HDL design techniques and how your design will be mapped into your target device. Targeting specific FPGA architectural features can increase the performance and reduce the amount of logic required for a given design. When prototyping an ASIC, design for the more limited number of configurations for memory and other features in most FPGAs to more easily prototype your design.

Clock Tree and Device Control Signal Resources

In ASIC design, an important part of the design process can include balancing the clock delay as it is distributed across the chip. Most FPGA technology provides device-wide routing for clocks, so there is no need to manually balance any delays on the clock tree. Use the FPGA's low-skew, high-fan-out, dedicated routing where available. For example, Altera FPGAs provide global clock routing resources and dedicated inputs. By assigning a clock input to one of these dedicated clock pins, or making a logic assignment to a global signal, you can take advantage of the dedicated routing available for clock signals.

For best performance, limit the number of global clocks in your design to the number of dedicated global clock resources available in your FPGA.

Today's FPGAs offer increasing numbers of global clocks to address large designs with many clock domains. For example, Altera Stratix devices provide sixteen dedicated global clock networks, sixteen regional clock networks (four per device quadrant), and 8 dedicated fast regional clock networks. There are sixteen dedicated clock pins to drive either the global or regional clock networks. The Stratix enhanced and fast PLL outputs can also drive the global and regional clock networks, and internal signals within the design can be routed onto the clock networks using global logic assignments within the Quartus II software. These clocks are organized into a hierarchical clock structure that allows for up to 22 clocks per device region with low skew and delay, providing up to 48 unique clock domains within Stratix devices.

To take full advantage of these routing resources, clock signal sources in a design (input clock pins or internally generated clocks) should drive only input clock ports of registers. If a clock signal feeds the data ports of registers, the signal may not be able to use the dedicated routing which can lead to decreased performance.

Reset Resources

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

Register Control Signals

Avoid using an asynchronous load signal if the design's target device architecture does not include registers with dedicated circuitry for asynchronous loads. Also, avoid using both asynchronous clear and preset if the architecture only provides one of those control signals. When the target device does not directly support the signals, the place-and-route software has to use some combinational logic to implement the same functionality. The combinational logic is less efficient and can cause glitches and other problems, so it is best to avoid the implementations if possible.

Vendor-Specific IP Functions

FPGA vendors generally provide intellectual property (IP) blocks for many functions to help make your design process easier. For example, Altera provides parameterizable megafunctions that are optimized for

Altera device architectures. Megafunctions include the library of parameterized modules (LPM), device-specific embedded megafunctions such as PLLs, DSP blocks, LVDS drivers, IP available as Altera MegaCore® functions, and IP available from Altera Megafunction Partners Program (AMPPSM) partners.

Using IP functions instead of coding your own logic can save valuable design time. In addition, these functions can offer more efficient logic synthesis and device implementation. It is easy to scale megafunctions to different sizes by simply setting parameters. You also need to use megafunctions to access some device-specific features, such as memory, DSP blocks, LVDS drivers, PLLs, and DDR I/O circuitry. You can use vendor-specific functions by instantiating them in your HDL code or in some cases inferring them from generic HDL code.

Instantiating Vendor-Specific IP

You can instantiate IP functions in your HDL design like any other module or component, according to the port and parameter definitions. There may also be a software wizard available to help to parameterize the function and create a wrapper file, such as the Altera MegaWizard® Plug-In Manager. Wizards can provide a graphical interface for customizing and parameterizing the megafunction, and ensuring that you set all function parameters properly. When you finish setting parameters, the wizard typically generates a VHDL or Verilog HDL wrapper file that instantiates the megafunction with the correct parameters.

The one downside to instantiating vendor-specific IP functions is that it can make your code vendor specific unless the function can be easily replaced by a function for another vendor. You'll need to weigh this factor when evaluating vendor IP functions. Device migration and design reuse within the same vendor is usually supported when using these functions because they can generally be migrated to other device families from the same vendor.

Inferring Vendor-Specific IP from HDL

Many synthesis tools can automatically recognize certain types of HDL code and infer the appropriate vendor-specific function. That is, your place-and-route software will use the vendor's IP code when compiling your design even though you did not specifically instantiate the function. Inferring vendor-specific functions from generic HDL makes your design more vendor-independent. The software uses inference because, as discussed above, these functions (such as counters or multipliers) are optimized for the FPGA architecture, so the area and/or performance may be better than generic HDL code. In addition, you must use specific functions to access certain architecture-specific features (such as RAM, shift registers, and multiply-accumulators or multiply-adders in Altera DSP blocks) that generally provide improved performance compared to basic logic and registers. Inference is only supported for specific functions that can be fully and efficiently described in HDL.

Inferring vendor-specific functions may require specific HDL coding styles, as recommended by your synthesis tool or FPGA vendor, and there may be certain restrictions due to the device architecture. In some cases, the functionality of the function may differ slightly from your original HDL source code. Synthesis and place-and-route tools often have options that you can use to disable inference if you want to use generic logic or registers instead of the vendor's function.

As an example, memory blocks are often specific to a particular FPGA vendor, and ASIC memory compilers generally don't support FPGAs. EDA synthesis tools can infer Altera memory blocks from generic HDL code, making the block vendor-independent. Altera and the major synthesis tools offer guidelines for inferring single-port and simple dual-port memory in different device architectures. You cannot describe FPGA true dual-port RAM in Verilog HDL or VHDL because of language limitations related to reading and writing at the same time. In addition, using an Altera RAM megafunction for simple

dual-port RAM with separate read and write clocks slightly changes the design functionality if the RAM reads from and writes to the same location. This difference is also due to the HDL language definitions. In these cases, the software issues a warning and explains the condition under which the functionality changes. Your target architecture may have specific requirements as well. For example, Altera's Stratix™ devices contain synchronous memory blocks so certain signals must be registered when going into the device memory.

Inferring vendor-specific functions can help improve your FPGA design performance by taking advantage of your FPGA's advanced architectural features, and using optimized code for logic blocks. Writing generic code makes your design more vendor-independent than instantiating the functions directly. Refer to your synthesis tool's documentation for specific coding style guidelines, and take note of any coding requirements due to HDL limitations or FPGA architecture.

Conclusions

This paper discusses the impact of different design methodologies on your quality of results. Fundamental FPGA design guidelines and architecture-specific recommendations are presented. To ensure optimal results in your FPGA design or ASIC prototype, understand the impact of your design practices, follow recommended design techniques, and take advantage of advanced architectural features in your FPGA.



101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com

Copyright © 2003 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries.* All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.