



XAPP404 (v1.2) April 20, 2001

Xilinx Alliance 3.1i Modular Design

Introduction

With the availability of large Virtex™ devices, designers should consider partitioning a single, large design into several modules. Design partitioning can provide several benefits, including:

- **Complexity Management** – Large designs are generally difficult to manage when kept as a single, monolithic entity. By dividing a design into smaller pieces, each piece can be understood and implemented separately
- **Work Flow Management** – The work flow of a large design can be managed more effectively if portions of the design are assigned for independent implementation by individual team members
- **Module Insulation** – If a design is partitioned, finalized static portions of the design are not affected by timing, layout, and other changes made to dynamic modules
- Design partitioning involves additional costs as well as benefits. These costs include:
- **Planning** - Additional overhead costs are necessary to correctly partition a design
- **Communication** – Additional design costs are incurred to implement communication between design partitions

Increased investment in planning and communication is typical of large digital design projects (customers accustomed to ASIC design flows should be familiar with the cost-benefit principles involved).

To ease the task of design partitioning, Xilinx provides a new **Modular Design** feature in Alliance™ Series 3.1i. The remainder of this application note describes recommended design entry, implementation and simulation flows when working in the Alliance Series Modular Design context.

Note: Appendix D lists known limitations associated with the initial release of Modular Design. These limitations will be addressed in future Alliance Series software releases.

Of course, whenever a new, complex design modality is used for the first time, a learning curve must be overcome. Therefore, it is particularly important to make sure that the design to be partitioned is an appropriate candidate for Modular Design. Modular Design is most appropriate when:

- The design is large and can be logically divided into self-contained pieces (see a further discussion of this point, below)
- The design consists of smaller, self contained sub-designs that will be implemented by different members of a design team

It is equally important to recognize when *not* to use Modular Design. Modular Design is not appropriate for implementing the following types of designs:

- An existing, large ASIC design that did not start with Modular Design concepts in mind. If the design began without an effective functional and physical partitioning of the design into appropriate modules, Modular Design will add time to the design flow, and provide few benefits
- A single, heavily-interconnected design that seeks to achieve the highest possible speeds. The overhead necessary to implement a Modular Design flow precludes obtaining the fastest clock speeds

To maximize the benefits of Modular Design, it is critically important to properly structure and partition the design at the HDL source level. Specifically, the design must be partitioned into separate modules. This requirement leads to the following two fundamental questions:

1. What exactly is a *module*? and more importantly,
2. What is a *properly designed* module?

While there are no rigid definitions to apply to answer these questions, there are guidelines that can aid proper module creation. Ultimately, designers develop their own set of rules and guidelines for module creation, but these initial guidelines should provide a helpful starting point. Generally, well-behaved modules should comply with the following rules:

- **Well-Defined Ports** – A port is a connection that goes into or out of a module. Typically, a port is connected to a wire or signal defined at the top level of the HDL source
- **Minimal Number of Ports** – A module with a small, rather than large, number of ports or connections to external logic is preferable. Self-contained modules with minimal functional dependence on outside modules can be optimally implemented
- **Minimal Global Logic** – Global logic is logic that is not **[not?]** distributed on the target chip. Examples **[“of global logic”?]** include I/O pins leading onto or off the chip, DLLs, and other global clock-modification resources
- **Registered Module Outputs** – Output registration leads to efficient logic packing during the implementation mapping phase
- **No Chip-Location Dependencies** – For example, a module should not require a BRAM to be located in the column adjacent to the module
- **Minimum Number of Modules** – As a general rule, the number of modules should equal the minimum number of engineers necessary to implement the entire design

Modular Design requires standard HDL synthesis coding practices. Designs that deviate from the norms are not suitable candidates for a Modular Design flow.

Design Entry and Synthesis Flow

This section explains how to take a modular design through FPGA Express/FPGA Compiler II, LeonardoSpectrum, or Synplify for input to the Xilinx tools.

Note: FPGA Express and FPGA Compiler II exhibit equivalent behavior with respect to the issues addressed in this application note.

Creating the Top-Level Netlist

Before beginning a modular design, designers must reach a consensus on a top-level design. The top-level design should include all design modules or partitions, global logic (such as clock resources), and logic to connect modules to each other and to I/O ports.

Each module is instantiated and synthesized separately, as explained, below, and exemplified in Appendix A. The top-level code in Appendix A instantiates three modules: `module_a`, `module_b`, and `module_c`, which are named `instance_a`, `instance_b`, and `instance_c`, respectively. The examples also include nets and logic that connect modules to other modules and to ports. Each module is instantiated as a ‘black box,’ with only ports and port directions defined.

Top-Level Code Guidelines:

- **All lower-level modules must be declared to define port directionality and bus width.** VHDL synthesis requires component declarations for all instantiated components in the HDL code. The component can be declared in the code or in a library package included in the HDL source. An undeclared component will produce a synthesis tool error

Verilog synthesis requires declarations for user modules only, not library primitives (see the example in Appendix A). If user modules are defined and described in the same project, module declarations are unnecessary—this is the case, for example, if synthesis tools can produce multiple EDIF netlists from a single project. However, if a user module is described in a different project, or if it is a CORE Generator module, then a module declaration is required.

Failure to assign port directions will result in the following error:

```
ERROR:NgdBuild:604 - No driver was found for logical net 'X' for this modular design.
```

Ports with an undefined direction were found connected to the net. Please assign a direction to these ports.

- **All lower-level modules must be synthesized as black boxes.** Black-box instantiation may require the use of a synthesis tool directive. If lower-level modules are not synthesized as black boxes, the resulting flat, non-modular design will produce an error during the ngdbuild initial mode, as follows:

```
ERROR:Ngd:819 - Modular design: initial mode must have at least one inactive module
```

```
ERROR:NgdBuild:558 - Modular Design cannot be annotated
```

- **Modular Design supports two levels of module hierarchy.** Modular Design currently supports a top level with modules instantiated within the top level. It currently does not allow a module to include another module
- **Modular Design does not allow multiple instantiations of a module.** Each module instantiation must have a separate module definition, even if module instantiations will use the same port definitions and functions

Example of *acceptable* coding if module_a and module_b are functionally identical:

```
--VHDL
instance_a: module_a port map(...)
instance_b: module_b port map(...)

//Verilog
module_a instance_a (...)
module_b instance_b (...)
```

Example of *unacceptable* coding (multiple instantiation of same module):

```
--VHDL
instance_a: module_a port map (...)
instance_b: module_a port map (...)

//Verilog
module_a instance_a (...)
module_a instance_b (...)
```

Although the unacceptable examples, above, conform to standard VHDL and Verilog coding practice, they are incompatible with the current Modular Design release. A separate synthesis is not required to produce an EDIF netlist for module_b: you can copy module_a.edf to module_b.edf. However, all implementation steps should be run separately for module_a and module_b, as explained below.

- **I/O registers must be inferred in top-level code.** This is a limitation of the current Modular Design release. To meet timing requirements, it may be necessary to move registers out of top-level modules to infer output registers. Otherwise, registration of module outputs is recommended.
- **3-state buffers that drive the same net/bus should be inferred at the top-level with local control logic.** Isolating this eases the mutually-exclusive requirement for the control *[further clarification? what's "this" and how does the mutually-exclusive requirement of the control tie to the caption?]*
- **If 3-state signals are outputs of a lower-level module, they must be declared in the HDL code as "inout" signal types in both the top-level component declaration and the module-level port map.**
- **Use meaningful signal names to connect to module ports or between modules in the top-level code.** Using the same name for the signal and its associated port is recommended since top-level signal names are used in the back-annotated simulation

netlist (see the *Simulation Flow* section, below, for additional information)

Creating Lower-Level Modules

Xilinx Alliance 3.1i Modular Design requires a separate EDIF netlist for each lower-level module. To generate EDIF netlists for a modular design, follow these suggestions and guidelines:

1. Most synthesis tools generate only one EDIF netlist for each project. To meet Modular Design's separate-netlist requirement, lower-level modules must be synthesized separately from the top level, and a separate project must be created for each lower-level module as well as for the top-level. (Exception: In LeonardoSpectrum, it is possible to create multiple EDIF netlists for one project. This technique is described in *Vendor Specific Notes* in Appendix C.) The EDIF netlists of lower-level modules must be *identical* to their module names—for example, `module_a.edf`, `module_b.edf` and `module_c.edf`. Do not use module instantiation names—for example, `instance_a`, `instance_b`, and `instance_c`—or `ngdbuild` will not be able to match the netlists to the module **[OK?]** names specified in the top-level EDIF
2. Modules must be synthesized *without I/O insertion*. This option is available in most synthesis tools (see *Vendor Specific Notes* in Appendix C).
3. Although the recommended practice is to declare external I/Os in the top-level, it is possible to include external I/Os in a module without modifying the top-level code. These I/Os should not directly connect to module ports since they cannot **[“currently” was inserted here; does it refer to the current release of Modular Design or to the synthesis tools? clarification is needed]** be processed by the **[“synthesis”?]** tools. For example, a designer may wish to add temporary external I/Os in the module for simulation. This can be accomplished by explicitly instantiating `IBUF/IBUFG/BUFGP` as well as `OBUF` connections (Appendix B contains sample module code that includes external I/O instantiations)

Constraining at the Module Level

Global and top-level constraints can be entered with synthesis tools when synthesizing the top level. However, most module-specific constraints must either be 1) manually entered, or 2) created using the Floorplanner through a UCF file. Module-specific constraints include constraints with hierarchical reference and area constraints (for a further discussion, see the *Implementation Flow* section, below).

Note: Some synthesis tools write default constraints for a synthesis project to an NCF file. To avoid conflict between NCF files, disable this NCF-writing option when synthesizing lower-level modules.

Implementation Flow

Since the initial release of Modular Design does not support the Design Manager/Flow Engine, the design flow described below is implemented with command line options to batch tools. The steps may seem involved, at first, but it is important to realize that many steps are only necessary when creating and modifying design partitions. Further, in a modular design, design tasks are divided among team members, and different steps can be performed by different team members at different times.

There are three primary phases in a Modular Design flow (a “-modular” switch with arguments has been added to the `ngdbuild` tool to allow a designer to specify the current Modular design phase). The three Modular Design phases are:

1. Initial Budgeting

This phase begins when a design, which has been partitioned with HDK tools, is brought into the Xilinx FPGA tools. During this phase, resources are allocated, and modules are sized, positioned, and connected to each other.

2. Active Module Implementation

During this phase, defined modules are implemented using information generated by the initial budgeting phase (initial budgeting module size and location information). The full suite of FPGA design and implementation tools are available for implementing each module.

3. Final Assembly

During this phase, the final design is assembled. Final logic assembly is guided by information generated by the active module implementation phase. The guided-assembly process is fast and efficient, allowing [*“synthesis”?*] tools to treat the whole design as an entity for constraint resolution, buffer alignment, and other operations.

Each of the three modular design phases is described in detail below.

Initial Budgeting Phase

The initial budgeting phase is at the top level of the design flow and is normally performed or overseen by the team leader. The purpose of the initial budgeting phase is threefold: 1) to position global logic—that is, logic not contained in a lower-level module, 2) to size and position each module (defined in the HDL netlists) on the target-implementation chip, and 3) to position the input/output ports (nets that flow into or out of a module) of each module so that later module implementations are aligned correctly. The team leader typically performs this critically important phase of the flow. Once the initial budgeting information is defined and phase-two implementation begins, revisions to the initial budgeting information are difficult and time-consuming.

During phase one, a target device for the design is specified (device, package and speed). Module definitions are also decided. It is recommended that all global logic and nets be placed in the top-level design, including all I/Os, clock nets, DLLs, RAMs, and other resources that are not evenly distributed across the target chip. By doing this, all of the top-level resources can be placed during this phase and used later in the active module implementation phase to provide modules with maximum context.

The main steps of this phase are 1) create an NGD file of the top-level design without module implementation information, 2) add timing constraints 3) use the Floorplanner to position global logic, size and position each module on the target chip, and position the module ports for correct alignment. Each of these steps is described below. It is assumed that the netlist for the top-level design is contained in the "top.edf" file (all files used and produced during this phase should be placed in a directory that phase-two module implementers can access).

1. Create an NGD file of the top-level design without module implementation information

This is the first step in the design flow that uses the `ngdbuild -modular` switch, which indicates that a Modular Design flow is in progress. The “initial” argument is passed with the `-modular` switch to indicate to `ngdbuild` that this is the initial budgeting phase of the modular design. The netlist describing each module should not be available to `ngdbuild`—this ensures that each module will be treated as a black box for the active module implementation phase. To perform this step, run the following command:

```
ngdbuild -modular initial top.edf
```

Two output files are created: `top.ngo` and `top.ngd`. The `top.ngo` file is used in the remaining steps of this phase and in all steps in the active module implementation phase. The `top.ngd` file is not used in any remaining step of the modular design.

2. Add timing constraints

Constrain any global clocks with a period constraint. PERIOD and IO timing constraints for all clocks and I/Os can be placed at the top level. After timing constraints are added, `ngdbuild` must be used to place these constraints in the `top.ngd` file, as follows:

```
constraints_editor top.ngd
```

These timing constraints will be saved in the top.ucf file, which is used by all steps in the active module implementation phase.

Note: If there are no clock loads at the top-level design, the clock net will not appear in the Constraints Editor. If this is the case, another method must be used to define PERIOD and IO timing constraints—for example, by using NCF constraints from the front-end tool. Another approach is to add a dummy register at the top level, driven by the clock net. This technique allows access to the clock net in the Constraints Editor. Since the dummy register does not have a load, it will be removed during the mapping phase.

3. Use the Floorplanner to position global logic, size and position each module on the target chip, and position module ports for correct alignment.

It is recommended that all logic that is present at the top level of the design be constrained to fixed locations in the top-level floorplan. As explained in the *Design Entry/Synthesis* section, below, HDL source should provide that all 3-state drivers reside at the top level of the design. This requirement is necessary to ensure that all BUFT symbols driving the same long-line net are manually placed in the same row.

Area assignment consists of sizing and positioning the target area for each module implementation on the target chip. It also includes positioning module ports for correct implementation alignment (any net that connects into or out of a module is considered a port net). Each module/port pair is indicated in the tools as a piece of “pseudo-logic,” which can be positioned to pull the logic within a module to the desired location. Pseudo-logic is not part of the final design, and only used during module implementation. At this point in the design flow, module size must be estimated (Floorplanner currently does not specify required module size, therefore this information must be estimated from other sources). Each module should be sized and positioned on the chip with the **Floorplan->Assign Area Constraint** menu item.

Next, each port of each module must be placed so buses align properly. For example, a net that connects module A and module B has two ports that must be constrained. (Again, these ports are “pseudo-logic,” used only to force alignment of internal module logic—they will be removed from the design during the final assembly phase.) The port for module A should be positioned outside the previously assigned boundary of Module A, and the port for Module B should be positioned outside the previously assigned boundary of Module B. If Module A is positioned to the left of Module B, the ports of Module A that connect to Module B should go in the column just to the right of boundary of Module A (it is permissible to place the ports associated with Module A into the area defined for Module B to avoid leaving empty rows and columns between modules).

The **"Floorplan->Distribute Options-**" menu item can be used to obtain Floorplanner port-placement assistance. Use the **Floorplan** option to position all unplaced ports. Use the **Enable** option to place ports as module areas. Use the **Ripup** option to remove all auto-floorplanned ports.

Note: Port placements made by Floorplanner may not be the best placements for a design—they should be reviewed before the design enters the active module implementation phase.

The commands used to perform the tasks described in this step are:

```
Floorplanner top.ngd
File->Read Constraints: top.ucf
Open "Primitives" hierarchy
Place each component
Select "Module" hierarchy
Floorplan->Assign Area Constraint
Use left mouse button to draw rectangular area
Floorplan->Distribute Options
Floorplan
Open "Module hierarchy"
Review and/or place each port (pseudo*) component
File->Write Constraints: top.ucf
```


Exit

Placement constraints are saved in the top.ucf file. This file will be used during the active module implementation phase.

In summary, the following commands are used during the initial budgeting phase of a modular design:

- **ngdbuild –modular initial top.edf**
- **constraints_editor top.ngd**
- **floorplanner top.ngd**

At this point in the modular design, all global logic should be positioned, each module should be sized and positioned, and the ports for each module should be positioned. To avoid having to return and repeat this initial phase later in the design flow, perform a final review to make sure modules are properly defined and all modules and ports are correctly positioned.

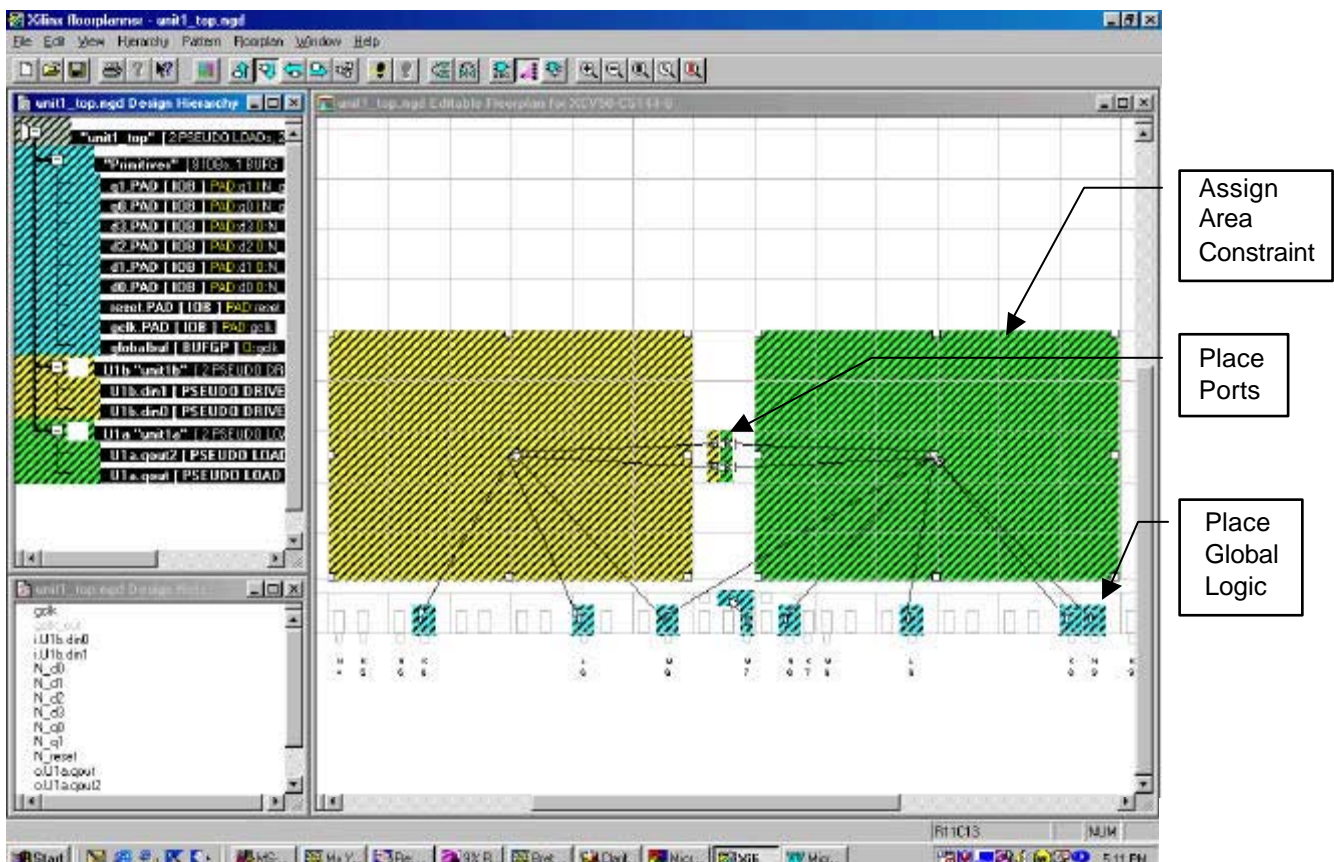


Figure 1: Example of Floorplanner Use During the Initial Budgeting Phase

Active Module Implementation Phase

The goal of the active module implementation phase is to implement each module defined in the initial budgeting phase. This phase is performed for each module. Place the files for each module in a separate directory—these module files will be used during the final assembly phase. Different team members—called “module implementers”—should be assigned separate modules, and each member should work with a separate set of module files. During this phase, each set of module files normally is invisible to the team leader, who works at the top level of the design (see the *Initial Budgeting Phase* description, above). Typically, a team leader will not also work as a model implementer unless the team has an insufficient

number of implementers to handle all design modules. Once this phase is complete (all modules have been implemented), the team leader can begin the final assembly phase.

Each module uses the ngo and ucf files produced in the initial budgeting phase. Each module is implemented independently from all other modules, and each uses the top-level design as a testbench to provide context during module implementation. Further, during the implementation of a module, the top-level design treats each module as the only module in the design. It is important to understand this point to understand file usage during module implementation. For example, when a module is implemented, the implementation output filename is taken from the top-level filename (top.ncd is the name of the output file), not from the name of the module being implemented. Module implementation information is contained in this output file, as well as top-level design implementation information. During this phase, the ngo file from the top level design will be used in read-only mode, while the ucf file from top-level design will be used in read/write mode. To simplify file usage, logic below the top.ngo file is referenced directly by the top.ucf file, which is copied into the module implementation directory to allow for modifications not needed by the top-level design.

The full suite of Xilinx implementation tools is available for individual module implementation, including the simulation of the module independently from the top-level design. This means that designers can use any of the map and par command line switches as well as the Floorplanner to achieve the desired implementation. The fpga_editor also can be used, but care must be taken not violate any of the area constraints or placement information previously generated. Simulation can be used to verify that each module meets its specifications. The mapper and par have been modified to recognize the active module implementation phase and to maintain all logic and routing within defined module boundaries. To guarantee that each module implementation is compatible with other module implementations, logic optimization does not occur across a module boundary defined in the top-level design.

The primary steps of this phase are 1) create an NGD file for the module, 2) if necessary, add timing constraints specific to the module, 3) map module logic, 4) place and route the module, 5) if necessary, floorplan placement information, 6) if desired, simulate the module, and 7) publish the module implementation information to the team leader.

The work for this phase should be performed in the directory created by the module implementer. It will only be accessible to the team leader during the final assembly, after it has been explicitly published to a central directory.

Here are the active mode implementation steps (it is assumed that the netlist for the module is contained in the file "moda.edf" after synthesis, and that the information for the top-level design, named "top.*," is contained in the "top_directory"):

1. Create an NGD file for the module

As explained above, the NGD file for a module will be given the name of the top-level design, and will contain implementation information for both the top-level design and the module. A copy of the top-level UCF file, which contains constraint information for the top-level design, will be used (the module implementer can append any module-specific constraints to this file).

To perform this step, first create an empty directory, then copy the top.ucf file into the directory. Next, copy or create the EDIF moda.edf file, which describes the logic of the module. Then, run ngdbuild using the `-modular` switch with the "module" argument to indicate that this is the active module implementation phase of the modular design. Also, the name of the current module should also be given with the `-active` switch, to allow future tools to use *[what?]*. Commands for this step are as follows (the example assumes moda.edf has been created, and top.ucf has been copied to the directory): ***[but doesn't the first command, below, perform the copy operation?]***

cp top_directory/top.ucf . (UNIX)

copy top_directory/top.ucf (PC)

ngdbuild -modular module -active moda -uc top.ucf top_directory/top.ngo

The output file, top.ngd, contains the implementation logic for the top-level design *and* the module.

2. Add timing constraints specific to this module

The Constraints Editor can be used to add any module-specific constraints. These are the timing requirements for the design that the module must meet. This step can be skipped if module constraint information is not needed. For module-level timing constraints, the user can enter a PERIOD for any local clocks and port timing by using the modular version of the OFFSET constraint. Port timing allows the user to specify the path delay requirement for data to propagate from the port to setup at the register (for inputs) or from the clock pin of the register to the port (for outputs). The Constraints Editor, which shows each port on the Ports Tab, can be used to create the appropriate OFFSET group/constraint. If the user is editing the UCF manually, a TPSYNC point must be created using the port net name. Several ports with equivalent timing requirements and clocks can use the same TPSYNC name. Next, an OFFSET constraint is needed to correlate with each TPSYNC group. The syntax is similar to OFFSET PAD GROUP syntax, except that the TPSYNC group name is used. An OFFSET constraint on a module port must be relative to the actual clock pad net—most likely located in the top-level design—not to the clock port on the module. Port-to-port timing can be specified using the TPSYNC names in a FROM TO constraint. Remember: the top.ngd file contains the constraint information referenced above..

The commands used here are:

constraints_editor top.ngd

ngdbuild -modular module -active moda top_directory/top.ngo

The timing constraints will be saved in the top.ucf file, then folded back into the top.ngd file for use by other implementation tools.

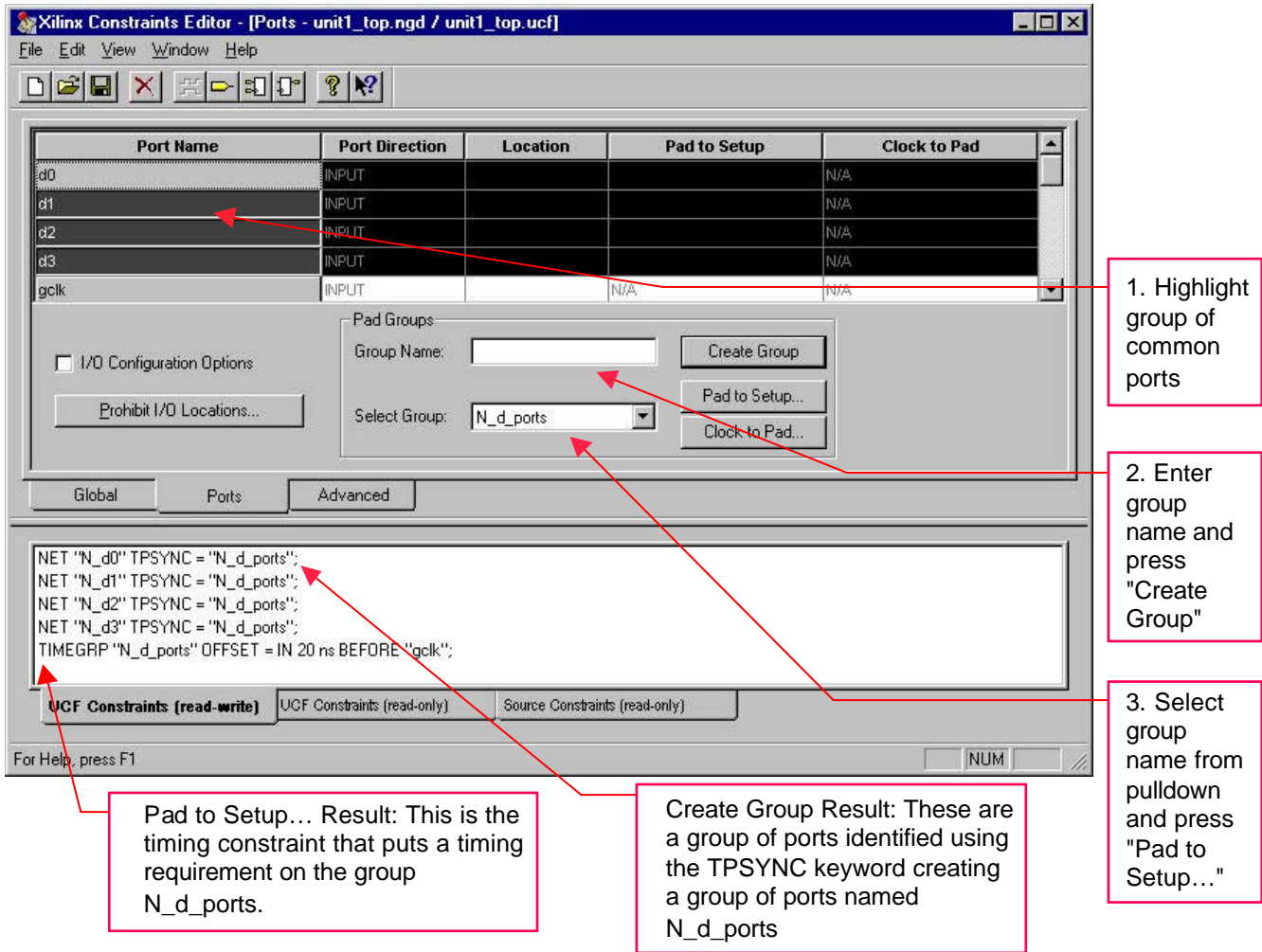


Figure 2: Example of Constraints Editor Usage

3. Map module logic

Since range constraints and active module information have been saved in the top.ngd file, additional switches are not required to map this design—the current mapping switches can be used to obtain the desired results. The map command identifies the NGD file as a module implementation, then scans the NGD file to identify the logical block with the name of the active module. Next, it scans each port on the active module, adding pseudo logic (discussed earlier) to port nets to provide drivers/loads. The pseudo logic will not be propagated into the final design. The command used here is:

map top.ngd <additional_map_switches>

Use <additional_map_switches> to specify any of the allowable mapper switches. The size and position of each module (its range constraints) are expanded as a range for each slice within the module, then written to the top.pcf output file. All information for the active module, as well as context information for the ["top-?"] level design, is written to the top.ncd output file.

4. Place and route the module

Since range constraints and active module information have been saved in the top.ncd file generated by the mapper, additional switches are not required for par to place and route the design—the current par switches can be used to obtain the desired results. All pseudo logic will be placed outside of the range defined for the module boundary. The command used here is:

```
par top.ncd output_ncd_filename <additional_par_switches>
```

Use <additional_par_switches> to specify any of the allowable par switches. The *output_ncd_filename* is the name to be given to the placed and routed NCD file, which is distinct from the mapped NCD file. This argument can be used, below, to quicken the publish step. If the area described for the module is too small or incorrectly sized to contain the physical logic for the module, the top.ucf generated by the initial budgeting phase must be regenerated. This means the initial budgeting phase must be repeated, and previously implemented modules may need to be reimplemented. **[more information? when is reimplementation necessary?]**

5. If necessary, floorplan placement information

If reports indicate that the module implementation does not meet timing constraints or is otherwise unsatisfactory, Floorplanner can be used to explicitly reposition logic. All Floorplanner commands are available to work with the module, but Floorplanner will report any attempt to violate the assigned range constraints for the module. After floorplanning is complete, the map and par steps must be repeated. The commands for this step are :

```
floorplanner top.ncd
```

```
map top.ngd ...
```

```
par top.ncd output_ncd_filename .....
```

Note: In most cases, this step will not be necessary.

6. If desired, simulate the module

There are two simulation modes available at this stage in the design flow. The first is normal simulation using the top-level design for context. The top-level design can be back-annotated and completely simulated. The commands to perform this simulation are the usual commands for correlated back annotation, as follows:

```
ngdanno -o top.nga <output_ncd_filename> top.ngm
```

```
ngd2ver top.nga
```

-or-

```
ngd2vhdl top.nga
```

The advantage of this approach is that the logic in the top-level design is included in the simulation. The disadvantage is that inactive modules are undefined, and signals connected to module ports are left dangling. Therefore, it normally is necessary to probe and/or stimulate dangling signals to obtain meaningful simulation results.

The second simulation mode is to simulate the module independent of the design context. This can be accomplished with following commands:

```
ngdanno -o mod.nga -module <output_ncd_filename>
```

```
ngd2{ver|vhdl} mod.nga
```

In the resulting simulation netlist, the top-level ports will be those of the module itself. This netlist can be instantiated in a testbench that exercises the module only (see the *Simulation Flow* section, below for additional information on simulation modes and operations).

7. Publish the module implementation information to the team leader

The final step of the active module implementation phase is the publication of the generated module implementation files to the team leader for use in the final assembly

phase. These files should be published to the PIM directory (the Previously Implemented Module directory created by the team leader) using the pimcreate executable. During this step, the module implementation files should be renamed to the name of the implemented module. Although not required, the pimcreate -ncd switch can be used to quicken the publish operation (this switch avoids having pimcreate load multiple NCD files to determine the required information. **[may be helpful to specify what type of information pimcreate normally looks for?]** The command to publish files to the team leader (*pim_path*, below, is the path to the PIM directory) is :

pimcreate *pim_path* -ncd *output_ncd_filename*

In summary, the following commands are used to perform the basic steps in the active module implementation phase of a modular design :

- **ngdbuild -modular module -active moda *top_directory/top.ngo***
- **constraints_editor top.ngd**
- **ngdbuild -modular module -active moda *top_directory/top.ngo***
- **map top.ngd ...**
- **par top.ncd *output_ncd_filename* ...**
- **floorplanner top.ngd**
- **map top.ngd ...**
- **par top.ncd *output_ncd_filename* ...**
- **ngdanno -module *output_ncd_filename***
- **pimcreate *pim_path* -ncd *output_ncd_filename***

At the conclusion of a single pass through the active module implementation phase as described, above, the implementation of one design module is complete. Each step in this stage must be repeated to implement each additional module in the modular design (an additional pass through this phase is required for each additional module).

Final Assembly Phase

The team leader is responsible for this phase of the design. It can begin once all implementation modules have been published to the team leader. The goal of the final assembly phase is to produce a complete design from the information generated in the previous two Modular Design phases. If the overall design meets its timing constraints and simulation values, the final assembly phase can begin. However, if overall design criteria are not met, the active module implementation phase and, in rare cases, the initial budgeting phase must be repeated before the design is ready for the final assembly phase.

This phase first builds a complete NGD file from the NGO file for the top-level design together with the NGO files for each of the referenced modules. Then, the complete design is mapped and routed using guide information contained in the NGM and NCD files published for each of the referenced modules. The use of guide information at this step greatly speeds the overall implementation run time. Since all top level-logic has been positioned, and each module and its ports have been sized and positioned, the design should assemble as specified. However, even if module assigned areas do not overlap, in some cases resource contention among modules can occur due to module use of global logic or routing resources. Also, even if each module meets its timing constraints, the overall design may not meet its timing constraints due to additive delays. If either of these conditions occur, or if the overall design is otherwise unacceptable, one or more modules will need to be reimplemented before the final design can be assembled. Although it is possible to use tools during this phase to directly constrain or manipulate resources contained in a module, this technique is not recommended since it renders published module information invalid.

Although the work for this phase can be performed in the directory where the initial budgeting phase was performed, doing so is not recommended since the assembly may change top.ucf, the top-level constraints file. If this occurs, the file cannot be used for future module implementation. To avoid this problem, a separate directory should be created for this phase. To

prepare the final assembly directory, copy each of the files generated in the initial budgeting phase to it.

Each step in the final assembly phase is described, below. The information in each step uses the following assumptions and conventions: 1) information for the top-level design is contained in "top.*" files, 2) the PIM directory is located at *pim_path*, and 3) each module is named "moda," "modb," and so forth.

1. Create the NGD file for the entire design

Since all NGO files for referenced modules have been published into the PIM directory, ngdbuild can be used to assemble all blocks into a complete design. Again, none of the pseudo logic used in the active module implementation phase appears in any of the associated NGO files. Consequently, this extra logic will NOT be added to the full design. Ngdbuild is run using the "assemble" argument to the "-modular" switch to indicate this is the final assembly phase of the modular design. The path to the PIM directory must be specified as well as the names of each module implementation. Currently, each module name must be specified, but future software release functionality will be added that implies the names of all modules if no module name is specified. The following command assumes there are two published modules in the modular design, "moda" and "modb:"

```
ngdbuild -modular assemble -pimpath pim_path -use_pim  
moda -use_pim modb top.ngo
```

This command reads the previously created top.ucf constraints file in the current directory. Command output is written to the top.ngd file, which represents the logic of the full design. The specified PIM information will be written into the top.ngd file for future tool use.

2. Map the full design

The mapping step uses the PIM information in the input NGD file to guide the mapping of all associated PIM logic. Use of guide information allows any top-level constraints pushed into module implementations to be correctly processed. Since the PIM information was placed into the top.ngd file during the previous step, no additional flags are required to perform the mapping process (use of mapping flags is optional). The command to use for this step is:

```
map top.ngd <other_map_flags>
```

Use <other_map_flags> to specify any of the available mapping flags. Command output will be mapped to the top.ncd and top.ngm files. Also, PIM information found in the top.ngd file is propagated into these files for later use. The map report, top.mrp, can be examined to see how guide information for each module was processed.

3. Place and route the full design

The placement and routing of the full design can now be performed. This step use the previously generated PIM information as guide information, which greatly speeds implementation time. Also, the use of guide information results in tool optimizations (though not across module boundaries) that increase the quality of the overall design result. Since previously provided PIM information is contained in the top.ncd file, no additional par flags are required (use of par flags is optional). Here is the command to use to perform this step:

```
par top.ncd top_impl.ncd <additional_par_flags>
```

The name assigned to the placed and routed NCD output file is "top_impl." Optional par flags can be specified in <additional_par_flags>. The par report file, top_impl.par, can be consulted to view the performance of the overall design together with associated guide information.

Note: Currently, the guide information in the par file incorrectly takes into account the failure to guide pseudo-logic in the PIM guide files, but not in the top.ncd file. This incorrect functionality should be ignored (it will be fixed in a future software release).

4. Simulate the full design

The final step in the Modular Design flow (other than bitstream generation) is the simulation of the full design to verify that it meets its timing specifications. This process is described in the next section.

Simulation Flow

Functional and timing simulation can be performed on each module during the active module implementation phase. Simulation of the entire design can be performed during the final assembly phase.

Simulating the Module

A module can be simulated during the active module implementation phase. There are two simulation methods:

1. A module can be simulated directly, independent of the top-level design. Independent module simulation is performed with the following commands:

```
ngdanno -o <module_name>.nga -module <toplevel>.ncd
```

Example: **ngdanno -module -o module_a.ngo top.ncd**

For VHDL, run:

```
ngd2vhdl -te <module_name> <module_name>.nga
```

Example: **ngd2vhdl -te module_a module_a.nga**

For Verilog, run:

```
ngd2ver -tm <module_name> <module_name>.nga
```

Example: **ngd2ver -tm module_a module_a.nga**

Notice the use of the `-te` and `-tm` options in the VHDL and Verilog examples, respectively. These options are used to rename the top-level entity to `module_a`, as indicated. If these renaming options are not used, the resulting VHDL/Verilog entity/module will be named "top."

The resulting simulation netlist will contain only module-level logic and ports. This netlist can be instantiated in a testbench that exercises just the module.

Current Limitations:

All ports and internal signal names appear in the back-annotated netlist in terms of the top-level netlist. The ports are named after the top-level signals they connect to, and the internal signals have the instance name tagged in front of them. For example, port `B2A_IN` of `module_a` will be named `B2A` (`B2A` is the top-level signal that connects to port `B2A_IN`). The internal signal `Q0_OUT` in `module_a` will appear as:

```
signal INSTANCE_A_Q0_OUT : std_logic; -- In VHDL
wire \instance_a/Q0_OUT ; // in Verilog
```

Timing simulation will reflect the timing of components within the module. However, delay and timing values of module ports should be ignored until a complete design simulation is performed. Boundary timing is meaningless when simulating a module since port loads and drivers are not known.

2. The top-level design with its active module can be back-annotated and completely simulated. This simulation can be performed using the standard commands for correlated back annotation, as follows:

```
ngdanno -o top.nga <toplevel>.ncd top.ngm
```

VHDL:

```
ngd2vhdl top.nga
```

Verilog:

```
ngd2ver top.nga
```

The advantage of this approach is that top-level design logic is included in the simulation. The disadvantage is that inactive modules will be undefined, and the signals connected to their ports will be left dangling. Therefore, it will be necessary to probe or stimulate (or both probe and stimulate) dangling signals to yield meaningful simulation results.

In VHDL, internal signals can not be driven from the testbench, but some simulation tools allow access to these signals from a script/command file or from the GUI. Please refer to tool documentation for more information.

In Verilog, users can access internal signals from the testbench as well as from a script or command file.

Simulating the Entire Design

The entire design can be simulated in the assembly mode. The commands are:

```
ngdanno -o top.nga <toplevel>.ncd top.ngm
```

For VHDL:

```
ngd2vhdl top.nga
```

For Verilog:

```
ngd2ver top.nga
```

There are no restrictions for simulation of the full design during the final assembly phase.

Appendix A – Synthesis Examples for the Top-Level Design

VHDL Example

```
library IEEE;
use IEEE.std_logic_1164.all;
entity top is port (ipad_dll_clk_in: in std_logic;
    dll_rst : in std_logic;
    top2a_c: in std_logic;
    top2b: in std_logic;
    obuft_out: out std_logic;
    mod_c_out: out std_logic;
    moda_clk_pad: in std_logic;
    moda_data: in std_logic;
    moda_out: out std_logic;
    modb_clk_pad: in std_logic;
    modb_data: in std_logic;
    modb_out: out std_logic;
    modc_clk_pad: in std_logic;
    modc_data: in std_logic;
    modc_out: out std_logic
) ;
end top;
architecture modular of top is
signal dll_clk_in : std_logic;
signal clk_top : std_logic;
signal dll_clk_out: std_logic;
signal a2top_obuft_i: std_logic;
signal a2c: std_logic;
signal a2b: std_logic;
signal b2top_obuft_t: std_logic;
signal b2c: std_logic;
signal b2a: std_logic;
signal c2and2: std_logic;
signal c2a: std_logic;
```

```
signal a_and_c: std_logic;
signal moda_clk: std_logic;
signal modb_clk: std_logic;
signal modc_clk: std_logic;
component IBUFG is port( I : in std_logic; O : out std_logic);
end component;
component CLKDLL is port (
  CLKIN : in std_logic;
  CLKFB : in std_logic;
  RST : in std_logic;
  CLK0 : out std_logic;
  CLK90 : out std_logic;
  CLK180 : out std_logic;
  CLK270 : out std_logic;
  CLKDV : out std_logic;
  CLK2X : out std_logic;
  LOCKED : out std_logic);
end component;
component BUFG
port (
  I : in std_logic;
  O : out std_logic);
end component;
component BUFGP
port (
  I : in std_logic;
  O : out std_logic);
end component;
-- Declare modules at top-level to get port directionality
component module_a is port( CLK_TOP: in std_logic;
  B2A_IN: in std_logic;
  TOP2A_IN: in std_logic;
  C2A_IN: in std_logic;
  MODA_DATA : in std_logic;
  MODA_CLK : in std_logic;
  A2B_OUT: out std_logic;
  A2TOP_OBUFT_I_OUT: out std_logic;
  A2c_out: out std_logic;
  MODA_OUT : out std_logic
);
end component;

component module_b is port( CLK_TOP: in std_logic;
  A2B_IN: in std_logic;
  TOP2B_IN: in std_logic;
  A_AND_C_IN: in std_logic;
  MODB_DATA: in std_logic;
  MODB_CLK: in std_logic;
  MODB_OUT : out std_logic;
  B2A_OUT: out std_logic;
  B2TOP_OBUFT_T_OUT: out std_logic;
  B2C_OUT: out std_logic);
end component;
component module_c is port( CLK_TOP: in std_logic;
  B2C_IN: in std_logic;
  TOP2A_C_IN: in std_logic;
  A2C_IN: in std_logic;
```

```

        MODC_DATA: in std_logic;
        MODC_CLK: in std_logic;
        MODC_OUT: out std_logic;
        C2A_OUT: out std_logic;
        C2TOP_OUT: out std_logic;
        C2AND2_OUT: out std_logic);
end component;
begin
ibuf_dll: IBUFG port map(I =>ipad_dll_clk_in,
    O => dll_clk_in);
dll_1: CLKDLL port map(CLKIN => dll_clk_in,
    CLKFB => clk_top,
    CLK0 => dll_clk_out,
    RST => dll_rst);
globalclk: BUFG port map(O => clk_top,
    I => dll_clk_out);
bufg_moda : BUFGP port map (O => moda_clk,
    I => moda_clk_pad);
bufg_modb : BUFGP port map (O => modb_clk,
    I => modb_clk_pad);
bufg_modc : BUFGP port map ( O => modc_clk,
    I => modc_clk_pad);

-- A simple piece of external logic at top level
a_and_c <= c2and2 and b2a;
-- Tri-state output
obuf_t_out <= a2top_obuft_i when b2top_obuft_t = '0' else 'Z';
instance_a: module_a port map (CLK_TOP =>clk_top,
    TOP2A_IN =>top2a_c,
    C2A_IN =>c2a,
    B2A_IN => b2a,
    MODA_DATA => moda_data,
    MODA_CLK => moda_clk,
    MODA_OUT => moda_out,
    A2B_OUT => a2b,
    A2TOP_OBUFT_I_OUT => a2top_obuft_i,
    A2C_OUT => a2c) ;
instance_b: module_b port map ( CLK_TOP => clk_top,
    TOP2B_IN => top2b,
    A2B_IN => a2b,
    A_AND_C_IN => a_and_c,
    MODB_DATA => modb_data,
    MODB_CLK => modb_clk,
    MODB_OUT => modb_out,
    B2TOP_OBUFT_T_OUT => b2top_obuft_t,
    B2C_OUT => b2c,
    B2A_OUT => b2a);
instance_c: module_c port map ( CLK_TOP => clk_top,
    TOP2A_C_IN => top2a_c,
    B2C_IN => b2c,
    A2C_IN => a2c,
    MODC_DATA => modc_data,
    MODC_CLK => modc_clk,
    MODC_OUT => modc_out,
    C2TOP_OUT => mod_c_out,
    C2AND2_OUT => c2and2,
    C2A_OUT => c2a);

```

```
end modular;
```

Verilog Example

```

module top (ipad_dll_clk_in, dll_rst, top2a_c, top2b, obuft_out,
mod_c_out, moda_data, moda_clk_pad, moda_out, modb_data,
modb_clk_pad, modb_out, modc_data, modc_clk_pad, modc_out) ;
input  ipad_dll_clk_in;
input  dll_rst;
input  top2a_c;
input  top2b;
output obuft_out;
output mod_c_out;
input  moda_data;
input  moda_clk_pad;
output moda_out;
input  modb_data;
input  modb_clk_pad;
output modb_out;
input  modc_data;
input  modc_clk_pad;
output modc_out;
//wire  ipad_dll_clk_out;
wire  clk_top;
wire  dll_clk_out;
wire  a2top_obuft_i;
wire  a2c;
wire  a2b;
wire  b2top_obuft_t;
wire  b2c;
wire  b2a;
wire  c2and2;
wire  c2a;
wire  a_and_c;
wire  moda_clk;
wire  modb_clk;
wire  modc_clk;
IBUFG ibuf_dll (.I(ipad_dll_clk_in),
.O(dll_clk_in));
CLKDLL dll_1 (.CLKIN(dll_clk_in),
.CLKFB(clk_top),
.CLK0(dll_clk_out),
.RST(dll_rst));
BUFG globalclk (.O(clk_top),
.I(dll_clk_out));
BUFGP bufg_moda (.O(moda_clk),
.I(moda_clk_pad));
BUFGP bufg_modb (.O(modb_clk),
.I(modb_clk_pad));
BUFGP bufg_modc (.O(modc_clk),
.I(modc_clk_pad));
// A simple piece of external logic at top level
assign a_and_c = c2and2 && b2a;
// Tri-state output
assign obuft_out = (!b2top_obuft_t) ? a2top_obuft_i : 1'bz;
module_a instance_a (.CLK_TOP(clk_top),
.B2A_IN(b2a),

```



```

        .TOP2A_IN(top2a_c),
        .C2A_IN(c2a),
        .MODA_DATA(moda_data),
        .MODA_CLK (moda_clk),
        .MODA_OUT (moda_out),
        .A2B_OUT(a2b),
        .A2TOP_OBUFT_I_OUT(a2top_obuft_i),
        .A2C_OUT(a2c)) ;
module_b instance_b ( .CLK_TOP(clk_top),
        .TOP2B_IN(top2b),
        .A2B_IN(a2b),
        .A_AND_C_IN(a_and_c),
        .MODB_DATA(modb_data),
        .MODB_CLK(modb_clk),
        .MODB_OUT(modb_out),
        .B2TOP_OBUFT_T_OUT(b2top_obuft_t),
        .B2C_OUT(b2c),
        .B2A_OUT(b2a));
module_c instance_c ( .CLK_TOP(clk_top),
        .TOP2A_C_IN(top2a_c),
        .B2C_IN(b2c),
        .A2C_IN(a2c),
        .MODC_DATA(modc_data),
        .MODC_CLK(modc_clk),
        .MODC_OUT(modc_out),
        .C2TOP_OUT(mod_c_out),
        .C2AND2_OUT(c2and2),
        .C2A_OUT(c2a));
endmodule
// Declare modules at top-level to get port directionality
module module_a ( CLK_TOP, B2A_IN, TOP2A_IN, C2A_IN, MODA_DATA,
MODA_CLK, MODA_OUT, A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT) ;
input CLK_TOP ;
input B2A_IN ;
input TOP2A_IN ;
input C2A_IN ;
input MODA_DATA;
input MODA_CLK;
output MODA_OUT;
output A2B_OUT ;
output A2TOP_OBUFT_I_OUT ;
output A2C_OUT ;
endmodule

module module_b ( CLK_TOP, A2B_IN, TOP2B_IN, A_AND_C_IN, MODB_DATA,
MODB_CLK, MODB_OUT, B2A_OUT, B2TOP_OBUFT_T_OUT, B2C_OUT) ;
input CLK_TOP ;
input A2B_IN ;
input TOP2B_IN ;
input A_AND_C_IN ;
input MODB_DATA;
input MODB_CLK;
output MODB_OUT;
output B2A_OUT ;
output B2TOP_OBUFT_T_OUT ;
output B2C_OUT ;
endmodule

```

```

module module_c ( CLK_TOP, B2C_IN, TOP2A_C_IN, A2C_IN, MODC_DATA,
MODC_CLK, MODC_OUT, C2A_OUT, C2TOP_OUT, C2AND2_OUT) ;
input CLK_TOP ;
input B2C_IN ;
input TOP2A_C_IN ;
input A2C_IN ;
input MODC_DATA;
input MODC_CLK;
output MODC_OUT;
output C2A_OUT ;
output C2TOP_OUT ;
output C2AND2_OUT ;
endmodule

```

Appendix B – HDL Code Examples for Inserting I/Os in a Module

VHDL Example

```

library IEEE;
use IEEE.std_logic_1164.all;
entity module_a is port ( CLK_TOP : in std_logic;
    B2A_IN: in std_logic;
    TOP2A_IN: in std_logic;
    C2A_IN: in std_logic;
    MODA_DATA : in std_logic;
    MODA_CLK : in std_logic;
    MODA_OUT : out std_logic;
    A2B_OUT: out std_logic;
    A2TOP_OBUFT_I_OUT: out std_logic;
    A2C_OUT: out std_logic) ;
end module_a;
architecture modular of module_a is
-- add your signal declarations here
signal Q0_OUT, Q1_OUT, Q2_OUT, Q3_OUT : std_logic;
signal AND4_OUT: std_logic ;
signal OR4_OUT : std_logic;
begin
AND4_OUT <= Q0_OUT and Q1_OUT and Q2_OUT and Q3_OUT ;
OR4_OUT <= Q0_OUT or Q1_OUT or Q2_OUT or Q3_OUT ;
TOP_CLK: process(CLK_TOP)
begin
if (CLK_TOP'event and CLK_TOP = '1') then
    Q0_OUT <= MODA_DATA ;
    Q2_OUT <= TOP2A_IN ;
    MODA_OUT <= OR4_OUT ;
    A2B_OUT <= AND4_OUT ;
end if;
end process TOP_CLK;
CLK_MODA: process(MODA_CLK)
begin
if (MODA_CLK'event and MODA_CLK = '1') then
    Q1_OUT <= B2A_IN ;
    Q3_OUT <= C2A_IN ;
    A2TOP_OBUFT_I_OUT <= AND4_OUT ;
    A2C_OUT <= OR4_OUT ;
end if;

```

```
end process CLK_MODA;
end modular;
```

Verilog Example

```
module module_a ( CLK_TOP, B2A_IN, TOP2A_IN, C2A_IN, MODA_DATA,
MODA_CLK, MODA_OUT, A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT);
input CLK_TOP ;
input B2A_IN ;
input TOP2A_IN ;
input C2A_IN ;
input MODA_DATA, MODA_CLK;
output MODA_OUT;
output A2B_OUT ;
output A2TOP_OBUFT_I_OUT ;
output A2C_OUT ;
// add your declarations here
reg Q0_OUT, Q1_OUT, Q2_OUT, Q3_OUT ;
reg A2B_OUT, A2TOP_OBUFT_I_OUT, A2C_OUT ;
reg MODA_OUT;
wire AND4_OUT ;
wire OR4_OUT ;
// add your code here
assign AND4_OUT = Q0_OUT && Q1_OUT && Q2_OUT && Q3_OUT ;
assign OR4_OUT = Q0_OUT || Q1_OUT || Q2_OUT || Q3_OUT ;
always @ (posedge CLK_TOP)
begin : TOP_CLK
    Q0_OUT <= MODA_DATA ;
    Q2_OUT <= TOP2A_IN ;

    MODA_OUT <= OR4_OUT ;
    A2B_OUT <= AND4_OUT ;
end
always @ (posedge MODA_CLK)
begin : CLK_MODA
    Q1_OUT <= B2A_IN ;
    Q3_OUT <= C2A_IN ;

    A2TOP_OBUFT_I_OUT <= AND4_OUT ;
    A2C_OUT <= OR4_OUT ;

end
endmodule
```

In the above example, the module has two external inputs (IPAD_MODA_CLK and IPAD_MODA_DATA), and one external output (OPAD_MODA_OUT). For these external I/Os, IBUF, OBUF, and BUFGP are instantiated.

The lower-level port declaration is different from top-level declaration of module_a. Specifically, lower-level module_a has three additional ports. With Modular Design, ngdbuild will ignore this port mismatch and use module_a.edf to describe module_a. These I/Os will be present in the design and available for simulation.

Appendix C – Vendor-Specific Notes

This section provides vendor-specific information on the following subjects:

- Creating a Separate Netlist for Each Module
- Disabling I/O Insertion
- Instantiating Primitives

Creating a Netlist for Each Module

In Synplify or FPGA Express/FPGA Compiler II version 3.3.1 or earlier, each design project creates one netlist. A project is required for the top level and for each lower-level module (four projects are created in the previous examples: for top, module_a, module_b, and module_c.). The top-level project will be synthesized with I/O insertion and the lower levels will be synthesized without I/O insertion.

FPGA Express/FPGA Compiler II (version 3.4 or later) includes a new Incremental Synthesis feature. This feature allows each design module to be synthesized individually within a project. Exporting the design produces a separate EDIF for each module, tagged with a Block Root designation. This attribute is set under the Modules tab within the FPGA Express/FPGA Compiler II Constraints Editor, as shown in the **Figure 3**.

	Name	Hierarchy	Primitives	Dont Touch	Block Partition	Operator Sharing	Optimize for	Effort	Duplicate Register Merge
1	<default>	Preserve	Preserve			On	Speed	High	Enable
2	top			False	Block Root				
3	IBUFG - ibuf_dll								
4	CLKDLL - dll_1	Preserve							
5	BUFG - globalclk								
6	BUFGP - bufg_modc								
7	module_a - instance_a				Block Root				
8	BUFGP - bufg_modb								
9	BUFGP - bufg_moda								
10	module_c - instance_c				Block Root				
11	module_b - instance_b				Block Root				

Figure 3: FPGA Constraint Editor, "How to Assign Block Root"

In LeonardoSpectrum, multiple netlists from a single project can be created from the GUI as well as from the script. The following script describes the method for a VHDL design:

```

set part v50ecs144
load_library xcve
read ./top.vhd
optimize -target xcve -hier preserve
present_design .work.top.modular
auto_write -format edf top.edf
read ./module_a.vhd
read ./module_b.vhd
read ./module_c.vhd
optimize -target xcve -hier preserve
present_design .work.module_a.modular
auto_write -format edf module_a.edf
present_design .work.module_b.modular
auto_write -format edf module_b.edf
present_design .work.module_c.modular
auto_write -format edf module_c.edf

```

The following script describes the method for a Verilog design:

```

set part v50ecs144
load_library xcve
read ./module_a.v

```

```
read ./module_b.v
read ./module_c.v
read ./top.v
optimize -target xcve -hier preserve
present_design .work.module_a.INTERFACE
auto_write -format edf module_a.edf
present_design .work.module_b.INTERFACE
auto_write -format edf module_b.edf
present_design .work.module_c.INTERFACE
auto_write -format edf module_c.edf
NOOPT .work.module_a.INTERFACE
NOOPT .work.module_b.INTERFACE
NOOPT .work.module_c.INTERFACE
present_design .work.top.INTERFACE
auto_write -format edf top.edf
```

Disabling I/O Insertion

In Synplify, click the Change button next to Target, or select **Target->Set Device Options**. Then, check the Disable I/O insertion box. (Figure 4)

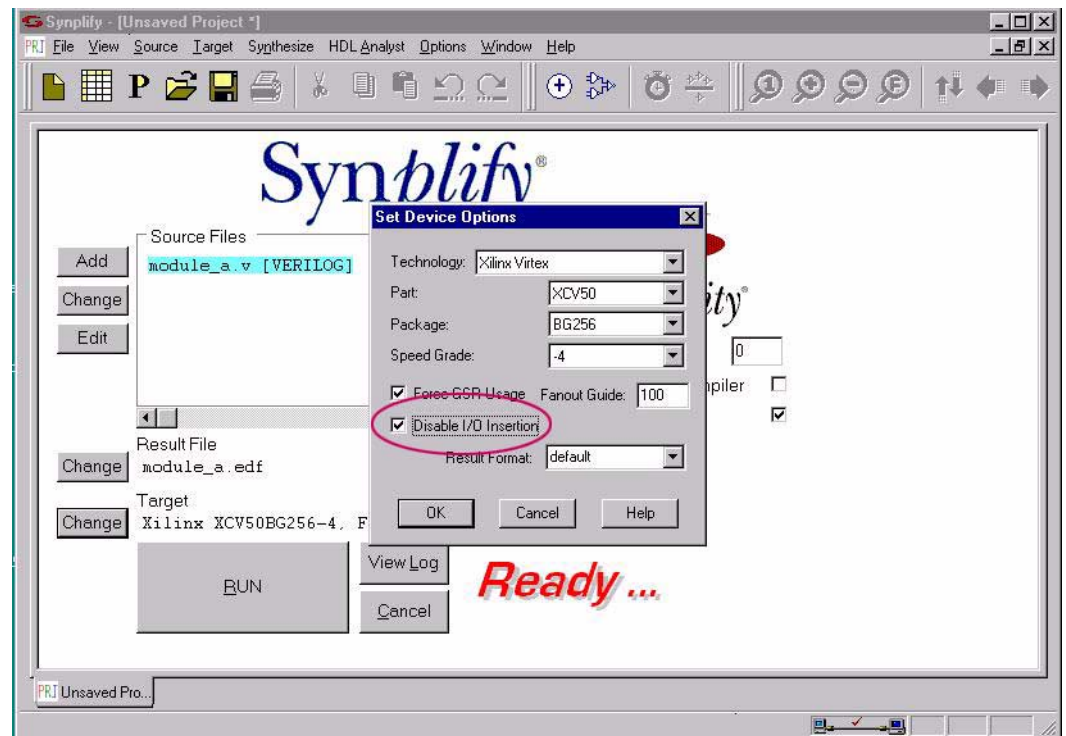


Figure 4: Disabling I/O Insertion in Synplify

In FPGA Express, click the Create Implementation icon after Update. Check “Do not insert I/O pads.” (Figure 5).

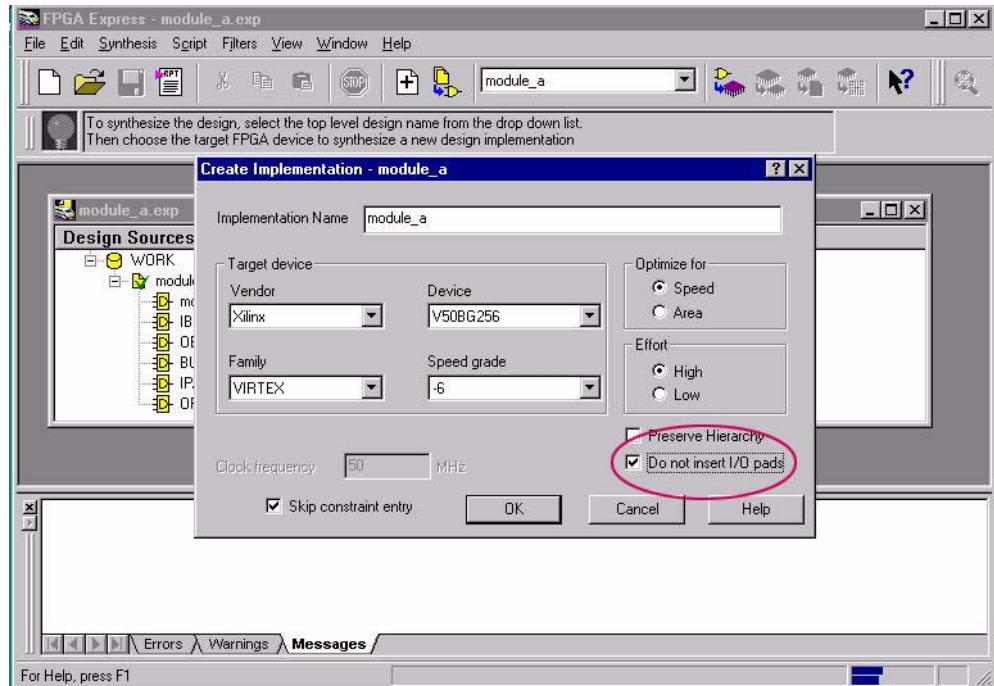


Figure 5: Disabling I/O Insertion in FPGA Express

In LeonardoSpectrum (Figure 6),

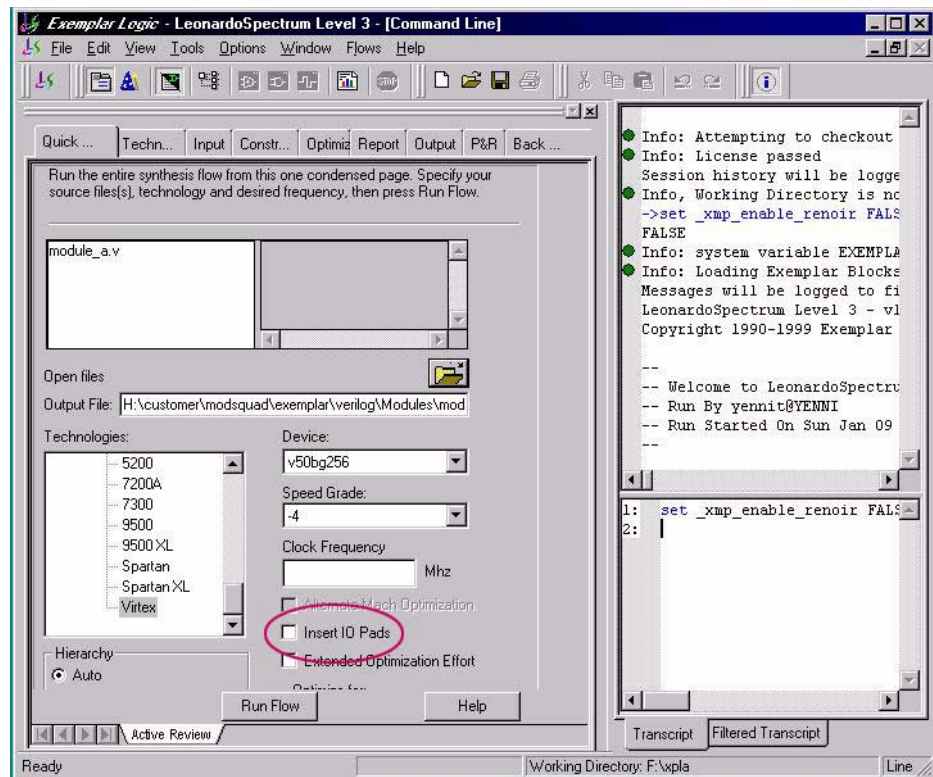


Figure 6: Disabling I/O Insertion in LeonardoSpectrum

Instantiating Primitives

Synthesis tools support different methods to instantiate primitive components. Most synthesis tools recognize primitive components when the correct library or device is targeted.

- In FPGA Express, all instantiated components must be declared in VHDL. Module declarations are not necessary in Verilog. Also, if an IBUFG is instantiated in the top-level Verilog code, FPGA express inserts IBUF before IBUFG, causing an ngdbuild error. A workaround is to instantiate the IPAD, omitting the port declaration. This is not an issue in a VHDL design.
- LeonardoSpectrum instantiates components in a manner similar to FPGA Express—that is, all VHDL components must be declared. However, Verilog module declaration is not necessary.
- Synplify provides Virtex primitives in “library virtex” (VHDL) and “virtex.v” (Verilog). These are available in \$SYNPLICITY/lib/xilinx. Primitives may be called and port mapped without component or module declarations.

Appendix D – Tool Limitations in the First Modular Design Release

The Constraints Editor identifies clock signals by recognizing clock pin loads. Consequently, the Constraints Editor will not recognize the clock signal during the initial budgeting phase unless a clock pin load exists at the top level of the design.

An OFFSET constraint on a module port must be relative to the actual clock pad net (most likely **["located" "specified"?]** in the top-level design), not to the module clock port

Par guide information in the report file refers to failures to guide pseudo logic.

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
04/14/00	1.0	Initial Xilinx release.
06/15/00	1.1	Added note to have all 3-state drivers within a lower level module declared as "inout". Added cmd line example to show copying of top-level UCF to module directories. Fixed NGDBUILD command line for module impl. Phase to include -uc switch. Added tcl script for exemplar verilog design
04/21/01	1.2	Moved Synthesis examples into an Appendix A, B and C. Renamed Appendix on known limitations to be Appendix D. Added description of floorplanning with auto port placement during Initial Budgeting. Removed notes about limitations that have been fixed