

Introduction

Today's design application and performance requirements are more challenging due to increased complexity. With the evolution of system-on-a-chip designs, designs have grown larger. Additionally, external memory interfaces and mixed signal devices bring a greater challenge to timing closure. If you use third-party IP in your designs, you may not have control over how these IP blocks are pipelined, or how they are partitioned. Your design must accommodate timing requirements for the IP used in the system to achieve a fully functional design. When performance requirements for any part of a design are not completely met, the system fails to function as desired.

This application note focuses on a generic methodology for timing closure. Whether you use Application Specific Standard Products (ASSPs), Application Specific Integrated Circuits (ASICs), or Field Programmable Gate Arrays (FPGAs), timing closure poses a challenge for system design.

The Quartus® II Fitter default settings can help you meet required timing constraints for most designs. However, for some designs that cannot meet timing requirements with default settings, follow the methodology in this application note to achieve timing closure requirements.

Furthermore, the guidelines and methodology presented in this document can help improve productivity, close timing for your design faster, and reduce the number of iterations.



For more guidelines for different stages in the Altera® design flow, refer to the [Quartus II Design Checklist](#) on the Altera website.

This document contains the following topics:

- "What Makes a Design Complex?" on page 2
- "One Size Does Not Fit All" on page 3
- "How Does the Quartus II Fitter Work?" on page 3
- "Planning for Timing Closure" on page 4
- "Early Decisions Influence Your Results" on page 4
- "Planning for Timing Closure" on page 4
- "General Tips: Designing for Speed" on page 8
- "Best Practices for Timing Closure" on page 8
- "Common Timing Issues Encountered" on page 17
- "Tips for Tackling Timing Problems" on page 29

Market Pressures and Short Design Cycles

Current systems are characterized by shorter product life cycles, and market pressures require short design cycles. To be successful, a product must be designed, tested, and brought to the market quickly. Therefore, emphasis is on design verification because getting successful products at the first trial is important for a product to be economically viable.

The Quartus II software provides features that help you meet your performance goals. It focuses on accurate timing models, timing analysis, and fine-tuned Fitter algorithms. With the default settings, you can achieve push-button timing closure for most of your FPGA designs. For those designs where it is difficult to achieve push-button timing closure, the Quartus II software helps to plan for timing closure at the beginning of your design cycle to avoid delays at the end.

Increased System Complexity

With process technology shrinking to 40 nm, available device capacity has increased. These larger devices can support implementation of true system-on-a-chip designs. To optimize such designs, EDA tools (such as the Quartus II software) must run more complex algorithms.

What Makes a Design Complex?

Many factors can make timing closure difficult to achieve. For example, resource location could be a concern. Specialty blocks, such as DSP, transceivers, and M-RAMs are located in areas of the FPGA where routing availability can be problematic as a result of congestion around these blocks. Poor resource placement can result in timing not meeting all requirements.

Conflicts can occur between resource, area, power, and timing requirements in your design. For example, mobile devices must trade power for speed considerations. If your design requires more resources, the resources have to be spread out across the target device you have selected for implementation. Spread-out resources have long interconnections. At smaller device geometries, delays are dominated by interconnect delays rather than cell delays. To have shorter net lengths, you would ideally have to have a smaller area. Therefore, these two requirements generally conflict.

Another common conflict is between reliability and the time available for verification. Because of the reduced market window that dictates your product's success, the goal of a system designer is to have a design working within the shortest amount of time, at the lowest possible cost, and is simple, scalable, and reliable. To maximize the window of opportunity, you must shrink the design cycle. However, the requirement to have a successful design results in having to spend more time verifying the design. Therefore, these factors make closing timing on a design a complex issue.

One Size Does Not Fit All

You must consider a variety of features before choosing a device or a specific technology for your applications. For example, cost, operating speed, and power are some key points that you may have to consider during your system design. The EDA tools used to design these systems have to serve the needs of a diverse group of users and applications. The Quartus II software comes with a large set of options to suit a variety of system applications. The default settings generally give the best optimization trade-offs, but you may have to choose settings that are different from the default settings, because each design is different.

How Does the Quartus II Fitter Work?

In Altera FPGA designs, timing margins for critical paths are determined by how the Quartus II Fitter can optimize your design. When you run a compilation, the Quartus II Fitter creates a random initial placement for the logic based on the initial condition of your design. The goal of the Fitter is to find a placement that can be successfully routed and meets all constraints. It may be possible to achieve this with different initial placements, which means the solution is not unique. The Quartus II Fitter searches for the best solution among a set of different possible and valid solutions within a solution space. The Fitter might converge to different solutions in the solution space based on the given initial conditions.

The initial condition of the design is a function of all the source files, optimization settings, and an integer called the Fitter seed value. A change in any of these parameters results in a change in the initial placement. A change in initial placement affects how optimizations proceed, producing a different Fitter result. This is called the “seed effect.”

The seed effect determines how optimizations happen during a Fitter run. Because the project seed consists of inputs in the Quartus II software, any modification, such as changing a net name, pin name, or making a new assignment, changes the project seed and affects the final results.

A new Fitter run with a different Fitter seed puts the Quartus II Fitter into a new area in the solution space, resulting in a different initial placement. If you change the Fitter seed, the Quartus II software might converge on a different solution due to the modified initial placements. Therefore, changing the Fitter seed can give you different results.

A non-negative integer called “Fitter seed value” is used as an input to the Fitter seed. Changing this value may or may not produce better fitting, but it makes the Quartus II Fitter use a different initial placement. This integer seed value allows you to try different initial placements without any change in the design or settings.

Trying different initial placements to guide a Fitter run with different seeds or settings to find the best performance results for a given design using the same optimization settings is called a “seed sweep.”

Use a seed sweep to determine an optimal seed value for your design if other initial conditions remain unchanged. The default seed value used by the Fitter is 1. You can change the seed value to any other non-negative integer value on the **Fitter Settings** page of the **Settings** dialog box, in the **Seed** box.

Using different seed values causes a variance in the Fitter optimization results for compilations on the same design. For example, a seed value of 2 might have the best results for one of your designs, but when you make other changes in the design, either in the source or in the settings, that value might not produce the best results. Also, in a different design, a seed value of 1 might give the best results. On average, you can expect to see about $\pm 5\%$ variance in the results across different seed values.

There is no definitive seed value that guarantees the best results for every design. The varying results with seed value changes are design dependent.

Early Decisions Influence Your Results

Planning for timing closure early in the design cycle can help you identify issues before they arise. Early design stage decisions have a great effect on later phases of your design, such as how to partition the design, what will be the simulation strategy, and what will be the verification strategy. By considering these factors at the preliminary stages of the design, you can avoid problems that might arise later. You should not wait for all the blocks to be coded to compile the entire design for the first time.

Always use good synchronous design practices and HDL coding practices that are independent of your preferred EDA tools. To have an effective design, choose the target device architecture and properly constrain your design for timing. Identify the false and multi-cycle paths in your design to get an accurate timing analysis report. Your design is only as good as your constraints. Proper constraints help the software apply extra effort only where it is required.

For more details about synchronous design practices, good coding practices, and constraining your design for timing, refer to the [“Referenced Documents” on page 32](#).

Planning for Timing Closure

Proper planning can help you achieve timing closure faster. Because there are no set rules that work with every design, the best practices are fairly generic and applicable in many situations. To reduce design iterations and debug time, follow the guidelines in this section.

During the Specification Stage


Start planning for timing closure at the specification stage and decide how you would like to interface with the device in the target system before coding for the design blocks.


Create a block diagram with required details of how you would partition the desired functionality into specific blocks. There is no limit to how big or small a block can be. Very small design blocks might be difficult to track and very large design blocks difficult to debug. Try creating blocks that encapsulate distinct functionality. Keep them to a size that is convenient for debugging during functional simulation and during timing closure.

Device Selection

The devices in each Altera device family are available with different design densities, speed grades, and packaging options that can accommodate different applications. Choose a device that can meet your timing requirements. Some feature requirements to consider are performance, logic and memory density, I/O density, power utilization, packaging, and cost. The Quartus II software optimizes and analyzes your design with different timing models for each speed grade. If you migrate to a device with a different speed grade, perform timing analysis to ensure that there are no timing violations due to changes in timing performance.


Even though FPGAs offer design reconfiguration and bug fixing advantages over ASSPs and ASICs, good planning and discipline can have additional benefits for achieving correct design results quickly.

 For more information about choosing a device, refer to the *Design Planning with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook* and the *Quartus II Design Checklist* on the Altera website.

 For more information and details about Altera device family features, refer to the literature in the *Selector Guide* page of the Altera website.

Plan for On-Chip Debugging

The Quartus II software has on-chip debugging tools that offer different advantages and trade-offs, depending on the system, design, and user. Evaluate on-chip debugging options early in the design process to ensure that your system board, Quartus II project, and design are all set up to support the appropriate options. Timing errors due to unspecified timing requirements often appear as functional failures of the design. If you are able to locate the functional block where the errors originated, it is easier to find the source of the errors.

 For more information about debugging tools, refer to the *In-System Design Debugging* section in volume 3 of the *Quartus II Handbook* and the *Virtual JTAG (sld_virtual_jtag) Megafunction User Guide*.

During Coding and Compilation

The coding and compilation approach you use can help you achieve fast timing closure.

Plan for a Good Design Hierarchy

Flat designs are generally difficult to debug. Using a hierarchical design approach offers several advantages, such as:

- Hierarchical designs are easier to debug
- You can assign the design into logical partitions that are functionally independent
- These partitions help in stand-alone verification
- You can use the Quartus II incremental compilation feature to preserve synthesis and timing results for unmodified blocks

Planning ahead by appropriately partitioning your design will avoid ad-hoc changes towards the end of the design cycle.


 For more information about how to partition your design, refer to the *Design Planning with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

Plan for Incremental Compilation

The Quartus II incremental compilation feature allows you to partition a design, compile partitions separately, and reuse results for unchanged partitions. In addition, you can preserve performance for unchanged blocks and reduce the number of design iterations. The performance preservation that the incremental compilation feature provides can help you focus on timing closure.

If you change code or settings for one block in the design, you might have different compilation results than previous compilation results. The different compilation results can cause timing violations in blocks that do not have code or setting changes. However, with the incremental compilation methodology, you can preserve earlier results for a block that you do not want to change.

You can also use incremental compilation as a strategy to optimize a block that has difficulty meeting timing requirements. For this type of block, create a partition and turn on specific optimizations for the partition as a stand-alone design or within the design. After getting satisfactory results, you can preserve the results of that block for future compilations by choosing the appropriate netlist type with the incremental compilation feature.

 For more information about effectively using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Early Compilation of Blocks

Identify the major functional blocks of the design. With larger designs, following a flat design methodology is not practical. Creating a hierarchical design is beneficial, because it divides the design into manageable blocks and also makes it possible for multiple designers to work on a project. Additionally, you have the option of optimizing your blocks independently.

A common problem that prolongs the design cycle is waiting for code completion to compile the design. With this approach, issues are not detected until the end.

Compile your major blocks as soon as you can, even if your design is not complete. This allows you to identify coding styles that may not be appropriate for the chosen device. By doing this, you can also identify resource issues early in the design cycle.

You should minimize inter-block connections when creating design blocks. Register all inputs and outputs from each block to avoid critical timing paths crossing between module boundaries. If you use incremental compilation partitions, details of other partitions are not visible when the Quartus II software works on one of the partitions; therefore, optimization (or logic minimization) across partitions is not possible.

Compile each major block in your design as soon as you complete the HDL coding. Verify each block individually to ensure that the desired requirements can be achieved on a block-by-block basis with stand-alone simulations. At this stage, you can also verify the floorplan and partition connections.

You can create a dummy project for low-level blocks. For these internal blocks, declare all ports to be virtual pins before compilation, because the number of ports on the blocks may exceed the number of I/O pins available on the FPGA.

Run timing analysis on internal portions of the design so that you can see if there are bottlenecks within a block. You can also find issues like incompatible coding style use or potential timing problems. If you address intra-block issues early, you can concentrate on issues between blocks later in the design cycle.

You can evaluate special optimization strategies for individual design blocks and enable features such as physical synthesis (that are generally compilation time intensive) and benchmark performance improvements on sub-blocks by compiling the major blocks independently.

By benchmarking specific optimization settings individually, you can decide the appropriate optimization settings each block should have when you run a top-level compilation. This reduces the impact on the overall compilation time.

To run timing analysis on a block, create block-level constraints, based on the top-level timing requirements. Even if timing requirements are not completely specified at the block level, running a timing analysis at the block level can help to fine tune the top-level timing constraints after all the blocks are coded and integrated together.

With this approach, you can find design issues early in the design cycle as opposed to late in the final timing closure stages. This approach is not limited to issues found in low-level design projects with incremental compilation, but can benefit any of your designs blocks.

Resolving timing issues may require allocating additional device resources to individual blocks or change timing budgets for different blocks. You can use fast synthesis and early timing estimation to compile a design if most of the parts are coded. Mark incomplete partitions as empty if you use an incremental compilation flow and compile the rest of the design to get an early timing estimate, as well as detect problems in design integration. Early timing estimations can help you find and resolve design problems during the early design stages.

For more information about using early timing estimate and fast synthesis options, refer to the *Design Planning with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

Plan for Verification

Exercise the design with a simulation test suite at the RTL level so that the desired functionality is correctly coded in HDL, allowing you to focus on any potential timing problems. If you have functional and timing problems at the same time, it can be difficult to isolate issues and make required fixes, which increases debug time. Designs that target large FPGAs are complex in nature and ad-hoc verification will not suffice. Elaborate planning and execution of verification is necessary, along with planning for the design and implementation.

Functional issues (such as incorrect interpretation of specifications, or implementation) may mask potential timing problems in the design, resulting in complicated engineering change orders (ECOs) at a later stage.

When partitioning the design, verify individual partitions. You can reuse some of your partition-level test benches and test cases in the top-level test suite.

Use any of the supported third-party simulators to run functional verification on your design.



For more information about using third-party simulators for simulation, refer to the *Simulation* section in volume 3 of the *Quartus II Handbook*.

General Tips: Designing for Speed

The following are general tips for performance planning:

- Divide and conquer by creating appropriate hierarchical blocks
- Register all block inputs and outputs
- Try using different blocks for optimal performance
- Optimize major blocks with higher than required speed when running a block level compilation
- Any additional margin (about 10%-20%) that you can achieve at the block level can be useful after integration
- Consider the available resources in the target device (such as the RAM blocks, DSP blocks, PLLs, and transceiver locations) while coding for your design
- Pipeline the design for better performance



For a detailed checklist to improve performance, refer to the optimization section of the *Quartus II Design Checklist* on the Altera website.

Best Practices for Timing Closure

This section discusses the following topics:

- “Following Synchronous Design Practices” on page 8
- “Following Good Coding Styles” on page 9
- “Creating a Good Design Hierarchy” on page 12
- “Use Partitions and Incremental Compilation” on page 14

Following Synchronous Design Practices

Following synchronous design practices can help with constraining the design. Although asynchronous techniques might save time in the short run and seem easy to implement, asynchronous design techniques rely on propagation delays and clock skews that do not scale well between different device families or architectures. Asynchronous circuits are prone to problems, such as glitches and race conditions, which render the resulting implementation unreliable, making it difficult to properly constrain your design. In the absence of appropriate constraints, synthesis or place-and-route tools may not perform the best optimizations, resulting in inaccurate timing analysis results.

These factors outweigh the advantages of quick design fixes with asynchronous techniques.

- For more information about using synchronous design practices for FPGA designs, refer to the “Using Synchronous Design Practices” and “Design Guidelines” sections of the *Design Recommendations for Altera Devices and the Quartus II Design Assistant* chapter in volume 1 of the *Quartus II Handbook*.

Following Good Coding Styles

Your coding style can have a significant impact on how the design is implemented in an FPGA, because the synthesis tool you use can optimize and interpret the design differently than you intended. Therefore, you must decide how to modify the design to assist the optimizations done by the synthesis tool.

FPGA devices are register rich, so pipelining your design can help you meet required performance, while not adversely affecting resource use. Adding adequate pipeline registers can help you avoid a large amount of combinational logic between registers.

In the following example, a counter increments with the following different values based on two inputs to the `select[1:0]` block:

- When select bits are '01', the counter increments by 1
- When select bits are '10', the counter increments by 2
- When select bits are '11', the counter increments by 3

Example 1 shows one way you can code the block.

Example 1.

```
// This example shows how latches could be unintentionally inferred
module latch_example (clock, reset, select, count_out);

input clock;
input reset;
input [1:0] select;
output [7:0] count_out;

reg [7:0] count;
reg [7:0] next_count;

//determine next value of counter based on select inputs
always@(select or count)begin
    case (select)
        2'b01 : next_count <= count+1;
        2'b10 : next_count <= count+2;
        2'b11 : next_count <= count+3;
    endcase
end

// register next value of the counter
always@(posedge clock or negedge reset) begin
if (!reset) begin
    count <=0;
end else begin
    count <= next_count;
end

end
end

assign count_out = count;
endmodule
```

[Example 2](#) shows the latch inference warnings when you compile this project in the Quartus II software.

Example 2.

```
Warning (10240): Verilog HDL Always Construct warning at latch_example.v(12): inferring latch(es)
for variable "next_count", which holds its previous value in one or more paths through the always
construct
Info (10041): Inferred latch for "next_count[0]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[1]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[2]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[3]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[4]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[5]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[6]" at latch_example.v(12)
Info (10041): Inferred latch for "next_count[7]" at latch_example.v(12)
Warning: Latch next_count[0] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[1] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[2] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[3] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[4] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[5] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[6] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Warning: Latch next_count[7] has unsafe behavior
  Warning: Ports D and ENA on the latch are fed by the same signal select[0]
Info: Implemented 60 device resources after synthesis - the final resource count might be different
```

The case statement in [Example 1](#) does not include a branch for 2'b00. The incomplete case statement caused the latches to be inferred. Latches can cause race conditions, are hard to constrain, and do not offer an advantage over using registers. Furthermore, analyzing results of designs with latches is difficult.



For an archived project with the code in [Example 1](#), refer to the [an584_latch_example.qar](#) located in the [an584_design_examples.zip](#) file.

To avoid problems caused by latches, add a default statement or supply the missing case to complete the case statement, as shown in [Example 3](#).

Example 3.

```
module latch_example_modified
(clock, reset, select, count_out);

input clock;
input reset;
input [1:0] select;
output [7:0] count_out;

reg [7:0] count;
reg [7:0] next_count;

always@(select or count)begin
  case (select)
    //when select = 2'b00, do nothing (keep previous value)
    2'b00 : next_count <= count ;
    2'b01 : next_count <= count+1;
    2'b10 : next_count <= count+2;
    2'b11 : next_count <= count+3;
  endcase
end

always@(posedge clock or negedge reset) begin
if (!reset) begin
  count <=0;
end else begin
  count <= next_count;
end
end

assign count_out = count;
endmodule
```

When you compile this module, the warning messages related to latch inference will not be in the compilation report.



For a project archive with this modification, refer to the **an584_latch_example_modified.qar** located in the [an584_design_examples.zip](#) file.

Follow coding guidelines described in the [Quartus II Handbook](#) to avoid latches, combinational loops, and other styles that are not suitable for FPGA implementation. Furthermore, following suggested coding styles can help you infer memories, multiplier blocks, and DSP blocks that are available in Altera FPGAs.

If you use unsupported operations (such as asynchronously clearing of RAM content), the Quartus II software might not map your logic into the RAM blocks and instead, implement the logic with logic cells.



For inferring specialty design blocks, such as DSP blocks, specific multiplier blocks, memories, or other Altera megafunctions from your code, and styles of coding suitable for Altera FPGA implementation, follow the recommendations in the [Recommended HDL Coding Styles](#) chapter in volume 1 of the [Quartus II Handbook](#).

You might under utilize some of the available resources if you are not aware of your hardware resources. In general, do not use constructs that lack the equivalent hardware implementation available in the device. For example, your code might not be mapped to any of the available RAM blocks in the device if you infer RAM locations and use synchronous resets on RAM locations to clear the contents or to initialize the values. This is because the RAM locations in Altera devices do not have asynchronous or synchronous resets available for RAM cells. Instead, logic that models a memory with a reset is implemented in logic cells. Review your specifications to verify if having a known initial value in the RAMs is necessary for proper design function (most often it is not required). If RAM cells have to be initialized to certain known values (such as all 0's), consider performing write cycles to the RAM immediately after power up.

You must consider hardware mapping when you code. Changes in the code can affect the number of logic levels and the corresponding timing. Although the software optimizes your design, unnecessary optimizations can affect software performance. Editing the HDL can improve the quality of results.

Creating a Good Design Hierarchy

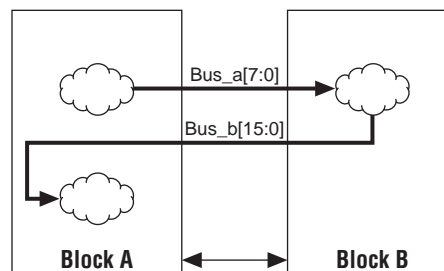
If you create a design hierarchy, make it reflect the functional interfaces in your design so that functionally different blocks might be designed by different designers. The interfaces between the blocks should be properly constrained. If timing failures are within specific blocks in the hierarchy, you could set your constraints so that the Quartus II software works harder to meet those constraints. After meeting the requirements for the specific block, you can lock down the block and work on the remaining blocks in the hierarchy.

When you create a hierarchy, consider reusing blocks by making parameterized modules. Try to reduce the number of inter-block connections. Register signals that pass across those module boundaries which you plan to compile as incremental compilation partitions.

Figure 1 shows an example of an output from some combinational logic, which is an 8-bit wide signal (`Bus_a[7:0]`), going from Block A to Block B. In Block B, the output goes through another combinational logic. A 16-bit output bus goes from Block B to Block A, where in it goes through another combinational logic.

If the two blocks in this example are not in the same incremental compilation partition, optimization on this path may not be possible.

Figure 1. Example of Poor Combinational Logic Placement



Assume that Block A and Block B are assigned to different incremental compilation partitions. To avoid the problem of critical paths going across a partition boundary, consider moving the combinational block in Block B to Block A, as shown in [Figure 2](#). If that is not possible, register each block output (possibly inputs too), as shown in [Figure 3](#). This arrangement splits critical paths and confines them into one block, rather than spreading them across multiple partitions.

Figure 2. Example of Combinational Logic Placed in Block A

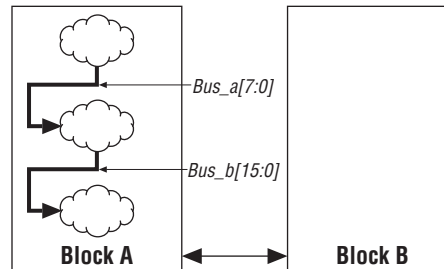
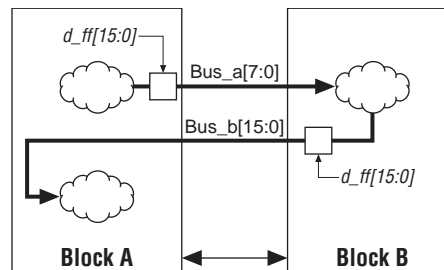


Figure 3. Example of Registering Outputs



In general, follow these guidelines:

- Create blocks that can be verified as stand-alone in HDL simulation
- Register all inputs and outputs of incremental compilation partitions



In cases where you cannot follow this guideline completely, do the best you can for your case

- Consider reusing a block in a design (or within multiple designs) while coding
- Follow uniform coding and signal naming conventions


Creating a good design hierarchy makes team-based designs efficient by allowing individuals or groups to separately design, verify, and implement different functional blocks in the design. When you partition your design, follow the incremental compilation methodology recommended in the *Quartus II Handbook*. If you compile one partition, the Quartus II software considers the rest of the design a black box. Therefore, it is essential to follow these guidelines so that there is appropriate utilization of resources in the device.

Use Partitions and Incremental Compilation

When you make changes to your design or to the optimization settings, the Quartus II software detects changes in the design. These changes force a recompile of the design, even if the changes were limited to a few modules.

Changes in the design or settings cause a different set of initial conditions for the Fitter. The initial placement the Fitter finds for your design might be entirely different than previous placements, which may result in many differences between compilations.

You can preserve previous compilation results and reduce compilation times with multiple compilations by using partitions and incremental compilation. Based on the preservation level, the Quartus II Fitter skips some compilation steps for selected partitions.

 For more information about creating effective partitions and partitioning examples, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Constraining and Compiling Your Design

This section shows the steps for constraining and compiling your design.

Setting Location Constraints

You can create location assignments to place logic and I/O blocks at specific locations in the chip with the Quartus II software. However, the Quartus II software logic placement is generally better than user-assigned placement.

For example, it may not help if you assign a block containing a critical path to a LogicLock™ region and constrain the path by squeezing the LogicLock region. The Quartus II Fitter is aware of the critical path and attempts to optimize the path by considering many physical constraints to find the best placement. By restricting placement, you might be deteriorating performance to some extent, so use this strategy judiciously. There are times when you could use location constraints effectively to aid in timing closure, directly or indirectly.

With LogicLock regions or location constraints, you can constrain logic blocks to use specific areas in the FPGA. You can benefit from using LogicLock regions to create a floorplan for your design with the incremental compilation flow.

In a team-based design, each major block in the design could be designed by a different engineer. In such cases, pre-assign each block to have its assigned area in the device. You can reserve areas in the device based on interfaces and resources such as transceivers or RAM blocks. You can reserve the region on a previous Fitter-assigned area as well.

 Having excessive location constraints may negatively affect performance.

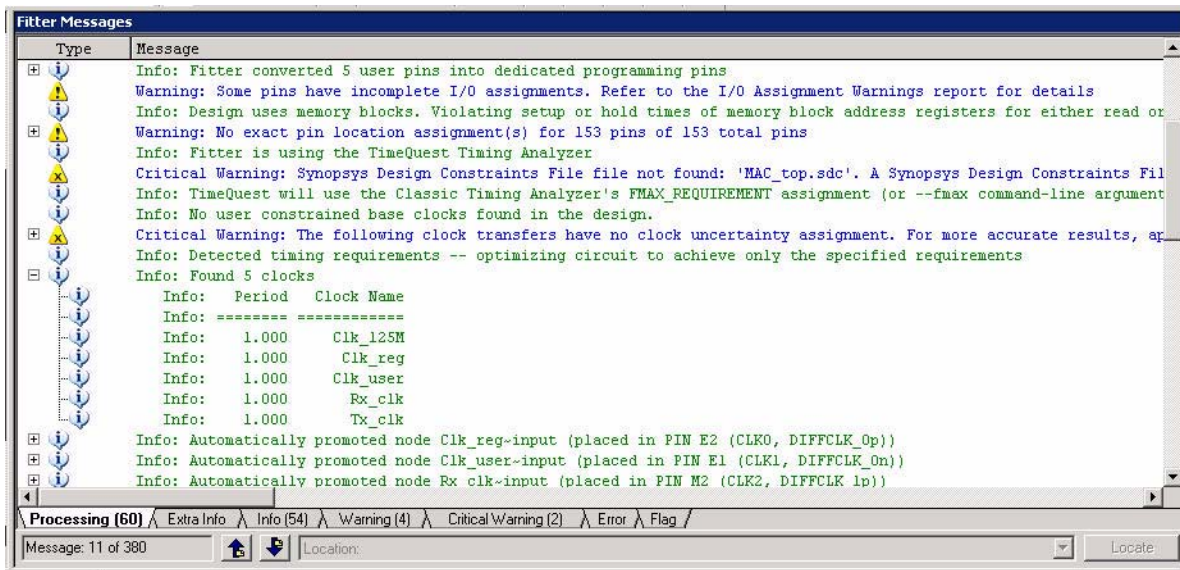
Setting Proper Timing Constraints

Quartus II software optimizations are dependent on design constraints. The Quartus II software has default settings that have good general results over a range of designs. You can change settings based on your requirements.

The **Mac_top_noconstraint.qar** design example, located in the [an584_design_examples.zip](#) file, shows the importance of setting proper timing constraints. Download and compile the design in the Quartus II software. The design is an OpenCore Ethernet core that is part of a bigger design. The OpenCore is assigned to a LogicLock region within a Cyclone II device.

After compilation, the timing analysis results show several failing clock domains. [Figure 4](#) shows the Fitter messages section of the Compilation Report.

Figure 4. Fitter Section of the Compilation Report Shows Critical Warning



Critical warning messages indicate that the design does not have a Synopsys design constraints file (**.sdc**), which can be used by the Fitter and for static timing analysis by the TimeQuest Timing Analyzer. The Quartus II software assumes a default frequency of 1 GHz for all clocks. Because timing analysis results are based on this incorrect assumption, the results are invalid.

To fix this critical warning about missing constraints, add an **.sdc** containing appropriate constraints. Open the **Mac_top_with_sdc.qar** design file located in the [an584_design_examples.zip](#) file. This design example contains a valid **.sdc** for constraining timing. Compile the design in the Quartus II software.

After compilation, check timing analysis results in the Compilation Report. To analyze the problematic paths, run the TimeQuest Timing Analyzer and run the Report Top Failing Paths command. Figure 5 shows a list of failing paths, many of which are between clock domains.

Figure 5. Timing Violations Across Clock Domains

Slack	From Node	To Node	Launch Clock	Latch Clock
-1.409	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[3]	Clk_reg	Clk_125M
-1.385	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[2]	Clk_reg	Clk_125M
-1.385	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[1]	Clk_reg	Clk_125M
-1.381	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[5]	Clk_reg	Clk_125M
-1.357	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[6]	Clk_reg	Clk_125M
-1.356	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[4]	Clk_reg	Clk_125M
-1.332	Reg_int:U_Reg_int:RegCPUData:U_0_007:RegOut[0]	MAC_tx:U_MAC_tx:MAC_tx_ctrl:U_MAC_tx_ctrl:TxD[4]	Clk_reg	Clk_125M
-1.306	Reg_int:U_Reg_int:RegCPUData:U_0_007:RegOut[0]	MAC_tx:U_MAC_tx:MAC_tx_ctrl:U_MAC_tx_ctrl:TxD[4]	Clk_reg	Clk_125M
-1.259	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[0]	Clk_reg	Clk_125M
-1.100	Reg_int:U_Reg_int:RegCPUData:U_0_034:RegOut[2]	Phy_int:U_Phys_int:Txd[7]	Clk_reg	Clk_125M
-1.079	Reg_int:U_Reg_int:RegCPUData:U_0_007:RegOut[0]	MAC_tx:U_MAC_tx:MAC_tx_ctrl:U_MAC_tx_ctrl:TxD[1]	Clk_reg	Clk_125M
-1.007	Reg_int:U_Reg_int:RegCPUData:U_0_007:RegOut[0]	MAC_tx:U_MAC_tx:MAC_tx_ctrl:U_MAC_tx_ctrl:TxD[1]	Clk_reg	Clk_125M

By default, the TimeQuest Timing Analyzer assumes clocks in a design are related to each other and analyzes all paths. It is possible that paths between some clock domains are false and require no analysis. If the clocks in your design are asynchronous to each other, you should add the `set_clock_groups` command in the `.sdc` to specify different clock groups.

Open the `Mac_top_modified_sdc.qar` design example located in the [an584_design_examples.zip](#) file. Compile it with the Quartus II software. Run the TimeQuest Timing Analyzer after compilation is completed, and run the Report Top Failing Paths command. There should be fewer timing violations (and on a single clock domain) remaining to fix.

Supplying the appropriate constraints helps you separate real violations from false violations. Make changes in your HDL or assignments to solve issues when you identify real violations.

In this example, we assumed that all clock domains were asynchronous to each other. You should investigate the relationship between various clocks in your design and categorize them appropriately.

When you have multiple interacting clock domains or when your design has high performance circuits, such as external memory interfaces or clock multiplexing, ensure that you have the right inter-clock constraints so that the Quartus II software can work on the most critical paths.

For examples that show different scenarios for constraining your design, refer to the [Quartus II TimeQuest Timing Analyzer Cookbook](#) and [The Quartus II TimeQuest Timing Analyzer](#) chapter in volume 3 of the [Quartus II Handbook](#).

Using Optimal Settings for Your Design

In addition to using the correct design methodology and proper timing constraints, use the right settings for the design. The Quartus II software offers many options to meet different design requirements. If you do not use the appropriate setting, design performance may suffer.

The Quartus II software has many settings to meet different design requirements. For example, if your design goes into a mobile device, you might optimize your design for the lowest power consumption. Or if your design targets a low-cost system, you might pick a device with a specific architecture and optimize your design for that goal. Or you may want to reduce the size of the design, and optimize it for area.

The Quartus II software default settings offer a balance of performance, area, power, optimization, and compilation time. To achieve a specific goal, such as performance, low power, area, or compilation time, you may have to choose a setting different from the default setting. By trading off one feature for another, you can meet your preferred design requirements.



There is no definitive set of settings that gets the best performance for all designs. Each design is unique and a settings set that derives the best performance for one design may not be able to improve performance in another design.

Be aware that using the best effort options might help in some cases, but some of the options might not make a positive impact on your primary goal. You might be making optimizations that increase the compilation time without a corresponding improvement in performance. You should only use settings that help meet your goals.

The settings you use for a project are specific for that project. When you make substantial changes to a design, it is possible that some of those settings have to be changed. Whenever you make large changes in your design, use the Design Space Explorer (DSE) to find the best settings.

Common Timing Issues Encountered

When you see failures in your timing report, verify if the failing requirements are correct. By default, the TimeQuest Timing Analyzer handles all clock domains as related and this may cause some invalid requirements. To fix this, group clocks in your design appropriately. Also, check for clock skew on the critical paths in the timing report. You may be able to fix clock skew issues by using clock enable signals or the ALTCLKCTRL block. You should also analyze the datapath to see how you can reduce the critical path delays. This section contains examples of commonly encountered timing issues and troubleshooting practices.



For details about how to group clocks, refer to *The Quartus II TimeQuest Timing Analyzer* chapter in volume 3 of the *Quartus II Handbook* or the *Quartus II TimeQuest Timing Analyzer Cookbook*.

Too Many Levels of Logic

The number of logic levels between registers may be difficult to see in the code itself. The number of levels of combinational logic can increase the delay on a path and cause it to become critical.

Additions of conditional statements generally get translated as additional levels of logic. Verify conditions before adding more conditional statements within nested conditional statements. Ensure that the modifications you make apply the conditions only on the branch where it is necessary.

Example 4 shows a section of Verilog HDL code. Assume `counter1` is an 8-bit counter and `counter2` is a 12-bit counter. `TC1` and `TC2` represent the terminal counts for these counters.

Example 4.

```

if (reset == 0) begin
    ...
    ...

end else begin

    ...
    ...

    if (incr ==1) begin
        if (counter1 == value1) begin
            counter1 <= 0;
            TC1 <= 1'b1;

            if (counter2 == value2) begin
                counter2 <= 0;
                TC2 <= 1'b1 ;

            end else begin
                counter2 <= counter2 +1 ;
                TC2 <= 1'b0;
            end

        end else begin
            counter1 <= counter1 +1 ;
            TC1 <= 1'b0 ;
        end
    end
end
...
....
end

```

The updates for `counter2` are based on the results of 8-bit and 12-bit comparators. Depending on the device architecture, this combinational logic is implemented in several levels. The manner in which you map the logic can affect the number of constituent delays. **Figure 6** shows the critical path through combinational logic between two register stages.

Figure 6. Critical Path

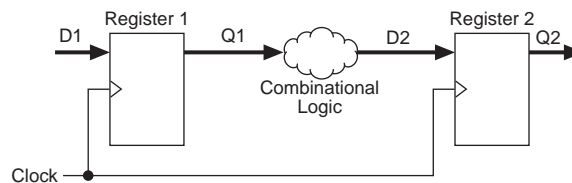
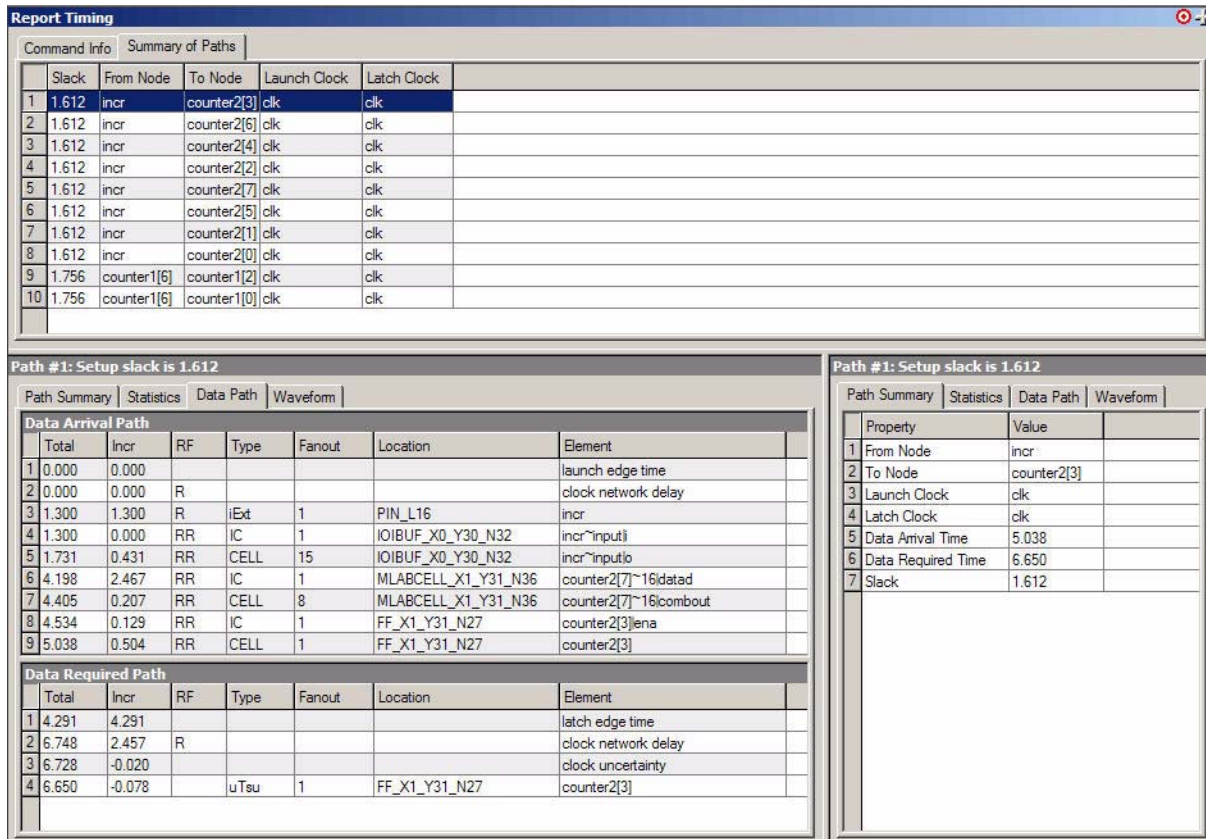


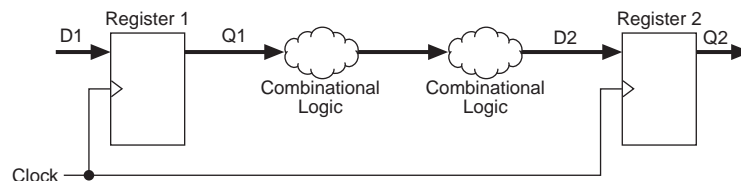
Figure 7 shows the TimeQuest Timing Analyzer report panel for one possible implementation with no timing failures. There are seven incremental delays (including cell delays and interconnect delays) contributing to the critical path delay.

Figure 7. TimeQuest Timing Analyzer Report



A critical path may fail required timing constraints if this segment is nested inside another nested conditional branch with several levels of logic. Figure 8 shows the critical register-to-register path. The additional logic cloud causes the timing failure.

Figure 8. Segment Nested Inside Another Nested Conditional Branch with Several Levels of Logic



Pay attention to the possible implications when you add conditional statements in your code. It is often possible to check some conditions in parallel with or before results are available from other logical operations. By doing so, you can reduce the delay on the critical path.

In some cases, it may not be possible to modify logic by parallel operations, so consider the possibility of using pipeline registers to split the logical operations that occur in one clock cycle. You must account for the effect of this added latency in other parts of the design if you take this approach.

If you have defined constants within your design, be aware that the values are propagated as desired in the design. Signals tied to a constant '1' or '0' can simplify logic and reduce the number of logic stages in a given path.

To analyze the number of logic stages in the critical path, you can use the Technology Map Viewer and the RTL Viewer in the Quartus II software, in addition to the path summary panel in the TimeQuest Timing Analyzer timing report (shown in [Figure 7](#)).



For more information about the RTL Viewer and Technology Map Viewer, refer to the *Analyzing Designs with Quartus II Netlist Viewers* chapter in volume 1 of the *Quartus II Handbook*.

Perform the following general guidelines to fix failing critical register-to-register paths:

- Give priority to improving the code instead of modifying the Quartus II software settings
- Analyze whether critical paths can be re-coded
- Check if logic can be pushed across register boundaries
- Check if part of the logic can be done in parallel, or in a different data cycle (a cycle before or later)
- When you modify code, be aware of how it is implemented in hardware
- If you are working on a block within a larger design, target a higher than required timing performance

Missing Timing Constraints

The Quartus II software tries to optimize your design for the provided constraints. Improper design constraints can contribute to timing failure. The TimeQuest Timing Analyzer does not analyze unconstrained paths. Review messages from the TimeQuest Timing Analyzer to ensure you have constrained all required timing paths. In the TimeQuest Timing Analyzer, you can also use the `report_unconstrained_paths` command to check paths that do not have timing constraints.

When some of the required constraints are missed for the operation of the design, you may get a timing report without any violations. But if the missing constraints are critical, then your design implementation may not perform as intended. Therefore, you must completely constrain your design.

In addition to missing timing constraints, another common issue is under-specification or over-specification of timing constraints. You must analyze your timing analysis report for false or multi-cycle paths. The Quartus II software tries to optimize all paths as valid, single-cycle paths when you do not identify them as false or multi-cycle paths. This might cause valid paths to fail timing requirements (depending on which paths the Fitter optimized first). To avoid this scenario, identify false and multi-cycle paths in the constraints file.

When you specify aggressive timing constraints on one domain, the Quartus II software attempts to optimize that domain earlier than other clock domains. You can use this to selectively apply more optimization effort to one domain instead of others. The benefit you derive from this is design dependent. Also, you must manually analyze the failing paths to determine if they have met your requirements even if they failed the over-constrained requirement. In general, use real requirements for constraining your design.

The TimeQuest Timing Analyzer assumes all clocks in a design are related, and checks for all possible paths in the design. Apply false path constraints between clock domains that do not have valid paths.

You can over-constrain your design if you want to improve timing performance on select domains, particularly in block-level compilations. If more stringent constraints are met at the block level, it can be easier to meet the timing after blocks are integrated together. This compensates for delays that cannot be accurately predicted at block-level implementation.

Conflicting Constraints

Designs with more than one clock domain can have conflicting timing constraints on paths between different clock domains. For example, failing recovery paths can cause logic movement towards the reset register. This can worsen setup paths. Similarly, failing hold paths cause the Fitter to add extra delays, which results in congestion that makes fitting difficult.

While you cannot avoid working on one domain at a time, ensure that you properly perform static timing analysis on the whole design. You can use the TimeQuest Timing Analyzer command, Report All Summaries, which reports setup, hold, recovery, and removal analyses for all constrained paths in the design.

Handling High Fan-Out Registers

FPGA signals are highly buffered. Critical timing paths occur because of where the registers are placed, regardless of fan-out.

Use the Chip Planner to view the location of fan-out registers and the locations they are driving. The main reason for excess delay on timing paths is if registers are spread far apart. On most occasions, it is difficult to place the high fan-out node better than the Quartus II software, as observed in the Chip Planner.




For more information about the Chip Planner, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*.

If specific registers suffer from this problem, consider duplicating the logic and place the duplicated register in a more suitable location. When a duplicate register is available, the Quartus II software may be able to find a suitable location during fitting that improves the resulting timing performance.

When you duplicate logic in your code, ensure that the Quartus II software does not optimize the duplicate registers. There are several ways of doing this:

- Put a synthesis attribute in the code to preserve the register, or turn off **Remove Duplicate Registers** in the register or hierarchy

- Use the Assignment Editor to manually select the register you want to duplicate, and turn on duplication
- Use the Assignment Editor to provide a Maximum Fan-out assignment. This creates as many duplicate registers as required, depending on the fan-out
- Turn on physical synthesis and enable register duplication

 For more information about using synthesis attributes, refer to the *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*. For more information about using the Assignment Editor, refer to the *Assignment Editor* chapter in volume 2 of the *Quartus II Handbook*. For more information about using physical synthesis options, refer to the *Netlist Optimization and Physical Synthesis* chapter in volume 2 of the *Quartus II Handbook*.

Metastability Issues

It is difficult to design systems that operate on a single clock. Even when clocks run at the same frequency, there could be a phase difference. Therefore, designs with multiple clock domains are common.

Signals with a source in one clock domain may be registered in another clock domain, so the data input may not meet the setup or hold time requirements of the destination register. This may cause the output of the latching register to go to an intermediate state between a logical 1 and a logical 0. Although this intermediate value eventually resolves to a '1' or '0,' the issue is the indeterminate time the value takes to resolve itself.

Signals that move cross one clock domain to another must meet the timing requirements in both domains. To reduce the possibility of inter-clock-domain signals from going metastable, you must synchronize all signals between asynchronous clock domains with multiple stages of synchronizing registers. A minimum of two stages of synchronizing registers is recommended.

The Quartus II software has features that analyze designs for metastability and offers recommendations to reduce metastability. Mean Time Between Failures (MTBF) can be computed on circuit parameters. You should increase the MTBF as much as possible. The Quartus II software provides a metastability advisor to improve the metastability performance of your design.

Isolating failures caused by metastability can be problematic, because they can appear as sporadic device failures, which are hard to debug. Increasing the MTBF might reduce the occurrence of such failures.

Metastability problems in your design can appear as incorrectly operating state machines. Symptoms include skipped states or state machines that do not recover from a stage or lock-up. State machines might also miss triggering events that cause state transitions. Such problems might occur when control signals to a state machine, coming from other clock domains, are not synchronized.

Example 5 shows a portion of Verilog HDL code for a state machine. If the signals used in some of the conditional branches are coming from other clock domains and used directly, you might miss some edges that will cause uncertain state transitions.

Example 5.

```
//process for registering next state
process (clock, reset)
always@(posedge clock or negedge reset) begin
if (~reset) begin
present_state <= reset_state ;
end else if begin
present_state <= next_state;
end

//process for next state
always @( <sensitivity list>) begin
if ( condition == condition 1" ) begin

.....
.....
.....
end
else if (condition == condition2) begin

.....
.....
.....
end
end process
```

By synchronizing all asynchronous control signals twice, you can make sure these signals remain stable for an integral number of clock cycles, and trigger state transitions appropriately.



For more information about metastability, refer to the *Managing Metastability with the Quartus II Software* chapter in volume 1 of the *Quartus II Handbook*.

Reset Related Issues

Designs can have synchronous or asynchronous resets. Most resets coming into devices are asynchronous. You can convert an external asynchronous reset to a synchronous reset by feeding it through a synchronizer circuit, and use it to reset the rest of the design. This clock creates a clean reset signal that is at least one cycle wide and synchronous to the domain in which it is used.

If you use a synchronous reset, it becomes part of the data path and affects the arrival times in the same manner as other signals in the data path. You should include the reset signal in the timing analysis along with the other signals in the data path. Using a synchronous reset takes additional routing resources, such as an additional data signal.

If you used an asynchronous reset, you can globally reset all registers. This dedicated resource helps you avoid the routing congestion that is caused by a reset signal. However, a reset that is completely asynchronous can cause metastability issues because the time when the asynchronous reset is removed is asynchronous to the clock edge. If the asynchronous reset signal is removed from its asserted state in the metastability zone, some registers could fail to reset. To avoid this problem, use synchronized asynchronous reset signals. A reset signal can reset registers asynchronously, but the reset signal is removed synchronous to a clock, reducing the possibility of registers going metastable.

You can avoid unrealistic timing requirements by adding a double synchronizer to the external asynchronous reset for each clock domain and then using the output from the synchronizer as the asynchronous reset to reset all registers in their respective clock domains.

Example 6 shows one way to implement a reset synchronizer. The reset structure resets asynchronously, but returns to the non-reset state synchronously. This works for designs where you want the device to come out of reset at the same cycle.

Example 6.

```
module safe_reset_sync
    (external_reset, clock, internal_reset) ;

    input external_reset;
    input clock;
    output internal_reset;

    reg data1, data2, q1, q2;

    always@(posedge clock or negedge external_reset) begin
        if (external_reset == 1'b0) begin
            q1 <= 0;
            q2 <= 0;
        end else begin
            q1 <= 1'b1;
            q2 <= q1 ;
        end
    end

endmodule
```

Recovery and Removal Problems

The TimeQuest Timing Analyzer performs recovery and removal analysis in addition to setup and hold analysis. Recovery time is analogous to setup time and removal time, and analogous to hold time. The difference is these timing parameters are associated with asynchronous signals (such as reset) with respect to the clock.

Recovery and removal analysis helps you ensure that your synchronous logic behaves correctly when asynchronous control signals are applied.

A problem with the reset signal, which spans across the device, is that it might not arrive at the same time relative to the clock edge for all the device registers. When the reset signal is deasserted, all the registers are supposed to come out of reset. However, if the reset signal does not meet the recovery time for some registers in the design, those registers might not come out of reset until after the next clock edge. If the

registers do not all come out of reset in the same clock cycle and if there are state machines with important transitions after this clock cycle, these state machines may not behave as expected and cause a design failure. Similarly, a removal error might happen when the reset is removed too early, relative to the clock, and some registers come out of reset one cycle earlier.

If you encounter recovery or removal errors, follow these troubleshooting tips:

- **Check if the source of the reset is placed close to a global driver**—Normally, the Quartus II Fitter places the source for high fan-out nets close to a global driver. If this does not happen, all the paths have some additional delay that might cause recovery or removal errors.
- **Check if the signal is placed on a global resource**—If the asynchronous control signal is not using a global signal, assign it to one of the global nets with the Assignment Editor. If it is already on a global net, but still failing recovery/removal timing, check if the registers being reset are spread out across the device or if they are concentrated in certain parts of the device. In the latter case, you can try to improve the timing with local routing.
- **Duplicate reset logic**—You can duplicate the reset logic and assign it to different local routing in different parts of the device.
- **Check if the requirements are realistic**—For designs with multiple clock domains, having reset structures spanning all the clock domains may impose very tough recovery and removal requirements. Altera recommends that you use a separate reset structure for each of the synchronous clock domains.
- **Turn on automatic asynchronous signal pipelining**—You can use this physical synthesis option to insert pipeline stages on asynchronous control signals during fitting to increase circuit performance. This option is useful for asynchronous signals that are failing recovery and removal timing because they feed registers that use a high-speed clock. To set this option, on the Assignments menu, click **Settings**. In the **Category** list, click the “+” icon to expand **Compilation Process Settings** and select **Physical Synthesis Optimizations**. Turn on **Perform automatic asynchronous signal pipelining** and click **OK**.
- **Decide if these timing errors are relevant**—If your design does not require all registers to come out of reset in the same clock cycle, you can set all such paths as false paths by adding the `set_false_path -from reset -to <all-registers>` command.

Missing Timing Requirements By A Small Margin

The DSE can help you select the optimal design settings. Depending on your criteria for optimization, the DSE runs multiple compilations and flags the best results. Using the DSE to run a seed sweep also helps you compensate for the seed effect, as described in [“How Does the Quartus II Fitter Work?” on page 3](#).

If you make changes in the code or settings, Altera recommends running a seed sweep to ensure that the improvements in performance are caused by the changes in the code or Quartus II software settings and not the seed effect.

After changing the code to improve performance, run a DSE seed sweep to verify if a specific seed value meets all your requirements. You can have up to a 10% spread of performance with certain seed values. This approach can help when the default seed value used for the compilation is at the lower end of the spread or even an outlier from the nominal range of the result spread.

A seed sweep with more than 10 seed values is generally not required.

Before running a DSE seed sweep, refer to the recommendations from the Quartus II Timing Optimization Advisor. If the recommendations are applicable and offer an improved performance result, use the updated settings for the DSE seed sweep.



You can run a DSE seed sweep or a complete DSE exploration for any design, regardless of whether the results are close to the desired results or not. In general, DSE helps you find settings that give the best results based on your criterion for optimization.



For more information about using the DSE, refer to the *Design Space Explorer* chapter in volume 2 of the *Quartus II Handbook*.

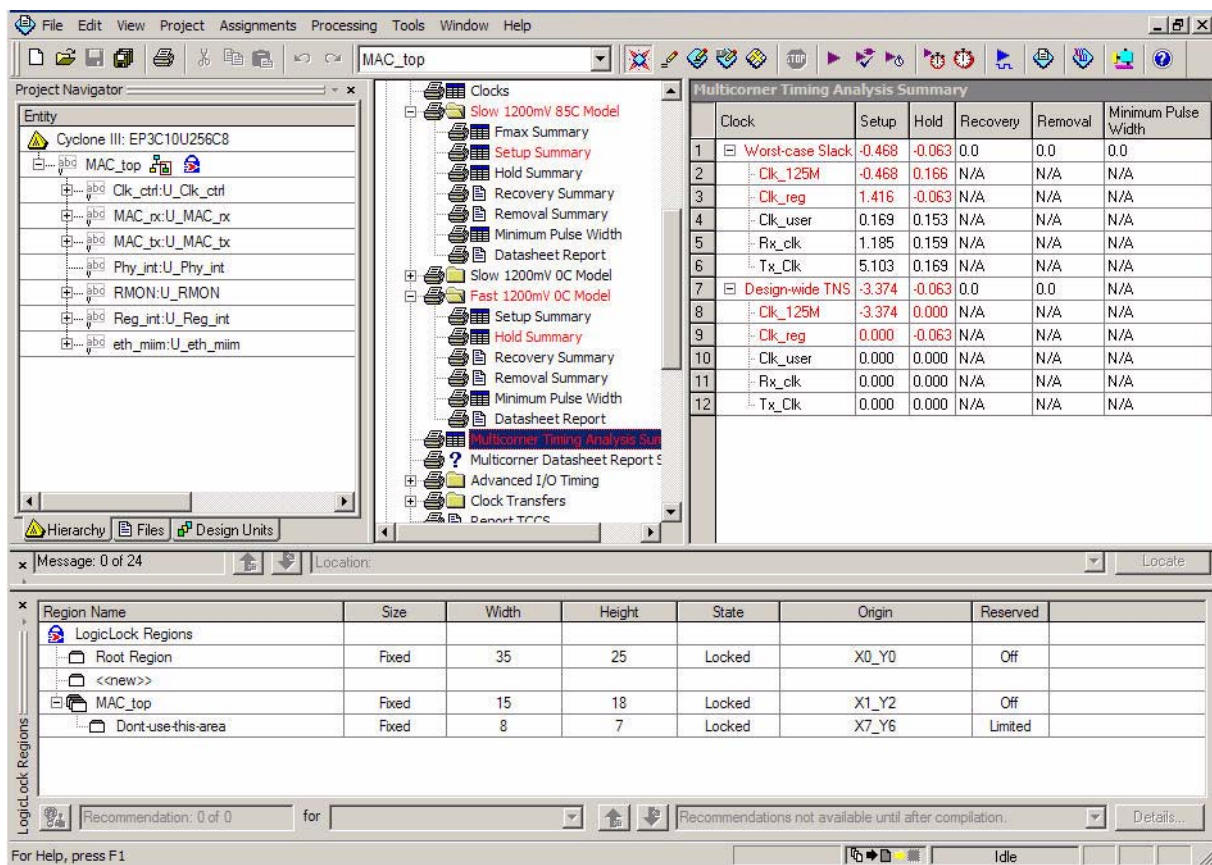
Overly Restrictive Location Constraints

Excessive location constraints can become an issue in meeting timing requirements. For example, if your design has many LogicLock areas and the design does not meet timing requirements, try removing one or more LogicLock assignments and run a compilation to see the impact.

The following example shows how location restrictions might affect your design. Open the **Mac_top_restricted.qar** design example, located in the [an584_design_examples.zip](#) file, and compile the project.

The **Mac_top_restricted.qar** design example, located in the [an584_design_examples.zip](#) file, is an OpenCore Ethernet design that is part of a larger design. The core is assigned to a LogicLock area. However, some of the area is assigned to a reserved LogicLock area, restricting the area available for placement. **Figure 9** shows the LogicLock region assignment and multi-corner timing analysis summary for this design example. There are timing violations in the fast and slow corners, and on two clock domains.

Figure 9. LogicLock Region Assignment and Multi-Corner Timing Analysis



Open the **Mac_top_unrestricted.qar** design example, located in the [an584_design_examples.zip](#) file, and compile it to see the same design without restrictions.

Figure 10 shows the Compilation Report. Even without restrictions, there are timing violations, but the errors are only on the slow corner and on one clock domain.

Figure 10. Compilation Report

Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
Worst-case Slack	-0.650	0.0	0.0	0.0	0.0
Clk_125M	-0.650	0.158	N/A	N/A	N/A
Clk_reg	1.119	0.057	N/A	N/A	N/A
Clk_user	0.294	0.159	N/A	N/A	N/A
Rx_clk	1.143	0.159	N/A	N/A	N/A
Tx_Clk	5.163	0.173	N/A	N/A	N/A
Design-wide TNS	-1.054	0.0	0.0	0.0	N/A
Clk_125M	-1.054	0.000	N/A	N/A	N/A
Clk_reg	0.000	0.000	N/A	N/A	N/A
Clk_user	0.000	0.000	N/A	N/A	N/A
Rx_clk	0.000	0.000	N/A	N/A	N/A
Tx_Clk	0.000	0.000	N/A	N/A	N/A

This example is not a generalization for or against using LogicLock areas, but it shows why you must review the location constraints in your design. You may still be required to constrain some design blocks with LogicLock regions, because you want to preserve placement and performance with incremental compilation, but consider how any restriction might affect your project.

For more details about defining LogicLock regions, refer to the *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*. For more information about using LogicLock regions for the floorplan when using incremental compilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

Long Compilation Times

If timing performance is your most important criterion, you may have long compilations. Abnormally long compilations may indicate resource issues, timing constraints that are too hard to meet, or logic loops that cannot be broken easily. Check the compilation warning messages to find the part of the design that is contributing most to long compilations.

Fitter messages can indicate resource congestion. Resource congestion can usually be found by looking at the utilization numbers. Utilization of 95% or more can be challenging to fit. In such cases, revisit your code to recode blocks that may reduce logic use and remove redundancies.

Tips for Tackling Timing Problems

When you have timing problems, isolate the problematic paths or blocks. You can strip the problematic blocks into a stand-alone project to find the best performance attainable to see if you can meet the required timing in the design.

This approach gives you an indication of the potential performance of a block, and it makes debugging or optimization easier. If you compile major blocks or partitions individually and test them for performance as suggested earlier in this application note, be aware of the potential for added design time in each block, so that you can reduce having to do this for many of failing blocks at the end of the design cycle.

One reason for unexplained failures on a board could be timing requirements that were not specified while running static timing analysis. If these were caused by setup failures, you may be able to run the design at a rate slower than the intended frequency and verify your design. However, if failures exist because of hold violations that were not resolved, the design cannot function reliably at any frequency. Also, depending on the timing requirements that are not met, your design may operate correctly at certain temperatures or voltages, but fail at a different temperature or voltage range.

You can use Quartus II resources to help you through the process.

Edit HDL Code for Performance

Editing the HDL might be the best way to improve the quality of results. If the Quartus II software improved timing margins on some register paths by register retiming, you can try to replicate those changes in the design itself. You can also make changes by duplicating registers, which is done as a part of physical synthesis optimizations to improve the timing performance. Use schematic viewers in the Quartus II software to analyze the optimization changes implemented by the Quartus II software and replicate them in the HDL code wherever possible.

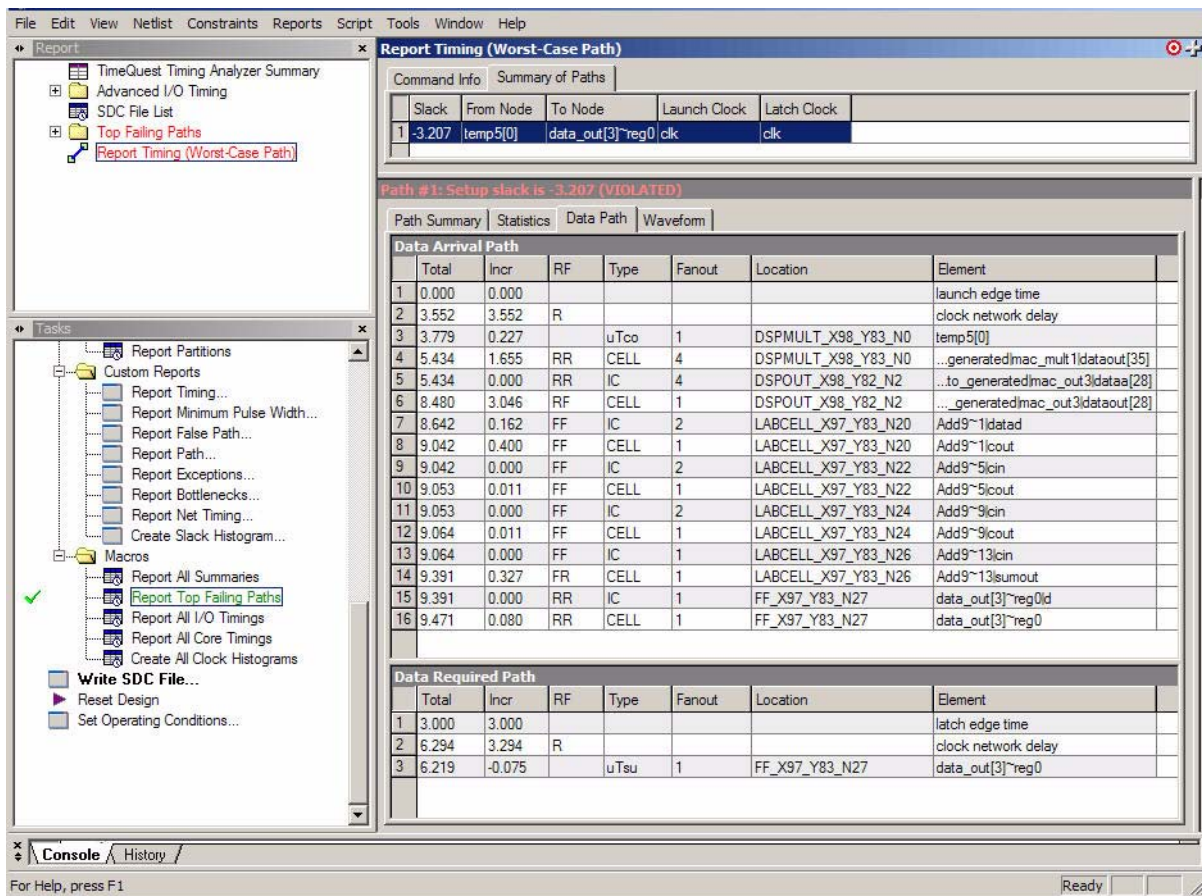
Making changes in the HDL leaves the implementation unchanged, unlike when the software makes the changes. Depending on the specific situation and other incremental changes on your design, the Quartus II software may choose to optimize the design differently in subsequent compilations. Many register optimizations are also computation resource and compilation time intensive. By making the required change in the HDL, you can ensure the preferred implementation with a better timing result, and also reduce the impact on compilation time.



For more information about using register retiming, refer to the *Netlist Analysis and Physical Synthesis* and the *Area and Timing Optimization* chapters in volume 2 of the *Quartus II Handbook*.

Figure 11 shows a Quartus II project TimeQuest Timing Analyzer report.

Figure 11. TimeQuest Timing Analyzer



The worst case path has 7 cell delays contributing to the path delay. This path is not meeting timing, because of the high number of logic levels. In such cases, consider splitting the critical path by adding pipeline registers or modifying the logic in some other way.

Use the Quartus II Advisors

The Quartus II software provides several advisors to help with design implementation. Among the different advisors available, the most useful for timing closure are the Timing Optimization Advisor and the Incremental Compilation Advisor.

The Timing Optimization Advisor helps you find settings that improve timing, along with the trade-offs of using those options. The recommendations are organized in different stages, based on how they affect the timing results. The needs and trade-offs with each design are different, so when you compile a design, run the Timing Optimization Advisor if your focus is on enhanced performance. The Timing Optimization Advisor suggests various ways to improve performance.

Often project settings are carried over from earlier design projects. A setting that was helpful for another design may not help in the current design. Always run the Timing Optimization Advisor to ensure you have appropriate settings.

Use the Timing Optimization Advisor for suggestions on various modifications to improve f_{MAX} , I/O timing, hold and minimum delay timing, and metastability optimization. On the right hand panel of the Timing Optimization Advisor, you can see how each of the selected changes might affect your design. For example, you might have set the **Fitter Effort** to **Fast Fit** during a trial run, but if you would like to optimize a block, you should set the **Fitter Effort** to **Auto Fit**. The Timing Optimization Advisor flags the settings you already have in the design, but that do not help with achieving your desired optimization goals. It also allows you to change settings to best suit your requirements.

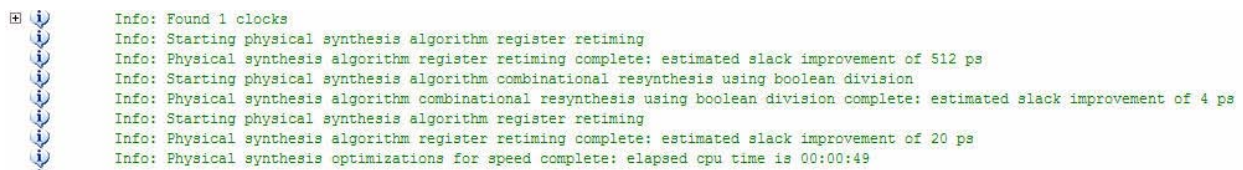
Review Quartus II Compilation Reports

Review the messages from the Quartus II Fitter to aid in timing closure, because the messages offer information for design performance improvement. The messages might also indicate problems with your design, such as unsuitable logic structures (for example, latches and combinational logic loops). The messages warn you about missing timing constraints. They also help you evaluate how settings optimize the design.

You can use the Compilation Report to observe the effectiveness of the compilation options used. If you use high effort options, such as physical synthesis options, the Fitter messages include the estimated slack improvement. The magnitude of the estimated slack improvement shows if the improvement expected is small (a few picoseconds). If so, the additional options implemented are likely to result in little to no improvement. Avoid using options that give you insignificant improvements to reduce compilation time.

Figure 12 shows an example from the Compilation Report of a Quartus II project.

Figure 12. Compilation Report of a Quartus II Project



```

Info: Found 1 clocks
Info: Starting physical synthesis algorithm register retiming
Info: Physical synthesis algorithm register retiming complete: estimated slack improvement of 512 ps
Info: Starting physical synthesis algorithm combinational resynthesis using boolean division
Info: Physical synthesis algorithm combinational resynthesis using boolean division complete: estimated slack improvement of 4 ps
Info: Starting physical synthesis algorithm register retiming
Info: Physical synthesis algorithm register retiming complete: estimated slack improvement of 20 ps
Info: Physical synthesis optimizations for speed complete: elapsed cpu time is 00:00:49
  
```

In Figure 12, physical synthesis optimizations for retiming is giving an estimated slack improvement of 512 ps, while the physical synthesis for combinational synthesis is estimated to give only 4 ps. To meet your design performance requirements, choose options that improve performance.

You can also find the fraction of the Fitter time that is spent on placement in the Compilation Report. If the routing time is too large, the Fitter may not have been able to find an optimal placement. In such a case, use a higher placement effort multiplier in the Fitter settings to get a better placement.

 For more information about different Fitter settings available in the Quartus II software, refer to the *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*.

Conclusion

Timing closure is a critical phase of your design cycle, which determines the success or failure of a product. Plan for timing closure rather than trying to meet the timing requirements with an ad-hoc approach. By following the guidelines indicated in this application note, the process of planning for timing closure can be done efficiently.

For the Design Planning, Implementation, Optimization and Timing Closure Checklist, refer to the [Quartus II Design Checklist](#).

Referenced Documents

The following documents are referenced in this application note:

- [Analyzing and Optimizing the Design Floorplan](#) chapter in volume 2 of the *Quartus II Handbook*
- [Analyzing Designs with Quartus II Netlist Viewers](#) chapter in volume 1 of the *Quartus II Handbook*
- [Area and Timing Optimization](#) chapter in volume 2 of the *Quartus II Handbook*
- [Design Planning with the Quartus II Software](#) chapter in volume 1 of the *Quartus II Handbook*
- [Design Recommendations for Altera Devices and the Quartus II Design Assistant](#) chapter in volume 1 of the *Quartus II Handbook*
- [Design Space Explorer](#) chapter in volume 2 of the *Quartus II Handbook*
- [In-System Design Debugging](#) section in volume 3 of the *Quartus II Handbook*
- [Managing Metastability with the Quartus II Software](#) chapter in volume 1 of the *Quartus II Handbook*
- [Netlist Analysis and Physical Synthesis](#) chapter in volume 2 of the *Quartus II Handbook*
- [Quartus II Design Checklist](#)
- [Quartus II Incremental Compilation for Hierarchical and Team-Based Design](#) chapter in volume 1 of the *Quartus II Handbook*
- [The Quartus II TimeQuest Timing Analyzer](#) chapter in volume 3 of the *Quartus II Handbook*
- [Quartus II TimeQuest Timing Analyzer Cookbook](#)
- [Recommended HDL Coding Styles](#) chapter in volume 1 of the *Quartus II Handbook*
- [Selector Guide](#) page of the Altera website
- [Simulation](#) section in volume 3 of the *Quartus II Handbook*
- [Virtual JTAG \(sld_virtual_jtag\) Megafunction User Guide](#)

Document Revision History

Table 1 shows the revision history for this application note.

Table 1. Revision History

Date and Version	Changes Made	Summary of Changes
August 2009, v1.0	Initial release.	—



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support

Copyright © 2009 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.

