# Tips for Incremental Compilation

# And LogicLock

Version 1.0 - February 6th, 2012 - Quartus II 11.1SP1 - by Ryan Scoville

## Table of Contents

# Introduction

This document was put together based on my experiences(and many others) with Incremental Compilation and LogicLock. I tried to explicitly put each "Tip" into its own section, so that the user could easily identify them and understand them. That being said, a user will have the most success with this document by reading most of it and trying to get an overall sense of what's possible, and then applying that knowledge to their specific design and their specific circumstance. Just randomly taking any specific tip and saying, "Let's try this" will unlikely be successful.

My first overall recommendation is to proceed thoughtfully, understanding the trade-offs with any particular approach, and using intelligence to see if it will work for their design. Then, no matter if the results are better or worse, analyzing the results to determine why. Quite often, I find an approach that fails will give me valuable information for my next step, as long as I take the time to analyze it. For example, let's say I floorplan a design but get worse timing results. Some users will see a worse slack, say the approach didn't work and move on. But a careful analysis of why it didn't work might show what paths broke, how they broke, and if the user can fix them. Sometimes they reveal an unknown connection in the design that pulls many hierarchies together, making them all worse, and by fixing that one pressure point, the user is able to loosen the design so the fitter can better optimize other blocks.

It is this type of analysis that is the most difficult to do, but can often provide the most value. Because of that, before even discussing Incremental Compilation or LogicLock, it's worth starting with a bit about timing analysis.

# Timing Analysis

TimeQuest is where we start when analyzing critical paths. TimeQuest's macro "Report All Summaries" is what I usually run first, to see failing setup, hold, recovery and removal paths. Most failures fall under setup analysis, and the user will right-click on the worst domain and run "Report Timing". They now have a detailed analysis of the worst paths. This is an excellent place to do low-level analysis, which is where to start. If the logic can be reduced, register's added, logic duplicated, timing requirements modified, or any of the myriad of tricks that allow a path to meet timing more easily, then please try those first. But once those methods are exhausted, user's can analyze the placement and routing and wonder if they can improve upon it.

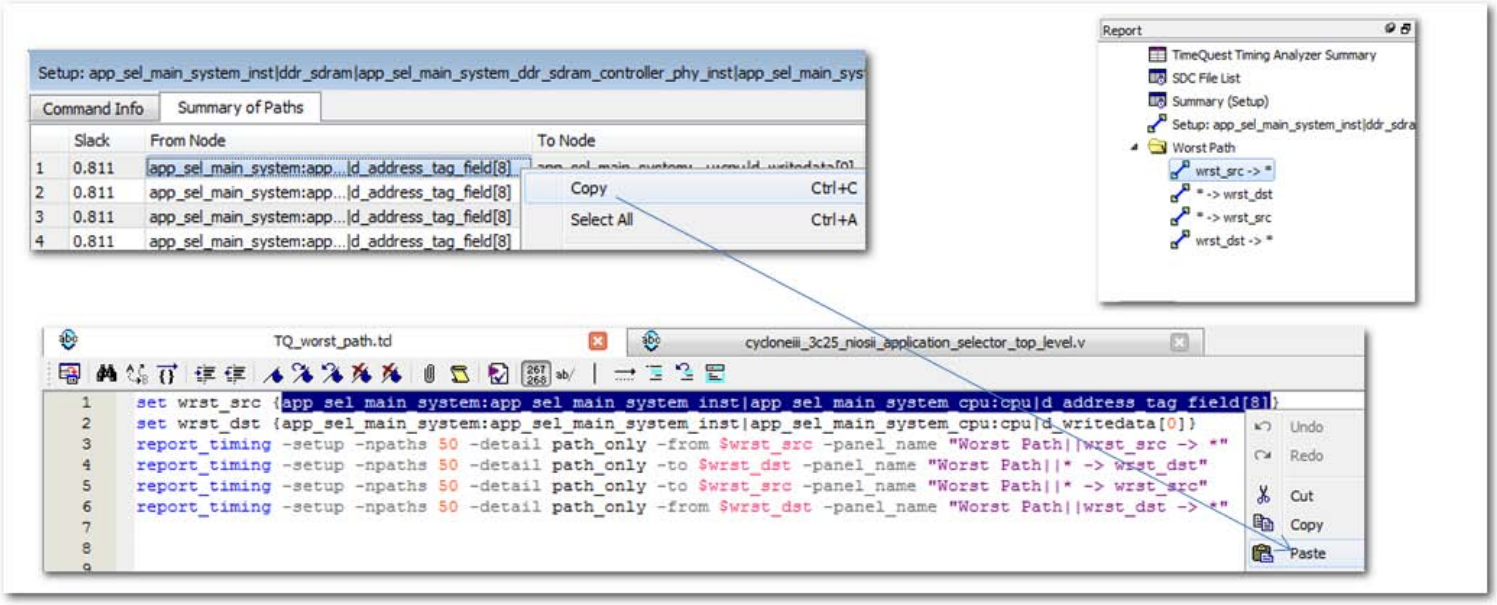
Most likely, the placement of the critical path is not ideal. There will hops between LABs, the distances may seem further than expected, and users often wonder why the fitter couldn't do a better job. It is important to remember that, of the hundreds of thousands of paths in the design, they all can't have perfect placement, and this is the worst one. Just as importantly, any path with multiple levels of logic will have other paths to/from these registers, all pulling in their own direction, and the fitter must balance timing between all these requirements.

## Tip – Analyze paths from/to the source and destination of critical path

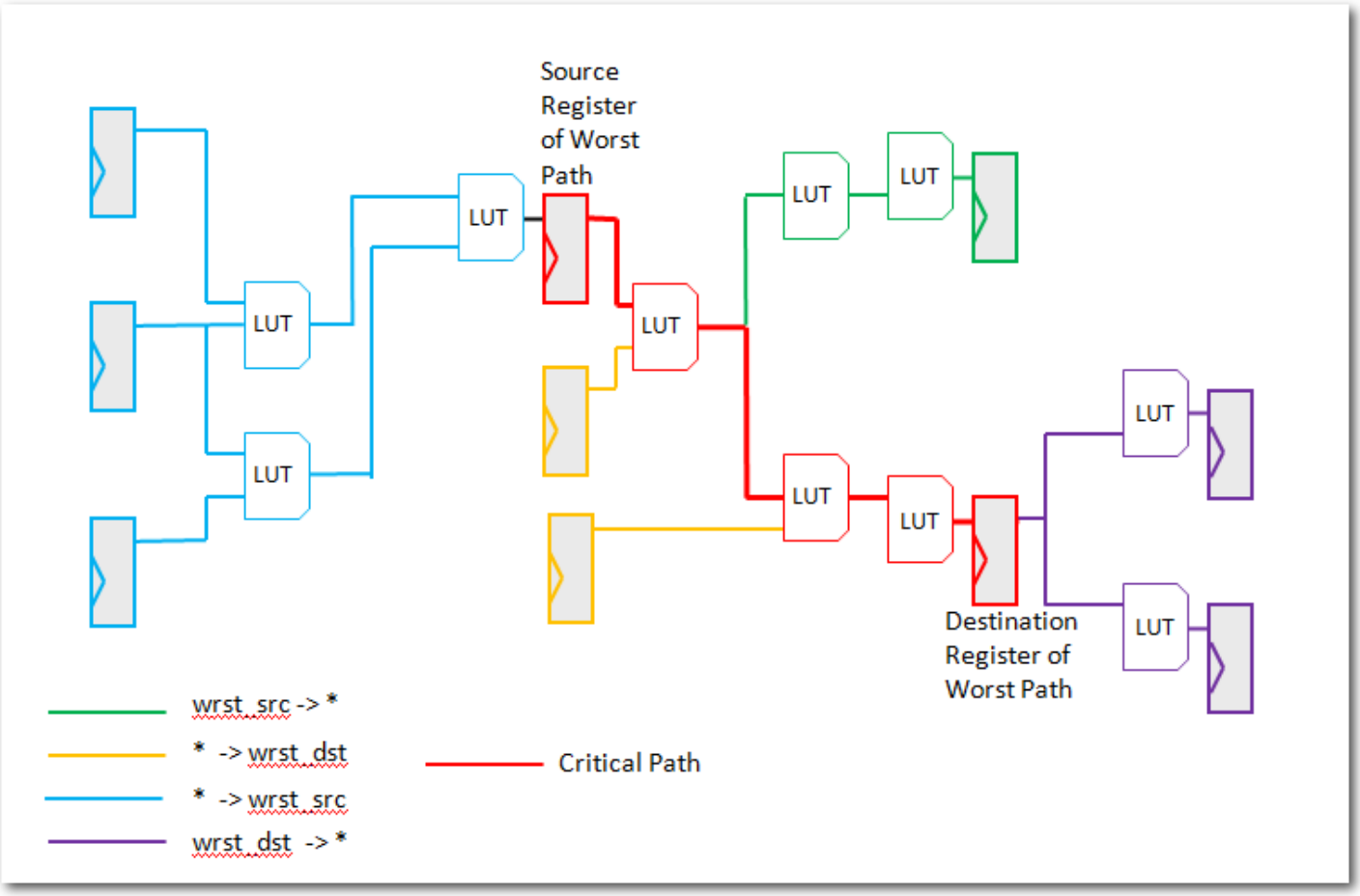
Sometimes I like to analyze what else is pulling all the nodes in my critical path, which can be achieved with the following Tcl commands, which can be put into a .tcl file in the project's directory. Only the names in red must be changed:

*set wrst_src {insert_source_of_worst_path_here}*
*set wrst_dst {insert_destination_of_worst_path_here}*
*report_timing -setup -npaths 50 -detail path_only -from $wrst_src -panel_name "Worst Path||wrst_src -> *"*
*report_timing -setup -npaths 50 -detail path_only -to $wrst_dst -panel_name "Worst Path||* -> wrst_dst"*
*report_timing -setup -npaths 50 -detail path_only -to $wrst_src -panel_name "Worst Path||* -> wrst_src"*
*report_timing -setup -npaths 50 -detail path_only -from $wrst_dst -panel_name "Worst Path||wrst_dst -> *"*

I would copy the node name from the columns "From Node" and "To Node" of my worst path into the first two variables, and then source the .tcl script from TimeQuest's Scripts pull-down menu:




Here is a simplified example of what these reports analyze:




The critical path of the design is in red and has already been analyzed between the worst source and destination registers. The script's first report_timing analyzes this path plus every other path that the source is driving, shown in green. The second report_timing analyzes the critical path plus every other path going to the destination, shown in yellow. In essence, they show everything inside these two endpoints that are pulling them in different directions. The last two report_timing commands show everything outside of the endpoints pulling them in other directions. If any of these reports have slacks near that of the critical path, then there is a good chance the fitter is balancing these paths with the critical one, trying to achieve the best slack. This diagram is quite simple compared to the critical path in most designs, but it's easy to see how this can get very complicated very quickly.

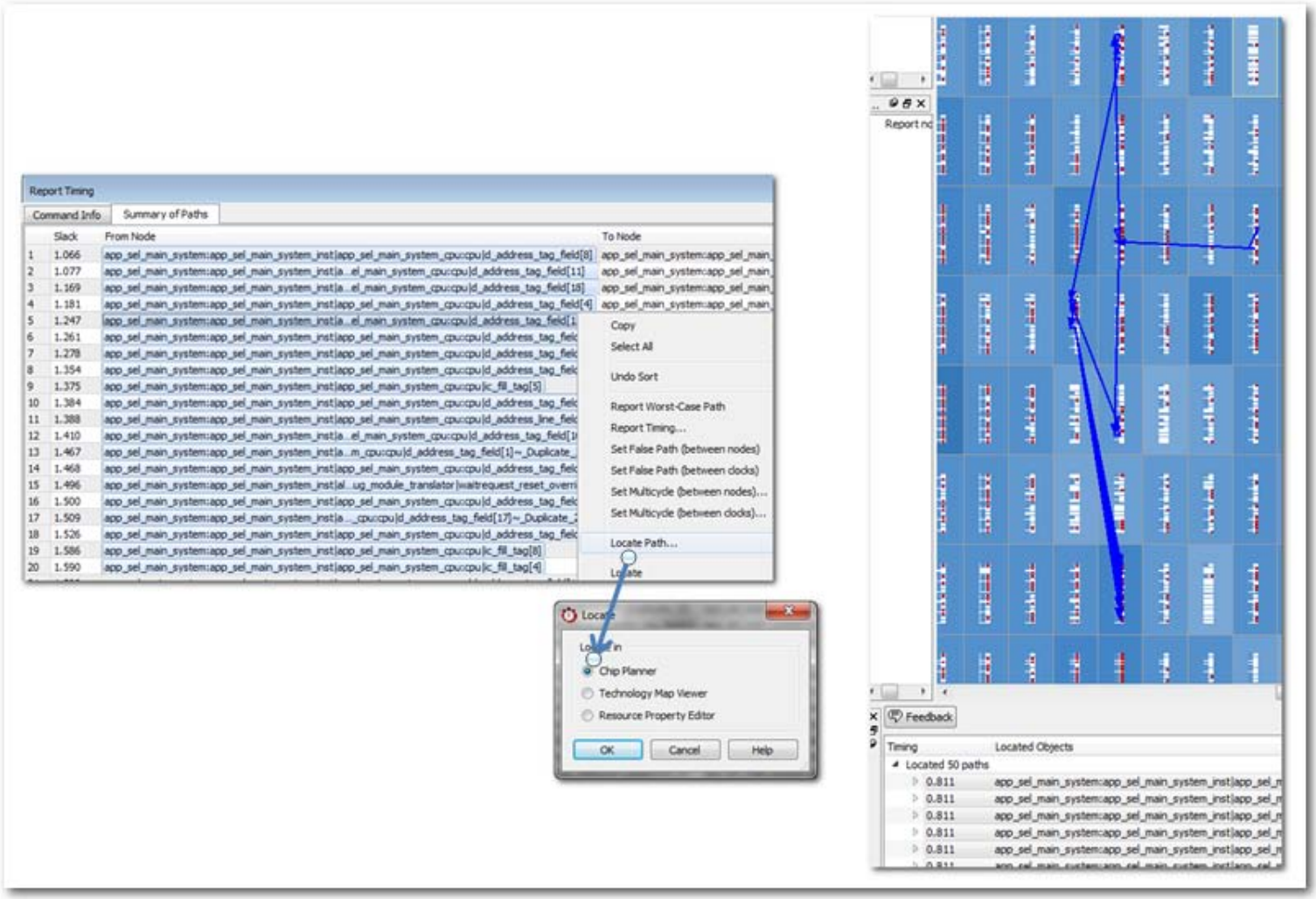
Note that you don't need a script to do this, as it is quite easy to right-click on a path and do Report Timing. For the first command, just delete the To: option, so that all paths from the source are reported. Repeat, but then delete the From: option instead. For the last two commands, you just cut the From: option and paste it into the To: option, and then do the reverse. Be sure to give the reports distinct names.

These timing reports are very useful for analyzing what is competing with the critical path, but not always good for examining how they might pull in different directions.

### Tip – Locate multiple paths to the Chip Planner

Within a timing report, multiple paths can be shown in the Chip Planner at the same time.

1) Run report_timing to show multiple paths. (See above script)
2) Select multiple rows of timing report. (Only a single column needs to be selected. In the picture below, I selected 50 rows from the From Node column.)
3) Right-click -> Locate Path -> Chip Planner
4) The selected paths will now show up in the Locate History window of Chip Planner, with the worst one shown in the Chip Planner.
5) Double-click the "Locate 50 Paths" to show all 50 paths at once, or select individual paths to view them in the Chip Planner.




So what does a user do with this information? It depends. In many cases, I've not seen an obvious next step to improving timing, but I do develop a much better sense of what else is pulling on my critical path. In some cases unexpected requirements jump out quite clearly. Examples include connections to other hierarchies/critical paths the user did not expect, or perhaps something physical

requiring logic to be spread out, such as logic feeding many embedded memory blocks that are spread out in the die.

### Tip – Create a .tcl script to monitor critical paths across compiles

Many designs have the same critical paths show up after each compile, but some suffer from having critical paths bounce around between different hierarchies, changing with each compile. There are many reasons this occurs. Designs that run at very high-speeds must have few levels of logic on all paths, and hence none of them have very much slack. A sub-optimal placement on any one of them can cause them to miss timing when they were making it before. I have also seen paths that have lots of long connections between various hierarchies have problems, since timing closure is dependent on many long hops all competing for better placement and faster routes. It may not be consistent who gets these better resources with each compile, making it seem like the critical path is bouncing around.

In designs like this, I might create a file called "TQ_critical_paths.tcl" in the project directory. For a given compile, I will look at the critical paths and try to write a generic report_timing command to capture those paths. For example, if a lot of paths fail in a low-level hierarchy I might add:

*report_timing –setup –npaths 50 –detail path_only –to "main_system: main_system_inst|app_cpu:cpu|*" –panel_name "Critical Paths||s: * -> app_cpu"*

If there is a specific path, such as a bit of a state-machine going to a bunch of registers called *count_sync*, I might add something like so:

*report_timing –setup –npaths 50 –detail path_only –from "main_system: main_system_inst|egress_count_sm:egress_inst|update" –to "*count_sync*" –panel_name "Critical Paths||s: egress_sm|update -> count_sync"*

The goal is to write something with a relatively broad stroke that captures a group of similar failing paths. With complex designs, I might end up with eight or more different groups I'm monitoring. The benefit of this is that I can source this file in TimeQuest after every compile, and I can add new report_timing commands as new critical paths pop up. This helps give a sense of what consistently fails and should have the highest priority, while also watching what is marginal. Again, this is really only necessary when a design's critical paths seem to change compile to compile.

## Incremental Compilation and LogicLock – Brief Overview

Incremental Compilation, at its most basic, allows the user to put a hierarchy of their design into a partition. Just the act of creating a partition means the logic will be synthesized independently of logic outside of the region. Parameters will still be passed in to make sure the synthesis is correct, but there will be no logic reduction across boundaries. This allows a partition to be put into Post-Synthesis mode, whereby Quartus II does not have resynthesize the logic if there were no changes to the source code. The major benefit of Post-Synthesis is to save synthesis time. The user can also put a partition into Post-Fit, in which case the logic will not only re-use the previous synthesis results, but also its previous place

and route information. This can be done independently of using LogicLock regions, although more often than not they are used together. One last important setting for partitions is Empty, which will remove the logic inside without having upstream/downstream logic getting synthesized out. We will study this in more detail in just a moment.

LogicLock is basically floorplanning. It allows the user to draw rectangles onto the floorplan and have logic remain in that rectangle. (LogicLock does allow non-rectangular shapes, but I find this seldom necessary and will not delve into that in this guide.) The Quartus II handbook has a good chapter about Incremental Compilation and LogicLock that I recommend reading first, called "Best Practices for Incremental Compilation Partitions and Floorplan Assignments" found in the Quartus II Handbook Volume 1:

http://www.altera.com/literature/hb/qts/qts_qii5v1.pdf

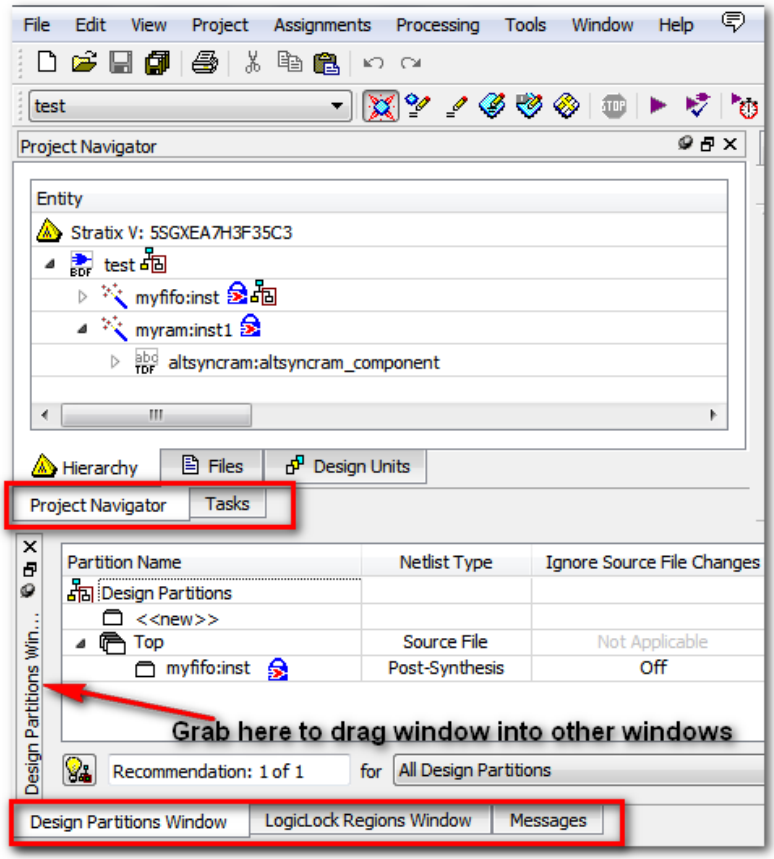
The Best Practices chapter gives many examples of how partitions prevent synthesis across boundaries and how to get around these limitations. This is good to be aware of when writing new code, but quite often partitions are put onto code that has already been written. In those cases, the designer is seldom aware of how preventing synthesis across boundaries will affect their design.

### Tip – Monitor area and slack differences when adding partitions

When adding partitions, I recommend keeping track of the Logic Utilization and timing slack before and after the partitions are added. If the area goes up significantly or the design gets much slower, than it's worth examining if there is an optimization across partition boundaries that is no longer occurring. If the design doesn't change much, then adding the partition has not harmed the design. Note that partitions will generally always make the design a little bit larger, but if it's more than a percentage or so, then most likely logic that previously had been getting optimized out is now being kept.

### Tip – Optimize Screen Space for LogicLock and Design Partition Windows

The Design Partition window, LogicLock window and pretty much every other window can be dragged so that it's tabbed with other windows. For example, I usually drag my Tasks window behind the Project Navigator, and put the LogicLock and Design Partition windows behind the Messages window, like so:




A layout like this makes it much easier to keep the Design Partition Window and LogicLock Window accessible, rather than constantly opening and closing them through the Assignments pull-down menu.

## Basic Incremental Compilation Tips

There are a number of quick tricks that can be done with partitions. A lot of these use the often overlooked option of Empty, which will have empty contents inside the hierarchy, but the partition boundaries will still be preserved so that nothing else gets synthesized out.

### Tip: Quick Block Stitch with Partitions

Early in the design phase, a user may know they will need to stitch together various blocks that previously had not been connected(perhaps multiple FPGAs are being merged into one) along with other IP such as a DDR3 controllers and maybe a PCI Express block. The designer may want to do a place-and-route of all these blocks together to get a better estimate of logic utilization and timing, but since they have not written the logic that stitches them together, it may seem a daunting task.

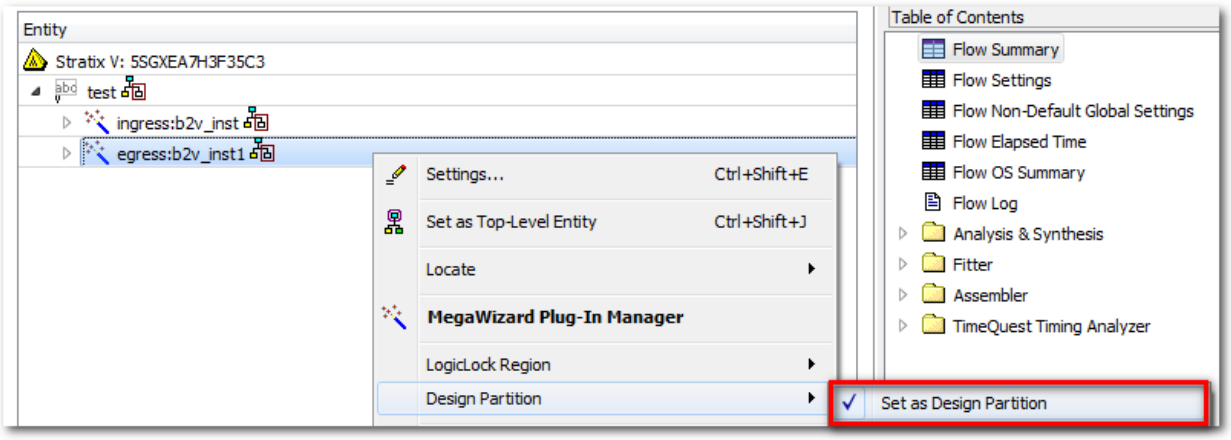
Note that Quartus II will synthesize out a partition if none of its outputs drive anything(either logic in another partition or a top-level I/O). But as long as a single output is connected, then the entire partition is kept. Knowing this, it is easy to create a top-level that only hooks up the minimum number

of connections, as well as the major control inputs, specifically clocks. For example, if the user had two blocks called ingress and outgress, then they might create a top-level file to stitch them together like so:

```
module top_stich (
        // Ingress
        input ingress_clk,
        ouput ingress_out,
        // Egress
        input egress_clk,
        output egress_out);
ingress ingress_inst (
        .sys_clk(ingress_clk),
        .rxp_data(),
        .rxp_parity(),
        .token(),
        .cal_err(),
        .burst(),
        .active(ingress_out),
        .rxp_calc());
egress egress_inst (
        .sys_clk(egress_clk),
        .carrier(),
        .prism_fold(),
        .thunder_data(),
        .thunder_address(),
        .thunder_write(),
        .thunder_read(),
        .thunder_r_data(),
        .thunder_busy(egress_out),
        .slip(),
        .rough());
endmodule
```

This module hooks up only two ports from each block to top-level I/O, an input clock and a single, randomly picked output. All other I/O are unconnected, but since each block will be put into a partition, none of the logic inside these hierarchies will be removed. There is flexibility with this method, as I could have driven both blocks with the same input clock, or if some connections between the blocks or to the top-level were known, I could add those in while leaving other ports unconnected. This method allows one to quickly instantiate many components without requiring any logic to stitch them together or any virtual pin assignments to make sure it fits without running out of I/O.

After creating the top-level verilog or VHDL file, the user would go to Processing -> Start -> Start Hierarchy Elaboration. This will add all the hierarchies to the Project Navigator. The user then right-clicks on each hierarchy and “Set as Design Partition”:




The partition type should be either Source or Post-Synthesis. This method is quite straightforward for general RTL. Where it gets more difficult is logic that uses hard logic in the FPGA. For example, PCI Express uses dedicated transceivers that MUST feed top-level I/O, and so for those cases the user must also bring these I/O to the top-level. For something like an Altera DDR3 interface(UniPHY), the user must also bring out the I/O that interface with the memory, as well as make the I/O standard assignments, as the <memory_name>_pin.tcl assignments dictate, which is created along with the UniPHY IP.

## Tip: Create a Black-Box for Incomplete or Invalid Logic

Another issue early in the design is that the user may be working on a new block that is incomplete. It is hooked up to other blocks in the hierarchy, but the actual contents may not exist, may cause synthesis errors, or may cause logic to be synthesized out. An easy fix is to create a Design Partition on that Hierarchy. If the contents are empty, the user only needs an HDL file that contains the ports and nothing else, like so:

```
module new_block (
        input clk,
input [31:0] din,
input [11:0] addr,
output [31:0] dout);
// Empty Contents
endmodule
```

or:

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY new_block IS
PORT(   clk : IN STD_LOGIC;
        din : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
        addr : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
        dout : OUT STD_LOGIC_VECTOR(31 DOWNTO 0));
END new_block;
```

```
ARCHITECTURE arch OF new_block IS
BEGIN
--Empty Contents
END arch;
```

The user should run Processing -> Start -> Start Hierarchy Elaboration to get the hierarchy, and then put a Design Partition on this hierarchy. The Partition should be set to type Empty. Note that the file does not necessarily have to have empty contents, but can't have any syntax errors in it.

### Tip: Put Non-Altera IP into Empty Partition

Another scenario I have used this for is when doing a quick analysis for converting a design from another technology (such as a Xilinx FPGA or an ASIC). Although most of the design may be basic RTL, there might be some very device-specific IP, such as a memory controller or transceiver interface, which Quartus II synthesis will not recognize. Rather than converting this IP, I may just comment out the contents of the top-file in that IP, turning it into a black-box, and put it into an Empty Partition, as just shown. This allows me to quickly analyze the size and timing of the rest of the design, and if it looks good, then worry about converting the IP.

### Tip: Create Empty Partitions to Save Space

There are multiple reasons for creating an Empty Design Partition on modules that have logic in them. Here are some of the ones I've done or heard of:

- I worked on a Stratix IV design with ~97% Logic Utilization. It fit but did not meet timing, and critical paths kept jumping around. We took a section of the design that easily met timing and created a Design Partition out of it and set it to Empty. The Logic Utilization dropped to 92% and the design began making timing. This helped show that the problem was not the logic being too slow, but that we were just putting too much into the device. We changed our timing closure methodology to look for logic that could be reduced/removed, rather than trying to make the critical paths run more quickly.
- If a design has an Empty Partition in it, Quartus will still write out a programming file for the FPGA. Obviously, the section of logic that is empty will no longer work, but this can still be useful. I saw one design that used 100% of the FPGA memory blocks. The designer took a section of logic that used a lot of internal memory and put it into an Empty Partition. They were then able to use this physical memory for their SignalTap II file. The major caveat is that, since a section of logic was now Empty, the portion of the design being tapped had to work independently of the section put into the Empty partition.
- Quick Compiles. It's easy to create an empty partition on logic outside of what you're interested in. This basically removes the logic from being synthesized, placed and routed, which can significantly improve compile times. When testing something specific in a design, such as seeing if a Maximum Fanout constraint in the assignment editor applies to a specific

node, I may turn a lot of the other major hierarchies into Empty Partitions. Once I have confirmed it works, I will delete the Partitions so everything is compiled like before.

- Similar to quick compiles, is the ability to isolate portions of a design to get the best performance, and then lock it down. This will be described in the section Incremental Compilation for Performance.

### Tip: Put Hierarchy Being Modified into a Partition for Quick Compiles

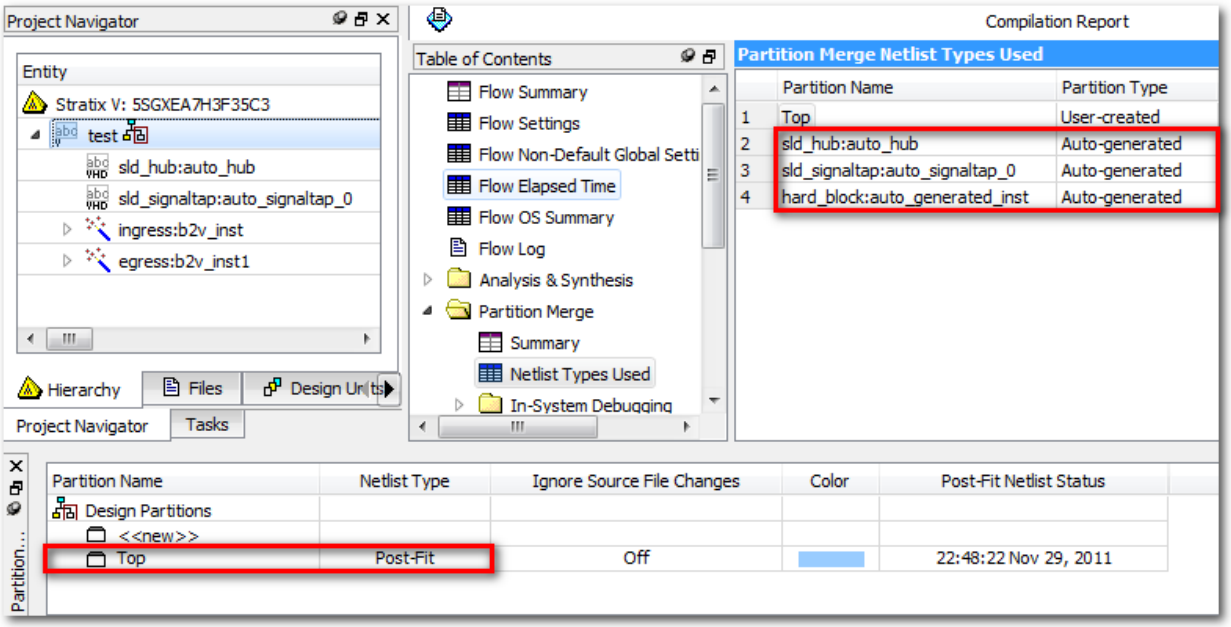
If there is a small section of code the user knows they are going to be making changes on, they may want to make a partition just on that block. They can set the Top partition to Post-Fit, and hence only this small block will be re-synthesized, placed and routed. This can make for quick compiles as the user tries different HDL changes. For example, there might be a state-machine that is not working correctly, and putting that hierarchy into a partition while the top is locked down allows quick iterations for the user to try different code.

The user might want to also put the hierarchy into a LogicLock region with the Reserved option, which prevents other logic from the top-level from being placed inside the region. This basically preserves a section of the die for the fitter to work with on subsequent re-fits of that hierarchy. This is probably only necessary if the hierarchy is pretty large or timing critical.

Once that section of code has been debugged, the user can simply delete the partition and LogicLock region for it and set the top-level back to Post-Synthesis or Source. Note: The user does not have to delete the Partition or LogicLock region, it's just an option if they do not need it anymore.

### Tip: Set the Top level partition to Post-fit when using SignalTap

Whether the user knows it or not, by default their design has one partition in it called Top, which contains all the logic in their design. SignalTap logic is actually created in other partitions. The following screenshot is from a simple design called test, which is using a SignalTap file:



The bottom Design Partitions Window shows that only the default Top partition exists and I have not added any others. Yet the Partition Merge Report shows three extra Auto-generated Partitions exist, which contain the SignalTap II logic. Now I can set the partition Top to Post-Fit, so that it is locked down, and keep modifying the SignalTap II file, whether it be tapping different points in the design, changing buffer sizes, modifying trigger conditions, etc. This has a number of advantages:

- SignalTap II compile times will be faster.
- The design won't change. This can be an extremely important when debugging a sporadic issue that comes and goes from compile to compile, allowing the user to work with a consistently failing build.

The only major down-side to post-fit is that the user must select nodes from the filter SignalTap II: Post Fitting. Only combinatorial logic that has been synthesized to be the output of an LUT will be accessible, and most combinatorial node names will be pretty difficult to understand. Whenever possible, the recommendation is to try and tap registers, since they have the most stable and understandable names.

### Tip: Partition I/O Interfaces

One design methodology is to put all the major I/O interfaces into partitions, specifically interfaces that have a lot of logic and/or that have trouble meeting timing. Examples include memory interfaces such as UniPHY(DDR3, QDRII+, RLDRAM, etc.) and transceiver interfaces(PCIe, XAUI, SRIO, Interlaken, etc.). Interfaces tend to have two major advantages that make this easy. First, their code is often complete early on. Once the user has determined the interface is working, they may never modify the code and hence a Post-Synthesis partitions means it will never have to be synthesized again, and a Post-Fit setting means it will never have to be fit again. Second, these interfaces are placed on the edges, so locking down their placement usually does not cause any problems with other logic in the design. For example, if a large block of logic were put in the middle of the device and set to Post-Fit, this would most likely impact other logic that needs to be placed around it and route through it. But I/O interfaces can be relegated to the I/O edges so they do not disturb internal logic.

When putting I/O interfaces into partitions, the user must decide if they want to floorplan. This is discussed in more detail in the LogicLock section.

Personally, I don't use this option very often. I find it doesn't save a significant enough amount of compile time, and once most I/O interfaces meet timing, they continually do. That being said, designers tend to like this flow if for no other reason than it locks down a significant portion of the design, giving the same results compile after compile, and therefore something the user does not have to worry about.

## Incremental Compilation for Isolating Hierarchies

When it comes to timing closure, it's safe to say that a design won't meet timing if the hierarchies within it don't meet timing by themselves. Because of this, it is often easiest and quickest to optimize a sub-hierarchy of the design and try to get the best performance out of it, before working on

the project as a whole. The conventional method for doing this is to build a custom Quartus II project with this hierarchy as the top level, make Virtual Pin assignments so the device doesn't run out of I/O, and create a custom .sdc to match the new hierarchy names. There are often other difficulties, such as clocks coming from PLLs or transceivers outside of the block, and the user must find a way to accommodate these issues.

Empty Partitions allow two methods that are much quicker and easier for analyzing a sub-hierarchy. Before covering those two methods, I want to emphasize a design suggestion. When designers are writing new code, they are primarily concerned with getting the correct functionality, and secondarily closing timing. That's the correct prioritization, but when it comes to closing timing, designers often just aim for positive slack and then consider it done. For example, if they have a 200Mhz requirement and find a compile comes in at 205MHz, their block is officially done and they move on to integration. Yet I think we all know that when a block is merged with other blocks, the performance tends to drop, sometimes a tiny amount and sometimes a substantial amount. There are many causes for this, such as competition for placement or routing wires, new critical connections that pull apart hierarchies or paths, or just a slightly worse fit because the fitter's solution space has grown so much when everything is put together.

### Tip: Get as Much Margin as Possible When Designing Blocks

The best time to improve performance on a block is when the user is originally designing it. They have a good understanding of how the code works, and hence changes are easy to make. The code usually has not been integrated into the system, so changing the latency or something like that is often much more acceptable. Once a hierarchy has been integrated into the full design, the test benches have been written and interfaces have been tested, so it is much more difficult to make changes to help timing, just because they can affect so much more.

I strongly recommend trying to optimize each block to run as fast as it can by itself. The major downside is that this takes time, but working on an individual block is often much faster than optimizing the full design at the end of the project. If it's possible to take the code that barely meets the 200MHz requirement and optimize it to run at 250MHz, there can be large benefits:

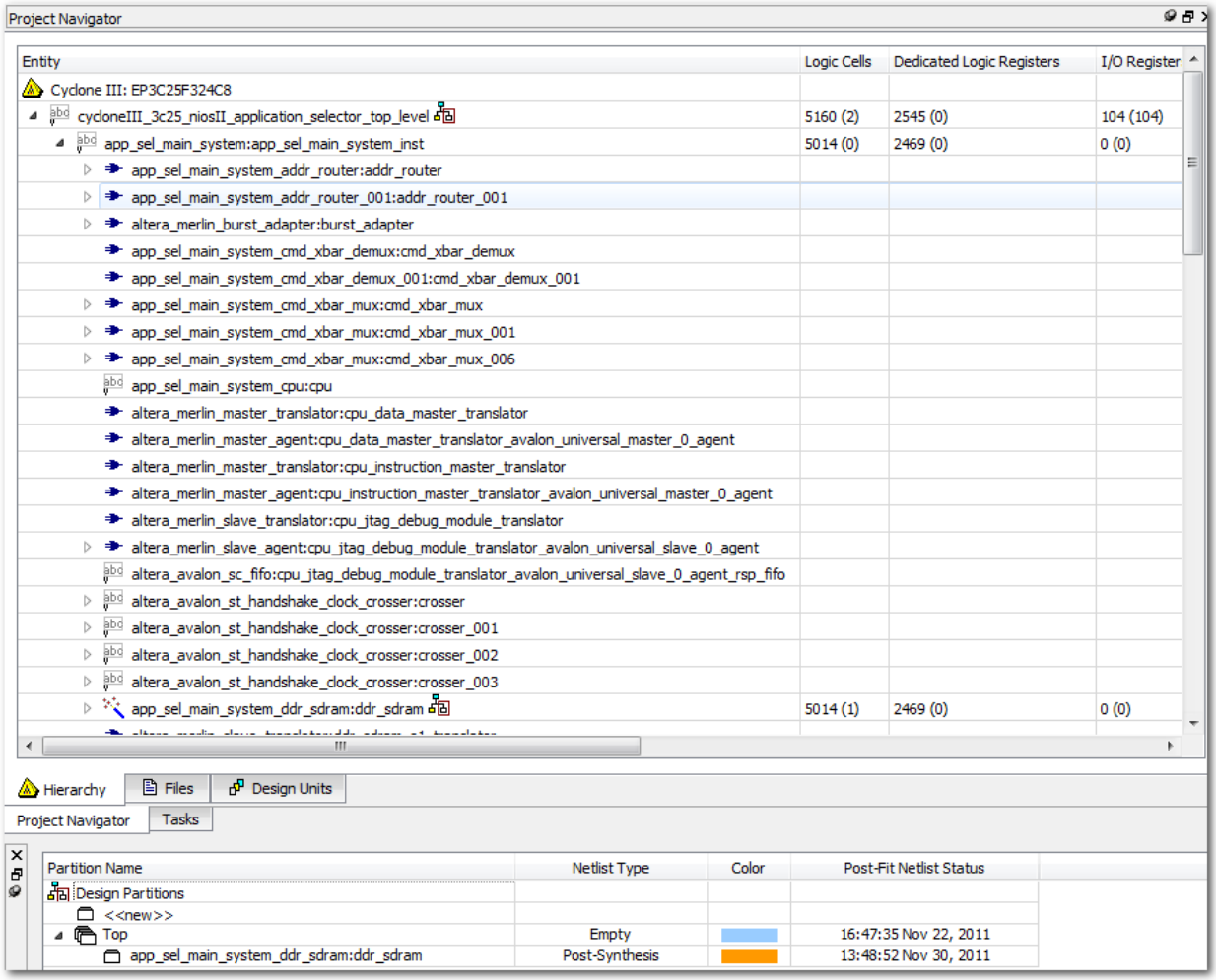
- Timing Closure - Integration into the rest of the design is much more likely to close timing right away.
- Compile Times - Compile times may be faster. This may just because the fitter, when set to Auto mode, can close timing more quickly, but quite often, it's because the user does not have to enable algorithms that hurt compile time, such as Physical Synthesis or the Placement Effort Multiplier.
- Density - The user can often pack more logic into a device, and even if the timing degrades a little, it will still close timing.
- Power - The designer can enable more options for saving power, even if they are at the expense of performance. (See Tools -> Advisors -> Power Optimization Advisor)
- Cost - The designer may be able to target a slower speed grade device and save money.

- Future Proofing - When the next generation comes around, and the specs inevitably require the design to run faster, that block is ready to go.

The bottom line is that the more margin you can get on each sub-block in a design, the better off you will be when integrating it all together. With that in mind, here are two methods for analyzing and optimizing just a portion of the design:

## Tip: Hierarchy Isolation Method 1 - Set Partition Top to Empty

An oft overlooked strategy is to set the Top partition to Empty, while creating a lower-level partition on the hierarchy or hierarchies the user wants to analyze. Here's a screen-shot of a QSYS system where I want to isolate the SDRAM interface:




In the Design Partitions Window at the bottom, the top-level is set to Empty while the ddr_sdram is set to Post-Synthesis(or Source). Looking at the Logic Cells and Registers across the hierarchies, most of them are empty except for the ddr_sdram. This flow is easy to do, preserves the hierarchy so .qsf/.sdc assignments still match, and allows for quick compiles. One interesting note is that, even if a partition is set to empty, Quartus II will try to find clocks in it and preserve them. So, for

example, if there were a PLL inside the empty partition Top that creates clocks for the partition I am analyzing, that PLL should still be instantiated. This keeps the physical clock structure intact, which makes timing analysis more accurate. It also allows the .sdc's derive_pll_clocks or create_generated_clock commands to match the PLL in the design, and hence have proper names and clock relationships.

Once the design has been compiled, analyzed, and the user feels good about it, they will set the top-level partition to something other than Empty, and they have the option of deleting the low-level partition so that it is now synthesized with the rest of the design, or they can keep the partition if they think they might revisit it. Of course they can set the partition to Post-Fit, which will lock down the placement and preserve performance, but we will discuss that more in the next section, Incremental Compilation for Performance.

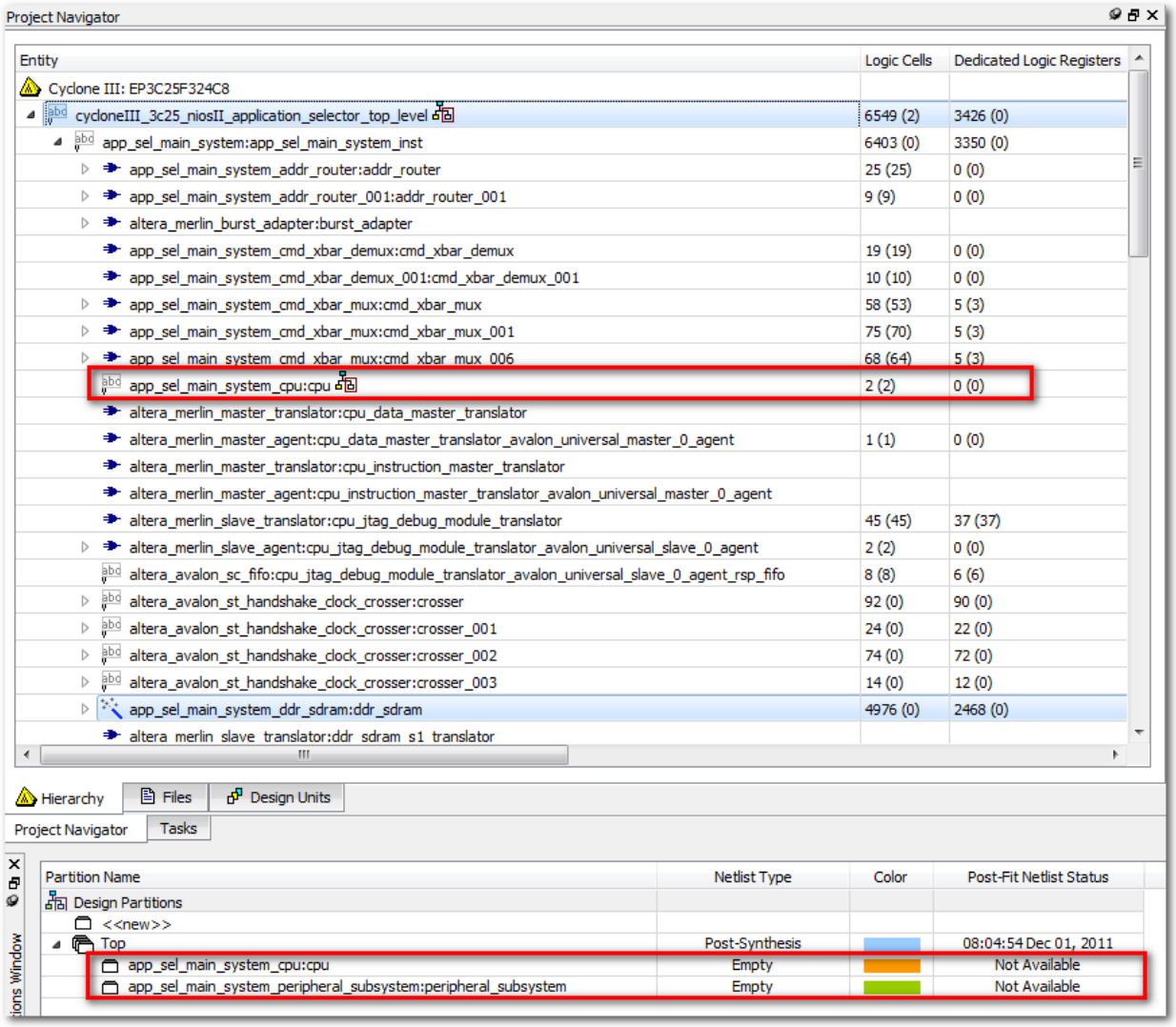
### Tip: Hierarchy Isolation Method 2 - Set Adjacent Partitions to Empty

This is very similar to the previous method, except rather than setting the top-level to Empty, the user will find some of the larger sister hierarchies and put them into partitions and set them to Empty. I have used this method the most, since there was a good period of time where setting the Top partition to Empty, which is used in Method 1, was not allowed in Quartus II.

There are a few other reasons to use this method:

- Sometimes setting the Top partition to Empty results in fitter errors. Hopefully these have been resolved, but this is a somewhat new, seldom used feature, and occasionally results in fitter errors, especially when dealing with hard IP like transceivers and memory interfaces. For example, let's say the user isolated a memory interface whose PLL, DLL and OCT blocks are slaves driven from a master memory interface. I haven't tested this scenario, but there's a good chance that if the master memory interface is in an empty partition, the design will not fit, most likely due to missing OCT block. This second method would allow the user to keep the master memory controller too.
- Just a different approach for determining what is synthesized, placed and routed. Method 1 set the Top partition to empty, thereby removing the whole design, and the user adds partitions to lower levels they want to add back in. This method starts with everything in the design existing and the user adds partitions to explicitly remove large chunks.
- By keeping the top-level and other hierarchies in the design, the fitter is now aware of them. If another hierarchy is locked down in the floorplan(either with LogicLock or a Post-Fit partition), the user would not put that logic into an Empty partition, allowing the fitter to be aware of it and optimize any connecting paths.

Here is a screen-shot of the same design shown for the previous method, but the Top partition is set to Post-Synthesis and two major components, the Nios CPU and a QSYS peripheral subsystem were put into partitions and set to Empty:




As can be seen in the Project Navigator, the cpu hierarchy only uses 2 LUTs, where it normally uses thousands. This layout has allowed me to isolate the ddr_sdram along with some other components. I have run designs with more than thirty hierarchies put into Empty partitions to remove them and isolate the hierarchy I want. Once I have optimized this hierarchy, I then go in and delete all the Empty partitions so that everything can be synthesized, placed and routed together.

## Incremental Compilation for Performance

The previous section covered how to use Incremental Compilation to Isolate Hierarchies for analysis and optimizations. The idea is the user would have quick compiles while they optimize the design, and once done, they would then compile everything together in a flat place-and-route. Incremental Compilation for performance relies on the same methodology, except the user sets the isolated partition to Post-Fit once it meets timing, thereby preserving the place-and-route of the that logic and having other logic fit around it.

Interestingly, incremental compilation is not really designed to improve performance. As mentioned earlier, just the act of adding partitions will usually make designs a little bit larger and could possibly make them slower if critical paths cross partition boundaries. That being said, partitions can be used to improve performance in two major ways:

- Performance Preservation. As discussed in the document on the Quartus II fitter, all designs have variation:

  http://www.alterawiki.com/wiki/The_Quartus_II_Fitter_and_Seed_Sweeps

  If the critical portion of a design varies between passing timing and failing, it can be a huge benefit to be able to lock down a block when it happens to place-and-route at the high-end of its variance. For example, if a block runs off a 5ns clock, but on average that block only meets 4.7ns with a standard deviation of ~250ps, then very few compiles are ever going to meet timing. But if the user is able to run many compiles overnight or over a weekend and then lock down results once it meets timing, they will have basically removed that variation on all subsequent compiles.

- Isolation. As just discussed, being able to isolate a sub-hierarchy of the design allows for quicker compiles, and just as importantly, allows the fitter to concentrate just on that portion of the design, and often gives it the space it needs to get the best results. Also, by isolating a portion of the design for the fitter, one significantly reduces the size of the solution space, allowing the fitter to try more combinations and will hence be more likely to get a better result.

Both of these methods require a Design Partition on the hierarchy the user wants to optimize, and leverage the two methods just discussed for design isolation, either setting the Top partition to Empty, or setting adjacent partitions to Empty. The major difference is that when good results are achieved, the user will re-use those results in future compiles by setting the critical partition to Post-Fit. This will direct the fitter to re-use the synthesis and place-and-route information from the previous compile.

It is possible to set the partition to Placement Only(right-click on Partition and go to Properties), which will only re-use the placement information but not the route information. This option can be useful since locking down the routing of a block of logic can make it more difficult for new logic to route from/to/through the locked down region. Setting a partition's back-annotate level to Placement Only will allow some variation on future compiles, but the router variation is significantly less than the placer variation. I don't recommend this to start with, but if it seems like the fitter is having trouble routing other logic, this is a possiblitiy.

### Tip: When Preserving Performance of a Partition, Create a .qxp

Although partition information is stored in the directory /incremental_db, a useful option is to export the partition as a .qxp file(Quartus eXport Partition). This stores the synthesis, place and route information in a single file and can be added to a project just like a source file, under Project -> Add

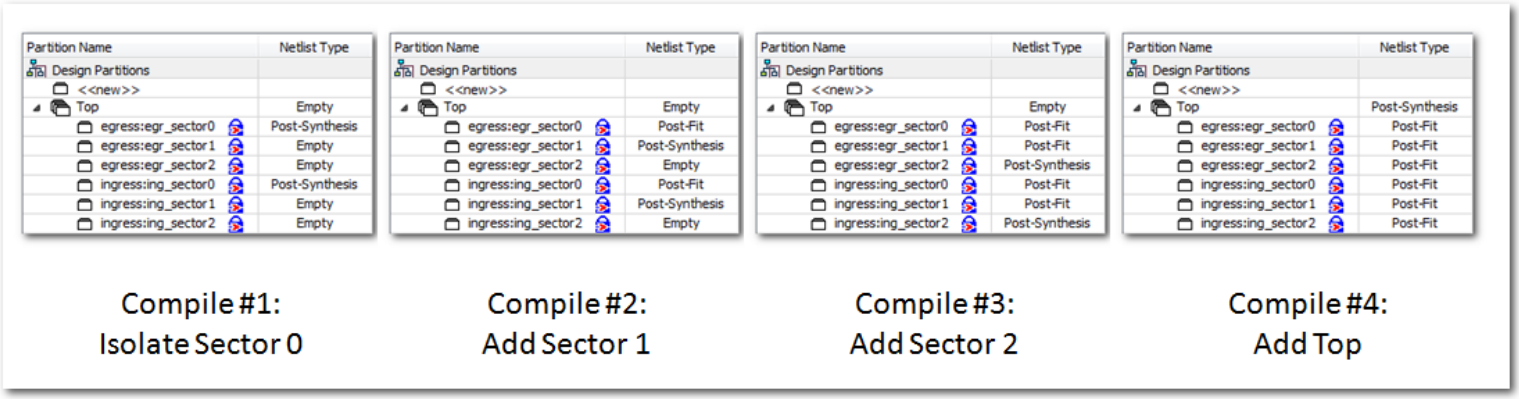
Files. The user will want to remove the HDL that represented the original hierarchy to avoid synthesis errors, as otherwise there will be two representations of the same module, one in HDL and one in the .qxp, and Quartus II will not know which one is correct. The user should still put the hierarchy represented by the .qxp in a Partition, which allows them to manually select how much of the .qxp information they want to use, e.g. they could just use the synthesis information, the placement information or the whole place-and-route.

Creating a .qxp adds a few steps to the user's flow, so it is not recommended if the user is constantly making changes to the partition and trying to preserve them. The .qxp flow works best when the code is finalized, or at least changing very seldom, and the user wants to store the partition information in a single file that can easily be archived.

### Tip: Incremental Compilation for Performance Method 1: Building Up a Design

This method consists of starting with a critical block, isolating it, and closing timing. Once that has been achieved, the user sets the partition to Post-Fit, and then adds in another critical partition and begins working on closing timing. (If there is only one truly critical block, then the user would just move to compiling the rest of the design). This iterative approach can be done as often as the user needs. I have seen DSP designs iterate over more than twenty instances as they work toward timing closure.

There are several variations based on the previous examples of isolating a design. The user can isolate by setting the partition Top to Empty like in the upcoming example, or by creating adjacent partitions and setting them to Empty.




In this example, Compile #1 isolates the egress and ingress hierarchies in Sector 0. Users generally only compile one partition at a time but it is not a requirement. In this case, the ingress and egress of Sector 0 may have critical paths between them, and so the user wants to make sure the fitter is able to fit them together. Also note that two partitions can be merged into a single partition by selecting them both and doing a right-click -> Advanced -> Merge, although I did not do that here. Once these two hierarchies in Sector 0 meet timing, I set them to Post-Fit for Compile #2 and set Sector 1 blocks to Post-Synthesis so that they can be placed-and-routed. If they were not previously synthesized, they will be synthesized too. Once Sector 1 is locked down, it will be set to Post-Fit and the next compile

will work on Sector 3. Finally, Compile #4 will use the post-fit information of all six partitions and fit the rest of the logic around them.

One benefit of this flow is that all the additional compiles will be aware of the locked down partitions. If there are any paths between them, the fitter can optimize placement to try and meet that timing. Note that the partitions have the LogicLock symbol by them, showing that the logic is also in a LogicLock region. This is not a requirement with this flow, but is often recommended, and discussed shortly.

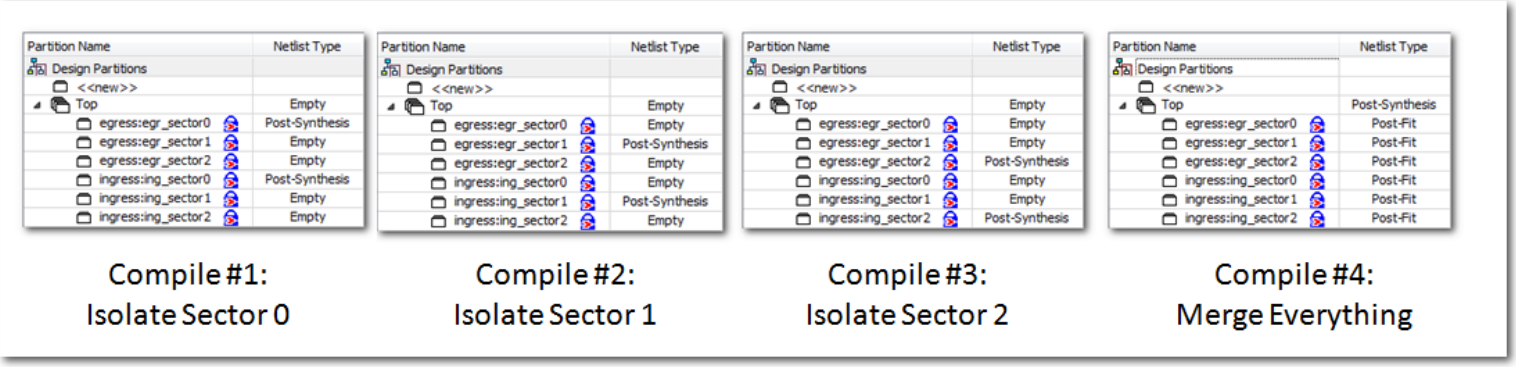
If the design is not floorplanned, it is much more difficult to go back and compile one of the earlier sections. If the user knows they will be making changes to the timing critical portions, this flow can become quite cumbersome to repeat. If the critical partitions have been verified and are unlikely to change, then most changes would only require the last compile or two to be repeated, and this flow can be very useful for achieving performance and having quick compiles

Between this method and the next one, this is by far the more commonly used approach.

### Tip: Incremental Compilation for Performance Method 2: Isolating Multiple Partitions

This method takes advantage of a little known feature of Empty partitions. Basically, if a partition has been compiled once so the /incremental_db has synthesis, placement and routing information, setting it to Empty and re-compiling will not remove the previous compilation information on that partition, it just doesn't use it. A user could isolate a partition, close timing, and then set that partition to Empty while they go work on another partition. Then, when the user wants, they just set that partition to Post-Fit, and the synthesis, place and route information is used from when it had previously met timing.

Taking advantage of this behavior, the user is able to isolate a hierarchy using the previously described method 1 or method 2, close timing, and then set that partition to Empty so they can go and isolate another partition and closing timing on that. They can repeat this as many times as necessary. Then, once they have each critical partition meeting timing in isolation, the user sets them all to Post-Fit, sets the Top to Source or Post-Synthesis, and compiles everything together. Each partition that was compiled in isolation will use that compilation's synthesis, placement and routing information, while the rest of the design fits around it. The benefit is that multiple partitions are truly isolated for their compiles, resulting in fast compiles and extreme focus from the fitter.




The above example shows the ingress and egress for Sector 0 placed-and-routed first. Most likely they are placed in the same LogicLock region. After that, they are set back to Empty and Sector 1 is compiled. This flow is repeated for Sector 2. On the last compile, all of the sector partitions are set to Post-Fit so the partition information from Compiles #1, #2 and #3 will be used for Compile #4, and the rest of the design will fit around these locked down partitions.

Notice that the partitions are in LogicLock regions. This is a must for this flow, since each individual compile is unaware of where the logic from another compile was placed, so without LogicLock regions there is a decent chance they will overlap locations and Compile #4 will fail. Another drawback to this method is that any paths between the different sectors will not be optimized, since they were placed without any knowledge of the other sector locations. Ideally there are few connections between these blocks and they can easily meet timing, so they still meet timing with sub-optimal placement.

### Tip: LogicLock when Preserving Performance with Incremental Compilation?

An important question when using Incremental Compilation for Performance is if the user should floorplan their design with LogicLock. We will discuss floorplanning in more detail in the next section, but will briefly touch on examples here. LogicLock is not always required, but when a hierarchy is isolated, it has the whole floorplan to work with, which may be too much freedom considering the rest of the design still needs to be fit. Here are some cases on when to use LogicLock when preserving performance:

- A very full design. If the design is very full, say over 90% Logic Utilization, then isolating a portion of the design might allow the fitter to spread it out too much. This might be good for timing on that hierarchy, but it might use too many resources so that the rest of the design will need to be packed even more tightly, making those results worse. In this case, a compact LogicLock region on the partition will direct the fitter to pack it more tightly.
- A general location. When a hierarchy is isolated, the fitter can put it just about anywhere, which may not be optimal when the rest of the design is fit into it. In this case, a LogicLock region may direct the fitter to a better general location. The LogicLock region does not have to be tight, as the user is just trying to maintain a general location. This location may be because the user thinks they will have better performance at that location, because they want to avoid having it placed where they know other logic should go, or they just want consistent placement over multiple compiles. One option is to see where it was placed on a

previous compile, by locating to the Chip Planner, and making a LogicLock region that overlaps that area.

- Method #2 absolutely requires LogicLock regions, since multiple partitions are isolated and hence fit without any knowledge of where the other partitions are being fit. LogicLock ensures they are not placed in overlapping locations, which would cause a fit failure once the partitions are all brought together as Post-Fit partitions.

In summary, if I am just isolating a design to analyze how fast it runs, I usually do not LogicLock it. The only caveat is if the design is very full and I want to try and mimic the constraints of having to pack the logic together very tightly. If I am isolating partitions for better performance which will be preserved in the full design, I generally recommend floorplanning, even if it's just a general location.

## Floorplanning with LogicLock Regions

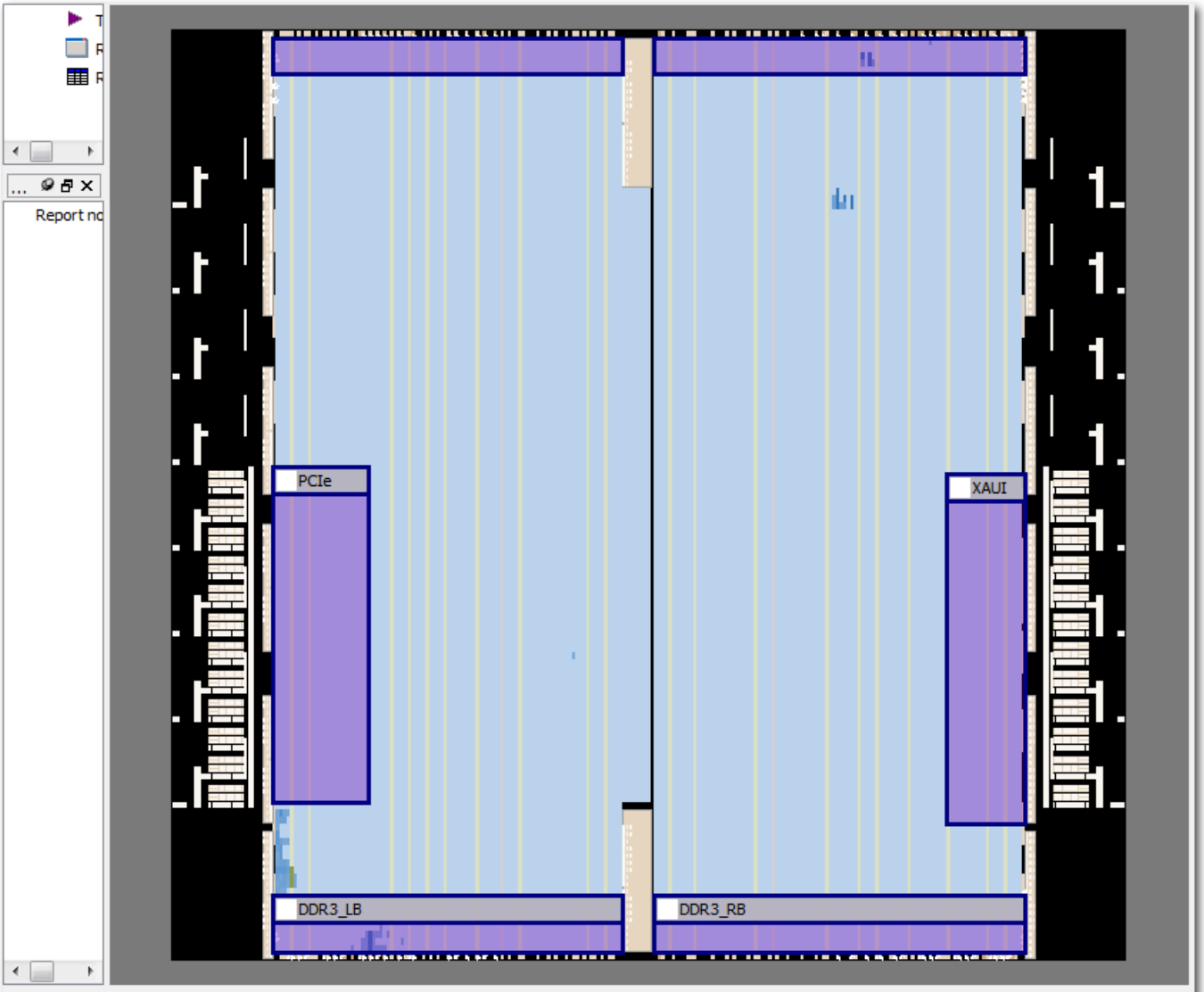
Technically, floorplanning is quite simple. The user draws rectangles (or other shapes) onto the Chip Planner and tells logic to stay within those boundaries. Yet with that simple explanation, floorplanning can be one of the more complex methodologies for closing timing. Let's look at a few reasons why:

- The FPGA resources of logic, memory blocks and DSP are spread somewhat uniformly throughout the device, yet design hierarchies may non-uniformly use these resources, making it difficult to draw rectangles that match the user's requirements. (Right-click on a LogicLock region and select Properties, select a Design Element and then Edit, and the user can exclude different types of resources, allowing for some creativity, such as having a small LogicLock region for a hierarchy's logic and a large overlapping LogicLock region for the many memory blocks it feeds.)
- Full designs are especially difficult. If a design's Logic Utilization is at 93%, and the user makes a LogicLock region that is under-utilized, say 85%, then other regions will have to be over-utilized. It is very difficult to keep each region's utilization at 93%. This is only true if the entire design is floorplanned though. If some blocks are not in LogicLock regions, the fitter can place them where there is open space.
- Designs change. A good floorplan that helps the user may suddenly become invalid as some of the logic outgrows the region it is in. The user must then expand that LogicLock region while shrinking others and try to maintain the benefits LogicLock was providing. Users can LogicLock too early, spending a lot of time developing floorplans that constantly change.
- LogicLock spreads logic out. Users tend to think of what it's grouping together, but it is also spreading out logic between different regions. It is easy for paths between LogicLock regions, which previously met timing, to start failing, because the user forces a long hop on paths between regions.

That being said, designers will floorplan for different reasons, and in some conditions the flow can be quite straightforward. These cases do not require that everything in the design be floorplanned, just some specific sub-blocks. A few examples:

### Tip: Floorplanning I/O Interfaces

This strategy is employed when using partitions to lock down I/O interfaces. From designs I have looked at, the Quartus II fitter tends to place interface logic pretty close to the I/O ports it hooks up to, and so a LogicLock region would not do a whole lot. One way to check this is to locate the hierarchies to the Chip Planner. If the layout is not what the user wants, then LogicLock regions may make sense. Another reason to use LogicLock regions is for a very full design, in which the user may compact the I/O interfaces as tight as possible while still making timing. Here's a quickly done example floorplan(it's not a real design so I don't know if the dimensions are correct for the logic):



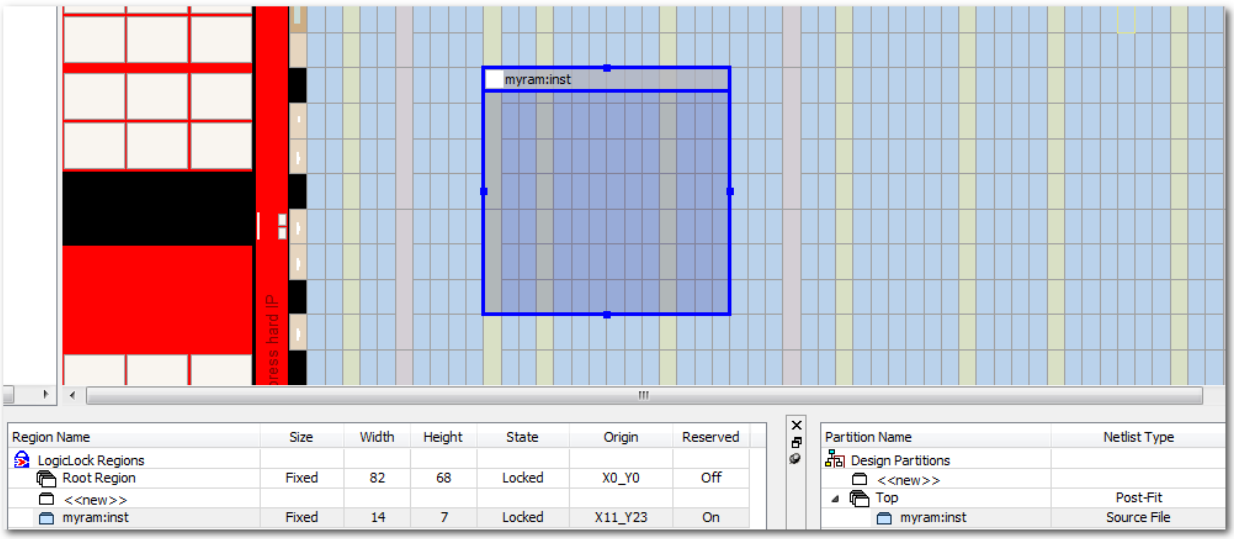

Regions Window

| Region Name | Size | Width | Height | State | Origin | Partial Reconfiguration |
|---|---|---|---|---|---|---|
| LogicLock Regions | | | | | | |
| Root Region | Fixed | 211 | 130 | Locked | X0_Y0 | No |
| <<new>> | | | | | | |
| DDR3_LB | Fixed | 98 | 8 | Locked | X0_Y1 | No |
| DDR3_RB | Fixed | 104 | 8 | Locked | X107_Y1 | No |
| DDR3_TL | Fixed | 98 | 5 | Locked | X0_Y124 | No |
| QDRIIPlus | Fixed | 104 | 5 | Locked | X107_Y124 | No |
| PCIe | Fixed | 27 | 47 | Locked | X0_Y22 | No |
| XAUI | Fixed | 22 | 49 | Locked | X189_Y19 | No |

### Tip: Floorplan for Incremental Compilation on a Single Hierarchy

This scenario is for when the user has a specific hierarchy in the design they are working on, and hence they know that for a while, all code changes will be contained within that hierarchy. This strategy is used for quick compiles, since only the hierarchy in question will be synthesized and fit for each new compile. The user will put the hierarchy they're working on into a Design Partition and a LogicLock Region that has been marked Reserved, so that no other logic floats into the region. This gives a nice

open section of the device for this logic to be fit and re-fit into. The user sets the Top partition (and any other partitions besides the one being worked on) to Post-Fit. This will lock down all the logic except the portion being worked on. Now the user can quickly do iterations on this section of logic. If they are unsure where to put the floorplan, they can locate it to the Chip Planner on a previous fit and use that for guidance.




In the attached example, a single hierarchy was put into a Partition and a LogicLock Region, which was set to Reserved. The Top partition is set to Post-Fit. The user can now make source changes to that hierarchy and only it will be synthesized and fit, making for fast compiles.

## Floorplanning the Entire Design for Incremental Compilation

Using LogicLock to floorplan a single hierarchy or two can be pretty straightforward, but floorplanning many hierarchies can quickly become complicated, as previously mentioned. One major reason a designer would go down this path is to use Incremental Compilation on their full design. The concept is similar to the just mentioned option of Incremental Compilation on a single hierarchy, except future modifications could be anywhere in the design and hence nearly all hierarchies will be in a Design Partition and LogicLock region.

First, let's remember why floorplanning is generally required when putting most, if not all, of a design into Design Partitions to reduce compile times. This is discussed in the handbook section "*Why Create a Floorplan*" here:

http://www.altera.com/literature/hb/qts/qts_qii5v1.pdf

The problem, as can be seen, is that the majority of the device needs to be floorplanned, which is not a trivial task. If large parts of the design are not floorplanned, they will float into the areas the partitions are fit into. Then, when a partition is re-fit and everything else is locked down, the partition needs to be placed in between the holes of the locked down logic. We call this a swiss-cheese floorplan, and it is a significantly more difficult fitting problem that is almost impossible to do and maintain good

results. If the whole design is floorplanned, then each partition will have a nice open rectangle of space to fit into, which is what the fitter is good at doing. When floorplanning an entire device, I have some basic recommendations:

### Tip: Do not create too many Design Partitions or LogicLock Regions

As a quick number, I would aim for having approximately four partitions. Why? Most importantly, four LogicLock regions is usually pretty easy to do, so hopefully the designer doesn't spend too much time laying them out. Note that there is some overhead that can't be reduced by Incremental Compilation. Let's say that is ~25% of the original compile time. So if the user locks down 3/4ths of their design, their compile time would be 25% for what needs to be refit plus 25% overhead, or 50% of the original compile time. Now let's say they went through the trouble of making 8 partitions and LogicLock regions. If only re-fitting one of those partitions, their compile time would be 12.5% to refit that logic plus 25% overhead, or 37.5% of the original compile. This is not significantly better than the 50% savings, but make eight LogicLock regions can be significantly more difficult than four.

This is not a hard rule though. Some designs have very clear boundaries that make for good partitions and fit into an easy floorplan, but these tend to be the exception.

Also, the designer should use what they know about the design to their advantage. If there are some small blocks they know will change a lot, it may be worthwhile to add them as partitions and LogicLock them. For example, the design might have four major partitions with LogicLock regions, but three additional partitions on very small, hand-selected parts of the design. If one of those small parts is only 5% of the device but they know they will change a lot, then whenever it changes they will be able to keep the other 95% locked down. This is basically the strategy of Floorplanning a Single Hierarchy being used in conjunction with floorplanning the whole design.

### Tip: Avoid over-using Floating or Auto-Sized LogicLock regions

When floorplanning, users often have trouble determining where to place LogicLock Regions and how large to make them, so they try the Auto/Floating properties to let the fitter determine this. These options work all right when there is only one or two LogicLock regions in the device, but even in those cases I would recommend fixing the size and locking them down very soon, which can be done by right-clicking on a LogicLock region and selecting "Set Size and Origin to Previous Fitter Results". When LogicLock regions are flexible, they're basically like a large blob that the fitter tries to move around during the fit. Anytime the region is moved, a large amount of other logic will be displaced, which is very disruptive and can hurt results.

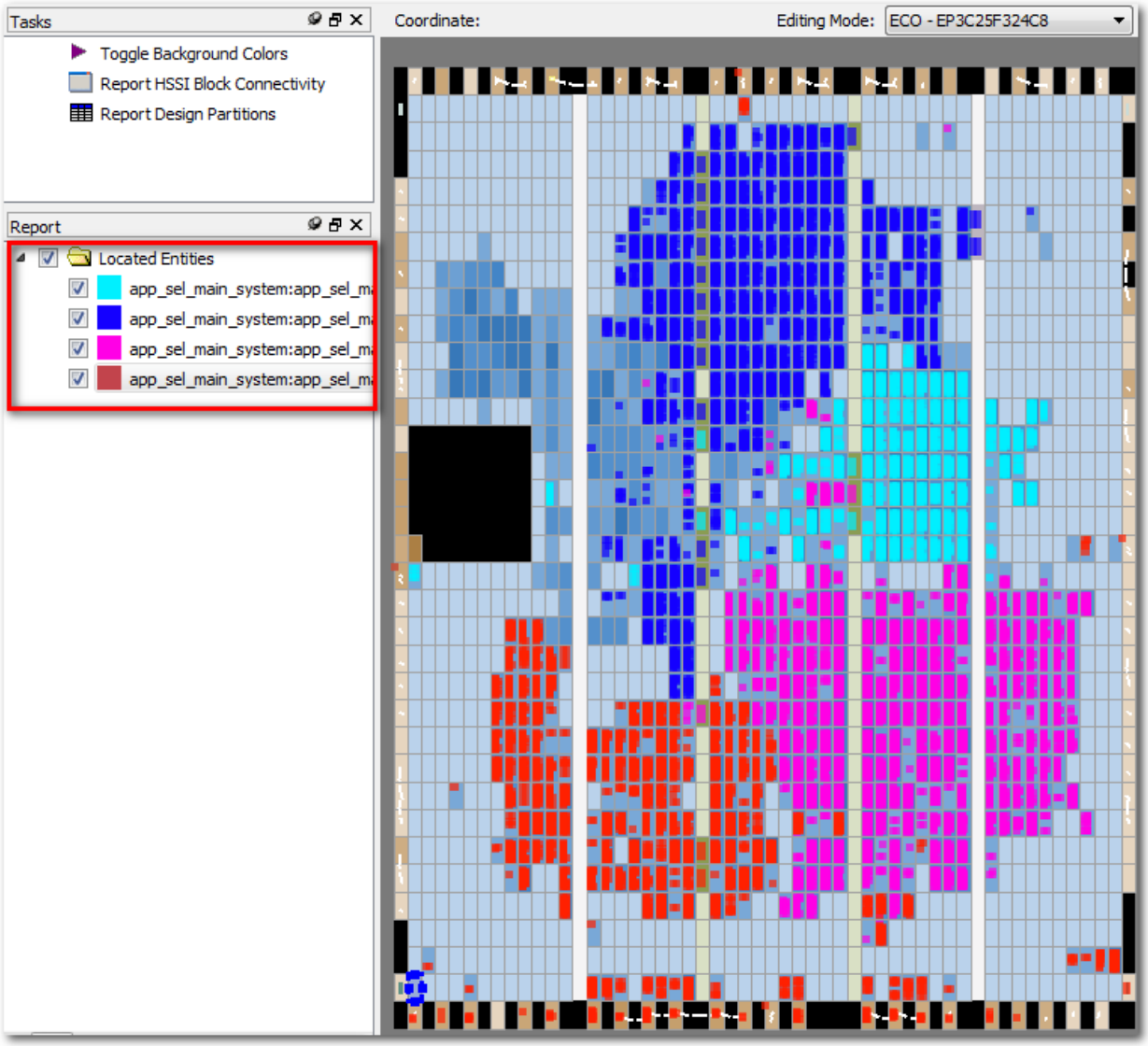
For designs with many LogicLock regions that need to fit across most of the device, Floating/Auto-Sized regions often result in a no-fit, or at least an inferior solution. For example, if 100% of the design were in Auto/Floating LogicLock regions, there is a good chance the final layout would have gaps in the floorplan that aren't covered by LogicLock regions, forcing some regions to be over-packed, while there are completely unused portions of the device.

Floorplanning is one of those things humans are pretty good at and computers are not. Note that most floorplanning tools are manual. If computers did a great job floorplanning, then most

floorplanning tools would have a button that says, “Floorplan Design”, and they would let it run before fitting. So though that Auto/Floating options for LogicLock can work and have some good use cases, they’re generally best to get started with something straightforward, and the user should quickly move to a fixed size and location for their LogicLock Regions.

### Tip: Right-Click Locate Hierarchies from Project Navigator to Chip Planner

Another trick for determining where to create a LogicLock region is to look at where the fitter placed the logic on the previous compile. As of Quartus II 11.1, users can highlight multiple hierarchies in the Project Navigator and right-click -> Locate in Chip Planner. The hierarchies will now show up as different colors, and along the left side of the Chip Planner the user can manually check/uncheck hierarchy colors to help with visualization:




Note that this is a design I would probably not floorplan, even if I had the four major hierarchies in partitions. There is plenty of room for them, and the fitter is doing a good job of segregating them, so

that if a single partition were re-fit, there would be a nice open area to work with. Of course, if subsequent fits on partitions started causing problems, I might then move to adding LogicLock regions, and being able to see where the fitter placed them might help me decide where to place the LogicLock regions.

## Floorplannining for Performance

When floorplanning for performance, there are generally two approaches: low-level floorplanning and high-level floorplanning. By low-level, I mean the user has a clear area of failing paths and is trying to address that through small, controlled floorplanning. High-level floorplanning entails large LogicLock regions that guide the fitter toward a better area in the overall solution space, from which it can then make better low-level decisions. To understand this, it is important to think about how the fitter works, which is described in the first section of this document:

http://www.alterawiki.com/wiki/The_Quartus_II_Fitter_and_Seed_Sweeps

The section on "Hierarchy" is most important, in that the fitter does not look at the design from a hierarchical perspective, and instead is always looking at low-level connections and timing requirements. Pulling these low-level connections together results in a fit that usually groups hierarchies together, but the fitter never does an overt act of high-level floorplanning. This correlates with the results I have seen, in that the fitter generally does a good job at low-level fitting, and hence low-level floorplanning does not help much, while high-level floorplanning often provides the best gains.

As an aside, floorplanning has become less fruitful as the fitter has improved. There was a time, almost ten years ago, when I could confidently gain improve performance by ten to twenty percent through floorplanning. As the fitter has improved, it has become more and more difficult to improve performance through floorplanning. Of course, having a good fitting algorithm is a good thing, but it can be annoying when the user wants even better results.

### Low-Level Floorplanning

The critical path in a design almost always looks to be poorly placed. It generally has multiple levels of hierarchy and numerous hops that fan-out to more than one location. Remember though, the worst path is usually part of a web of connections that are all critical, all fighting for better placement and better routing, and the fitter is directly trying to balance those requirements.

#### Tip: Putting critical paths in a LogicLock Region usually does not improve timing

I often see users look at a critical path, see that the placement is spread out, and think that putting the hierarchy containing that path into a LogicLock region will "keep it together" so that it meets timing. My experience is that this usually does not work. The fitter is already looking directly at critical paths and trying to modify the placement to improve them. Without a LogicLock region, the fitter has the entire die to work with, and is willing to move less critical paths out of the way in order to meet timing. By drawing a rectangle and putting the critical path into it, the user is just limiting the options of what the fitter is already optimizing.

I don't want to say this never works. Forcing the logic close together early in the fit may point the fitter down the road to a better solution. It's also possible the placer may be over-estimating the routing resources a path will have available or what their final timing will look like, and therefore place them too far away. But more often than not, just confining the critical paths to a rectangle will not help, and quite often, make things worse.

If things do get worse, it may be worth examining why. Perhaps other paths that were not as critical now get squeezed out so they have worse timing. It is worth seeing how these paths interact with the original critical paths. Maybe the LogicLock region pulls nodes away from nodes they connect to elsewhere. There may not be a direct "next step" to help close timing, but the more information the user has and the better they understand how their critical paths interact, the more likely they will be able to find a solution.

### Tip: Over-Constraining

One "low-level" technique I wanted to mention that might work better, or minimally be easier to implement, is over-constraining. The basic premise is that by tightening the constraints on specific paths, the fitter will see them as being more critical and work on them more. The .sdc syntax would be something like so:

```
if {$::quartus(nameofexecutable) != "quartus_sta"} {
        set_max_delay –from {*egress:e_inst|control_sm:inst|*} -to {*|data_pipe[*]} 4.7
}
```

If these paths were in a 5ns domain, these constraints would over-constrain them by 300ps during all modules except TimeQuest. So synthesis and the fitter would see this over-constraint and see these paths as being more critical and try to get better slack on them. The nice thing about this syntax is that the over-constraint won't be used during final timing sign-off, and instead the original 5ns will be used. Without the conditional "if", the path might run at 4.9ns, which would actually meet the user's requirements, would show up in red because it fails the over-constrained 4.7ns requirement.

Note that this technique does not just tell the fitter to "try harder" on these paths, but that these paths are more important than other paths that might have similar slacks. As a result, other paths that might be competing for better placement or routing, will no longer have access to them. If the fitter is doing its job, then it was already be working on these critical paths, and this technique will only confuse the fitter's goals and result in worse results. That is one of the reasons this technique is not overly recommended by Altera.

That being said, it does occasionally work. The cases I've seen are usually when the placer over-estimates the routing resources that will be available or the final timing numbers those paths will have, and so it thinks a path will meeting timing, but after routing and timing sign-off, it really is not. In this scenario, the user is imparting this knowledge to the fitter. The other thing to watch is if other paths suddenly get worse, try to understand how they relate to the original paths and if they're competing for

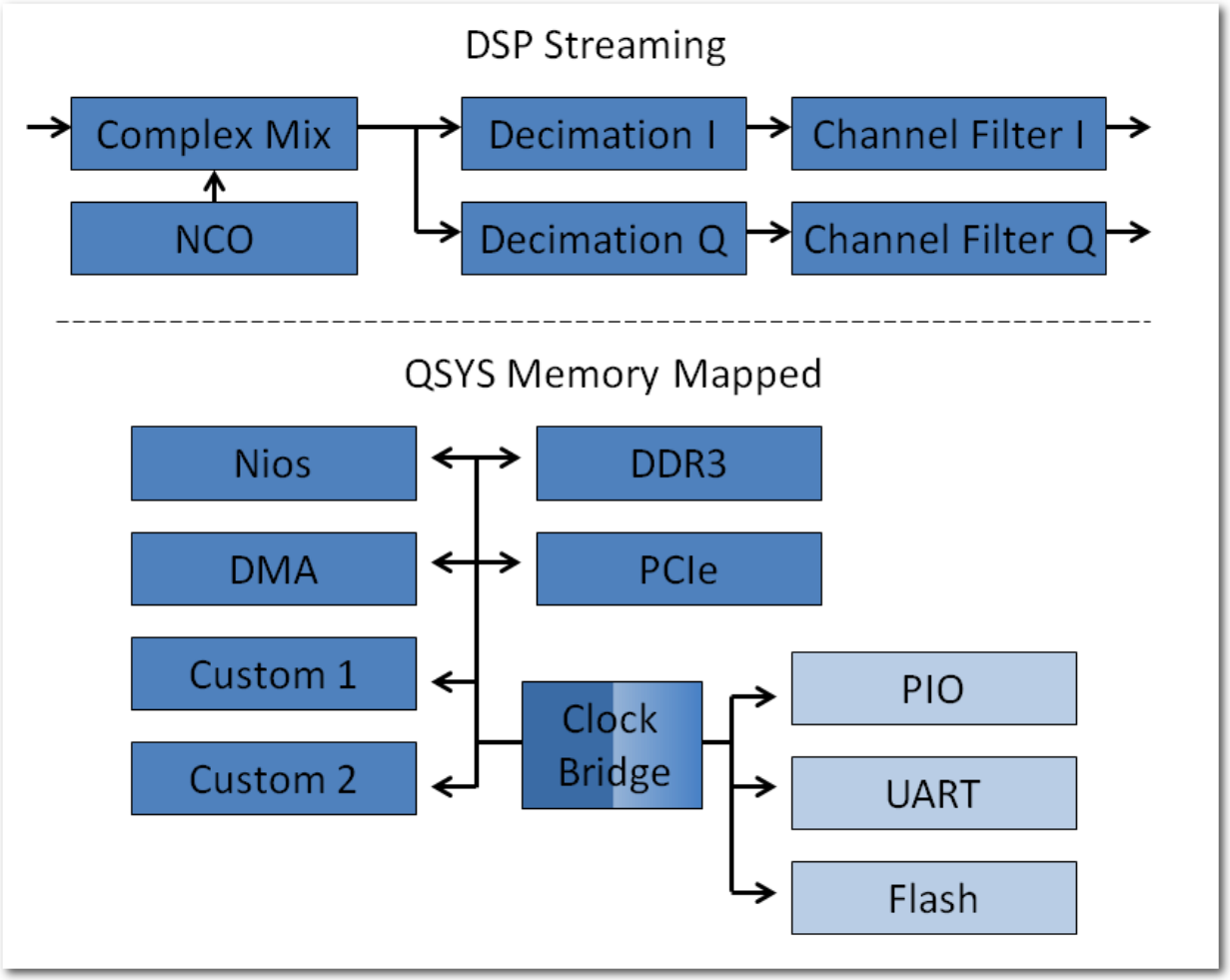
performance.  This information helps the user understand what the fitter is balancing when closing timing.

## High-Level Floorplanning

Low-level floorplanning is just looking at the critical path and trying to squeeze it into a better result.  As mentioned, the fitter should already be aware of the critical paths and working on it, so low-level floorplanning is unlikely to help.  High-level floorplanning is taking into account how all the design blocks fit together and trying to impart that information to the fitter.  With high-level floorplanning, the user is limiting the fitter's solution space, but hopefully limiting it to a better solution space.

### Tip: Think Up-Front If Design Can Be Floorplanned

The ability to get better performance through floorplanning is very design dependent.  Some designs naturally fall into a nice, 2D layout, while others will break when the user tries to fence hierarchies into distinct regions.  Remember that floorplanning is moving logic apart just as much as it is putting blocks together, and if the design can't handle this spreading, results can get worse instead of better.  Two types of designs I often use as an example are DSP streaming designs versus a QSYS memory-mapped design.  Let's look at a simple block diagram for each:




The DSP design has processing blocks which are nicely self-contained, where most connections are within each block and when done, sends the data to the next hierarchy. These data transfers are usually pipelined, so the blocks can be spread out quite a bit and still not hurt performance. These designs tend to work well when floorplanned. The critical paths are usually well contained within each block, and the user's floorplan guides the fitter into understanding this topology, rather than starting with all the logic smeared across the device and making the fitter discover this layout.

The QSYS design, on the other hand, has a very complex interconnect between blocks, which does address decoding, read/write handshaking, arbitration, etc. This logic often overlaps too, where some of the address decode may be common to multiple blocks, another part of the address decode is common to other blocks, and hence it is difficult to pin down which hierarchy a section of interconnect is driving. Quite often, the critical path in a QSYS system is inside the interconnect instead of an individual block. In these cases, fencing each hierarchy into separate regions just spreads the critical paths out, often making the design run slower. Letting the Quartus II fitter work within an open floorplan allows it to find a more subtle layout, where common address decoding may overlap, more critical logic may be pulled closer to the source while less critical logic is squeezed out, and certain areas of the device may contain blocks of logic from many hierarchies.

Note that I am only talking about high-level floorplanning, where the user is trying to put most of the design into LogicLock regions. If a particular hierarchy is especially troublesome in closing timing, isolating that block in the floorplan so the user can do Incremental Compilation for performance still makes sense. The user might also be able to floorplan large chunks of the system, such as separating the high-speed section in dark blue from the low-speed section in light blue. I have also seen systems where the QSYS system is only a portion of the design, say 30%, and so putting the whole QSYS system into a LogicLock region made sense.

Designs also don't always split into such clear boundaries as those that can be floorplanned and those that cannot. Some have a portion of DSP alongside a memory-mapped system. Some have a clear data-flow portion that is overlapped with a control portion(see Floorplanning to Break Timing). There is no clear indicator of what designs are good for floorplanning and which are not, but most users have a good idea of how the major hierarchies in their design hook up, where the critical paths are, and how floorplanning will most likely affect their design. From there, they can evaluate if floorplanning most of the design will likely help.

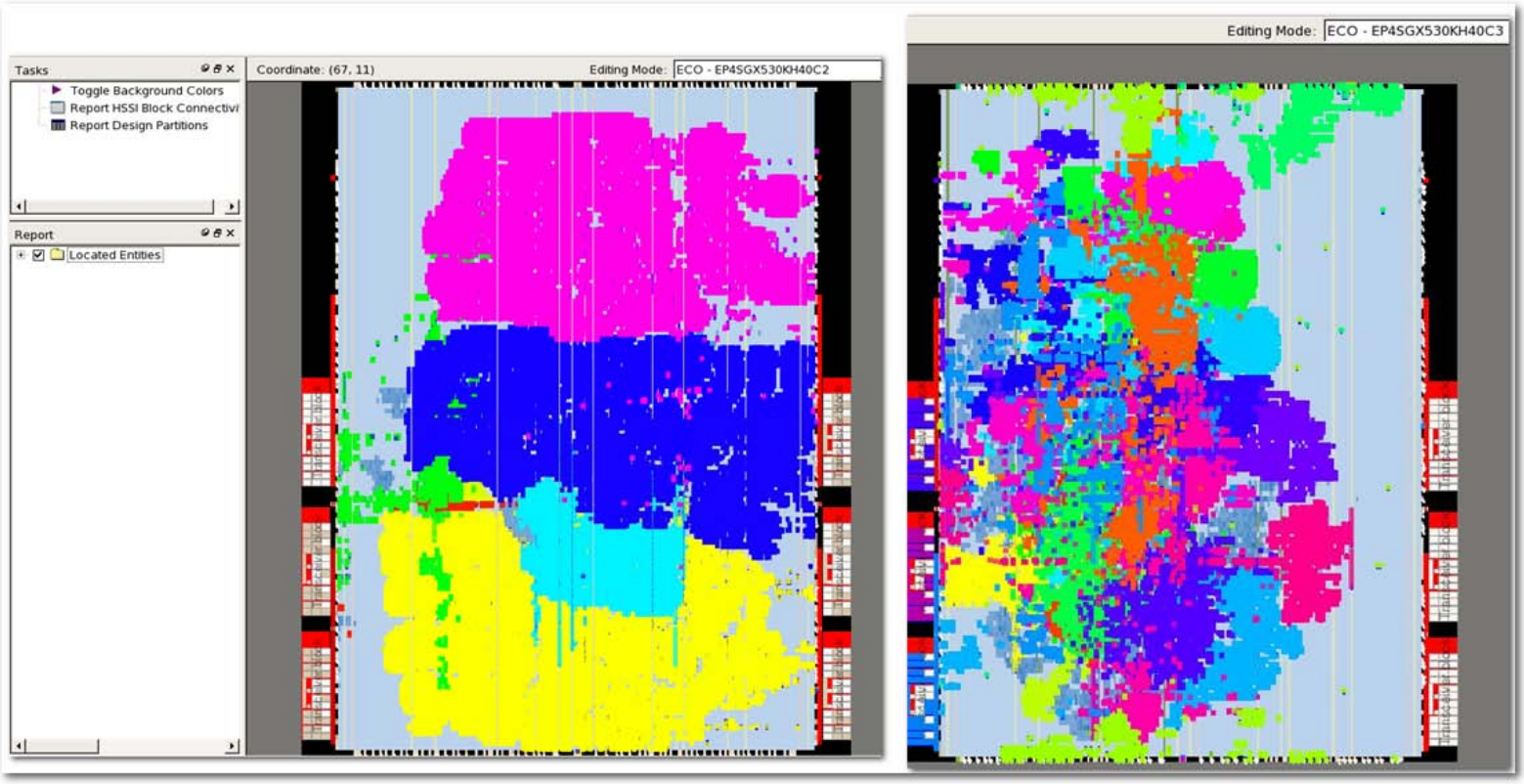
### Tip: Use Block Diagram

Most large designs have a block diagram, if not many block diagrams. I find these are often the best place to start when floorplanning. They tend to be drawn where most connections are contained within a hierarchy, and designers often design to the block diagram, trying to maintain logic within each block, only passing it on when it is ready to be processed by another block. The corollary to this is, if there is no block diagram, try making one first, before worrying about floorplanning the whole design. If the designer can't take a blank sheet of paper and do a rough drawing that shows how blocks connect and how data flows, they are unlikely to be successful doing it with their real design mapped onto real hardware.

Also, make sure the block diagram represents the I/O interfaces. If the DDR3 interface is on the top edge, draw it there. If the XAUI interface is on the bottom of the left edge, draw it there. I worked on one design where the block diagram clearly showed data received through transceivers on one side, processed through various blocks in the middle, and then sent out the right side. Once we got into the analyzing the design, I found they used the transmit and receive path of every transceiver in the device, which are laid out along the left and right edges. So half the data came in on the left and half on the right before being processed, and was then split again to be sent out the left and right sides. Until we realized this, the initial attempts to floorplan the receive and transmit blocks failed miserably.

### Tip: Analyze the Unfloorplanned Fit

Users can select multiple hierarchies in the Project Navigator and right-click -> Locate -> Chip Planner. Prior to Quartus II 11.1, they were all highlighted in the same color of blue. Starting with 11.1, they now show up in an individual color, and the Reports window in the Chip Planner allows the user to selectively check/uncheck each hierarchy. Although a new feature, improvements should also be coming, such as the ability to script this procedure. Here is a screen-shot from two different designs in an EP4S530GX device:




The designs are not complete, so there is a good amount of space in each. Note that the left screen-shot shows the Located Entities folder, which if opened, would show each hierarchy that is being displayed, its color, and the ability to select/deselect its view. The Chip Planner's pull-down menu View -> Report Window will access this window.

So starting with the design on the left, there were some very clear, large hierarchies in the design. On the one hand, this design looks like it would be pretty straightforward to floorplan, and hence might be a good candidate. If floorplanning so the design can use Incremental Compilation, this layout looks promising. The flip-side is that the fitter seems to be doing a good job of understanding the hierarchy and fitting to it, so going through the process of creating LogicLock regions that mimic the pink, blue and yellow blocks won't really impart a lot of information that Quartus II does not already know. Also, floorplanning may break some of the subtle placement occurring. Notice the pink dots inside the dark blue region. Notice the streak of green cutting through the yellow block. These may be paths that are not timing critical and the fitter just put them there and never had a reason to take them out. But they might also be part of critical paths between these major blocks, which the fitter was able to intersperse and achieve better results. It is difficult to determine this upfront, but even when floorplanning breaks timing, analyzing why can be very useful.

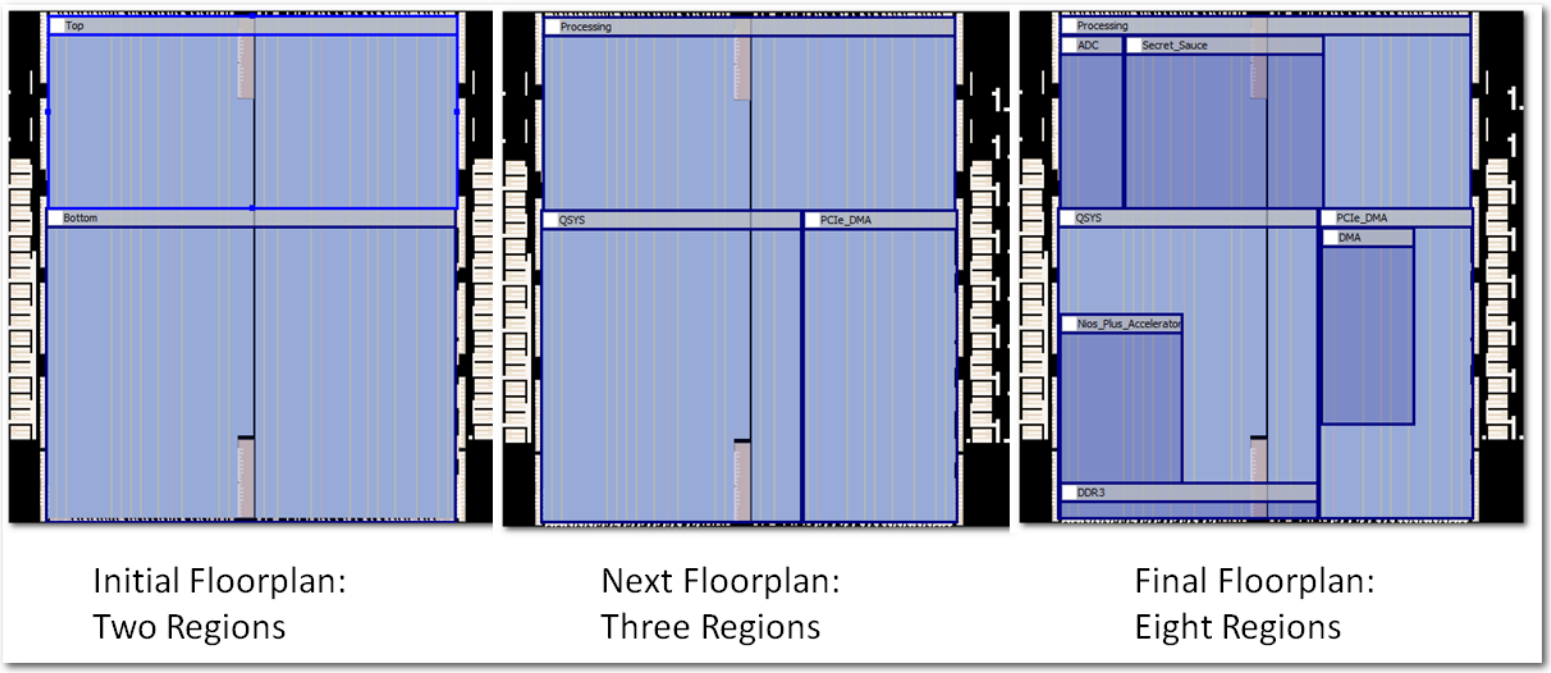
Now let's look at the floorplan on the right. There were no large hierarchies in the design, and so I highlighted the top twenty-three. This floorplan is a mess. If I had to layout these blocks into distinct regions, I wouldn't know where to start. So on the one hand, I would be very leery about floorplanning this design, as there is a good chance it is just a rat's nest of connections. On the flip-side,

the fitter may be doing a poor job of grouping hierarchies together, and a good floorplan could really clean up the layout and let the fitter work within a better area of the solution space.

So there are two ways to view the fitter's results. When it does a good layout, the design may be easy to floorplan, but there may be less upside to floorplanning. When the fitter does not find a clean layout, it may be because one does not exist, but if the user is able to create a good floorplan, there may be more upside.

### Tip: Keep It Simple

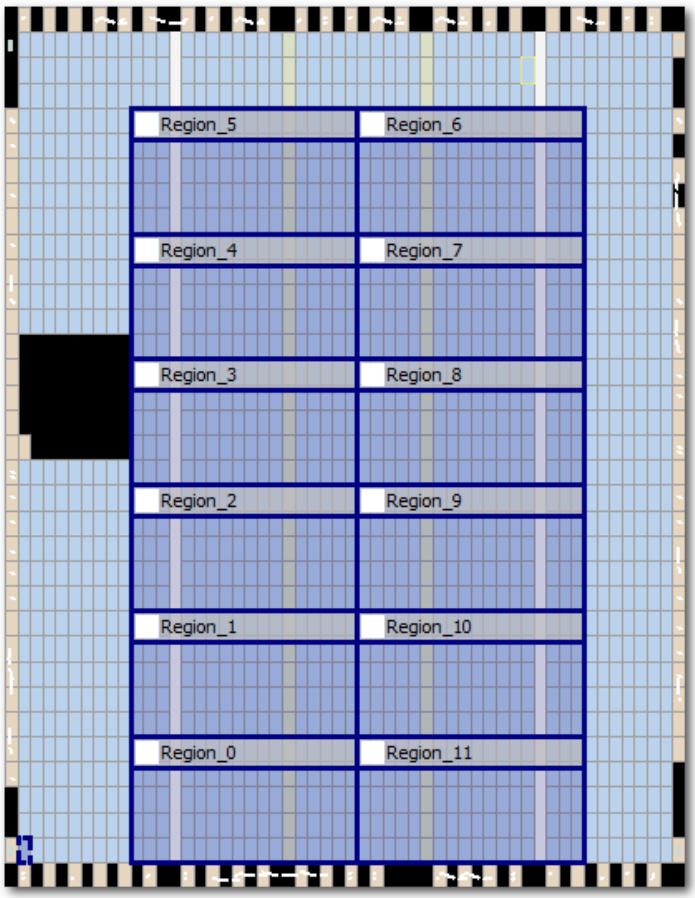
The previous floorplan(on the right) looks like it would be extremely difficult to floorplan, but remember, there is no requirement to have a LogicLock region for each major hierarchy. It would be perfectly acceptable to floorplan the design into two LogicLock regions, and placing each hierarchy into one of those two regions. For many designs, I recommend starting off with no more than four LogicLock regions, and quite often will start with only two. This tends to keep the floorplanning process easy. From that point, one can evaluate the results to see timing got better, stayed the same, or broke. If it broke, they can evaluate why and hopefully learn from that. If it stayed the same or got better, they can keep that simple floorplan(which I often do) or add more regions to refine the layout. Here are some example floorplans:




Though the user started with just two regions, they slowly added more and refined the layout. To be honest, if I opened a project and saw the final floorplan, I would normally be skeptical, thinking the user had added unnecessary LogicLock regions. Quite often nothing more than two or three LogicLock regions is perfectly acceptable and more than enough to help guide the fitter to a better solution.

### Tip: Over Floorplan when Obvious

The last tip said to keep the floorplanning simple. Although that is true most of the time, there are occasions when a floorplan is obvious; in which case making many LogicLock regions makes sense. As an example, I worked on a design that had a small processing block that needed to be repeated as many times as possible and run as fast as possible. The block used 5 rows of DSP blocks and we found that the closer those DSP blocks were put together, the better the result. When floorplanning, we came up with the following layout:




First, note that there are 12 blocks, but originally we had 13, since there were enough DSP blocks. Once we started floorplanning, we found that one of those blocks would have to straddle the two white columns of DSP blocks, and its performance would be considerably worse than the other 12 blocks. Since all blocks had to run at the same speed, it was deemed better to run with 12 fast blocks instead of 13 slow blocks. Although it does not happen very often, in this case just the act of floorplanning helped explain why the fitter was not giving the results we thought it could achieve.

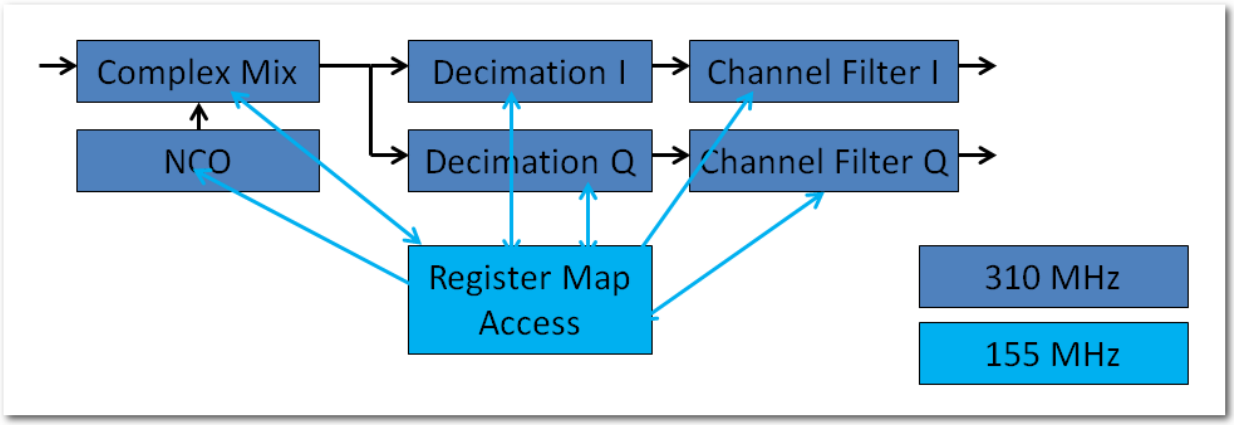
Going to 12 blocks still did not give nearly as good results as an individual block, but once we floorplanned them like above, the results got significantly better. In the end, we isolated each block and ran multiple seeds until it got the very best performance, then merged them altogether like the previously described method for isolating multiple partitions. This achieved the desired results.

The floorplan above is really just an example, as the original design was in a much larger device and had more than 30 of the processing blocks, each in their own LogicLock region. So my advice to start with less than 4 regions didn't apply here, because there was a very clear layout for 30 high speed blocks, all needing to be placed compactly around a few DSP blocks. Also, the rest of the logic around these blocks was highly pipelined and could be placed after these blocks were locked down and still meet timing. If the rest of the logic was timing critical, there is a decent chance this layout would cause timing degradation on the logic fit around it.

As can be seen, there are many factors in determining if to floorplan a design, and if so, how to do it. My biggest recommendation is to put some thought into it up front, and be sure to spend time analyzing the results. In fact, that leads to another tip that user's seldom follow.

### Tip: Floorplan to Break Timing

This one may be done on purpose or accident, but the basic premise is that when a user floorplans their design and timing gets worse, they try to figure out why. Let's start with the simple DSP block diagram show in the previous tip on floorplanning. We stated that this was an excellent candidate for floorplanning, since timing critical signals are nicely contained within each block. So let's say the user floorplans these blocks across the die and finds that timing gets worse. But instead of just stating that floorplanning made things worse, they spend time carefully analyzing what got worse and why. They may look at the critical paths and see how they fit in an older, flat version of the design, as well as see how critical paths from the flat compile fare in the new version. With careful analysis, they find new critical paths that are part of a register mapped control plane. Going back to the block diagram, they realize they neglected to show this control plane because it runs at half the rate of the DSP blocks. A new block diagram may look like so:




First off, this may help explain why the DSP blocks, which ran very fast when isolated, run so slow with a flat compile. The fitter must now pull part of their logic toward the register map access point, which then pulls the logic it's connected to, and forces the fitter to overlap these components, making the results much worse than when each block was isolated. Even though the critical paths might still be in the 310Mhz domain, paths from the 155Mhz domain may have been near the top, since the

fitter is always balancing slack and trying to get the least slack across the whole design. By floorplanning, the user has forced the 155Mhz domain to the top.

I have seen cases where it may not even be anything so large as a memory map. For example, I have seen registers with identical behavior get merged across distant hierarchies, which now forces them to be pulled together. This is not something in the user's original code, so they are unaware of it, and these paths aren't the worst case ones so they don't show up at the top of timing reports, but they still caused timing degradation. Once the user floorplanned these hierarchies apart, they became aware of the problem and were able to fix it, alleviating this stress point on the design.

Finally, in this example, the Register Map hierarchy is probably much easier to fix. The user should be able to pipeline it without adversely affecting the design. They might try replicating the addressing for each hierarchy, so the addressing logic can be spread out to each distinct endpoint. There are probably a number of things they could do to the Register Map that could help timing, where they had previously been working on the DSP blocks, which had already been highly optimized and had little room for improvement. This analysis leads to another tip:

### Tip: Add set_false_path to Test How Potential Modifications Effect Overall Fit

In the previous example, the user thinks the low-speed Register Map block is hurting timing on the rest of the design. Before modifying the code, they could add the following to their .sdc:

```
set_false_path -from {*|register_map:*} -to {*|NCO:*}
set_false_path -from {*|register_map:*} -to {*|complex_mix:*}
set_false_path -from {*|register_map:*} -to {*|decimation:*}
set_false_path -from {*|register_map:*} -to {*|channel_filter:*}
```

These assignments are wrong, and hence the fit could never be used in hardware, but it would be a real quick experiment to see if fixing timing on paths from the register_map to the DSP blocks helps the overall design. If it has no effect, the user will probably have to pursue another method for optimizing the design, but if the slack on blocks inside the DSP blocks get faster, then they know they're on a promising track and can start modifying the register map access.

This method is most useful for "what if" scenarios, to quickly see how fixing one section of paths might affect the overall design. Another scenario is to cut timing between various blocks the user wants to LogicLock, with something like so in the user's .sdc:

```
set hierarchies { \
        *|ingress:ing_inst|* \
        *|egress:egr_inst|* \
        *|sopc_subsystem:sopc_inst|* \
        *|rf:rf_inst|* \
}
soreach src_hier $hierarchies {
        foreach dst_hier $hierarchies {
                if {$src_hier != $dst_hier} {
```

```
                        set_false_path -from $src_hier -to $dst_hier
                        post_message -type info "Adding invalid constraint: Set_false_path -from $src_hier -to $dst_hier"
                }
        }
}
```

This can be done before the user has floorplanned, which would allow the fitter to spread these blocks apart and see if the results got better. It can also be done after the user has floorplanned to see if their LogicLock sizes and locations can provide good results within each block, ignoring paths between the blocks.

This type of analysis can be useful, but it can also cut out quite a bit that the fitter has to work on, and therefore may not always be representative of the real design. I would not recommend spending too much time on this type of analysis, but it can open the door to interesting insights.

## Conclusion

A large number of strategies and tactics have been provided as "tips". Users that blindly take these tips and start pushing them onto their design are unlikely to have success. Most of these tips require careful analysis of the design and its timing. From there, trying to do some up-front analysis of which ones are most likely to satisfy the user's goals is the most likely way to have success. Even when a tip is tried and it hurts results, determining why it hurt results can be extremely useful and lead to the next steps. Careful planning and detailed analysis of results are a must for having success.