Using Xilinx and Exemplar for Incremental Designing (ECO)

XAPP165 August 9, 1999 (Version 1.0) Application Note

Summary

Guided place and route (PAR) can help you reduce runtimes when incremental changes are made to a design, such as for an Engineering Change Order (ECO). By making only small changes to a design along with optimizing only the changed block or blocks, you allow guided PAR to perform at its best, preserving timing and reducing PAR runtimes. To localize the design changes without affecting the remainder of your design, either a top-down preserving hierarchy or a bottom-up methodology must be used.

Synthesis Trade-off for Incremental Designing

At the synthesis stage you must decide to keep the hierarchy of the design or flatten it. As designs become larger with the Xilinx Virtex devices, the trade-off of a single flattened run must be weighed against a hierarchical approach. In a hierarchical approach, you must consider an overall design flow that includes constant changes and recompiles. With a single flattened run approach, the hierarchy is flattened for optimization. Each methodology has its advantages and disadvantages, but as higher density FPGAs are introduced, the advantages of hierarchical designs outweigh any disadvantages.

Top-down (flattened design):

- Longer synthesis times (incremental or non-incremental).
- Better optimization.
- Automatic detection of resources such as global buffers, GSR, and so on.
- Single EDIF netlist and ncf file.
- Must re-optimize the entire design when making changes.

Top-down preserving hierarchy:

- Faster run times (non-incremental and incremental), as opposed to a top-down methodology.
- Automatic detection of resources such as global buffers, GSR, and so on.
- Expect the resulting design to run slower than a top-down design do to the boundary optimization limitation.
- Single EDIF netlist and ncf file.
- Only the HDL files that change are re-optimized and a single new EDIF and ncf file are written. This results in faster runtimes for the incremental as opposed to the top-down approach.
- All bi-directional I/O must be on the top level.
- Supports team-based designing.
- Supports incremental design changes.

Bottom Up:

- Faster run times (non-incremental and incremental), as opposed to a top-down methodology.
- Expect the resulting design to run slower than a top-down design do to the boundary optimization limitation.
- All bi-directional I/O must be on the top level.
- Supports team-based designing.
- Supports incremental design changes.
- Only the HDL files that change are re-optimized and a new EDIF/ncf or XDB file are written, which results in faster runtimes for the incremental as opposed to the top-down approach.

- Two different bottom-up methods:

1. Allowing Exemplar to “stitch” all the separate modules together:
   - One EDIF file for the entire design
   - Auto detection of resources such as global buffers for clocks
   - When a module or sub-module change is made, the top XDB file must be read back into Exemplar as well as either the sub-block XDB or the updated HDL file (along with optimize) to write out a new top-level netlist. (XDB is Exemplar binary files, so reading them in is fast)

2. Allowing the Xilinx tools to “stitch” the design together:
   - One EDIF file for each HDL file
   - No automatic detection of global resources

For a sub-module design change the entire design will not have to be read back in to Exemplar to write out a new netlist. Just pass the newly updated EDIF file to Xilinx.

ECO Synthesis Guidelines

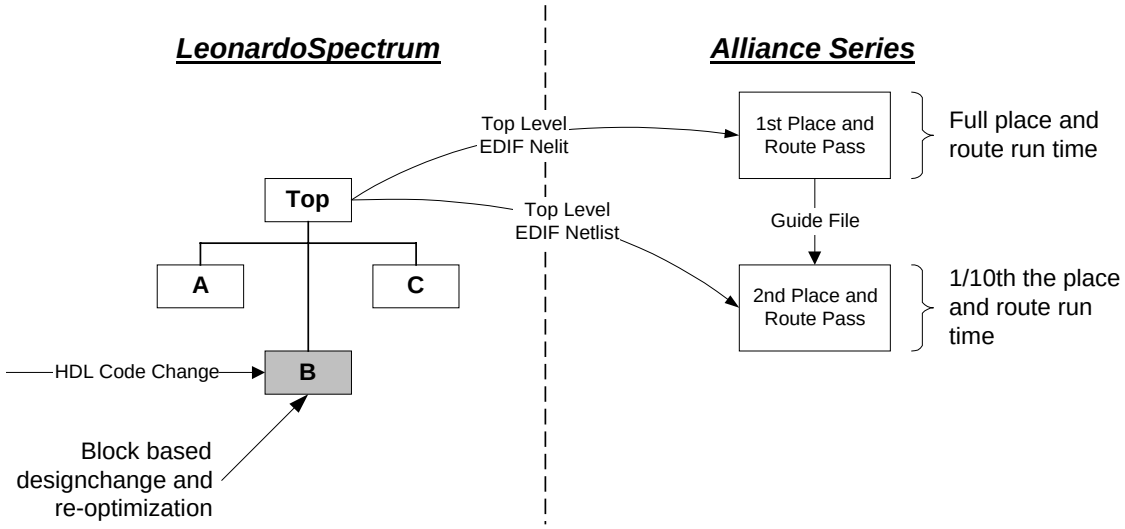
To benefit from a hierarchical approach, strategies are required to effectively partition the design, optimize the hierarchy, and fine-tune the hierarchical synthesis process. Effectively partitioning the design gives you better results, faster run times, and simplified scripts.

Partition your design based on functionality, clarity, and how you plan to constrain the design. Partitioning should result in smaller individual blocks. The following guidelines are general and vary with design style, constraints, and device architecture.

1. Keep critical paths within one block for better synthesis optimization.
2. Incremental design changes should be focused on Boolean logic changes. Arithmetic changes (such as addition, multiplication), or anything using carry logic that is modified, could be drastically changed.
3. Keep all I/O at the top level.
4. Place registers on the I/O’s of the module/entity cells. A good design practice is to make all input signals or all output signals registered at the hierarchy boundaries. Using registered inputs/outputs simplifies the time budgeting calculations because this allows critical path timing to be budgeted automatically between registers, based on clock constraints. Also, registering the I/O’s design hierarchy can eliminate any possible problems with logic optimization across hierarchical boundaries.
5. The top level should only contain only blocks and interconnects. Try to avoid glue logic. Otherwise, group the extra logic into a small sub-design.
6. Modularize shared logic. Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.
7. Isolate FSMs (Finite State Machine). This recommendation enables easy state transformation and experimentation.
8. Use one module/entity-architecture per file.
9. Module/entity name should match the base filename.
10. The filename, entity, and component declaration must be the same case. For the bottom-up method, the base filename is used to write the EDIF file. It is recommended using all lower-case characters.
11. Exemplar can not always infer a BUFG (global buffer) if the bottom-up method is used (allowing Xilinx to “stitch” the design methodology), and you may have to use the `buffer_sig` attribute to apply a global buffer to a signal, or use the clock constraint depending on the bottom-up methodology. The buffer_list variable is used to force BUFGs on the specified signals in the provided script.
12. Synthesis timing constraints for the bottom-up method must be applied individually to each block. Top-down hierarchy is able to propagate the clock constraint to lower-level blocks for optimization of the blocks. Any timing will be written to a base filename NCF file and Xilinx will automatically pick it up. All other constraints are written into the resulting EDIF file.

LeonardoSpectrum Synthesis Methodologies

Top-down Preserving Hierarchy



Exemplar recommends using the top-down preserving hierarchy method to do incremental designing and to use the bottom up method for team-based designing. LeonardoSpectrum Level 3 provides the necessary hierarchy control and optimization capabilities necessary to accommodate the top-down preserving hierarchy and bottom-up methodologies. The top-down preserving hierarchy is done by either setting the `hierarchy_preserve` attribute to TRUE, or by selecting the appropriate box in the Optimize power tab in the GUI. Also the following attribute MUST be set in order to prevent optimization across boundaries:

```
set bubble_tristates FALSE
```

This can also be accomplished by de-selecting this "`bubble tristates`" option in the `Optimize` → `Advanced setting` tab in the GUI.

For top-down preserving hierarchy and bottom-up compiles, all bi-directional I/O must be specified in the top-level HDL file.

Initial Optimization Procedure Using Top-down Preserving Hierarchy

Set the technology environment for Virtex. In this example we will use the v50bg256-4 device

```
set part v50bg256
set process 4
load_library xcv
```

Certain optimizations must be disabled to preserve the integrity of the sub-blocks and insure that the incremental optimization flow will work. Please note that some small sacrifices in quality of results must be made in order to create a synthesized design that will facilitate an incremental design flow. This can be accomplished by setting some system variables that will disable "across-boundary" optimizations.

```
set bubble_trisates FALSE
set no_boundary_optimization TRUE
set auto_dissolve_hierarchy FALSE
set virtex_map_iob_registers FALSE
```

Read in the design. We will assume a design with three sub-blocks, ***A,B,C*** and a top-level block, ***TOP.***

```
read {A.vhd B.vhd C.vhd TOP.vhd}
```

Optimize the design from the top-level with hierarchy preserved. The "-chip" option will insert I/O buffers at the top-level.

```
optimize –ta xcv –effort quick –area –chip –hier preserve
```

Once optimization completes, and if results are satisfactory, then save the netlist in LeonardoSpectrum's native "xdb" database. This is the netlist we will use when making incremental design changes.

```
write top.xdb
```

Save a netlist for the Alliance Series software. We will use the "auto_write" command which will post-process the netlist to conform to all Alliance Series syntax rules.

```
auto_write top.edif
```

The design is now ready for place and route with the Alliance Series software.

Incremental Optimization Procedure Using Top-down Preserving Hierarchy

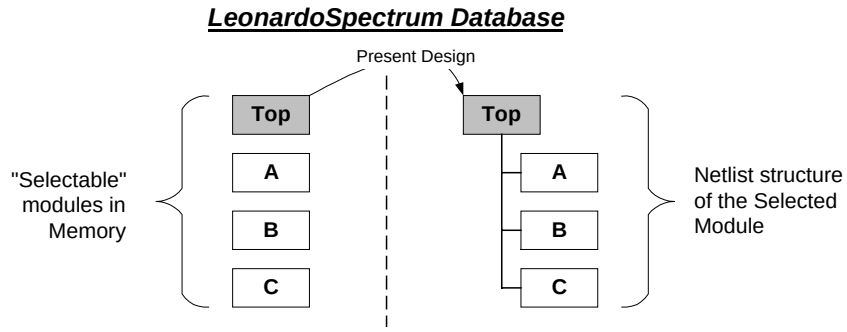
The key to successful incremental synthesis and guided place and route is to minimize the number of instance and net names that change. To achieve this, we will use LeonardoSpectrum's ability to re-load and re-optimize sub-blocks in a design while leaving remaining blocks unchanged.

1. Make an RTL design correction to one of the sub-blocks.

2. Restore the original design into LeonardoSpectrum by reading the "xdb" file saved earlier. The Virtex technology should already be loaded.

   ```
   load_library xcv
   read top.xdb
   ```

   Note: *LeonardoSpectrum's database is capable of "swapping" different modules into the optimizer using the* ***present_design*** *command. The design browser displays all the modules in memory that can be set to the "present design". The "selected" module is highlighted with bold lettering.*



3. Read the modified VHDL or Verilog file

   ```
   read A.vhd
   ```

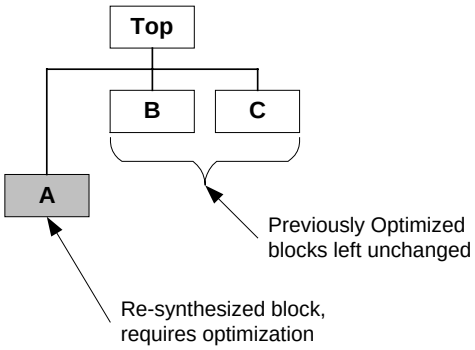
4. After the read command has completed the module "A" will be set to the present design. Use the ***present_design*** command to switch back to the top-level design

   ```
   present_design .work.top_level.rtl
   ```

   Note: *Because both VHDL and EDIF support multiple netlists (architectures, views) for a single module, LeonardoSpectrum's database also supports multiple "views" for each cell. For that reason the* ***present_design*** *command references the view and not just the cellname.*

VHDL Viewnames = Architecture Name (All viewnames set to lower case)
Verilog Viewnames = INTERFACE (Interface is always UPPER case)

5. Optimize on the incrementally loaded block while leaving all remaining blocks untouched. This block can be referenced directly by the optimize command. The ***"-macro"*** switch is used to prevent I/O buffer instantiation occurring on the sub-block.




```
optimize .work.top.a.rtl –ta xcv –effort quick –area -macro
```

6. When optimization completes, save the netlists in both the Exemplar "xdb" and Xilinx EDIF formats.

```
write –format xdb top_reoptimzed.xdb
auto_write –format edif top.edif
```

Synthesis runtimes when re-synthesizing an updated HDL file are the same as when doing bottom-up, because Exemplar writes out and reads in XDB files, which are Exemplar binary files. (Only the updated HDL file will get re-synthesized.)

The main benefit to using the top-down preserving hierarchy and bottom-up method is name preservation, which is the key to getting the best results from guiding a design.

Bottom-up Method

For the bottom-up methodology there are two different ways to integrate with the Xilinx tools:

1. Write XDB files for all the lower-level modules to be read in by Exemplar to "stitch" the files together and optimize the top level only. Then a single EDIF netlist can be written out to be implemented in the Xilinx tools.

2. Write a separate EDIF netlist for each piece of the design and allow the Xilinx tools to read these in to "stitch" the design together.

Initial Optimization Procedure for the Bottom-up Method

(I) Allowing Exemplar to "stitch" the design together

Note: You can run separate synthesis runs for each block (step 1-4), for example, if multiple designers are involved. The "XDB" files for the lower levels will be read in, then the top-level HDL file will be read in and optimized, and a single EDIF file will be written out for implementation in the Xilinx tools.

1) Set the technology environment for Virtex. In this example we will use the v50bg256-4 device.

```
set part v50bg256
set process 4
load_library xcv
```

2) Read in the HDL block of the design to optimize. We will assume a design with three sub-blocks, ***A,B,C*** and a top-level block, ***TOP.***

```
read {A.vhd}
```

3) Optimize the block as a macro (Blocks A,B, and C), or optimize the top level as a chip (Top), which will insert I/O buffers at the top level.

```
Optimize –ta xcv –effort quick –area –macro
```

4) Once optimization completes and if results are satisfactory, then save the netlist in LeonardoSpectrum's native "xdb" database. This is the netlist we will use when making incremental design changes.

```
write A.xdb
```

5) Optimize and write out the "XDB" files for blocks B and C using the same method as for step 2 to step 4.

6) The top level will be read and optimized only after all the XDB files are read, to write out a single EDIF netlist.

```
read {A.xdb B.xdb C.xdb}
```

```
read {top.vhd}
```

7) Optimize the top-level using the "chip" option to insert the I/O buffers.

```
optimize –ta xcv –single_level –effort quick –area –chip
```

Once optimization completes, and if results are satisfactory, then save the netlist in LeonardoSpectrum's native "xdb" database. This is the netlist we will use when making incremental design changes.

```
write top.xdb
```

Then save a netlist for the Alliance Series software. For this we will use the "auto_write" command which will post-process the netlist to conform to all Alliance Series syntax rules.

```
auto_write top.edif
```

The design is now ready for place and route with the Alliance Series software.

Note: Remember when doing a new synthesis run that Step 1 of loading the library must be done before doing a read, such as when you are reading in "XDB" files.

(II) Allowing Xilinx to "Stitch" the Design Together

1) Set the technology environment for Virtex. In this example we will use the v50bg256-4 device

```
set part v50bg256
set process 4
load_library xcv
```

2) Read in the HDL block of the design to optimize. We will assume a design with three sub-blocks, ***A,B,C*** and a top-level block, ***TOP.***

```
read {A.vhd}
```

3) Optimize the sub-block as a macro (blocks A,B, and C).

```
Optimize –ta xcv –effort quick –area –macro
```

4) Once optimization completes and if results are satisfactory, then save the netlist for the Alliance Series software. For this we will use the "auto_write" command which will post-process the netlist to conform to all Alliance Series syntax rules. This netlist will be used directly by the Xilinx software unless the macro changes, then a new EDIF netlist will be written out.

```
auto_write A.edif
```

5) Optimize and write out the EDIF files for block B and C using the same method as used in step 2 to step 4.

6) Read the top-level HDL file.

```
read {top.vhd}
```

If you have not instantiated a BUFG in your top-level design and you want one on a clock signal, you must use an Exemplar attribute to get the global buffer inserted. (A top-down preserving hierarchy approach would do this automatically.)

```
Set_attribute –port clk1 –name BUFFER_SIG –value BUFG
```

7) Optimize the top level using the “chip” option to insert the I/O buffers.

```
optimize –ta xcv –single_level –effort quick –area –chip
```

Once optimization completes, and if results are satisfactory, save the netlist for the Alliance Series software. For this we will use the “auto_write” command which will post-process the netlist to conform to all Alliance Series syntax rules.

```
auto_write top.edif
```

8) The design is now ready for place and route with the Alliance Series software.

Notes for using the bottom-up method, allowing Xilinx to “stitch” the design:

- For VHDL users: if a configuration is used and references a lower-level synthesis, you will get an error.
- For Verilog users: all lower-level macros will be “black boxed”, and you will therefore have to add a module declaration for these lower-levels macros.
- The Global buffer will not be automatically inserted using this method, therefore you must use the Exemplar attribute used in step 6 above.
- The STARTUP block may not be inferred and will have to be instantiated.

Incremental Optimization Procedure for the Bottom-up Method

I. Allowing Exemplar to “Stitch” the Design Together

If this is a single-engineer design, the top XDB file can be read in to restore the design. The HDL file can be read in and optimized, then the top level can be set as the present design and optimized as a single level, and a new netlist can be written out. If this is a multi-engineer design, then the individual designer can re-synthesize the design and write out the XDB file for that particular block. Then the lead engineer can read in the top-level XDB file and the new XDB file for the sub-block, set the top level as the present design and optimize, then write out the new EDIF netlist.

1. Set the technology environment for Virtex. In this example we will use the v50bg256-4 device

```
set part v50bg256
set process 4
load_library xcv
```

2. Restore the original design into Spectrum by reading the “XDB” file saved earlier. Only the top-level XDB file needs to be read in. Previously we read in the sub-block XDB files, but then wrote out the new top.xdb, which included all the lower levels.

```
read {top.xdb}
```

3. Either read in the XDB file (Run Initial Optimization Procedure for bottom up (I) Allowing Exemplar to "stitch" the design together as in steps 1-4 on the updated HDL file), or read in the modified HDL file and optimize that block.

```
read {b.xdb}
```

or

```
read {b.vhd}
optimize –ta xcv –effort quick –area -macro
```

4. Set the present design back to the top level.

```
Present_design .work.top_level.rtl
```

5. Optimize the top-level using the "chip" option to insert the I/O buffers.

```
optimize –ta xcv –single_level –effort quick –area –chip
```

6. Save the netlist in both the Exemplar "xdb" and Xilinx EDIF formats.

```
write –format xdb top_reoptimized.xdb
auto_write –format edif top.edif
```

7. The design is now ready for place and route with the Alliance Series software.

II. Allowing Xilinx to "Stitch" the Design Together

The HDL file that is modified must be re-synthesized and a new EDIF netlist written out for it.

1. Set the technology environment for Virtex. In this example we will use the v50bg256-4 device.

```
load_library xcv
```

2. Read in the HDL file to be re-synthesized.

```
Read {b.vhd}
```

3. Optimize the design as a macro or chip as needed.

```
Optimize –ta xcv –effort quick –area –macro
```

4. Write out the new EDIF netlist.

```
Auto_write –format edif b.edif
```

5. The design is now ready for place and route with the Alliance Series software. Make sure all the EDIF files are in the correct directory for implementation.

Exemplar Sample Scripts (variables to set): TDH.tcl, BU.tcl

Script variables users must set in the Tcl script:

`set_working_dir`
: An Exemplar variable to determine where the work directory is and where to write log files.

`target_tech`
: Sets the target technology library to be loaded by the synthesis to software. For example for Virtex, target_tech would be set to xcv

`part`
: List the device and package. For example, `v150pq240`

`src_dir`
: List the directory where the HDL source files are located.

`dest_dir`
List the directory where the output EDIF, NCF, and XDB files will be written.

`lib_list`
List the library file or files if they exist. Include the extensions.

`sub_list`
List all the HDL files in the design except the top-level file. Include the extension. Used for the bottom-up methodology only.

`file_list`
List the HDL files starting from the lowest level and ending with the top level. Include the extensions. Used for the top-down and top-down preserving hierarchy methodologies.

`top_file`
List the top-level HDL file. Include the extension.

`top_entity`
List the top-level HDL file entity name. This is needed to optimize and write out the top-level output EDIF file. Used for the top-down preserving hierarchy method only. Must be used with the top_arch variable.

`top_arch`
List the top-level HDL file architecture. This is needed to optimize and write out the top-level output EDIF file. Used for top-down preserving hierarchy method only. Must be used with the top_entity variable.

`buffer_list`
List the top-level port names that need to have a BUFGP (global buffer) forced onto them. This will use a "`set_attribute`" command to force the BUFGP on the listed port names.

Running the Scripts

There are two scripts: TDH.tcl, and BU.tcl

Tcl can be used as a scripting language to run Exemplar, and can also run the Xilinx tools. Tcl scripts have a .tcl extension and can be executed from the Exemplar GUI or in batch mode in a shell in which the Exemplar environment is setup. To run Exemplar in batch mode, you can type the following:

```
spectrum -file synth.tcl
```

This will execute the Tcl script and log everything to a file named exemplar.log, as well as output to the screen. All Exemplar commands as well as shell commands are supported.

Note: In the Exemplar GUI you can do the following:

```
Source synth.tcl
```

Script Behavior

Top-down preserving hierarchy:

- Only the HDL files that change are re-optimized and new EDIF and ncf files are written for the entire design.
- All HDL files will be re-synthesized if any one of the HDL files is updated and newer than the resulting EDIF file.
- One EDIF file and one ncf file for the entire design.

Bottom-up:

- Only the HDL files will be re-synthesized if the HDL file is newer than the EDIF file.
- One EDIF file and one ncf file for each HDL file.
- All HDL files will be re-synthesized if any one of the HDL files is updated and newer than the resulting EDIF file.

Using Guided PAR in the Xilinx 2.1i Software Release

Guided PAR can accelerate iterative implementations by reusing the unchanged sections from a previous implementation. This is advantageous because the software spends time generating implementations only for sections of the design that have changed. The guide process is more effective when the net names and instance names in the design remain constant between iterations, except for those specific parts of the design that are modified at the source level. This is the reason for using the top-down preserving hierarchy or bottom-up synthesis methodology.

Guided PAR depends on signal and component names remaining unchanged, and synthesis designs often have a low “match rate” when guided. Therefore, guided PAR is not recommended for most synthesis based designs, although if the design has followed the synthesis guidelines strictly, then you can benefit from this technique.

The guide file is an NCD file used as a template for placing and routing the input design. This is useful if minor incremental changes have been made to create a new design. To increase productivity, you can use your last design iteration as a guide design for the next design iteration; your output NCD file becomes the guide design file for your next iteration of the design.

Two command line options control guided PAR. The `-gf` option specifies the NCD guide file, and the `-gm` option determines whether exact mode or leveraged mode is used to guide PAR.

The guide design is used as follows:

- If a component in the new design has the same name as a component in the guide, it is placed where it was in the guide design.
- If an unnamed component in the new design is of the same type as an unnamed component in the guide design, and the two components have identical signals attached to them, the component is placed where the matching component was placed in the guide design.
- If the signals attached to a component in the new design match the signals attached to the component in the guide design, the pins are swapped to match the guide design, if possible.
- If the signal names in the input design match the guide design, and have the same sources and loads, the routing information from the guide design is copied to the new design.

When PAR runs using a guide design as input, PAR first places and routes any components and signals that fulfill the matching criteria described above. Then PAR places and routes the remainder of the logic.

To place and route the remainder of the logic, PAR does the following:

- If you have selected exact guided PAR (the **`-gm exact`** option), the placement and routing of the matching logic are locked. Neither placement nor routing can be changed to accommodate the additional logic.
- If you have selected leveraged guided PAR (the **`-gm leverage`** option), PAR tries to maintain the placement and routing of the matching logic, but changes placement or routing if it is necessary, to place and route to completion and achieve your timing constraints.

Some cases where the leveraged mode is necessary:

- You have added logic that makes it impossible to meet your timing constraints without changing the placement and routing in the guide design.
- You have added logic that demands a particular site or routing resource that is already being used in the guide design.

If you enter the `-gm` (guide mode) option, but do not specify a guide file with the `-gf` option, PAR is guided by the placement and routing information in the input NCD file. Depending on whether you specify exact mode or leveraged mode, PAR locks the input NCD file's existing placement and routing (exact mode), or tries to maintain the placement and routing, but modifies them in an effort to place and route to completion and achieve your timing constraints (leveraged mode).

Expectations for Guide

- Use guided PAR alone if reducing runtime is your main objective. If prior design iterations meet your timing requirements and the design changes are relatively localized, your design is likely a good candidate for taking advantage of the leverage-mode guide to reduce PAR runtimes.
- Incremental design changes should be limited to Boolean logic changes. Arithmetic changes involving comparator operations, addition, subtraction, multiplication, and division should be avoided because this drastically changes the carry chain structures.

If the leverage guided PAR can maintain greater than 90% of the component placement, the guide will offer better turnaround times. If the percentage drops much below the 75%-85% range, results will not generally be as good as if the whole design was re-compiled from scratch. Below 75%, it is definitely not recommended that you use a guide. If the match percentage is too low, it is usually best to restart the PAR run and not use guide at all.

You can judge whether your design is in the "good zone" by doing a trial guided PAR run in "place only" mode (such as `par -r`). The .par file will contain messages such `as "Successfully maintained guided placement of 855 out of 948 comps (mapped physical logic cells)`". The most important of these messages is the last one in the .par file.

Run PAR with the `-c` option set to `0` to prevent a delayed-base clean-up which tries to minimize resource utilization overall. However, the advantages achieved by using a guide file are impacted because the utilization of the resources of the design is defined within the guide file.

Specifying a Guide Design

Using the command line or script file you can specify the usage of a guide file from a previous run by specifying the "`-gf path/to/guide/file/file.ncd`" when running PAR. By default, Exact mode will be used, but to specify Leveraged mode use the "`-gm leverage`" option in addition to the `-gf` option. For example:

```
PAR -c 0 -gm exact -gf /home/test/M1/run1/par_out.ncd mapped.ncd
par_out_guided
```

Conclusion

Using Xilinx and Exemplar for Incremental Designing (ECO) can significantly increase your productivity.