Using Xilinx and Synplify for Incremental Designing (ECO)

XAPP164 August 6, 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(s), 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.

A flattened design (Top-down):

- Longer execution times because more memory is used. More memory means longer run times due to swapping.
- Presents run-time capacity problems for large designs. Blindly flattening the entire design may produce a single block of logic large enough to overwhelm the capacity of the synthesis tool.
- Calculating timing budgets between blocks is simplified because you rely on the synthesis tool optimization strengths.

A hierarchical design (Bottom-up):

- Supports team-based engineering. Allows several engineers to work on one design at the same time.
- Supports a mix of options for individual HDL files. Lets you mix synthesis options on different subsections of the design instead of using one set of options across the entire design. For example, you can synthesize modules on the critical path with tight timing constraints, and synthesize all other modules with looser default timing constraints.
- Supports incremental design changes.
- Design mapping into FPGAs may not be as optimal across hierarchical boundaries; this can cause inefficient device utilization and decreased design performance.
- Faster overall synthesis run times because it is possible to re-synthesize a small portion of the design as changes are made, instead of having to synthesize the entire design.
- Allows you to efficiently manage the design flow.
- Easier to debug (verification/simulation).
- Requires partitioning designs to optimize smaller blocks.
- Provides better pre-layout timing accuracy because you must provide details of timing budgets between blocks.
- Reduces design time by allowing design module re-use for future designs.

Managing Hierarchy through Synthesis

Of particular importance is how a synthesis tool handles a design's hierarchy.

Synplicity recommends synthesizing the whole design. Because Synplify is fast, you can easily and frequently synthesize the whole design. However, the benefits come from guiding the place and route tools to reduce compile times and to preserve timing, so name preservation is of high importance.

- Don't change logic in one module if another changes
- Don't propagate changes outside the boundaries of a module

To achieve hierarchy preservation, you can follow a bottom-up approach, or top-down approach with synthesis attributes that manage the hierarchy (context-based optimization). Because the Synplify resource sharing is restricted within the module boundaries, this may result in a larger design, but could be faster, and the logic may be packed better. For example, consider three cascaded 2-input AND gates (1 per LUT) vs. a 4-input AND gate in one LUT. This is logically the same, but the 2-input AND circuit consume more CLB resources than a 4-input AND circuit.

A bottom-up optimization creates hierarchy borders through synthesizing the design blocks individually and then creating a black_box for the blocks in the top-level. However, you must manually estimate and manage timing constraints on the design blocks. This can be time consuming and error-prone because you must pay specific attention to determine and keep track of the timing constraints for each block.

For bottom-up optimization, Synplify allows for full timing characterization of the black_box module/entity through the use of three types of timing attributes that are attached to the black_box definition in the HDL source code:

- Timing propagation delay through a black box (`syn_tpd<n>`).
- Timing setup delay required for input pins, relative to the clock (`syn_tsu<n>`).
- Timing clock to output delay through a black box (`syn_tco<n>`).

Overall, timing constraints can be specified in the UCF to cover the entire design.

A context-based optimization allows synthesis to retain hierarchy while extracting the timing context of lower-level modules, and the timing constraints are propagated downward through the lower levels. Timing delays on higher levels are used to derive constraints for lower levels.

Synplify provides two attributes to manage hierarchy for context-based optimization:

- The `syn_netlist_hierarchy` attribute, when `true`, maintains hierarchy in the EDIF netlist. When `false`, it gives a flattened netlist (though synthesis may still be maintaining hierarchy). The `syn_netlist_hierarchy` does not affect synthesis, just the output from synthesis.
- The `syn_hier` attribute controls hierarchy boundaries during synthesis. The `syn_hier` attribute affects synthesis, and hence, may also affect the output EDIF file.

Using the `syn_netlist_hierarchy` Attribute

When targeting Virtex designs, hierarchy can be preserved in the EDIF netlist with the `syn_netlist_hierarchy` attribute. To specify the `syn_netlist_hierarchy` attribute globally in the SDC file, use:

```
define_global_attribute syn_netlist_hierarchy 1
```

For non-Virtex designs, you should continue to accept the default of a flattened hierarchy (`syn_netlist_hierarchy = 0`) of the EDIF netlist. Setting this attribute to 0 will also cause the hierarchy in the Technology View of HDL Analyst to be flattened.

Using the syn_hier Attribute

The syn_hier attribute controls the amount of logic hierarchy flattening inside a module or instance. The syn_hier attribute is applied to instances, modules, or architectures. This option only affects the design unit to which it was specified. It takes one of five options: “soft”, “firm”, “hard”, “remove”, and “flatten”. To achieve hierarchy preservation of the design, this is accomplished with the "hard" option, which preserves the interface of the design unit with no exceptions.

The syn_hier attribute is specified on an instance-by-instance basis. To specify the syn_hier attribute in the SDC file, use:

```
define_attribute {sub_level1.block1} syn_hier {hard}
```

If the syn_hier attribute is applied on an instance that is replicated using the VHDL "generate" construct, the syn_hier attribute must be applied to the architecture of an entity, not to the instance. So no matter how you instantiate the entity, using a generate statement; otherwise, the syn_hier attribute applies to all instances of that entity.

These options control the way Synplify handles a design during optimization only. Regardless of which option is selected (remove, soft, hard, firm, or hard), Synplify will rebuild the hierarchy before the final netlist is created, ensuring that the netlist created by Synplify is efficient with regard to hierarchical boundary optimizations, and structurally as close as possible to the source code.

Guidelines for Synthesis

To benefit from a hierarchical approach, effective strategies are required to 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 a smaller individual blocks. The following guidelines are general and vary with design style, constraints, and device architecture.

Overall Rules for Partitioning

- Write each module/entity to its own file.
- Module/entity name must match the base filename.
- Keep I/Os at the top-level.
- Keep critical paths within a single block for synthesis because a critical path in one logic block allows the synthesis tool to optimize without any boundary structure imposed on the logic. Avoid paths that cross hierarchy boundaries; if a critical path is partitioned across boundaries, logic optimization is restricted.
- Place registers on the I/Os of module/entity cells. A good design practice is to make all input signals or all outputs signals registered at 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 a clock constraint. Also, registering the I/Os of your design hierarchy can eliminate any possible problems with logic optimization across hierarchical boundaries.
- 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.
- 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.
- Isolate FSMs. This recommendation enables easy state transformation and experimentation.
- VHDL designs must be compiled to the "work" library.

Top-down Rules for Partitioning

- Attach the `syn_hier` attribute to all levels of hierarchy in the design.

Bottom-up Rules for Partitioning

- The filename and entity/module declaration needs to be in lower case to match the names listed in the EDIF file.
- You must specify all clock ports on modules because synthesis will not to recognize a common clock signal. Synplify provides the `syn_isclock` attribute to infer the BUFG or you may need to instantiate a BUFG cell.
- Assign default values for all generics and parameters assigned to the entity/module. You cannot pass generics and parameters from a higher-level of hierarchy to the instance declaration.
- Add the black_box attribute to Verilog modules. For VHDL synthesis, Synplify automatically treats undefined instantiations as black boxes. A warning is issued to let the you know that a behavioral description for the component was not found.
- You may need to instantiate the STARTUP block because the bottom-up approach has the disadvantage for synthesis of not recognizing a single reset/set signal for GSR.

Running the Scripts

Provided are two Tcl scripts: bottom_up.tcl and top-down.tcl

Synplicity has extended the Tcl language with some synthesis commands so Tcl can be used as a scripting language to run Synplify. Tcl scripts have a .tcl extension and are executed in Synplify from the "`File -> Run Tcl Script`" menu command.

As part of the 5.0 release, batch mode operation is standard with a floating license. Please contact Synplicity to request this feature. The script file, synthesis.tcl, has been created for you to illustrate this feature. To run Synplify in batch mode, type the following.

```
synplify -batch design.tcl
```

This executes the Tcl script file and exits when finished. The files design.edf and design.srr are created. The flow through Synplify is fully defined by the commands in the script. The script can use any Synplify command including all Tcl and shell commands that can be found in the path.

Script Functionality

Synthesis control is directed by a set of Tcl variables within the script files. The following variables are used to customize the scripts:

`part_type -`
List the part as "device-package-speed." For example, `XCV100-BG256-4`

`library_files -`
List the VHDL packages, or Verilog libraries that are common to the individual HDL files.

`design_files -`
List user design files. The top-level module or entity/architecture is listed last

`syn_output_dir -`
Synthesis output directory

`frequency -`
Overall frequency. The default is `0`.

`fanout_limit -`
Overall fanout limit. The default is `100`.

`maxfan_hard -`
A hard max fanout limit. The default is `false`.

`default_enum_encoding -`
Sets the encoding style for the state registers of all state machines. Legal values are "`onehot`", "`sequential`", and "`gray`". The default is "`onehot`". However, you have the option of overriding the encoding style on an individual basis in the HDL file using the "`syn_encoding`" attribute.

`symbolic_fsm_compiler -`
Use the optimization features of Symbolic FSM Compiler. Default is `false`.

`resource_sharing -`
Perform automatic sharing of operator resources, including adders, subtractors, incrementors, and decrementors. Default is `true`.

Bottom-up Script Behavior

1. Overall synthesis time is reduced if the compiled EDIF file has a date newer than the date of the source HDL file, then synthesis recompilation is unnecessary. Otherwise, if the date is newer than the EDIF file, recompilation of the HDL is performed.
2. Recompilation is also forced if any user defined library or constraint file has been modified more recently than the source HDL file.
3. An EDIF file is generated for each HDL file listed in the `design_files` list.
4. SDC (Synplify Design Constraints) files are loaded automatically when the basename of the HDL file matches the basename of the corresponding SDC file. There is a 1-1 ratio of a SDC file to a HDL file.

Top-down Script Behavior

1. Recompilation is forced whenever the script is invoked.
2. Only one EDIF file is generated for the design project.
3. SDC (Synplify Design Constraints) file is loaded automatically when the basename of the top-level HDL file matches the basename of the corresponding SDC file. There is only one SDC file, and it is associated with the top-level HDL file.

Using Xilinx Alliance 2.1i with Guiding

When you re-synthesize modules, you will typically cause signal and instance names in the final netlist to be significantly different from the netlist obtained in earlier synthesis runs. This occurs even if the source-level Verilog or VHDL code only contains a small change. Because guided PAR depends on signal and component names, synthesis designs often have a low "match rate" when guided. Therefore, guided PAR is not recommended for most synthesis-based designs, although if you have 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; that is, 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 certain site or certain routing resource, and that site or routing resource 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 comp 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 (e.g. `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 since the utilization of the resources of the design is defined within the guide file.

Conclusion

Using Xilinx and Synplify for incremental designing (ECO) can significantly increase your productivity.