Block-Based Design User Guide

Intel® Quartus® Prime Pro Edition

Updated for Intel® Quartus® Prime Design Suite: 18.0
## Contents

1. Block-Based Design Flows ................................................................. 3  
   1.1. Block-Based Design Terminology ................................................... 3  
   1.2. Design Block Reuse Overview ...................................................... 4  
      1.2.1. Design Block Reuse Examples ................................................. 5  
   1.3. Incremental Block-Based Compilation Overview ............................ 7  
   1.4. Preservation and Reuse with Compiler Snapshots.......................... 8  
   1.5. Design Partitioning ..................................................................... 8  
      1.5.1. Planning Design Partitions ......................................................... 9  
      1.5.2. Creating Design Partitions ....................................................... 12  
   1.6. Design Block Reuse Flows ............................................................ 14  
      1.6.1. Reusing Core Partitions .......................................................... 14  
      1.6.2. Reusing Root Partitions ......................................................... 17  
   1.7. Incremental Block-Based Compilation Flow .................................... 20  
      1.7.1. Creating Empty Partitions ....................................................... 21  
   1.8. Top-Down, Bottom-Up, and Team-Based Design Flows .................. 22  
      1.8.1. Combining Top-Down and Bottom-Up Design ........................... 23  
   1.9. Combining Incremental Block-Based Compilation and Design Block Reuse 25  
   1.10. Debugging Block-Based Designs Using Signal Tap ......................... 26  
      1.10.1. Signal Tap with Core Partition Reuse ..................................... 27  
      1.10.2. Signal Tap with Root Partition Reuse ...................................... 29  
   1.11. Block-Based Design Flows Revision History .................................. 33  

A. Intel Quartus Prime Pro Edition User Guides ..................................... 34
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 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 interface, 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>
<thead>
<tr>
<th>Table 1. Block-Based Design Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
</tr>
<tr>
<td>Black Box File</td>
</tr>
<tr>
<td>Block</td>
</tr>
<tr>
<td>Consumer</td>
</tr>
<tr>
<td>Core Partition</td>
</tr>
<tr>
<td>Design Partition</td>
</tr>
</tbody>
</table>

(1) This feature has some known limitations. Refer to Why I can't compile Intel Stratix 10 partitions exported from another project with a different top level and Internal Error: Sub-system: PTI, File: /quartus/tsm/pti/pti_tdb_builder.cpp.
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>A Developer creates and exports a design partition as a .qdb for use in a Consumer project.</td>
</tr>
<tr>
<td>Floorplanning</td>
<td>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.</td>
</tr>
<tr>
<td>Logic Lock Region Constraints</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>• <strong>Reserved</strong>—prevents the Fitter from placing non-member logic within the region.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Core-Only</strong>—applies the constraint only to core logic in the region, and does not include periphery logic in the region.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Routing Region</strong>—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.</td>
</tr>
<tr>
<td></td>
<td>• <strong>Size/State</strong>—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.</td>
</tr>
<tr>
<td>Preservation</td>
<td>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.</td>
</tr>
<tr>
<td>Project</td>
<td>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.</td>
</tr>
<tr>
<td>Root Partition</td>
<td>The Intel Quartus Prime software automatically creates a top-level root partition for each project. This 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 periphery reuse core) by a Consumer.</td>
</tr>
<tr>
<td>Snapshot</td>
<td>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.</td>
</tr>
</tbody>
</table>

**Related Information**

AN 839: Design Block Reuse Tutorial for Intel Arria® 10 FPGA Development Board

**1.2. 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. You can share one of the following compilation snapshots for a partition across projects or with other designers:

- Synthesized snapshot
- Placed 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. To avoid resource conflicts, floorplanning is essential when reusing placed or routed partitions.

### 1.2.1. 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.

#### 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.

---

(2) 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 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 a development kit that can be used by multiple Developers and projects.

Figure 3. Root Partition Reuse Example

In root partition reuse, each project must target the same Intel FPGA part number, 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 periphery reuse core partition that contains custom logic changes between Consumer projects.
1.3. Incremental Block-Based Compilation Overview

Incremental block-based compilation enables you to close timing for design blocks incrementally. You can preserve the compiled results for specific design partitions, while at the same time adding and compiling new 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 preserve only core partitions. You can preserve the core partitions at any of following compilation stages:

- Synthesized
- Final

When you preserve the compilation results for a partition, the snapshot remains unchanged for subsequent compilations. The preserved partition becomes the source for each subsequent compilation.
As part of this flow, you can define empty partitions to represent parts of your design that are incomplete or missing. Setting a partition to **Empty** reduces the total compilation time because the Compiler does not process design logic associated with an empty partition.

### 1.4. Preservation and Reuse with Compiler Snapshots

The Compiler generates a snapshot of compilation results, for each partition, at each stage of compilation. You can preserve partitions after Synthesis or after the Final stage of the Fitter. You can reuse partitions after Synthesis, Place, or Final stage. 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**

<table>
<thead>
<tr>
<th>Snapshot</th>
<th>Description</th>
<th>Preserve/Reuse</th>
</tr>
</thead>
</table>
| 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. |
| Placed    | Preserves device resources. Preserves placement through subsequent placement stages. | • Applies to .qdb files only.  
• Routing is not complete and is likely to change.  
• Intel Arria 10 High-Speed/Low-Power Tile settings may change.  
• Retiming may change Intel Stratix® 10 register locations. |
| 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. |

### 1.5. 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.

By default, every Intel Quartus Prime project includes a single, root partition. 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 periphery reuse 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, and the child partition inherits any preservation attribute of the parent. The partition creates a logical boundary that prevents merging or optimizing between partitions. The **Design Partitions in Design Hierarchy** diagram illustrates design partition relationships and boundaries.
**1.5.1. Planning Design Partitions**

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.
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 Creating Design Partitions on page 12 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 floorplanned 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.

1.5.1.1. 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.

1.5.1.2. 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.

1.5.2. 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, click Design Partition ➤ Set as Design Partition. 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**

<table>
<thead>
<tr>
<th>Partition Name</th>
<th>Hierarchy Path</th>
<th>Type</th>
<th>Preservation Level</th>
<th>Empty Partition Database File</th>
<th>Entity Re-binding</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;new&gt;&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>root_partition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>speed_ch</td>
<td>speed</td>
<td>Default</td>
<td>Not Set</td>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

**Table 3. Design Partition Settings**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition Name</td>
<td>Specifies the partition name. Each partition name in the project must be unique and consist of only alphanumeric characters.</td>
</tr>
<tr>
<td>Hierarchy Path</td>
<td>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. Double-click &lt;&lt;new&gt;&gt; in the Design Partitions Window to open the Create New Partition dialog box.</td>
</tr>
</tbody>
</table>
| 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. Select this option when defining a core partition for export.  
  - Reconfigurable—Identifies a reconfigurable partition in a Partial Reconfiguration flow. Specify the Reconfigurable type to preserve synthesis results, while allowing reft of the partition in the PR flow.  
  - Periphery Reuse Core—Identifies a partition in a block-based design flow that is reserved for core development when reusing the device periphery (that is, the root partition). |
| 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 Periphery Reuse 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, placed, or final). Assign the .qdb to a partition to reuse those results in another context. |
| Entity Re-binding       | Specifies (binds) the entity name for the PR partition you are changing in the current implementation revision. This setting specifies the entity that replaces the default persona in each implementation revision in the PR flow. |
| Color                   | Specifies the color coding of the partition in the Chip Planner and Design Partition Planner displays. |
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.

![Design Partition Window Compilation View Tab](image)

### 1.6. 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 and reusing the block in another project. Effective design block reuse requires careful 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, placed, 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, placed, or final snapshots 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.

#### 1.6.1. 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 only makes changes to the partition for the compilation stages that occur after the stage that the .qdb preserves.
The following steps describe the core partition reuse flow in detail.

**1.6.1.1. Step 1: Developer: Create a Core 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. Follow the steps in Creating Design Partitions on page 12 to define a core partition. Select Default for the partition Type.

**1.6.1.2. Step 2: Developer: Compile and Export a Core Partition**

This step describes generating a final snapshot for export. Following compilation, you can export a partition as a .qdb at the synthesized, placed, or final snapshot. You can then reuse the core partition in the same project or in another project.

To compile and 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.

4. Confirm the **File name** for the **Partition Database File**, and then click **OK**.

**Figure 11. Export Design Partition**

![Export Design Partition GUI]

The following command corresponds to partition export in the GUI:

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

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

1.6.1.4. 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 the core partition and assigns the .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 part number 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 12 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**.

1.6.2. 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.
1.6.2.1. Step 1: Developer: Create a Periphery Reuse Core Partition

To export and reuse the root partition, the Developer creates a periphery reuse 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 periphery reuse core partition:
1. Follow the steps in Creating Design Partitions on page 12 to create a periphery reuse core partition.
2. When defining the design partition, select Periphery Reuse Core for the partition Type. Ensure that all other partition options are set to the default values.

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

To reserve core resources in a Consumer project for the periphery reuse 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 area 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 develops. For projects with multiple core partitions, constrain each partition in a non-overlapping Logic Lock routing region.
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 13. Logic Lock Regions Window**

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.

**1.6.2.3. Step 3: Developer: Compile and Export the Root Partition**

After compilation, the Developer exports the root partition at the synthesized, placed, 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, placed, 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 periphery reuse core to the Consumer.
1.6.2.4. 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 periphery reuse 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 periphery reuse 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. Assign the exported root partition .qdb in the Consumer project. You can specify the root partition .qdb on the General page of the Settings dialog box (Assignments ➤ Settings ➤ General), or in the Partition Database File option in the Design Partitions Window (Assignments ➤ Design Partitions Window).

Figure 14. General Page of Settings Dialog Box

<table>
<thead>
<tr>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Top-level entity:</strong></td>
</tr>
<tr>
<td><strong>This project uses a Partition Database (qdb) file for the root partition:</strong></td>
</tr>
<tr>
<td><strong>Root Partition Database file:</strong></td>
</tr>
<tr>
<td><strong>Revision type:</strong></td>
</tr>
<tr>
<td><strong>Recently selected top-level entities:</strong></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
</tr>
</tbody>
</table>

4. The Consumer adds RTL and .sdc constraints for the periphery reuse core partition.
5. To run all compilation stages, click Processing ➤ Start Compilation. The Compiler implements the reused root partition and constraints.

1.7. Incremental Block-Based Compilation Flow

The incremental block-based compilation flow allows you to iteratively preserve partitions as they meet your design requirements.
After initial project setup, you define the design periphery in the top-level hierarchy of the design. Next, create a design hierarchy that allows you to create the core partitions that your design requires.

After you define the initial project structure and add source design files, run **Analysis & Elaboration** to display the project **Hierarchy** in the Project Navigator and begin defining partitions. Following design compilation, analyze partitions for specific results, such as timing closure, or resource utilization. When the partition meets the design requirements, you can preserve the partitions at the synthesized or final snapshot, depending on the requirements of the design.

### 1.7.1. Creating Empty Partitions

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. If you need to preserve the contents of a partition, export the `.qdb` for the partition before setting the partition to **Empty**. Then, you can assign a `.qdb` to the partition when you no longer require the partition to be empty.
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 Creating Design Partitions on page 12 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>
```

### 1.7.1.1. 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

### 1.8. Top-Down, Bottom-Up, and Team-Based Design Flows

In a top-down design flow, 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 lower-level design blocks separately, but one team lead manages the implementation project for the entire design.

In a bottom-up design flow, 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 flow, individual developers or IP providers can complete the placement and routing optimization of their design in separate projects, and then reuse each lower-level block in one top-level project. These methodologies 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.

Each developer should use a copy of the same project to ensure a consistent view of the top-level design framework. When another developer provides and optimizes timing-critical portions of the design, each developer must have the complete top-level design framework to maintain timing closure and achieve the best results during integration.

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.

### 1.8.1. Combining Top-Down and Bottom-Up Design

You can use elements of both top-down and bottom-up design methodologies together to implement a successful team-based design flow.

A top-level design can include one or more partitions that different Developers or IP providers create and optimize, as well as partitions for incremental block-based design compilation. In a team-based environment, portions of your design may initially be undefined with the expectation of developing those portions later. The team lead creates empty partitions in the top-level design for any incomplete partitions.
Developers or IP providers can then create and verify HDL code separately, and the team lead later integrates the code into the single Consumer project. For easiest integration of partitions, add the completed partitions to the design incrementally, and confine these optimizations to the top-level design.

The Developer or IP provider can also work on a copy of the same top-level project. The Developer or IP provider can create a partition for their respective design block, compile the design, and then export the partition. The team lead then integrates each design block as a design partition into the top-level design. To simplify full design optimization by allowing 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.

1.8.1.1. 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 16 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 12 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 18 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.

1.8.1.2. Preparing 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 the design only uses 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 procedures in Step 2: Developer: Compile and Export a Core Partition on page 15. The project lead can now reuse the partition in the top-level project.

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

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

<table>
<thead>
<tr>
<th>Supported Combinations</th>
<th>Unsupported Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Export of a partition as a .qdb, reuse of the .qdb, and preservation of the child. The reused partition .qdb file and the child's preservation level must be the same. Otherwise, this combination is not supported.</td>
<td>• Export and reuse of .qdb, combined with preservation of the child, when the child's preservation level is different from the partition .qdb.</td>
</tr>
<tr>
<td>• 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 different from the partition preservation level.</td>
<td>• Reuse of a child .qdb and the parent is Empty.</td>
</tr>
<tr>
<td>• 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.</td>
<td>• Reuse of a .qdb and the child is Empty.</td>
</tr>
<tr>
<td></td>
<td>• Preservation of a parent partition and the child is Empty.</td>
</tr>
<tr>
<td></td>
<td>• Preservation of a parent partition and the child partition and the parent preservation level is higher than child's preservation level.</td>
</tr>
</tbody>
</table>

### 1.10. Debugging Block-Based Designs Using Signal Tap

You can use the Signal Tap logic analyzer to debug block-based designs. The following section describes Signal Tap debugging of designs that include reusable blocks.

When reusing core partitions, the Developer creates partition boundary ports for each point that you want Signal Tap to debug. All boundary ports that the Developer creates are then available in the Consumer project for debugging.

When reusing root partitions, the Developer and Consumer use a JTAG debug bridge to extend Signal Tap debugging into the core partition. The Consumer of a root partition can then use Signal Tap to debug in the periphery reuse core by connecting to the debug bridge.

To use the debug bridge, you must instantiate the SLD JTAG Bridge Agent Intel FPGA IP and SLD JTAG Bridge Host Intel FPGA IP pair for each periphery reuse core boundary in your design. You instantiate the SLD JTAG Bridge Agent IP in the parent partition, and the SLD JTAG Bridge Host IP in the child partition. You can then connect to a Signal Tap instance through the bridge.
Note: For step-by-step block-based design debugging instructions, refer to AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board.

Related Information
- AN 847: Signal Tap Tutorial with Design Block Reuse for Intel Arria 10 FPGA Development Board
- Instantiating the SLD JTAG Bridge Agent IP
- Instantiating the SLD JTAG Bridge Host IP

1.10.1. Signal Tap with Core Partition Reuse

The Intel Quartus Prime software supports different methods for using Signal Tap to debug core partitions, depending on usage of the partition’s .qdb file. The following sections describe each core partition debugging method.

1.10.1.1. Using HDL Signal Tap Instances

You can instantiate HDL Signal Tap instances in reusable core partitions without any additional steps. Simply instantiate the Signal Tap component in the HDL of the Developer and Consumer projects and compile.

When using HDL Signal Tap instances, the Signal Tap instance is part of the project and the exported partition. When you export a .qdb file, the file includes any Signal Tap HDL instances, unless you remove them, recompile the core, and re-export.

1.10.1.2. Partition Boundary Ports

The core partition Developer must identify and expose the potential Signal Tap points of the partition to the Consumer by defining partition boundary ports. The Consumer can then use the ports that the Developer defines for Signal Tap debugging in the Consumer project. The following figure shows two new boundary ports: Module_A|Reg_C and Module_A|Module_B|Reg_C

Figure 17. Adding Ports at Partition Boundary

The partition boundary ports make a connection to the logic and tunnel the ports to the top-level partition. A new boundary port is then available at every child partition boundary. You can use any boundary ports that you create in the same way as any output for the purposes of debugging.
To use Signal Tap with a preserved partition, the Developer must create additional ports at the partition boundary, and connect the logic for probing to the new ports. Manual tunneling of connections through layers of RTL requires design changes, and can create issues with different versions of code. To avoid these complexities, use the **Create Partition Boundary Ports** assignment in the Assignment Editor (**Assignments ➤ Assignment Editor**) to automatically create the additional boundary ports connected to the logic that you want to debug. Using the boundary ports does not require RTL changes.

### 1.10.1.2.1. Defining Partition Boundary Ports

Follow these steps to define partition boundary ports in a Developer project for subsequent Signal Tap debugging in a Consumer project:

1. Create the Developer project with the core partition for export.
2. Elaborate the design hierarchy and create a design partition, as Creating Design Partitions on page 12 describes.
3. To connect logic to a partition boundary port, click **Assignments ➤ Assignment Editor**, and then assign **Create Partition Boundary Ports** to one or more ports. When you assign a bus, the assignment applies to the root name of the debug port, with each bit enumerated. When the Consumer synthesizes the reused partition, all valid ports with the **Create Partition Boundary Ports** are visible in the Consumer project.
4. Compile the design. Following synthesis, verify the partition boundary ports in the Create Partition Boundary Ports report. This report generates in the In-System Debugging folder under Synthesis reports.
5. To export the partition from the Developer project, click **Project ➤ Export Design Partition**.
6. From the original core partition source file, create a black box port definitions file that contains only port declarations. Add the partition boundary ports to the black box file.
7. Add the partition’s `.qdb`, black box file, and any other data you require to the Consumer project.

### 1.10.1.2.2. Connecting to Debug Ports in a Reused Core Partition

After defining partition boundary ports, follow these steps to connect to debug ports from a `.qdb` in a Consumer project:

1. Define the boundary ports in the Developer project, as Defining Partition Boundary Ports on page 28 describes.
2. Export the core partition from the Developer project, as Step 2: Developer: Compile and Export a Core Partition on page 15 describes.
3. Create the Consumer project and add the black box file the Developer provides. The black box file must include the boundary ports from the Developer project. If the ports are not present in the ports list, the Consumer cannot add Signal Tap to the ports.
4. Elaborate the design hierarchy.
5. Click **Assignments ➤ Design Partitions Window**, and then define one or more partitions for the hierarchy branches that use the `.qdb` files. Specify the `.qdb` for the **Partition Database File** option for each partition that uses them, as Creating Design Partitions on page 12 describes.
6. Click **File ➤ New ➤** and then define a new Signal Tap Logic Analyzer file (.stp).

7. In the Signal Tap logic analyzer, use the Node Finder to locate the debug ports with the Signal Tap Pre-synthesis ports filter.

8. Add the clock source to your Signal Tap file.

9. Compile the design.

10. In the Compilation Report, under **Synthesis**, view the In-System Debugging report to verify the connection of the ports.

11. Debug your design in the Signal Tap logic analyzer.

### 1.10.1.3. Debugging with the Synthesis Snapshot

If the .qdb you use is for the synthesis snapshot, adding pre-synthesis Signal Tap nodes is not possible because that requires resynthesis of the core. However, you can add post-fit Signal Tap nodes, because the Fitter can connect and route the post-fit nodes.

To tap the post-fit nodes, the Consumer must do the following:

1. Add the core partition to a Consumer project, as [Step 4: Consumer: Add the Core Partition and Compile](#) on page 17 describes.

2. Compile the partition through the Fitter stage in the Consumer project.

3. Add Signal Tap in the Consumer design and add the post-fit Signal Tap nodes.

4. To recompile the design from the Place stage, click **Processing ➤ Start ➤ Start Fitter (Place)**. The Fitter attaches the Signal Tap nodes to the existing synthesized nodes.

### 1.10.1.4. Debugging with the Placed or Final Snapshot

If the partition you reuse contains the placed or final snapshot, adding new Signal Tap nodes is not possible. This limitation requires the partition Developer define additional boundary ports, before exporting the partition, that enable subsequent connection to the Signal Tap nodes. The partition Consumer then taps these ports with Signal Tap for debugging in the Consumer project.

### 1.10.2. Signal Tap with Root Partition Reuse

Root partition reuse with Signal Tap requires a JTAG debug bridge to extend Signal Tap debugging from the root partition into the periphery reuse core partition. The Consumer of a root partition then adds and uses Signal Tap to debug into the periphery reuse core partition by connecting to the debug bridge. The debug bridge allows independent debugging in the root and core partitions. The logic in the root and core partitions are isolated and require separate .stp files in each partition. You implement the JTAG debug bridge by use of the SLD JTAG Bridge Host Intel FPGA IP (SLD JTAG Bridge Host) and the SLD JTAG Bridge Agent Intel FPGA IP (SLD JTAG Bridge Agent).
In root partition reuse, you must treat each partition individually at the time of integration. Each instance of Signal Tap can only connect within the partition that the instance resides. Therefore, the root partition and periphery reuse core partition each require separate Signal Tap files in this flow.

To enable the Consumer to debug the root partition, the Developer must include Signal Tap instances in the root partition. To enable the Consumer to debug the periphery reuse core partition, the Developer must include the SLD JTAG Bridge Agent in the root partition. Then, any Signal Tap or SLD JTAG Bridge that the Developer instantiates in the root partition is exported, and is available for use in the Consumer project. If you do not want to expose Signal Tap points in the exported root partition, remove the Signal Tap file and recompile the Developer project prior to exporting the .qdb.

1.10.2.1. SLD JTAG Bridge

The SLD JTAG Bridge components include the SLD JTAG Bridge Agent Intel FPGA IP and SLD JTAG Bridge Host Intel FPGA IP. You instantiate an SLD JTAG Bridge Agent IP and SLD JTAG Bridge Host IP pair at the boundary of each periphery reuse core partition that requires debug.

The SLD JTAG Bridge Agent IP and SLD JTAG Bridge Host IP provide the following functionality:
• SLD JTAG Bridge Agent IP—enables debug of partial reconfiguration or periphery reuse core partitions, by extending the JTAG debug fabric from a higher-level partition to a partial reconfiguration or periphery reuse core partition containing the SLD JTAG Bridge Host IP. Instantiate the SLD JTAG Bridge Agent IP in the higher-level partition and connect the interface to an SLD JTAG Bridge Host IP in the child partition.

• SLD JTAG Bridge Host IP—enables debug of partial reconfiguration or periphery reuse core partitions by connecting the JTAG debug fabric in a partial reconfiguration or periphery reuse core partition to a higher-level partition containing the SLD JTAG Bridge Agent IP. Instantiate the SLD JTAG Bridge Host IP in the child partition and connect the interface to an SLD JTAG Bridge Agent IP in the higher-level partition.

Figure 20. SLD JTAG Bridge Agent and SLD JTAG Bridge Host Configuration

1.10.2.1.1. SLD JTAG Bridge Index

The Intel Quartus Prime software supports multiple instances of the SLD JTAG Bridge in partitions and their children. The Compiler assigns an index number to distinguish each instance. The bridge index for the root partition is always None.

When configuring the Signal Tap logic analyzer for the root partition, set the Bridge Index value to None in the JTAG Chain Configuration window.

Figure 21. JTAG Chain Configuration Bridge Index
Following design synthesis, the Compilation Report lists the index numbers for the SLD JTAG Bridge Agents in the periphery reuse core partition. Open the Synthesis ➤ In-System Debugging ➤ JTAG Bridge Instance Agent Information report for details about how the bridge indexes are enumerated. The reports shows the hierarchy path and the associated index. The Developer must provide this information to the Consumer, so the Consumer understands the index mapping.

1.10.2.2. Adding the SLD JTAG Bridge to the Root Partition

Follow these steps to instantiate Signal Tap in the root partition:

1. Create an Intel Quartus Prime project that is setup for root partition reuse, as Reusing Root Partitions on page 17 describes.

2. From the IP Catalog, select, parameterize, and instantiate the SLD JTAG Bridge Agent IP in the parent module of the core partition. Similarly, instantiate the SLD JTAG Bridge Host IP in the periphery reuse core partition. To display IP Catalog, click Tools ➤ IP Catalog.

3. Export the root partition, as Step 3: Developer: Compile and Export the Root Partition on page 19 describes. After the Developer creates the .qdb, the Consumer can reuse the exported .qdb file and optionally add Signal Tap to the core partition.

4. Compile and debug the Developer project. To run full compilation, click Processing ➤ Start Compilation.

1.10.2.3. Adding Signal Tap to the Periphery Reuse Core Partition

Signal Tap debugging in the periphery reuse core partition requires the SLD JTAG Bridge Agent IP in the root partition, and the SLD JTAG Bridge Host IP in the periphery reuse core partitions. The periphery reuse core partition Consumer instantiates SLD
JTAG Bridge Host IP to communicate with the SLD JTAG Bridge Agent IP that the
Developer provides in the root partition. After design synthesis, the Consumer can
insert pre-synthesis Signal Tap points in the Consumer project periphery reuse core
partition.

1. Assign the exported root partition .qdb in the Consumer project, as Step 4:
Consumer: Add the Root Partition and Compile on page 20 describes.
2. In the periphery reuse core partition, instantiate the SLD JTAG Bridge Host IP core
from the IP Catalog.
3. To run synthesis, click **Processing ➤ Start ➤ Start Analysis & Synthesis**.
4. Create a new Signal Tap file and add the pre-synthesis Signal Tap points from the
core partition.
   
   **Note:** Adding new Signal Tap points to a reused root partition is invalid. Any new
   Signal Tap points that you add to the root partition are unusable in the core
   partition’s Signal Tap file.
5. To run full compilation, click **Processing ➤ Start Compilation**.

### 1.11. Block-Based Design Flows Revision History

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 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.  
|                   |                             | • 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.  

Block-Based Design User Guide Intel® Quartus® Prime Pro Edition

33
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

- **Getting Started User Guide**
  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.

- **Platform Designer User Guide**
  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.

- **Design Recommendations User Guide**
  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.

- **Compiler User Guide**
  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.

- **Design Optimization User Guide**
  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, and optimization of device resource usage.

- **Programmer User Guide**
  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.

- **Block-Based Design User Guide**
  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.
• Partial Reconfiguration User Guide
  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.

• Third-party Simulation User Guide
  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.

• Third-party Synthesis User Guide
  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.

• Debug Tools User Guide
  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.

• Timing Analyzer User Guide
  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.

• Power Analysis and Optimization User Guide
  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.

• Design Constraints User Guide
  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.

• PCB Design Tools User Guide
  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.

• Scripting User Guide
  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.