Best Practices for Incremental Compilation Partitions and Floorplan Assignments

December 2007, ver. 1.0 Application Note 470

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 up to 70%
- Preserves performance for unchanged design blocks
  - Provides repeatable results and reduces the number of compilations
- Enables true team-based design

This document 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.

This document contains the following sections:

Overview: Quartus II 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" or non-incremental full compilation flow.

This document contains a brief description of the Quartus II incremental compilation feature and its uses. For details 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*.

In the default full compilation, all the source code is synthesized and all the logic is placed and routed when the design is recompiled. This means that the entire design must be recompiled even if only part of the design source code changes, if you apply any new timing constraints, or if you want to optimize part of the design using different optimization techniques.

A full recompilation can result in longer than expected compilation times, even for small changes to the design. The results, including timing performance, can change for unrelated parts of the design. In addition, designers who try to implement a bottom-up or team-based flow typically find it difficult to ensure that lower-level designs will achieve their performance requirements when integrated with other logic.

Incremental compilation is recommended for the following general scenarios to avoid the issues just described:

- Large or complex designs with long compilation times
- Designs that contain speed-critical portions
- Team-based design environments in which different designers perform placement and routing independently

The feature is especially useful in the largest Stratix® series devices. Incremental compilation is becoming increasingly important in the FPGA industry to reduce compilation times for large devices and to more easily achieve timing closure.

Summary of Steps for an Incremental Compilation Flow

The following procedure outlines the Quartus II incremental compilation flow:

1. Set up your design hierarchy and source code to support partitioning along logical hierarchy boundaries. If you are using a third-party synthesis tool, set up your tool to generate separate netlist files.

2. Create Design Partition assignments in the Quartus II software to specify which hierarchy blocks will be compiled independently as partitions (including empty partitions for any missing or incomplete logic blocks).

3. When the design is compiled, Quartus II Analysis & Synthesis and the Fitter create separate netlists for each partition. These netlists are internal post-synthesis and post-fit database representations of the design.

4. Select which netlist type to preserve for each partition in the subsequent compilation. You can reuse the synthesis or fitting database, or instruct the software to resynthesize the source files.

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 the previous step.

In some cases, as described in "Introduction to Design Floorplans" on page 31", you should create a design "floorplan" with placement assignments to constrain each part of the design to a specific region of the device.

If you are using a bottom-up or team-based methodology, you can create design partition scripts to pass top-level constraints (such as floorplan assignments or optimization constraints) to the designers of lower-level blocks. After parts of the design are optimized in separate Quartus II projects, you can import the partition netlists and assignments into the top-level project.

Choosing the Netlist Type and Fitter Preservation Level

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

- Source File: Resynthesize the source code (with any new assignments)
  - If you modify a source file, the software automatically resynthesizes the source file for most Netlist Type settings
  - Assignments do not trigger an automatic recompilation
- Post-Synthesis (default): Re-fit the design (with any new Fitter assignments) but preserve the synthesis results
- Post-Fit: Preserve placement and performance results
  - The default setting for post-fit is to preserve placement and reroute the entire design; this usually allows the router to find the best routing for all partitions given their placement on the design, and gives very good performance preservation
- Post-Fit with Fitter Preservation Level = Placement and Routing: Preserve routing, only if necessary
  - Use post-fit with routing if necessary to meet the timing requirements for specific partitions

Top-Down versus Bottom-Up Compilation Flows

The Quartus II incremental compilation feature supports both top-down and bottom-up compilation flows.

With top-down compilation, one designer compiles the entire design in the software. You can use a top-down flow to optimize all blocks of the design together, or to optimize one or more critical design blocks or IP cores before adding the rest of the design. You can preserve fitting results and performance for completed blocks while other parts of the design are changing, which also reduces the compilation times for each design iteration. Different designers or IP providers can create and verify HDL code separately, but one person (generally the project lead or system architect) compiles and optimizes the design as a single top-level project.

Bottom-up design flows allow individual designers or IP providers to complete the optimization of their design in separate projects and then integrate each lower-level project into one top-level project. Incremental compilation provides export and import features to enable this design methodology. Designers of lower-level blocks can export the optimized placed and routed netlist for their design, along with a set of assignments such as LogicLock regions. The project lead then imports each design block as a design partition in a top-level project.

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

A top-down flow is generally simpler to perform than its bottom-up counterpart. For example, the need to export and import lower-level designs is eliminated. In addition, a top-down approach provides the design software with information about the entire design so it can perform global placement and routing optimizations. Therefore, it is often easier to ensure good quality of results with a top-down flow than with a bottom-up flow.

The Quartus II incremental compilation feature is very flexible and supports numerous design methodologies. You can mix top-down and bottom-up flows within a single project. If the top-level design includes one or more design blocks that are optimized by different designers or IP providers, you can import those blocks (using a bottom-up methodology) into a project that also includes partitions for a top-down incremental methodology. 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.

By following a mixed design methodology, you can take advantage of the team-based capabilities of a bottom-up flow while maintaining the advantages of a top-down flow for most of the design logic.

Bottom-up incremental compilation is not supported in HardCopy® II or Stratix II device compilations when there is a migration device setting. You cannot use a bottom-up methodology if you want to migrate to a HardCopy II device. 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, using the bottom-up flow and importing partitions does not support this requirement.

Why Plan for Incremental Compilation?

Incremental compilation flows may require more up-front planning than full “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 that you be more rigorous about following good design practices than flat compilations.

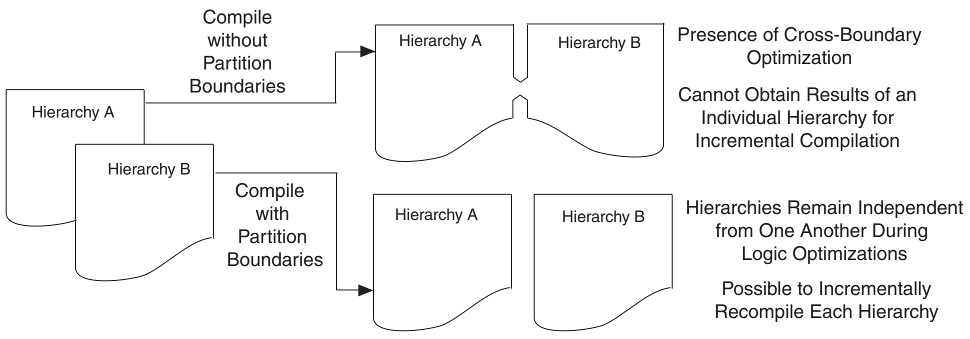
Planning consists of planning logic for partitioning and planning placement assignments for creating a floorplan. Not all design flows require floorplan assignments (as discussed later in this document). If you decide to add floorplan assignments later, when the design is close to completion, well-planned partitions make floorplan creation much easier. Beware: poor partition or floorplan assignments can hurt the design area utilization and performance, making timing closure more difficult. Use the guidelines provided in this document when planning your design to help ensure good quality of results.

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. As FPGA devices get larger and more complex, following good design practices becomes more important for all design flows. 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. To get the most out of the feature, follow the guidelines presented in this document.

Partition Boundaries and Optimization

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

Figure 1. Effects of Partition Boundaries during Logic Optimization


Compile without Partition Boundaries
Hierarchy A
Hierarchy B
Presence of Cross-Boundary Optimization
Cannot Obtain Results of an Individual Hierarchy for Incremental Compilation
Hierarchy A
Hierarchy B
Compile with Partition Boundaries
Hierarchy A
Hierarchy B
Hierarchies Remain Independent from One Another During Logic Optimizations
Possible to Incrementally Recompile Each Hierarchy

To avoid cross-boundary optimizations, each partition is synthesized without using any information about logic contained in other partitions. In a flat compilation, the software uses unconnected signals, constants, inversions, and other design information to perform optimizations. When a design is split into partitions, 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 cannot perform any logic optimizations such as physical synthesis across the partition boundary.) When partitions are fit separately in a bottom-up flow 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 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 might decrease as the number of partitions increases. Although more partitions allow for greater reduction in compilation time, you might want to limit 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. Plan your design with incremental compilation partitioning guidelines in mind. This section includes the following topics:

- "Plan Design Hierarchy and Source Design Files" on page 9
- "Partition Design by Functionality and Block Size" on page 11
- "Partition Design by Clock Domain and Timing Criticality" on page 11
- "Consider What Is Changing" on page 12

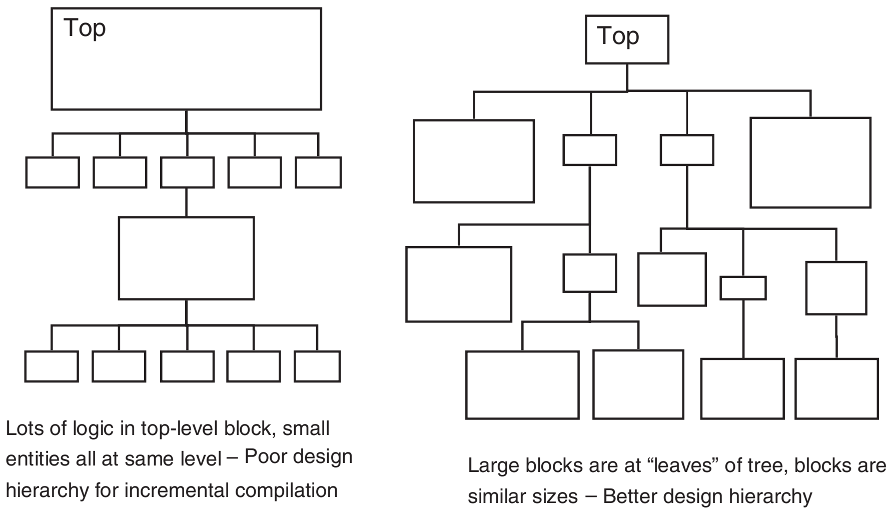
Plan Design Hierarchy and Source Design Files

Start by planning the entities in the design hierarchy. When you assign a hierarchical entity as a design partition, the partition includes the assigned entity and any entities instantiated below it that are not defined as separate partitions. You cannot group separate hierarchical entities into one 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 even-sized partitions. For example, do not create a lot of small entities at the same hierarchy level because it will be difficult to group them to form reasonably sized partitions.

Figure 2 represents the logic blocks in a design hierarchy. The left side does not follow the recommendations for entity organization, while the right side provides much more flexibility for creating good partitions.

Figure 2. Design Hierarchy Recommendations


Top
Lots of logic in top-level block, small entities all at same level – Poor design hierarchy for incremental compilation
Top
Large blocks are at "leaves" of tree, blocks are similar sizes – Better design hierarchy

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 (for most Netlist Type settings). If the design entities for two partitions are defined in the same file, changes to the logic in one partition trigger recompilation for both partitions.

Design dependencies also affect which partitions are compiled when a source file changes. 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.

Planning your design hierarchy and setting up design files to accommodate incremental compilation provides a good foundation to create design partitions as you develop the design source code.

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 are using a third-party synthesis tool, set up your tool to create a separate VQM or EDIF netlist for each hierarchical partition. In the Quartus II software, assign the top-level entity from each netlist to be a Design Partition. The VQM or EDIF netlist file is treated as the source file for the partition in the Quartus II software.

Synplicity Synplify Pro and Mentor Graphics Precision RTL Plus

The Synplify Pro software includes the MultiPoint synthesis feature to perform incremental synthesis for each design block assigned as a Compile Point in the user interface or a script. The Precision RTL Plus software includes an incremental synthesis feature that performs block-based synthesis based on Partition assignments in the source HDL code.

These features provide automated block-based incremental synthesis flows and create different output netlist files for each block when set up for an Altera® device. Using incremental synthesis within the synthesis tool ensures that only those sections of a design that have been updated are resynthesized when the design is compiled, reducing synthesis run time and preserving the results for the unchanged blocks. You can change and resynthesize one section of a design without affecting other sections of the design.

For more information about these incremental synthesis flows, refer to your tool vendor's documentation.

Other Synthesis Tools

You can also partition your design and create different netlist files manually with the Synplify software (non-Pro), the Precision RTL software (non-Plus), or any other supported synthesis tool by creating a separate project or implementation for each partition, including the top level. Set up each higher-level project to instantiate the lower-level VQM/EDIF netlists as black boxes. Synplify, Precision, and most synthesis tools automatically treat a design block as a black box if the logic definition is missing from the project. Each tool also includes options or attributes to specify that the design block should be treated as a black box, which you can use to avoid warnings about the missing logic.

Partition Design by Functionality and Block Size

In the first pass, partition your design along functional boundaries. Start with the top-level system block diagram, and you will often find that each block is a natural design partition. Typically, each block of a system is independent and has more signal interaction internally than interaction between blocks, which helps reduce the need for 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, because that helps dictate 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, as explained in a previous section, 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 ~2000 LEs or ALMs. If your design is not yet complete, use previous designs to help you estimate the size of each block as you partition the design.

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 any 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.

As with any good synchronous design, you should 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, think about what is changing in the design. Is there any 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 once and immediately preserve the results, and you will not have to compile that part of the design again. Is some 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 stays the same.

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 of these recommendations have to be followed exactly to be successful with incremental compilation, but adhering to as many as possible will maximize your chances of success.

This section includes the following topics:

- "Register Partition Inputs and Outputs" on page 12
- "Minimize Cross-Partition-Boundary I/O" on page 13
- "Avoid the Need for Logic Optimization Across Partitions" on page 15

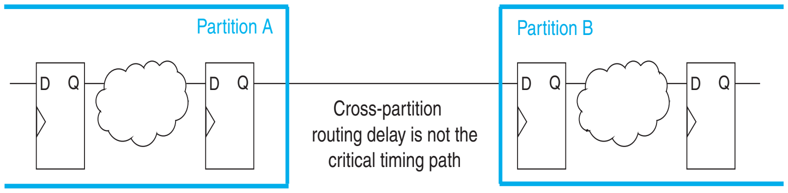
This last subsection includes examples of the types of optimizations that are prevented by partition boundaries, and describes how you should structure your partitions to avoid the need for such optimizations.

Register Partition Inputs and Outputs

Register partition input and output connections that are potentially timing-critical. Registers minimize the delays on inter-partition paths, and avoid the need for cross-boundary logic optimizations.

If every partition boundary has a register as shown in Figure 3, every register-to-register timing path between partitions includes only routing delay. Because the timing paths between partitions are not critical, the placement of each partition does not depend on other partitions. This advantage makes it easier to create floorplan location assignments for each separate partition, and is especially important in bottom-up flows in which each partition is placed independently. 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 3. Registering Partition I/O


Partition A
D
Q
D
Q
Cross-partition routing delay is not the critical timing path
Partition B
D
Q
D
Q

If a design cannot include both inputs and output registers for each partition due to latency or resource utilization concerns, designers typically 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 at least 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 Reports" on page 42 list how many I/O are registered or unregistered, and the advisor described in "Incremental Compilation Advisor" on page 43 lists the unregistered ports for each partition.

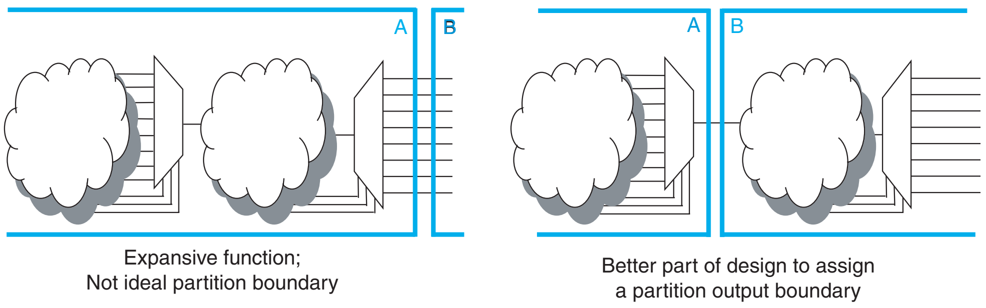
Minimize Cross-Partition-Boundary I/O

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. 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.

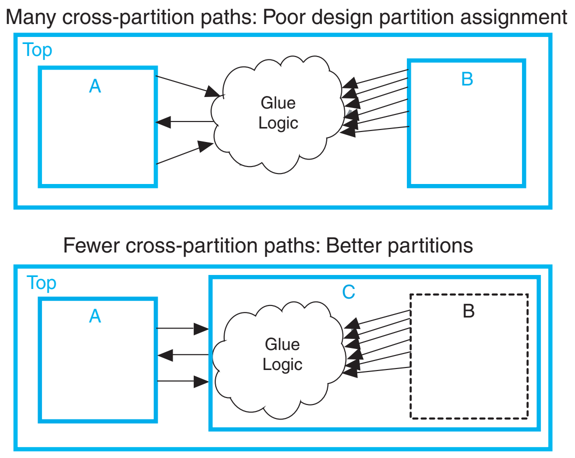
When dividing your design into partitions, consider the types of functions at the partition boundaries. Figure 4 shows an expansive function with more outputs than inputs, making a poor partition boundary. Adding registers to one or both sides of the cross-partition path in this example would improve the partition quality even more.

Figure 4. Minimizing I/O between Partitions by Moving the Partition Boundary


A
B
A
B
Expansive function;
Not ideal partition boundary
Better part of design to assign
a partition output 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 5 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 5. Minimizing I/O between Partitions by Modifying Glue Logic


Many cross-partition paths: Poor design partition assignment
Top
A
Glue Logic
B
Fewer cross-partition paths: Better partitions
Top
A
C
Glue Logic
B

The statistics described in "Partition Statistics Reports" on page 42 list the number of I/O ports as well as the number of inter-partition connections for each partition. The advisor described in "Incremental Compilation Advisor" on page 43 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 7, 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 already 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 the 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 design 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:

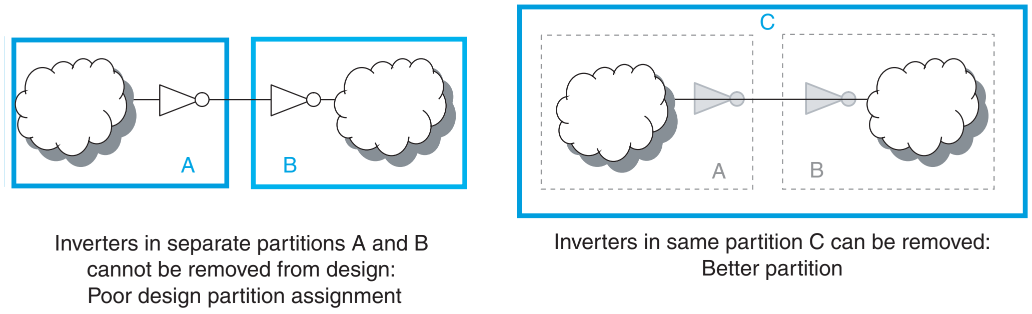
- "Keep Logic in the Same Partition for Optimization and Merging"
- "Keep Constants in the Same Partition as Logic" on page 18
- "Avoid Unconnected Partition I/O" on page 18
- "Avoid Signal Driving Multiple Partition I/O, or Connecting I/O Together" on page 20
- "Invert Clocks in Destination Partitions" on page 21
- "Do Not Use Internal Tri-States" on page 22
- "Include All Tri-State Output Logic in the Same Partition" on page 23
- "Include All I/O Registers in the Same Partition" on page 24

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 6. 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. On the right side of the figure, a new hierarchy block C has been created and defined as a partition to group the logic in blocks A and B instead of having two separate partitions. With the logic contained in one partition, the software can optimize the logic and remove the two registers (shown in gray color), which reduces the delay for that logic path. Removing two registers 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 6. Keeping Logic in the Same Partition for Optimization


A
B
C
A
B
Inverters in separate partitions A and B cannot be removed from design: Poor design partition assignment
Inverters in same partition C 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, transceiver altgxb/alt2gxb instances can share high-speed serial interface (HSSI) resources in the same quad as other instances.

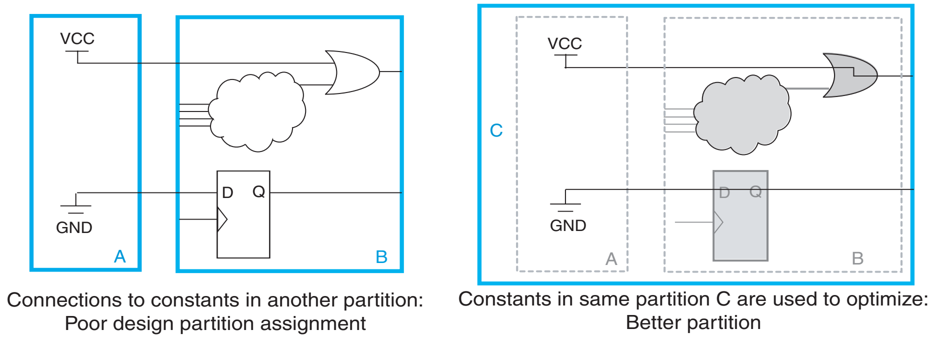
Beginning with the Quartus II software version 7.2, the Fitter can merge multiple instantiations of these blocks into the same device resource, even if it requires optimization across partitions. Therefore, there are no restrictions for PLLs and high-speed transceiver blocks when setting up partitions.

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/VCC or 0/GND) in one partition cannot affect another partition.

For example, the left side of Figure 7 shows part of a design in which partition A defines some signals as constants. Constants like this could appear due to parameter/generic settings or configurations with parameters, or 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 7 shows a new partition C that groups the logic in blocks A and B, instead of having the two separate partitions. 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 7. Keeping Constants in the Same Partition as the Logic They Feed


VCC
GND
A
D
Q
B
Connections to constants in another partition:
Poor design partition assignment
C
VCC
GND
A
D
Q
B
Constants in same partition C are used to optimize:
Better partition

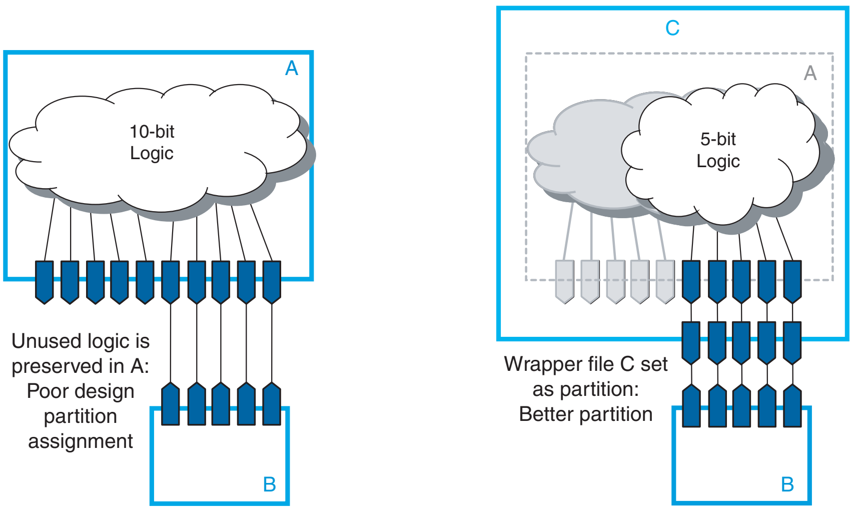
The statistics described in "Partition Statistics Reports" on page 42 list how many input ports are fed by ground or VCC, and the advisor described in "Incremental Compilation Advisor" on page 43 lists the ports.

Avoid Unconnected Partition I/O

When a port is left unconnected, optimizations might be able to remove logic driving that port and improve results. However, these optimizations are not allowed across partitions in incremental compilation, because they would create cross-partition dependence. 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 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 case, 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. Avoiding Unconnected Partition I/O by Creating a Wrapper File


A
10-bit Logic
Unused logic is preserved in A: Poor design partition assignment
B
C
A
5-bit Logic
Wrapper file C set as partition: Better partition
B

The statistics described in "Partition Statistics Reports" on page 42 list how many I/O are unconnected, and the advisor described in "Incremental Compilation Advisor" on page 43 lists the unconnected ports.

Avoid Signal Driving Multiple Partition I/O, or Connecting I/O Together

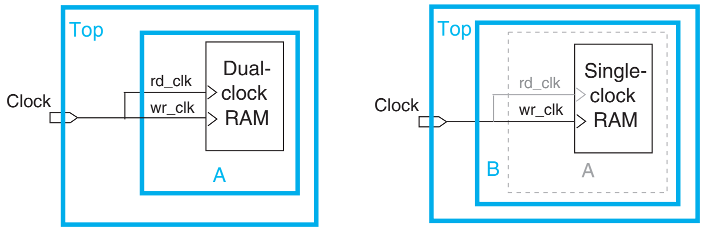
Do not use the same signal to drive multiple ports on a single partition, or directly connect two ports of a partition.

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 in these cases.

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.

As an example of one signal driving more than one port, refer to Figure 9. 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 9. Preventing One Signal from Driving Multiple Partition Inputs



Two clocks cannot be treated as the same signal: Poor design partition assignment

With Partition B, RAM can be optimized for one clock: Better partition

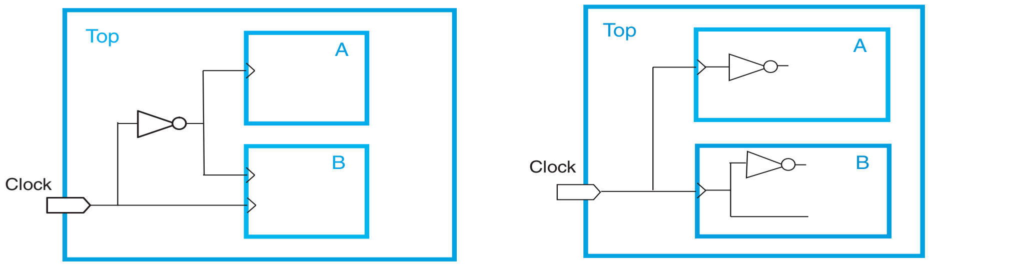
The advisor described in "Incremental Compilation Advisor" on page 43 lists partition ports that have the same driving signal, and ports that are directly connected together.

Invert Clocks in Destination Partitions

Clock inversion should be done in the destination logic array block (LAB) for best results, 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 10, 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 10. Inverting Clock Signal in Destination Partitions



Inverter acts as clock gating (skew!): Poor design partition assignment

Clock inverted inside destination LABs, only one global routing signal: Better partition

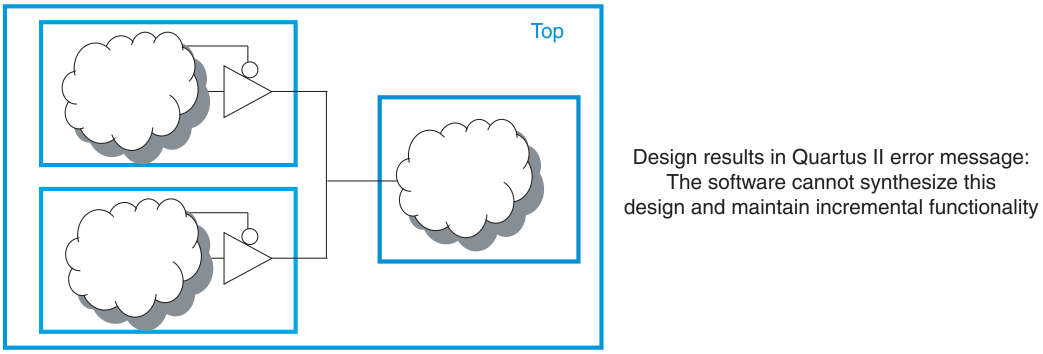
Notice that this diagram also shows another example of a single pin feeding two ports of a partition boundary. In the left diagram, partition B does not have the information that the clock and inverted clock come from the same source. In the right diagram, partition B has more information to help optimize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

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 typically 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 11 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 11. Unsupported Internal Tri-State Signals


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

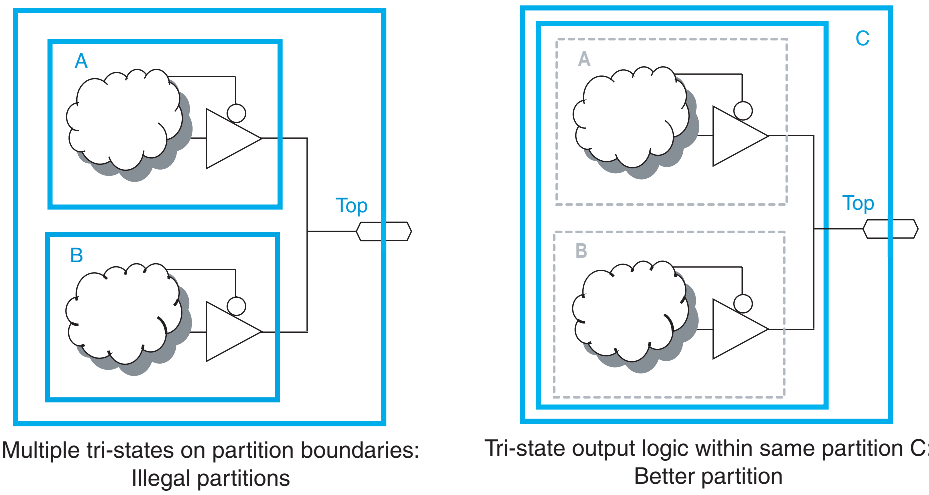
Do not use tri-state signals or bidirectional ports on hierarchical partition boundaries, unless the port is connected directly to a top-level I/O pin on the device. If you must use internal tri-state logic, ensure that all the control and destination logic is contained in the same partition, in which case the software can convert the internal tri-state signals into multiplexing logic like 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 Output 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.

Figure 12 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 C, 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 12. Including All Tri-State Output Logic in the Same Partition

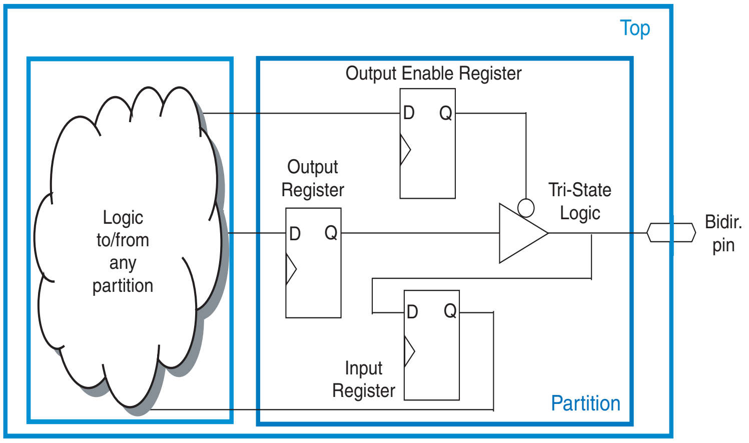

A
B
Top
Multiple tri-states on partition boundaries:
Illegal partitions
A
B
C
Top
Tri-state output logic within same partition C:
Better partition

Include All 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 to the Stratix II family, Cyclone® II family, and all older Altera device families that include I/O registers. 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 13, 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 13. 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

Following the guidelines presented in this section will 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 signal driving multiple partition I/O, or connecting I/O together
- Invert clocks in destination partitions
- Do not use internal tri-states
- Include all tri-state output logic in the same partition
- Include all I/O registers in the same partition

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 already complete, assess whether assigning a partition affects the resource utilization or timing performance of a design block as compared to the flat design, and make the appropriate changes to your design or hierarchy to improve your results.

Creating Design Partitions: Consider Additional Design Suggestions

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

- "Consider a Cascaded Reset Structure"
- "Balance Partition Resources if Required" on page 27
- "Drive Clocks Directly in Bottom-Up Flows" on page 30
- "Recreate PLLs for Lower-Level Partitions if Required in Bottom-Up Flows" on page 30

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.

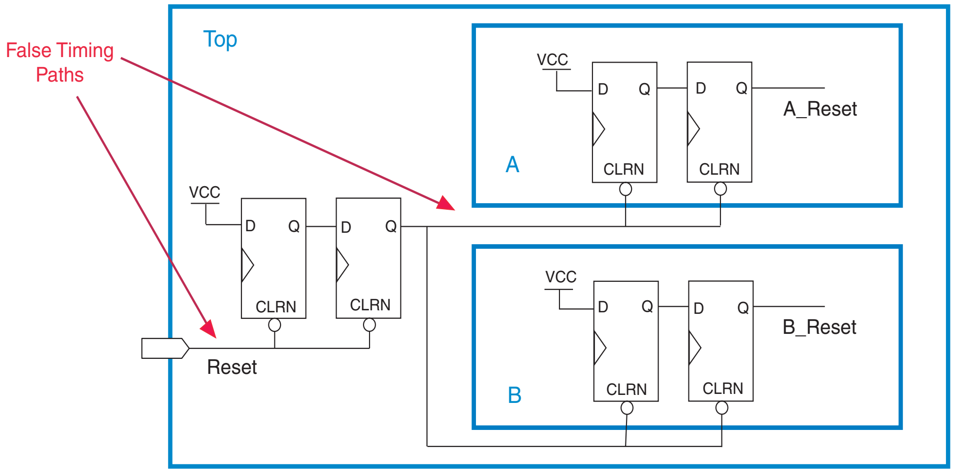
For incremental compilation, minimizing the impact of global structures is helpful. To isolate each partition, consider adding reset synchronizers. By using cascaded reset structures, the design intent is to reduce the inter-partition fan-out of the reset signal, thereby minimizing the effect of the 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 need 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 14 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 become 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 14. Cascaded Reset Structure



Altera suggests this circuit as one that 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.

Balance Partition Resources if Required

When 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, and thus the design may not fit in the target device when the partitions are merged.

In a bottom-up 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.

To avoid these effects, you may have to perform manual resource balancing across partitions. This is more applicable to bottom-up design flows, because top-down compilation usually handles resource balancing without any user intervention.

RAM and DSP Blocks

In the standard synthesis flow, when DSP blocks or RAM blocks are overused, the Quartus II Compiler can perform automated resource balancing and convert some of the logic into regular logic cells. Without data about resources used in other partitions, it is possible for the logic in each separate partition to maximize the use of a particular device resource, such that the design does not fit after all the partitions are merged.

In such a case, you may be able to manually balance the resources by using the Quartus II synthesis options to control inference of megafunctions that use the DSP or RAM blocks. You can also use the MegaWizard® Plug-In Manager to customize your RAM or DSP megafunctions to use regular logic instead of the dedicated hardware blocks.

For more information about resource balancing 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*.

As discussed later in this document, it is often helpful to create a LogicLock region to isolate each partition's placement, especially in bottom-up flows, 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.

Global Routing Signals

If your design is very complex and has many clocks, you may have to allocate global routing resources. 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 will 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.

Use the **Global Signal** assignment to force or prevent the use of a global routing line. 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.

If you have all partition logic 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 when optimizing the lower-level partitions in separate Quartus II projects.

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 you want to disable the automatic global promotion performed in the Fitter, turn off the **Auto Global Clock** and **Auto Global Register Control Signals**.

If you are performing a bottom-up flow using the 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 "Generating Bottom-Up Design Partition Scripts" on page 45.

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.

Drive Clocks Directly in Bottom-Up Flows

In bottom-up flows, 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 bottom-up 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 the "Invert Clocks in Destination Partitions"section.

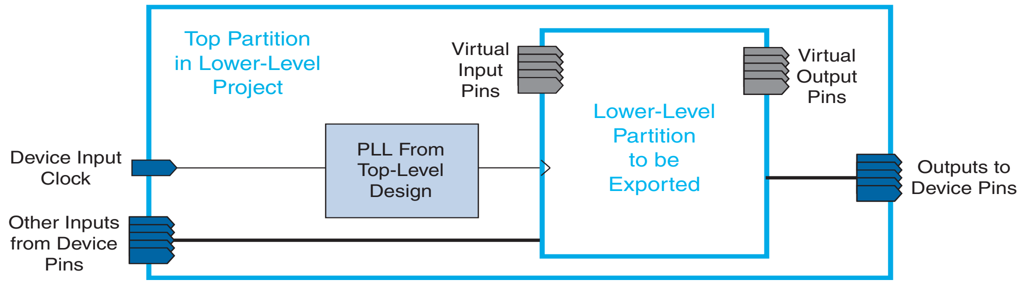
Direct connections to input clock pins also allows design partition scripts to send constraints from the top-level device pin to the lower-level partitions.

Recreate PLLs for Lower-Level Partitions if Required in Bottom-Up 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 unconstrained or constrained with an incorrect frequency. Alternately, you can manually 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.

Include a copy of the top-level PLL in your lower-level project as shown in Figure 15, and create a design partition for the rest of 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 (**.qxp**) file but does not include logic defined outside the partition (the PLL in this example).

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


Top Partition in Lower-Level Project
Virtual Input Pins
Virtual Output Pins
Lower-Level Partition to be Exported
PLL From Top-Level Design
Device Input Clock
Other Inputs from Device Pins
Outputs to Device Pins

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 are using 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 using a LogicLock region assignment. Assigning design partitions to LogicLock regions to create a design floorplan is recommended to improve the quality of results and avoid placement conflicts in many situations for incremental compilation. Refer to the following section for details. 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 in bottom-up flows
- To ensure a good quality of results when recompiling individual partitions in top-down flows

Why Create a Floorplan in Bottom-Up Flows?

Creating a design floorplan is required if you want to preserve placement for lower-level partitions in a bottom-up flow 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 in a bottom-up flow.

Why Create a Floorplan in Top-Down Flows?

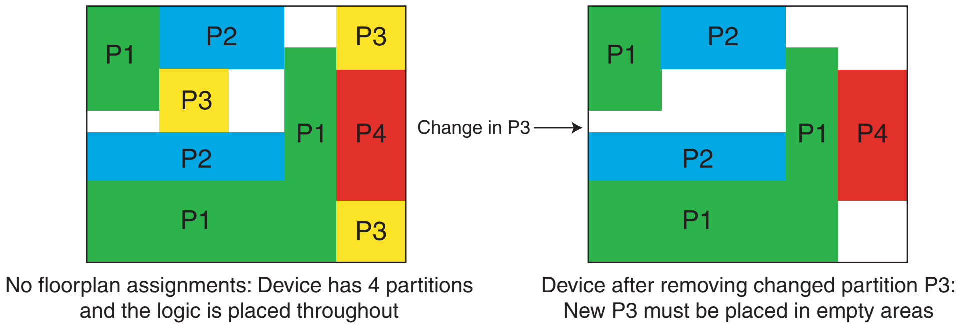
Creating a floorplan is highly recommended for timing-critical partitions to maintain good quality of results when the design changes.

Floorplan assignments are not required for non-critical partitions in a top-down flow. 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 already 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 16 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 using the scattered white space shown in the figure. The placement of the post-fit netlists for other partitions forces the Fitter to implement P3 with the device resources that have not already been used.

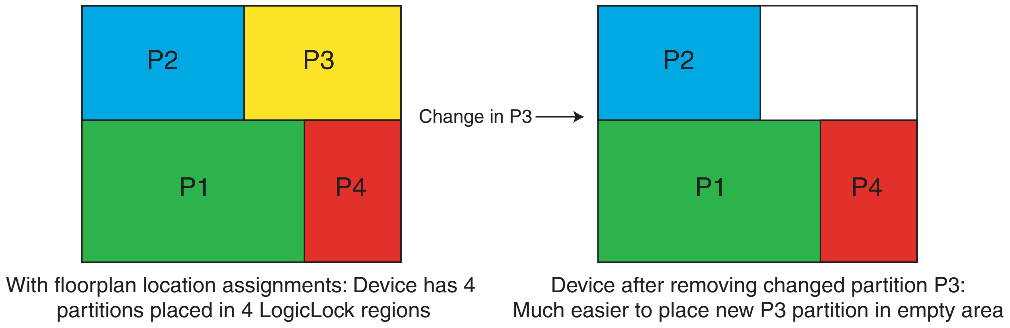
Figure 16. Representation of Device Floorplan without Location Assignments


P1
P2
P3
P3
P1
P4
P2
P1
P3
Change in P3
P1
P2
P1
P4
P2
P1
No floorplan assignments: Device has 4 partitions and the logic is placed throughout
Device after removing changed partition P3: New P3 must be placed in empty areas

The Fitter must work harder because of the 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 17 shows the initial placement of a four-partition design with floorplan location assignments. Each partition has been 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 17. Representation of Device Floorplan with Location Assignments


P2
P3
P1
P4
Change in P3
P2
P1
P4
With floorplan location assignments: Device has 4 partitions placed in 4 LogicLock regions
Device after removing changed partition P3: Much easier to place new P3 partition in empty area

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

You can create a floorplan at different stages of the design flow. This section describes two major categories of floorplans based on the design stage: early floorplan and late floorplan.

Regardless of when you create the floorplan, it is important that you plan early to incorporate partitions into the design, and ensure that each design partition follows the partitioning guidelines. These guidelines will 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 chip. 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 as a rough draft of a floorplan for top-down flows as well, to roughly divide up the design partitions into LogicLock regions while iterating through the design cycle.

In a top-down flow, or after you have integrated the first version of all design partitions in a bottom-up 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 has been created: The code is close to complete and the design structure is likely to stay consistent. When the design is complete, you can take advantage of Quartus II analysis features to check the floorplan quality. To tune the floorplan, you can perform iterative compilations as needed and assess the results of different assignments.

☞ It may not be possible to create a good-quality late floorplan if you have not planned for the partitions in 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 will enhance timing performance, as compared to a flat compilation with no location assignments. This may be true with other tools, but is not generally true in the Quartus II software. 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 in Top-Down Flows?" on page 32. 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 strategy 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 details 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 in a bottom-up design and can lead to better performance preservation when locking down parts of a project in a top-down design.

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. For best results, ensure that you recompile all such partitions 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 though 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.

Alternately, 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.

After a compilation or early timing estimate, save the Fitter's size and/or 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. On average, regions with fixed size and location yield better timing performance than auto-sized regions.

Modifying Region Size and Origin

After you have saved 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 region location and borders in the Chip Planner.

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. This is a requirement in bottom-up flows and a recommendation in top-down flows. In a bottom-up flow, if two partitions are allocated an overlapping portion of the chip, each may independently claim some common resources in this region. This will lead to resource conflicts when importing bottom-up results into a final top-level design. In a top-down flow, 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 bottom-up 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).

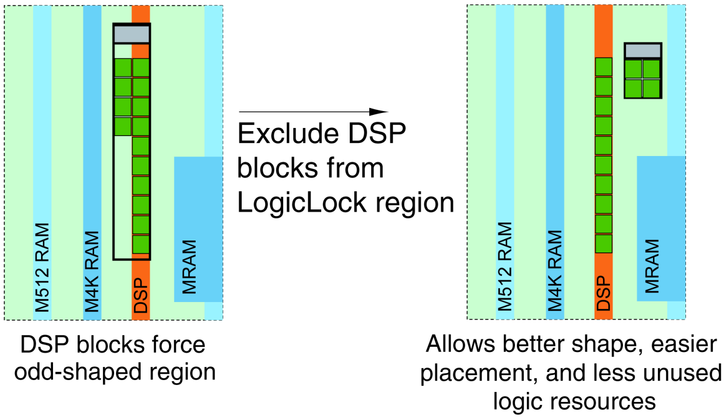
LogicLock Resource Exclusions

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. Note that 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, because of 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 18 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, and the DSP blocks are placed outside the region.

Figure 18. LogicLock Resource Exclusion Example


M512 RAM
M4K RAM
DSP
MRAM
DSP blocks force odd-shaped region
Exclude DSP blocks from LogicLock region
M512 RAM
M4K RAM
DSP
MRAM
Allows better shape, easier placement, and less unused logic resources

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.

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

Quartus II Tools for Partitions and Floorplans

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

Locate Design Instance in the Floorplan

After the first compilation of a complete design with no partitions and no LogicLock regions, you can view where the Fitter placed the logic for a specific design instance.

From the Project Navigator, right-click on an instance, point to **Locate**, and click **Locate in Chip Planner (Floorplan & Chip Editor**). The instance is highlighted in a dark blue color.

This information can help you understand the natural groupings between design instances in the flat design, which can help you decide which instances should remain grouped together within a single partition and possibly LogicLock region, and which instances are independently placed.

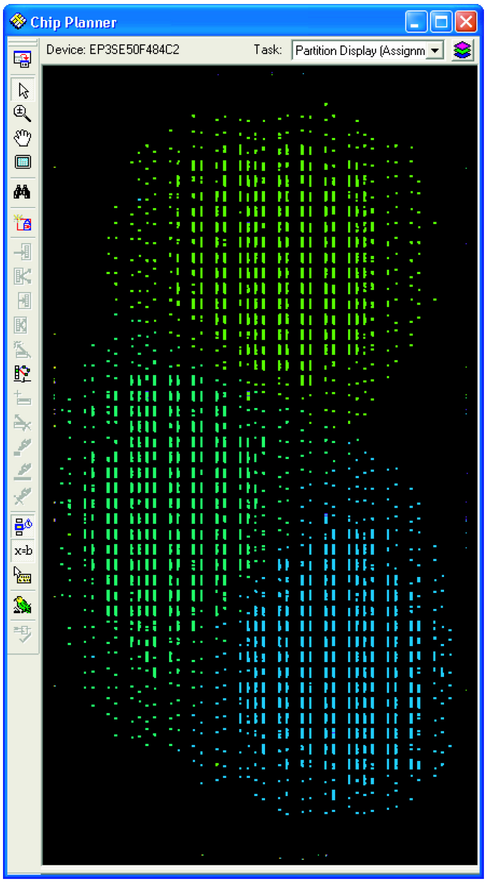
Floorplan Partition Coloring

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.

To take advantage of this feature, you can assign many instances as partitions and then view the natural placement grouping. This information can help you decide which instances should be grouped together, and which ones make good independent partitions.

In the Chip Planner **Task** list, choose **Partition Display (Assignment)**, as shown in Figure 19. In this figure, you can see that the three different-colored partitions are grouped in three fairly independent areas of the device.

Figure 19. Partition Display in the Chip Planner



Partition Statistics Reports

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 in a top-down compilation flow, or when you are compiling the full top-level design in a bottom-up 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 VCC 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.

Incremental Compilation Advisor

You can use the Incremental Compilation Advisor to check that your design follows the recommendations for creating design partitions and floorplan location assignments that are presented in this document.

On the Tools menu, point to **Advisors**, and click **Incremental Compilation Advisor**. Recommendations are split into General Recommendations that apply to all compilation flows and Bottom-Up Design Recommendations that apply to bottom-up design methodologies. 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 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 of the partition I/O ports follow the Register All Ports recommendation, the advisor displays a list of unregistered ports with the partition name and the source and destination nodes for the port. When the 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.

☞ The first time you open the RTL or Technology Map Viewer, a preprocessor stage runs. This preprocessor resets the Incremental Compilation Advisor, so you must rerun the Check Recommendations process. Alternatively, you can open the appropriate netlist viewer before you use the Incremental Compilation Advisor if you want to locate nodes in the viewer.

Critical Path Display for the Quartus II Classic Timing Analyzer

The Critical Path Display option shows the top critical paths from Classic 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 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.

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 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 Reports" on page 42, 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.

Inter-Region Connection Bundles

The Chip Planner can display bundles of connections between LogicLock regions, with filtering options 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, choose **Generate Inter-region Bundles**.

Routing Utilization

The Chip Planner includes a mode to display a "thermal 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 over a LAB to see a tool tip reporting the logic and routing utilization information.

Generating Bottom-Up Design Partition Scripts

The bottom-up design partition scripts automate the process of transferring top-level project information (including floorplan assignments) to lower-level projects. This feature provides a project manager interface for managing resource and timing budgets in the top-level design. This interface makes it easier for designers of lower-level modules 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 to further optimize the designs after integration, and improves overall designer productivity and team collaboration.

This feature creates Tcl files that each designer can run to set up a project and makefiles for designers who use a make environment.

Set up the top-level project with appropriate constraints and floorplan to be passed to lower levels. Then generate design partition scripts after successful compilation of the top-level design. You can perform an Early Timing Estimation instead of full compilation to reduce compilation time. The top-level design can have empty partitions when you generate the scripts.

On the Project menu, click **Generate Bottom-Up Design Partition Scripts** and set the appropriate options.

Virtual Pins in Bottom-Up Flows

Virtual pins map lower-level design I/O to internal cells. Use them when the number of I/O 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.

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.

Potential Issues with Creating Partitions and Floorplan Assignments

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

☞ Refer to the *Quartus II Incremental Compilation for Hierarchical and Team-Based Design* chapter in volume 1 of the *Quartus II Handbook* for complete details about restrictions and limitations.

Consider the documented limitations and restrictions as you plan your design flow and select partitions. 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. These are described in the following subsections. Consider these effects if your design is close to using all of the device resources before adding partition or floorplan assignments.

Logic/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 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 the 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 the routing optimizations, and to see the percentage of routing utilization. This information will help highlight any 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 for bottom-up flows, and are recommended for timing-critical partitions in top-down flows. 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 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 as closely as possible will maximize your chances of success.

Referenced Documents

This application note 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

The following table shows the revision history for this application note.

Date and Document Version	Changes Made	Summary of Changes
December 2007 v1.0	Initial release.	—

101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support:
www.altera.com/support/
Literature Services:
literature@altera.com

I.S. EN ISO 9001