



AN 584: Timing Closure Methodology for Advanced FPGA Designs

Updated for Intel® Quartus® Prime Design Suite: **21.3**



Online Version

Send Feedback

AN-584

ID: **683145**

Version: **2021.10.08**

Contents

1. AN 584: Timing Closure Methodology for Advanced FPGA Designs.....	3
1.1. Plan Early for Timing Closure.....	4
1.2. Customize Settings Per Application.....	5
1.3. Change Fitter Placement Seeds.....	5
1.4. Planning for Timing Closure.....	7
1.4.1. Timing Closure Planning at Specification Stage.....	7
1.4.2. Timing Closure Planning at Coding and Compilation Stage.....	8
1.5. Best Practices for Timing Closure.....	11
1.5.1. Performance Planning General Guidelines.....	11
1.5.2. Follow Synchronous Design Practices.....	12
1.5.3. Follow Recommended Coding Styles.....	12
1.5.4. Constraining and Compiling Your Design.....	13
1.6. Resolving Common Timing Issues.....	16
1.6.1. Excessive Logic Levels.....	17
1.6.2. Improper Timing Constraints.....	19
1.6.3. Handling High Fan-Out Registers.....	20
1.6.4. Metastability Issues.....	21
1.6.5. Reset Signal Related Issues.....	24
1.6.6. Small Margin Timing Constraint Failures.....	26
1.6.7. Long Compilation Times.....	26
1.7. Conclusion.....	27
1.8. Document Revision History for AN 584: Timing Closure Methodology for Advanced FPGA Designs.....	28



1. AN 584: Timing Closure Methodology for Advanced FPGA Designs

Today's design application and performance requirements are more challenging than ever due to increased complexity. With the evolution of system-on-a-chip, the size of typical designs are ever increasing. Complexities such as external memory interfaces and mixed signal devices can challenge timing closure. You may not have control over the pipelining or partitioning of IP blocks.

Nevertheless, your design must accommodate timing requirements for the IP in the system to achieve a fully functional design. If you are unable to completely meet performance requirements for any part of a design, the system may fail to function as you want.

This application note presents design independent techniques for timing closure. Whether you use Application Specific Standard Products (ASSPs), Application Specific Integrated Circuits (ASICs), or Field Programmable Gate Arrays (FPGAs), rapid timing closure poses a challenge for system design that you can overcome using these methods.

The Intel® Quartus® Prime Fitter's default settings are set to help you meet required timing constraints for most designs. However, for designs that cannot meet timing requirements with default settings, use the methodology in this application note to shorten design cycles, reduce complexity, and achieve timing closure requirements as quickly as possible.

Shorten Design Cycles

Typically, current FPGA systems are characterized by shorter product life cycles driven by market pressures. To be successful, designers must design, test, and bring the product to the market as quickly as possible. Often designers must prioritize design verification because delivering successful products at the first trial is essential for economic viability.

The Intel Quartus Prime software provides features that help you meet market pressure and stringent performance goals along with shorter design cycles. The Intel Quartus Prime software provides accurate timing models, advanced timing analysis, and fine-tuned Fitter algorithms to meet your goals.

With the default Intel Quartus Prime Compiler settings, you can often achieve push-button timing closure for typical FPGA designs. For those designs where push-button timing closure is difficult, the Intel Quartus Prime software helps you to plan for timing closure at the beginning of your design cycle to accelerate timing closure at the end of the cycle.

Reduce Timing Closure Complexities and Conflicts

Many factors can increase the difficulty of timing closure. For example, the placement of specific resources in a location on the FPGA could be a complexity. Specialty blocks, such as DSPs, RAMs, and transceivers are sometimes located in areas of the FPGA where routing availability can be problematic because of congestion around the blocks. Poor resource placement by the Fitter can result in timing not meeting all requirements.

Timing closure conflicts can occur between the resource, area, power, and timing requirements that you specify for your design. For example, often mobile devices must trade power for speed considerations. If your design requires more resources, you must distribute the resources across the target FPGA device. Widely distributed resources tend to have long interconnections. At smaller device geometries, delays are dominated by interconnect delays rather than cell delays. To have shorter net lengths, you ideally have a smaller area. Therefore, these two requirements generally conflict.

Another common timing closure conflict occurs between reliability and the time available for verification. Because of the reduced market window that dictates your product's success, system designer's want to have a design working within the shortest amount of time, at the lowest possible cost, for a product that 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. All of these complexities increase the challenge of closing timing on a design.

Follow these guidelines and methodology to improve productivity, close timing faster, and reduce the number of Compiler iterations:

- [Plan Early for Timing Closure](#) on page 4
- [Customize Settings Per Application](#) on page 5
- [Change Fitter Placement Seeds](#) on page 5
- [Planning for Timing Closure](#) on page 7
- [Best Practices for Timing Closure](#) on page 11
- [Resolving Common Timing Issues](#) on page 16

1.1. Plan Early for Timing Closure

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

Always practice synchronous design techniques and follow Intel FPGA recommended HDL coding practices that are independent of other EDA tools. For an effective design, you must choose the target device architecture and properly constrain your design for timing. Identify any false and multicycle paths in your design to generate an accurate timing analysis report. The accuracy of timing analysis is dependent on the proper application of timing constraints and exceptions. Proper constraints and exceptions cause the Compiler to apply extra effort in specific areas to meet the constraints.

For more details about good design and coding practices, and constraining your design for timing, refer to *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*.

1.2. Customize Settings Per Application

You must consider a variety of features before choosing a device or a specific technology for your applications. For example, factors such as device cost, operating speed, and power are some key points for consideration early in your system design.

The Intel Quartus Prime software includes many precise settings that help you to achieve your specific design goals. The default settings generally provide the most balanced performance, power, and resource optimization trade-offs, but you can choose non-default settings to tailor the implementation for your application.

1.3. Change Fitter Placement Seeds

The Intel Quartus Prime Compiler's Fitter stage performs the place and route of your design. When run, the Fitter creates a semi-random initial placement for the logic it derives from the initial condition of your design RTL. The Fitter's goal is to find a placement that the Compiler can successfully route and that also meets all constraints that you specify.

It is possible to achieve a successful place and route using different initial placements. The Fitter searches for the best solution among a set of different possible and valid solutions within the solution space. The Fitter might converge to different solutions in the solution space, depending on the given initial conditions.

The initial condition of the design is a function of all the source files, optimization settings, and the **Fitter Initial Placement Seed** setting value. A change in any of these variables results in a change in the initial placement. A change in initial placement affects how optimizations proceed, producing a different Fitter result. This variation in Fitter results is the "Fitter seed effect."

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

A new Fitter run with a different seed places the Fitter into a new area in the solution space, resulting in a different initial placement. If you change the Fitter seed, the Compiler might converge on a different solution because of the modified initial placements. Therefore, changing the Fitter seed can produce different place and route results.

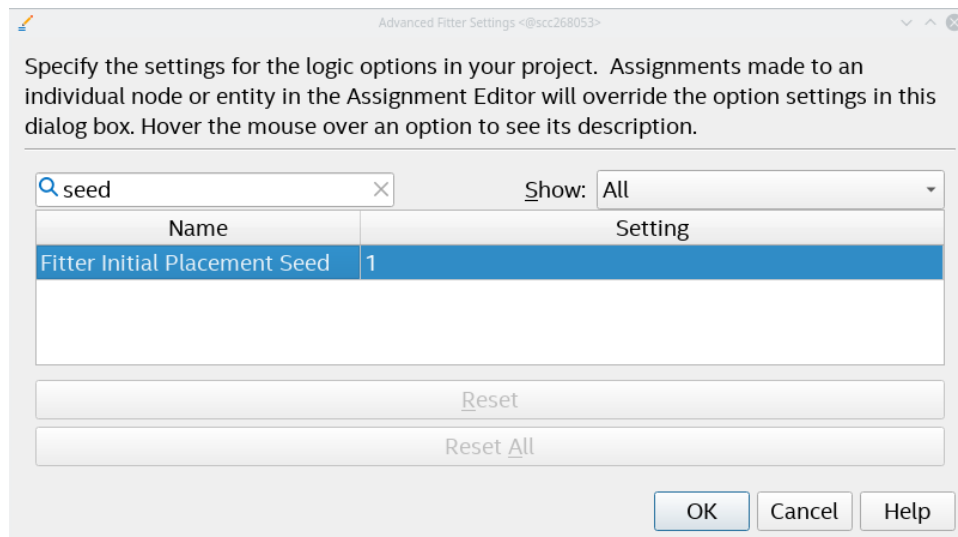
You specify a non-negative integer as an input to the Fitter seed. Changing the Fitter seed value may not produce a better fit, but the change does make the Fitter use a different initial placement. This integer seed value allows you to try different initial placements without any change in the design or settings.

You can run a "seed sweep" to try different initial placements to guide a Fitter, using different seeds or settings to find the best performance results for a given design using the same optimization settings.

You can use a seed sweep to determine the optimal seed value for your design if other initial conditions remain unchanged. The default **Fitter Initial Placement Seed** value is 1. To change the default **Fitter Initial Placement Seed** value:

1. Click **Assignments > Settings > Compiler Settings**.
2. On the **Optimization Mode** tab, click the **Advance Settings (Fitter)** button.
3. In the search field, type `seed`.
4. For **Fitter Initial Placement Seed**, specify a new non-negative integer value.

Figure 1. Fitter Initial Placement Seed



As an alternative to the GUI setting, you can specify the following equivalent setting in the .qsf file:

```
set_global_assignment -name SEED <value>
```

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.

1.4. Planning for Timing Closure

Proper planning can help you achieve timing closure faster at the end of the design cycle. Because there are no set rules for timing closure 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.

1.4.1. Timing Closure Planning at 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 that shows partitioning of 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, while very large design blocks can be difficult to debug. Try creating blocks that encapsulate distinct functionality. Keep blocks to a size that is convenient for debugging during functional simulation and timing closure.

Refer to the following topics for timing closure planning at the specification stage:

[Plan for the Target FPGA Device](#) on page 7

[Plan for On-Chip Debugging](#) on page 8

1.4.1.1. Plan for the Target FPGA Device

The devices in each Intel FPGA family are available with different design densities, speed grades, and packaging options to accommodate different applications. In design planning, choose a device with a specification that meets your timing requirements.

Some FPGA device feature requirements to consider are:

- Performance
- Logic and memory density
- I/O density
- Power utilization
- Packaging
- Cost

The Intel Quartus Prime software optimizes and analyzes your design using different timing models for each speed grade. If you migrate to a device with a different speed grade, you must perform timing analysis again to ensure that there are no timing violations due to changes in the device speed grade.

For more information about choosing a device, refer to *Design Planning* in *Intel Quartus Prime Pro Edition User Guide: Getting Started*.

To quickly find and compare the specifications and features of Intel FPGA devices and development kits, refer to the Product Selector tool on the Intel website.

Related Information

- [Design Planning, Intel Quartus Prime Pro Edition: Getting Started](#)

- [Product Selector tool](#)

1.4.1.2. Plan for On-Chip Debugging

The Intel Quartus Prime software includes 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, Intel Quartus Prime project, and design files are all set-up to support the appropriate options.

Timing errors due to unspecified timing requirements can appear as functional failures of the design. If you are able to locate the functional block where errors originate, it is easier to find the source of the errors.

For more information about in-system design debugging, as well as the Intel Quartus Prime debugging tools, refer to *Intel Quartus Prime Pro Edition User Guide: Debug Tools*.

Related Information

[Intel Quartus Prime Pro Edition User Guide: Debug Tools](#)

1.4.2. Timing Closure Planning at Coding and Compilation Stage

The coding and compilation approach that you use can help you achieve faster timing closure. The following section describes planning for design hierarchies and partitions, planning for early compilation of individual design blocks, and planning for design verification during the coding and compilation stage of the design flow:

[Plan for Design Hierarchy and Block Partitioning](#) on page 8

[Plan for Early Compilation of Design Blocks](#) on page 9

[Plan for Verification](#) on page 10

1.4.2.1. Plan for Design Hierarchy and Block Partitioning

Flat designs are generally more difficult to optimize and debug because you cannot always isolate the timing issue. Using a hierarchical design methodology offers several advantages, such as:

- Hierarchical designs allow easier debug and optimization of individual design blocks.
- You can assign the design hierarchy elements into logical partitions that are functionally independent.
- These partitions allow stand-alone block verification.
- You can use design blocks for reuse and preserve synthesis and timing results for blocks that are fully coded and meeting timing.

Planning ahead by appropriately partitioning your design reduces the need for unplanned changes when closing timing at the end of the design cycle.

When you change RTL code or Compiler settings for one block in the design, this produces different compilation results compared to previous settings. The different compilation results can cause timing violations in blocks that do not reflect the same

corresponding code or setting changes. However, with the block-based incremental compilation flow, you can preserve earlier results for a block that you do not want to change.

The incremental block-based compilation feature allows you to partition a design, compile the design partitions separately, and reuse the results for unchanged partitions. You can preserve performance of unchanged blocks and reduce the number of design iterations. The performance preservation of incremental block-based compilation allows you to focus timing closure on unpreserved partitions, or on blocks that have difficulty meeting timing requirements.

Design block reuse and incremental block-based compilation flows are available in the Intel Quartus Prime software. For more information on effectively using these flows, refer to *Intel Quartus Prime Pro Edition User Guide: Block-Based Design*.

Related Information

[Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)

1.4.2.2. Plan for Early Compilation of Design Blocks

After first identifying the major functional blocks of the design, you can partition the design into manageable blocks to facilitate multiple designers and independent and incremental optimization of design blocks.

Compile the major blocks in your design as soon as you can, even if your design is not complete. By doing this, you can also identify resource issues early in the design cycle.

A common reason for prolonged design cycles is waiting for all code completion before compiling the design. With this approach, you do not detect significant issues until the design and dependencies are very mature.

Using the incremental block-based compilation flow, you can assign blocks that are not yet available, or that are being developed independently as "empty" partitions.

When creating design partitions:

- Try to minimize inter-block connections.
- Register all inputs and outputs from each block to avoid critical timing paths crossing between partition boundaries.
- Details of other partitions contents are not visible to the Compiler when compiling a single partition block. Therefore, optimization (or logic minimization) across partitions cannot occur.
- Add each major block or partition in your design as soon as HDL coding is completed and compile incrementally.
- If certain block partitions are finished and no modification is needed, you have the option to preserve these partitions as soon as they meet timing requirements.

Incremental timing closure can help your design meet timing faster, as the Fitter works more on optimizations and small RTL changes without influencing any preserved partitions.

Compiling blocks early using the incremental block-based compilation helps you to uncover and resolve design issues early before they can become complicated problems in the final timing closure stages.

You can set the following options to reduce the total compile time during the initial stages of your design. These options are especially helpful for large or complex blocks that you want to initially compile independently, or when ready to start integrating the block:

Click **Assignments > Settings > Compiler Settings > Optimization Mode > Compile Time**, and then choose one of the following:

- **Aggressive Compile Time**
- **Fast Functional Test**

Both of these modes perform initial timing and functional analyses without requiring excessive compile time.

Figure 2. Compile Time Optimization Mode Settings

Optimization Mode Register Optimization Incremental Compile

Performance:

- ☒ High performance effort
- ☐ High performance with maximum placement effort
- ☐ High performance with aggressive power effort
- ☐ Superior performance (adds synthesis optimizations for speed)
- ☐ Superior performance with maximum placement effort

Area:

- ☐ Aggressive Area (reduces performance)

Routability:

- ☐ High placement routability effort
- ☐ High packing routability effort
- ☐ Optimize netlist for routability

Power:

- ☐ Aggressive power (reduces performance)

Compile Time:

- ☐ Aggressive Compile Time (reduces performance)
- ☐ Fast Functional Test (hold-timing optimization only)

For more information about incremental block-based compilation, refer to *Incremental Block-Based Compilation Flow* and *Setting-Up Team-Based Designs* sections of the Intel Quartus Prime Pro Edition User Guide: Block-Based Design.

Related Information

- [Setting-Up Team-Based Designs, Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)
- [Incremental Block-Based Compilation Flow, Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)

1.4.2.3. Plan for Verification

In general, designs that target large FPGAs are complex in nature, requiring comprehensive planning for design verification. You can use any of the supported third-party simulators to run functional verification on your design.

Plan to verify the design by exercising it with a simulation test suite at the RTL level. This level of verification ensures that the HDL contains the functionality you want, allowing you to focus on any potential timing problems.

If you have functional and timing problems simultaneously, it can be difficult to isolate issues and the correction that you need, which increases the necessary debug time.

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

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

For more information about using third-party simulators for simulation, refer to *Intel Quartus Prime Pro Edition User Guide: Third-party Simulation*.

Related Information

[Intel Quartus Prime Pro Edition: Third-party Simulation](#)

1.5. Best Practices for Timing Closure

The following sections provide general guidelines for performance planning, and describe best practices for coding styles and constraints:

[Performance Planning General Guidelines](#) on page 11

[Follow Synchronous Design Practices](#) on page 12

[Follow Recommended Coding Styles](#) on page 12

[Constraining and Compiling Your Design](#) on page 13

1.5.1. Performance Planning General Guidelines

The following are general guidelines for performance planning:

- Optimize and debug incrementally by creating appropriate hierarchical blocks and partitions in your project.
- Register all block inputs and outputs.
- Optimize major blocks with higher than required speed when running compilation on individual design blocks.

Note: Retaining approximately 10%-20% timing margin at the block level can help achieve timing closure after integration of the blocks.

- Plan for the available resource types in the target device (such as the necessary RAM blocks, DSP blocks, PLLs, and transceiver locations) while coding RTL for your design.
- Pipeline the design for better performance.

Related Information

[Intel Quartus Prime Pro Edition User Guide: Design Optimization](#)

1.5.2. Follow Synchronous Design Practices

Following synchronous design practices can simplify the specification of timing constraints.

Although asynchronous techniques might save time in the short run and seem easier 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 glitches and race conditions that can render the resulting implementation unreliable, increasing the complexity of constraints.

In the absence of appropriate constraints, synthesis or place-and-route tools may not perform the best optimizations, resulting in inaccurate timing analysis results.

Moreover, in a synchronous design, a clock signal triggers every event. With the Fitter working to achieve all timing requirements, a synchronous design behaves in a more reliable manner for all PVT conditions. Synchronous designs easily migrate to different device families or speed grades.

For more information about using synchronous design practices for FPGA designs, refer to *Synchronous FPGA Design Practices* in *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*.

Related Information

[Following Synchronous Design Practices, Intel Quartus Prime Pro Edition User Guide: Design Recommendations](#)

1.5.3. Follow Recommended Coding Styles

Your coding style has a significant impact on implementation in the FPGA because synthesis tools can optimize and interpret the design differently based on style. Therefore, you must decide how to code 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, without adversely affecting resource use. Adding adequate pipeline registers can help you avoid a large amount of combinational logic between registers.

Another practice to avoid is unintended latch inference. Intel Quartus Prime Synthesis issues a warning message when unintended latch inference occurs. The FPGA architecture is not optimized for latch implementation. Latches generally have slower timing performance compared to equivalent registered circuitry.

Improper use of latch schemes can also cause glitches to pass from input to output when signals are enabled, putting latches in transparent mode. You must design such structures properly to avoid this unexpected circumstance. However, timing analysis cannot identify these safe applications.

Refer to *HDL Design Guidelines* in *Intel Quartus Prime Pro Edition User Guide: Design Recommendations* for information on how to avoid latches, combinational loops, and other styles that are not suitable for FPGA implementation.

If you use unsupported operations (such as asynchronously clearing of RAM content) with memory blocks, the Intel Quartus Prime software may implement the code with logic cells rather than more suitable RAM blocks.

If you are unaware of your hardware resources, you can easily underutilize some of the available resources. In general, do not use constructs that lack the equivalent hardware implementation available in the device. For example, if you infer RAM locations and use synchronous resets on RAM locations to clear the contents, or to initialize the values, your code might not be mapped to any of the available RAM blocks in the device.

This condition occurs because the RAM locations in the FPGA device 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 device specifications to confirm whether having a known initial value in the RAMs is necessary for proper design function. This condition is not typically required. If RAM cells must initialize to certain known values (such as all 0's), you can perform write cycles to the RAM immediately after power up.

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

Improper coding of RAM blocks can also cause inference of the native RAM surrounded by an unnecessary cloud of logic cells and registers required to implement the equivalent functionality. This cloud of logic is barely noticeable, but can cause extra resource utilization and worsen timing closure in the area.

Refer to *Recommended HDL Coding Styles in Intel Quartus Prime Pro Edition User Guide: Design Recommendations* for information on inferring memories, multiplier blocks, DSP blocks, and how to use provided HDL RAM templates.

Related Information

- [HDL Design Guidelines, Intel Quartus Prime Pro Edition User Guide: Design Recommendations](#)
- [Recommended HDL Coding Styles, Intel Quartus Prime Pro Edition User Guide: Design Recommendations](#)

1.5.4. Constraining and Compiling Your Design

This section describes best practices for constraining and compiling your design.

1.5.4.1. Setting Location Constraints

You can specify location constraints that place logic and I/O blocks at specific locations in the chip during place and route. However, the Compiler's default logic placement is generally more suitable than specifying specific location constraints.

For example, it may not be helpful to assign a block containing a critical path to a Logic Lock region that you subsequently constrain and squeeze the path. The Fitter identifies and attempts to optimize the critical path by considering many physical constraints to find the best placement. Restricting this auto-placement can reduce performance and requires careful use. There are times when location constraints effectively aid in timing closure, directly or indirectly.

You can define Logic Lock regions or location constraints to constrain logic blocks to specific areas in the FPGA. This technique allows you to create a floorplan for your design that supports the incremental block-based compilation flow and team-based designs.

In a team-based designs, a different designer can design each major block in the project. For such cases, you define locations for 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 also reserve the region on a previous Fitter-assigned area.

Note: Excessive location constraints can negatively affect design performance.

Refer to *Setting-Up Team-Based Designs* in *Intel Quartus Prime Pro Edition User Guide: Block-Based Design*.

Related Information

[Setting-Up Team-Based Designs, Intel Quartus Prime Pro Edition User Guide: Block-Based Design](#)

1.5.4.2. Setting Proper Timing Constraints

The Intel Quartus Prime software preserves timing constraints in Synopsys Design Constraints (.sdc) file format that uses the Tcl syntax. You can embed these .sdc constraints in a scripted compilation flow, and even create sets of .sdc files for timing optimization.

The Fitter uses the .sdc timing constraints to further optimize your design. The Timing Analyzer uses the .sdc timing constraints for static timing analysis.

By default, the Timing Analyzer assumes that all 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 asynchronous to each other, use the `set_clock_groups` command in your constraint file to specify this relationship in your clock groups.

Investigate the relationship between various clocks in your design and categorize them appropriately. 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.

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 the correct inter-clock constraints are present. Otherwise, the Compiler cannot focus effort on only the most critical paths.

For examples that show different scenarios for constraining your design, refer to the *Intel Quartus Prime Timing Analyzer Cookbook* and *Applying Timing Constraints* section of *Intel Quartus Prime Pro Edition User Guide: Timing Analyzer*.

Related Information

- [Intel Quartus Prime Timing Analyzer Cookbook](#)
- [Applying Timing Constraints, Intel Quartus Prime Pro Edition User Guide: Timing Analyzer](#)

1.5.4.3. Using Optimal Compiler Settings for Your Design

In addition to using the correct design methodology and proper timing constraints, use the optimal Compiler settings for your design goals.

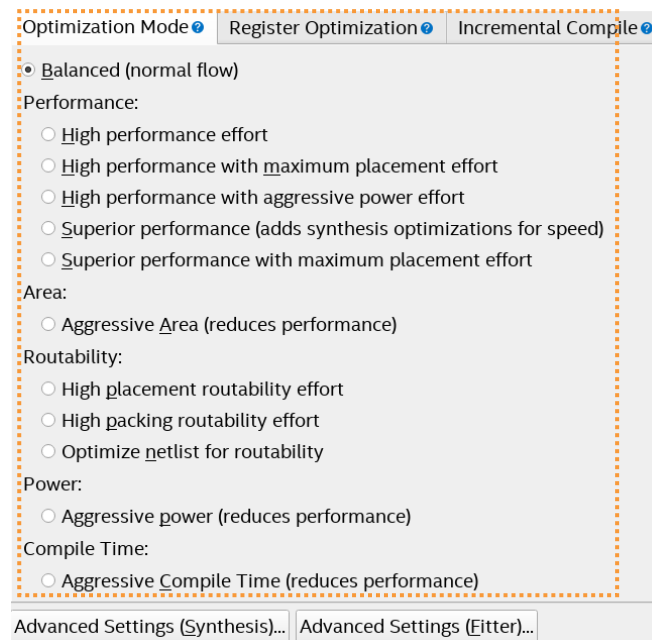
The Intel Quartus Prime software offers many settings to help you meet different design requirements. If you do not specify the appropriate setting, design performance may suffer.

The Intel Quartus Prime software has various settings to help meet different design requirements. For example, if your design targets a mobile device, you might specify the **Aggressive power** setting to optimize your design for lowest power consumption. Similarly, if your design targets a low-cost system, you might pick a device with a specific architecture and optimize your design for that architecture. If you want to reduce the size of the design, you can optimize for **Aggressive area**.

The Compiler's default **Optimization Mode** settings offer a balance of performance, area, routability, power, and compilation time. By trading off one optimization for another, you can meet your preferred design requirements.

Select the **Optimization Mode** in **Assignments > Settings > Compiler Settings > Optimization Mode**.

Figure 3. Compiler Optimization Modes

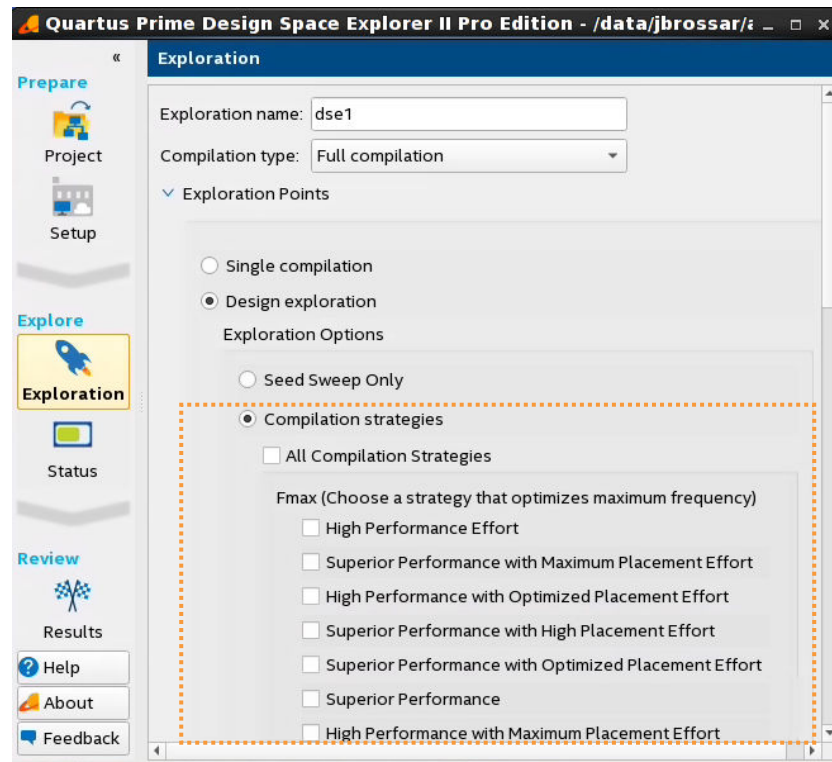


There is no single collection of settings that achieves the best performance for all designs. Each design is unique, and a setting set that derives the best performance for one design has a different impact in another design.

Specifying the **Superior performance** optimization modes can help in some designs, but some of the modes can have a detrimental impact on your primary goal. You might be making optimizations that increase the compilation time without a corresponding improvement in performance. Only apply settings that help meet your design goals.

The settings that you specify for a project revision are specific to that revision. When you make substantial changes to a design, you may need to modify some of the Compiler settings. Whenever you make large changes in your design, you can use the Design Space Explorer II (DSE) to run seed sweeps that identify the best collections of settings for your revision. You can do multiple DSE runs with different compile settings and using multiple seeds by clicking **Tools > Launch Design Space Explorer II**.

Figure 4. Design Space Explorer II Exploration Options



1.6. Resolving Common Timing Issues

When the Timing Analyzer reports failing paths, verify if the failing requirements are correct. By default, the Timing Analyzer handles all clock domains as related, and this can cause some invalid requirements. To correct this condition, group clocks in your design appropriately. Analyze the datapath to determine how you can reduce any critical path delays.

This section contains examples of commonly encountered timing issues and troubleshooting practices:

[Excessive Logic Levels](#) on page 17

[Improper Timing Constraints](#) on page 19

[Handling High Fan-Out Registers](#) on page 20

[Metastability Issues](#) on page 21

[Reset Signal Related Issues](#) on page 24

[Small Margin Timing Constraint Failures](#) on page 26

[Long Compilation Times](#) on page 26

1.6.1. Excessive Logic Levels

Excessive levels of combinational logic in your design can increase the delay on a path and cause that path to become critical. The number of logic levels between registers may be difficult to visualize in your RTL code alone.

As an example, conditional statements are always translated as additional levels of logic. Verify situations before adding conditional statements within another conditional statement. Ensure that the modifications you make apply the conditions only on the branch where it is necessary.

The example below shows a section of a Verilog HDL code. Assume that `counter1` is an 8-bit counter, and that `counter2` is a 12-bit counter. `TC1` and `TC2` represent the status count signals for these counters:

```
if (reset == 0) begin
    counter1 <= 0;
    counter2 <= 0;
    TC1 <= 1'b0;
    TC2 <= 1'b0;

    end else begin
        if (incr ==1) begin
            if (counter1 == value1) begin
                TC1 <= 1'b1;
                if (counter2 == value2) begin
                    TC2 <= 1'b1 ;
                end else begin
                    counter2 <= counter2 +1 ;
                end
            end else begin
                counter1 <= counter1 +1 ;
            end
        end
    end
end
```

The updates for `counter2` are based on the results of the nested 8-bit and 12-bit comparators. Depending on the device architecture, this combinational logic that implements in several levels can form a critical path. The way the logic maps can affect the number of constituent delays.

The Timing Analyzer design metric reports provide a summary of the logic depths in your design per clock domain.

In the Timing Analyzer, click **Tasks > Reports > Design Metrics > Report Logic Depth** to generate this report.

[Figure 5](#) on page 18 shows `clk_2x` domain's worst depth is at 4. You can check the paths with this depth by right-clicking on the clock domain and depth element to generate a timing report.

Figure 5. Report Logic Depth

Report Logic Depth						
Show:	Visible	Hide	Q <<Filter>>			
	Clock Name	Clock Period	Depth 0	Depth 1	Depth 2	Depth 3
1	clk	1.500	910	6	0	0
2	clk_2x	1.500	79	6	29	621

Figure 6. Report Paths of Depth 4 or More

Path #1: Setup slack is -0.172 (VIOLATED) (logic depth: 4)						
Path Summary	Statistics	Data Path	Waveform			
1	Setup Relationship	1.500				
2	Clock Skew	-0.046				
3	Data Delay	1.724				
4	Number of Logic Levels	4				
5	Physical Delays					
1	Arrival Path					
1	Clock					
1	Clock [uncapped]					
1	Data					
1	IC	5	0.775	45	0.078	0.251
2	Cell	10	0.699	41	0.000	0.124
3	IC	2	0.250	25	0.250	0.250
2	Required Path					
1	Clock					
1	Clock [uncapped]	1	2.441	100	2.441	2.441

Path #1: Setup slack is -0.172 (VIOLATED) (logic depth: 4)						
Path Summary	Statistics	Data Path	Waveform			
1	Total	Incr	RF	Type	Fanout	Location
1	0.000	0.000				
2	2.730	2.730				
1	2.0	2.730	R			
2	2.0	0.000				
3	4.454	1.724				
1	2.0	0.250	RR	uTco	1	FF...N31
2	3.8	0.078	RR	IC	1	LAB...N27
3	3.2	0.124	RF	CELL	2	LAB...N27
4	3.9	0.037	FR	CELL	7	LAB...N27
5	3.5	0.116	RR	IC	1	LAB...N5
6	3.1	0.076	RF	CELL	2	LAB...N5
7	3.5	0.094	FR	CELL	2	LAB...N5
8	3.2	0.097	RR	IC	1	LAB...N3
9	3.4	0.122	RR	CELL	3	LAB...N3
10	3.8	0.084	RF	CELL	2	LAB...N3
11	4.9	0.251	FF	IC	1	ML...N3
12	4.3	0.084	FR	CELL	1	ML...N3
13	4.0	0.017	FR	CELL	1	FF...N5
14	4.1	0.061	RR	CELL	2	FF...N5
15	4.4	0.233	RF	IC	1	BLO...J5
16	4.4	0.000	FF	CELL	1	BLO...J5

The report indicates that the path with the longest depth belongs to the nested comparators in the example, and is contributing to several critical paths on the design.

The **Data Path** tab report on the right shows all the interconnected elements, while the **Statistics** tab report on the left shows a summary of the total cells and interconnect delays (IC). In this case, the delay contributes the biggest percentage on the total delay of the data path at 41% and 45% respectively.

To correct this condition, it is possible to check the required conditions in parallel with, or before the 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 the logic by parallel operations. In these cases, you can consider using pipeline registers to split the logic operations that occur in one cycle. You must account for the effect of this added latency in other parts of the design if you use this approach.

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

- Give priority to improving the code instead of modifying the Compiler 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 the implementation in hardware.
- If you are working on a block within a larger design, target a higher than required timing performance prior to block integration.

1.6.2. Improper Timing Constraints

The Compiler attempts to optimize your design according to the constraints that you provide. Therefore, improper or missing timing constraints can contribute to timing failure. The Timing Analyzer does not analyze unconstrained paths. You must review the Timing Analyzer report to ensure you have constrained all required timing paths.

If you omit some of the required constraints, the Timing Analyzer may generate a report without any violations. However, if the missing constraints are critical, then your design implementation may not perform as intended. Therefore, you must fully constrain your design.

In addition to missing timing constraints, another common cause of timing closure issues is under-specification or over-specification of timing constraints.

You must analyze your timing analysis report for all false or multi-cycle paths. The Timing Analyzer attempts to optimize all paths as valid, single-cycle paths, unless you identify them as false or multi-cycle paths. This might cause valid paths to fail timing requirements (depending on which paths the Fitter optimizes first). To avoid this scenario, identify false and multi-cycle paths in the `.sdc` file.

When you specify aggressive timing constraints on one domain, the Timing Analyzer attempts to optimize that domain earlier than other clock domains. You can use this technique to selectively apply more optimization effort to only one domain. 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 the paths failed the over-constrained requirement. In general, use real requirements for constraining your design.

The 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 the design if you want to improve timing performance on select domains, particularly if compiling individual design blocks. If you can meet more stringent constraints at the block level, it can be easier to meet the timing after you integrate the blocks together. This technique compensates for delays that cannot be accurately predicted at block-level implementation.

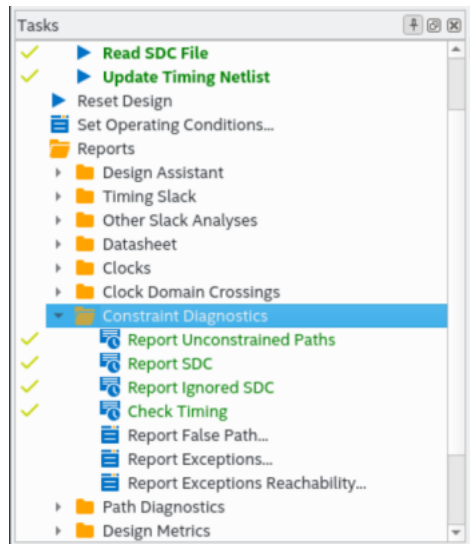
Ensure that you target the right signals that you want to constrain. Be careful in use of wild card characters in defining timing constraints. You can unknowingly constrain unexpected targets, leading to constraints on the wrong paths, or could lead to getting false positive timing results.

1.6.2.1. Check Constraint Diagnostics

The Timing Analyzer can generate a set of diagnostic reports that help check timing constraints.

Click **Tasks** ► **Reports** ► **Constraint Diagnostics** to generate these reports.

Figure 7. Timing Analyzer Constraint Diagnostics



You can use the following commands to generate the reports to verify your constraints:

- **Report Unconstrained Paths**—reports illegal or unconstrained clocks, input or output ports, and paths.
- **Report SDC**—reports all SDC constraints that apply.
- **Report Ignored SDC**—reports all SDC assignments that do not apply.
- **Check Timing**—reports issues on critical fields like latches, loops, no clock drivers, and others.
- **Report False Path**—reports all of the false paths that apply in the design.
- **Report Exceptions**—reports a list all exceptions that apply in the design.
- **Report Exceptions Reachability** —reports the percentage of reachability on the targets you specified in your exceptions.

1.6.3. Handling High Fan-Out Registers

Wide distribution of registers is one of the main causes of excess delay on timing paths.

You can use the Chip Planner to view the location of high fan-out registers and the locations they are driving. In most circumstances, the Fitter placement of registers is superior to manual placement.

For more information about the Chip Planner, refer to *Analyzing and Optimizing the Design Floorplan* in *Intel Quartus Prime Pro Edition User Guide: Design Optimization*.

High fan-out registers, such as broadcast signals driving multiple blocks, placed in different directions, can have placement-warping effects on the floorplan that impact maximum clock speed. You can identify high fan-out registers by clicking **Tasks** ► **Reports** ► **Design Metrics** ► **Report Register Spread**.

Figure 8. Report Register Spread

Report Register Spread (endpoint tension)					
Show:	Visible	Hide	<<Filter>>		
	Register Name	Register Location	Number of Endpoints	Endpoint Centroid	Total Distance of Endpoints to Centroid (Tension)
1	u1_bus_sync[en_sync_u1sync_signal[0]	(172, 22)	74	(175, 24)	3.9
2	u0_bus_sync[en_sync_u0sync_signal[0]	(168, 19)	75	(165, 22)	2.2
3	hp_sclr_sx_u1hp[0][0]	(164, 25)	84	(163, 26)	1.4
4	sel_sync_2x_u1sync_signal[0]	(168, 21)	38	(166, 23)	2.9
5	sel_sync_2x_u1sync_signal[1]	(165, 22)	38	(166, 23)	2.9
6	TC_inst[counter1[2]-RTM_2	(166, 25)	49	(166, 25)	1.3
7	TC_inst[counter1[2]-RTM_11	(168, 25)	49	(166, 25)	1.3
8	hp_sclr_u1hp[5][0]	(174, 21)	6	(170, 20)	10.2
9	TC_inst[counter1[2]-RTM_13	(168, 25)	48	(166, 25)	1.3
10	TC_inst[counter1[2]-RTM_12	(164, 25)	23	(167, 25)	0.9

The report shows the list of possible registers being pulled by sink registers in various directions. To learn more about this report, refer to *Report Register Spread* in *Intel Quartus Prime Pro Edition User Guide: Design Optimization*.

You can correct this problem by duplicating the register using the `DUPLICATE_REGISTER` or `DUPLICATE_HIERARCHY_DEPTH` .qsf assignment.

Alternatively, you can manually modify the RTL to implement the duplication, but you must also add the `preserve_syn_only` synthesis attributes to the duplicates to preserve the nodes through synthesis. The attribute allows the Fitter to do retiming or other optimizations if needed.

```
logic dup_reg /* synthesis preserve_syn_only */;
(*preserve_syn_only*) logic dup_reg;
```

For more information on register duplication commands and techniques, refer to *Duplicate Registers for Fan-Out Control* in *Intel Quartus Prime Pro Edition User Guide: Design Optimization*.

Related Information

- [Analyzing and Optimizing the Design Floorplan, Intel Quartus Prime Pro Edition User Guide: Design Optimization](#)
- [Report Register Spread, Intel Quartus Prime Pro Edition User Guide: Design Optimization](#)
- [Duplicate Registers for Fan-Out Control, Intel Quartus Prime Pro Edition User Guide: Design Optimization](#)

1.6.4. Metastability Issues

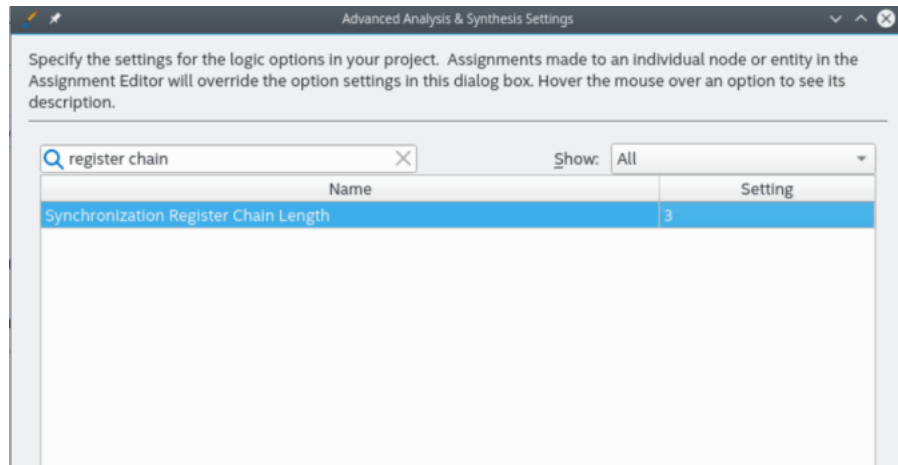
Clock-domain-crossing (CDC) signals can cause metastability on the capturing clock domain register. To reduce the possibility of going metastable, you must synchronize all signals between asynchronous clock domains with multiple stages of synchronizing registers.

Signals with a source in one clock domain may be registered in another clock domain, causing the data input to fail 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.

Generally, a minimum of two stages is recommended. The Intel Quartus Prime software by default is set to look for three register stages to place as close together to obtain better Mean Time Between Failures (MTBF).

To change this default value, click **Assignments > Settings > Compiler Settings > Optimization Mode > Advanced Settings (Synthesis)** and specify a value for **Synchronization Register Chain Length**.

Figure 9. Setting Synchronization Register Chain Length



The Intel Quartus Prime software can analyze designs for metastability, and offers recommendations to reduce metastability. You can compute MTBF on circuit parameters. You should increase the MTBF as much as possible.

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 you do not synchronize control signals to a state machine coming from other clock domains.

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

For more information about metastability, refer to the *Managing Metastability* in *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*.

Related Information

[Managing Metastability, Intel Quartus Prime Pro Edition: Design Recommendations](#)

1.6.4.1. Check CDC Design Assistant Rule Violations

The Intel Quartus Prime software Design Assistant uses design rule checks (DRC) to identify potential issues on signals or buses that are crossing clock domains.

Click **Compilation Report > Timing Analyzer > Design Assistant (Signoff) > Results** to view the CDC report.

Figure 10. Design Assistant CDC DRCs

Name	Description
✓ CDC-50001	1-Bit Asynchronous Transfer Not Synchronized
✓ CDC-50002	1-Bit Asynchronous Transfer with Insufficient Constraints
✓ CDC-50003	CE-Type CDC Transfer with Insufficient Constraints
✓ CDC-50004	MUX-type CDC Transfer with Insufficient Constraints
✓ CDC-50005	CDC Bus Constructed with Multi-bit Synchronizer Chains of Different Lengths
✓ CDC-50006	CDC Bus Constructed with Unsynchronized Registers
✓ CDC-50007	CDC Bus Constructed with Multi-bit Synchronizer Chains with Insufficient Constraints
✓ CDC-50008	CDC Bus Constructed with Multi-bit Synchronizer Chains
✓ CDC-50011	Combinational Logic Before Synchronizer Chain
✓ CDC-50012	Multiple Clock Domains Driving a Synchronizer Chain
✓ CDC-50101	Intra-Clock False Path Synchronizer
✓ CDC-50102	Synchronizer after CDC Topology with Control Signal
✓ CDC-50103	Unsynchronized Intra-Clock Forced Synchronizer

Design Assistant Document:
[CDC-50001](#)
 Tags: synchronizer

Description:
 Violations of this rule identify a single-bit asynchronous data transfer not followed by a synchronizer chain. Such transfers may experience metastability.

A data transfer is considered asynchronous if its launch and latch clocks are unrelated or asynchronous. Clocks are unrelated if they do not share a common parent clock. Clocks are asynchronous if they are explicitly designated as such via a clock group or clock-to-clock false path. Data transfers are also asynchronous if their destination register has the **SYNCHRONIZER_IDENTIFICATION FORCED** instance assignment.

Recommendation:
 Protect single-bit asynchronous data transfers by a synchronizer chain. To do this, ensure that the destination of an asynchronous transfer forms a chain of two or more registers, with each register in the same clock domain as the destination of the transfer, and with no combinational logic between any of the registers in the chain. Also, ensure that there is no combinational logic on the path to the first register in the chain.

To confirm whether the chain is long enough to prevent metastability, run the **report_metastability** command in the Timing Analyzer.

If you do not intend a violating transfer to be asynchronous, ensure that the launch clock of the transfer is correct and is related to the latch clock of the transfer.

The top section of the figure shows the list of CDC rules that Design Assistant verifies. The bottom section provides a description and recommendation for the rule violation.

Refer to *Design Assistant Design Rule Checking* in *Intel Quartus Prime Pro Edition User Guide: Design Recommendations*

Related Information

Design Assistant Design Rule Checking, Intel Quartus Prime Pro Edition User Guide: Design Recommendations

1.6.4.2. Check CDC Report

The Timing Analyzer generates a report on the MTBF of signals crossing clock domains. The Timing Analyzer can also generate a summary list of all the signals, buses, and resets that cross clock domains.

You can generate the summary list by clicking **Tasks > Reports > Clock Domain Crossings** in the Timing Analyzer.

Figure 11. Report Asynchronous CDC Full Report

Report Asynchronous CDC						
Asynchronous CDC Full Report						
Q <<Filter>>						
	CDC Type	Source Nodes	Destination Nodes	Source Clocks	Destination Clocks	Protected Chain Length
1	▼ Intra-Clock False Path Synchronizer (1)					
1	--	sclr	sclr_sync_u1sync_s1[0]	clk	clk	2
2	▼ Compliant Transfer (5)					
1	--	sel[0]	sel_sync_2x_u1sync_s1[0]	clk	clk_2x	2
2	--	sel[1]	sel_sync_2x_u1sync_s1[1]	clk	clk_2x	2
3	--	u0_bus_sync[en_async_reg	u0_bus_sync[en_sync_u1sync_s1[0]	clk	clk_2x	2
4	--	u1_bus_sync[en_async_reg	u1_bus_sync[en_sync_u1sync_s1[0]	clk	clk_2x	2
5	--	sclr	sclr_sync_2x_u1sync_s1[0]	clk	clk_2x	2
3	Synchronizer Not Protected for MTBF (0)					
4	Transfer with Control Signal and Synchronizer (0)					
5	Unsynchroized Intra-Clock Forced Synchronizer (0)					
6	Synchronizer Driven by Multiple Clock Domains (0)					
7	Transfer Preceded by Combinational Logic (0)					
8	Unconstrained Transfer (0)					
9	Unsynchroized Transfer (0)					
2	▼ Multi-bit CDC					
1	▼ Compliant MUX-type CDC Bus (2)					
1	--	u0_bus_sync[bus_async_reg[*]	u0_bus_sync[bus_out_p[*]	clk	clk_2x	1 to 2
2	--	u1_bus_sync[bus_async_reg[*]	u1_bus_sync[bus_out_p[*]	clk	clk_2x	1 to 2
2	Compliant Synchronizer Bus (0)					
3	Unconstrained Synchronizer Bus (0)					
4	Unsynchroized Synchronizer Bus (0)					
5	Synchronizer Bus with Uneven Lengths (0)					
6	Synchronizer after CDC Bus with Control Signal (0)					
7	Unconstrained MUX-type CDC Bus (0)					
8	Compliant CE-type CDC Bus (0)					
9	Unconstrained CE-type CDC Bus (0)					
3	▼ Asynchronous Reset CDC					
1	▼ Compliant Reset Synchronizer (1)					
Single-bit CDC						
Single-bit CDC						
Single-bit CDC		Single-bit CDC				
Property		Value				
1	Description	A topology that transfers a single bit of data across asynchronous clock domains. CDC transfers must feed to compliant synchronizer chains to ensure that the data is not metastable when used, or use a control signal to synchronize				
Single-bit CDC		Single-bit CDC				
Property		Value				
1	Description	A topology that transfers a single bit of data across asynchronous clock domains. CDC transfers must feed to compliant synchronizer chains to ensure that the data is not metastable when used, or use a control signal to synchronize				

1.6.5. Reset Signal Related Issues

Your design can have synchronous or asynchronous reset signals. Typically resets coming into FPGA devices are asynchronous. You can convert an external asynchronous reset to a synchronous reset by feeding it through a synchronizer circuit. You can then use this signal 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 applies.

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. Include the reset signal in the timing analysis along with the other signals in the data path. Using a synchronous reset requires additional routing resources, such as an additional data signal.

If you use an asynchronous reset, you can globally reset all registers. This dedicated resource helps you to avoid the routing congestion that a single reset signal causes. However, a reset that is completely asynchronous can cause metastability issues. This metastability occurs because the time when the asynchronous reset is removed is asynchronous to the clock edge. If you remove the asynchronous reset signal 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 reset synchronizer to the external asynchronous reset for each clock domain and then using the output of the synchronizers to drive the all register resets in their respective clock domains.

The following example shows an example VDD-based (voltage drain drain) reset synchronizer reset synchronizer implementation:

```
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
end

endmodule
```

1.6.5.1. Recovery and Removal Issues

The Timing Analyzer performs recovery and removal analysis in addition to setup and hold analysis. Providing an appropriate reset structure for your design helps the Fitter to place logic to meet recovery and removal timing requirements.

Recovery time is analogous to setup time, and removal time is analogous to hold time. The difference between these sets of timing parameters is that recovery and removal analysis occurs for asynchronous signals (such as reset) with respect to the clock. Recovery and removal analysis helps you to ensure that your synchronous logic behaves correctly when you assert and deassert an asynchronous control signal.

A problem that can occur with a reset signal that spans across the device, is that the signal may not arrive at the same time relative to the clock edge for all the device registers.

When such a reset signal is deasserted, all of the registers should exit reset. However, if the reset signal does not meet the recovery time for some registers in the design, those registers may not exit reset until after the next clock edge.

If such 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 you expect. This condition can cause a design failure. Similarly, a removal error can occur if you remove the reset too early, relative to the clock, and some registers exit reset one cycle earlier.

Using a VDD-based reset synchronizer, and adding another register at the end for duplication to drive different block resets, helps the Fitter to manage the placements to meet these timing requirements.

For more information on resets, refer to *AN 917: Reset Design Techniques for Intel Hyperflex Architecture FPGAs*.

Related Information

[AN 917: Reset Design Techniques for Intel Hyperflex™ Architecture FPGAs](#)

1.6.6. Small Margin Timing Constraint Failures

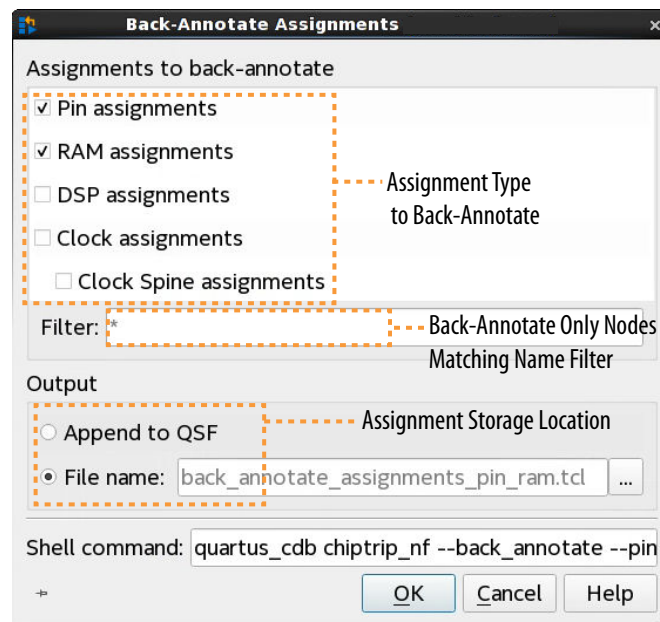
If your design fails to meet timing requirements by a small margin, you can use DSE II to help you select the optimum project settings to meet timing. DSE II runs multiple compilations and compares the results to determine the best combination of settings. Using DSE II to run a seed sweep also helps you compensate for the seed effect, as [Change Fitter Placement Seeds](#) on page 5 describes.

If you make changes to your RTL code or project settings, run a DSE II seed sweep to ensure that the RTL or settings changes are the true cause of improvements, rather than the seed effect.

If your design still fails to meet timing by a small margin, you can use the best performing seed to lock down the placement of clocks, RAMs, and DSPs from that compilation. You lock these placements through assignment back-annotation, and then run another round of seed-sweep using the locked-down placements. The subsequent compilations can benefit from the initial, desired placement settings of the previous best performing seed.

To lock down placement through assignment back-annotation, click **Assignments** ► **Back-Annotate Assignments**, and then select the types of assignments to back-annotate. You can choose to append the back-annotated assignments to the .qsf, or to create a Tcl file of assignments that you can source to the .qsf.

Figure 12. Back-Annotate Assignments Dialog Box



1.6.7. Long Compilation Times

If timing performance is your most important criterion, the Compiler may require additional time to meet stringent requirements. Abnormally long compilations may indicate:

- Resource constraint issues
- Timing constraints that are impossible to meet
- Logic loops that the Compiler cannot easily resolve

For any of these conditions, review the compilation warning messages to determine the portion of the design that is contributing most to the long compilation.

Fitter messages indicate any resource congestion that occurs. You can identify resource congestion by reviewing the resource utilization numbers in the Compilation Report. Utilization of 95% or more can be challenging to fit. In such cases, review your RTL code and consider recoding of blocks to reduce logic use and remove any redundancies.

1.7. Conclusion

Timing closure is a critical phase of your design cycle. The speed of closing timing can determine the success or failure of a product.

Plan for timing closure early, rather than trying to meet the timing requirements with an ad-hoc debugging effort at the end of the design cycle. By following the guidelines of this application note, you can close timing efficiently.

Refer to the following resources for more information on design planning, recommendations, and optimization for rapid design timing closure:

Related Information

- [Intel® Quartus® Prime Pro Edition User Guide: Getting Started](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Design Recommendations](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Design Compilation](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Design Optimization](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Block-Based Design](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Partial Reconfiguration](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Timing Analyzer](#)
- [Intel® Quartus® Prime Timing Analyzer Cookbook](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Power Analysis and Optimization](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Design Constraints](#)
- [Intel® FPGAs Product Selector Guide](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Debug Tools](#)
- [Intel® Quartus® Prime Pro Edition User Guide: Third-party Simulation](#)
- [AN 917: Reset Design Techniques for Intel Hyperflex Architecture FPGAs](#)
- [AN 903: Accelerating Timing Closure in Intel® Quartus® Prime Pro Edition](#)

1.8. Document Revision History for AN 584: Timing Closure Methodology for Advanced FPGA Designs

Document Version	Intel Quartus Prime Version	Changes
2021.10.08	21.3	<ul style="list-style-type: none"> Heavily revised document throughout to update for latest Intel document style, legal, and formatting conventions. Revised the order of topics and grouped related items. Revised headings for precision and clarity. Updated all figures. First version to include an HTML publication of the document.
2014.12.19	14.1	<ul style="list-style-type: none"> Updated product name for DSE II.
2009.08.28	9.0	<ul style="list-style-type: none"> Initial release.