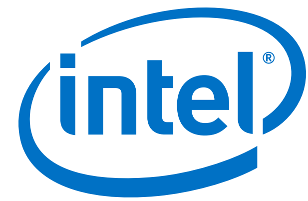
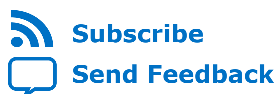


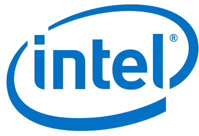
Intel® Quartus® Prime Pro Edition User Guide

Block-Based Design

Updated for Intel® Quartus® Prime Design Suite: **19.4**



UG-20135 | 2019.12.16
Latest document on the web: PDF | HTML



Contents

1. Block-Based Design Flows.. 3
1.1. Block-Based Design Terminology..3
1.2. Block-Based Design Overview..4
1.2.1. Design Block Reuse Overview..5
1.2.2. Incremental Block-Based Compilation Overview.................................... 8
1.3. Design Methodologies Overview...9
1.3.1. Top-Down Design Methodology Overview... 9
1.3.2. Bottom-Up Design Methodology Overview..9
1.3.3. Team-Based Design Methodology Overview... 10
1.4. Design Partitioning..11
1.4.1. Planning Partitions for Periphery IP, Clocks, and PLLs.......................... 13
1.4.2. Design Partition Guidelines...14
1.4.3. Partition Snapshot Preservation and Reuse... 15
1.4.4. Creating Design Partitions.. 16
1.5. Design Block Reuse Flows.. 18
1.5.1. Reusing Core Partitions..18
1.5.2. Reusing Root Partitions..22
1.5.3. Reserved Core Entity Re-Binding...25
1.5.4. Viewing Quartus Database File Information... 26
1.6. Incremental Block-Based Compilation Flow..28
1.6.1. Incremental Timing Closure..28
1.6.2. Design Abstraction.. 30
1.7. Combining Design Block Reuse and Incremental Block-Based Compilation.................. 31
1.8. Setting-Up Team-Based Designs.. 33
1.8.1. Creating a Top-Level Project for a Team-Based Design......................... 33
1.9. Bottom-Up Design Considerations.. 34
1.10. Debugging Block-Based Designs with the Signal Tap Logic Analyzer......................... 35
1.11. Block-Based Design Flows Revision History... 36
1.12. Intel Quartus Prime Pro Edition User Guide: Block-Based Design Document Archive.... 37

A. Intel Quartus Prime Pro Edition User Guides.. 38

1. Block-Based Design Flows

The Intel® Quartus® Prime Pro Edition software supports block-based design flows, also known as modular or hierarchical design flows.

You can designate a design block as a design partition in order to preserve or reuse the block. A design partition is a logical, named, hierarchical boundary assignment that you can apply to a design instance. Block-based design flows enable the following:

- Design block reuse—export and reuse of design blocks in other projects
- Incremental block-based compilation—preservation of design blocks (or logic that comprises a hierarchical design instance) within a project

You can reuse design blocks with the same periphery configuration, share a synthesized design block with another designer, or replicate placed and routed IP in another project. Design, implement, and verify core or periphery blocks once, and then reuse those blocks multiple times across different projects that use the same device.

1.1. Block-Based Design Terminology

This document refers to the following terms to explain block-based design methods:

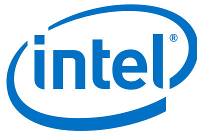
Table 1. Block-Based Design Terminology

Term	Description
Black Box File	RTL source file that contains only port and module or entity definitions, without any logic. Include parameters or generics passed to the module or entity to ensure that the configuration matches the implementation in the Consumer project.
Block	Logic that comprises a hierarchical design instance, typically represented by a Verilog module or VHDL entity. You designate a design block as a design partition to preserve, empty, or export the block.
Consumer	A Consumer can reuse a design partition that a Developer exports as a Partition Database File (`.qdb`) from another project.
Core Partition	A design partition that contains only FPGA resources for the implementation of core logic, such as LUTs, registers, M20K memory blocks, and DSPs. A core partition cannot include periphery resources.
Design Partition	A logical, named, hierarchical boundary assignment that you can apply to a design instance. Creating a partition creates a logical boundary and prevents logic optimization and merging with parent or child partitions. Design partitions facilitate incremental block-based compilation and design block reuse by logically separating instances.
Developer	A Developer creates and exports a design partition as a `.qdb` for use in a Consumer project.

continued...

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered



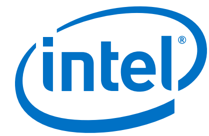
Term	Description
Fast Preserve	The **Fast Preserve** Compiler option simplifies the logic of a preserved partition to only interface logic during compilation, thereby reducing the compilation time for the partition.
Floorplanning	Planning the physical layout of FPGA device resources. The manual process of assigning the logical design hierarchy and periphery to physical regions in the device and I/O.
Logic Lock Region Constraints	Constrains the placement and routing of logic to a specific region in the target device. Specify the region origin, height, width, and any of the following options: • **Reserved**—prevents the Fitter from placing non-member logic within the region. • **Core-Only**—applies the constraint only to core logic in the region, and does not include periphery logic in the region. • **Routing Region**—restricts the routing of connections between region members to the specified area. The routing region is non-exclusive. Other resources in the parent or sibling hierarchy levels can use that routing area. You can restrict the routing region to areas equal to or larger than the Logic Lock region, up to the entire chip. The default routing region is the entire chip. • **Size/State**—fixes the size and locks the state of the region. The **Fixed/Locked** option defines a region of fixed size and locked location. The **Auto/Floating** option defines a region with a floating location that automatically sizes to the logic.
Preservation	The Compiler can preserve a snapshot of compilation results, for each partition, at specific stages of compilation. You can preserve design blocks after synthesis or after the final stage of the Fitter.
Project	The Intel Quartus Prime software organizes the source files, settings, and constraints within a project of one or more revisions. The Intel Quartus Prime Project File (`.qpf`) stores the project name and references each project revision that you create.
Root Partition	The Intel Quartus Prime software automatically creates a top-level "root_partition" with a hierarchy path of | for each project revision. The root partition includes all device periphery resources (such as I/O, HSSIO, memory interfaces, and PCIe*) and associated core resources. You can export and reuse periphery resources by exporting the root partition and reserving a region for subsequent development (the reserved core) by a Consumer.
Snapshot	A snapshot is a view of the design after a compilation stage. The Intel Quartus Prime Compiler generates a snapshot of the compilation database after each compilation stage. You can preserve or export a specific snapshot for incremental block-based compilation, design block reuse, and team based designs.

Related Information

- AN-899: Reducing Compile Time with Fast Preservation
- https://www.intel.com/content/www/us/en/programmable/support/training/course/ofastpres.html

1.2. Block-Based Design Overview

This section provides an overview of design block reuse and incremental block-based compilation flows. Design Block Reuse Flows on page 18 and Incremental Block-Based Compilation Flow on page 28 describe the step-by-step details of these block-based flows.



1.2.1. Design Block Reuse Overview

In design block reuse flows, you export a core or root partition for reuse in another project that targets the same Intel FPGA device family. You can share one of the following compilation snapshots for a partition across projects or with other designers:

- Synthesized snapshot
- Final snapshot

Core partition reuse enables preservation and export of compilation results for a core partition. Reuse of the core partition allows an IP developer to create and optimize an IP once and share it across multiple projects.

Root partition reuse enables preservation and export of compilation results for a top-level (or root) partition that describes the device periphery, along with associated core logic. Reuse of the periphery allows a board developer to create and optimize a platform design with device periphery logic once, and then share that root partition with other board users who create custom core logic. The periphery resources include all the hardened IP in the device periphery, such as general purpose I/O, PLLs, high-speed transceivers, PCIe, and external memory interfaces.

Team members can work on different partitions separately, and then bring them together later, facilitating a team-based design environment. A team lead integrates the partitions in the system and provides guidance to ensure that each partition uses the appropriate device resource and achieves design requirements during the full design integration. A Developer initially creates and exports a block as a partition in one Intel Quartus Prime project. Subsequently, a *Consumer* reuses the partition in a different project.(1) To avoid resource conflicts, floorplanning is essential when reusing final snapshot partitions.

Related Information

Design Block Reuse Flows on page 18

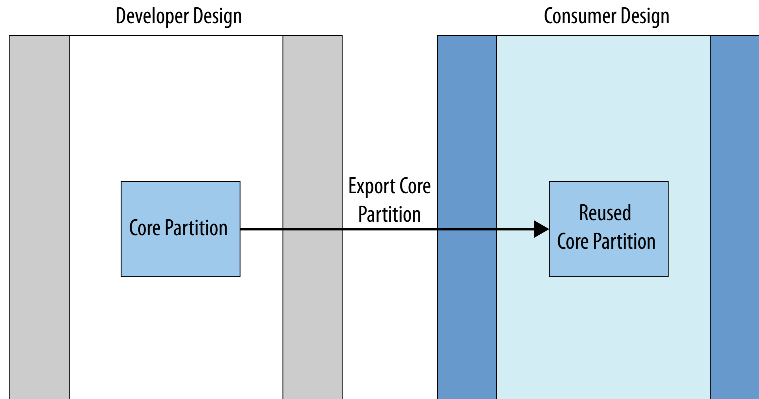
1.2.1.1. Design Block Reuse Examples

Core Partition Reuse Example

In a typical core partition reuse example, a Developer exports a core partition that already meets design requirements. The Developer optimizes and exports the block, and then the Consumer can simply reuse the block without requiring re-optimization in the Consumer project that targets the same device family.

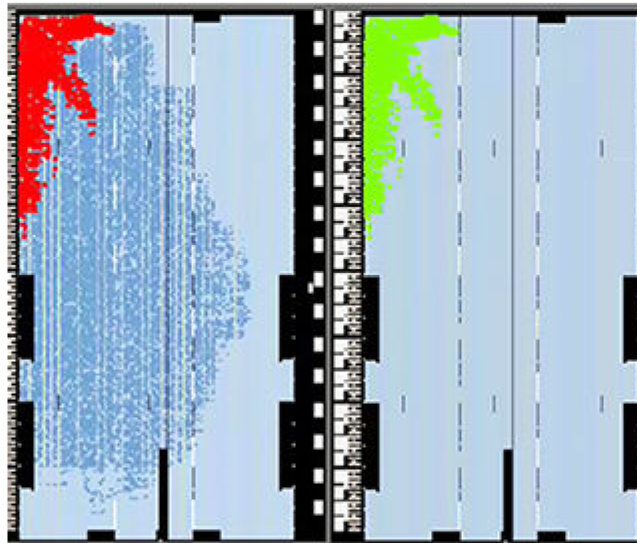
(1) For brevity, this document uses Developer to indicate the person or project that originates a reusable block, and uses Consumer to indicate the person or project that consumes a reusable block.

Figure 1. Core Partition Reuse Example



You can export a core block with unique characteristics that you want to retain, and then replicate that functionality or physical implementation in other projects. In the following figure, a Developer reuses the red-colored partition in the floorplan in another project shown in green in the floorplan on the right.

Figure 2. IP Replication and Physical Implementation



Root Partition Reuse Example

In a typical root partition reuse example, a Developer defines a root partition that includes periphery and core resources that are appropriate for reuse in other projects. An example of this scenario is reuse of the periphery for a development kit that can be reused by multiple Developers and projects.

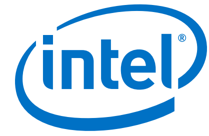
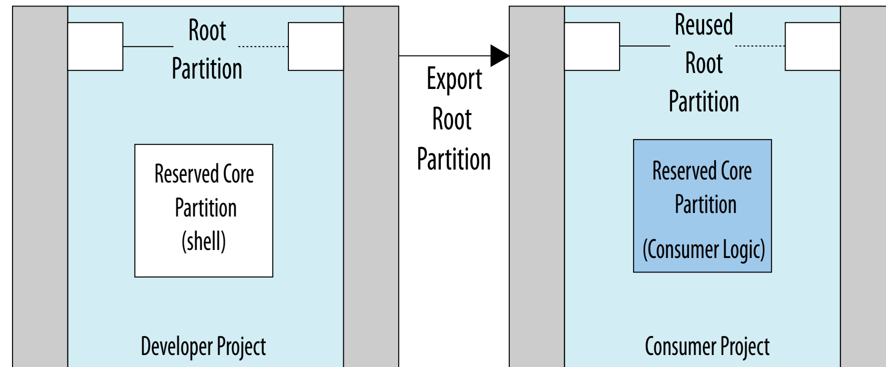
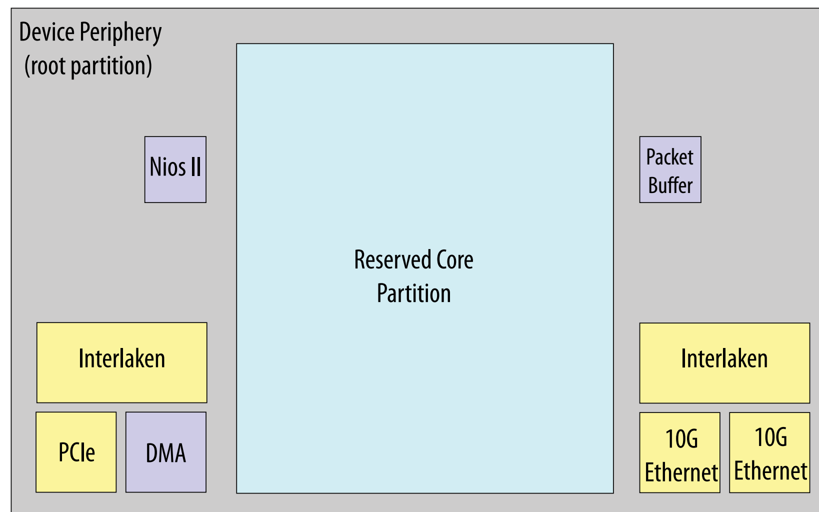


Figure 3. Root Partition Reuse Example



In root partition reuse, each project must target the same Intel FPGA device, must have the same interfaces, and use the same version of the Intel Quartus Prime Pro Edition software. The following example shows reuse of an optimized root partition that contains various periphery interfaces. Only the reserved core partition that contains custom logic changes between Consumer projects.

Figure 4. Root Partition Reuse Example



You can reuse the root partition with multiple other boards. The root partition Developer creates a design that meets expected design requirements with all the required resources, while also reserving a region (the reserved core) for Consumer development. The Developer then exports the root partition as a `.qdb` file and passes the `.qdb` to the Consumers of the partition.

The Consumer reuses the root partition, and adds their own RTL for the reserved core. This flow allows for development on several different boards with a common root partition. Reusing the root partition saves the Consumer development time, because the root partition is pre-optimized by the Developer.

1.2.2. Incremental Block-Based Compilation Overview

You can use incremental block-based compilation to help close timing faster through use of design partition preservation, or to reduce overall compile time through design abstraction with empty design partitions.

You can partition your design and preserve the compilation results for specific design partitions, while at the same time changing and re-compiling the RTL. The Compiler modifies only the non-preserved partitions in the design, while retaining the results of preserved partitions. You can also target optimization techniques to specific design partitions, while leaving other partitions unchanged. This flow can reduce design iterations, improve the predictability of results during iterations, and achieve faster timing closure for teams and individual designers.

You can specify a design partition as *Empty* to represent parts of your design that are incomplete or missing. Setting a partition to **Empty** can reduce the total compilation time if the Compiler does not process design logic associated with the empty partition.

Empty partitions allow you to:

- Set aside incomplete portions of the design, mark them as **Empty**, and complete the design incrementally.
- Designate complete portions of the design as **Empty** to focus all subsequent Compiler efforts on uncompleted portions of the design

Related Information

- Incremental Block-Based Compilation Flow on page 28
- AN 851: Incremental Block-Based Compilation Tutorial

1.2.2.1. Incremental Block-Based Compilation Examples

You can use incremental block-based compilation to optimize a timing-critical partition or in Signal Tap debugging.

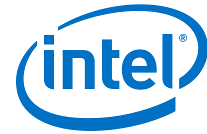
- Optimize the results of one partition, without impacting the results of other design partitions that already meet their requirements.
- Iteratively lock down the performance of one partition, and then move on to optimization of another partition.
- Improve the predictability of results during iterations by preserving the partitions of the design that meet timing.

The following describe typical incremental block-based compilation examples:

Example 1: Optimizing a Timing-critical Partition

After performing a lengthy full compilation of a design with multiple partitions, the Timing Analyzer reports that the clock timing requirement is not met for a specific design partition.

You apply optimization techniques to the specific partition, such as raising the **Placement Effort Multiplier** option value. Because **Placement Effort Multiplier** optimization of the entire design requires significant compilation time, you can apply the optimization only to the partition in question.



Example 2: Design Debugging

After performing some early diagnostics, your design is not functioning as you expect. You decide to debug the design using the Signal Tap logic analyzer, but you want to ensure that adding Signal Tap logic to your design does not negatively affect completed portions of your design.

You preserve the compilation results and add Signal Tap to your design without recompiling the full design from source code. This flow improves the predictability of results during iterations whenever you need to add the logic analyzer to debug your design, or when you want to modify the configuration of the Signal Tap file (`.stp`) without modifying your design logic or placement.

Note: No recompilation of preserved partitions is only possible if the signals that you add for Signal Tap are from the logic you do not plan to preserve. Refer to *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria® 10 FPGA Development Board* for considerations when using Signal Tap with block-based design flows.

Related Information

AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board

1.3. Design Methodologies Overview

Block-based design flows support top-down, bottom-up, and team-based design methodologies. The following sections describe these methods.

1.3.1. Top-Down Design Methodology Overview

In a top-down design methodology, you plan the design at the top level, and then split the design into lower-level design blocks. Different developers or IP providers can create and verify HDL code for the lower-level design blocks separately, but one team lead manages the implementation project for the entire design.

In the top-down methodology, you can use incremental block-based compilation to preserve the core logic with the **Preservation Level** setting, or another developer working on the same project preserves the partitions and provides a `.qdb` file.

You can use incremental block-based compilation in combination with the root partition reuse flow, exporting the root partition, and then reusing that root partition in the same project.

1.3.2. Bottom-Up Design Methodology Overview

In a bottom-up design methodology, you create lower-level design blocks independently of one another, and then integrate the blocks at the top-level.

To implement a bottom-up design, individual developers or IP providers can complete the placement and routing optimization of their design in separate projects, and then reuse lower-level blocks in the top-level project. This methodology can be useful for team-based design flows with developers in other locations, or when third-parties create design blocks.

However, when developing design blocks independently in a bottom-up design flow, individual developers may not have all the information about the overall design, or understand how their block connects with other blocks. The absence of this information can lead to problems during system integration, such as difficulties with timing closure, or resource conflicts. To reduce such difficulties, plan the design at the top level, whether optimizing within a single project, or optimizing blocks independently in separate projects, for subsequent top-level integration.

Teams that use a bottom-up design method can optimize placement and routing of design partitions independently. However, the following drawbacks can occur when optimizing the design partitions in separate projects:

- Achieving timing closure for the full design may be more difficult if you compile partitions independently without information about other partitions in the design. Avoiding this problem requires careful timing budgeting and observance of design rules, such as always registering the ports at the module boundaries.
- The design requires resource planning and allocation to avoid resource conflicts and overuse. Floorplanning with Logic Lock regions can help you avoid resource conflicts while developing each part independently in a separate Intel Quartus Prime project.
- Maintaining consistency of assignments and timing constraints is more difficult if you use separate Intel Quartus Prime projects. The team lead must ensure that the assignments and constraints of the top-level design, and those developers define in the separate projects and reuse at the top-level, are consistent.

Partitions that you develop independently all must share a common set of resources. To minimize issues that can arise when sharing a common set of resources between different partitions, create the partitions within in a single project, or in copies of the top-level project, with full design-level constraints, to ensure that resources do not overlap. Correct use of partitions and Logic Lock regions can help to minimize issues that can arise when integrating into the top-level design.

If a developer has no information about the top-level design, the team lead must at least provide a specific Intel FPGA device part number, along with any required physical timing constraints. The developer can then create and export the partition from a separate project. When a developer lacks information, the developer should overconstrain or create additional timing margin on the critical paths. The technique helps to reduce the chance of timing problems when integrating the partitions with other blocks.

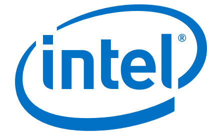
You can use the bottom-up design methodology in conjunction with the core partition reuse flow to independently develop and export a core partition `.qdb` for reuse by a the team lead.

Related Information

Bottom-Up Design Considerations on page 34

1.3.3. Team-Based Design Methodology Overview

In a team-based design methodology, a team lead sets up the top-level project and constraints (including the top-level clock, I/O, and inter-partition constraints), and determines which portions of the design that other team members develop.



Team-based design combines a top-down methodology (where all developers must be aware of the top-level project structure and constraints), with elements of bottom-up flows (where developers work separately on lower-level blocks and integrate them into the top-level). In the top-down methodology, you can use incremental block-based compilation to preserve the core logic with the **Preservation Level** partition setting, or another developer working on a copy of the same project preserves the partitions and provides a `.qdb` file.

The project lead must ensure that the top-level project contains all the interfaces for the design blocks that other team leaders add later. Each team member then develops their portion of the design, and may specify other constraints specific to their design partition. The team members implement the individual design blocks in the context of the top-level design, to avoid integration issues later. As the project nears completion, the team lead then integrates partitions from team members into the top-level project, accounting for any new constraints for the imported partitions.

Individual team members can optionally work on a copy of the same top-level project. The team member creates a partition for their respective design block, compiles the design, and then exports the partition. The team lead then integrates each design partition into the top-level design.

To simplify full design optimization, allow full-chip placement and routing of the partition at the top-level. Export and reuse only the synthesized snapshot, unless the top-level design requires optimized post-fit results.

Related Information

Setting-Up Team-Based Designs on page 33

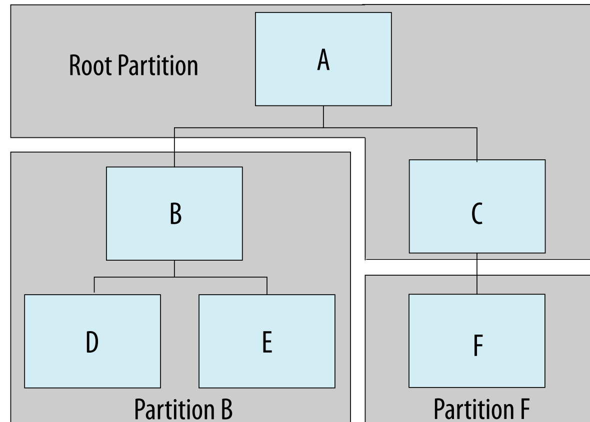
1.4. Design Partitioning

To use block-based design flows, you must first create design partitions from your design's hierarchical instances. The Compiler then treats the design partitions separately to allow the block-based functionality.

The Intel Quartus Prime software automatically creates a top-level (|) "root_partition" for each project revision. The root partition contains all the periphery resources, and may also include core resources. When you export the root partition for reuse, the exported partition excludes all logic in reserved core partitions. To export and reuse periphery elements, you export the root partition.

When you create partitions, every hierarchy within that partition becomes part of the parent partition. Any child partitions inherit any preservation attribute of the parent. The partition creates a logical boundary that prevents merging or optimizing between partitions. The following *Design Partitions in Design Hierarchy* diagram illustrates design partition relationships and boundaries.

Figure 5. Design Partitions in Design Hierarchy



Note:
- Instances B and F are design partitions.
- Partition B includes sub-instances D and E.
- The root partition contains the top-level instance A and instance C, because C is unassigned to any partition.

Design partitions facilitate incremental block-based compilation and design block reuse by logically separating instances. This logical separation allows the Compiler to synthesize and optimize each partition separately from the other parts of the design. The logical separation also prevents Compiler optimizations across partition boundaries.

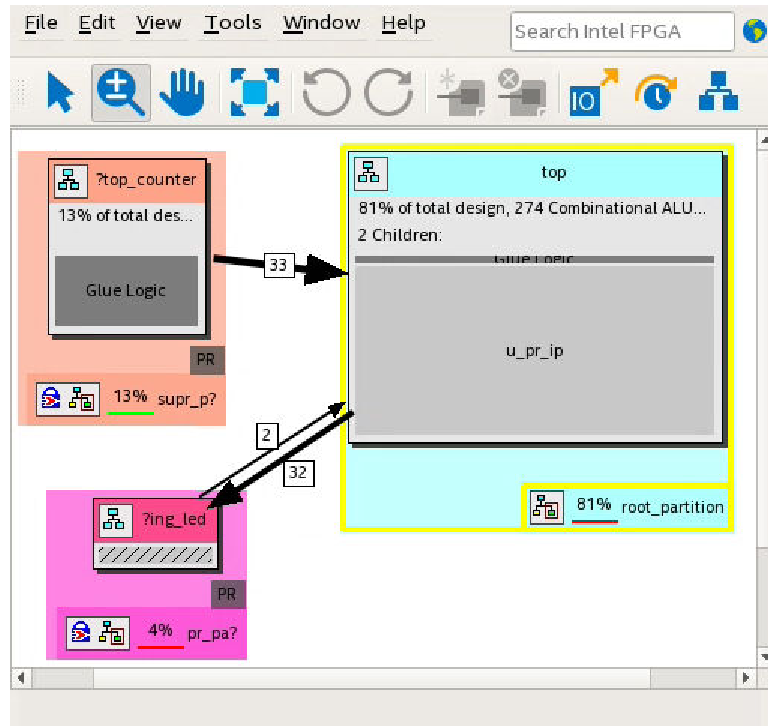
Block-based design requires that you plan and structure the source code and design hierarchy to ensure proper logic grouping for optimization. Implementing the correct logic grouping is easiest early in the design cycle.

Creating or removing a design partition changes the synthesis and subsequent physical implementation and quality of results. When planning the design hierarchy, be aware of the size and scope of each partition, and the possibility of different parts of the design changing during development. Separate logic that changes frequently from the fixed parts of the design.

Group design blocks in your design hierarchy so that highly-connected blocks have a shared level of design hierarchy for assignment to one partition. Structuring your design hierarchy appropriately reduces the required number of partition boundaries, and allows maximum optimization within the partition.

The Design Partition Planner (**Tools ➤ Design Partition Planner**) helps you to visualize and refine a design's partitioning scheme by showing timing information, relative connectivity densities, and the physical placement of partitions. You can locate partitions in other viewers, or modify or delete partitions in the Design Partition Planner.

Figure 6. Design Partition Planner



Consider creating each design entity that represents a partition instance in a separate source file. This approach helps you correlate which partitions require recompilation, instead of reusing preserved results, when you make source code changes. As you make design changes, you can designate partitions as empty or preserved to instruct the Compiler which partitions to recompile from source code, as Design Abstraction on page 30 describes.

If your design has timing-critical partitions that are changing through the design flow, or partitions exported from another project, use design floorplan assignments to constrain the placement of the affected partitions. A properly partitioned and floor-planned design enables partitions to meet top-level design requirements when you integrate the partitions with the rest of your design. Poorly planned partitions or floorplan assignments negatively impact design area utilization and performance, thereby increasing the difficulty of timing closure.

The following design partition guidelines help ensure the most effective and efficient results. Block-based design flows add steps and requirements to the design process, but can provide significant benefits in design productivity.

Related Information

Step 1: Developer: Create a Design Partition on page 19

1.4.1. Planning Partitions for Periphery IP, Clocks, and PLLs

Use the following guidelines to plan partitions for periphery IP, clocks, and PLLs:

Planning Partitions for Periphery IP

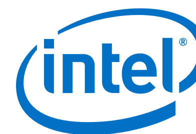
- Plan the design periphery to segregate and implement periphery resources in the root partition. Ensure that IP blocks that utilize both core and periphery resources (such as transceiver and external memory interface Intel FPGA IP) are part of the root partition.
- When creating design partitions for an existing design, remove all periphery resources from any entity you want to designate as a core partition. Also, tunnel any periphery resource ports to the top level of the design. Implement the periphery resource in the root partition.
- You cannot designate instances that use periphery resources as separate partitions. In addition, you cannot split an Intel FPGA IP core into more than one partition.
- The Intel Quartus Prime software generates an error if you include periphery interface Intel FPGA IP cores in any partition other than the top-level root partition.
- You must include Intel FPGA IP cores for the Hybrid Memory Cube (HBM) or Hard Processor System (HPS) in the root partition.

Planning Partitions for Clocks and PLLs

- Plan clocking structures to retain all PLLs and corresponding clocking logic in the root partition. This technique allows the Compiler to control PLLs in the root partition, if necessary.
- Consider creating a design block for all clocking logic that you instantiate in the top-level of the design. This technique ensures that the Compiler groups clocking logic together, and that the Compiler treats clocking logic as part of the root partition. Clock routing resources belong to the root partition, but the Compiler does not preserve routing resources with a partition.
- Include any signal that you want to drive globally in the root partition, rather than the core partition. Signals (such as clocks or resets) that you generate inside core partitions cannot drive to global networks without a clock buffer in the root partition.
- To support existing Intel Arria 10 designs, the Compiler allows I/O PLLs in core partitions. However, creating a partition boundary prevents such PLLs from merging with other PLLs. The design may use more PLLs without this merging, and may have suboptimal clocking architecture.

1.4.2. Design Partition Guidelines

Creating a design partition creates a logical hierarchical boundary around that instance. This partition boundary can limit the Compiler's ability to merge the partition's logic with other parts of the design. A partition boundary can also prevent optimization that reduces cell and interconnect delay, thereby reducing design performance. To minimize these effects, follow these general design partition guidelines:



- Register partition boundary ports. This practice can reduce unnecessary long delays by confining register-to-register timing paths to a single partition for optimization. This technique also minimizes the effect of the physical placement for boundary logic that the Compiler might place without knowledge of other partitions.
- Minimize the timing-critical paths passing in or out of design partitions. For timing critical-paths that cross partition boundaries, rework the partition boundaries to avoid these paths. Isolate timing-critical logic inside a single partition, so the Compiler can effectively optimize each partition independently.
- Avoid creating a large number of small partitions throughout the design. Excessive partitioning can impact performance by preventing design optimizations.
- Avoid grouping unrelated logic into a large partition. If you are working to optimize an independent block of your design, assigning that block as a small partition provides you more flexibility during optimization.
- When using incremental block-based design within a single project, the child partition must have an equal or higher preservation level than the parent. The Compiler generates an error during synthesis when the parent partition has a higher preservation level than the child partition.

1.4.3. Partition Snapshot Preservation and Reuse

The Compiler generates a snapshot of compilation results, for each partition, at each stage of compilation. You can preserve and reuse partitions after synthesis or after the final stage of the Fitter. The following table describes what you can preserve or reuse from each Compiler snapshot:

Table 2. Description of Preservation and Reuse with each Compiler Snapshot

Snapshot	Description	Preserve/Reuse
Synthesized	Preserves a synthesized netlist that represents the logic for a design partition.	• Applies to `.qdb` and partition preservation. • Preserves logical boundaries. • No physical preservation. • Supports snapshot reuse across devices in the same family.
Final	Preserves final device utilization, placement, routing, and hold time fix-up.	• Applies to `.qdb` files and partition preservation. • Intel Arria 10 High-Speed/Low-Power Tile settings do not change. • Intel Stratix® 10 retimed register locations do not change. • Does not preserve global signals or signals that cross partition boundaries. • Supports snapshot reuse across projects targeting the same device.

When reusing snapshots in a project targeting another device in the same family, you can only reuse the Synthesized snapshot, rather than the Final snapshot. The following additional limitations apply to reuse of snapshot across devices within the same family.

Table 3. Limitations on Snapshot Reuse Across Devices in Family

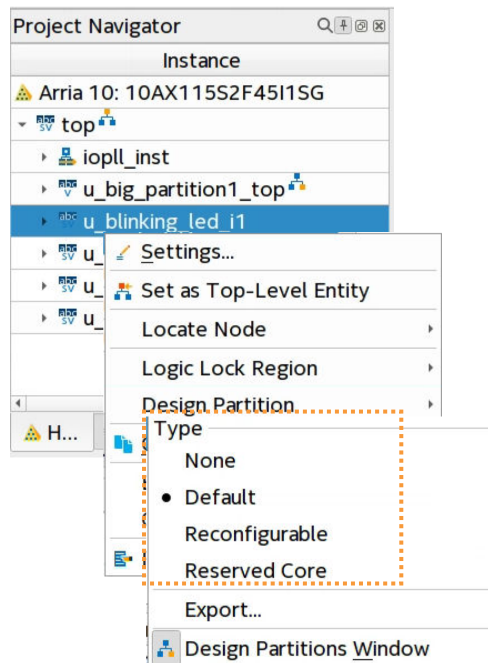
Snapshot Reuse Scenario	Result
Import of Final snapshot across devices in same family.	The Fitter issues an error during compilation in the Consumer project.
Import of Synthesized snapshot across devices in different families	
Import from or export to a project that targets the AUTO device	The Compiler supports this scenario and issues no error if target device is within the same family.
Import of the root_partition (device periphery) Synthesized snapshot	The Compiler supports this scenario as long as the imported partition contains no invalid Fitter assignments (such as conflicting Logic Lock regions). Invalid Fitter assignments produce unexpected results.

1.4.4. Creating Design Partitions

Follow these steps to create and modify design partitions:

1. Click **Processing ➤ Start ➤ Start Analysis & Elaboration**.
2. In the Project Navigator, right-click an instance in the **Hierarchy** tab, point to **Design Partition**, and click a design partition **Type**. A design partition icon appears next to each instance you assign.

Figure 7. Creating a Design Partition from the Project Hierarchy



This setting corresponds to the following assignment in the .qsf:

```
set_instance_assignment -name PARTITION <name> \
     -to <partition hierarchical path>
```

3. To view and edit all design partitions in the project, click **Assignments ➤ Design Partitions Window**.

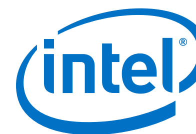


Figure 8. Design Partitions Window

Design Partitions...

Assignments View | Compilation View

Partition Name	Hierarchy Path	Type	Partition Database File	Entity Re-binding	Post Synthesis Export File
<<new>>					
root_partition	|				
blinking_led	u_blinking_led	Default			

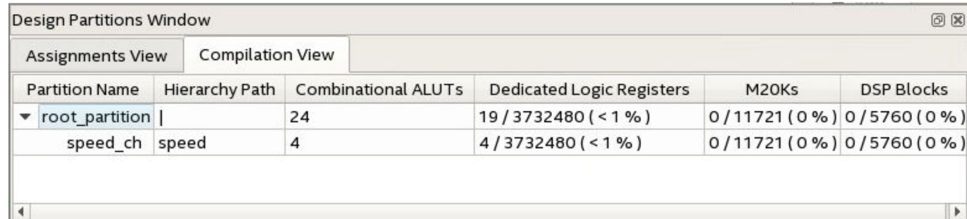
4. Specify the properties of the design partition in the Design Partitions Window. The following settings are available:

Table 4. 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: • **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.
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.

Following compilation, you can view details about design partition implementation in the **Compilation View** tab of the Design Partitions Window. The synthesis and Fitter reports provide additional information about preservation levels and `.qdb` file assignments.

Figure 9. Design Partition Window Compilation View Tab



Note: You can only preserve paths inside a partition, and cannot preserve the paths crossing from one partition to another. Although you cannot merge partitions together, you can create a RTL wrapper to wrap modules that you want to group into a partition, and then assign a design partition to the RTL wrapper.

1.5. Design Block Reuse Flows

Design block reuse allows you to preserve a design partition as an exported `.qdb` file, and reuse this partition in another project. Reuse of core or root partitions involves partitioning and constraining the block prior to compilation, and then exporting the block for reuse in another project. Effective design block reuse requires planning to ensure that the source code and design hierarchy support the physical partitioning of device resources that these flows require.

- **Core partition reuse**—allows reuse of synthesized or final snapshots of a core partition. A core partition can include only core resources (LUTs, registers, M20K memory blocks, and DSPs).
- **Root partition reuse**—allows reuse of a synthesized or final snapshot of a root partition. A root partition includes periphery resources (including I/O, HSSIO, PCIe, PLLs), as well as any associated core resources, while leaving a core partition open for subsequent development.

At a high level, the core and root partition reuse flows are similar. Both flows preserve and reuse a design partition as a `.qdb` file. The Developer defines, compiles, and preserves the block in the Developer project, and the Consumer reuses the block in one or more Consumer projects.

The following sections describe the core and root partition reuse flows in detail.

Related Information

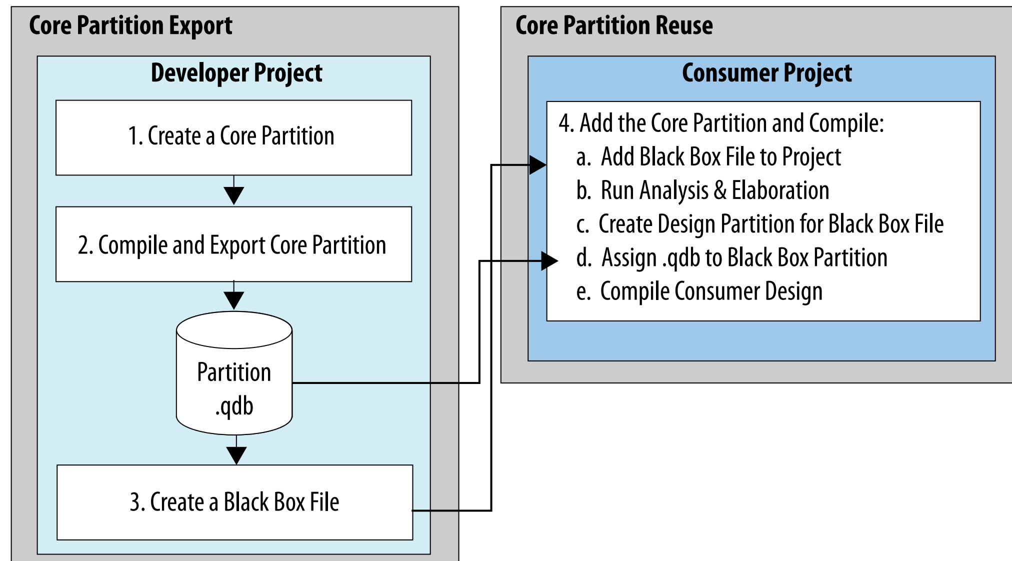
- Design Block Reuse Overview on page 5
- AN 839: Design Block Reuse Tutorial for Intel Arria 10 FPGA Development Board

1.5.1. Reusing Core Partitions

Reusing core partitions involves exporting the core partition from the Developer project as a `.qdb`, and then reusing the `.qdb` in a Consumer project.

The Consumer assigns the .qdb to an instance in the Consumer project. In the Consumer project, the Compiler runs stages not already exported with the partition.

Figure 10. Core Partition Reuse Flow



The following steps describe the core partition reuse flow in detail.

1.5.1.1. Step 1: Developer: Create a Design Partition

Define design partitions to create logical boundaries in the design hierarchy. Confine each core instance for export within a design partition. You can define partition instances from the Project Navigator or in the Design Partitions Window.

To define a core design partition:

1. Review the project to determine design elements suitable for reuse, and the appropriate snapshot for export.
2. Refer to Creating Design Partitions on page 16 to define a core partition. Select **Default** for the partition **Type**.

Related Information

Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration

1.5.1.2. Step 2: Developer: Compile and Export the Core Partition

This step describes generating and exporting a final snapshot of the core partition. You can manually export the core partition as a .qdb after compilation, or you can specify settings to automate export each time you compile. You can then reuse the core partition in the same project or in another project, starting the partition's compilation at the stage following the snapshot.

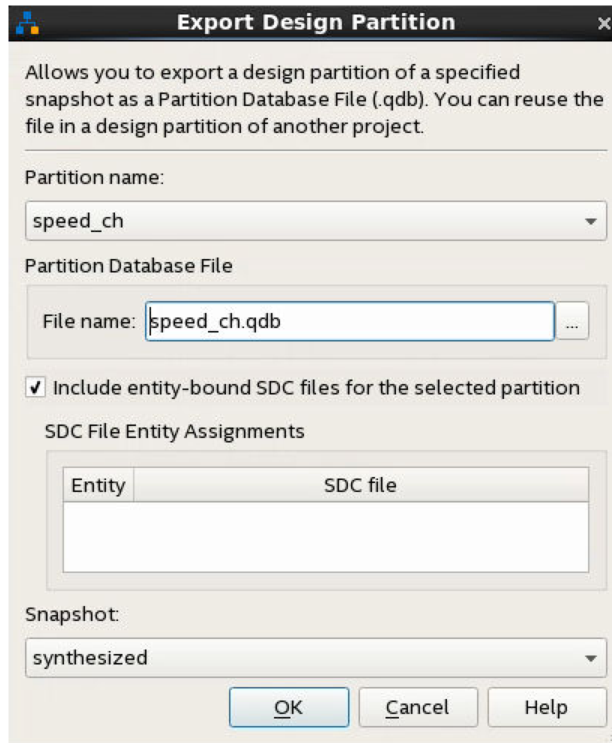
Manual Partition Export

To compile and manually export a core partition:

1. To run all compilation stages through Fitter (Finalize) and generate the final snapshot, click **Processing ➤ Start ➤ Start Fitter**.
2. To export the core partition, click **Project ➤ Export Design Partition**. Select the **Design Partition** name and the compilation **Snapshot** for export.
3. To include any entity-bound `.sdc` files in the exported `.qdb`, turn on **Include entity-bound SDC files for the selected partition**. By default, all Intel FPGA IP targeting Intel Stratix 10 devices use entity-bound `.sdc` files.

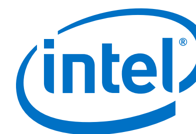
   Note: Intel FPGA IP targeting Intel Arria 10 devices do not use entity-bound `.sdc` files by default. To use this option for Intel Arria 10 devices, you must first bind the `.sdc` file to the entity in the `.qsf`. Refer to "Using Entity-bound SDC Files," in *Intel Quartus Prime Pro Edition User Guide: Timing Analyzer*.
4. Confirm the **File name** for the **Partition Database File**, and then click **OK**.

Figure 11. Export Design Partition



The following command corresponds to partition export in the GUI:

```
quartus_cdb <project name> -c <revision name> \
     --export_partition "<name>" --snapshot synthesized \
     --file <name>.qdb --include_sdc_entity_in_partition
```

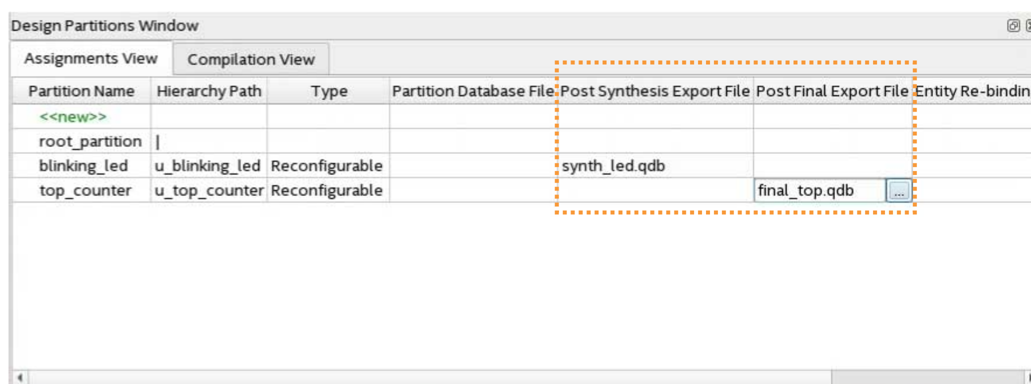


Automated Design Partition Export

Follow these steps in the Design Partitions Window to automatically export one or more design partitions following each compilation:

1. To automatically export a partition with synthesis results after each time you run synthesis, specify the a `.qdb` export path and file name for the **Post Synthesis Export File** option for that partition. If you specify only a file name without path, the file exports to the project directory after compilation.
2. To automatically export a partition with final snapshot results each time you run the Fitter, specify a `.qdb` file name for the **Post Final Export File** option for that partition.

Figure 12. Specifying Export File in Design Partitions Window



`.qsf` assignment syntax:

```
set_instance_assignment -name EXPORT_PARTITION_SNAPSHOT_SYNTHESIZED \
     <qdb file name> -to <hierarchy path> -entity <entity name>
```

Related Information

"Using Entity-bound SDC Files," Intel Quartus Prime Pro Edition User Guide: Timing Analyzer

1.5.1.3. Step 3: Developer: Create a Black Box File

Reusing a core partition `.qdb` file also requires that you add a supporting black box file to the Consumer project. A black box file is an RTL source file that only contains port and module or entity definitions, but does not contain any logic. The black box file defines the ports and port interface types for synthesis in the Consumer project. Follow these steps to create a block box port definitions file for the partition.

The Compiler analyzes and elaborates any RTL that you include in the black box file. Edits to the RTL do not affect a partition that uses a `.qdb` file.

1. Create an HDL file (`.v`, `.vhd`, `.sv`) that contains only the port definitions for the exported core partition. Include parameters or generics passed to the module or entity. For example:

```
module bus_shift #(
    parameter DEPTH=256,
    parameter WIDTH=8
  )(
```

```
    input clk,
    input enable,
    input reset,
    input [WIDTH-1:0] sr_in,
    output [WIDTH-1:0] sr_out
  );
endmodule
```

2. Provide the black box file and exported core partition `.qdb` file to the Consumer.

1.5.1.4. Step 4: Consumer: Add the Core Partition and Compile

To add the core partition, the Consumer adds the black box as a source file in the Consumer project. After design elaboration, the Consumer defines a core partition and assigns the exported `.qdb` file to an instance in the Design Partitions Window. Because the exported `.qdb` includes compiled netlist information, the Consumer project must target the same FPGA device family, and use the same Intel Quartus Prime software version, as the Developer project. The Consumer must supply a clock and any other constraints required for the interface to the core partition.

To add the core partition and compile the Consumer project:

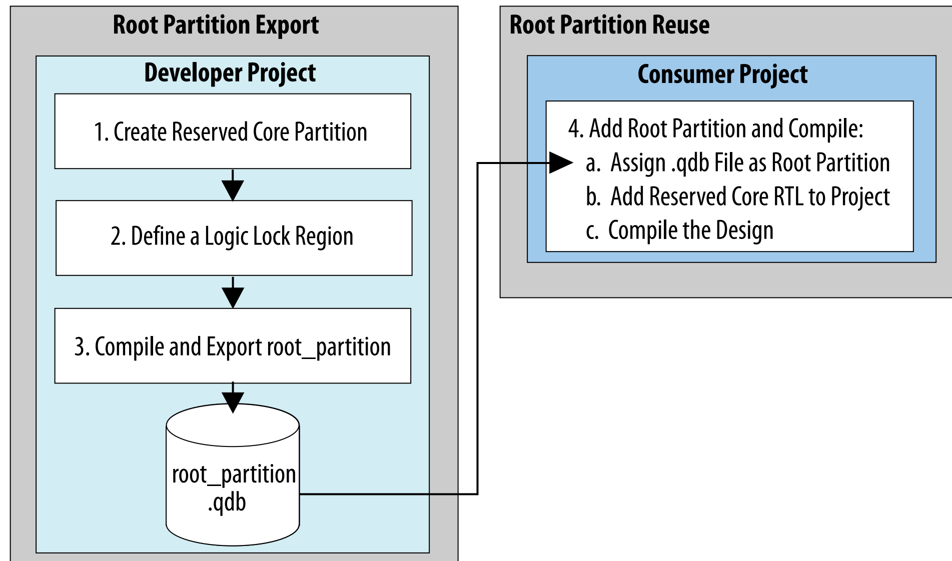
1. Create or open the Intel Quartus Prime project that you want to reuse the core partition.
2. To add one or more black box files to the consumer project, click **Project ➤ Add/ Remove Files in Project** and select the black box file.
3. Follow the steps in Creating Design Partitions on page 16 to elaborate the design and define a core partition for the black box file. When defining the design partition, click the **Partition Database File** option and select the exported `.qdb` file for the core partition.
4. To run all compilation stages through Fitter (Finalize) and generate the final snapshot, click **Processing ➤ Start ➤ Start Fitter**.

1.5.2. Reusing Root Partitions

The root partition contains all the periphery resources, and may also include some core resources. To export and reuse periphery elements, you export the root partition. Reuse of root partitions allows you to design an FPGA-to-board interface and associated logic once, and then replicate that interface in other projects.

Note: When reusing the root partition across different devices within the same family, you can only reuse the Synthesized snapshot, and you must ensure that any Fitter constraints (such as Logic Lock regions) from the Developer project do not conflict with constraints in the Consumer project.

Figure 13. Root Partition Reuse Flow



1.5.2.1. Step 1: Developer: Create a Reserved Core Partition

To export and reuse the root partition, the Developer creates a reserved core partition for later core logic development in the Consumer project. The Compiler preserves post-fit results for the partition and reuses the post-fit netlist, if the netlist is available from previous compilation, and you make no partition changes requiring re-synthesis. Otherwise, the Compiler reuses the post-synthesis netlist if available, or resynthesizes the partition from source files.

To create a reserved core partition:

1. Adapt the steps in Step 1: Developer: Create a Design Partition on page 19 to create a reserved core partition.
2. When defining the design partition, select **Reserved Core** for the partition **Type**. Ensure that all other partition options are set to the default values.

1.5.2.2. Step 2: Developer: Define a Logic Lock Region

To reserve core resources in a Consumer project for the reserved core partition, the Developer defines a fixed size and location, core-only, reserved Logic Lock region with a defined routing region. The Consumer uses this same region in their project for core development. This region can contain only core logic. Ensure that the reserved placement region is large enough to contain all core logic that the Consumer plans to develop. For projects with multiple core partitions, constrain each partition in a non-overlapping Logic Lock routing region.

Note: When reusing the root partition across different devices within the same family, you can only reuse the Synthesized snapshot, and you must ensure that any Fitter constraints (such as Logic Lock regions) from the Developer project do not conflict with constraints in the Consumer project.

Follow these steps to define a Logic Lock region for core development in the Developer project:

1. Right-click the design instance in the **Project Navigator** and click **Logic Lock Region ➤ Create New Logic Lock Region**. The region appears in the Logic Lock Regions Window. You can also verify the region in the Chip Planner (**Locate Node ➤ Locate in Chip Planner**).
2. Specify the placement region co-ordinates in the **Origin** column.
3. Enable the **Reserved** and **Core-Only** options.
4. For **Size/State**, select **Fixed/Locked**.
5. Click the **Routing Region** cell. The **Logic Lock Routing Region Settings** dialog box appears.

Figure 14. Logic Lock Regions Window

Logic Lock Regions Window

Region Name	Members	Width	Height	Origin	Reserved	Core-Only	Size/State	Routing Region
Logic Lock Regions								
speed_ch	speed	20	20	X1_Y1	Off	On	Fixed/Locked	Fixed with expansion 1
<<new>>								

6. Specify **Fixed with expansion** with **Expansion Length** of **1** for the **Routing Type**. For this flow you can select any value other than **Unconstrained**
7. Click **OK**.
8. Click **File ➤ Save Project**.

1.5.2.3. Step 3: Developer: Compile and Export the Root Partition

After compilation, the Developer exports the root partition at the synthesized or final stage. The Developer supplies any Synopsys* Design Constraints (`.sdc`) file for the partition. The Developer uses the `.sdc` files to drive placement and routing. The Consumer uses `.sdc` files for evaluation of partitions that reuse `.qdb` files, and to drive placement in the Fitter for non-reused or non-preserved partitions.

1. To run all compilation stages through Fitter (Finalize), click **Processing ➤ Start ➤ Start Fitter**.
2. To export the root partition to a `.qdb` file, click **Project ➤ Export Design Partition**. Select the **root_partition** and the **synthesized** or **final** snapshot.
3. To include any entity-bound `.sdc` files in the exported `.qdb`, turn on **Include entity-bound SDC files for the selected partition**. By default, all Intel FPGA IP targeting Intel Stratix 10 devices use entity-bound `.sdc` files.

   The following command corresponds to the root partition export in the GUI:

```
quartus_cdb <project name> -c <revision name> \
     --export_partition "root_partition" --snapshot final \
     --file root_partition.qdb --include_sdc_entity_in_partition
```

4. The Developer provides the exported `.qdb` file and `.sdc` files for the reserved core to the Consumer.

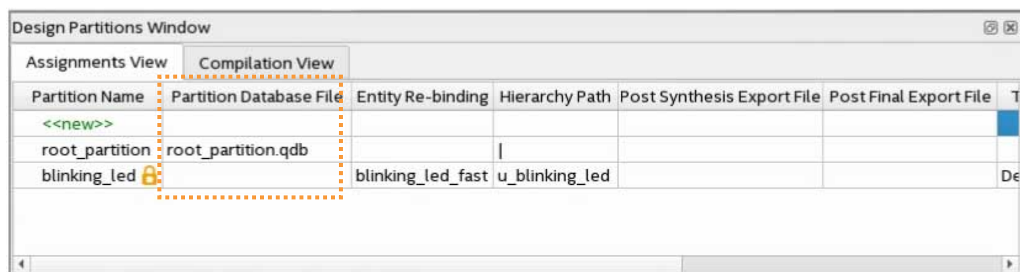
1.5.2.4. Step 4: Consumer: Add the Root Partition and Compile

To reuse the root partition in another project, the Consumer assigns the exported root partition `.qdb` in the Consumer project settings. The root partition `.qdb` includes all Logic Lock region and partition information for the reserved core from the Developer project. There is no need to recreate these constraints in the Consumer project. After assigning the `.qdb`, the Consumer project includes all additional information from the Developer project compilation snapshot. The Consumer then adds RTL for the reserved core partition.

Follow these steps to reuse the root partition in a Consumer project:

1. The Consumer obtains the exported root partition `.qdb` file from the Developer.
2. Open the project that you want to reuse the exported root partition.
3. In the Design Partitions Window, specify a `.qdb` in **Partition Database File** to replace the root_partition logic.

Figure 15. Partition Database File Option in Design Partitions Window



4. The Consumer adds RTL and any `.sdc` constraints for the reserved core partition.
5. To enable the **Fast Preserve** option that simplifies the logic of the preserved partition to only interface logic[2] during compilation, click **Assignments ➤ Settings ➤ Compiler Settings ➤ Incremental Compile ➤ Fast Preserve**.
6. To run all compilation stages, click **Processing ➤ Start Compilation**. The Compiler implements the reused root partition and constraints.

   Note: To use the Entity Re-binding option, you add the .qdb to the project by specifying a .qdb for the **Partition Database File** option in the Design Partitions Window. Refer to Reserved Core Entity Re-Binding on page 25 for more information.

1.5.3. Reserved Core Entity Re-Binding

Entity re-binding allows the Consumer in a root partition reuse flow to use an entity name that is different from the Developer's reserved core partition name in the `root_partition.qdb`.

Entity Re-binding Example

The following example illustrates application of **Entity Re-binding**. You specify a value for the **Entity Re-binding** option in the Design Partitions Window to identify the entity bound to a reserved core partition.

---

(2) Interface logic is logic at the partition boundary that interfaces with the rest of the design.

Figure 16. Entity Rebinding Option in Design Partitions Window

Design Partitions Window

Assignments View | Compilation View

Partition Name	Partition Database File	Entity Re-binding	Hierarchy Path	Post Synthesis Export File	Post Final Export File	T
<<new>>						
root_partition	root_partition.qdb		|			
blinking_led		blinking_led_fast	u_blinking_led			De

In a root partition reuse example, the Developer's project includes the shell with entity name `blinking_led`, and the `u_blinking_led` instance. The Developer exports the `root_partition.qdb` that includes the root partition, and a reserved core region defined by a Logic Lock placement constraint. The reserved region is associated with the `u_blinking_led` instance.

The Consumer reuses the `root_partition.qdb` in their project. However, the entity that replaces the reserved core has a different name (`blinking_led_fast`) than the reserved core name (`u_blinking_led`) in the Developer project.

If the Consumer simply adds the `u_blinking_led` entity to the project without entity re-binding, an error occurs.

Rather, the Consumer can re-bind the new entity name to the reserved core partition name by setting the **Entity Re-binding** option to `blinking_led_fast`.

1.5.4. Viewing Quartus Database File Information

Although you cannot directly read a `.qdb` file, you can view helpful attributes about the file to quickly identify its contents and suitability for use.

The Intel Quartus Prime software automatically stores metadata about the project of origin when you export a Quartus Database File (`.qdb`). The Intel Quartus Prime software automatically stores metadata about the project of origin and resource utilization when you export a Partition Database File (`.qdb`) from your project. You can then use the Quartus Database File Viewer to display the attributes any of these `.qdb` files.

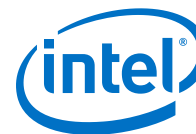


Figure 17. Quartus Database File Viewer

blinking_led.qdb

Project Summary:

Attribute	Value
Project Information	
Contents	Partition
Date	Thu Aug 16 10:37:40 2018
Device	1SG280HN1F43E2VGS1
Entity	blinking_led
Family	Stratix 10
Partition Name	blinking_led
Revision Name	blinking_led
Revision Type	Unspecified
Snapshot	synthesized
Version	18.1.0
Version-Compatible	No
Resource Utilization	
Average fan-out	0.16
Combinational ALUT usage for logic	0
Dedicated logic registers	2
Estimate of Logic utilization (ALMs needed)	1
I/O pins	35
Maximum fan-out	2
Maximum fan-out node	u_blinking_led|counter[23]
Total DSP Blocks	0
Total fan-out	6

Follow these steps to view the attributes of a .qdb file:

1. In the Intel Quartus Prime software, click **File ➤ Open**, select **Design Files** for **Files of Type**, and select a .qdb file.
2. Click **Open**. The Quartus Database File Viewer displays project and resource utilization attributes of the .qdb.

   Alternatively, run the following command-line equivalent:

```
quartus_cdb --extract_metadata --file <archive_name.qdb> \
     --type quartus --dir <extraction_directory> \
     [--overwrite]
```

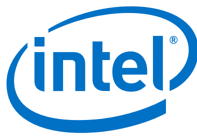
1.5.4.1. QDB File Attribute Types

The Quartus Database Viewer can display the following attributes of a .qdb file:

Table 5. QDB File Attributes

QDB Attribute Types	Attribute	Example
Project Information	Contents	`Partition`
	Date	`Thu Jan 23 10:56:23 2018`
	Device	`10AX016C3U19E2LG`
	Entity (if Partition)	`Counter`
	Family	`Arria 10`
	Partition Name	`root_partition`

continued...



	Revision Name	Top
	Revision Type	PR_BASE
	Snapshot	synthesized
	Version	18.1.0 Pro Edition
	Version-Compatible	Yes
Resource Utilization (exported for partition QDB only)	For synthesized snapshot partition lists data from the **Synthesis Resource Usage Summary** report.	Average fan-out.16 Dedicated logic registers:14 Estimate of Logic utilization:1 I/O pins:35 Maximum fan-out:2 Maximum fan-out node:counter[23] Total DSP Blocks:0 Total fan-out:6 ...
	For the final snapshot partition, lists data from the **Fitter Partition Statistics** report.	Average fan-out:.16 Combinational ALUTs: 16 I/O Registers M20Ks ...

1.6. Incremental Block-Based Compilation Flow

The incremental block-based compilation flow allows you incrementally close timing by preserving specific design partitions, and perform design abstraction by emptying a partition.

Related Information

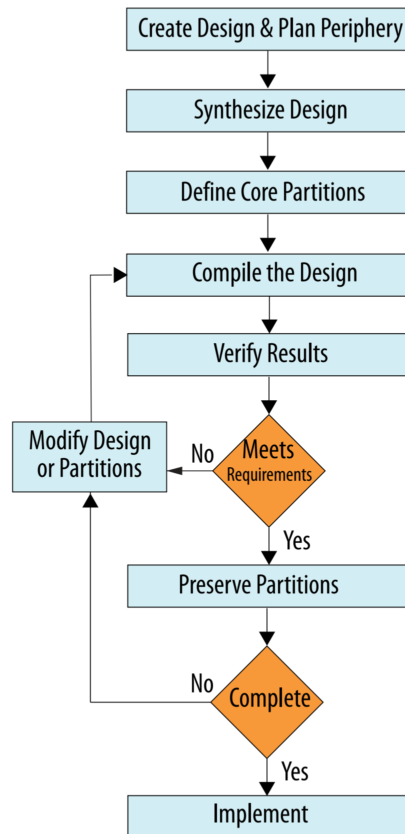
Incremental Block-Based Compilation Overview on page 8

1.6.1. Incremental Timing Closure

In incremental timing closure, you iteratively preserve partitions as they meet requirements, thereby reducing timing closure complexity to a single design partition, and allowing Fitter optimizations and small RTL changes without affecting other partitions.

When you preserve the compilation results for a core partition, the snapshot remains unchanged for subsequent compilations. The preserved core partition becomes the source for each subsequent compilation.

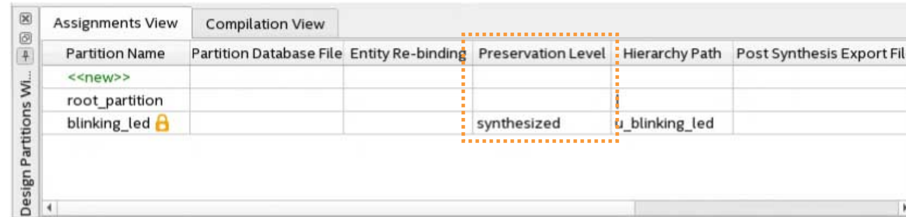
Figure 18. Incremental Timing Closure Flow



The following steps describe this flow:

1. Design partitioning—define the design partitions that your design requires, as Creating Design Partitions on page 16 describes.
2. **Processing ➤ Start Compilation**
3. Initial full compilation and timing analysis—run a full compilation of the design (, and run static timing analysis (**Tools ➤ Timing Analyzer**) to identify the partitions that meet requirements.
4. Incremental timing closure—as the core design partitions meet timing requirements, preserve timing-closed partitions with the **Preservation Level** option. Incrementally recompile the remaining partitions to further optimize RTL, or to apply additional Fitter optimization settings. Resynthesize any changes to RTL, without resynthesis of the preserved partitions.
5. Final compilation—when you are ready to run full compilation of the entire design, the partitions you preserve remain unchanged. The preserved partitions use the preserved snapshot results during compilation. Non-preserved partitions use the synthesized snapshot during compilation.

Figure 19. Settings Partition Preservation in Design Partitions Window



1.6.1.1. Incremental Timing Closure Recommendations and Limitations

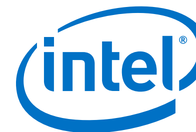
The following consideration and limitation apply to incremental timing closure:

Table 6. Incremental Timing Closure Considerations and Limitations

Recommendation/ Limitation	Description
Recommendation 1	Set partition boundaries near major registered block interfaces, where there is minimal connectivity to external blocks, as these partition boundaries prevent cross-block optimization.
Recommendation 2	For incremental timing closure, there is no requirement to floorplan partitions with Logic Lock regions. However, use of Logic Lock regions can help the Compiler to achieve timing closure more quickly.
Recommendation 3	Lock down the source of the clock.
Recommendation 4	Lock down the region covered by the clock.
Limitation 1	The Compiler does not support partial periphery preservation. You can preserve only the full periphery (root_partition).
Limitation 2	Preserving a partition does not preserve any incoming and outgoing routing. The Compiler attributes routing that crosses partition boundaries to the common parent partition. More precisely, the Compiler determines routing based on the hierarchies that the net traverses in the RTL. Preserving parent partitions allows preservation of inter-partition routing.
Limitation 3	The Compiler generally routes global clocks from the top-level pins (in the root partition) to lower-level partitions. As a result, preserving a partition does not preserve these cross-partition global routes. When using incremental timing closure, the partitions you preserve can still be subject to slight timing variations due to difference in clock arrival times between compilations.
Limitation 4	The following factors can affect the timing of partitions you preserve: • The global clock network, which has a more significant impact for Intel Stratix 10 designs. • Cross talk. The impact is similar for Intel Stratix 10 and Intel Arria 10 designs. • Routing muxes.

1.6.2. Design Abstraction

Empty partitions are useful to account for undefined partitions developed independently or later in the design cycle. The Compiler uses an empty placeholder netlist for the partition, ties the partition output ports to ground, and removes the input ports. The Compiler removes any existing synthesis, placement, and routing information for an empty partition.



If you remove the **Empty** setting from a partition, the Compiler re-implements the partition from the source. Setting a partition to **Empty** can reduce design compilation time because the top-level design netlist does not include the logic for the empty partition. The Compiler does not run full synthesis and fitting algorithms on the empty partition logic. Emptying a preserved partition removes all preserved information.

Note: To avoid resource conflicts when using empty partitions, floorplan any empty partitions that you intend to subsequently replace with a `.qdb`.

Follow these steps to define an empty partition:

1. Create a design partition, as Step 1: Developer: Create a Design Partition on page 19 describes. Set the partition **Type** to **Default**. Any other setting is incompatible with empty partitions.
2. Set the **Preservation Level** to **Not Set**, and make sure there is no `.qdb` for the **Partition Database File** option. Any other setting combination is incompatible with empty partitions.
3. For the **Empty** option, select **Yes**. This setting corresponds to the following assignment in the `.qsf`.

```
set_instance_assignment -name EMPTY ON -to \
     <hierarchal path of partition> -entity <name>
```

1.6.2.1. Empty Partition Clock Source Preservation

Empty partitions preserve clock sources that the Intel Quartus Prime software recognizes.

The Intel Quartus Prime software recognizes and preserves the following as clock sources for a partition:

- Signals from a PLL.
- Feeds from internal clock inputs on flip-flops, memories, HSSIO, I/O registers, or PLLs outside the partition that you empty.

The Intel Quartus Prime software does not recognize the following as clock sources for a partition:

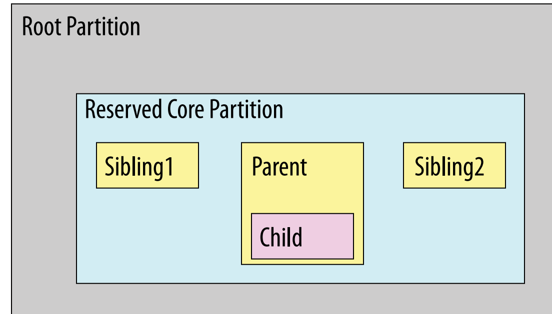
- Nets with sources external to the FPGA that do not feed a clock input inside the FPGA.
- Nets that connect only to combinatorial logic.
- Nets that connect only to an output pin.
- Nets that feed only logic within an empty partition.

1.7. Combining Design Block Reuse and Incremental Block-Based Compilation

Design block reuse enables a Developer or IP vendor to deliver a block or IP to a Consumer without delivering the source code. Incremental block-based compilation allows the IP or block Consumer to independently develop wrapper logic. Enabling developers to preserve parts of the design, while third-party IP is under development, reduces the time and effort required for timing closure of the full design. The following figure illustrates this parallel development model.

In the following figure, `Sibling2` is a third-party IP, or a partition independently designed by another Developer. A Consumer can create a Logic Lock region for `Sibling2`, compile the other partitions, and preserve the results. When the Developer subsequently delivers the `Sibling2 .qdb`, the Consumer then can import this to the full design.

Figure 20. Partitioned Design Hierarchy



The Intel Quartus Prime Pro Edition provides support for the following combinations of incremental block-based compilation and design block reuse:

Table 7. Support for Combined Incremental Block-Based Compilation and Design Block Reuse

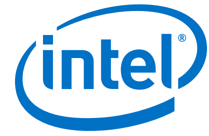
Supported Combinations	Unsupported Combinations
• Compilation and export of the parent `.qdb` file. In a consecutive compilation, you reuse the parent and preserve the child. The parent `.qdb` snapshot level is the same as the child's preservation level.[3] • Export of a partition as a `.qdb`, reuse of the `.qdb`, and preservation of the same partition. The snapshot level that the `.qdb` preserves must be lower than the partition preservation level.[4] • Preservation of a partition and its child. The child preservation level must be higher or equal to the parent preservation level. Otherwise, the combination is not supported	• Compilation and export of the parent `.qdb` file. In a consecutive compilation, you reuse the parent and preserve the child. The child's preservation level is different from the partition `.qdb`. Therefore, the Compiler issues a critical warning, and ignores the child preservation level. • Reuse of a child `.qdb` and the parent is **Empty**. • Reuse of a parent `.qdb` and the child is **Empty**. • Preservation of a child partition and the parent partition is **Empty**. • Preservation of a parent partition and the child is **Empty**. • Reuse a `.qdb` and the partition is **Empty**. • Reuse of a child `.qdb` and preserving the parent partition. • Preservation of a parent and child partition and the parent preservation level is higher than child's preservation level.

Note: All supported combinations require an initial compilation, without any reuse or preservation. You then apply reuse or preservation after the initial compilation is complete. Otherwise, the Compiler ignores the preservation level and issues a critical warning.

---

[3] Although the GUI does not support reuse of a partition .qdb and preservation of its child, you can implement this by modifying the `.qsf` file and compiling at the command line.

[4] Supported via `.qsf` file and command line compilation only.



1.8. Setting-Up Team-Based Designs

Use the following procedures to setup a project for a team-based design methodology.

1.8.1. Creating a Top-Level Project for a Team-Based Design

In team-based designs that reuse design blocks, all team members ideally work within the same top-level project framework. Using copies of the same project among team members ensures that everyone has the same settings and constraints that their partition requires.

This method helps the team to integrate the partitions into the top-level design. If some Developers do not have access to the top-level project framework, the team lead must provide information about the project and constraints to those Developers.

The following steps describe preparing a top-level project that enables other Developers to provide optimized lower-level design partitions. The top-level project specifies the top-level entity, and then instantiates other design entities that other Developers optimize in a separate Intel Quartus Prime project.

1. Set up the top-level project and add source files. You can represent incomplete sections of the design by adding black box files, as Step 3: Developer: Create a Black Box File on page 21 describes.
2. Define design partitions for any instance that you want to maintain as a separate Intel Quartus Prime project, as Creating Design Partitions on page 16 describes.
3. Define an empty partition for each design partition with unknown or incomplete definition.
4. Create a Logic Lock region constraint for each design block that you plan to integrate as a separate Intel Quartus Prime project. This physical partitioning of the device allows multiple team members to design independently without placement conflicts, as Step 2: Developer: Define a Logic Lock Region on page 23 describes.
5. To run full compilation, click **Processing ➤ Start Compilation**.
6. Use one of the following methods to provide the top-level project information to design Developers:

- If Developers have access to the top-level project framework, the team lead includes all settings and constraints. This framework may include clocks, PLLs, and other periphery interface logic that the Developer requires to develop their partition. If Developers are part of the same design environment, they can check out a copy of the project files they require from the same source control system. This is the best method for sharing a set of project files. Otherwise, the team lead provides a copy of the top-level project (the design and corresponding `.qsf` assignments), so that each Developer creates their partition within the same project framework.
- If Developers do not have access to the top-level project framework, the team lead provides a Tcl script or other specifications to create a separate Intel Quartus Prime project that matches the top-level. The team lead also adds logic around the design block for export, so that the partition is consistent with the key characteristics of the top-level design environment. For example, the team lead can include a top-level PLL in the project, outside of the partition for export, so that Developers can optimize the design with information about the clocks and PLL parameters. This technique provides more accurate timing requirements. Export the partition for the top-level design, without exporting any auxiliary components that you instantiate outside the partition you are exporting.

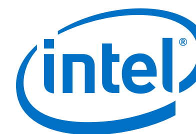
1.8.1.1. Prepare a Design Partition for Project Integration

Follow these steps to prepare a lower-level design partition for integration with the top-level project:

1. Obtain a copy of the top-level project, or create a new project with the same assignments and constraints as the top-level project. Ensure that your partition uses only the resources that the team lead allocates.
2. For each design partition that is incomplete in the top-level project, set the **Empty** option to **Yes** in the Design Partitions Window. This setting creates an empty partition for later development. For the Compiler to elaborate this partition, you must provide at least the port definitions and any parameters or generics passed to the RTL.
3. When the lower-level design partition is complete, follow the procedure in Step 2: Developer: Compile and Export the Core Partition on page 19. The project lead can now reuse the partition in the top-level project.

1.9. Bottom-Up Design Considerations

Consider the following recommendations and limitations when using a bottom-up design methodology



Recommendations and Limitations

Table 8. Bottom-Up Design Recommendations and Limitations

Recommendation/ Limitation	Description
Recommendation 1	Define Logic Lock constraints that are **Reserved**, **Core-Only**, **Fixed/Locked**, with a specified **Routing Region**. While exporting partitions from a different project with a different top-level, generate the partitions with non-overlapping Logic Lock routing regions by setting the routing region to **Fixed with expansion** of 0.
Limitation 1	If you compile two partitions, in two different projects, with `top_level_1.sv` and `top_level_2.sv`, and reuse the partitions in a third project with `top_level_3.sv`, the Compiler cannot support two partitions with overlapping row clock regions. Apply Logic Lock region constraints in the Developer project to avoid two partitions occupying the same row clock region in the Consumer project. For example: 1. From the Consumer project, determine the approximate placement of the two partitions. Choose the Logic Lock constraints for the two partitions such that there is no overlap of the row clock region. 2. In the Developer project with `top_level_1.sv`, apply Logic Lock region constraints that the Consumer identifies for the first partition, followed by compilation and export of the partition at final snapshot. 3. In the Developer project with `top_level_2.sv`, apply Logic Lock region constraints that the Consumer identifies for the second partition, followed by compilation and export of the partition at final snapshot. 4. When reusing the exported partitions in the consumer project with `top_level_3.sv`, the partitions maintain the placement defined in the Developer projects using non-overlapping Logic Lock constraints.
Limitation 2	System-level design errors may not become apparent until late in the design cycle, which can require additional design iterations to resolve.

Related Information

Bottom-Up Design Methodology Overview on page 9

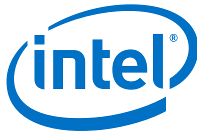
1.10. Debugging Block-Based Designs with the Signal Tap Logic Analyzer

The Intel Quartus Prime Pro Edition software supports debugging of block-based designs with the Signal Tap logic analyzer.

Signal Tap debugging of block-based designs requires specific preparation. For step-by-step details on debugging block-based designs with Signal Tap, refer to "Debugging Block-Based Designs with the Signal Tap Logic Analyzer" in *Intel Quartus Prime Pro Edition User Guide: Debug Tools*.

Related Information

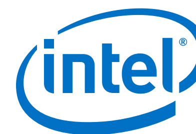
"Debugging Block-Based Designs with the Signal Tap Logic Analyzer," Intel Quartus Prime Pro Edition User Guide: Debug Tools



1.11. Block-Based Design Flows Revision History

Document Version	Intel Quartus Prime Version	Changes
2019.12.16	19.4.0	• Updated for cross-device snapshot reuse support and limits throughout.
2019.11.11	19.2.0	• Described Fast Preserve option in "Block-Based Design Terminology" and "Step 4: Add the Root Partition and Compile."
2019.07.15	19.2.0	• Changed default file export location from output_files to project directory. • Updated description of partition type GUI. • Updated Support for "Combined Incremental Block-Based Compilation and Design Block Reuse" table for latest supported combinations. • Added note about merging partitions to "Creating Design Partitions."
2018.10.01	18.1.0	• Removed reference to Placed snapshot from "Step 3: Compile and Export the Root Partition." Only Synthesized and Final snapshots are supported for design block reuse.
2018.09.24	18.1.0	• Reorganized order of topics in chapter. • Added the following new topics: — Viewing Quartus Database File Information — Reserved Core Entity Re-Binding — Incremental Block-Based Compilation Examples — Design Methodologies — Top-Down Design Methodology Overview — Bottom-Up Design Methodology Overview — Bottom-Up Design Recommendations and Limitations — Team-Based Design Methodology Overview — Incremental Timing Closure — Incremental Timing Closure Recommendations and Limitations — Design Abstraction • Replaced references to "periphery reuse core" with "reserved core" to reflect latest GUI. • Added description of | as root partition hierarchy path in Design Partitions Window. • Replaced details in "Debugging Block-Based Designs with the Signal Tap Logic Analyzer" section with link to *AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board*. • Minor wording and graphic updates throughout. • Removed references to unsupported Planned snapshot.
2018.05.07	18.0.0	• First release of chapter as part of stand-alone *Block-Based Design User Guide*. • Added footnote and links to known issues. • Updated all design flow steps to match current GUI. • Updated description of Include entity-bound SDC files option. • Updated statement defining parent to child partition attribute inheritance. • Added *Design Partition Settings* topic. • Added *Block-Based Design Terminology* topic. • Added *Preservation and Reuse with Compiler Snapshots* topic. • Added *Empty Partition Clock Source Preservation* topic.

continued...

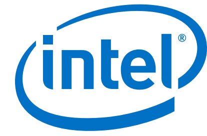


Document Version	Intel Quartus Prime Version	Changes
		• Added *Design Partitions Properties* table. • Added *Combining Incremental Block-Based Compilation and Design Block Reuse* topic. • Updated *Debugging Block-Based Designs* topic and linked to new Application Note.
2017.11.06	17.1.0	• Reorganization of introduction and Incremental Block-Based Compilation. • Added *Design Partitioning* section. • Added *Debugging Block-Based Designs* section. • Added *Use Empty Partitions to Reduce Compilation Time* topic. • Removed requirement to add `.psmf`, `.msf`, and `.sof` to Consumer project. • Added Intel Stratix 10 support, including information about bundling of `.sdc` with exported partitions for Intel Stratix 10 designs. • Documented changes to Design Partitions window, Export Design Partition dialog box, and Logic Lock Regions window. • Added reference to new Design Partition Planner. • Updated references to corresponding `.qsf` assignments. • Changed references from periphery reuse to root partition reuse. • Rebranded for latest Intel standards.
2017.05.08	17.0.0	• First public release.

1.12. Intel Quartus Prime Pro Edition User Guide: Block-Based Design Document Archive

If the table does not list a software version, the user guide for the previous software version applies.

Intel Quartus Prime Version	User Guide
19.2	Intel Quartus Prime Pro Edition User Guide: Block-Based Design
18.1	Intel Quartus Prime Pro Edition User Guide: Block-Based Design



A. Intel Quartus Prime Pro Edition User Guides

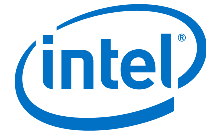
Refer to the following user guides for comprehensive information on all phases of the Intel Quartus Prime Pro Edition FPGA design flow.

Related Information

- Intel Quartus Prime Pro Edition User Guide: Getting Started
  Introduces the basic features, files, and design flow of the Intel Quartus Prime Pro Edition software, including managing Intel Quartus Prime Pro Edition projects and IP, initial design planning considerations, and project migration from previous software versions.
- Intel Quartus Prime Pro Edition User Guide: Platform Designer
  Describes creating and optimizing systems using Platform Designer, a system integration tool that simplifies integrating customized IP cores in your project. Platform Designer automatically generates interconnect logic to connect intellectual property (IP) functions and subsystems.
- Intel Quartus Prime Pro Edition User Guide: Design Recommendations
  Describes best design practices for designing FPGAs with the Intel Quartus Prime Pro Edition software. HDL coding styles and synchronous design practices can significantly impact design performance. Following recommended HDL coding styles ensures that Intel Quartus Prime Pro Edition synthesis optimally implements your design in hardware.
- Intel Quartus Prime Pro Edition User Guide: Design Compilation
  Describes set up, running, and optimization for all stages of the Intel Quartus Prime Pro Edition Compiler. The Compiler synthesizes, places, and routes your design before generating a device programming file.
- Intel Quartus Prime Pro Edition User Guide: Design Optimization
  Describes Intel Quartus Prime Pro Edition settings, tools, and techniques that you can use to achieve the highest design performance in Intel FPGAs. Techniques include optimizing the design netlist, addressing critical chains that limit retiming and timing closure, optimizing device resource usage, device floorplanning, and implementing engineering change orders (ECOs).
- Intel Quartus Prime Pro Edition User Guide: Programmer
  Describes operation of the Intel Quartus Prime Pro Edition Programmer, which allows you to configure Intel FPGA devices, and program CPLD and configuration devices, via connection with an Intel FPGA download cable.
- Intel Quartus Prime Pro Edition User Guide: Block-Based Design
  Describes block-based design flows, also known as modular or hierarchical design flows. These advanced flows enable preservation of design blocks (or logic that comprises a hierarchical design instance) within a project, and reuse of design blocks in other projects.

Intel Corporation. All rights reserved. Agilex, Altera, Arria, Cyclone, Enpirion, Intel, the Intel logo, MAX, Nios, Quartus and Stratix words and logos are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.
*Other names and brands may be claimed as the property of others.

ISO 9001:2015 Registered



- Intel Quartus Prime Pro Edition User Guide: Partial Reconfiguration
  Describes Partial Reconfiguration, an advanced design flow that allows you to reconfigure a portion of the FPGA dynamically, while the remaining FPGA design continues to function. Define multiple personas for a particular design region, without impacting operation in other areas.
- Intel Quartus Prime Pro Edition User Guide: Third-party Simulation
  Describes RTL- and gate-level design simulation support for third-party simulation tools by Aldec*, Cadence*, Mentor Graphics*, and Synopsys that allow you to verify design behavior before device programming. Includes simulator support, simulation flows, and simulating Intel FPGA IP.
- Intel Quartus Prime Pro Edition User Guide: Third-party Synthesis
  Describes support for optional synthesis of your design in third-party synthesis tools by Mentor Graphics*, and Synopsys. Includes design flow steps, generated file descriptions, and synthesis guidelines.
- Intel Quartus Prime Pro Edition User Guide: Third-party Logic Equivalence Checking Tools
  Describes support for optional logic equivalence checking (LEC) of your design in third-party LEC tools by OneSpin*.
- Intel Quartus Prime Pro Edition User Guide: Debug Tools
  Describes a portfolio of Intel Quartus Prime Pro Edition in-system design debugging tools for real-time verification of your design. These tools provide visibility by routing (or "tapping") signals in your design to debugging logic. These tools include System Console, Signal Tap logic analyzer, Transceiver Toolkit, In-System Memory Content Editor, and In-System Sources and Probes Editor.
- Intel Quartus Prime Pro Edition User Guide: Timing Analyzer
  Explains basic static timing analysis principals and use of the Intel Quartus Prime Pro Edition Timing Analyzer, a powerful ASIC-style timing analysis tool that validates the timing performance of all logic in your design using an industry-standard constraint, analysis, and reporting methodology.
- Intel Quartus Prime Pro Edition User Guide: Power Analysis and Optimization
  Describes the Intel Quartus Prime Pro Edition Power Analysis tools that allow accurate estimation of device power consumption. Estimate the power consumption of a device to develop power budgets and design power supplies, voltage regulators, heat sink, and cooling systems.
- Intel Quartus Prime Pro Edition User Guide: Design Constraints
  Describes timing and logic constraints that influence how the Compiler implements your design, such as pin assignments, device options, logic options, and timing constraints. Use the Interface Planner to prototype interface implementations, plan clocks, and quickly define a legal device floorplan. Use the Pin Planner to visualize, modify, and validate all I/O assignments in a graphical representation of the target device.
- Intel Quartus Prime Pro Edition User Guide: PCB Design Tools
  Describes support for optional third-party PCB design tools by Mentor Graphics* and Cadence*. Also includes information about signal integrity analysis and simulations with HSPICE and IBIS Models.
- Intel Quartus Prime Pro Edition User Guide: Scripting
  Describes use of Tcl and command line scripts to control the Intel Quartus Prime Pro Edition software and to perform a wide range of functions, such as managing projects, specifying constraints, running compilation or timing analysis, or generating reports.