AN 903: Accelerating Timing Closure

in Intel® Quartus® Prime Pro Edition

Updated for Intel® Quartus® Prime Design Suite: **19.3**

Subscribe

Send Feedback

AN-903 | 2020.03.23

Latest document on the web: PDF | HTML

Contents

Send Feedback

1. AN 903: Accelerating Timing Closure in Intel® Quartus® Prime Pro Edition

The density and complexity of modern FPGA designs, that combine embedded systems, IP, and high-speed interfaces, present increasing challenges for timing closure. Late architectural changes and verification challenges can lead to time consuming design iterations.

This document summarizes three steps to accelerate timing closure using a verified and repeatable methodology in the Intel® Quartus® Prime Pro Edition software. This methodology includes initial RTL analysis and optimization, as well as automated techniques to minimize compilation time and reduce design complexity and iterations required for timing closure.

Figure 1. Timing Closure Acceleration Steps

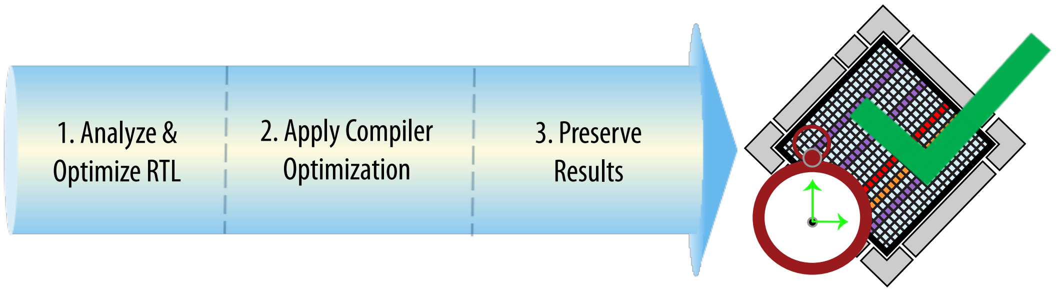


Table 1. Timing Closure Acceleration Steps

Timing Closure Step	Timing Closure Activity	Detailed Info
Step 1: Analyze and Optimize RTL	• Correct Design Assistant Violations on page 4 • Reduce Logic Levels on page 7 • Reduce High Fan-Out Nets on page 9	• Intel Quartus Prime Pro Edition User Guide: Design Optimization • Intel Quartus Prime Pro Edition User Guide: Design Recommendations
Step 2: Apply Compiler Optimization	• Apply Compiler Optimization Modes and Strategies on page 13 • Reduce Congestion for High Utilization on page 16	• Intel Quartus Prime Pro Edition User Guide: Design Compilation • Intel Quartus Prime Pro Edition User Guide: Design Optimization
Step 3: Preserve Satisfactory Results	• Lock Down Clocks, RAMs, and DSPs on page 20 • Preserve Design Partition Results on page 21	• Intel Quartus Prime Pro Edition User Guide: Block-Based Design • AN-899: Reducing Compile Time with Fast Preservation

1.1. Step 1: Analyze and Optimize Design RTL

Optimizing your design's source code is typically the first and most effective technique for improving the quality of your results. The Intel Quartus Prime Design Assistant helps you to quickly correct basic design rule violations, and recommends RTL changes that simplify design optimization and timing closure.

Timing Closure Problems

- Excessive logic levels influences Fitter processing order, duration, and quality of results.
- High fan-out nets cause resource congestion and add additional pull on data paths, needlessly increasing the path criticality, and complicating timing closure.

Timing Closure Solutions

- Correct Design Assistant Violations on page 4—to quickly identify and correct basic design rule violations relevant to your design.
- Reduce Logic Levels on page 7—to ensure that all elements of the design can receive the same Fitter optimizations and to reduce compile times.
- Reduce High Fan-Out Nets on page 9—to reduce resource congestion and simplify timing closure.

Related Information

- "Design Rule Checking with Design Assistant," Intel Quartus Prime Pro Edition User Guide: Design Recommendations
- "Optimize Source Code," Intel Quartus Prime Pro Edition User Guide: Design Optimization
- "Duplicate Registers for Fan-Out Control," Intel Quartus Prime Pro Edition User Guide: Design Optimization

1.1.1. Correct Design Assistant Violations

Performing initial design analysis to eliminate known timing closure issues significantly increase productivity. After running an initial compilation with default settings, you can review the Design Assistant reports for initial analysis. When enabled, Design Assistant automatically reports any violations against a standard set of Intel FPGA-recommended design guidelines.

You can run Design Assistant in Compilation Flow mode, allowing you to view the violations relevant for the compilation stages you run. Alternatively, Design Assistant is available in analysis mode in the Timing Analyzer and Chip Planner.

- Compilation Flow Mode—runs automatically during one or more stages of compilation. In this mode, Design Assistant utilizes in-flow (transient) data during compilation.
- Analysis Mode—run Design Assistant from Timing Analyzer and Chip Planner to analyze design violations at a specific compilation stage, before moving forward in the compilation flow. In analysis mode, Design Assistant uses static compilation snapshot data.

Design Assistant designates each rule violation with one of the following severity levels. You can specify which rules you want the Design Assistant to check in your design, and customize the severity levels, thus eliminating rule checks that are not important for your design.

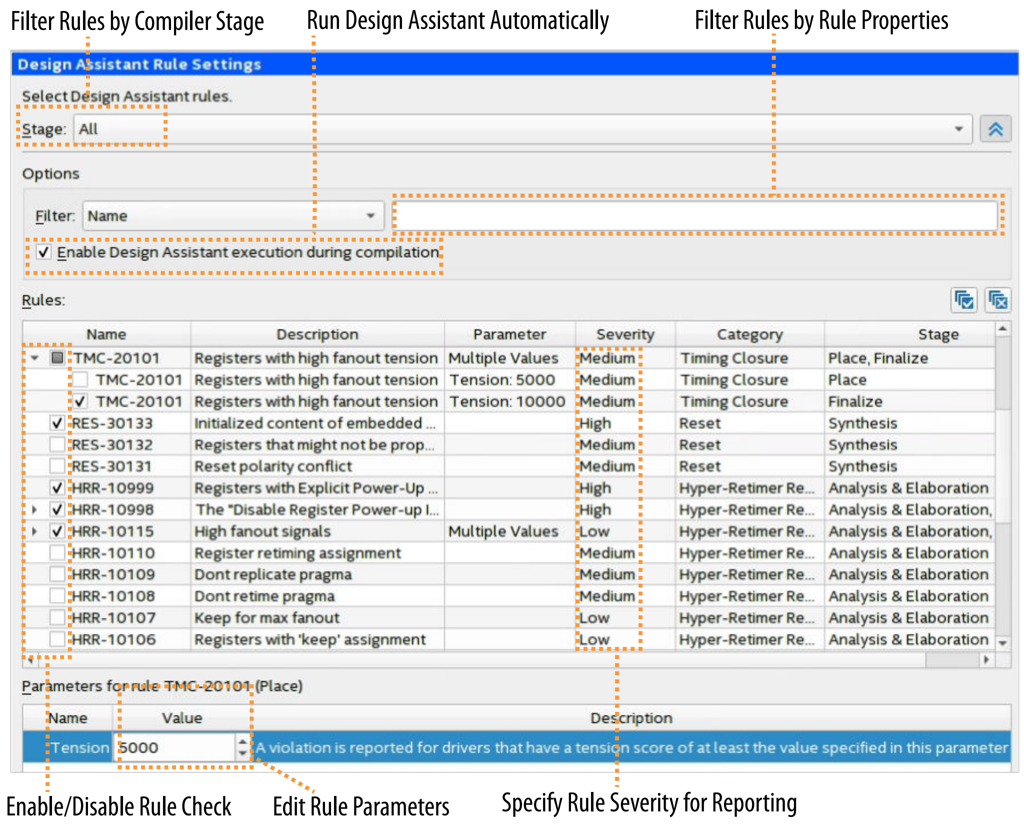
Table 2. Design Assistant Rule Severity Levels

Categories	Description	Severity Level Color
Critical	Address issue for hand-off.	Red
High	Potentially causes functional failure. May indicate missing or incorrect design data.	Orange
Medium	Potentially impacts quality of results for f_{MAX} or resource utilization.	Brown
Low	Rule reflects best practices for RTL coding guidelines.	Blue

Setting Up Design Assistant

You can fully customize the Design Assistant for your individual design characteristics and reporting requirements. Click **Assignments ➤ Settings ➤ Design Assistant Rule Settings** to specify options that control which rules and parameters apply to the various stages of design compilation for design rule checking.

Figure 2. Design Assistant Rule Settings


Filter Rules by Compiler Stage
Run Design Assistant Automatically
Filter Rules by Rule Properties
Design Assistant Rule Settings
Enable/Disable Rule Check
Edit Rule Parameters
Specify Rule Severity for Reporting

Select Design Assistant rules.

Stage: All

Options

Filter: Name

✓ Enable Design Assistant execution during compilation

Rules:

	Name	Description	Parameter	Severity	Category	Stage
▪	TMC-20101	Registers with high fanout tension	Multiple Values	Medium	Timing Closure	Place, Finalize
	TMC-20101	Registers with high fanout tension	Tension: 5000	Medium	Timing Closure	Place
✓	TMC-20101	Registers with high fanout tension	Tension: 10000	Medium	Timing Closure	Finalize
✓	RES-30133	Initialized content of embedded ...		High	Reset	Synthesis
	RES-30132	Registers that might not be prop...		Medium	Reset	Synthesis
	RES-30131	Reset polarity conflict		Medium	Reset	Synthesis
✓	HRR-10999	Registers with Explicit Power-Up ...		High	Hyper-Retimer Re...	Analysis & Elaboration
✓	HRR-10998	The "Disable Register Power-up I...		High	Hyper-Retimer Re...	Analysis & Elaboration,
✓	HRR-10115	High fanout signals	Multiple Values	Low	Hyper-Retimer Re...	Analysis & Elaboration,
	HRR-10110	Register retiming assignment		Medium	Hyper-Retimer Re...	Analysis & Elaboration
	HRR-10109	Dont replicate pragma		Medium	Hyper-Retimer Re...	Analysis & Elaboration
	HRR-10108	Dont retime pragma		Medium	Hyper-Retimer Re...	Analysis & Elaboration
	HRR-10107	Keep for max fanout		Low	Hyper-Retimer Re...	Analysis & Elaboration
	HRR-10106	Registers with 'keep' assignment		Low	Hyper-Retimer Re...	Analysis & Elaboration

Parameters for rule TMC-20101 (Place)

Name	Value	Description
Tension	5000	A violation is reported for drivers that have a tension score of at least the value specified in this parameter

Running Design Assistant

When enabled, the Design Assistant runs automatically during compilation and reports enabled design rule violations in the Compilation Report. Alternatively, you can run Design Assistant in Analysis Mode on a specific compilation snapshot to focus analysis on only that stage.

To enable automated Design Assistant checking during compilation:

- Turn on **Enable Design Assistant execution during compilation** in the **Design Assistant Rule Settings**.

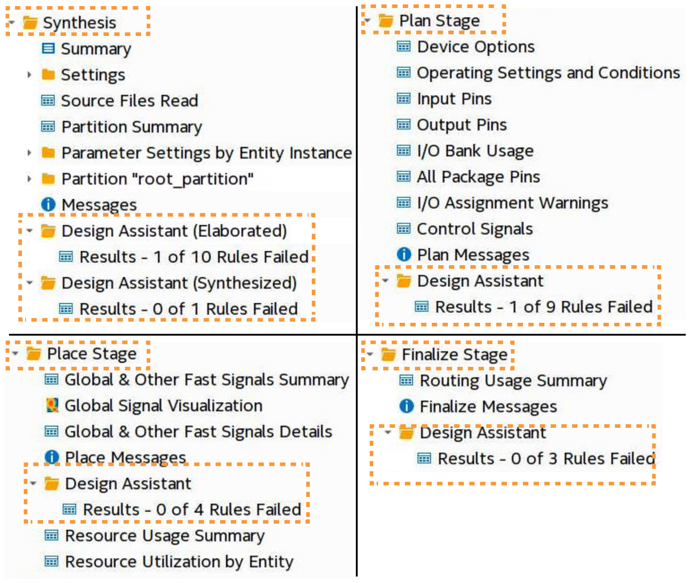
To run Design Assistant in analysis mode to validate a specific snapshot against any design rules that apply to the snapshot:

- Click **Report DRC** in the Timing Analyzer or Chip Planner **Tasks** panel.

Viewing and Correcting Design Assistant Results

The Design Assistant reports enabled design rule violations in the various stages of the Compilation Report.

Figure 3. Design Assistant Results in Synthesis, Plan, Place, and Finalize Reports



Synthesis
- Summary
- Settings
- Source Files Read
- Partition Summary
- Parameter Settings by Entity Instance
- Partition "root_partition"
- Messages
- Design Assistant (Elaborated)
  - Results - 1 of 10 Rules Failed
- Design Assistant (Synthesized)
  - Results - 0 of 1 Rules Failed

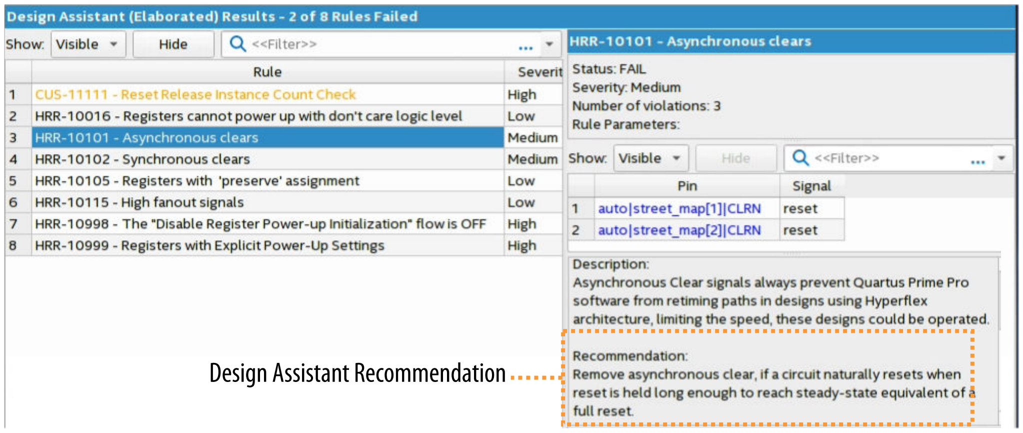
Plan Stage
- Device Options
- Operating Settings and Conditions
- Input Pins
- Output Pins
- I/O Bank Usage
- All Package Pins
- I/O Assignment Warnings
- Control Signals
- Plan Messages
- Design Assistant
  - Results - 1 of 9 Rules Failed

Place Stage
- Global & Other Fast Signals Summary
- Global Signal Visualization
- Global & Other Fast Signals Details
- Place Messages
- Design Assistant
  - Results - 0 of 4 Rules Failed
- Resource Usage Summary
- Resource Utilization by Entity

Finalize Stage
- Routing Usage Summary
- Finalize Messages
- Design Assistant
  - Results - 0 of 3 Rules Failed

To view the results for each rule, click the rule in the **Rules** list. A description of the rule and design recommendations for correction appear.

Figure 4. Design Assistant Rule Violation Recommendation



Design Assistant (Elaborated) Results - 2 of 8 Rules Failed

Show: Visible | Hide | <<Filter>>

	Rule	Severit
1	CUS-11111 - Reset Release Instance Count Check	High
2	HRR-10016 - Registers cannot power up with don't care logic level	Low
3	HRR-10101 - Asynchronous clears	Medium
4	HRR-10102 - Synchronous clears	Medium
5	HRR-10105 - Registers with 'preserve' assignment	Low
6	HRR-10115 - High fanout signals	Low
7	HRR-10998 - The "Disable Register Power-up Initialization" flow is OFF	High
8	HRR-10999 - Registers with Explicit Power-Up Settings	High

HRR-10101 - Asynchronous clears

Status: FAIL
Severity: Medium
Number of violations: 3
Rule Parameters:

Show: Visible | Hide | <<Filter>>

	Pin	Signal
1	auto|street_map[1]|CLRN	reset
2	auto|street_map[2]|CLRN	reset

Description:
Asynchronous Clear signals always prevent Quartus Prime Pro software from retiming paths in designs using Hyperflex architecture, limiting the speed, these designs could be operated.

Recommendation:
Remove asynchronous clear, if a circuit naturally resets when reset is held long enough to reach steady-state equivalent of a full reset.

Modify your RTL to correct the design rule violations.

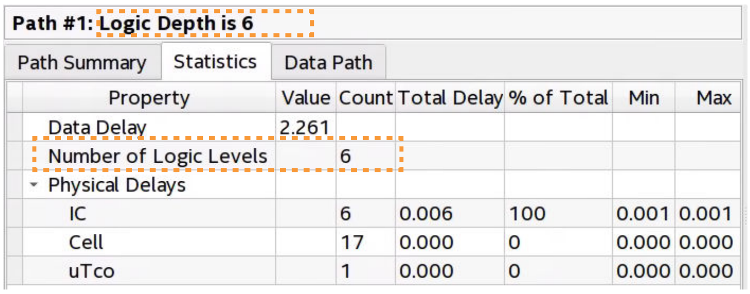
1.1.2. Reduce Logic Levels

Excessive logic levels can impact the Fitter's quality of results because the design critical path influences Fitter processing order and duration.

The Fitter places and routes the design based on timing slack. The Fitter places longer paths with the least slack first. The Fitter generally prioritizes higher logic-level paths over lower-logic level paths. Typically, after the Fitter stage is complete, the critical paths remaining are not the highest logic level paths. The Fitter gives preferred placement, routing, and retiming to higher level logic. Reducing the logic level helps to ensure that all elements of the design receive the same Fitter priority.

Run **Reports ➤ Custom Reports ➤ Report Timing** in the Timing Analyzer to generate reports showing the levels of logic in the path. If the path fails timing and the number of logic levels is high, consider adding pipelining in that part of the design to improve performance.

Figure 5. Logic Depth in Path Report



Path #1: Logic Depth is 6

Path Summary | Statistics | Data Path

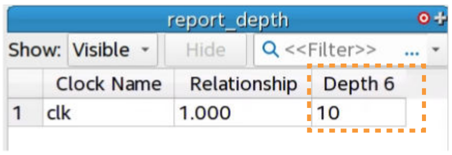
Property	Value	Count	Total Delay	% of Total	Min	Max
Data Delay	2.261					
Number of Logic Levels		6				
Physical Delays						
IC		6	0.006	100	0.001	0.001
Cell		17	0.000	0	0.000	0.000
uTco		1	0.000	0	0.000	0.000

Reporting Logic Level Depth

After the Compiler's Plan stage, you can run `report_logic_depth` in the Timing Analyzer Tcl console to view the number of logic levels within a clock domain. `report_logic_depth` shows the distribution of logic depth among the critical paths, allowing you to identify areas where you can reduce logic levels in your RTL.

```
report_logic_depth -panel_name <name> -from [get_clocks <name>] \
     -to [get_clocks <name>]
```

Figure 6. **report_logic_depth Output**



	Clock Name	Relationship	Depth 6
1	clk	1.000	10

To obtain data for optimizing RTL, run `report_logic_depth` after the Compiler's Plan stage, before running remaining Fitter stages. Otherwise, the post-Fitter reports also include results from physical optimization (retiming and resynthesis).

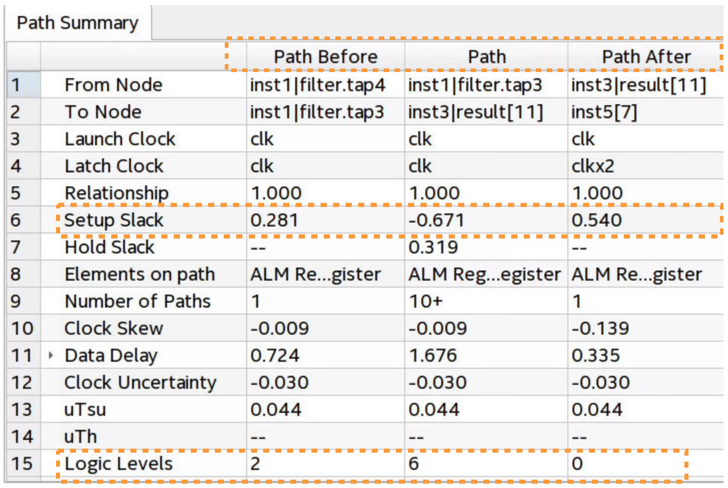
Reporting Neighbor Paths

After running the Fitter (Finalize) stage, you can run `report_neighbor_paths` to help determine the root cause of the critical path (for example, high logic level, retiming limitation, sub-optimal placement, I/O column crossing, hold-fix, or others):

```
report_neighbor_paths -to_clock <name> -npaths <number> -panel_name <name>
```

`report_neighbor_paths` reports the most timing-critical paths in the design, including associated slack, additional path summary information, and path bounding boxes.

Figure 8. **`report_neighbor_paths` Output**



		Path Before	Path	Path After
1	From Node	inst1|filter.tap4	inst1|filter.tap3	inst3|result[11]
2	To Node	inst1|filter.tap3	inst3|result[11]	inst5[7]
3	Launch Clock	clk	clk	clk
4	Latch Clock	clk	clk	clkx2
5	Relationship	1.000	1.000	1.000
6	Setup Slack	0.281	-0.671	0.540
7	Hold Slack	--	0.319	--
8	Elements on path	ALM Re...gister	ALM Reg...egister	ALM Re...gister
9	Number of Paths	1	10+	1
10	Clock Skew	-0.009	-0.009	-0.139
11	Data Delay	0.724	1.676	0.335
12	Clock Uncertainty	-0.030	-0.030	-0.030
13	uTsu	0.044	0.044	0.044
14	uTh	--	--	--
15	Logic Levels	2	6	0

`report_neighbor_paths` shows the most timing-critical **Path Before** and **Path After** each critical **Path**. Retiming or logic balancing of the path can simplify timing closure if there is negative slack on the **Path**, but positive slack on the **Path Before** or **Path After**.

To enable retiming, make sure the following options are turned on:

- For Registers—enable **Assignments ➤ Settings ➤ Compiler Settings ➤ Register Optimization ➤ Allow Register Retiming**
- For RAM Endpoints—enable **Assignments ➤ Settings ➤ Compiler Settings ➤ Fitter Settings (Advanced) ➤ Allow RAM Retiming**
- For DSP Endpoints—enable **Assignments ➤ Settings ➤ Compiler Settings ➤ Fitter Settings (Advanced) ➤ Allow DSP Retiming**

If further logic balancing is required, you must manually modify your RTL to move logic from the critical **Path** to the **Path Before** or **Path After**.

Note: If a register's output is connected to its input, one or both of the neighbor paths may be identical to the current path. When looking for neighbor paths with the worst slack, all operating conditions are considered, not just the operating conditions of the main path itself.

Visualizing Logic Levels in Technology Map Viewer

The Technology Map Viewer also provides schematic, technology-mapped, representations of the design netlist, and can help you see which areas in a design can benefit from reducing the number of logic levels. You can also investigate the physical layout of a path in detail in the Chip Planner.

To locate a timing path in one of the viewers, right-click a path in the timing report, point to **Locate Path**, and select **Locate in Technology Map Viewer**.

1.1.3. Reduce High Fan-Out Nets

High fan-out nets can cause resource congestion, thereby complicating timing closure. In general, the Compiler automatically manages high fan-out nets related to clocks. The Compiler automatically promotes recognized high fan-out nets to the global clock network. The Compiler makes a higher optimization effort during the Place and Route stages, which results in beneficial register duplication.

In the following corner cases, you can additionally reduce congestion by making the following manual changes to your design RTL:

Table 3. High Fan-Out Net Corner Cases

Design Characteristic	Manual RTL Optimization
High fan-out nets that reach many hierarchies or physically far destinations	Specify the `duplicate_hierarchy_depth` assignment on the last register in a pipeline to manually duplicate high fan-out networks across hierarchies. Specify the `duplicate_register` assignment to duplicate registers during placement.
Designs with control signals to DSP or M20K memory blocks from combinational logic	Drive the control signal to the DSP or M20K memory from a register.

Register Duplication Across Hierarchies

You can specify the `duplicate_hierarchy_depth` assignment on the last register in a pipeline to guide the creation of register duplication and fan-outs. The following figures illustrate the impact of the following `duplicate_hierarchy_depth` assignment:

```
set_instance_assignment -name duplicate_hierarchy_depth -to \
     <register_name> <level_number>
```

Where:

- `register_name`—the last register in a chain that fans out to multiple hierarchies.
- `level_number`—the number of registers in the chain to duplicate.

Figure 9. Before Register Duplication

Set the `duplicate_hierarchy_depth` assignment to implement register duplication across hierarchies, and create a tree of registers following the last register in the chain. You specify the register name and the number of duplicates represented by *M* in the following example. Red arrows show the potential locations of duplicate registers.

```
set_instance_assignment –name DUPLICATE_HIERARCHY_DEPTH –to regZ M
```

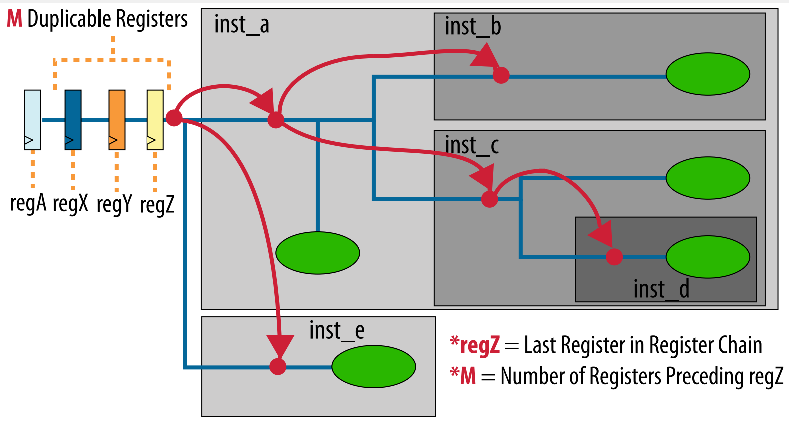

M Duplicable Registers
inst_a
inst_b
inst_c
inst_d
inst_e
regA regX regY regZ
*regZ = Last Register in Register Chain
*M = Number of Registers Preceding regZ

Figure 10. Register Duplication = 1

Specifying the following single level of register duplication (M=1) duplicates one register (`regZ`) down one level of the design hierarchy:

```
set_instance_assignment –name DUPLICATE_HIERARCHY_DEPTH –to regZ 1
```

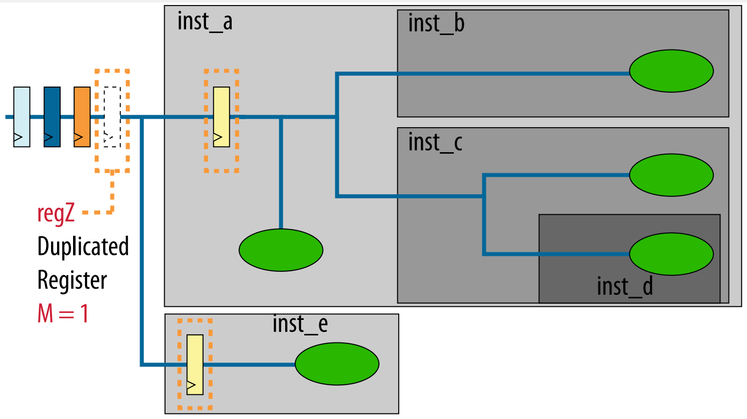
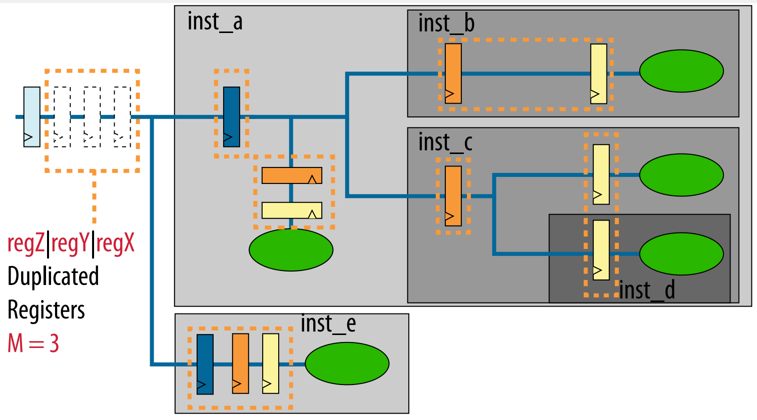


Figure 11. Register Duplication = 3

Specifying three levels of register duplication (M=3) duplicates three registers (`regZ, regY, regX`) down three, two, and one level of the hierarchy, respectively:

```
set_instance_assignment -name DUPLICATE_HIERARCHY_DEPTH -to regZ 3
```

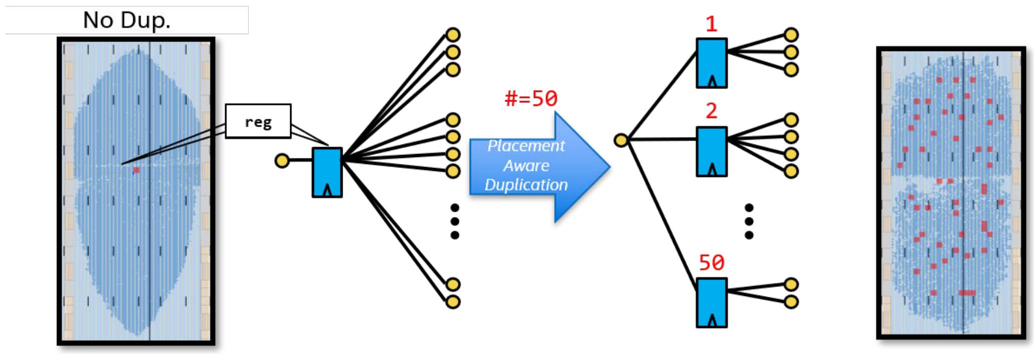


By duplicating and pushing the registers down into the hierarchies, the design retains the same number of cycles to all the destinations, while greatly accelerating performance on these paths.

Register Duplication During Placement

Figure 12 on page 11 shows a register with high fan-out to a widely spread area of the chip. By duplicating this register 50 times, you can reduce the distance between the register and the destinations that ultimately result in faster clock performance. Assigning `duplicate_register` allows the Compiler to leverage physical proximity to guide the placement of new registers feeding a subset of fan-outs.

Figure 12. Register Duplication During Placement



Note: To broadcast a signal across the chip, use a multistage pipeline. Apply the `duplicate_register` assignment to each of the registers in the pipeline. This technique creates a tree structure that broadcasts the signal across the chip.

Viewing Duplication Results

Following design synthesis, view duplication results in the **Hierarchical Tree Duplication Summary** report in the **Synthesis** folder of the Compilation Report. The report provides the following:

- Information on the registers that have the `duplicate_hierarchy_depth` assignment.
- Reason for the chain length that you can use as a starting point for further improvements with the assignment.
- Information about the individual registers in the chain that you can use to better understand the structure of the implemented duplicates.

The Fitter report also includes a section on registers that have the `duplicate_register` setting.

1.2. Step 2: Apply Compiler Optimization Techniques

Designs that utilize a very high percentage of FPGA device resources can cause resource congestion, resulting in lower f_{MAX} and more complex timing closure.

The Compiler's **Optimization Mode** settings allow you specify the focus of Compiler efforts during synthesis. For example, you optimize synthesis for **Area**, or **Routability** when addressing resource congestion. You can experiment with combinations of these same **Optimization Mode** settings in the Intel Quartus Prime Design Space Explorer II. These settings and other manual techniques can help you to reduce congestion in highly utilized designs.

Timing Closure Problem

- Designs with very high device resource utilization complicate timing closure.

Timing Closure Solutions

- Apply Compiler Optimization Modes and Strategies on page 13—specify the primary optimization mode goal for design synthesis.
- Experiment with Area and Routability Options on page 16—apply additional collections of settings to reduce congestion and meet area and routability goals.
- Consider Fractal Synthesis for Arithmetic-Intensive Designs on page 16—For high-throughput, arithmetic-intensive designs, fractal synthesis reduces device resource usage through multiplier regularization, retiming, and continuous arithmetic packing.

Related Information

- "Timing Closure and Optimization" Chapter, Intel Quartus Prime Pro Edition User Guide: Design Optimization
- Intel Quartus Prime Pro Edition User Guide: Design Compilation

1.2.1. Apply Compiler Optimization Modes and Strategies

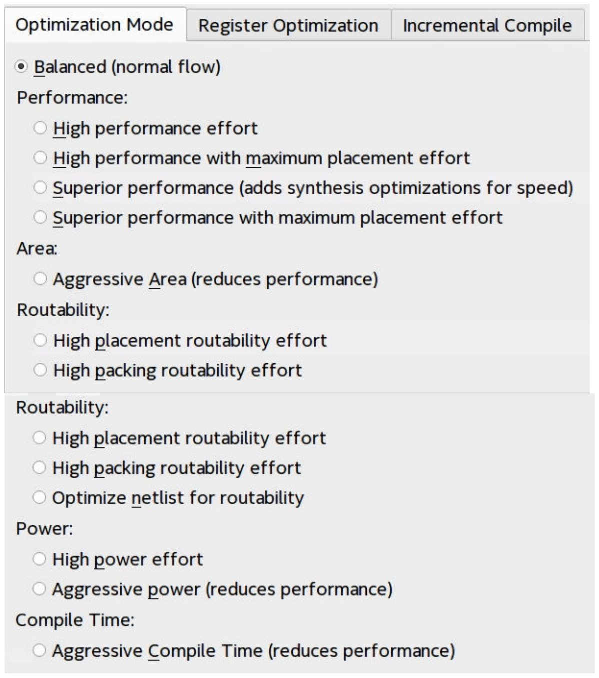
Use the following information to apply Compiler optimization modes and Design Space Explorer II (DSE II) compilation strategies.

Experiment with Compiler Optimization Mode Settings

Follow these steps to experiment with Compiler optimization mode settings:

1. Create or open an Intel Quartus Prime project.
2. To specify the Compiler's high-level optimization strategy, click **Assignments ➤ Settings ➤ Compiler Settings**. Experiment with any of the following mode settings, as Table 4 on page 14 describes.
3. To compile the design with these settings, click **Start Compilation** on the Compilation Dashboard.
4. View the compilation results in the Compilation Report.
5. Click **Tools ➤ Timing Analyzer** to view the results of optimization settings on performance.

Figure 13. Compiler Optimization Mode Settings



(●) Balanced (normal flow)

Performance:

- ○ High performance effort
- ○ High performance with maximum placement 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

Routability:

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

Power:

- ○ High power effort
- ○ Aggressive power (reduces performance)

Compile Time:

- ○ Aggressive Compile Time (reduces performance)

Table 4. Optimization Modes (Compiler Settings Page)

Optimization Mode	Description
Balanced (normal flow)	The Compiler optimizes synthesis for balanced implementation that respects timing constraints.
High Performance Effort	The Compiler increases the timing optimization effort during placement and routing, and enables timing-related Physical Synthesis optimizations (per register optimization settings). Each additional optimization can increase compilation time.
High Performance with Maximum Placement Effort	Enables the same Compiler optimizations as **High Performance Effort**, with additional placement optimization effort.
Superior Performance	Enables the same Compiler optimizations as **High Performance Effort**, and adds more optimizations during Analysis & Synthesis to maximize design performance with a potential increase to logic area. If design utilization is already very high, this option may lead to difficulty in fitting, which can also negatively affect overall optimization quality.
Superior Performance with Maximum Placement Effort	Enables the same Compiler optimizations as **Superior Performance**, with additional placement optimization effort.
Aggressive Area	The Compiler makes aggressive effort to reduce the device area required to implement the design at the potential expense of design performance.
High Placement Routability Effort	The Compiler makes high effort to route the design at the potential expense of design area, performance, and compilation time. The Compiler spends additional time reducing routing utilization, which can improve routability and also saves dynamic power.
High Packing Routability Effort	The Compiler makes high effort to route the design at the potential expense of design area, performance, and compilation time. The Compiler spends additional time packing registers, which can improve routability and also saves dynamic power.
Optimize Netlist for Routability	The Compiler implements netlist modifications to increase routability at the possible expense of performance.

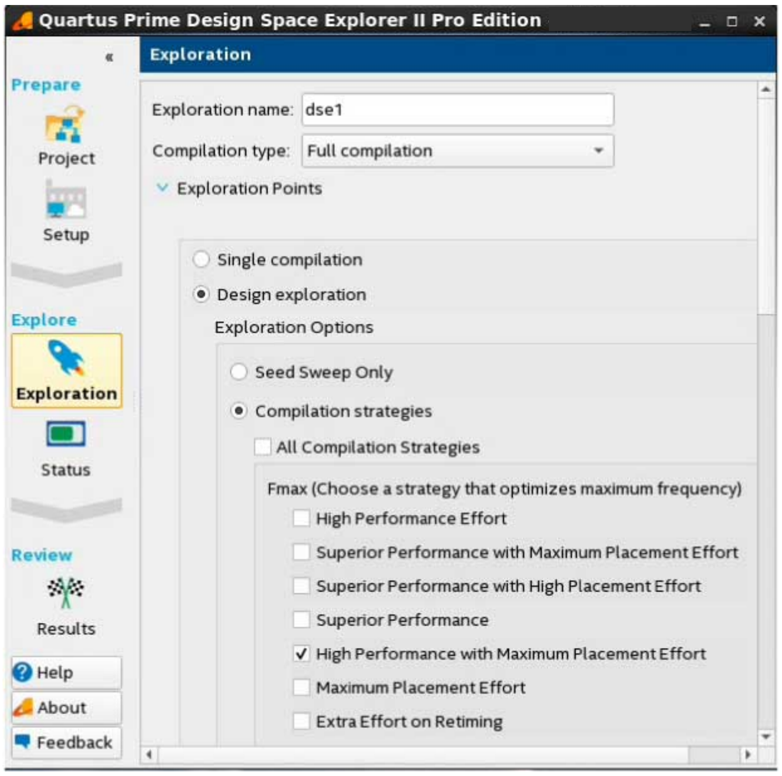
continued...

Optimization Mode	Description
High Power Effort	The Compiler makes high effort to optimize synthesis for low power. **High Power Effort** increases synthesis run time.
Aggressive Power	Makes aggressive effort to optimize synthesis for low power. The Compiler further reduces the routing usage of signals with the highest specified or estimated toggle rates, saving additional dynamic power but potentially affecting performance.
Aggressive Compile Time	Reduces the compile time required to implement the design with reduced effort and fewer performance optimizations. This option also disables some detailed reporting functions. *Note:* Turning on **Aggressive Compile Time** enables Intel Quartus Prime Settings File (`.qsf`) settings which cannot be overridden by other `.qsf` settings.

Design Space Explorer II Compilation Strategies

DSE II allows you to find optimal project settings for resource, performance, or power optimization goals. DSE II allows you to iteratively compile a design using different preset combinations of settings and constraints to achieve a specific goal. DSE II then reports the best settings combination to meet your goals. DSE II can also take advantage of parallelization abilities to compile seeds on multiple computers. DSE II **Compilation Strategy** settings echo the **Optimization Mode** settings in Table 4 on page 14

Figure 14. Design Space Explorer II



Exploration

Exploration name: dse1

Compilation type: Full compilation

Exploration Points

( ) Single compilation

(●) Design exploration

Exploration Options

( ) Seed Sweep Only

(●) Compilation strategies

☐ All Compilation Strategies

Fmax (Choose a strategy that optimizes maximum frequency)

☐ High Performance Effort

☐ Superior Performance with Maximum Placement Effort

☐ Superior Performance with High Placement Effort

☐ Superior Performance

✓ High Performance with Maximum Placement Effort

☐ Maximum Placement Effort

☐ Extra Effort on Retiming

Follow these steps to specify **Compilation Strategy** for DSE II:

1. To launch DSE II (and close the Intel Quartus Prime software), click **Tools ➤ Launch Design Space Explorer II**. DSE II opens after the Intel Quartus Prime software closes.
2. On the DSE II toolbar, click the **Exploration** icon.
3. Expand **Exploration Points**.
4. Select **Design exploration**. Enable any of the **Compilation strategies** to run design explorations targeting those strategies.

1.2.2. Reduce Congestion for High Utilization

Designs that utilize over 80% of device resources typically present the most difficulty in timing closure.

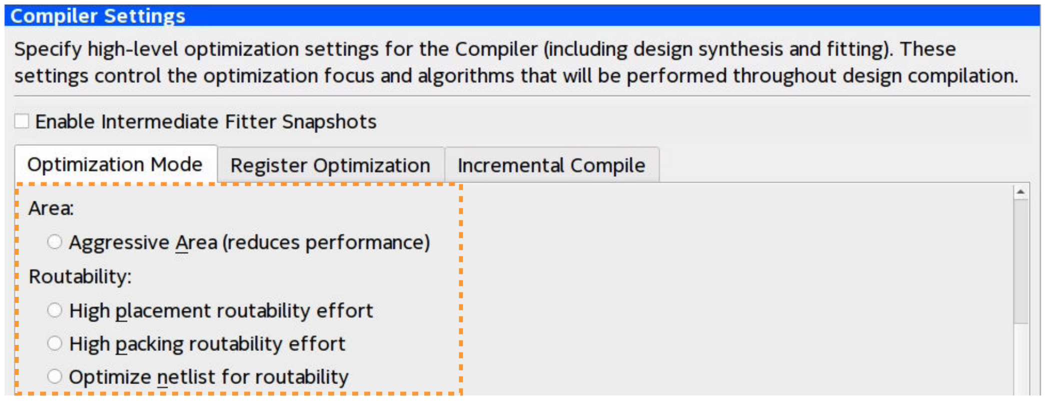
You can apply the following manual and automated techniques to further reduce congestion and simplify timing closure.

- Experiment with Area and Routability Options on page 16
- Consider Fractal Synthesis for Arithmetic-Intensive Designs on page 16

1.2.2.1. Experiment with Area and Routability Options

When device utilization causes routing congestion, you can experiment with the **Area** and **Routability** optimization settings to reduce resource utilization and congestion for your design. Click **Assignments ➤ Settings ➤ Compiler Settings ➤ Optimization Mode** to access these settings:

Figure 15. Area and Routability Options



Compiler Settings

Specify high-level optimization settings for the Compiler (including design synthesis and fitting). These settings control the optimization focus and algorithms that will be performed throughout design compilation.

☐ Enable Intermediate Fitter Snapshots

Area:
○ Aggressive Area (reduces performance)
Routability:
○ High placement routability effort
○ High packing routability effort
○ Optimize netlist for routability

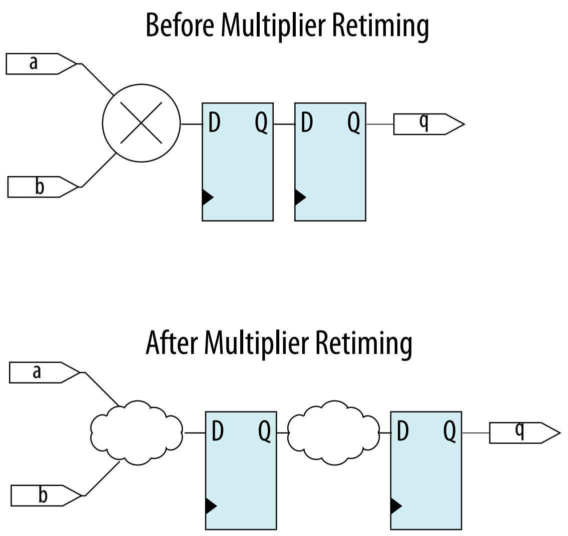
1.2.2.2. Consider Fractal Synthesis for Arithmetic-Intensive Designs

For high-throughput, arithmetic-intensive designs, you can enable automatic fractal synthesis optimizations to improve use of device resources. Fractal synthesis optimizations include multiplier regularization and retiming, as well as continuous arithmetic packing. The optimizations target designs with large numbers of low-precision arithmetic operations (such as additions and multiplications). You can enable fractal synthesis globally or for only specific multipliers. Under ideal conditions, fractal synthesis optimization can achieve 20-45% area reduction.

Multiplier Regularization and Retiming

Multiplier regularization and retiming performs inference of highly optimized soft multiplier implementations. The Compiler may apply backward retiming to two or more pipeline stages if required. When you enable fractal synthesis, the Compiler applies multiplier regularization and retiming to signed and unsigned multipliers.

Figure 16. Multiplier Retiming



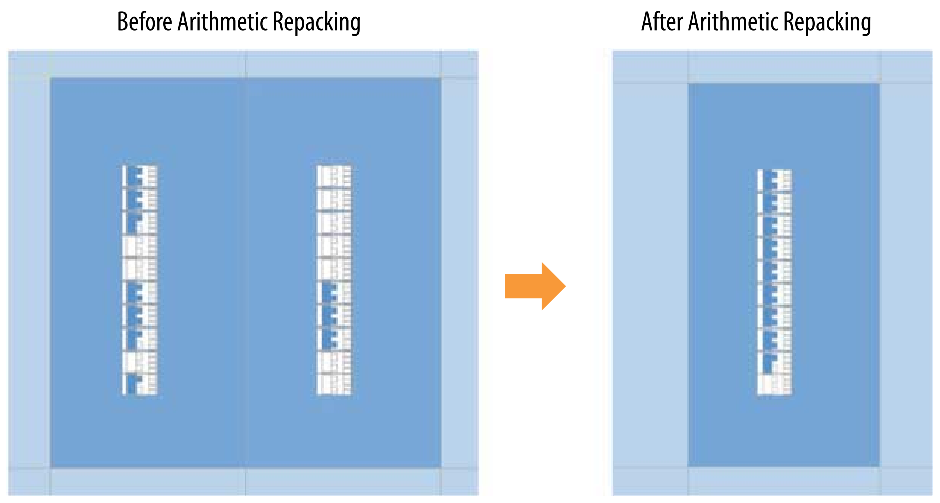
Note:
- Multiplier regularization uses only logic resources, and does not use DSP blocks.
- Multiplier regularization and retiming is applied to both signed and unsigned multipliers in modules where the `FRACTAL_SYNTHESIS` QSF assignment is set.

Continuous Arithmetic Packing

Continuous arithmetic packing re-synthesizes arithmetic gates into logic blocks optimally sized to fit into Intel FPGA LABs. This optimization allows up to 100% utilization of LAB resources for the arithmetic blocks.

When you enable fractal synthesis, the Compiler applies this optimization to all carry chains and two-input logic gates. This optimization can pack adder trees, multipliers, and any other arithmetic-related logic.

Figure 17. Continuous Arithmetic Packing



Note that continuous arithmetic packing works independently of multiplier regularization. So, if you are using a multiplier that is not regularized (such as writing your own multiplier) then continuous arithmetic packing can still operate.

Note: Fractal synthesis optimization is most suitable for designs with deep-learning accelerators or other high-throughput, arithmetic-intensive functions that exceed all DSP resources. Enabling fractal synthesis project-wide can cause unnecessary bloat on modules that are not suitable for fractal optimizations.

1.2.2.2.1. Enabling or Disabling Fractal Synthesis

For Intel Stratix® 10 and Intel Agilex™ devices, fractal synthesis optimization runs automatically for small multipliers (any A*B statement in Verilog HDL or VHDL where bit-width of the operands is 7 or less). You can also disable automatic fractal synthesis for small multipliers for these devices using either of the following methods:

- In RTL, set the DSP multstyle, as "Multstyle Verilog HDL Synthesis Attribute" describes. For example:

```
(* multstyle = "dsp" *) module foo(...);
module foo(..) /* synthesis multstyle = "dsp" */;
```

- In the `.qsf` file, add as an assignment as follows:

```
set_instance_assignment -name DSP_BLOCK_BALANCING_IMPLEMENTATION \
     DSP_BLOCKS -to r
```

In addition, for Intel Stratix 10, Intel Agilex, Intel Arria® 10, and Intel Cyclone® 10 GX devices, you can enable fractal synthesis globally or for specific multipliers with the **Fractal Synthesis** GUI option or the corresponding FRACTAL_SYNTHESIS `.qsf` assignment:

- In RTL, use `altera_attribute` as follows:

```
(* altera_attribute = "-name FRACTAL_SYNTHESIS ON" *)
```

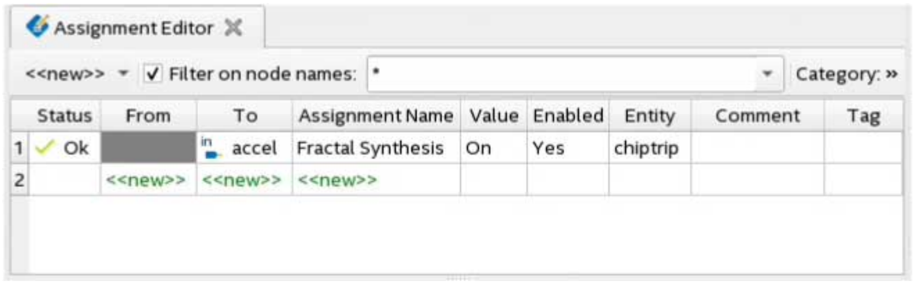
- In the `.qsf` file, add as an assignment as follows:

```
set_global_assignment -name FRACTAL_SYNTHESIS ON -entity <module name>
```

In the user interface, follow these steps:

1. Click **Assignments ➤ Assignment Editor**.
2. Select **Fractal Synthesis** for **Assignment Name**, **On** for the **Value**, the arithmetic-intensive entity name for **Entity**, and an instance name in the **To** column. You can enter a wildcard (*) for **To** to assign all instances of the entity.

Figure 18. Fractal Synthesis Assignment in Assignment Editor



	Status	From	To	Assignment Name	Value	Enabled	Entity	Comment	Tag
1	✓ Ok		accel	Fractal Synthesis	On	Yes	chiptrip		
2		<<new>>	<<new>>	<<new>>					

Related Information

Multstyle Verilog HDL Synthesis Attribute
In Intel Quartus Prime Help.

1.3. Step 3: Preserve Satisfactory Results

You can simplify timing closure by back-annotating satisfactory compilation results to lock down placement of large blocks related to clocks, RAMs, and DSPs.

Similarly, the design block reuse technique enables you to preserve satisfactory compilation results for specific FPGA periphery or core logic design blocks (logic that comprises a hierarchical design instance), and then reuse those blocks in subsequent compilations. In design block reuse, you assign the hierarchical instance as a design partition, and then preserve and export the partition following successful compilation.

Preserving and reusing satisfactory results allows you to focus the Compiler's effort and time on only portions of the design that have not closed timing.

Timing Closure Problem

- Unless locked down, the Compiler may implement design blocks, clocks, RAMs, and DSPs differently from compilation to compilation depending on various factors.

Timing Closure Solutions

- Lock Down Clocks, RAMs, and DSPs on page 20—back-annotate satisfactory compilation results to lock down placement of large blocks related to clocks, RAMs, and DSPs.
- Preserve Design Partition Results on page 21—preserve the partitions for blocks that meet timing, and focus optimization on the other design blocks.

Related Information

- Back-Annotate Assignments Dialog Box Help
- AN-899: Reducing Compile Time with Fast Preservation
- Intel Quartus Prime Pro Edition User Guide: Block-Based Design

1.3.1. Lock Down Clocks, RAMs, and DSPs

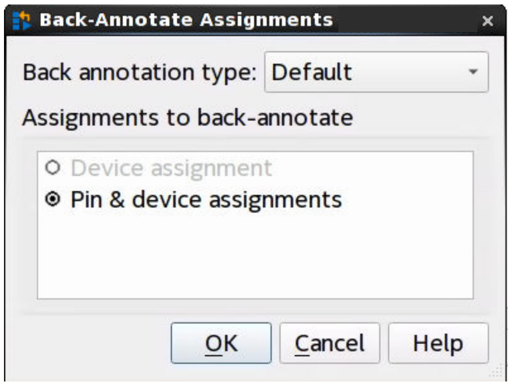
You can simplify timing closure by back-annotating satisfactory compilation results to lock down placement of large blocks related to Clocks, RAMs, and DSPs. Locking down large block placement can produce higher f_{MAX} with less noise.

Locking down large blocks like RAMs and DSPs can be effective because these blocks have heavier connectivity than regular LABs, complicating movement during placement. When a seed produces good results from suitable RAM and DSP placement, you can capture that placement with back-annotation. Subsequent compiles can then benefit from the high quality RAM and DSP placement from the good seed. This technique does not significantly benefit designs with very few RAMs or DSPs.

Click **Assignments ➤ Back-Annotate Assignments** to copy the device resource assignments from the last compilation to the `.qsf` for use in the next compilation. Select the back-annotation type in the **Back-annotation type** list.

Send Feedback

Figure 19. Back-Annotate Assignments Dialog Box


Back-Annotate Assignments
Back annotation type: Default
Assignments to back-annotate
Device assignment
Pin & device assignments
OK
Cancel
Help

Alternatively, you can run back-annotation with the following `quartus_cdb` executable.

```
quartus_cdb <design_name> --back_annotate [--dsp] [--ram] [--clock]
```

Note: The executable supports the additional `[--dsp]`, `[--ram]`, and `[--clock]` variables that the **Back-Annotate Assignments** dialog box does not yet support.

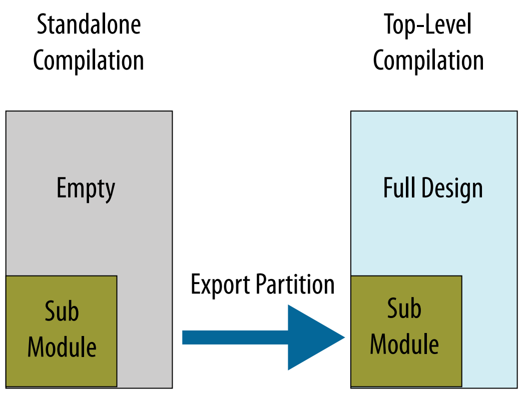
1.3.2. Preserve Design Partition Results

After partitioning the design, you can preserve the partitions for blocks that meet timing, and focus optimization on the other design blocks. In addition, the **Fast Preserve** option simplifies the logic of a preserved partition to only interface logic during compilation, thereby reducing the compilation time for the partition.

Note: **Fast Preserve** only supports root partition reuse and partial reconfiguration designs.

For designs with sub-modules that are challenging for timing closure, you can perform stand-alone optimization and compilation of the module's partition, and then export the timing-closed module to preserve the implementation in subsequent compilations.

Figure 20. Preserving Design Partition Results

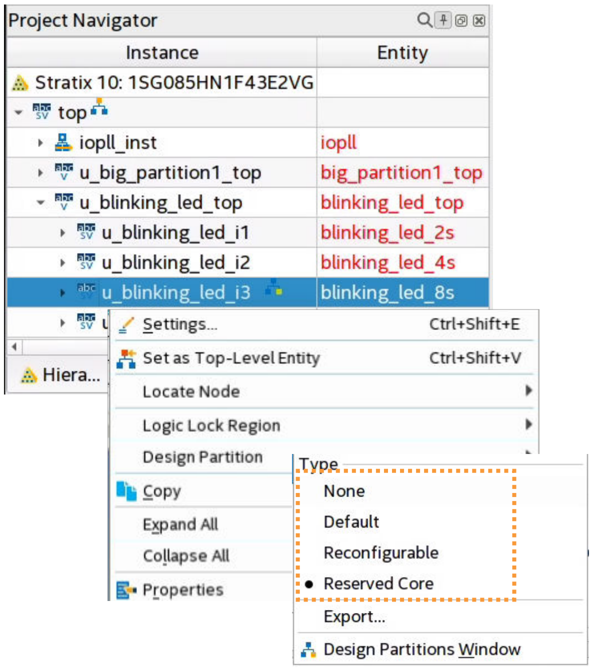


Block-based design requires design partitioning. Design partitioning allows you to preserve individual logic blocks in your design, but can also introduce potential performance loss due to partition crossing and floorplan effects. You need to balance these factors when using block-based design techniques.

The following high level steps describe the partition preservation flow for root partition reuse designs:

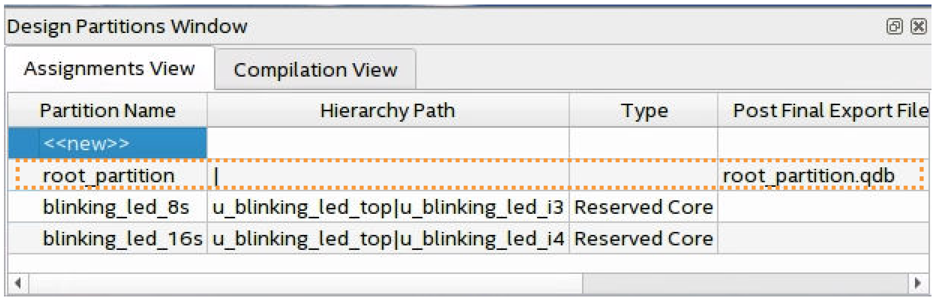
1. Click **Processing ➤ Start ➤ Start Analysis & Elaboration**.
2. In the Project Navigator, right-click the timing closed design instance, point to **Design Partition**, and select a partition **Type**, as Design Partition Settings on page 23 describes.

Figure 21. Create Design Partitions



3. Define Logic Lock floorplanning constraints for the partition. In the Design Partitions Window, right-click the partition and then click **Logic Lock Region ➤ Create New Logic Lock Region**. Ensure that the region is large enough to enclose all logic in the partition.
4. To export the partition results following compilation, in the Design Partitions Window, specify the partition `.qdb` as the **Post Final Export File**.

Figure 22. Post Final Export File

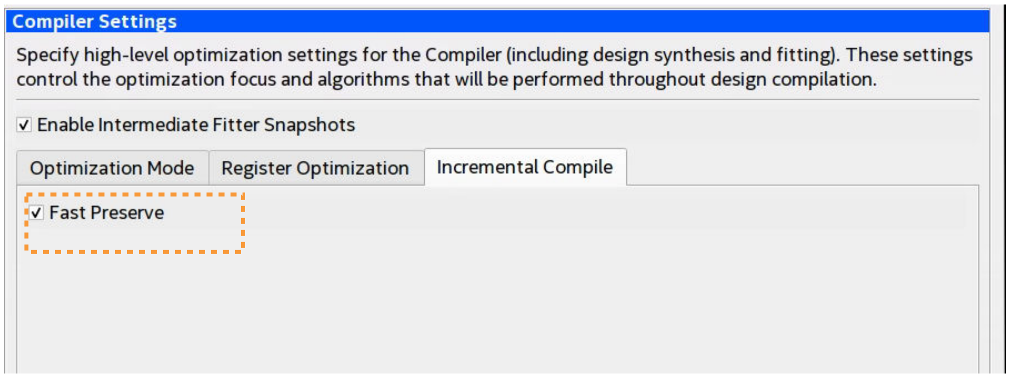

Design Partitions Window

Assignments View | Compilation View

Partition Name	Hierarchy Path	Type	Post Final Export File
<<new>>			
root_partition	|		root_partition.qdb
blinking_led_8s	u_blinking_led_top|u_blinking_led_i3	Reserved Core	
blinking_led_16s	u_blinking_led_top|u_blinking_led_i4	Reserved Core	

5. To compile the design and export the partition, click **Compile Design** on the Compilation Dashboard.
6. Open the top-level project in the Intel Quartus Prime software.
7. Click **Assignments ➤ Settings ➤ Compiler Settings ➤ Incremental Compile**. Turn on the **Fast Preserve** option.

Figure 23. Fast Preserve Option


Compiler Settings

Specify high-level optimization settings for the Compiler (including design synthesis and fitting). These settings control the optimization focus and algorithms that will be performed throughout design compilation.

✓ Enable Intermediate Fitter Snapshots

Optimization Mode | Register Optimization | Incremental Compile

✓ Fast Preserve

8. Click **OK**.
9. In the Design Partitions Window, specify the exported `.qdb` as the **Partition Database File** for the partition in question. This `.qdb` is now the source for this partition in the project. When you enable the **Fast Preserve** option, the Compiler reduces the logic of the imported partition to only interface logic, thereby reducing the compilation time the partition requires.

1.3.2.1. Design Partition Settings

Table 5. Design Partition Settings

Option	Description
Partition Name	Specifies the partition name. Each partition name must be unique and consist of only alphanumeric characters. The Intel Quartus Prime software automatically creates a top-level (|) "root_partition" for each project revision.
Hierarchy Path	Specifies the hierarchy path of the entity instance that you assign to the partition. You specify this value in the **Create New Partition** dialog box. The root partition hierarchy path is |.
Type	Double-click to specify one of the following partition types that control how the Compiler processes and implements the partition:

continued...

Option	Description
	• **Default**—Identifies a standard partition. The Compiler processes the partition using the associated design source files. • **Reconfigurable**—Identifies a reconfigurable partition in a partial reconfiguration flow. Specify the **Reconfigurable** type to preserve synthesis results, while allowing refit of the partition in the PR flow. • **Reserved Core**—Identifies a partition in a block-based design flow that is reserved for core development by a Consumer reusing the device periphery.
Preservation Level	Specifies one of the following preservation levels for the partition: • **Not Set**—specifies no preservation level. The partition compiles from source files. • **synthesized**—the partition compiles using the synthesized snapshot. • **final**—the partition compiles using the final snapshot. With **Preservation Level** of **synthesized** or **final**, changes to the source code do not appear in the synthesis.
Empty	Specifies an empty partition that the Compiler skips. This setting is incompatible with the **Reserved Core** and **Partition Database File** settings for the same partition. The **Preservation Level** must be **Not Set**. An empty partition cannot have any child partitions.
Partition Database File	Specifies a Partition Database File (`.qdb`) that the Compiler uses during compilation of the partition. You export the `.qdb` for the stage of compilation that you want to reuse (synthesized or final). Assign the `.qdb` to a partition to reuse those results in another context.
Entity Re-binding	• PR Flow—specifies the entity that replaces the default persona in each implementation revision. • Root Partition Reuse Flow —specifies the entity that replaces the reserved core logic in the consumer project.
Color	Specifies the color-coding of the partition in the Chip Planner and Design Partition Planner displays.
Post Synthesis Export File	Automatically exports post-synthesis compilation results for the partition to the `.qdb` that you specify, each time Analysis & Synthesis runs. You can automatically export any design partition that does not have a preserved parent partition, including the root_partition.
Post Final Export File	Automatically exports post-final compilation results for the partition to the `.qdb` that you specify, each time the final stage of the Fitter runs. You can automatically export any design partition that does not have a preserved parent partition, including the root_partition.

1.4. AN 903 Document Revision History

This document has the following revision history:

Document Version	Intel Quartus Prime Version	Changes
2020.03.23	19.3.0	Corrected syntax error in code sample in "Lock Down Clocks, RAMs, and DSPs" topic.
2019.12.03	19.3.0	• First public release.