8. Best Practices for Incremental Compilation Partitions and Floorplan Assignments

QII51017-9.1.0

This chapter provides a set of 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 to support the flow.

Introduction

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

- Reduces compilation times by as much as 70%
- Preserves performance for unchanged design blocks
- Provides repeatable results and reduces the number of compilations
- Enables true team-based design

For more information about feature usage 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*.

This document contains the following sections:

- "Overview: Incremental Compilation" on page 8–2
- "Why Plan Partitions and Floorplan Assignments for Incremental Compilation?" on page 8–5
- "Creating Design Partitions: General Partitioning Guidelines" on page 8–6
- "Creating Design Partitions: Design Guidelines" on page 8–8
- "Creating Design Partitions: Consider Additional Design Suggestions" on page 8–23
- "Checking Partition Quality" on page 8–29
- "Importing SDC Constraints from Lower-Level Partitions in Team-Based Designs" on page 8–35
- "Introduction to Design Floorplans" on page 8–39
- "Creating a Design Floorplan: Placement Guidelines" on page 8–42
- "Checking Floorplan Quality" on page 8–47
- "Recommended Design Flows and Application Examples" on page 8–48
- "Potential Issues with Creating Partitions and Floorplan Assignments" on page 8–51

Overview: Incremental Compilation

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

The following procedure outlines the general Quartus II incremental compilation flow:

1. Set up your design hierarchy and source code to support partitioning along logical hierarchy boundaries. If you use a third-party synthesis tool, set up your tool to generate separate netlist files for each partition.
2. Create design partition assignments in the Quartus II software to specify which hierarchy blocks are compiled independently as partitions (including empty partitions for any missing or incomplete logic blocks).
3. During design compilation, Quartus II Analysis and Synthesis and the Fitter create separate netlists for each partition. These netlists are internal post-synthesis and post-fit database representations of your design.
4. Select which netlist type 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 import compilation results from another project, as described in "Incremental and Team-Based Design Flows".
5. After part of the design changes, the software recompiles only the required partitions and merges the new compilation results with existing netlists for other partitions, according to the settings from step 4.

In some cases, as described in "Introduction to Design Floorplans" on page 8–39, you should create a design floorplan with placement assignments to constrain each part of the design to a specific region of the device.

Incremental and Team-Based Design Flows

The Quartus II incremental compilation feature supports various design flows. Your design flow affects how much impact design partitions have on design optimization.

For more information about the different types of incremental design flows and example applications, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

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 source code is not yet complete for a design partition, you can create a placeholder for the partition until the code is ready and added to the top-level design. To enable team-based development and third-party IP delivery, you can design and optimize each partition in isolation, and later integrate the results into the top-level design with the Quartus II software export and import features.

Keeping design partitions in one Quartus II project is generally a more simple design flow to use than when partitions are imported from separate Quartus II projects. Keeping all design partitions in one project provides the Quartus II software with information about the entire design, allowing it to perform global placement and routing optimizations. Therefore, it is often easier to ensure good quality of results when partitions are imported from other team members or third-party IP providers.

You can combine design flows and use imported partitions when it is necessary to support your design environment. If the top-level design includes one or more design blocks that are optimized by remote designers or IP providers, you can import those blocks into a project that also includes partitions for a standard incremental design flow. In addition, as you perform timing closure for a design, you can create a subproject for one block of the design to be optimized by another designer in a separate Quartus II project, and pass information about the rest of the design to the subproject to obtain the best results.

You cannot use an imported partition if you want to migrate to a HardCopy ASIC. The Revision Compare feature requires that the HardCopy and FPGA netlists are the same, and all operations performed on one revision must also occur on the other revision. Unfortunately, importing partitions does not support this requirement.

Recommendations for the Netlist Type and Fitter Preservation Level

You must specify which post-compilation netlist you want to use in subsequent compilations by specifying a **Netlist Type** setting for each partition. For post-fit netlists, you can also specify a **Fitter Preservation Level** setting to indicate the amount of fitting information you want to preserve. Use the following general guidelines for these standard Netlist Type settings:

- **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 appropriate partitions with standard Netlist Type settings, so setting the partition to **Source File** is optional in this case
  - Most assignments do not trigger an automatic recompilation, so setting the partition to **Source File** is required 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
- Post-Fit: Use this setting to preserve Fitter and performance results
  - The default setting for post-fit is to use the highest available level of netlist preservation
- Post-Fit with Fitter Preservation Level set to **Placement**: Use these settings to allow more flexibility to find the best routing for all partitions given their placement on the design. Although routing can change with these options, there is typically very good performance preservation.

The Quartus II software also includes a Rapid Recompile feature, which allows you to reuse previous compilation results for unchanged logic when you have changed a very small portion of the design. You can set the **Rapid Recompile** option to preserve compatible placement or compatible placement and routing to reduce compilation time when you make small changes inside a partition or the full design. If you choose a netlist type that specifies recompilation and the **Rapid Recompile** option is turned on, then the specified compatible compilation results are preserved and reused. To ensure you compile from new source files with no compilation results reused, you can turn off the **Rapid Recompile** option.

Project Management in Team-Based Designs

In a team-based design methodology in which some partitions are developed independently, the project lead must pass top-level constraints (such as floorplan and pin assignments, timing constraints, and optimization settings) to the designers of lower-level partitions.

One option is for the lead designer to make a copy of the top-level project framework for all team members. This option ensures that all design developers have all the settings and constraints needed for the design and makes design integration easier. Each lower-level project designer can export their completed design as a partition, and the lead designer can then integrate each partition into the top-level design.

An alternate option is for each lower-level project designer to 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 design framework. In this case, each designer of a lower-level project must create a project with all the relevant assignments and constraints. When lower-level projects are developed independently, it is sometimes referred to as a bottom-up design methodology.

The bottom-up design partition script provide a project manager interface for managing resource and timing budgets in the top-level design. This interface makes it easier for designers of independent lower-level projects to implement the instructions from the project lead, and avoid conflicts between projects when importing and incorporating the projects into the top-level design. Using the scripts also helps reduce the need for further optimization to the designs after integration and improves overall designer productivity and team collaboration.

The scripting feature creates Tcl files that an independent designer can run to set up a project and makefiles for designers who use a make environment. To use this feature, first set up the top-level project with appropriate constraints and floorplan assignments to be passed to lower levels. Then generate design partition scripts after successful compilation of the top-level design. You can perform a Fast Synthesis and Early Timing Estimation instead of full compilation to reduce compilation time. The top-level design can have empty partitions when you generate the scripts. To generate the scripts, on the Project menu, click **Generate Bottom-Up Design Partition Scripts** and set the appropriate options.

For more information about different design flows and features to support the flows, 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 for Incremental Compilation?

Incremental compilation flows require more up-front planning than flat compilations. For example, you might have 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. Incremental compilation generally requires you to be more rigorous about following good design practices than flat compilations.

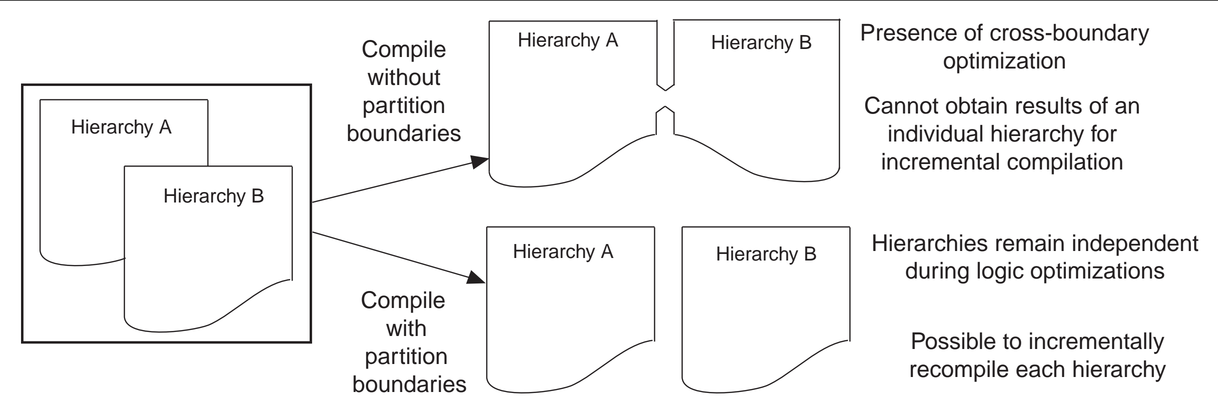
Planning involves setting up the design logic for partitioning and may involve planning placement assignments to create a floorplan. Not all design flows require floorplan assignments. If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation much easier. Poor partition or floorplan assignments can worsen design area utilization and performance, making timing closure more difficult.

As FPGA devices get larger and more complex, following good design practices becomes more important for all design flows.These planning issues are similar to the requirements for a multiple-chip solution if you were using smaller devices, although planning for one chip is much easier. Adhering to the 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 times.

Partition Boundaries and Optimization

If there are any cross-boundary optimizations between partitions, the software cannot obtain separate results for each individual partition. The logical hierarchical boundaries between partitions are treated as hard boundaries for logic optimization to allow the software to synthesize and place each partition independently. Figure 8–1 shows the effects of partition boundaries during logic optimization. It is important to understand this effect so that you can effectively plan your design partitions.

Figure 8–1. Effects of Partition Boundaries During Logic Optimization



To avoid cross-boundary optimizations, the software synthesizes each partition without using any information about logic in other partitions. In a flat compilation, the software uses unconnected signals, constants, inversions, and other design information to perform optimizations. When you partition a design, these types of optimizations do not take place on partition I/O ports. Good design partitions do not rely on these types of logic optimizations.

When all partitions are placed together, the Fitter can perform placement optimizations on the design as a whole to optimize the placement of cross-partition paths. However, the Fitter can never perform any 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.

Because cross-boundary logic and placement optimizations cannot occur, the quality of results may decrease as the number of partitions increases. Although more partitions allow for 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 improve the performance results for cross-partition paths. Guidelines for creating these assignments are discussed in the following sections.

Creating Design Partitions: General Partitioning Guidelines

The first stage 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, following the design guidelines presented in this section ensures better results. This section includes the following topics:

■ “Plan Design Hierarchy and Source Design Files” on page 8–6
■ “Partition Design by Functionality and Block Size” on page 8–7
■ “Partition Design by Clock Domain and Timing Criticality” on page 8–8
■ “Consider What Is Changing” on page 8–8

Plan Design Hierarchy and Source Design Files

Start by planning the design hierarchy. When you assign a hierarchical instance as a design partition, the partition includes the assigned instance and any entities instantiated below it that are not defined as separate partitions. You can also use the **Merge** command to combine hierarchical partitions into a single partition, as long as they have the same immediate parent partition. However, in the Quartus II software version 9.0, logic is not merged or optimized across hierarchical blocks that are merged into the same partition.

Take advantage of the design hierarchy to provide flexibility for partitioning and to support different design flows. Keep logic in the “leaves” of the hierarchy tree instead of having a lot of logic at the top level of the design. Doing so ensures that you can isolate partitions if required.

Create entities that can lead to partitions of approximately equal size. For example, do not instantiate a lot of 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 compiler uses a file checksum to detect changes, and automatically recompiles a partition if its source file changes and their 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 initiate 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 affect both partitions. Commands such as VHDL `use` and Verilog HDL `` `include `` create dependencies between files, so that changes to one file can trigger recompilations in all dependent files. Avoid these types of file dependencies if they are not required. The **Partition Dependent Files** report for each partition in the **Analysis & Synthesis** folder of the Compilation Report lists which files contribute to each partition.

For more information about what changes initiate an automatic recompilation, refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*.

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 your tool vendor's documentation 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, partition your design along functional boundaries. In a top-level system block diagram, each block often is 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 how large each partition should be. How much compilation time reduction you want to achieve 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 be more than approximately 2,000 logic elements (LEs) or adaptive logic modules (ALMs). If your design is not yet complete when you partition the design, use previous designs to help you estimate the size that each block is likely to be.

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 going to be constrained to one region of the design. In addition, limiting the number of clocks within each partition simplifies the timing requirements for each partition during optimization. Use an appropriate subsystem to handle 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.

Try to isolate timing-critical logic from logic that you expect to meet its timing requirements easily. Doing so allows you to preserve the satisfactory results for non-critical partitions and focus optimization iterations on just 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, then you will 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.

Creating Design Partitions: Design Guidelines

Follow the partitioning guidelines presented in this section when creating or modifying the HDL code for each design block that you might want to assign as a design partition. Not all these recommendations have to be followed exactly to be successful with incremental compilation, but adhering to as many as possible maximizes your chances of success.

This section includes the following topics:

- “Register Partition Inputs and Outputs” on page 8–9
- “Minimize Cross-Partition-Boundary I/O” on page 8–9
- “Avoid the Need for Logic Optimization Across Partitions” on page 8–11

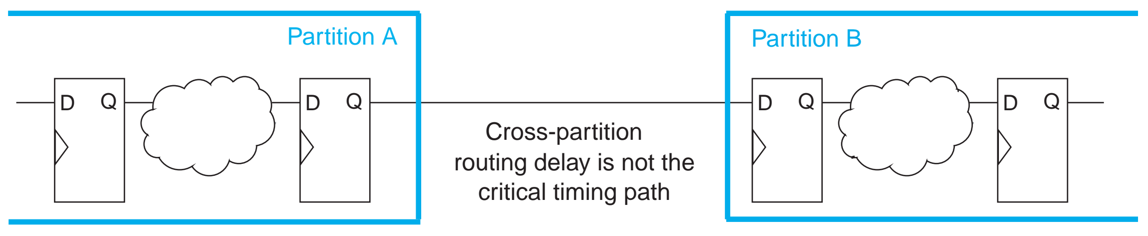
This last subsection includes examples of the types of optimizations that are prevented by partition boundaries, and describes how you can structure or modify your partitions to avoid such optimizations.

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 logic optimizations.

If every partition boundary has a register as shown in Figure 8–2, 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 each partition is placed completely independently in separate projects. In addition, the partition boundary does not affect combinational logic optimization because each register-to-register logic path is contained within a single partition.

Figure 8–2. 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 also 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.

The statistics described in “Partition Statistics Report” on page 8–33 list how many I/Os are registered or unregistered. The Incremental Compilation Advisor described on page 8–47 lists the unregistered ports for each partition.

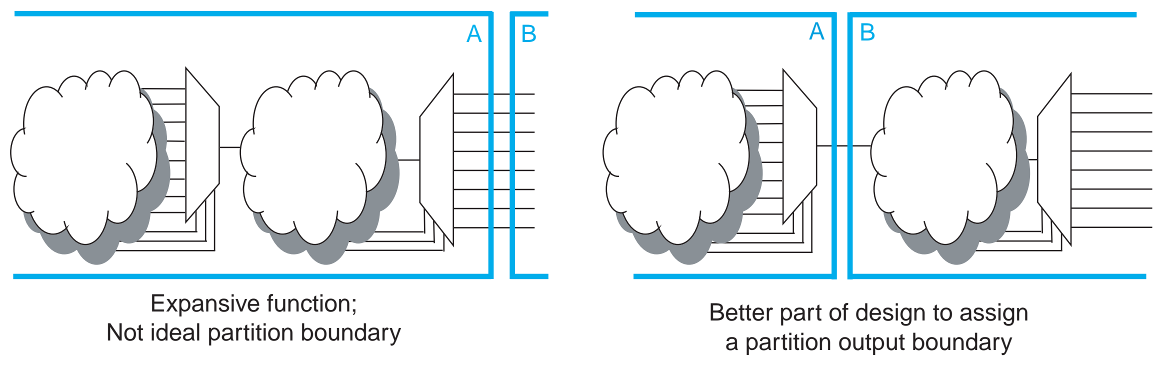
Minimize Cross-Partition-Boundary I/O

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 the 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 the partitions to avoid these inter-partition paths.

When dividing your design into partitions, consider the types of functions at the partition boundaries. Figure 8–3 shows an expansive function with more outputs than inputs on the left side, which makes a poor partition boundary, and a better place to assign the partition boundary that minimizes cross-partition I/Os on the right side. Adding registers to one or both sides of the cross-partition path in this example would improve the partition quality even more.

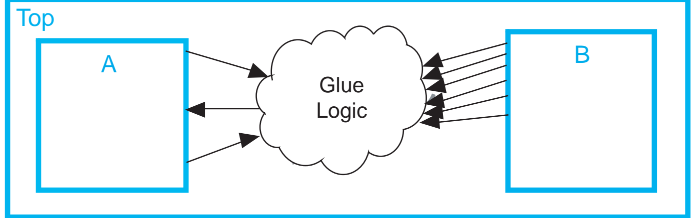
Figure 8–3. Minimizing I/O Between Partitions by Moving the Partition Boundary



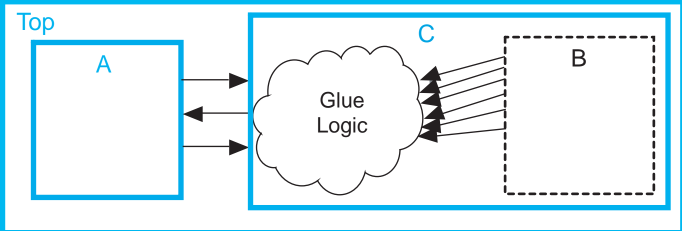
Another way to minimize connections between partitions is to avoid using combinational "glue logic" between partitions. You can often move the logic to the partition at one end of the connection to keep more logic paths within one partition. For example, the bottom diagram in Figure 8–4 includes a new level of hierarchy C that is defined as a partition instead of block B. It is clear that there are fewer I/O connections between partitions A and C than between partitions A and B in the top diagram.

Figure 8–4. Minimizing I/O between Partitions by Modifying Glue Logic

Many cross-partition paths: Poor design partition assignment



Fewer cross-partition paths: Better partitions



The statistics described in “Partition Statistics Report” on page 8–33 list the number of I/O ports as well as the number of inter-partition connections for each partition. The Incremental Compilation Advisor described on “Incremental Compilation Advisor” on page 8–47 lists the number of intra-partition (within a partition) and inter-partition (between partitions) timing edges.

Avoid the Need for Logic Optimization Across Partitions

As discussed in “Partition Boundaries and Optimization” on page 8–5, partition boundaries prevent logic optimizations across partitions. Remember this rule: Logic cannot be optimized or merged across a partition boundary.

To ensure correct and optimal logic optimization, follow the guidelines in this section. In some cases, especially if part of the design is complete or comes from another designer, these guidelines may not have been followed 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 provide suggestions for making simple changes to your partition definitions or hierarchy to move the partition boundary and improve your results.

These guidelines ensure that your design does not require any logic optimization across partitions:

■ “Keep Logic in the Same Partition for Optimization and Merging” on page 8–12
■ “Keep Constants in the Same Partition as Logic” on page 8–13
■ “Avoid Unconnected Partition I/O” on page 8–14
■ “Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together” on page 8–15

- "Invert Clocks in Destination Partitions" on page 8–16
- "Connect I/O Directly to I/O Register for Packing Across Partition Boundaries" on page 8–16
- "Do Not Use Internal Tri-States" on page 8–20
- "Include All Tri-State and Enable Logic in the Same Partition" on page 8–20
- "Include Bidirectional I/O Registers in the Same Partition" on page 8–21

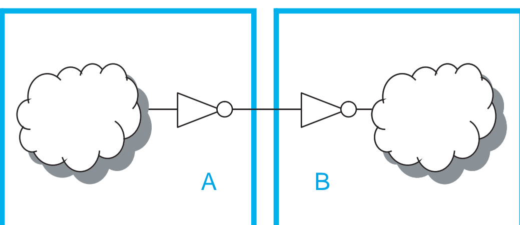
Keep Logic in the Same Partition for Optimization and Merging

If any design logic requires logic optimization or merging to obtain optimal results, ensure all the logic is part of the same partition.

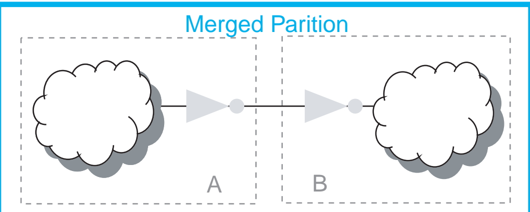
If a combinational logic path is split across two partitions, the logic cannot be optimized or merged into one logic cell in the device. This effect can result in an extra logic cell in the path, increasing the logic delay. As a very simple example, consider two inverters on the same signal in two different partitions, A and B, as shown in the left side of Figure 8–5. To maintain correct incremental functionality, these two inverters cannot be removed from the design during optimization because they occur in different design partitions. The 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 have been grouped into one partition C. 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 color), 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, it is a good demonstration of the types of logic optimization that are prevented by partition boundaries.

Figure 8–5. Keeping Logic in the Same Partition for Optimization



Inverters in separate partitions A and B cannot be removed from design: Poor design partition assignment



Inverters in merged partition can be removed: Better partition

In a flat design, the Quartus II 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 a concern only if merging is required to fit the design in the target device. Therefore, you are more likely to encounter this issue during troubleshooting 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.

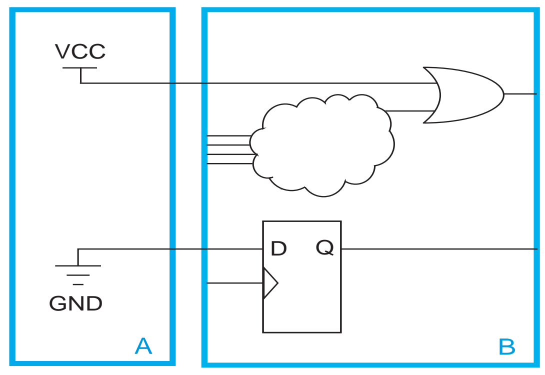
The Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions. Therefore, there are no restrictions for PLLs and high-speed transceiver blocks when setting up partitions.

Keep Constants in the Same Partition as Logic

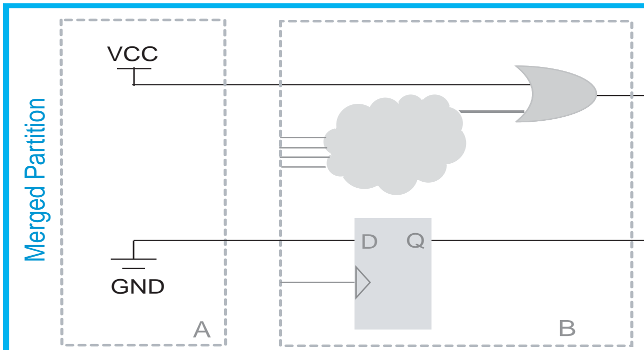
Because the software cannot optimize across a partition boundary, constants are not propagated across partition boundaries. A signal that is constant (1/V_{CC} or 0/GND) in one partition cannot affect another partition.

For example, the left side of Figure 8–6 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 like this could appear due to parameter/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 8–6 shows new partition C that groups the logic in blocks A and B. 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 partition, the software can use the constants to optimize and remove much of the logic in block B (shown in gray color).

Figure 8–6. Keeping Constants in the Same Partition as the Logic They Support



Connections to constants in another partition:
Poor design partition assignment



Constants in merged partition are used to optimize:
Better partition

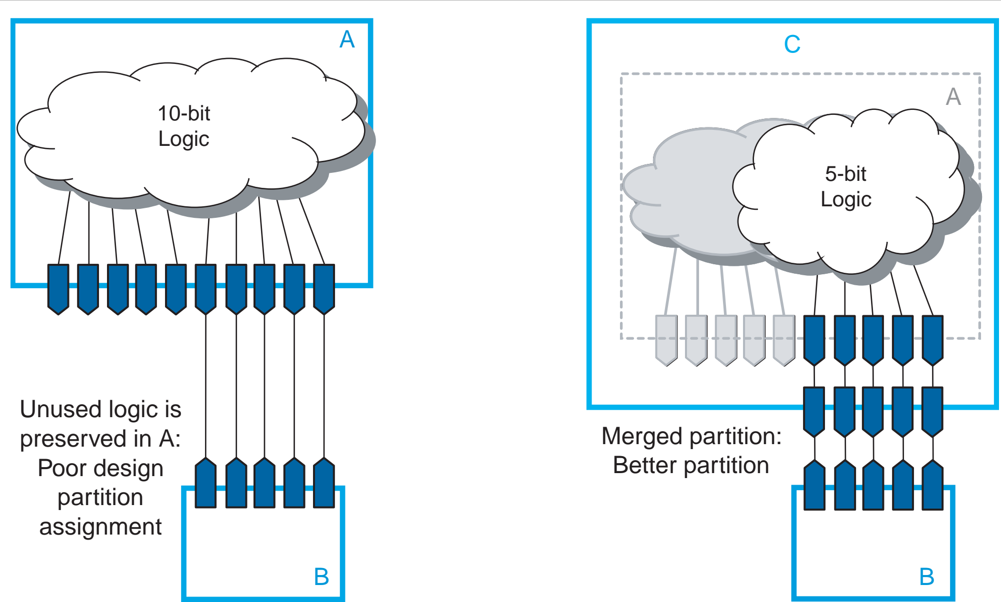
The statistics described in “Partition Statistics Report” on page 8–33 list how many input ports are fed by GND or V_{CC}. The Incremental Compilation Advisor described on page 8–47 lists the ports.

Avoid Unconnected Partition I/O

When a port is left unconnected, optimizations might be able to remove logic driving that port and improve results, similar to a constant connection. However, these optimizations are not allowed across partitions in incremental compilation, because they would create cross-partition dependence. For best results, connect ports to an appropriate node or remove them from the partition. If you know a port will not be used, consider defining a wrapper module with a port interface that reflects this fact.

For example, the left side of Figure 8–7 shows a design that has a 10-bit function defined in partition A, but has only 5 bits connected in partition B. In a flat design, you would expect the logic for the other unused 5 bits to be removed during synthesis. With incremental compilation, synthesis does not remove the unused logic from partition A because partition B is allowed to change independently from partition A. Therefore, you could later connect all 10 bits in partition B and use all 10 bits from partition A. In this design, if you know that you will not use the other 5 bits of partition A, you should remove the unconnected ports and replace them with ground signals inside A. You can create a new wrapper file in the design hierarchy to do this, as shown on the right side of the figure. A new partition C contains the logic from A but includes only the 5 output ports required for connection with partition B. Within this new partition C, the logic for the unused 5 bits can be removed from the design, reducing area utilization.

Figure 8–7. Avoiding Unconnected Partition I/O by Creating a Wrapper File



The statistics described in “Partition Statistics Report” on page 8–33 list how many I/Os are unconnected. The Incremental Compilation Advisor described on page 8–47 lists the unconnected ports.

Avoid Signals That Drive Multiple Partition I/O or Connect I/O Together

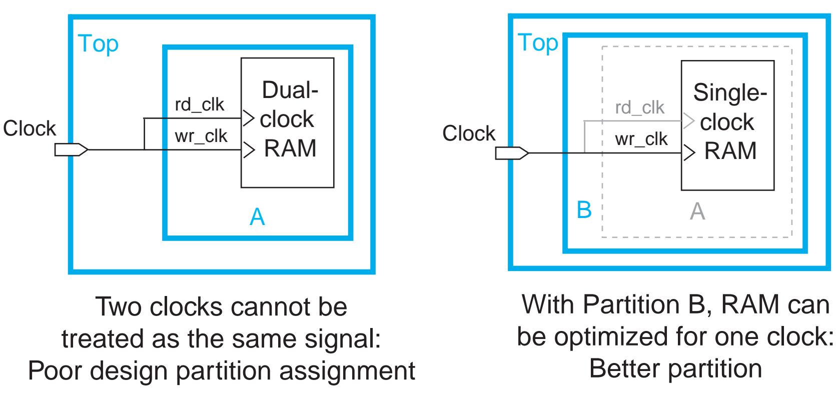
Do not use the same signal to drive multiple ports of a single partition or directly connect two ports of a partition.

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, because the software has no information about connections made in another partition (including the Top partition), the compilation cannot take advantage of the equivalence. This restriction usually results in 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 design partition.

Figure 8–8 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 partition in this case), redefine the port interface so 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. In addition, 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.

Figure 8–8. Preventing One Signal from Driving Multiple Partition Inputs



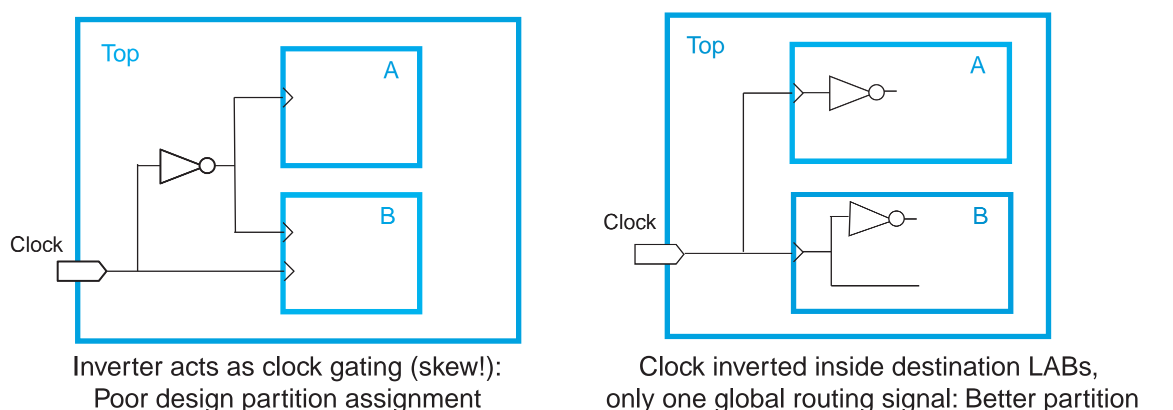
The Incremental Compilation Advisor described on "Incremental Compilation Advisor" on page 8–47 lists partition ports that have the same driving signal, and ports that are directly connected together.

Invert Clocks in Destination Partitions

For best results, clock inversion should be done in the destination logic array block (LAB), because each LAB contains clock inversion circuitry in the device architecture. In a flat compilation, the 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 to take advantage of the inversion architecture in the destination LABs.

With partition boundaries as shown on the left side of Figure 8–9, the Quartus II software uses logic to invert the signal in the partition that defines the inversion (the Top 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 figure. In this case the correct logic and routing resources can be used, and the signal is not a gated clock.

Figure 8–9. 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 Directly to I/O Register for Packing Across Partition Boundaries

Cross-partition register packing of I/O registers is allowed in certain cases where your input and output pins exist in the top-level hierarchy (and the Top partition), but the corresponding I/O registers exist in other partitions.

The following specific circumstances are required for input pin cross-partition register packing:

- 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.

The following specific circumstances are required for output register cross-partition register packing:

- The register feeds exactly one output pin.

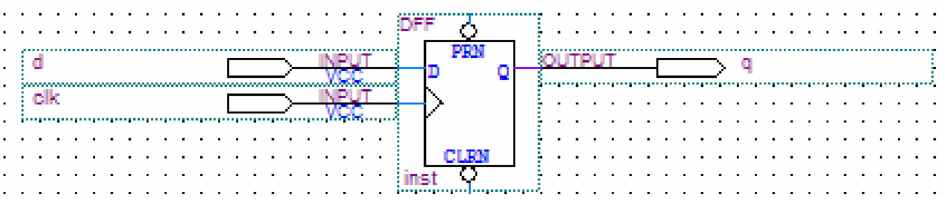
- The output pin is fed by only one signal.
- The path between the register and output pin includes only output ports of partitions that have one fan-out each.

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 8–10.

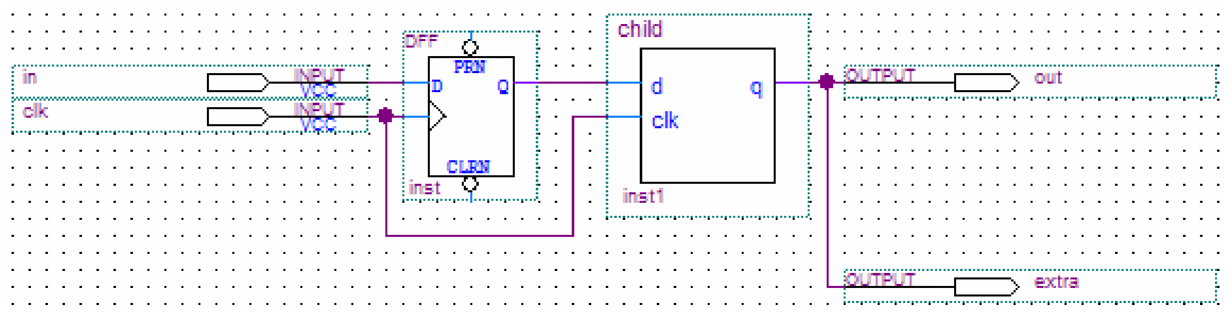
Figure 8–10. Subdesign with One Register, Designated as a Separate Partition



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 this example, the top-level design instantiates the subdesign as an output register with more than one fan-out signal, as shown in Figure 8–11.

Figure 8–11. Top-Level Design Instantiating the Subdesign in Figure 8–10 with Two Output Pins



In this case, the 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 permitted because it requires modification to the interface of the subdesign partition. To perform incremental compilation, the software must preserve the interface of design partitions.

To allow the software to pack the register in the subdesign from Figure 8–10 with the output pin `out` in Figure 8–11, restructure your HDL code so that output registers directly connects output pins by making one of the following changes:

- Place the register in the same partition as the output pin. The simplest option is to move the register from the subdesign partition into the partition containing the output pin. This guarantees that the Fitter can optimize the two nodes without violating any partition boundaries.
- Duplicate the register in your subdesign HDL as in Figure 8–12 so that each register feeds only one pin, then connect the extra output pin to the new port in the top-level design as shown in Figure 8–13. This converts the cross-partition register packing into the simplest case where each register has a single fan-out.

Figure 8–12. Modified Subdesign from Figure 8–10 with Two Output Registers and Two Output Ports

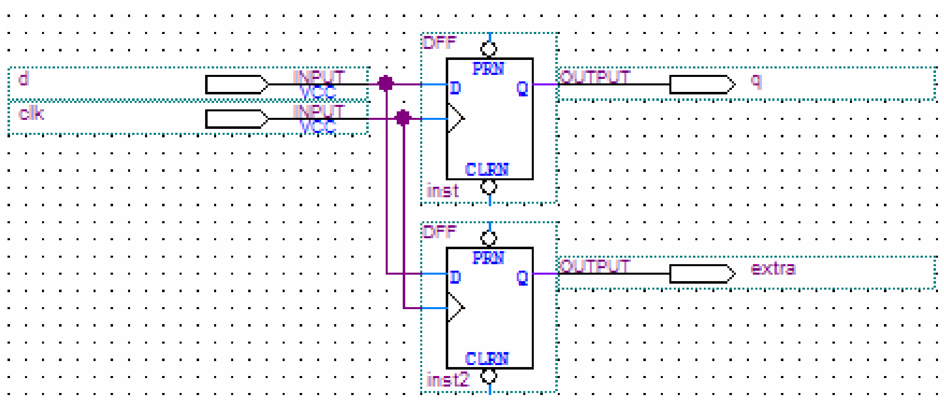
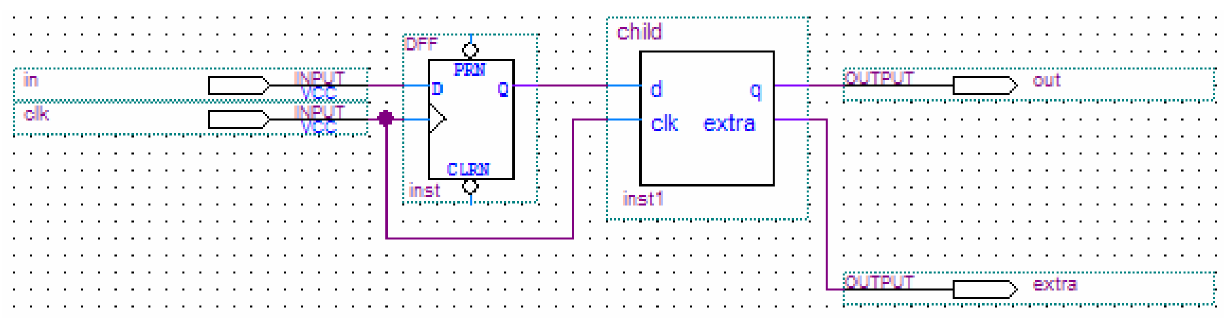


Figure 8–13. Modified Top-Level Design from Figure 8–11 Connecting Two Output Ports to Output Pins



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 8–10. The top-level design in Figure 8–14 instantiates the subdesign as an input register with the input pin inverted. The top-level design in Figure 8–15 instantiates the subdesign as an output register with the signal inverted before feeding an output pin.

Figure 8–14. Top-Level Design Instantiating the Subdesign in Figure 8–10 as an Input Register with an Inverted Input Pin

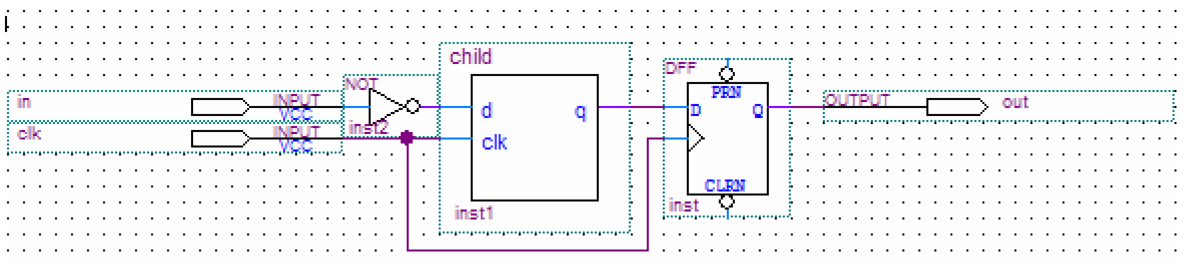
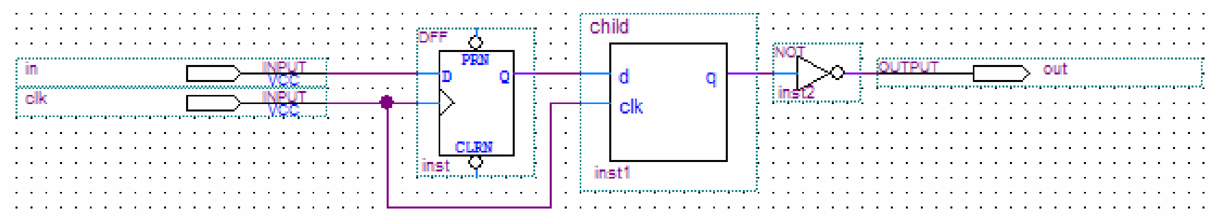


Figure 8–15. Top-Level Design Instantiating the Subdesign in Figure 8–10 as an Output Register Feeding an Inverted Output Pin



In these cases, the software does not perform register packing. If there is a **Fast Input Register** assignment on pin `in` in Figure 8–14 or a **Fast Output Register** assignment on pin `out` in Figure 8–15, the 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 permitted 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 software to pack the register in the subdesign from Figure 8–10 with the input pin `in` in Figure 8–14 or the output pin `out` in Figure 8–15, restructure your HDL code to place the register in the same partition as the inverter by making one of the following changes:

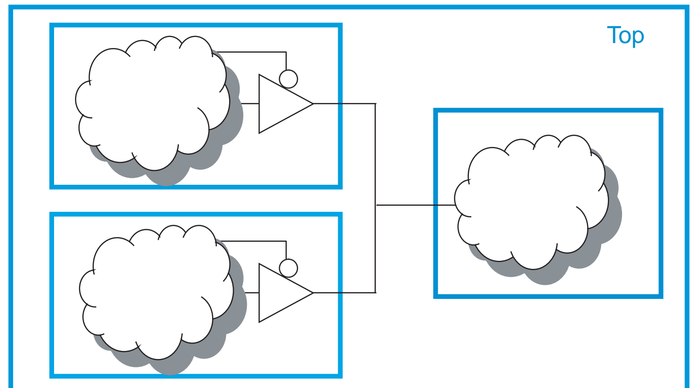
- Move the register from the subdesign partition into the top-level partition containing the pin. This ensures that the Fitter can optimize the I/O register and inverter without violating any partition boundaries.
- Move the inverter from the top-level block into the subdesign, then connect the subdesign directly to a pin in the top-level design. This allows the Fitter to optimize the inverter into the register implementation, so 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 do use internal tri-states in a flat design (with no partitions), the tri-state logic is usually converted to OR gates or multiplexing logic. But if tri-state logic occurs on a hierarchical partition boundary, the 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 8–16 shows 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 software to convert the tri-state signals into logic.

Figure 8–16. Unsupported Internal Tri-State Signals



Design results in Quartus II error message:
The software cannot synthesize this
design and maintain incremental functionality

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 software can convert the internal tri-state signals into multiplexing logic like in 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.

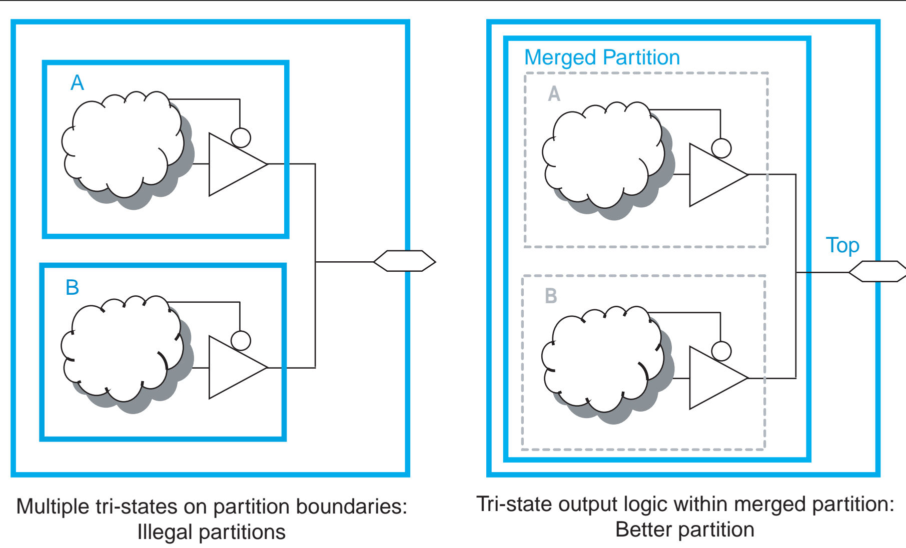
Include All Tri-State and Enable Logic in the Same Partition

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 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.

 © November 2009 Altera Corporation

Figure 8–17 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). On the left side of the figure, the tri-state output signals appear as the outputs of two separate partitions. In this case, the software cannot implement the specified logic and maintain incremental functionality. On the right side, another level of hierarchy C has been created to group the logic from blocks A and B. 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 8–17. Including All Tri-State Output Logic in the Same Partition

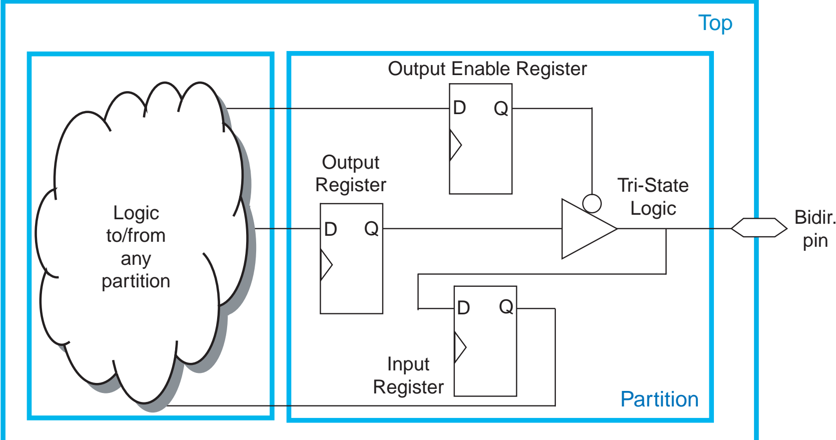


Include Bidirectional I/O Registers in the Same Partition

For a bidirectional partition port that feeds a bidirectional I/O pin at the top level, all the logic that forms the bidirectional I/O cell must reside in the same partition. This guideline applies only to the Stratix II, Stratix, Cyclone® II, Cyclone families, but not newer devices. In addition, as discussed in the previous two recommendations, the I/O logic must feed the I/O pin without any intervening logic.

In Figure 8–18, for the software to implement all three registers in the I/O element along with the tri-state logic, all the I/O logic must be defined inside the same partition. 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.

Figure 8–18. Including All Bidirectional I/O Registers in the Same Partition



Bidirectional logic is within one partition, and I/O logic directly feeds I/O pin

Summary of Guidelines Related to Logic Optimization Across Partitions

Follow the guidelines presented in this section to ensure that your design does not require any logic optimization across partitions:

- Keep logic in the same partition for optimization and merging
- Keep constants in the same partition as logic
- Avoid unconnected partition I/O
- 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 strict rules to implement an incremental compilation flow, but can improve the quality of results. When creating source design code, keep these guidelines in mind 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 to improve your results.

Creating Design Partitions: Consider Additional Design Suggestions

This section includes additional design practices that may improve success in incremental compilation flows, if they are applicable to your design:

- "Balance Logic Resources" on page 8–23
- "Balance Global Routing Signals and Clock Networks if Required" on page 8–24
- "Assign Virtual Pins in Team-Based Flows" on page 8–25
- "Perform Timing Budgeting if Required" on page 8–26
- "Consider a Cascaded Reset Structure" on page 8–26
- "Drive Clocks Directly in Team-Based Flows" on page 8–27
- "Recreate PLLs for Lower-Level Partitions if Required in Team_Based Flows" on page 8–28

Balance Logic Resources

If you are using incremental compilation, the software synthesizes each partition separately with no data about the resources used in other partitions. This means that device resources could be overused in the individual partitions during synthesis, thus, the design may not fit in the target device when the partitions are merged.

In a design flow in which designers optimize their lower-level designs and export them to a top-level design, the software places and routes each partition separately. In some cases, partitions can use conflicting resources when combined at the top level.

For example, in the standard synthesis flow, the Quartus II Compiler can perform automated resource balancing for DSP blocks or RAM blocks and convert some of the logic into regular logic cells to prevent overuse. Without data about DSP and RAM blocks used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource.

To avoid these effects, you may have to perform manual resource balancing across partitions. This is more applicable with imported partitions, because compilation usually handles resource balancing without any user intervention if all resource information is in one Quartus II project.

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.

You can also assign a number of LAB, DSP or RAM resources for each partition. Use the following logic options to specify the maximum number of logic blocks that the software can use in the specified partition: **Maximum Number of LABs**, **Maximum DSP Block Usage**, **Maximum Number of M4K/M9K Memory Blocks**, or **Maximum Number of M-RAM/M144K Memory Blocks**. You can set these options globally for all partitions. To set an option for all partitions, on the Assignments menu, click **Settings**. Under **Category**, select **Analysis & Synthesis Settings**. Click **More Settings,** and in the **Existing option settings** list, select the appropriate option. You can also set the option for a specific partition with the Assignment Editor. Select the assignment

name, apply it to the root entity of a partition, and set an integer as the value. The partition-specific assignment overrides the global assignment, if any. However, each partition that does not have a partition-specific assignment can use the number of LAB, DSP, or RAM blocks set by the global assignment. Be aware that this behavior can lead to over-allocation of logic blocks, eventually resulting in a no-fit error.

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 more 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*.

It is often helpful to create a LogicLock region to isolate the placement of each partition, especially when partitions are imported, to minimize the chance that the logic in more than one partition uses the same logic resource. However, there are situations in which partition placement may still cause conflicts at the top level. For example, you can design a partition one way in a lower-level design (such as using an M-RAM memory block) and then instantiate it in two different ways in the top level (such as one using an M-RAM block and another using an M4K block). In this case, you can export a post-fit netlist with no placement information from the lower-level design and allow the software to refit the logic at the top level.

Balance Global Routing Signals and Clock Networks if Required

If your design is very complex and has many clocks, you may have to allocate global routing resources between the different design partitions. In most cases, you do not have to allocate routing because the software finds the best solution for the global signals.

Global routing signals can cause conflicts when multiple projects are imported into a top-level design. The Quartus II software automatically promotes high fan-out signals to use global routing resources available in the device. Lower-level partitions can use the same global routing resources, thus causing conflicts at the top level. In addition, LAB placement depends on whether the inputs to the logic cells within the LAB are using a global clock signal. Therefore, problems can occur if a design does not use a global signal in the lower-level design, but does use a global signal in the top-level design.

To avoid these problems, the project lead can first determine which partitions use which type of global routing signals. Each designer of a lower-level partition can then assign the appropriate type of global signals manually and prevent other signals from using global routing resources, or set a maximum number of clocks 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 that cover only part of the device. Alternatively, designers of lower-level partitions can specify the number of clocks allowed in the project using the maximum clocks allowed options. On the Assignments menu, click **Settings**. Under **Category**,

select **Fitter Settings**. Click **More Settings**, and in the **Existing option settings** list, select the appropriate option. Choose **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 given type in the project.

You can 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. If 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 chip. You can use the Fitter's results to make assignments when optimizing the lower-level partitions in separate Quartus II projects.

If you require more control when planning a design with imported 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. Use the **Global Clock CLKCTRL Location** logic option. 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. Set the assignment value to the name of the clock control block: `CLKCTRL_G`<*global network number*> to choose one of the global routing networks or `CLKCTRL_R`<*regional network number*> to choose one of the dedicated regional routing networks 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. On the Assignments menu, click **Settings**. On the **Fitter Settings** page, click **More Settings** and change the settings to **Off**.

If you are using design partition scripts, the software can automatically write the commands to pass global constraints and turn off the automatic options. For more information, refer to "Project Management in Team-Based Designs" on page 8–4.

Alternatively, to avoid problems when importing, direct the Fitter to discard the placement and routing of the imported netlist by setting the Fitter preservation level property of the partition to Netlist Only. With this option, the Fitter reassigns all the global signals for this particular partition when compiling the top-level design.

Assign Virtual Pins in Team-Based Flows

Virtual pins map lower-level design I/Os to internal cells. Use them when the number of I/Os on a lower-level design exceeds the device I/O count, and to increase the timing accuracy of cross-partition paths.

Make a virtual pin assignment in the Assignment Editor for lower-level design I/Os that will become internal nodes in the top level. Leave 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. You can 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 the **Generate Bottom-Up Design Partition Scripts** command. The scripts place the virtual pins to correspond with other partitions' placement from the top-level design. For more information, refer to "Project Management in Team-Based Designs" on page 8–4. 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 lower-level partitions independently and import them to the top level, or compile with empty partitions, any unregistered paths that cross between partitions are not optimized as an entire path. In these cases, the Compiler 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 the timing requirements. You can reduce this effect by ensuring that input and output ports of the partitions are registered whenever possible.

To ensure that the Compiler correctly optimizes the input and output logic in each partition, you may 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 Compiler optimizes the paths appropriately.

When performing manual timing budgeting in a lower-level partition for I/O ports that become internal partition connections in a top-level design, you can assign location and/or timing constraints to the virtual pin that represents each connection to further improve the quality of the timing budget. Refer to the previous section for a description of virtual pins.

If you are using the design partition scripts, the software can write I/O timing budget constraints automatically for virtual pins. For more information, refer to "Project Management in Team-Based Designs" on page 8–4.

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. For incremental flows, the global reset signal also creates a timing dependency between the Top partition and lower-level partitions.

For incremental compilation, minimizing the impact of global structures is helpful. To isolate each partition, consider adding reset synchronizers. By using cascaded reset structures, the design 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 may help recovery and removal times in some cases.

This suggestion 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, you probably do not have to follow this recommendation. It 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, it adds latency to the reset path, so be sure that this is acceptable in your design. In addition, parts of the design may come out of reset in different clock cycles. You can balance the latency or add hand-shaking logic between partitions, if necessary, to accommodate these differences.

Figure 8–19 shows a cascaded reset structure. The signal is first synchronized as it comes on the chip, following good synchronous design practices. This logic means 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, each partition's reset paths are independent. This 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 partition's reset signal 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 8–19. Cascaded Reset Structure

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

Drive Clocks Directly in Team-Based Flows

When partitions are imported from a 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 cause trouble especially in team-based flows because the lower-level partitions have no information about any gating that takes place at the top level 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 8–16.

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

Recreate PLLs for Lower-Level Partitions if Required in Team_Based Flows

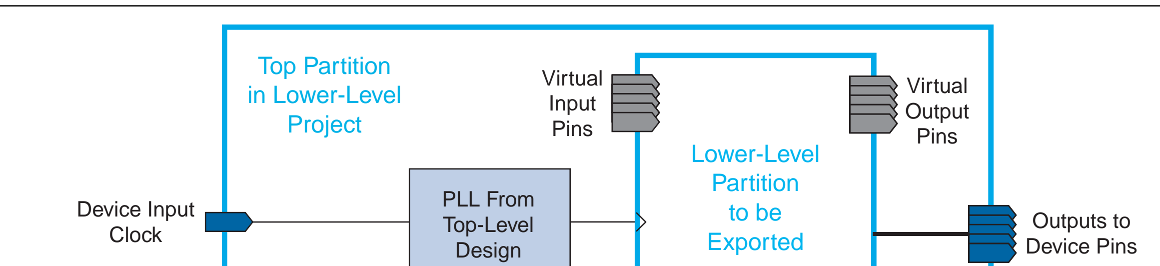
If you use a PLL in your top-level design and connect it to lower-level partitions, the lower-level partitions do not have information about the multiplication, phase shift, or compensation delays for the PLL. To accommodate the PLL timing, you can make appropriate timing assignments in your lower-level Quartus II project 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 lower-level design file to ensure that you have the correct PLL parameters and clock delays for complete, accurate timing analysis.

One methodology for team-based design is for the lead designer to create a 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 lower-level designs.

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), you can include a copy of the top-level PLL in the lower-level project as shown in Figure 8–20.

In either case, the project for the lower-level design should include a a design partition to contain the lower-level design logic that will be exported to the top level. When the design is complete, you can export just the lower-level partition, without exporting any auxiliary PLL components to the top-level design. When you use the feature to export a partition within a project, the 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 8–20. Recreating a Top-Level PLL in a Lower-Level Partition



Checking Partition Quality

This section provides an overview of tools you can use as you make 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 check that your design follows the recommendations for creating design partitions that are presented in this document.

On the Tools menu, point to **Advisors** and click **Incremental Compilation Advisor**. Recommendations are split into General Recommendations, which apply to all compilation flows, and Bottom-Up Design Recommendations, which apply when partitions are developed independently and are exported to the top level design project so that floorplan recommendations are important to isolate the partition. Each recommendation provides an explanation, describes the effect of the recommendation, and provides the action required to make the suggested change.

To check whether the design follows the recommendations, go to the **Timing Independent Recommendations** page or the **Timing Dependent Recommendations** page (for the TimeQuest Timing Analyzer or the Classic Timing Analyzer), and click **Check Recommendations**. For large designs, these operations can take a few minutes.

After you check the design, a symbol appears next to each recommendation that indicates whether or not your design follows that particular recommendation. Refer to the Legend on the **How to use the Incremental Compilation Advisor** page in the Incremental Compilation Advisor for more information.

In some items, there is a link to the appropriate Quartus II settings page where you can make a suggested change to assignments or settings. For many items, if your design does not follow the recommendation, the Check Recommendations operation creates a table that lists any nodes or paths in the design that could be improved.

For example, if not all the partition I/O ports follow the Register All Ports recommendation, the Incremental Compilation Advisor displays a list of unregistered ports with the partition name and the source and destination nodes for the port. When the Incremental Compilation Advisor provides a list of nodes, you can right-click on a node and click **Locate** to cross-probe to other Quartus II features such as the RTL Viewer, Chip Planner, or the design source code in the text editor.

Opening a new TimeQuest report resets the Incremental Compilation Advisor results, so you must rerun the Check Recommendations process.

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 document. 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, first compile the design, or perform Analysis and Synthesis. On the Tools menu, click **Design Partition Planner**. The design appears as a single top-level design block, containing its lower-level instances as boxes.

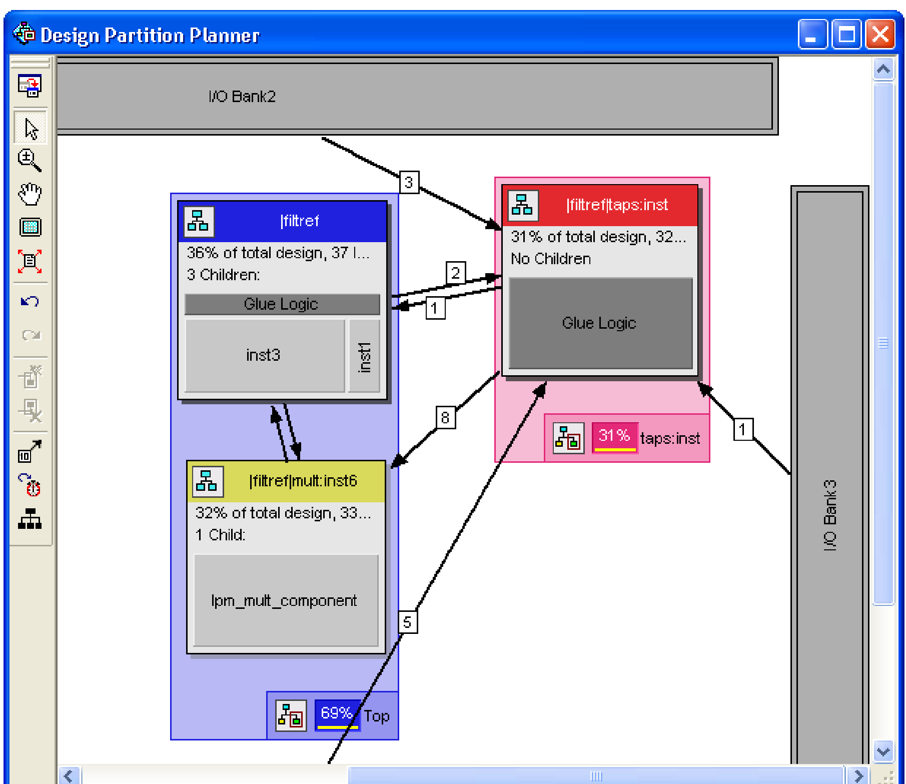
To show connectivity between blocks, extract instances from the top-level design block. Click on a design block and drag it into the surrounding white space, or right-click an entity and click **Extract from Parent** on the Shortcut menu.

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 on that design block and 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. Right-click on the design block you want to partition (such as the top-level design hierarchy), and choose **Auto-Partition**. You can then analyze and adjust the partition assignments as required.

Figure 8–21 shows the Design Partition Planner after making a design partition assignment to one instance (in the pale red shaded box), 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 number of connections between each partition and information about the size of each design instance.

Figure 8–21. Design Partition Planner



To switch between connectivity display mode and hierarchical display mode, click **Hierarchy Display** on the View menu. Alternately, to switch temporarily to a view-only hierarchy display, click and hold the hierarchy icon in the top-left corner of any entity.

To control the way the connection bundles are displayed, right-click in the white space and choose **Bundle Configuration**. For example, you can remove the connection lines between partitions and I/O banks by turning off **Display connections to I/O banks**. You can also use the settings on the **Connection Counting** tab to adjust how the connections are counted in the bundles.

To optimize design performance, it is desirable to confine failing paths within individual design partitions, so that there are no failing paths passing between partitions, as discussed in earlier sections. To view the critical timing paths from a timing analyzer report, perform the following steps:

1. Open the TimeQuest Timing Analyzer and perform a timing analysis on the design.
2. In the Design Partition Planner, click **Show Timing Data** on the View menu.

In the top-level entity, child entities containing failing paths are marked by a small red dot in the upper right corner of the entity box.

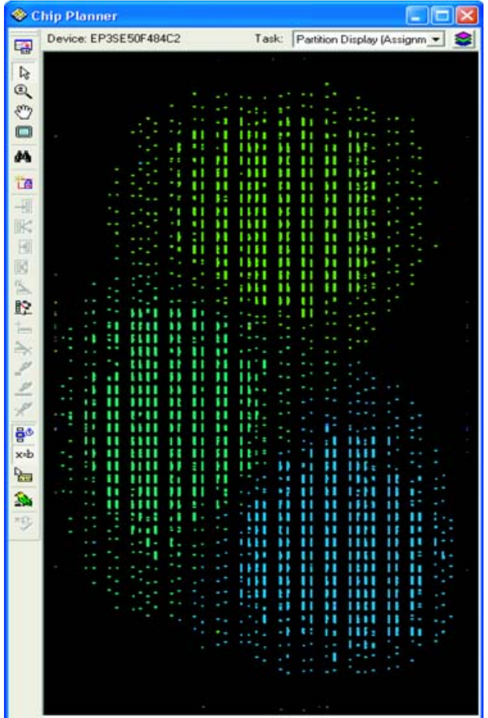
For more information about how to use the Design Partition Planner to analyze your design and create partitions, refer to "Using the Design Partition Planner" in the Quartus II Help.

Floorplan Partition Coloring

After making a set of partition assignments, it can be useful to view how the partitions are placed in the device. The Chip Planner can display nodes for each partition in a different color.

After compilation, in the Chip Planner **Task** list, select **Partition Display (Assignment)**, as shown in Figure 8–22. In this figure, you can see that the three different-colored partitions are grouped in three fairly independent areas of the device.

Figure 8–22. Partition Display in the Chip Planner Showing Three Partitions with Different Color Shades



Viewing Design Partition Planner and Floorplan Side-by-Side

You can view the Design Partition Planner together with the Chip Planner's Partition Planner, to analyze natural placement groupings in the floorplan view. This information can help you decide whether the design blocks should be grouped together in one partition, or whether they will make good partitions for the next compilation. It can also help determine whether the logic can easily be constrained by a LogicLock region to create a design floorplan. If logic naturally groups together when compiled without placement constraints, you can probably assign a reasonably sized LogicLock region to constrain the placement for future 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 last compilation.

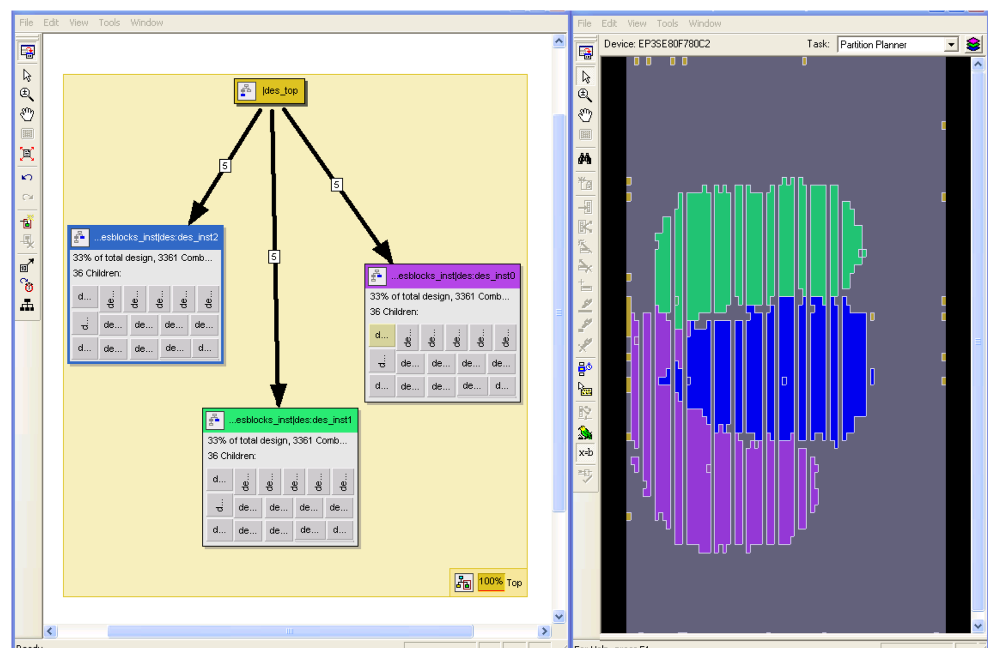
Open the Design Partition Planner, then open the Chip Planner and select the **Partition Planner** task in the **Task** list. This task selection displays the physical locations of design entities with the same colors as the Design Partition Planner display. For ease of viewing, drag and size the Chip Planner and Design partition Planner windows so they are side-by-side.

In the Design Partition Planner, you can extract instances of interest from their parents with the drag and drop method or 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 to be physically dispersed over a noncontiguous area of the device

after compilation. You can use the Design Partition Planner as described in "Design Partition Planner" on page 8–29 to analyze the design connections. For child instances that are unsuitable to be set as separate design partitions or placed in LogicLock regions, you can return those instances to their parent with the drag and drop method or the **Collapse to Parent** command.

Figure 8–23 shows a design displayed in both viewers, with different colors for the top-level design and the three major design instances.

Figure 8–23. Top-Level Design and Three Major Instances Shown in Both Viewers



Partition Statistics Report

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

The **Partition Statistics** page under the **Partition Merge** folder 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 it contains, 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 V_{CC} or GND. You can use this information to assess whether you have followed the guidelines for partition boundaries.

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. On the Assignments menu, click **Design Partitions Window**. Right-click on a partition and click **Properties** to open the dialog box. Click **Show All Partitions** to view all the partitions in the same report. The Design Partition Properties report also shows statistics for the Internal Congestion: Total Connections and Registered Connections. This represents how many signals are connected within the partition. It then lists the inter-partition connections for each partition, which helps you see how partitions are connected to each other.

Report Partition Timing in the TimeQuest Timing Analyzer

The TimeQuest Timing Analyzer includes a diagnostic report called Report Partitions, and the `report_partitions` SDC command. The resulting Partition Timing Overview lists the design partitions and provides the number of failing paths and the worst case timing slack within that partition. The function also creates a Partition Timing Details table that lists the number of failing paths and worst-case slack from each partition to the others.

You can use this report to analyze where the critical timing paths in the design are with respect to design partitions. If a certain partition contains many failing paths, or failing inter-partition paths, you may be able to change your partitioning scheme and improve your timing performance.

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

Ensure Partition Assignments Do Not Impact the Quality of Results

There is often a trade-off between compilation time and quality of results when you vary the number of partitions in a project. You can ensure that you limit any negative effect on the quality of results by following an iterative methodology during the partitioning process. In any incremental compilation flow in which 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 assignments.
2. To perform a placement and timing analysis estimate, on the Processing menu, point to **Start** and click **Start Early Timing Estimate**.

   You must perform Analysis and Synthesis and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.
3. Record the quality of results from the Compilation Report (f_{MAX}, area, and any other relevant results).
4. Create design partitions following the guidelines described in this chapter.
5. Perform another Early Timing Estimate or a full compilation.

6. 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 4 through step 6 to change your partition assignments and use a different partitioning scheme.
7. Even if the quality of results is acceptable, you can repeat step 4 through step 6 by further dividing a large partition into several smaller partitions. Doing so improves compilation time in future incremental compilations. You can repeat this step 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).

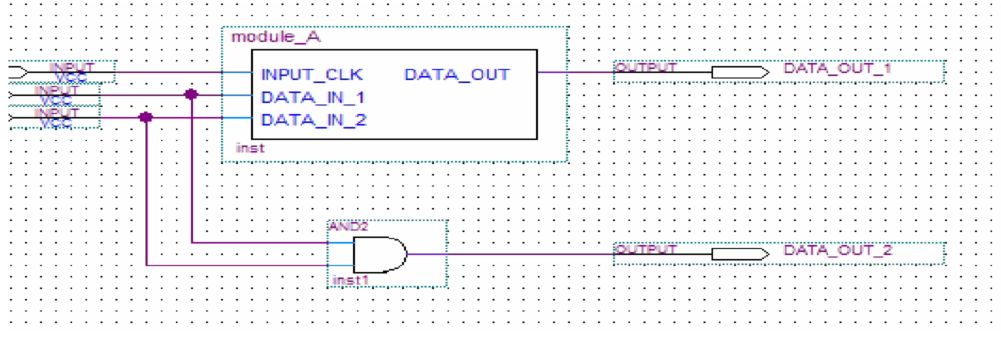
Importing SDC Constraints from Lower-Level Partitions in Team-Based Designs

In a team-based design environment, the project lead must transfer the top-level project information and constraints to the lower-level projects, so that lower-level designers each have a consistent view of the constraints that apply to the entire design. You can copy the top-level project for each designer, or use the design partition scripts to automate the process of sending assignments and constraints. If the lower-level 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. You can use the **Import** command to import assignments from lower-level partition projects into the top-level project; however, the automatic import does not include SDC format constraints for the TimeQuest Timing Analyzer.

Passing additional timing constraints from a lower-level project to the top-level project must be managed carefully. This section provides recommendations for managing the timing constraints in a team-based incremental compilation flow.

To ensure that there are no conflicts between the project lead's top-level constraints and those added by the lower-level designer, use two Synopsys Design Constraint Files (**.sdc**) for each lower-level project: an **.sdc** created by the project lead that includes project-wide constraints and an **.sdc** created by the lower-level partition designer that includes partition-specific constraints. This section uses the example design shown in Figure 8–24 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 another designer in a separate Quartus II project.

Figure 8–24. Example Design to Illustrate SDC Constraints



In this top-level project, 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:

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

Creating an .sdc with Project-Wide Constraints

The **.sdc** with project-wide constraints for the lower-level project should contain all constraints that are not completely localized to the lower-level partition. The **.sdc** should be maintained by the top-level 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 lower-level projects. This can be challenging when the design is in flux and hierarchies change. The project lead can use the Generate Bottom-Up Design Partition Scripts tool to automatically generate some of these constraints, as described in the previous section.

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

The **.sdc** should include clock creation and clock constraints for any clock used by more than one lower-level project. This is particularly important when dealing 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 which are consistently used for source synchronous interfaces
- Clock uncertainties

In addition, 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 on a lower-level project’s ports, because these represent delays external to a given partition. If a lower-level designer wants to set these constraints within the lower-level project, the team must ensure that the I/O port names are identical in the two projects so the assignments can be imported successfully without any 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 single lower-level project.

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

The device input `top_level_clk` in Figure 8–24 drives the `input_clk` port of `module_A`. To make sure the clock constraint is passed correctly to the lower-level project, the project lead creates an **.sdc** with project-wide constraints for `module_A` that contains the following command:

```
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 lower-level project.

Creating an .sdc with Partition-Specific Constraints

The **.sdc** with partition-specific constraints should contain all constraints that affect only the lower-level partition. For example, a `set_false_path` or `set_multicycle_path` constraint for a path entirely within the lower-level 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 lower-level projects.

The partition-specific **.sdc** should be maintained by the individual partition designer; it is their responsibility to add any constraints required to properly compile and analyze their partition.

The partition-specific **.sdc** is used in the lower-level 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 import steps:

- Create a hierarchy variable for this partition (such as `module_A_hierarchy`) and set it to an empty string because the partition is the top-level instance in the separate 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 project does not use this empty hierarchy path: `if {![info exists 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 '`*`').
- Be careful with 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's interface depends on the top-level design. If you want to set I/O constraints within the lower-level project, the team must ensure that the I/O port names are identical in the two projects so the assignments can be imported successfully without any changes.

- Be careful 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, importing them unexpectedly into the top-level design could cause problems.

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

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

The lower-level 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 they must specify a false path between the register called `reg_in_1` and all destinations in this design block with the wildcard character *. 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:

```
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 lower-level designers complete their designs, they export the results to the top-level project lead. The project lead receives the exported **.qxp** 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 lower-level projects, the top-level **.sdc** should define the hierarchy variables specified in the lower-level **.sdc** files. List the variable for each lower-level partition and set it to the hierarchy path, up to and including the instantiation of the lower-level partition in the top-level project, 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 project lead's 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 project, 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 projects is managed effectively.

Introduction to Design Floorplans

A floorplan represents the layout of the physical resources on the device. The expressions "creating a design floorplan" and "floorplanning" describe the process of mapping the logical design hierarchy onto physical regions in the device floorplan.

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

It is a common misconception that logic from a design partition is always grouped together on the device when you use incremental compilation. This is not true. Logic from a partition can be placed anywhere in the device if it is not constrained to a LogicLock region. A logical design partition does not refer to any physical area of the device and does not directly control *where* instances are placed on the device.

If you want to control the placement of the 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. Creating a design floorplan by assigning design partitions to LogicLock regions is recommended to improve the quality of results and avoid placement conflicts in many situations for incremental compilation. For more information, refer to "Why Create a Floorplan?" on page 8–39.

Another misconception is that LogicLock assignments are used to preserve placement results for incremental compilation. This is also not true. LogicLock regions only *constrain* logic to a physical region of 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 the previous compilation.

Why Create a Floorplan?

Floorplan location planning can be important for a design that uses full incremental compilation, for the following two reasons:

- To avoid resource conflicts between partitions, predominantly when importing partitions from another Quartus II project
- To ensure a good quality of results when recompiling individual partitions in a single Quartus II project

Creating a design floorplan is required if you want to preserve placement for lower-level partitions that will be exported into another project, to avoid resource conflicts between partitions.

Location assignments for each partition ensure that there are no placement conflicts between different partitions. If there are no LogicLock region assignments, or if LogicLock regions are set to auto-size or floating, no device resources are specifically allocated for the logic associated with the region. If you do not clearly define this resource budget, logic placement can conflict when you import the partitions to a top-level project.

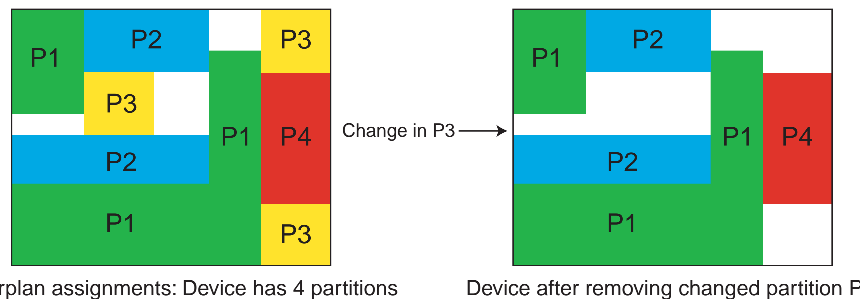
Creating a floorplan is also highly recommended for timing-critical partitions to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions compiled in one Quartus II project. The logic for partitions that are not timing-critical (such as simple top-level glue logic) 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 8–25 illustrates the problems associated with refitting designs that do not have floorplan location assignments. It shows the initial placement of a four-partition design (P1-P4) without any floorplan location assignments. The second part of the figure shows the device if a change occurs to P3. After removing the logic for the changed partition, the Fitter must replace and reroute the new logic for P3 in the scattered white space shown in Figure 8–25. 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.

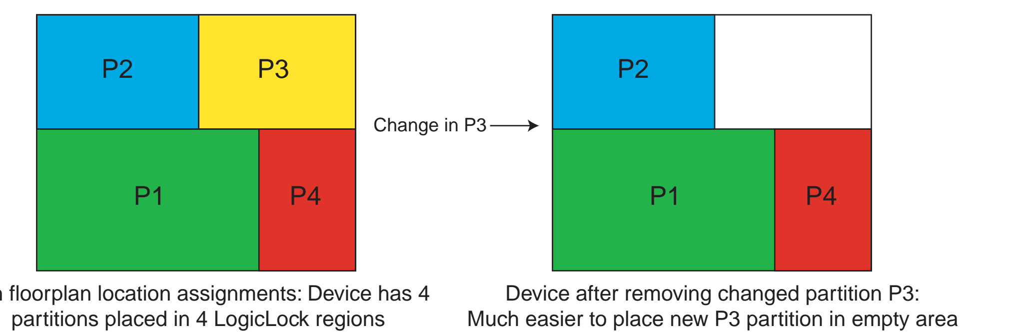
Figure 8–25. Representation of Device Floorplan without Location Assignments



The Fitter must work harder due to 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. In addition, 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 8–26 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 8–26. Representation of Device Floorplan with Location Assignments



Altera recommends that you create a LogicLock floorplan assignment for any timing-critical blocks 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 design partition follows the partitioning guidelines. You can make the floorplan assignments at different stages of the design flow, early or late in the flow. These guidelines help ensure better results when you start 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 team-based design to give each designer a portion of the device resources. Doing so allows each designer to create the logic for their design partition without conflicting with other logic. Each design partition can be implemented independently and integrated later in the top-level project.

You can use an early floorplan in a standard incremental compilation flow as well to roughly divide up the design partitions into LogicLock regions while iterating through the design cycle.

When you have your complete design compiled, or after you have integrated the first version of all design partitions in a team-based flow, 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. When the design is complete, you can take advantage of the Quartus II analysis features to check the floorplan quality. To tune 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.

Creating a 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

It is a common misconception that creating a floorplan enhances timing performance, as compared to a flat compilation with no location assignments. The Quartus II 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 8–39. However, bad placement assignments can often hurt performance results, as compared to a flat compilation, because the assignments limit the options for the Fitter. Investing some 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 may have to iterate through these steps several times to find the best combination of design partitions and LogicLock regions that meet the design's resource and timing goals.

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

To create a full floorplan: Create a LogicLock region for each partition (including the top-level) to assign all logic to a place in the device.

To create a partial floorplan: Create a LogicLock region for any critical or often-changing partitions.

Before compiling the design with new LogicLock assignments, ensure the affected partitions' Netlist Type is set so that the Fitter does not reuse previous placement results.

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

The software is flexible and does allow 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 assigned 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. In addition, 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 any conflicts with other logic which is not locked into any 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.

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 Fitter-chosen regions as a guideline. Start with the default Auto size and Floating origin location. 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, in a late floorplan, you can specify the size based on the synthesis results and use Fitter-chosen locations. Right-click on a region in the **LogicLock Regions** dialog box, and choose **Set to Estimated Size**. 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's size and origin location. Click on each LogicLock region in the LogicLock Regions Window while holding the Ctrl key to select all regions (including the top-level region). Right-click on the last selected LogicLock region and click **Set Size and Origin to Previous Fitter Results**.

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 chip. 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's 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, you can use the guidelines in this section to set the size and origin, even though there is no initial Fitter placement for a basis.

The easiest way to move and resize regions is to drag the region location and borders in the Chip Planner. Ensure 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 the 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 future 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. As a very rough suggestion, try to have each region 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 an overlapping portion of the chip, each may independently claim some common resources in this region. This leads to resource conflicts when importing results into a final 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. It is best to 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).

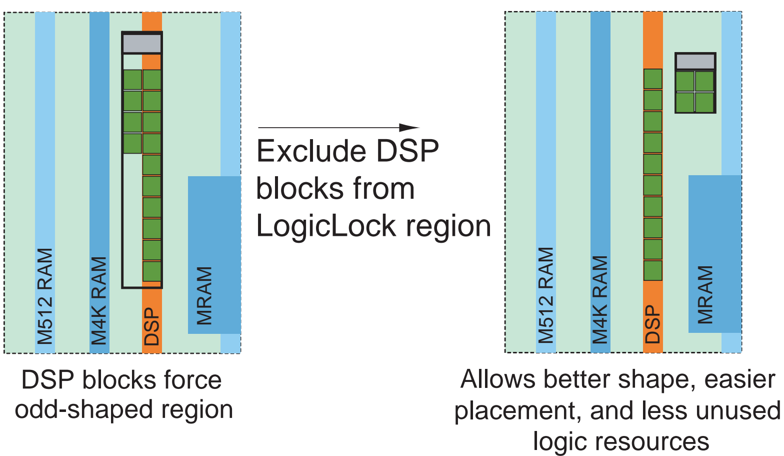
LogicLock Resource Exclusions

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 elements of certain types 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 region's **Reserved** property 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 8–27 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. The DSP blocks are placed outside the region.

Figure 8–27. LogicLock Resource Exclusion Example



To view any resource exceptions, right-click in the LogicLock Regions Window and click **Properties**. In the **LogicLock Region Properties** dialog box, highlight the design element (module/entity) in the **Members** box and click **Edit**. To set up a resource exception, click the browse button under **Excluded element types**, 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 Non-Rectangular Regions

To constrain placement to non-rectangular areas of the device, you can connect multiple rectangular regions together using the **Merge** command. To merge regions, select one or more rectangles that should be part of the same region (using the Ctrl key), right-click and choose **LogicLock Region Properties,** and then click **Merge**.

For devices that do not support the **Merge** command (Arria™ GX, Cyclone, Cyclone II, HardCopy, HardCopy 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. 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 a region's **Reserved** option to **On** prevents the Fitter from placing nodes that are not assigned to the region inside the boundary of the region. Setting a region's **Reserved** option to **Limited** prevents the Fitter from placing nodes that are assigned to the immediate parent LogicLock region's hierarchy inside the boundary of the region. Any other logic can be placed inside the region. To create non-rectangular regions for a specific entity, you can place child LogicLock regions inside a parent region and set the **Reserved** setting of the child regions to **Limited**. The child region prevents the parent region hierarchy from using that area of the device floorplan, but leaves it open for the rest of the design. You can assign other LogicLock regions to cover that area of the device if required.

For information and examples of creating non-rectangular regions with the **Reserved** property, refer to Examples of *Creating Non-Rectangular LogicLock Regions with the Limited Reserved Setting* in the Quartus II Help.

For information about creating non-rectangular regions with the **Merge** command, refer to Create LogicLock Region/Merge LogicLock Region Commands in the 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. Take advantage of 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 document. For more information, refer to "Incremental Compilation Advisor" on page 8–29.

LogicLock Region Resource Estimates

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

Right-click in the LogicLock Regions Window, choose **Properties**, and select the **Size & Origin** tab. Specify a size and an origin to see the **Available resources** estimate in the dialog box.

LogicLock Region Properties Statistics Report

The LogicLock Region Properties Statistics are similar to the Design Partition Properties described in "Partition Statistics Report" on page 8–33, but include resource usage details after compilation.

The statistics report the number of resources used and the total resources covered by the region. The statistics 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).

Right-click in the LogicLock Regions Window, choose **Properties** and select the **Statistics** tab. Click **Show All Regions** to see all regions displayed in the same report.

Critical Path Settings for Chip Planner

The **Critical Path Settings** dialog box allows you to display the most critical paths from the Timing Analyzer report in the Chip Planner floorplan view. You can specify a threshold for which paths to highlight in the Chip Planner. Use this information to identify inter-region critical paths and improve your partition or floorplan assignments.

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

In the TimeQuest user interface, you can locate a specific path in the Chip Planner to view its placement. Perform a report timing operation (for example, report timing for all paths with less than 0 ns slack). Right-click in the detailed path report (**Data Path** tab) for a specific path and choose **Locate Path**. Click **OK** to choose the Chip Planner.

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 visualize how many connections there are between each LogicLock region, to improve floorplan assignments, or to change partition assignments if required.

With the Chip Planner open, on the View menu, click **Generate Inter-region Bundles**.

Routing Utilization

The Chip Planner includes a mode 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, click the Layer Settings icon next to the **Task** list. Change the **Background Color Map** to **Routing Utilization** (the default is Block Utilization).

The darker-colored LAB blocks indicate higher routing congestion. Move your mouse pointer over a LAB to see a tool tip that reports the logic and routing utilization information.

Ensure Floorplan Assignments Do Not 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 the results before creating your floorplan location assignments to the results after doing so. Consider using another scheme if any of the following guidelines are not met:

- You should see no degradation in f_{MAX} after the design is partitioned and floorplan location assignments are created. In many cases, a slight increase in f_{MAX} is possible
- 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 by 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. Doing so 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 the Entire Design

Use this flow for incremental compilation designs in which you would like to assign a floorplan location for each design block that is assigned as a separate partition. This is the standard floorplan procedure described in the *Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*. 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 LogicLock region for each design partition, perform the following steps:

1. On the Assignments menu, click **Design Partitions Window** and 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 automatically considered a partition).
3. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to place auto-sized, floating-location LogicLock regions.

   You must perform Analysis and Synthesis, and Partition Merge before performing an Early Timing Estimate.

   To run a full compilation instead of the Early Timing Estimate, on the Processing menu, click **Start Compilation**.
4. On the Assignments menu, click **LogicLock Regions Window**, and click on each LogicLock region while holding the Ctrl key to select all regions (including the top-level region).
5. Right-click on the last selected LogicLock region, and click **Set Size and Origin to Previous Fitter Results**.
6. If required, modify the size and location with the LogicLock Regions Window or the Chip Planner. For example, make the regions bigger to fill up the device and allow for future logic changes.
7. On the Processing menu, point to **Start** and click **Start Early Timing Estimate** to estimate the timing performance of your design with these LogicLock regions.
8. Repeat step 6 and 7 until you are satisfied with the quality of results for your design floorplan. On the Processing menu, click **Start Compilation** to run a full compilation.

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

Use this approach when you have several lower-level subdesigns that will be implemented separately by different designers. The subdesign designers want to optimize their designs independently and pass the results on to you, the project lead.

As the project lead in this scenario, perform the following steps to prepare the 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 while 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 and set the Netlist Type for each design partition that will be imported to **Empty** in the Design Partitions window.
5. Create LogicLock regions for each of the lower-level partitions 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 project to lower-level designers using one of the following procedures:
    a. Provide a copy of the top-level Quartus II project framework. Use the **Copy Project** command on the Project menu or create a project archive. Provide each lower-level designer with the project.
    b. Use scripts to pass constraints and generate separate Quartus II projects. On the Project menu, click **Generate Bottom-Up Design Partition Scripts**, or run the script generator from a Tcl or command prompt. Make changes to the default script options as required for your project. Altera recommends that you pass all the default constraints, including LogicLock regions, for all partitions and virtual pin location assignments. Altera further recommends that you add a maximum delay timing constraint for the virtual I/O connections in each partition to help timing closure during integration at the top level. If lower-level projects have not been created by the other designers, use the partition script to set up the projects so that you can easily take advantage of makefiles. Provide each lower-level designer with the Tcl file to create their project with the appropriate constraints. If you are using makefiles, provide the makefile for each partition.
    c. Use documentation or scripts to manually pass all constraints and assignments to each lower-level designer.

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 may not have to create floorplan assignments for the entire design. 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 else in the device. To create a region, perform the following steps:

1. Divide up your design into partitions to reduce compilation time. Consider the guidelines in this chapter while determining the partition boundaries. Ensure that you isolate the timing-critical logic in a separate design partition.

2. Define a LogicLock region for the timing-critical design 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 as in the scenario "Create a Floorplan for the Entire Design" on page 8–49 to place the LogicLock region. Save the Fitter-determined size and origin, then enlarge the region to provide more flexibility and allow for future design changes.
3. As the rest of the design is completed, and the device fills up, the timing-critical region has a reserved area of the floorplan. When you make changes to the design block, the logic can be re-placed in the same part of the device, which helps ensure good quality of results.

Potential Issues with Creating Partitions and Floorplan Assignments

There are some limitations and restrictions when using incremental compilation and using certain design flows with certain Altera features.

For more information about 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*.

Consider documented limitations and restrictions as you plan your design flow and select partitions. Although most limitations and restrictions do not affect most users, but it is helpful to know if you must modify your partitions or design flow to accommodate certain restrictions.

There are also possible utilization effects due to partitioning and creating a floorplan. Consider these effects if your design is close to using all the device resources before adding partition or floorplan assignments.

The following subsections describe the utilization effects:

- "Logic and Resource Utilization Effects"
- "Routing Utilization Effects"

Logic and Resource Utilization Effects

Partitions can increase resource utilization due to cross-partition optimization limitations. Floorplan assignments can increase resource utilization because regions sometimes lead to unused logic. Follow the recommendations in this document to reduce these effects.

If your device is very full with the flat version of your design, you might not be able to use a complete incremental flow for the entire design. You can use a "partial" incremental flow instead to get compilation time and performance preservation benefits for key parts of the design. Focus on creating partitions and floorplan assignments for timing-critical or often-changing blocks to get the most benefit out of the feature.

Routing Utilization Effects

Partitions and floorplan assignments typically increase routing utilization compared to a flat design. Follow the recommendations in this document to reduce the effect.

If long compilation times are due to routing congestion, you might not be able to use incremental flows to reduce compilation time. Focus on creating partitions and floorplan assignments for parts of the design that are not routing-critical to get some benefit.

You can also use incremental compilation to lock routing for routing-critical blocks only (with other partitions empty), and then compile the rest of the design after the critical block meets its requirements.

Review the Fitter Messages to check how much time is spent during routing optimizations and to see the percentage of routing utilization. This information helps highlight routing issues.

Conclusion

Incremental compilation provides a number of benefits, especially to large, complex designs. To take advantage of the feature, it is worth spending some time to create quality partition and floorplan assignments.

Follow the guidelines to set up your design hierarchy and source code for incremental compilation. Keep partitions independent of each other and do not rely on any cross-boundary logic optimizations.

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.

Take advantage of the numerous Quartus II software tools to assess partition quality and analyze the floorplan to make good partition and LogicLock location assignments. 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.

 © November 2009 Altera Corporation

Referenced Documents

This chapter references the following documents:

- *Analyzing and Optimizing the Design Floorplan* chapter in volume 2 of the *Quartus II Handbook*
- *Area and Timing Optimization* chapter in volume 2 of the *Quartus II Handbook*
- *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook*
- *Quartus II Integrated Synthesis* chapter in volume 1 of the *Quartus II Handbook*

Revision History

Table 8–1 shows the revision history for this chapter.

Table 8–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
October 2009 v9.1.0	■ 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. ■ Added "Importing SDC Constraints from Lower-Level Partitions in Team-Based Designs" from the Quartus II Incremental Compilation for Hierarchical and Team-Based Design chapter in volume 1 of the *Quartus II Handbook*. ■ Reorganized the "Recommended Design Flows and Application Examples" on page 8–48 section. ■ Removed HardCopy APEX and HardCopy Stratix Devices section.	Updated for the Quartus II software version 9.1 release.
March 2009 v9.0.0	■ Added I/O register packing examples from *Incremental Compilation for Hierarchical and Team-Based Designs* chapter ■ Moved "Incremental Compilation Advisor" section ■ Added "Viewing Design Partition Planner and Floorplan Side-by-Side" section ■ Updated Figure 8–21 ■ Chapter 8 was previously Chapter 7 in software release 8.1.	Updated for the Quartus II software version 9.0 release.

Table 8–1. Document Revision History

Date and Document Version	Changes Made	Summary of Changes
November 2008 v8.1.0	Changed to 8-1/2 x 11 page size. No change to content.	Updated for the Quartus II software version 8.1 release.
May 2007 v8.0.0	Initial release.	This content of this chapter is based on information that was contained in Application Note 470.

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

© November 2009 Altera Corporation