15. Best Practices for Incremental Compilation Partitions and Floorplan Assignments

This chapter provides guidelines to help you partition your design to take advantage of Quartus® II incremental compilation, and to help you create a design floorplan using LogicLock™ regions when they are recommended to support the compilation flow.

The Quartus II incremental compilation feature allows you to partition a design, compile partitions separately, and reuse results for unchanged partitions. Incremental compilation provides the following benefits:

- Reduces compilation times by an average of 75% for large design changes
- Preserves performance for unchanged design blocks
- Provides repeatable results and reduces the number of compilations
- Enables team-based design flows

For more information about the incremental compilation feature and application examples, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook. For feature support, refer to About Incremental Compilation in Quartus II Help.

This document contains the following sections:

- “Overview: Incremental Compilation” on page 15–2
- “Design Flows Using Incremental Compilation” on page 15–3
- “Why Plan Partitions and Floorplan Assignments?” on page 15–6
- “General Partitioning Guidelines” on page 15–8
- “Design Partition Guidelines” on page 15–10
- “Checking Partition Quality” on page 15–30
- “Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery” on page 15–36
- “Introduction to Design Floorplans” on page 15–40
- “Design Floorplan Placement Guidelines” on page 15–43
- “Checking Floorplan Quality” on page 15–49
- “Recommended Design Flows and Application Examples” on page 15–51
Overview: Incremental Compilation

Quartus II incremental compilation is an optional compilation flow that enhances the default Quartus II compilation. If you do not partition your design for incremental compilation, your design is compiled using the default “flat” compilation flow. This section provides an overview of the incremental compilation flow and highlights best practices.

To prepare your design for incremental compilation, you first determine which logical hierarchy boundaries should be defined as separate partitions in your design, and ensure your design hierarchy and source code is set up to support this partitioning. You can then create design partition assignments in the Quartus II software to specify which hierarchy blocks are compiled independently as partitions (including empty partitions for missing or incomplete logic blocks).

During compilation, Quartus II Analysis & Synthesis and the Fitter create separate netlists for each partition. Netlists are internal post-synthesis and post-fit database representations of your design.

In subsequent compilations, you can select which netlist to preserve for each partition. You can either reuse the synthesis or fitting netlist, or instruct the Quartus II software to resynthesize the source files. You can also use compilation results exported from another Quartus II project.

When you make changes to your design, the Quartus II software recompiles only the designated partitions and merges the new compilation results with existing netlists for other partitions, according to the degree of results preservation you set with the netlist for each design partition.

In some cases, as described in “Introduction to Design Floorplans” on page 15–40, Altera recommends that you create a design floorplan with placement assignments to constrain parts of the design to specific regions of the device.

For step-by-step information about using incremental compilation to recompile only changed parts of your design, refer to Using the Incremental Compilation Design Flow in Quartus II Help.

You must use incremental compilation in conjunction with the partial reconfiguration (PR) feature for Stratix® V device families. Partial reconfiguration allows you to reconfigure a portion of the FPGA dynamically, while the remainder of the device continues to operate as intended.

For more information about partial reconfiguration, refer to the Design Planning for Partial Reconfiguration chapter in volume 1 of the Quartus II Handbook.
Recommendations for the Netlist Type

For subsequent compilations, you specify which post-compilation netlist you want to use with the netlist type for each partition.

Use the following general guidelines to set the netlist type for each partition:

- **Source File**—Use this setting to resynthesize the source code (with any new assignments, and replace any previous synthesis or Fitter results).
  - If you modify the design source, the software automatically resynthesizes the partitions with the appropriate netlist type, which makes the **Source File** setting optional in this case.
  - Most assignments do not trigger an automatic recompilation, so you must set the netlist type to **Source File** to compile the source files with new assignments or constraints that affect synthesis.

- **Post-Synthesis** (default)—Use this setting to re-fit the design (with any new Fitter assignments), but preserve the synthesis results when the source files have not changed. If it is difficult to meet the required timing performance, you can use this setting to allow the Fitter the most flexibility in placement and routing. This setting does not reduce compilation time as much as the **Post-Fit** setting or preserve timing performance from the previous compilation.

- **Post-Fit**—Use this setting to preserve Fitter and performance results when the source files have not changed. This setting reduces compilation time the most, and preserves timing performance from the previous compilation.

- **Post-Fit with Fitter Preservation Level set to Placement**—Use the **Advanced Fitter Preservation Level** setting on the **Advanced** tab in the **Design Partition Properties** dialog box to allow more flexibility and find the best routing for all partitions given their placement.

The Quartus II software Rapid Recompile feature instructs the Compiler to reuse the compatible compilation results if most of the design has not changed since the last compilation. This feature reduces compilation time and preserves performance when there are small and isolated design changes within a partition, and works with all netlist type settings. With this feature, you do not have control over which parts of the design are recompiled; the Compiler determines which parts of the design must be recompiled. You can turn on the **Rapid Recompile** option in the Quartus II software on the **Incremental Compilation** page of the **Settings** dialog box.

Design Flows Using Incremental Compilation

The Quartus II incremental compilation feature supports various design flows. Your design flow affects the impact design partitions have on design optimization and the amount of design planning required to obtain optimal results.
In the standard incremental compilation flow, the top-level design is divided into partitions, which can be compiled and optimized together in one Quartus II project. If another team member or IP provider is developing source code for the top-level design, they can functionally verify their partition independently, and then simply provide the partition’s source code to the project lead for integration into the top-level design. If the project lead wants to compile the top-level design when source code is not yet complete for a partition, they can create an empty placeholder for the partition until the code is ready and added to the top-level design.

Compiling all design partitions in a single Quartus II project ensures that all design logic is compiled with a consistent set of assignments, and allows the software to perform global placement and routing optimizations. Compiling all design logic together is beneficial for FPGA design flows because all parts of the design must use the same shared set of device resources. Therefore, it is often easier to ensure good quality of results when partitions are developed within a single top-level Quartus II project.

In the team-based incremental compilation flow, you can design and optimize partitions by accessing the top-level project from a shared source control system or creating copies of the top-level Quartus II project framework. As development continues, designers export their partition so that the post-synthesis netlist or post-fitting results can be integrated into the top-level design.

If required for third-party IP delivery, or in cases where designers cannot access a shared or copied top-level project framework, you can create and compile a design partition logic in isolation and export a partition that is included in the top-level project. If this type of design flow is necessary, planning and rigorous design guidelines might be required to ensure that designers have a consistent view of project assignments and resource allocations. Therefore, developing partitions in completely separate Quartus II projects can be more challenging than having all source code within one project or developing design partitions within the same top-level project framework.

You can also combine design flows and use exported partitions only when it is necessary to support your design environment. For example, if the top-level design includes one or more design blocks that will be optimized by remote designers or IP providers, you can integrate those blocks into the reserved partitions in top-level design when the code is complete, but also have other partitions that will be developed within the top-level design.

If any partitions are developed independently, the project lead must ensure that top-level constraints (such as timing constraints, any relevant floorplan or pin assignments, and optimization settings) are consistent with those used by all designers working independently.

**Project Management in Team-Based Design Flows**

If possible, each team member should work within the same top-level project framework. Using the same project framework amongst team members ensures that designers have the settings and constraints needed for their partition and allows designers to analyze how their design block interacts with other partitions in the top-level design.
In a team-based environment where designers have access to the project through source control software, each designer can use project files as read-only and develop their partition within the source control system. As designers check in their completed partitions, other team members can see how their partitions interact. If designers do not have access to a source control system, the project lead can provide each designer with a copy of the top-level project framework to use as they develop their partitions. In both cases, each designer exports their completed design as a partition, and then the project lead integrates the partition into the top-level design. The project lead can choose to use only the post-synthesis netlist and rerun placement and routing, or to use the post-fitting results to preserve the placement and routing results from the other designer’s projects. Using post-synthesis partitions gives the Fitter the most flexibility and is likely to achieve a good result for all partitions, but if one partition has difficulty meeting timing, the designer can choose to preserve their successful fitting results.

Alternatively, designers can use their own Quartus II project for their independent design block. You might use this design flow if a designer, such as a third-party IP provider, does not have access to the entire top-level project framework. In this case, each designer must create their own project with all the relevant assignments and constraints. As mentioned at the beginning of this section, this type of design flow requires more planning and rigorous design guidelines. If the project lead plans to incorporate the post-fitting compilation results for the partition, this design flow requires especially careful planning to avoid resource conflicts.

The project lead also has the option to generate design partition scripts to manage resource and timing budgets in the top-level design when partitions are developed outside the top-level project framework. Scripts make it easier for designers of independent Quartus II projects to follow instructions from the project lead. The Quartus II design partition scripts feature creates Tcl scripts or .tcl files and makefiles that an independent designer can run to set up an independent Quartus II project.

For more information about how to generate design partition scripts, refer to Generating Design Partition Scripts for Project Management in Quartus II Help.

If designers create Quartus II assignments or timing constraints for their partitions, they must ensure that the constraints are integrated into the top-level design. If partition designers use the same top-level project framework (and design hierarchy), the constraints or Synopsys Design Constraints File (.sdc) can be easily copied or included in the top-level design. If partition designers use a separate Quartus II project with a different design hierarchy, they must ensure that constraints are applied to the appropriate level of hierarchy in the top-level design, and design the .sdc for easy delivery to the project lead, as described in “Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery” on page 15–36.

For more information about the different types of incremental design flows and example applications, as well as documented restrictions and limitations, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.
Why Plan Partitions and Floorplan Assignments?

Incremental design flows typically require more planning than flat compilations, and require you to be more rigorous about following good design practices. For example, you might need to structure your source code or design hierarchy to ensure that logic is grouped correctly for optimization. It is easier to implement the correct logic grouping early in the design cycle than to restructure the code later.

Planning involves setting up the design logic for partitioning and may also involve planning placement assignments to create a floorplan. Not all design flows require floorplan assignments. For more information, refer to “Introduction to Design Floorplans” on page 15–40. If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation easier. Poor partition or floorplan assignments can worsen design area utilization and performance and make timing closure more difficult.

As FPGA devices get larger and more complex, following good design practices become more important for all design flows. Adhering to recommended synchronous design practices makes designs more robust and easier to debug. Using an incremental compilation flow adds additional steps and requirements to your project, but can provide significant benefits in design productivity by preserving the performance of critical blocks and reducing compilation time.

Partition Boundaries and Optimization

The logical hierarchical boundaries between partitions are treated as hard boundaries for logic optimization (except for some limited cross-boundary optimizations) to allow the software to size and place each partition independently. Figure 15–1 shows the effects of partition boundaries during logic optimization.

You can use the Merge command in the Design Partitions window to combine hierarchical partitions into a single partition, as long as they share the same immediate parent partition. Merging partitions allows additional optimizations for partition I/O ports that connect between or feed more than one of the merged hierarchical design blocks.
When partitions are placed together, the Fitter can perform placement optimizations on the design as a whole to optimize the placement of cross-boundary paths. However, the Fitter can never perform logic optimizations such as physical synthesis across the partition boundary. If partitions are fit separately in different projects, or if some partitions use previous post-fitting results, the Fitter does not place and route the entire cross-boundary path at the same time and cannot fully optimize placement across the partition boundaries. Good design partitions can be placed independently because cross-partition paths are not the critical timing paths in the design.

There are possible timing performance utilization effects due to partitioning and creating a floorplan. Not all designs encounter these issues, but you should consider these effects if a flat version of your design is very close to meeting its timing requirements, or is close to using all the device resources, before adding partition or floorplan assignments:

- Partitions can increase resource utilization due to cross-boundary optimization limitations if the design does not follow partitioning guidelines. For more information, refer to “Design Partition Guidelines” on page 15–10. Floorplan assignments can also increase resource utilization because regions can lead to unused logic. If your device is full with the flat version of your design, you can focus on creating partitions and floorplan assignments for timing-critical or often-changing blocks to benefit most from incremental compilation. For more information, refer to “Checking Floorplan Quality” on page 15–49.

- Partitions and floorplan assignments might increase routing utilization compared to a flat design. If long compilation times are due to routing congestion, you might not be able to use the incremental flow to reduce compilation time. Review the Fitter messages to check how much time is spent during routing optimizations to determine the percentage of routing utilization. When routing is difficult, you can use incremental compilation to lock the routing for routing-critical blocks only (with other partitions empty), and then compile the rest of the design after the critical blocks meet their requirements.

- Partitions can reduce timing performance in some cases because of the optimization and resource effects described above, causing longer logic delays. Floorplan assignments restrict logic placement, which can make it more difficult for the Fitter to meet timing requirements. Use the guidelines in this chapter to reduce any effect on your design performance.

### Turning On Supported Cross-boundary Optimizations

You can improve the optimizations performed between design partitions by turning on the cross-boundary optimizations feature. You can select the optimizations as individual assignments for each partition. This allows the cross-boundary optimization feature to give you more control over the optimizations that work best for your design.

You can turn on the cross-boundary optimizations for your design partitions on the **Advanced** tab of the **Design Partition Properties** dialog box. Once you change the optimization settings, the Quartus II software recompiles your partition from source automatically. Cross-boundary optimizations include the following: propagate constants, propagate inversions on partition inputs, merge inputs fed by a common source, merge electrically equivalent bidirectional pins, absorb internal paths, and remove logic connected to dangling outputs.
Cross-boundary optimizations are implemented top-down from the parent partition into the child partition, but not vice-versa. The cross-boundary optimization feature cannot be used with partitions with multiple personas (partial reconfiguration partitions).

For more information about cross-boundary optimizations, refer to Design Partition Properties Dialog Box in Quartus II Help.

Although more partitions allow for a greater reduction in compilation time, consider limiting the number of partitions to prevent degradation in the quality of results. Creating good design partitions and good floorplan location assignments helps to improve the design resource utilization and timing performance results for cross-partition paths.

General Partitioning Guidelines

The first step in planning your design partitions is to organize your source code so that it supports good partition assignments. Although you can assign any hierarchical block of your design as a design partition or merge hierarchical blocks into the same partition, following the design guidelines presented in this section ensures better results.

Plan Design Hierarchy and Design Files

You begin the partitioning process by planning the design hierarchy. When you assign a hierarchical instance as a design partition, the partition includes the assigned instance and entities instantiated below that are not defined as separate partitions. You can use the Merge command in the Design Partitions window to combine hierarchical partitions into a single partition, as long as they have the same immediate parent partition.

When planning your design hierarchy, keep logic in the “leaves” of the hierarchy instead of having logic at the top-level of the design so that you can isolate partitions if required.

Create entities that can form partitions of approximately equal size. For example, do not instantiate small entities at the same hierarchy level, because it is more difficult to group them to form reasonably-sized partitions.

Create each entity in an independent file. The Quartus II software uses a file checksum to detect changes, and automatically recompiles a partition if its source file changes and its netlist type is set to either post-synthesis or post-fit. If the design entities for two partitions are defined in the same file, changes to the logic in one partition initiates recompilation for both partitions.

Design dependencies also affect which partitions are compiled when a source file changes. If two partitions rely on the same lower-level entity definition, changes in that lower-level entity affect both partitions. Commands such as VHDL use and Verilog HDL include create dependencies between files, so that changes to one file can trigger recompiations in all dependent files. Avoid these types of file dependencies if possible. The Partition Dependent Files report for each partition in the Analysis & Synthesis section of the Compilation report lists which files contribute to each partition.
Using Partitions with Third-Party Synthesis Tools

Incremental compilation works well with third-party synthesis tools in addition to Quartus II Integrated Synthesis. If you use a third-party synthesis tool, set up your tool to create a separate Verilog Quartus Mapping File (.vqm) or EDIF Input File (.edf) netlist for each hierarchical partition. In the Quartus II software, designate the top-level entity from each netlist as a design partition. The .vqm or .edf netlist file is treated as the source file for the partition in the Quartus II software.

For more information about incremental synthesis in third-party tools, refer to the documentation provided by your tool vendor, or the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.

Partition Design by Functionality and Block Size

Initially, you should partition your design along functional boundaries. In a top-level system block diagram, each block is often a natural design partition. Typically, each block of a system is relatively independent and has more signal interaction internally than interaction between blocks, which helps reduce optimizations between partition boundaries. Keeping functional blocks together means that synthesis and fitting can optimize related logic as a whole, which can lead to improved optimization.

Consider how many partitions you want to maintain in your design to determine the size of each partition. Your compilation time reduction goal is also a factor, because compiling small partitions is typically faster than compiling large partitions.

There is no minimum size for partitions; however, having too many partitions can reduce the quality of results by limiting optimization. Ensure that the design partitions are not too small. As a general guideline, each partition should contain more than approximately 2,000 logic elements (LEs) or adaptive logic modules (ALMs). If your design is incomplete when you partition the design, use previous designs to help estimate the size of each block.

Partition Design by Clock Domain and Timing Criticality

Consider which clock in your design feeds the logic in each partition. If possible, keep clock domains within one partition. When a clock signal is isolated to one partition, it reduces dependence on other partitions for timing optimization. Isolating a clock domain to one partition also allows better use of regional clock routing networks if the partition logic is constrained to one region of the design. Additionally, limiting the number of clocks within each partition simplifies the timing requirements for each partition during optimization. Use an appropriate subsystem to implement the required logic for any clock domain transfers (such as a synchronization circuit, dual-port RAM, or FIFO). You can include this logic inside the partition at one side of the transfer.

For more information about clock domains and their affect on partition design, refer to the Analyzing and Optimizing the Design Floorplan with the Chip Planner chapter in volume 2 of the Quartus II Handbook.
Try to isolate timing-critical logic from logic that you expect to easily meet timing requirements. Doing so allows you to preserve the satisfactory results for non-critical partitions and focus optimization iterations on only the timing-critical portions of the design to minimize compilation time.

**Consider What Is Changing**

When assigning partitions, you should consider what is changing in the design. Is there intellectual property (IP) or reused logic for which the source code will not change during future design iterations? If so, define the logic in its own partition so that you can compile one time and immediately preserve the results and not have to compile that part of the design again. Is logic being tuned or optimized, or are specifications changing for part of the design? If so, define changing logic in its own partition so that you can recompile only the changing part while the rest of the design remains unchanged.

As a general rule, create partitions to isolate logic that will change from logic that will not change. Partitioning a design in this way maximizes the preservation of unchanged logic and minimizes compilation time.

**Design Partition Guidelines**

Follow the partitioning guidelines presented in this section when you create or modify the HDL code for each design block that you might want to assign as a design partition. You do not need to follow all the recommendations exactly to achieve a good quality of results with the incremental compilation flow, but adhering to as many as possible maximizes your chances for success.

This section includes examples of the types of optimizations that are prevented by partition boundaries, and describes how you can structure or modify your partitions to avoid these limitations.

**Register Partition Inputs and Outputs**

Use registers at partition input and output connections that are potentially timing-critical. Registers minimize the delays on inter-partition paths and prevent the need for cross-boundary optimizations.
If every partition boundary has a register as shown in Figure 15–2, every register-to-register timing path between partitions includes only routing delay. Therefore, the timing paths between partitions are likely not timing-critical, and the Fitter can generally place each partition independently from other partitions. This advantage makes it easier to create floorplan location assignments for each separate partition, and is especially important for flows in which partitions are placed independently in separate Quartus II projects. Additionally, the partition boundary does not affect combinational logic optimization because each register-to-register logic path is contained within a single partition.

Figure 15–2. Registering Partition I/O

If a design cannot include both input and output registers for each partition due to latency or resource utilization concerns, choose to register one end of each connection. If you register every partition output, for example, the combinational logic that occurs in each cross-partition path is included in one partition so that it can be optimized together.

It is a good synchronous design practice to include registers for every output of a design block. Registered outputs ensure that the input timing performance for each design block is controlled exclusively within the destination logic block. For more information about I/O ports and registers for each partition, refer to “Partition Statistics Report” on page 15–34, and “Incremental Compilation Advisor” on page 15–49.

Minimize Cross-Partition-Boundary I/O

Minimize the number of I/O paths that cross between partition boundaries to keep logic paths within a single partition for optimization. Doing so makes partitions more independent for both logic and placement optimization.

This guideline is most important for timing-critical and high-speed connections between partitions, especially in cases where the input and output of each partition is not registered. Slow connections that are not timing-critical are acceptable because they should not impact the overall timing performance of the design. If there are timing-critical paths between partitions, rework or merge the partitions to avoid these inter-partition paths.
When dividing your design into partitions, consider the types of functions at the partition boundaries. Figure 15–3 shows an expansive function with more outputs than inputs in the left diagram, which makes a poor partition boundary, and, on the right side, a better place to assign the partition boundary that minimizes cross-partition I/Os. Adding registers to one or both sides of the cross-partition path in this example would further improve partition quality.

Another way to minimize connections between partitions is to avoid using combinational “glue logic” between partitions. You can often move the logic to the partition at one end of the connection to keep more logic paths within one partition. For example, the bottom diagram in Figure 15–4 includes a new level of hierarchy C defined as a partition instead of block B. Clearly, there are fewer I/O connections between partitions A and C than between partitions A and B.
Examine the Need for Logic Optimization Across Partitions

As discussed in “Partition Boundaries and Optimization” on page 15–6, partition boundaries prevent logic optimizations across partitions (except for some limited cross-boundary optimizations).

In some cases, especially if part of the design is complete or comes from another designer, the designer might not have followed these guidelines when the source code was created. These guidelines are not mandatory to implement an incremental compilation flow, but can improve the quality of results. If assigning a partition affects resource utilization or timing performance of a design block as compared to the flat design, it might be due to one of the issues described in this section. Many of the examples suggest simple changes to your partition definitions or hierarchy to move the partition boundary to improve your results.

The following guidelines ensure that your design does not require logic optimization across partition boundaries:

■ “Keep Logic in the Same Partition for Optimization and Merging” on page 15–13
■ “Keep Constants in the Same Partition as Logic” on page 15–15
■ “Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together” on page 15–16
■ “Invert Clocks in Destination Partitions” on page 15–17
■ “Connect I/O Pin Directly to I/O Register for Packing Across Partition Boundaries” on page 15–17
■ “Do Not Use Internal Tri-States” on page 15–21
■ “Include All Tri-State and Enable Logic in the Same Partition” on page 15–22
■ “Include Bidirectional I/O Registers in the Same Partition (For Older Device Families)” on page 15–23
■ “Summary of Guidelines Related to Logic Optimization Across Partitions” on page 15–23

Keep Logic in the Same Partition for Optimization and Merging

If your design logic requires logic optimization or merging to obtain optimal results, ensure that all the logic is part of the same partition because only limited cross-boundary optimizations are permitted.
If a combinational logic path is split across two partitions, the logic cannot be optimized or merged into one logic cell in the device. This effect can result in an extra logic cell in the path, increasing the logic delay. As a very simple example, consider two inverters on the same signal in two different partitions, A and B, as shown in the left diagram of Figure 15–5. To maintain correct incremental functionality, these two inverters cannot be removed from the design during optimization because they occur in different design partitions. The Quartus II software cannot use information about other partitions when it compiles each partition, because each partition is allowed to change independently from the other.

On the right side of the figure, partitions A and B are merged to group the logic in blocks A and B into one partition. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchy block as the partition. With the logic contained in one partition, the software can optimize the logic and remove the two inverters (shown in gray), which reduces the delay for that logic path. Removing two inverters is not a significant reduction in resource utilization because inversion logic is readily available in Altera device architecture. However, this example is a simple demonstration of the types of logic optimization that are prevented by partition boundaries.

In a flat design, the Fitter can also merge logical instantiations into the same physical device resource. With incremental compilation, logic defined in different partitions cannot be merged to use the same physical device resource.

For example, the Fitter can merge two single-port RAMs from a design into one dedicated RAM block in the device. If the two RAMs are defined in different partitions, the Fitter cannot merge them into one dedicated device RAM block. This limitation is only a concern if merging is required to fit the design in the target device. Therefore, you are more likely to encounter this issue during troubleshooting rather than during planning, if your design uses more logic than is available in the device.
Merging PLLs and Transceivers (GXB)

Multiple instances of the ALTPLL megafunction can use the same PLL resource on the device. Similarly, GXB transceiver instances can share high-speed serial interface (HSSI) resources in the same quad as other instances. The Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions. Therefore, there are no restrictions for PLLs and high-speed transceiver blocks when setting up partitions.

Keep Constants in the Same Partition as Logic

Because the Quartus II software cannot fully optimize across a partition boundary, constants are not propagated across partition boundaries, except from parent partition to child partition. A signal that is constant (1/VCC or 0/GND) in one partition cannot affect another partition.

For example, the left diagram of Figure 15–6 shows part of a design in which partition A defines some signals as constants (and assumes that the other input connections come from elsewhere in the design and are not shown in the figure). Constants such as these could appear due to parameter or generic settings or configurations with parameters, setting a bus to a specific set of values, or could result from optimizations that occur within a group of logic. Because the blocks are independent, the software cannot optimize the logic in block B based on the information from block A. The right side of Figure 15–6 shows a merged partition that groups the logic in blocks A and B. If the two blocks A and B are not under the same immediate parent partition, you can create a wrapper file to define a new level of hierarchy that contains both blocks, and set this new hierarchical block as the partition.

Within the single merged partition, the Quartus II software can use the constants to optimize and remove much of the logic in block B (shown in gray), as shown in Figure 15–6.

Figure 15–6. Keeping Constants in the Same Partition as the Logic They Feed

For more information about how many input ports are fed by GND or VCC, refer to “Partition Statistics Report” on page 15–34. For more information about port connections, refer to “Incremental Compilation Advisor” on page 15–49.
Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together

Do not use the same signal to drive multiple ports of a single partition or directly connect two ports of a partition. If the same signal drives multiple ports of a partition, or if two ports of a partition are directly connected, those ports are logically equivalent. However, the software has limited information about connections made in another partition (including the top-level partition), the compilation cannot take advantage of the equivalence. This restriction usually produces sub-optimal results.

If your design has these types of connections, redefine the partition boundaries to remove the affected ports. If one signal from a higher-level partition feeds two input ports of the same partition, feed the one signal into the partition, and then make the two connections within the partition. If an output port drives an input port of the same partition, the connection can be made internally without going through any I/O ports. If an input port drives an output port directly, the connection can likely be implemented without the ports in the lower-level partition by connecting the signals in a higher-level partition.

Figure 15–7 shows an example of one signal driving more than one port. The left diagram shows a design where a single clock signal is used to drive both the read and write clocks of a RAM block. Because the RAM block is compiled as a separate partition A, the RAM block is implemented as though there are two unique clocks. If you know that the port connectivity will not change (that is, the ports will always be driven by the same signal in the top-level partition), redefine the port interface so that there is only a single port that can drive both connections inside the partition. You can create a wrapper file to define a partition that has fewer ports, as shown in the diagram on the right side. With the single clock fed into the partition, the RAM can be optimized into a single-clock RAM instead of a dual-clock RAM. Single-clock RAM can provide better performance in the device architecture. Additionally, partition A might use two global routing lines for the two copies of the clock signal. Partition B can use one global line that fans out to all destinations. Using just the single port connection prevents overuse of global routing resources.

For more information about partition ports that have the same driving signal and ports that are directly connected together, refer to “Incremental Compilation Advisor” on page 15–49.
Invert Clocks in Destination Partitions

For best results, clock inversion should be performed in the destination logic array block (LAB) because each LAB contains clock inversion circuitry in the device architecture. In a flat compilation, the Quartus II software can optimize a clock inversion to propagate it to the destination LABs regardless of where the inversion takes place in the design hierarchy. However, clock inversion cannot propagate through a partition boundary (except from a parent partition to a child partition) to take advantage of the inversion architecture in the destination LABs.

With partition boundaries as shown in the left diagram of Figure 15–8, the Quartus II software uses logic to invert the signal in the partition that defines the inversion (the top-level partition in this example), and then routes the signal on a global clock resource to its destinations (in partitions A and B). The inverted clock acts as a gated clock with high skew. A better solution is to invert the clock signal in the destination partitions as shown on the right side of the diagram. In this case, the correct logic and routing resources can be used, and the signal does not behave like a gated clock.

Figure 15–8 shows the clock signal inversion in the destination partitions.

Figure 15–8. Inverting Clock Signal in Destination Partitions

Notice that this diagram also shows another example of a single pin feeding two ports of a partition boundary. In the left diagram, partition B does not have the information that the clock and inverted clock come from the same source. In the right diagram, partition B has more information to help optimize the design because the clock is connected as one port of the partition.

Connect I/O Pin Directly to I/O Register for Packing Across Partition Boundaries

The Quartus II software allows cross-partition register packing of I/O registers in certain cases where your input and output pins are defined in the top-level hierarchy (and the top-level partition), but the corresponding I/O registers are defined in other partitions.

Input pin cross-partition register packing requires the following specific circumstances:

- The input pin feeds exactly one register.
The path between the input pin and register includes only input ports of partitions that have one fan-out each.

Output pin cross-partition register packing requires the following specific circumstances:

- The register feeds exactly one output pin.
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

The following examples of I/O register packing illustrate this point using Block Design File (.bdf) schematics to describe the design logic.

**Example 1—Output Register in Partition Feeding Multiple Output Pins**

In this example, a subdesign contains a single register, as shown in Figure 15–9.

If the top-level design instantiates the subdesign with a single fan-out directly feeding an output pin, and designates the subdesign as a separate design partition, the Quartus II software can perform cross-partition register packing because the single partition port feeds the output pin directly.

In Example 1, the top-level design instantiates the subdesign in Figure 15–9 as an output register with more than one fan-out signal, as shown in Figure 15–10.

In this case, the Quartus II software does not perform output register packing. If there is a Fast Output Register assignment on pin out, the software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and the I/O cell are connected across a design partition boundary.
This type of cross-partition register packing is not allowed because it requires modification to the interface of the subdesign partition. To perform incremental compilation, the Quartus II software must preserve the interface of design partitions.

To allow the Quartus II software to pack the register in the subdesign from Figure 15–9 with the output pin out in Figure 15–10, restructure your HDL code so that output registers directly connect to output pins by making one of the following changes:

- Place the register in the same partition as the output pin. The simplest method is to move the register from the subdesign partition into the partition containing the output pin. Doing so guarantees that the Fitter can optimize the two nodes without violating partition boundaries.

- Duplicate the register in your subdesign HDL as shown in Figure 15–11 so that each register feeds only one pin, and then connect the extra output pin to the new port in the top-level design as shown in Figure 15–12. Doing so converts the cross-partition register packing into the simplest case where each register has a single fan-out.

Figure 15–11. Modified Subdesign from Figure 15–9 with Two Output Registers and Two Output Ports

Figure 15–12. Modified Top-Level Design from Figure 15–10 Connecting Two Output Ports to Output Pins
Example 2—Input Register in Partition Fed by an Inverted Input Pin or Output Register in Partition Feeding an Inverted Output Pin

In this example, a subdesign designated as a separate partition contains a register, as shown in Figure 15–9. The top-level design in Figure 15–13 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 15–14 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

In these cases, the Quartus II software does not perform register packing. If there is a Fast Input Register assignment on pin \( \text{in} \), as shown in Figure 15–13, or a Fast Output Register assignment on pin \( \text{out} \), as shown in Figure 15–14, the Quartus II software issues a warning that the Fitter cannot pack the node to an I/O pin because the node and I/O cell are connected across a design partition boundary. This type of register packing is not allowed because it requires moving logic across a design partition boundary to place into a single I/O device atom. To perform register packing, either the register must be moved out of the subdesign partition, or the inverter must be moved into the subdesign partition to be implemented in the register.

To allow the Quartus II software to pack the register in the subdesign from Figure 15–9 with the input pin \( \text{in} \), as shown in Figure 15–13 or the output pin \( \text{out} \), as shown in Figure 15–14, restructure your HDL code to place the register in the same partition as the inverter by making one of the following changes:

- Move the register from the subdesign partition into the top-level partition containing the pin. Doing so ensures that the Fitter can optimize the I/O register and inverter without violating partition boundaries.
Move the inverter from the top-level block into the subdesign, and then connect the subdesign directly to a pin in the top-level design. Doing so allows the Fitter to optimize the inverter into the register implementation, so that the register is directly connected to a pin, which enables register packing.

**Do Not Use Internal Tri-States**

Internal tri-state signals are not recommended for FPGAs because the device architecture does not include internal tri-state logic. If designs use internal tri-states in a flat design, the tri-state logic is usually converted to OR gates or multiplexing logic. If tri-state logic occurs on a hierarchical partition boundary, the Quartus II software cannot convert the logic to combinational gates because the partition could be connected to a top-level device I/O through another partition.

Figure 15–15 and Figure 15–16 show a design with partitions that are not supported for incremental compilation due to the internal tri-state output logic on the partition boundaries. Instead of using internal tri-state logic for partition outputs, implement the correct logic to select between the two signals. Doing so is good practice even when there are no partitions, because such logic explicitly defines the behavior for the internal signals instead of relying on the Quartus II software to convert the tri-state signals into logic.
Do not use tri-state signals or bidirectional ports on hierarchical partition boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you must use internal tri-state logic, ensure that all the control and destination logic is contained in the same partition, in which case the Quartus II software can convert the internal tri-state signals into combinational logic as in a flat design. In this example, you can also merge all three partitions into one partition, as shown in Figure 15–16, to allow the Quartus II software to treat the logic as internal tri-state and perform the same type of optimization as a flat design. If possible, you should avoid using internal tri-state logic in any Altera FPGA design to ensure that you get the desired implementation when the design is compiled for the target device architecture.

**Include All Tri-State and Enable Logic in the Same Partition**

When multiple output signals use tri-state logic to drive a device output pin, the Quartus II software merges the logic into one tri-state output pin. The Quartus II software cannot merge tri-state outputs into one output pin if any of the tri-state logic occurs on a partition boundary. Similarly, output pins with an output enable signal cannot be packed into the device I/O cell if the output enable logic is part of a different partition from the output register. To allow register packing for output pins with an output enable signal, structure your HDL code or design partition assignments so that the register and enable logic are defined in the same partition.

Figure 15–17 shows a design with tri-state output signals that feed a device bidirectional I/O pin (assuming that the input connection feeds elsewhere in the design and is not shown in the figure). In the left diagram below, the tri-state output signals appear as the outputs of two separate partitions. In this case, the Quartus II software cannot implement the specified logic and maintain incremental functionality. In the right diagram, partitions A and B are merged to group the logic from the two blocks. With this single partition, the Quartus II software can merge the two tri-state output signals and implement them in the tri-state logic available in the device I/O element.

**Figure 15–17. Including All Tri-State Output Logic in the Same Partition**
Include Bidirectional I/O Registers in the Same Partition (For Older Device Families)

For a bidirectional partition port that feeds a bidirectional I/O pin at the top level, all logic that forms the bidirectional I/O cell must reside in the same partition in the Stratix II, Stratix, Cyclone® II, and Cyclone device families (this restriction does not apply to newer devices). Additionally, as discussed in the previous two recommendations, the I/O logic must feed the I/O pin without any intervening logic.

In Figure 15–18, all the I/O logic must be defined inside the same partition for the Quartus II software to implement all three registers in the I/O element along with the tri-state logic in the affected devices. The logic connected to the registers can occur in the same partition or any other partition; only the I/O registers must be grouped with the tri-state logic definition. The bidirectional I/O port of the partition must be directly connected to the bidirectional device pin at the top level. The signal can go through several partition boundaries if necessary, as long as the connection path contains no logic.

Summary of Guidelines Related to Logic Optimization Across Partitions

To ensure that your design does not require logic optimization across partitions, follow the guidelines in this section:

- Include logic in the same partition for optimization and merging
- Include constants in the same partition as logic
- Avoid signals that drive multiple partition I/O or connect I/O together
- Invert clocks in destination partitions
- Connect I/O directly to I/O register for packing across partition boundaries
- Do not use internal tri-states
- Include all tri-state and enable logic in the same partition
- Include bidirectional I/O registers in the same partition (in older device families)
Remember that these guidelines are not mandatory when implementing an incremental compilation flow, but can improve the quality of results. When creating source design code, follow these guidelines and organize your HDL code to support good partition boundaries. For designs that are complete, assess whether assigning a partition affects the resource utilization or timing performance of a design block as compared to the flat design. Make the appropriate changes to your design or hierarchy, or merge partitions as required, to improve your results.

**Consider a Cascaded Reset Structure**

Designs typically have a global asynchronous reset signal where a top-level signal feeds all partitions. To minimize skew for the high fan-out signal, the global reset signal is typically placed onto a global routing resource.

In some cases, having one global reset signal can lead to recovery and removal time problems. This issue is not specific to incremental flows; it could be applicable in any large high-speed design. In an incremental flow, the global reset signal creates a timing dependency between the top-level partition and lower-level partitions.

For incremental compilation, it is helpful to minimize the impact of global structures. To isolate each partition, consider adding reset synchronizers. Using cascaded reset structures, the intent is to reduce the inter-partition fan-out of the reset signal, thereby minimizing the effect of the global signal. Reducing the fan-out of the global reset signal also provides more flexibility in routing the cascaded signals, and might help recovery and removal times in some cases.

This recommendation can help in large designs, regardless of whether you are using incremental compilation. However, if one global signal can feed all the logic in its domain and meet recovery and removal times, this recommendation may not be applicable for your design. Minimizing global structures is more relevant for high-performance designs where meeting timing on the reset logic can be challenging. Isolating each partition and allowing more flexibility in global routing structures is an additional advantage in incremental flows.

If you add additional reset synchronizers to your design, latency is also added to the reset path, so ensure that this is acceptable in your design. Additionally, parts of the design may come out of the reset state in different clock cycles. You can balance the latency or add hand-shaking logic between partitions, if necessary, to accommodate these differences.

The signal is first synchronized on the chip following good synchronous design practices, meaning that the design asynchronously resets, but synchronously releases from reset to avoid any race conditions or metastability problems. Then, to minimize the impact of global structures, the circuit employs a divide-and-conquer approach for the reset structure. By implementing a cascaded reset structure, the reset paths for each partition are independent. This structure reduces the effect of inter-partition dependency because the inter-partition reset signals can now be treated as false paths for timing analysis. In some cases, the reset signal of the partition can be placed on local lines to reduce the delay added by routing to a global routing line. In other cases, the signal can be routed on a regional or quadrant clock signal.
Figure 15–19 shows a cascaded reset structure.

**Figure 15–19. Cascaded Reset Structure**

This circuit design can help you achieve timing closure and partition independence for your global reset signal. Evaluate the circuit and consider how it works for your design.

For more information and design recommendations for reset structures, refer to the Recommended Design Practices chapter in volume 1 of the Quartus II Handbook.

**Design Partition Guidelines for Third-Party IP Delivery**

This section includes additional design guidelines that can improve incremental compilation flows where exported partitions are developed independently. These guidelines are not always required, but are usually recommended if the design includes partitions compiled in a separate Quartus II project, such as when delivering intellectual property (IP). A unique challenge of IP delivery for FPGAs is the fact that the partitions developed independently must share a common set of resources. To minimize issues that might arise from sharing a common set of resources, you can design partitions within a single Quartus II project, or a copy of the top-level design. A common project ensures that designers have a consistent view of the top-level design framework, as described in “Project Management in Team-Based Design Flows” on page 15–4.

Alternatively, an IP designer can export just the post-synthesis results to be integrated in the top-level design when the post-fitting results from the IP project are not required. Using a post-synthesis netlist provides more flexibility to the Quartus II Fitter, so that less resource allocation is required. If a common project is not possible, especially when the project lead plans to integrate the IP’s post-fitting results, it is important that the project lead and IP designer clearly communicate their requirements.
Allocate Logic Resources

In an incremental compilation design flow in which designers, such as third-party IP providers, optimize partitions and then export them to a top-level design, the Quartus II software places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level. Allocation of logic resources requires that you decide on a set of logic resources (including I/O, LAB logic blocks, RAM and DSP blocks) that the IP block will “own”. This process can be interactive; the project lead and the IP designer might work together to determine what resources are required for the IP block and are available in the top-level design.

You can constrain logic utilization for the IP core using design floorplan location assignments, as described in “Introduction to Design Floorplans” on page 15–40. The design should specify I/O pin locations with pin assignments.

You can also specify limits for Quartus II synthesis to allocate and balance resources. This procedure can also help if device resources are overused in the individual partitions during synthesis.

In the standard synthesis flow, the Quartus II software can perform automated resource balancing for DSP blocks or RAM blocks and convert some of the logic into regular logic cells to prevent overuse.

You can use the Quartus II synthesis options to control inference of megafunctions that use the DSP, or RAM blocks. You can also use the MegaWizard™ Plug-In Manager to customize your RAM or DSP megafunctions to use regular logic instead of the dedicated hardware blocks.

For more information about resource balancing DSP and RAM blocks when using Quartus II synthesis, refer to the “Megafunction Inference Control” section in the Quartus II Integrated Synthesis chapter in volume 1 of the Quartus II Handbook. For tips about resource balancing and reducing resource utilization, refer to the appropriate “Resource Utilization Optimization Techniques” section in the Area and Timing Optimization chapter in volume 2 of the Quartus II Handbook.

For information about how to set global logic options for partitions, refer to More Analysis & Synthesis Settings Dialog Box in Quartus II Help.

Allocate Global Routing Signals and Clock Networks if Required

In most cases, you do not have to allocate global routing signals because the Quartus II software finds the best solution for the global signals. However, if your design is complex and has multiple clocks, especially for a partition developed by a third-party IP designer, you may have to allocate global routing resources between various partitions.

Global routing signals can cause conflicts when independent partitions are integrated into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Third-party partitions can use the same global routing resources, thus causing conflicts in the top-level design. Additionally, LAB placement depends on whether the inputs to the logic cells within the LAB use a global clock signal. Problems can occur if a design does not use a global signal in a lower-level partition, but does use a global signal in the top-level design.
If the exported IP core is small, you can reduce the potential for problems by using constraints to promote clock and high fan-out signals to regional routing signals that cover only part of the device, instead of global routing signals. In this case, the Quartus II software is likely to find a routing solution in the top-level design because there are many regional routing signals available on most Altera devices, and designs do not typically overuse regional resources.

To ensure that an IP block can utilize a regional clock signal, view the resource coverage of regional clocks in the Chip Planner, and then align LogicLock regions that constrain partition placement with available global clock routing resources. For example, if the LogicLock region for a particular partition is limited to one device quadrant, that partition’s clock can use a regional clock routing type that covers only one device quadrant. When all partition logic is available, the project lead can compile the entire design at the top level with floorplan assignments to allow the use of regional clocks that span only a part of the device.

If global resources are heavily used in the overall design, or the IP designer requires global clocks for their partition, you can set up constraints to avoid signal overuse at the top-level by assigning the appropriate type of global signals or setting a maximum number of clock signals for the partition.

You can use the Global Signal assignment to force or prevent the use of a global routing line, making the assignment to a clock source node or signal. You can also assign certain types of global clock resources in some device families, such as regional clocks. For example, if you have an IP core, such as a memory interface that specifies the use of a dual regional clock, you can constrain the IP to part of the device covered by a regional clock and change the Global Signal assignment to use a regional clock. This type of assignment can reduce clocking congestion and conflicts.

Alternatively, partition designers can specify the number of clocks allowed in the project using the maximum clocks allowed options in the More Fitter Settings dialog box. Specify Maximum number of clocks of any type allowed, or use the Maximum number of global clocks allowed, Maximum number of regional clocks allowed, and Maximum number of periphery clocks allowed options to restrict the number of clock resources of a particular type in your design.

If you require more control when planning a design with integrated partitions, you can assign a specific signal to use a particular clock network in Stratix II and newer device families by assigning the clock control block instance called CLKCTRL. You can make a point-to-point assignment from a clock source node to a destination node, or a single-point assignment to a clock source node with the Global Clock CLKCTRL Location logic option. Set the assignment value to the name of the clock control block: CLKCTRL_G<global network number> for a global routing network, or CLKCTRL_R<regional network number> for a dedicated regional routing network in the device.

If you want to disable the automatic global promotion performed in the Fitter to prevent other signals from being placed on global (or regional) routing networks, turn off the Auto Global Clock and Auto Global Register Control Signals options in the More Fitter Settings dialog box.

For information about how to disable automatic global promotion, refer to More Fitter Settings Dialog Box in Quartus II Help.
If you are using design partition scripts for independent partitions, the Quartus II software can automatically write the commands to pass global constraints and turn off automatic options.

For more information about how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

For more information about how clock networks affect partition design, refer to the *Analyzing and Optimizing the Design Floorplan with the Chip Planner* chapter in volume 2 of the *Quartus II Handbook*.

Alternatively, to avoid problems when integrating partitions into the top-level design, you can direct the Fitter to discard the placement and routing of the partition netlist by using the post-synthesis netlist, which forces the Fitter to reassign all the global signals for the partition when compiling the top-level design.

**Assign Virtual Pins**

Virtual pins map lower-level design I/Os to internal cells. If you are developing an IP block in an independent Quartus II project, use virtual pins when the number of I/Os on a partition exceeds the device I/O count, and to increase the timing accuracy of cross-partition paths.

You can create a virtual pin assignment in the Assignment Editor for partition I/Os that will become internal nodes in the top-level design. Leave the clock pins mapped to I/O pins to ensure proper routing.

You can specify locations for the virtual pins that correspond to the placement of other partitions, and also make timing assignments to the virtual pins to define a timing budget, as described in the following section. Virtual pins are created automatically from the top-level design if you use design partition scripts. The scripts place the virtual pins to correspond with the placement of the other partitions in the top-level design.

For more information about how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

Tri-state outputs cannot be assigned as virtual pins because internal tri-state signals are not supported in Altera devices. Connect the signal in the design with regular logic, or allow the software to implement the signal as an external device I/O pin.

**Perform Timing Budgeting if Required**

If you optimize partitions independently and integrate them to the top-level design, or compile with empty partitions, any unregistered paths that cross between partitions are not optimized as entire paths. In these cases, the Quartus II software has no information about the placement of the logic that connects to the I/O ports. If the logic in one partition is placed far away from logic in another partition, the routing
delay between the logic can lead to problems in meeting timing requirements. You can reduce this effect by ensuring that input and output ports of the partitions are registered whenever possible. Additionally, using the same top-level project framework helps to avoid this problem by providing the software with full information about other design partitions in the top-level design.

To ensure that the software correctly optimizes the input and output logic in any independent partitions, you might be required to perform some manual timing budgeting. For each unregistered timing path that crosses between partitions, make timing assignments on the corresponding I/O path in each partition to constrain both ends of the path to the budgeted timing delay. Assigning a timing budget for each part of the connection ensures that the software optimizes the paths appropriately.

When performing manual timing budgeting in a partition for I/O ports that become internal partition connections in a top-level design, you can assign location and timing constraints to the virtual pin that represents each connection to further improve the quality of the timing budget. Refer to “Assign Virtual Pins” on page 15–28 for a description of virtual pins.

If you use design partition scripts, the Quartus II software can write I/O timing budget constraints automatically for virtual pins.

For more information about how to generate design partition scripts, refer to Generating Design Partition Scripts for Project Management in Quartus II Help.

**Drive Clocks Directly**

When partitions are exported from another Quartus II project, you should drive partition clock inputs directly with device clock input pins.

Connecting the clock signal directly avoids any timing analysis difficulties with gated clocks. Clock gating is never recommended for FPGA designs because of potential glitches and clock skew. Clock gating can be especially problematic with exported partitions because the partitions have no information about gating that takes place at the top-level design or in another partition. If a gated clock is required in a partition, perform the gating within that partition, as described for clock inversion in “Invert Clocks in Destination Partitions” on page 15–17.

Direct connections to input clock pins also allows design partition scripts to send constraints from the top-level device pin to lower-level partitions.

**Recreate PLLs for Lower-Level Partitions if Required**

If you connect a PLL in your top-level design to partitions designed in separate Quartus II projects by third-party IP designers, the IP partitions do not have information about the multiplication, phase shift, or compensation delays for the PLL in the top-level design. To accommodate the PLL timing, you can make appropriate timing assignments in the projects created by IP designers to ensure that clocks are not left unconstrained or constrained with an incorrect frequency. Alternatively, you can duplicate the top-level PLL (or other derived clock logic) in the design file for the project created by the IP designer to ensure that you have the correct PLL parameters and clock delays for a complete and accurate timing analysis.
If the project lead creates a copy of the top-level project framework that includes all the settings and constraints needed for the design, this framework should include PLLs and other interface logic if this information is important to optimize partitions.

If you use a separate Quartus II project for an independent design block (such as when a designer or third-party IP provider does not have access to the entire design framework), include a copy of the top-level PLL in the lower-level partition as shown in Figure 15–20.

In either case, the IP partition in the separate Quartus II project should contain just the partition logic that will be exported to the top-level design, while the full project includes more information about the top-level design. When the partition is complete, you can export just the partition without exporting the auxiliary PLL components to the top-level design. When you export a partition, the Quartus II software exports any hierarchy under the specified partition into the Quartus II Exported Partition File (.qxp), but does not include logic defined outside the partition (the PLL in this example).

**Figure 15–20. Recreating a Top-Level PLL in a Lower-Level Partition**

**Checking Partition Quality**

This section provides an overview of tools you can use to create and analyze partitions in the Quartus II software. Take advantage of these tools to assess your partition quality, and use the information to improve your design or assignments as required to achieve the best results.

**Incremental Compilation Advisor**

You can use the Incremental Compilation Advisor to ensure that your design follows Altera’s recommendations for creating design partitions and implementing the incremental compilation design flow methodology. Each recommendation in the Incremental Compilation Advisor provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change.

For more information about the Incremental Compilation Advisor, refer to Incremental Compilation Advisor Command and Example of Using the Incremental Compilation Advisor to Identify Non-Global Ports That Are Not Registered in Quartus II Help, and the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.
Design Partition Planner

The Design Partition Planner allows you to view design connectivity and hierarchy, and can assist you in creating effective design partitions that follow the guidelines in this chapter. You can also use the Design Partition Planner to optimize design performance by isolating and resolving failing paths on a partition-by-partition basis.

To view a design and create design partitions in the Design Partition Planner, you must first compile the design, or perform Analysis & Synthesis. In the Design Partition Planner, the design appears as a single top-level design block, with lower-level instances displayed as color-specific boxes.

In the Design Partition Planner, you can show connectivity between blocks and extract instances from the top-level design block. When you extract entities, connection bundles are drawn between entities, showing the number of connections existing between pairs of entities. When you have extracted a design block that you want to set as a design partition, right-click that design block, and then click **Create Design Partition**.

The Design Partition Planner also has an auto-partition feature that creates partitions based on the size and connectivity of the hierarchical design blocks. You can right-click the design block you want to partition (such as the top-level design hierarchy), and then click **Auto-Partition Children**. You can then analyze and adjust the partition assignments as required.
Figure 15–21 shows the Design Partition Planner after making a design partition assignment to one instance and dragging another instance away from the top-level block within the same partition (two design blocks in the pale blue shaded box). The figure shows the connections between each partition and information about the size of each design instance.

You can switch between connectivity display mode and hierarchical display mode, or temporarily to a view-only hierarchy display. You can also remove the connection lines between partitions and I/O banks by turning off Display connections to I/O banks, or use the settings on the Connection Counting tab in the Bundle Configuration dialog box to adjust how the connections are counted in the bundles.

To optimize design performance, confine failing paths within individual design partitions so that there are no failing paths passing between partitions, as discussed in earlier sections. In the top-level entity, child entities that contain failing paths are marked by a small red dot in the upper right corner of the entity box.

To view the critical timing paths from a timing analyzer report, first perform a timing analysis on your design, and then in the Design Partition Planner, click Show Timing Data on the View menu.

For more information about the Design Partition Planner, refer to About the Design Partition Planner and Using the Design Partition Planner in Quartus II Help.
Viewing Design Partition Planner and Floorplan Side-by-Side

You can use the Design Partition Planner together with the Chip Planner to analyze natural placement groupings. This information can help you decide whether the design blocks should be grouped together in one partition, or whether they will make good partitions in the next compilation. It can also help determine whether the logic can easily be constrained by a LogicLock region. If logic naturally groups together when compiled without placement constraints, you can probably assign a reasonably sized LogicLock region to constrain the placement for subsequent compilations. You can experiment by extracting different design blocks in the Design Partition Planner and viewing the placement results of those design blocks from the previous compilation.

To view the Design Partition Planner and Chip Planner side-by-side, open the Design Partition Planner, and then open the Chip Planner and select the Design Partition Planner task. The Design Partition Planner task displays the physical locations of design entities with the same colors as in the Design Partition Planner.

In the Design Partition Planner, you can extract instances of interest from their parents by dragging and dropping, or with the Extract from Parent command. Evaluate the physical locations of instances in the Chip Planner and the connectivity between instances displayed in the Design Partition Planner. An entity is generally not suitable to be set as a separate design partition or constrained in a LogicLock region if the Chip Planner shows it physically dispersed over a noncontiguous area of the device after compilation. Use the Design Partition Planner to analyze the design connections. Child instances that are unsuitable to be set as separate design partitions or placed in LogicLock regions can be returned to their parent by dragging and dropping, or with the Collapse to Parent command.
Figure 15–22 shows a design displayed in the Design Partition Planner and the Chip Planner with different colors for the top-level design and the three major design instances.

For more information about the Design Partition Planner, refer to About the Design Partition Planner and Using the Design Partition Planner in Quartus II Help.

**Partition Statistics Report**

You can view statistics about design partitions in the Partition Merge Partition Statistics report and the Statistics tab of the Design Partitions Properties dialog box. These reports are useful when optimizing your design partitions, or when compiling the completed top-level design in a team-based compilation flow to ensure that partitions meet the guidelines discussed in this chapter.

The Partition Merge Partition Statistics report in the Partition Merge section of the Compilation report lists statistics about each partition. The statistics for each partition (each row in the table) include the number of logic cells, as well as the number of input and output pins and how many are registered. This report also lists how many ports are unconnected, or driven by a constant VCC or GND. You can use this information to assess whether you have followed the guidelines for partition boundaries.
Checking Partition Quality

You can also view statistics about the resource and port connections for a particular partition on the Statistics tab of the Design Partition Properties dialog box. The Show All Partitions button allows you to view all the partitions in the same report. The Partition Merge Partition Statistics report also shows statistics for the Internal Congestion: Total Connections and Registered Connections. This information represents how many signals are connected within the partition. It then lists the inter-partition connections for each partition, which helps you to see how partitions are connected to each other.

For more information about the Partition Merge Reports, refer to Partition Merge Reports in Quartus II Help.

Report Partition Timing in the TimeQuest Timing Analyzer

The Report Partitions diagnostic report and the report_partitions SDC command in the TimeQuest analyzer produce a Partition Timing Overview and Partition Timing Details table, which lists the partitions, the number of failing paths, and the worst case timing slack within each partition.

You can use these reports to analyze the location of the critical timing paths in the design in relation to partitions. If a certain partition contains many failing paths, or failing inter-partition paths, you might be able to change your partitioning scheme and improve timing performance.

For more information about the TimeQuest report_timing command and reports, see the Quartus II TimeQuest Timing Analyzer chapter in volume 3 of the Quartus II Handbook.

Check if Partition Assignments Impact the Quality of Results

You can ensure that you limit negative effect on the quality of results by following an iterative methodology during the partitioning process. In any incremental compilation flow where you can compile the source code for every partition during the partition planning phase, Altera recommends the following iterative flow:

1. Start with a complete design that is not partitioned and has no location or LogicLock region assignments.
   After Analysis & Synthesis and Partition Merge, perform a placement and timing analysis estimate with the Start Early Timing Estimate command. To run a full compilation instead, use the Start Compilation command.
2. Record the quality of results from the Compilation report (timing slack or $f_{MAX}$, area and any other relevant results).
3. Create design partitions following the guidelines described in this chapter.
4. Perform another early timing estimate or a full compilation.
5. Record the quality of results from the Compilation report. If the quality of results is significantly worse than those obtained in the previous compilation, repeat step 3 through step 5 to change your partition assignments and use a different partitioning scheme.
6. Even if the quality of results is acceptable, you can repeat step 3 through step 5 by further dividing a large partition into several smaller partitions, which can improve compilation time in subsequent incremental compilations. You can repeat these steps until you achieve a good trade-off point (that is, all critical paths are localized within partitions, the quality of results is not negatively affected, and the size of each partition is reasonable).

You can also remove or disable partition assignments defined in the top-level design at any time during the design flow to compile the design as one flat compilation and get all possible design optimizations to assess the results. To disable the partitions without deleting the assignments, use the `Ignore partition assignments during compilation` option on the `Incremental Compilation` page of the `Settings` dialog box in the Quartus II software. This option disables all design partition assignments in your project and runs a full compilation, ignoring all partition boundaries and netlists. This option can be useful if you are using partitions to reduce compilation time as you develop various parts of the design, but can run a long compilation near the end of the design cycle to ensure the design meets its timing requirements.

**Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery**

When exported partitions are compiled in a separate Quartus II project, such as when a third-party designer is delivering IP, the project lead must transfer the top-level project framework information and constraints to the partitions, so that each designer has a consistent view of the constraints that apply to the entire design. If the independent partition designers make any changes or add any constraints, they might have to transfer new constraints back to the project lead, so that these constraints are included in final timing sign-off of the entire design. Many assignments from the partition are carried with the partition into the top-level design; however, SDC format constraints for the TimeQuest analyzer are not copied into the top-level design automatically.

Passing additional timing constraints from a partition to the top-level design must be managed carefully. This section provides recommendations for managing the timing constraints in a third-party IP delivery flow. You can design within a single Quartus II project or a copy of the top-level design to simplify constraint management.

To ensure that there are no conflicts between the project lead’s top-level constraints and those added by the third-party IP designer, use two `.sdc` files for each separate Quartus II project: an `.sdc` created by the project lead that includes project-wide constraints, and an `.sdc` created by the IP designer that includes partition-specific constraints.
This section uses the example design shown in Figure 15–23 to illustrate these recommendations. The top-level design instantiates a lower-level design block called module_A that is set as a design partition and developed by an IP designer in a separate Quartus II project.

**Figure 15–23. Example Design to Illustrate SDC Constraints**

In this top-level design, there is a single clock setting called clk associated with the FPGA input called top_level_clk. The top-level .sdc contains the following constraint for the clock:

```bash
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 } [get_ports {TOP_LEVEL_CLK}]
```

**Creating an .sdc File with Project-Wide Constraints**

The .sdc with project-wide constraints for the separate Quartus II project should contain all constraints that are not completely localized to the partition. The .sdc should be maintained by the project lead. The project lead must ensure that these timing constraints are delivered to the individual partition owners and that they are syntactically correct for each of the separate Quartus II projects. This communication can be challenging when the design is in flux and hierarchies change. The project lead can use design partition scripts to automatically pass some of these constraints to the separate Quartus II projects.

For more information about how to generate design partition scripts, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

The .sdc with project-wide constraints is used in the partition, but is not exported back to the top-level design. The partition designer should not modify this file. If changes are necessary, they should be communicated to the project lead, who can then update the SDC constraints and distribute new files to all partition designers as required.

The .sdc should include clock creation and clock constraints for any clock used by more than one partition. These constraints are particularly important when working with complex clocking structures, such as the following:

- Cascaded clock multiplexers
- Cascaded PLLs
- Multiple independent clocks on the same clock pin
- Redundant clocking structures required for secure applications
- Virtual clocks and generated clocks that are consistently used for source synchronous interfaces
- Clock uncertainties

Additionally, the .sdc with project-wide constraints should contain all project-wide timing exception assignments, such as the following:

- Multicycle assignments, `set_multicycle_path`
- False path assignments, `set_false_path`
- Maximum delay assignments, `set_max_delay`
- Minimum delay assignments, `set_min_delay`

The project-wide .sdc can also contain any `set_input_delay` or `set_output_delay` constraints that are used for ports in separate Quartus II projects, because these represent delays external to a given partition. If the partition designer wants to set these constraints within the separate Quartus II projects, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.

Similarly, a constraint on a path that crosses a partition boundary should be in the project-wide .sdc, because it is not completely localized in a separate Quartus II project.

**Example Step 1: Project Lead Produces .sdc with Project-Wide Constraints for Lower-Level Partitions**

The device input `top_level_clk` in Figure 15–23 drives the `input_clk` port of `module_A`. To make sure the clock constraint is passed correctly to the partition, the project lead creates an .sdc with project-wide constraints for `module_A` that contains the following command:

```plaintext
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 } [get_ports {INPUT_CLK}]
```

The designer of `module_A` includes this .sdc as part of the separate Quartus II project.

**Creating an .sdc with Partition-Specific Constraints**

The .sdc with partition-specific constraints should contain all constraints that affect only the partition. For example, a `set_false_path` or `set_multicycle_path` constraint for a path entirely within the partition should be in the partition-specific .sdc. These constraints are required for correct compilation of the partition, but need not be present in any other separate Quartus II projects.

The partition-specific .sdc should be maintained by the partition designer; they must add any constraints required to properly compile and analyze their partition.
The partition-specific .sdc is used in the separate Quartus II project and must be exported back to the project lead for the top-level design. The project lead must use the partition-specific constraints to properly constrain the placement, routing, or both, if the partition logic is fit at the top level, and to ensure that final timing sign-off is accurate. Use the following guidelines in the partition-specific .sdc to simplify these export and integration steps:

- Create a hierarchy variable for the partition (such as module_A_hierarchy) and set it to an empty string because the partition is the top-level instance in the separate Quartus II project. The project lead modifies this variable for the top-level hierarchy, reducing the effort of translating constraints on lower-level design hierarchies into constraints that apply in the top-level hierarchy. Use the following Tcl command first to check if the variable is already defined in the project, so that the top-level design does not use this empty hierarchy path: if {![info exists module_AHierarchy]}. If the variable is already defined, use the following code:

```tcl
if {![info exists module_A_hierarchy]} {
    set module_A_hierarchy ""
    set_false_path -from [get_registers ${module_A_hierarchy}reg_in_1] -to [get_registers ${module_A_hierarchy}*]
}
```

- Use the hierarchy variable in the partition-specific .sdc as a prefix for assignments in the project. For example, instead of naming a particular instance of a register reg:inst, use ${module_A_hierarchy}reg:inst. Also, use the hierarchy variable as a prefix to any wildcard characters (such as "*").

- Pay attention to the location of the assignments to I/O ports of the partition. In most cases, these assignments should be specified in the .sdc with project-wide constraints, because the partition interface depends on the top-level design. If you want to set I/O constraints within the partition, the team must ensure that the I/O port names are identical in all projects so that the assignments can be integrated successfully without changes.

- Use caution with the derive_clocks and derive_pll_clocks commands. In most cases, the .sdc with project-wide constraints should call these commands. Because these commands impact the entire design, integrating them unexpectedly into the top-level design might cause problems.

If the design team follows these recommendations, the project lead should be able to include the .sdc with the partition-specific constraints provided by the partition designer directly in the top-level design.

**Example Step 2: Partition Designer Creates .sdc with Partition-Specific Constraints**

The partition designer compiles the design with the .sdc with project-wide constraints and might want to add some additional constraints. In this example, the designer realizes that he or she must specify a false path between the register called reg_in_1 and all destinations in this design block with the wildcard character (such as "* *"). This constraint applies entirely within the partition and must be exported to the top-level design, so it qualifies for inclusion in the .sdc with partition-specific constraints. The designer first defines the module_A_hierarchy variable and uses it when writing the constraint as follows:

```tcl
if {![info exists module_A_hierarchy]} {
    set module_A_hierarchy ""
}
set_false_path -from [get_registers ${module_A_hierarchy}reg_in_1] -to [get_registers ${module_A_hierarchy}*]
```
Consolidating the .sdc in the Top-Level Design

When the partition designers complete their designs, they export the results to the project lead. The project lead receives the exported .qxp files and a copy of the .sdc with partition-specific constraints.

To set up the top-level .sdc constraint file to accept the .sdc files from the separate Quartus II projects, the top-level .sdc should define the hierarchy variables specified in the partition .sdc files. List the variable for each partition and set it to the hierarchy path, up to and including the instantiation of the partition in the top-level design, including the final hierarchy character “|”.

To ensure that the .sdc files are used in the correct order, the project lead can use the Tcl Source command to load each .sdc.

Example Step 3: Project Lead Performs Final Timing Analysis and Sign-off

With these commands, the top-level .sdc file looks like the following example:

```
create_clock -name {clk} -period 3.000 -waveform { 0.000 1.500 }
[get_ports {TOP_LEVEL_CLK}]
# Include the lower-level SDC file
set module_A_hierarchy "module_A:inst|"  # Note the final '|' character
source <partition-specific constraint file such as
..\module_A\module_A_constraints>.sdc
```

When the project lead performs top-level timing analysis, the false path assignment from the lower-level module_A project expands to the following:

```
set_false_path -from module_A:inst|reg_in_1 -to module_A:inst|*
```

Adding the hierarchy path as a prefix to the SDC command makes the constraint legal in the top-level design, and ensures that the wildcard does not affect any nodes outside the partition that it was intended to target.

By following the guidelines in this section, constraint propagation between the separate Quartus II projects can be managed effectively.

Introduction to Design Floorplans

A floorplan represents the layout of the physical resources on the device. Creating a design floorplan, or floorplanning, describes the process of mapping the logical design hierarchy onto physical regions in the device.

In the Quartus II software, LogicLock regions can be used to constrain blocks of a design to a particular region of the device. LogicLock regions represent an area on the device with a user-defined or Fitter-defined size and location in the device layout.

For more information about design floorplans and LogicLock regions, refer to the Analyzing and Optimizing the Design Floorplan chapter in volume 2 of the Quartus II Handbook.

The Difference between Logical Partitions and Physical Regions

Design partitions are logical entities based on the design hierarchy. LogicLock regions are physical placement assignments that constrain logic to a particular region on the device.
A common misconception is that logic from a design partition is always grouped together on the device when you use incremental compilation. Actually, logic from a partition can be placed anywhere in the device if it is not constrained to a LogicLock region, although the Fitter can pack related logic together to improve timing performance. A logical design partition does not refer to any physical area on the device and does not directly control where instances are placed on the device.

If you want to control the placement of logic from a design partition and isolate it to a particular part of the device, you can assign the logical design partition to a physical region in the device floorplan with a LogicLock region assignment. Altera recommends creating a design floorplan by assigning design partitions to LogicLock regions to improve the quality of results and avoid placement conflicts in some situations for incremental compilation. For more information, refer to “Why Create a Floorplan?” on page 15–41.

Another misconception is that LogicLock assignments are used to preserve placement results for incremental compilation. Actually, LogicLock regions only constrain logic to a physical region on the device. Incremental compilation does not use LogicLock assignments or any location assignments to preserve the placement results; it simply reuses the results stored in the database netlist from a previous compilation.

**Why Create a Floorplan?**

Creating a design floorplan is usually required if you want to preserve placement for partitions that will be exported, to avoid resource conflicts between partitions in the top-level design. Floorplan location planning can be important for a design that uses incremental compilation, for the following reasons:

- To avoid resource conflicts between partitions, predominantly when integrating partitions exported from another Quartus II project.
- To ensure good quality of results when recompiling individual timing-critical partitions.

Location assignments for each partition ensure that there are no placement conflicts between partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating location, no device resources are specifically allocated for the logic associated with the region. If you do not clearly define resource allocation, logic placement can conflict when you integrate the partitions in the top-level design if you reuse the placement information from the exported netlist.

Creating a floorplan is also recommended for timing-critical partitions that have little timing margin to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions compiled in the same Quartus II project. The logic for partitions that are not timing-critical can be placed anywhere in the device on each recompilation if that is best for your design.

Design floorplan assignments prevent the situation in which the Fitter must place a partition in an area of the device where most resources are used by other partitions. A LogicLock region provides a reasonable region to re-place logic after a change, so the Fitter does not have to scatter logic throughout the available space in the device.
Figure 15–24 illustrates the problems that may be associated with refitting designs that do not have floorplan location assignments. The left floorplan shows the initial placement of a four-partition design (P1-P4) without any floorplan location assignments. The right floorplan shows the device if a change occurs to P3. After removing the logic for the changed partition, the Fitter must re-place and reroute the new logic for P3 in the scattered white space shown in Figure 15–24. The placement of the post-fit netlists for other partitions forces the Fitter to implement P3 with the device resources that have not been used.

![Figure 15–24. Representation of Device Floorplan without Location Assignments](image)

The Fitter has a more difficult task because of more difficult physical constraints, and as a result, compilation time often increases. The Fitter might not be able to find any legal placement for the logic in partition P3, even if it could in the initial compilation. Additionally, if the Fitter can find a legal placement, the quality of results often decreases in these cases, sometimes dramatically, because the new partition is now scattered throughout the device.

Figure 15–25 shows the initial placement of a four-partition design with floorplan location assignments. Each partition is assigned to a LogicLock region. The second part of the figure shows the device after partition P3 is removed. This placement presents a much more reasonable task to the Fitter and yields better results.

![Figure 15–25. Representation of Device Floorplan with Location Assignments](image)

Altera recommends that you create a LogicLock floorplan assignment for timing-critical blocks with little timing margin that will be recompiled as you make changes to the design.
When to Create a Floorplan

It is important that you plan early to incorporate partitions into the design, and ensure that each partition follows partitioning guidelines. You can create floorplan assignments at different stages of the design flow, early or late in the flow. These guidelines help ensure better results as you begin creating floorplan location assignments.

Early Floorplan

An early floorplan is created before the design stage. You can plan an early floorplan at the top level of a design to allocate each partition a portion of the device resources. Doing so allows the designer for each block to create the logic for their design partition without conflicting with other logic. Each partition can be optimized in a separate Quartus II project if required, and the design can still be easily integrated in the top-level design. Even within one Quartus II project, each partition can be locked down with a post-fit netlist, and you can be sure there is space in the device floorplan for other partitions.

When you have compiled your complete design, or after you have integrated the first versions of partitions developed in separate Quartus II projects, you can use the design information and Quartus II features to tune and improve the floorplan, as described in the following section.

Late Floorplan

A late floorplan is created or modified after the design is created, when the code is close to complete and the design structure is likely to remain stable. Creating a late floorplan is typically necessary only if you are starting to use incremental compilation late in the design flow, or need to reserve space for a logic block that becomes timing-critical but still has HDL changes to be integrated. When the design is complete, you can take advantage of the Quartus II analysis features to check the floorplan quality. To adjust the floorplan, you can perform iterative compilations as required and assess the results of different assignments.

It may not be possible to create a good-quality late floorplan if you do not create partitions in the early stages of the design.

Design Floorplan Placement Guidelines

The following guidelines are key to creating a good design floorplan:

- Capture correct resources in each region.
- Use good region placement to maintain design performance compared to flat compilation.

A common misconception is that creating a floorplan enhances timing performance, as compared to a flat compilation with no location assignments. The Fitter does not usually require guidance to get optimal results for a full design.
Floorplan assignments can help maintain good performance when designs change incrementally, as described in “Why Create a Floorplan?” on page 15–41. However, poor placement assignments in an incremental compilation can often adversely affect performance results, as compared to a flat compilation, because the assignments limit the options for the Fitter. Investing time to find good region placement is required to match the performance of a full flat compilation.

Use the following general procedure to create a floorplan:

1. Divide the design into partitions.
2. Assign the partitions to LogicLock regions.
3. Compile the design.
4. Analyze the results.
5. Modify the placement and size of regions, as required.

You might have to perform these steps several times to find the best combination of design partitions and LogicLock regions that meet the resource and timing goals of the design.

For more information about performing these steps, refer to the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.

Assigning Partitions to LogicLock Regions

Before compiling a design with new LogicLock assignments, ensure that the partition netlist type is set to Post-Synthesis or Source File, so that the Fitter does not reuse previous placement results.

In most cases, you should include logic from one partition in each LogicLock region. This organization helps to prevent resource conflicts when partitions are exported and can lead to better performance preservation when locking down parts of a design in a single project.

The Quartus II software is flexible and allows exceptions to this rule. For example, you can place more than one partition in the same LogicLock region if the partitions are tightly connected, but you do not want to merge the partitions into one larger partition. For best results, ensure that you recompile all partitions in the LogicLock region every time the logic in one partition changes. Additionally, if a partition contains multiple lower-level entities, you can place those entities in different areas of the device with multiple LogicLock regions, even if they are defined in the same partition.

You can use the Reserved LogicLock option to ensure that you avoid conflicts with other logic that is not locked into a LogicLock region. This option prevents other logic from being placed in the region, and is useful if you have empty partitions at any point during your design flow, so that you can reserve space in the floorplan. Do not make reserved regions too large to prevent unused area because no other logic can be placed in a region with the Reserved LogicLock option.

For more information about LogicLock region properties, refer to the LogicLock Region Properties Dialog Box in Quartus II Help.
How to Size and Place Regions

In an early floorplan, assign physical locations based on design specifications. Use information about the connections between partitions, the partition size, and the type of device resources required.

In a late floorplan, when the design is complete, you can use locations or regions chosen by the Fitter as a guideline. If you have compiled the full design, you can view the location of the partition logic in the Chip Planner. Refer to “Checking Partition Quality” on page 15–30 for information about viewing placement results for each partition in the device floorplan. You can use the natural grouping of each unconstrained partition as a starting point for a LogicLock region constraint. View the placement for each partition that requires a floorplan constraint, and create a new LogicLock region by drawing a box around the area on the floorplan, and then assigning the partition to the region to constrain the partition placement.

For a step-by-step procedure to create a LogicLock region, refer to Creating and Manipulating LogicLock Regions in Quartus II Help.

Instead of creating regions based on the previous compilation results, you can start with the Fitter results for a default auto size and floating origin location for each new region when the design logic is complete. After compilation, lock the size and origin location. Instead of a full compilation, you can use the Start Early Timing Estimate command to perform a fast placement.

Alternatively, if the design logic is complete with auto-sized or floating location regions, you can specify the size based on the synthesis results and use the locations chosen by the Fitter with the Set to Estimated Size command. Like the previous option, start with floating origin location. After compilation, lock the origin location. Again, instead of a full compilation, you can use the Start Early Timing Estimate command to perform a fast placement. You can also enable the Fast Synthesis Effort setting to reduce synthesis time.

After a compilation or early timing estimate, save the Fitter size and origin location of the Fitter with the Set Size and Origin to Previous Fitter Results command.

It is important that you use the Fitter-chosen locations only as a starting point to give the regions a good fixed size and location. Ensure that all LogicLock regions in the design have a fixed size and have their origin locked to a specific location on the device. On average, regions with fixed size and location yield better timing performance than auto-sized regions.

Modifying Region Size and Origin

After saving the Fitter results from an initial compilation for a late floorplan, modify the regions using your knowledge of the design to set a specific size and location. If you have a good understanding of how the design fits together, you can often improve upon the regions placed in the initial compilation. In an early floorplan, when the design has not yet been created, you can use the guidelines in this section to set the size and origin, even though there is no initial Fitter placement.
The easiest way to move and resize regions is to drag the region location and borders in the Chip Planner. Make sure that you select the User-Defined region in the floorplan (as opposed to the Fitter-Placed region from the last compilation) so that you can change the region.

Generally, you can keep the Fitter-determined relative placement of the regions, but make adjustments if required to meet timing performance. If you find that the early timing estimate did not result in good relative placements, try performing a full compilation so that the Fitter can optimize for a full placement and routing.

If two LogicLock regions have several connections between them, ensure they are placed near each other to improve timing performance. By placing connected regions near each other, the Fitter has more opportunity to optimize inter-region paths when both partitions are recompiled. Reducing the criticality of inter-region paths also allows the Fitter more flexibility when placing other logic in each region.

If resource utilization is low in the overall device, enlarge the regions. Doing so usually improves the final results because it gives the Fitter more freedom to place additional or modified logic added to the partition during subsequent incremental compilations. It also allows room for optimizations such as pipelining and physical synthesis logic duplication.

Try to have each region evenly full, with the same "fullness" that the complete design would have without LogicLock regions; Altera recommends approximately 75% full.

Allow more area for regions that are densely populated, because overly congested regions can lead to poor results. Allow more empty space for timing-critical partitions to improve results. However, do not make regions too large for their logic. Regions that are too large can result in wasted resources and also lead to suboptimal results.

Ideally, almost the entire device should be covered by LogicLock regions if all partitions are assigned to regions.

Regions should not overlap in the device floorplan. If two partitions are allocated on an overlapping portion of the chip, each may independently claim common resources in this region. This leads to resource conflicts when integrating results into a top-level design. In a single project, overlapping regions give more difficult constraints to the Fitter and can lead to reduced quality of results.

You can create hierarchical LogicLock regions to ensure that the logic in a child partition is physically placed inside the LogicLock region for its parent partition. This can be useful when the parent partition does not contain registers at the boundary with the lower-level child partition and has a lot of signal connectivity. To create a hierarchical relationship between regions in the LogicLock Regions window, drag and drop the child region to the parent region.

**I/O Connections**

Consider I/O timing when placing regions. Using I/O registers can minimize I/O timing problems, and using boundary registers on partitions can minimize problems connecting regions or partitions. However, I/O timing might still be a concern. It is most important for flows where each partition is compiled independently, because the Fitter can optimize the placement for paths between partitions if the partitions are compiled at the same time.
Place regions close to the appropriate I/O, if necessary. For example, DDR memory interfaces have very strict placement rules to meet timing requirements. Incorporate any specific placement requirements into your floorplan as required. You should create LogicLock regions for internal logic only, and provide pin location assignments for external device I/O pins (instead of including the I/O cells in a LogicLock region to control placement).

**LogicLock Resource Exclusions**

You can exclude certain resource types from a LogicLock region to manage the ratio of logic to dedicated DSP and RAM resources in the region.

If your design contains memory or Digital Signal Processing (DSP) elements, you may want to exclude these elements from the LogicLock region. LogicLock resource exceptions prevent certain types of elements from being assigned to a region. Therefore, those elements are not required to be placed inside the region boundaries. The option does not prevent them from being placed inside the region boundaries unless the **Reserved** property of the region is turned on.

Resource exceptions are useful in cases where it is difficult to place rectangular regions for design blocks that contain memory and DSP elements, due to their placement in columns throughout the device floorplan. Exclude RAMs, DSPs, or logic cells to give the Fitter more flexibility with region sizing and placement. Excluding RAM or DSP elements can help to resolve no-fit errors that are caused by regions spanning too many resources, especially for designs that are memory-intensive, DSP-intensive, or both. Figure 15–26 shows an example of a design with an odd-shaped region to accommodate DSP blocks for a region that does not contain very much logic. The right side of the figure shows the result after excluding DSP blocks from the region. The region can be placed more easily without wasting logic resources.

**Figure 15–26. LogicLock Resource Exclusion Example**

![LogicLock Resource Exclusion Example](image-url)
To view any resource exceptions, right-click in the LogicLock Regions window, and then click LogicLock Regions Properties. In the LogicLock Regions Properties dialog box, select the design element (module or entity) in the Members box, and then click Edit. In the Edit Node dialog box, to set up a resource exception, click the Edit button next to the Excluded element types box, and then turn on the design element types to be excluded from the region. You can choose to exclude combinational logic or registers from logic cells, or any of the sizes of TriMatrix memory blocks, or DSP blocks.

If the excluded logic is in its own lower-level design entity (even if it is within the same design partition), you can assign the entity to a separate LogicLock region to constrain its placement in the device.

You can also use this feature with the LogicLock Reserved property to reserve specific resources for logic that will be added to the design.

Creating Floorplan Location Assignments With Tcl Commands—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks)

To assign a code block to a LogicLock region, with exclusions, use the following command:

```
set_logiclock_contents -region <LogicLock region name> -to <block>
-exceptions "<keyword>:<keyword>"
```

- `<LogicLock region name>`—The name of the LogicLock region to which the code block is assigned.
- `<block>`—A code block in a Quartus II project hierarchy, which can also be a design partition.
- `<keyword>`—The list of exceptions made during assignment. For example, if DSP was in the keyword list, the named block of code would be assigned to the LogicLock region, except for any DSP block within the code block. You can include the following exceptions in the set_logiclock_contents command:

Keyword variables:

- REGISTER—Any registers in the logic cells.
- COMBINATIONAL—Any combinational elements in the logic cells.
- SMALL_MEM—Small TriMatrix memory blocks (M512 or MLAB).
- MEDIUM_MEM—Medium TriMatrix memory blocks (M4K or M9K).
- LARGE_MEM—Large TriMatrix memory blocks (M-RAM or M144K).
- DSP—Any DSP blocks.
- VIRTUAL_PIN—Any virtual pins.

Resource filtering uses the optional Tcl argument -exclude_resources in the set_logiclock_contents function. If left unspecified, no resource filter is created. In the .qsf, resource filtering uses an extra LogicLock membership assignment called LL_MEMBER_RESOURCE_EXCLUDE. For example, the following line in the .qsf is used to specify a resource filter for the alu:alu_unit entity assigned to the ALU region.

```
set_instance_assignment -name LL_MEMBER_RESOURCE_EXCLUDE "DSP:SMALL_MEM" -to "alu:alu_unit" -section_id ALU
```
Creating Non-Rectangular Regions

To constrain placement to non-rectangular or non-contiguous areas of the device, you can connect multiple rectangular regions together using the **Merge** command.

For devices that do not support the **Merge** command (Arria™ GX, Cyclone, Cyclone II, MAX™ II, Stratix, Stratix II, Stratix II GX, and Stratix GX devices), you can limit entity placement to a sub-area of a LogicLock region to create non-rectangular constraints. In these devices, construct a LogicLock hierarchy by creating child regions inside of parent regions, and then use the **Reserved** option to control which logic can be placed inside these child regions. Setting the **Reserved** option for the region prevents the Fitter from placing nodes that are not assigned to the region inside the boundary of the region.

For more information and examples of creating non-rectangular regions, refer to "Creating and Manipulating LogicLock Regions" in Quartus II Help.

Checking Floorplan Quality

This section provides an overview of tools that you can use as you create a floorplan in the Quartus II software. You can use these tools to assess your floorplan quality and use the information to improve your design or assignments as required to achieve the best results.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows the recommendations for creating floorplan location assignments that are presented in this chapter. For more information, refer to “Incremental Compilation Advisor” on page 15–30.

LogicLock Region Resource Estimates

You can view resource estimates for a LogicLock region to determine the region’s resource coverage, and use this estimate before compilation to check region size. Using this estimate helps to ensure adequate resources when you are sizing or moving regions.

For information about how to view the properties of a LogicLock region, refer to "LogicLock Region Properties Dialog Box" in Quartus II Help.

LogicLock Region Properties Statistics Report

LogicLock region statistics are similar to design partition properties, but also include resource usage details after compilation.

The statistics report the number of resources used and the total resources covered by the region, and also list the number of I/O connections and how many I/Os are registered (good), as well as the number of internal connections and the number of inter-region connections (bad).
Checking Floorplan Quality

For information about the Statistics tab in the LogicLock Region Properties dialog box, refer to LogicLock Region Properties Dialog Box in Quartus II Help.

Locate the Quartus II TimeQuest Timing Analyzer Path in the Chip Planner

In the TimeQuest analyzer user interface, you can locate a specific path in the Chip Planner to view its placement and perform a report timing operation (for example, report timing for all paths with less than 0 ns slack).

For information about how to locate paths between the TimeQuest analyzer and the Chip Planner, refer to Locate Dialog Box in Quartus II Help.

Inter-Region Connection Bundles

The Chip Planner can display bundles of connections between LogicLock regions, with filtering options that allow you to choose the relevant data for display. These bundles can help you to visualize how many connections there are between each LogicLock region to improve floorplan assignments or to change partition assignments, if required.

For information about how to display bundles of connections between LogicLock regions, refer to Inter-region Bundles Dialog Box in Quartus II Help.

Routing Utilization

The Chip Planner includes a feature to display a color map of routing congestion. This display helps identify areas of the chip that are too tightly packed.

In the Chip Planner, red LAB blocks indicate higher routing congestion. You can position the mouse pointer over a LAB to display a tooltip that reports the logic and routing utilization information.

For information about how to view a color map of routing congestion in the Chip Planner, refer to About the Chip Planner in Quartus II Help.

Ensure Floorplan Assignments Do Not Significantly Impact Quality of Results

The end results of design partitioning and floorplan creation differ from design to design. However, it is important to evaluate your results to ensure that your scheme is successful. Compare your before and after results, and consider using another scheme if any of the following guidelines are not met:

- You should see only minor degradation in $f_{\text{MAX}}$ after the design is partitioned and floorplan location assignments are created. There is some performance cost associated with setting up a design for incremental compilation; approximately 3% is typical.
- The area increase should be no more than 5% after the design is partitioned and floorplan location assignments are created.
- The time spent in the routing stage should not significantly increase.
The amount of compilation time spent in the routing stage is reported in the Messages window with an Info message that indicates the elapsed time for Fitter routing operations. If you notice a dramatic increase in routing time, the floorplan location assignments may be creating substantial routing congestion. In this case, decrease the number of LogicLock regions, which typically reduces the compilation time in subsequent incremental compilations and may also improve design performance.

Recommended Design Flows and Application Examples

This section provides design flows for partitioning and creating a design floorplan during common timing closure and team-based design scenarios. Each flow describes the situation in which it should be used, and provides a step-by-step description of the commands required to implement the flow.

Create a Floorplan for Major Design Blocks

Use this incremental compilation flow for designs when you want to assign a floorplan location for each major block in your design. A full floorplan ensures that partitions do not interact as they are changed and recompiled—each partition has its own area of the device floorplan.

To create a floorplan for major design blocks, follow this general methodology:

1. In the Design Partitions window, ensure that all partitions have their netlist type set to **Source File** or **Post-Synthesis**. If the netlist type is set to **Post-Fit**, floorplan location assignments are not used when recompiling the design.

2. Create a LogicLock region for each partition (including the top-level entity, which is set as a partition by default).

3. Run a full compilation of your design to view the initial Fitter-chosen placement of the LogicLock regions as a guideline.

4. In the Chip Planner, view the placement results of each partition and LogicLock region on the device.

5. If required, modify the size and location of the LogicLock regions in the Chip Planner. For example, enlarge the regions to fill up the device and allow for future logic changes.

   You can also, if needed, create a new LogicLock region by drawing a box around an area on the floorplan.

6. Run an early timing estimate with the **Start Early Timing Estimate** command to estimate the timing performance of your design with the modified or new LogicLock regions.

7. Repeat steps 5 and 6 until you are satisfied with the quality of results for your design floorplan. Once you are satisfied with your results, run a full compilation of your design.
Create a Floorplan Assignment for One Design Block with Difficult Timing

Use this flow when you have one timing-critical design block that requires more optimization than the rest of your design. You can take advantage of incremental compilation to reduce your compilation time without creating a full design floorplan.

In this scenario, you do not want to create floorplan assignments for the entire design. Instead, you can create a region to constrain the location of your critical design block, and allow the rest of the logic to be placed anywhere on the device. To create a region for critical design block, follow these steps:

1. Divide up your design into partitions. Consider the guidelines in “Design Partition Guidelines” on page 15–10 to determine partition boundaries. Ensure that you isolate the timing-critical logic in a separate partition.

2. Define a LogicLock region for the timing-critical partition. Ensure that you capture the correct amount of device resources in the region. Turn on the **Reserved** property to prevent any other logic from being placed in the region.
   - If the design block is not complete, reserve space in the design floorplan based on your knowledge of the design specifications, connectivity between design blocks, and estimates of the size of the partition based on any initial implementation numbers.
   - If the critical design block has initial source code ready, compile the design to place the LogicLock region. Save the Fitter-determined size and origin, and then enlarge the region to provide more flexibility and allow for future design changes.

As the rest of the design is completed, and the device fills up, the timing-critical region reserves an area of the floorplan. When you make changes to the design block, the logic will be re-placed in the same part of the device, which helps ensure good quality of results.

Create a Floorplan as the Project Lead in a Team-Based Flow

Use this approach when you have several designs that will be implemented in separate Quartus II projects by different designers, or third-party IP designers who want to optimize their designs independently and pass the results to the project lead.

As the project lead in this scenario, follow these steps to prepare the top-level design for a successful team-based design methodology with early floorplan planning:

1. Create a new Quartus II project that will ultimately contain the full implementation of the entire design.

2. Create a “skeleton” or framework of the design that defines the hierarchy for the subdesigns that will be implemented by separate designers. Consider the partitioning guidelines in this chapter when determining the design hierarchy.

3. Make project-wide settings. Select the device, make global assignments for clocks and device I/O ports, and make any global signal constraints to specify which signals can use global routing resources.

4. Make design partition assignments for each major subdesign. Set the netlist type for each partition that will be implemented in a separate Quartus II project and later exported and integrated with the top-level design set to **Empty**.
5. Create LogicLock regions for each partition to create a design floorplan. This floorplan should consider the connectivity between partitions and estimates of the size of each partition based on any initial implementation numbers and knowledge of the design specifications. Use the guidelines described in this chapter to choose a size and location for each LogicLock region.

6. Provide the constraints from the top-level design to partition designers using one of the following procedures:
   a. Create a copy of the top-level Quartus II project framework by checking out the appropriate files from a source control system, using the Copy Project command, or creating a project archive. Provide each partition designer with the copy of the project.
   b. Provide the constraints with documentation or scripts.

To use design partition scripts to pass constraints and generate separate Quartus II projects, refer to *Generating Design Partition Scripts for Project Management* in Quartus II Help.

**Conclusion**

Incremental compilation can significantly improve your design productivity, especially for large, complex designs. To take advantage of the feature, it is worth spending time to create quality partition and floorplan assignments. Follow the guidelines to set up your design hierarchy and source code for incremental compilation.

Floorplan location assignments are required when design blocks are developed independently and are recommended for timing-critical partitions that are expected to change. Follow the guidelines to create and modify LogicLock regions to create good placement assignments for your design partitions.

Remember that you do not have to follow all the guidelines exactly to implement an incremental compilation design flow, but following the guidelines can maximize your chances of success.

**Document Revision History**

Table 15–1 shows the revision history for this chapter.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>November 2013</td>
<td>13.1.0</td>
<td>Removed HardCopy device information.</td>
</tr>
<tr>
<td>June 2012</td>
<td>12.0.0</td>
<td>Removed survey link.</td>
</tr>
<tr>
<td>November 2011</td>
<td>11.0.1</td>
<td>Template update.</td>
</tr>
<tr>
<td>May 2011</td>
<td>11.0.0</td>
<td>Updated links.</td>
</tr>
<tr>
<td>Date</td>
<td>Version</td>
<td>Changes</td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>December 2010</td>
<td>10.1.0</td>
<td>• Changed to new document template.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Moved &quot;Creating Floorplan Location Assignments With Tcl Commands—Excluding or Filtering Certain Device Elements (Such as RAM or DSP Blocks)&quot; from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Consolidated Design Partition Planner and Incremental Compilation Advisor information between the Quartus II Incremental Compilation for Hierarchical and Team-Based Design and Best Practices for Incremental Compilation Partitions and Floorplan Assignments handbook chapters.</td>
</tr>
<tr>
<td>July 2010</td>
<td>10.0.0</td>
<td>• Removed the explanation of the “bottom-up design flow” where designers work completely independently, and replaced with Altera’s recommendations for team-based environments where partitions are developed in the same top-level project framework, plus an explanation of the bottom-up process for including independent partitions from third-party IP designers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Expanded the Merge command explanation to explain how it now accommodates cross-partition boundary optimizations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Restructured Altera recommendations for when to use a floorplan.</td>
</tr>
<tr>
<td>October 2009</td>
<td>9.1.0</td>
<td>• Redefined the bottom-up design flow as team-based and reorganized previous design flow examples to include steps on how to pass top-level design information to lower-level projects.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added &quot;Including SDC Constraints from Lower-Level Partitions for Third-Party IP Delivery&quot; from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the Quartus II Handbook.</td>
</tr>
<tr>
<td>March 2009</td>
<td>9.0.0</td>
<td>• Added I/O register packing examples from Incremental Compilation for Hierarchical and Team-Based Designs chapter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Moved “Incremental Compilation Advisor” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Added “Viewing Design Partition Planner and Floorplan Side-by-Side” section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Updated Figure 15-22</td>
</tr>
<tr>
<td>November 2008</td>
<td>8.1.0</td>
<td>• Changed to 8-1/2 x 11 page size. No change to content.</td>
</tr>
<tr>
<td>May 2007</td>
<td>8.0.0</td>
<td>• Initial release.</td>
</tr>
</tbody>
</table>

For previous versions of the Quartus II Handbook, refer to the Quartus II Handbook Archive.