XAPP404 (v1.1) June 19, 2000

Xilinx Alliance 3.1i Modular Design

Summary

This application note addresses the Modular Design feature of Xilinx Alliance Series 3.1i. Modular Design allows designers to partition large, complex FPGA projects into several independent pieces which are more easily managed and updated through the product's lifetime. Design entry, design implementation, and simulation flows for such projects are discussed in detail, with specific instructions and screen graphics provided for using FPGA Express, LeonardoSpectrum, and Synplify software. Illustrative VHDL and Verilog code segments are provided.

Introduction

With the availability of large Virtex™ devices, designers often need to partition a single large design into several smaller designs. This partitioning may be done for a number of reasons that include:

- **Complexity Management:** Large designs are generally difficult to manage if kept as a single, monolithic entity. By dividing a design into smaller pieces, each piece can be separately understood and implemented.
- **Team Design:** A team of designers might be working together on a single design and want to be able to work independently of one another as much as possible. By dividing a design into smaller pieces, each piece can be independently worked on in parallel.
- **Engineering Change Order (ECO):** Changes made to one portion of the design should not force unnecessary modification of unrelated parts of the design. By dividing a design into pieces, the relatively static pieces can be implemented once and not be subject to timing or layout changes as the more dynamic pieces of the design are changed.

As with any system that is partitioned, there are certain costs that will be incurred in working with this methodology. Dividing a large design up into partitions is going to incur greater costs in the following areas:

- **Planning:** A greater amount of overhead for planning to make sure the partitioning is done correctly and accurately.
- **Communication:** A greater amount of overhead for communicating information between different partitions.

To ease this task, Xilinx has provided a new feature in Alliance™ Series 3.1i called Modular Design. This Application Note will further describe the design entry, design implementation and simulation flows that should be used when working within a Modular Design context. Appendix A lists some of the known limitations of the initial release of Modular Design. These limitations are planned to be addressed in a future software offering.

Recognizing that the Modular Design flow will require learning a significantly new, and more complex design flow, it is imperative that the designer first properly evaluate whether or not this new flow is appropriate for his design.

Modular Design is most appropriate if the following applies:

- The design is large and can be "logically" divided into mostly self-contained pieces. Further discussion of what to look for in a reasonable division is presented below.
- The design will be created and implemented across a team of more than one person.

This increased investment in planning and communication is highly typical of most large digital design projects. Customers accustomed to large ASIC designs should be familiar with the principles involved.

With this initial release, it is also very important to know when *not* to use Modular Design. Modular Design will not be useful for the implementing the following types of designs:

- A large ASIC design has not been designed from the beginning with Modular Design concepts in mind. Without an effective functional and physical partitioning of the design into proper modules, little benefit will likely be gained, and development time can be wasted.
- The design is heavily interconnected in order to squeeze every last MHz of speed out of the design. The overhead incurred from using Modular Design flow may preclude obtaining the fastest clock speeds for such designs.

The most important thing that can be done to maximize the benefits of Modular Design is to properly structure and partition the design at the HDL source level. Since most designs initially targeted to this flow may not have been written with this exact flow in mind, it is very important to analyze the source and assure a reasonable confidence that it is suitable for Modular Design.

The question that is most apparent at this time is: What exactly is a module? More importantly, what is a properly designed module? There is not a right and wrong way to partition a module, but certain guidelines can be used as a starting point to help in the creation of modules. Over time, each designer will develop their own set of rules and guidelines to help them design with modules. Roughly speaking, a "good" module should include these attributes:

- **Ports are well defined.** A port is any 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.
- **Modules use a minimal number of ports.** A module with a minimal number of ports is more self-contained than one with a large number. The ability of a module to be self-contained will generally allow that module to be more optimally implemented due to its minimal dependence on the function of other modules.
- **Modules do not contain too much global logic.** Global logic is defined as logic that is not evenly distributed on a target chip. Examples of this are I/O pins leading onto or off of the chip, DLLs, or other global clock modification resources.
- **Outputs of modules are registered.** This will lead to the most efficient logic packing during the mapping phase of implementation.
- **Modules are not defined such that they are dependent upon specific on-chip locations**, for example by requiring a certain set of BRAMs.
- **The number of modules is reasonably small.** A good rule of thumb is to envision the number of engineers it would take to efficiently implement the whole design. The number of modules should be similar to this estimate.

Modular Design places an exacting premium on the generally accepted principles of sound HDL design for synthesis. Designs that deviate from sound coding practice will be particularly ill suited for the Modular design flow.

Design Entry / Synthesis Flow

This section will discuss how to bring a modular design through FPGA Express/FPGA Compiler II, LeonardoSpectrum, or Synplify.

NOTE: For all issues addressed within this document, there is no difference between FPGA Express and FPGA Compiler II.

Creating the Top-Level Netlist

A top-level design structure must be constructed and agreed upon by the designers before beginning modular design. The top-level design should include all modules/partitions of the design, as well as global logic such as clock resources or logic that connects I/O ports to

modules and connects between modules. The modules are instantiated and synthesized separately as described in the following example:

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
obuft_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
```

The top-level code above instantiated three modules: module_a, module_b, and module_c, which are named instance_a, instance_b, and instance_c respectively. There are also nets and logic that connect the ports to the modules as well as between the modules.

Guidelines for top-level code:

- **All lower-level modules must be declared** to define port directionality and bus width.

  VHDL synthesis requires component declarations of all instantiated components in the HDL code. The component can be declared in the code or in a library package included in the HDL code. Synthesis tools will produce errors on undeclared components.

  Verilog synthesis requires declarations for user modules only, but not library primitives as illustrated in the example above. If the user modules are defined and described within the same project, module declarations are not necessary. This is the case if the synthesis tools can produce multiple EDIF netlists from a single project. If the user modules are described in a separate project, or if CORE Generator modules are used, then module declaration is a must.

- **All lower-level modules must be synthesized as black boxes.** To accomplish this synthesis, tools may require a directive. If the design is synthesized such that the lower-level modules are not black boxes, the result is a flat non-modular design which will error out in ngdbuild during 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 only two levels of hierarchy** (top level and modules instantiated within top level).
- Currently, there is a limitation that **the code cannot make multiple instantiations of the same module.** Each module instantiation must correspond directly with a single module definition, even if modules have exactly the same port definition and function.

  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 (...)
```

Though this is technically correct VHDL and Verilog coding, the current modular design flow has a limitation that does not accept this coding style. A separate synthesis is not necessary to produce EDIF netlist for module_b; module_a.edf may be copied to module_b.edf. However, all the implementation steps should be run separately for module_a and module_b.

- **I/O registers must be inferred in top-level code.** To meet timing, it may be necessary to move registers out of modules to the top level in order to infer output register. Otherwise, it is recommended to register outputs of a module.
- **3-state buffers driving the same net/bus should all be inferred on one level**, preferably at the top-level, with local control logic. Isolating this will ease the mutually exclusive requirement for the control.
- **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.
- **Meaningful signal names must be used to connect to ports of a module or between modules on top-level code.** Using the same name for the signal and the ports which are connected with that signal is recommended. These top-level signal names will be used on the back-annotated simulation netlist. See the **Simulation Flow** section for more information.

Creating Lower-Level Modules

Xilinx Alliance 3.1i Modular Design requires each lower-level module to be represented by its own EDIF netlist. The following guidelines describe how to properly create a lower-level EDIF netlist:

1. Most synthesis tools will only generate one EDIF netlist for each project. For this reason, 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. In LeonardoSpectrum, it is possible to create multiple EDIF netlists from one project. This will be described in **Vendor-Specific Notes**. The EDIF netlists of lower-level modules must be identical to the module names (i.e., module_a.edf, module_b.edf, and module_c.edf). Do not use the instantiation name (instance_a, instance_b, and instance_c), for ngdbuild will not be able to match the names specified in top-level EDIF.
2. Modules must be synthesized without I/O insertion. This option is available in most synthesis tools as described in **Vendor-Specific Notes**.
3. Although declaring all external I/Os in the top level is recommended, it is possible to include external I/Os in a module without modifying the top-level code. This capability allows a module designer to add temporary external I/Os in the module and simulate the module with them. This can be done by explicitly instantiating IBUF/IBUFG/BUFGP as well as OBUF connections. The following example shows a module with external I/O instantiations.

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.

Note that the lower-level port declaration is different from the top-level declaration of module_a (i.e., the 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.

Vendor-Specific Notes

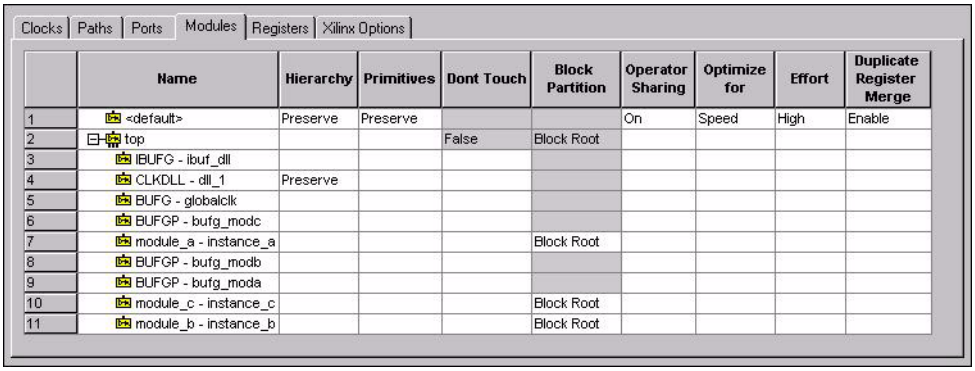
This section will discuss vendor specific information on the following subjects:

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

Creating a Separate Netlist for Each Module

In Synplify or FPGA Express / FPGA Compiler II version 3.3.1 or earlier, each design project will create one netlist. A project is required for the top level and for each lower-level module. For example, four projects are created in the example above: 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.

In FPGA Express / FPGA Compiler II version 3.4 or later, a new feature called *Incremental Synthesis* has been added. This feature allows each module of a design to be synthesized individually within a project. Exporting the design will produce a separate EDIF for every 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 Figure 1.



	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 1: **FPGA Constraint Editor, "How to Assign Block Root"**

In LeonardoSpectrum, it is possible to create multiple netlists from a single project from the GUI interface as well as from the script. The following script describes the method for 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 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 box next to "Disable I/O Insertion" (Figure 2).

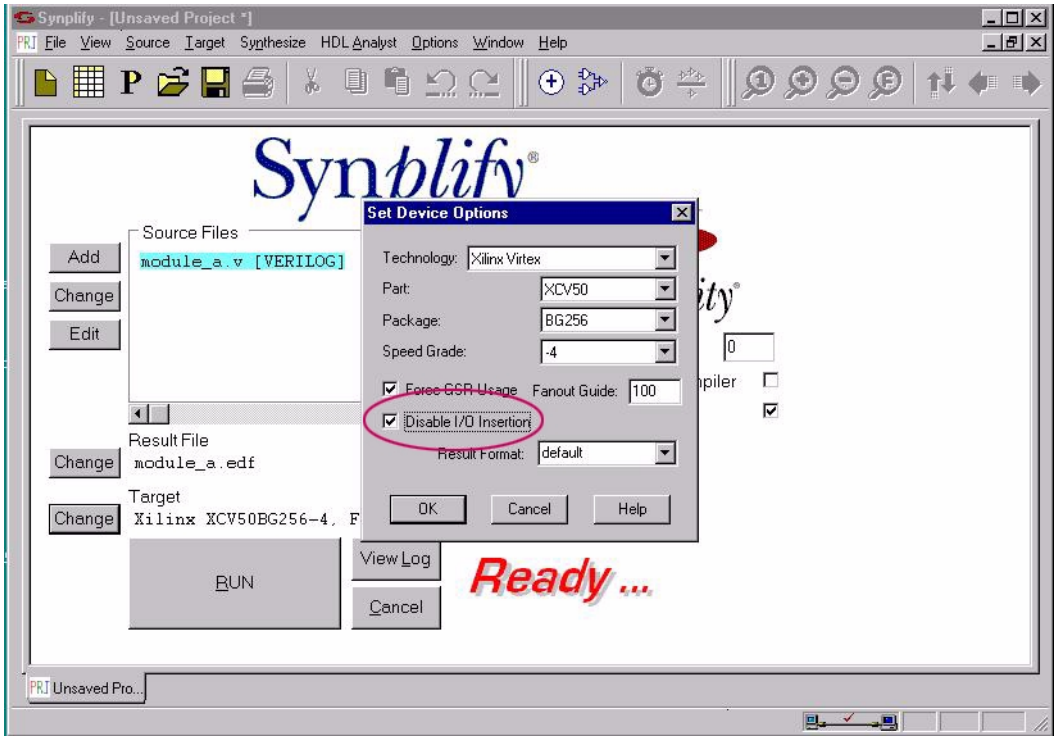


Figure 2: **Disabling I/O Insertion in Synplify**

In FPGA Express, click the "Create Implementation" icon after "Update". Check the "Do not insert I/O pads" checkbox (Figure 3).

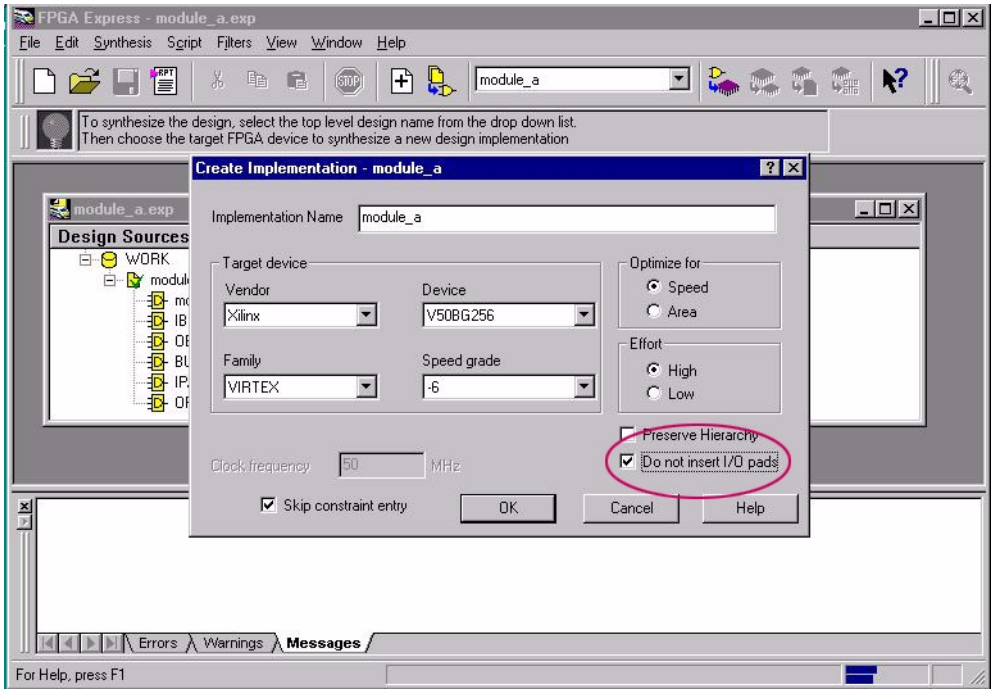


Figure 3: **Disabling I/O Insertion in FPGA Express**

In LeonardoSpectrum (Figure 4):

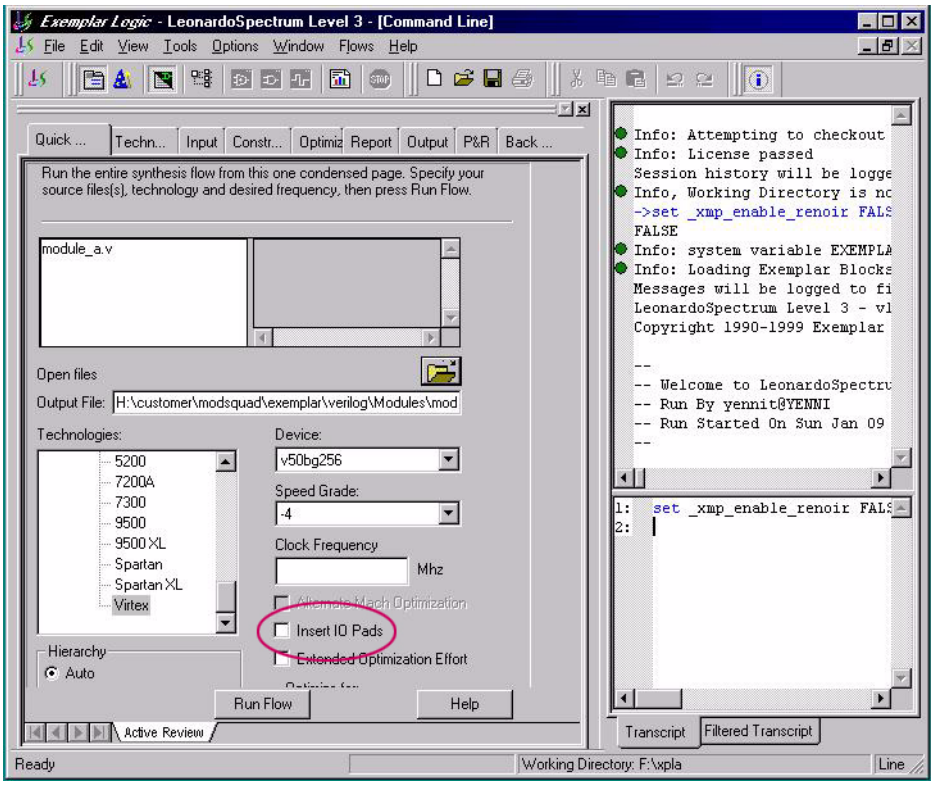


Figure 4: **Disabling I/O Insertion in LeonardoSpectrum**

Instantiating Primitives

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

- In FPGA Express, all instantiated components need to be declared in VHDL. Module declarations are not necessary in Verilog. Also, if an IBUFG is instantiated in top-level verilog code, FPGA Express will insert IBUF in front of IBUFG, causing an ngdbuild error. A workaround for this is to instantiate the IPAD and omit the port declaration. This is not an issue in VHDL design.
- LeonardoSpectrum behaves similarly to FPGA Express when instantiating components in that all VHDL components must be declared. In Verilog modules, declaration is not necessary.
- Synplify provides Virtex primitives in "library virtex" (VHDL) and "virtex.v" (Verilog). These are available under $SYNPLICITY/lib/xilinx. Primitives may be called and port-mapped without component/module declarations.

Constraining at the Module Level

Global and top-level specific constraints can be entered via synthesis tools when synthesizing the top level. However, most module-specific constraints will need to be either manually entered or created using Floorplanner through a UCF file. Among them are constraints with hierarchical reference and area constraints; this is discussed in more detail in the **Implementation Flow** section of this application note.

Some synthesis tools write out default constraints in an NCF file for each synthesis project. To avoid conflict between NCF files, disable the NCF writing option when synthesizing lower-level modules.

Implementation Flow

The Modular Design flow described below is currently implemented as command line options to batch tools only in this first release. There is no Design Manager/Flow Engine support at this time. The scripting tool Xflow can be used to run the different phases of Modular Design as described below. Sample Xflow scripts have been included below that can be modified to work for a specific design. It must be remembered in the following descriptions that Modular Design is addressing issues that only arise when the need to partition a design arises. Many of the phases or steps within a phase may seem overly complex or redundant when viewed from the viewpoint of a single designer implementing a design as they do today. On the other hand when these steps are viewed from the point of multiple designers, divide and conquer strategy or ECO they seem less complex since they would be done by different people or at different times.

There are three main phases or operations in the Modular Design flow. Sets of new command line switches have been added to the tool ngdbuild to indicate the current phase of Modular Design. The "-modular" switch and its argument indicate these phases. These phases are:

1. **Initial Budgeting**

   This is where a design that was previously partitioned by the HDL tools is first brought into the Xilinx FPGA tools. At this time resources are allocated, and modules are sized, positioned, and connected to one another.

2. **Active Module Implementation**

   This is where each of the previously defined modules are independently implemented using the information generated from Initial Budgeting. In this phase the different module sizes and locations as previously described are used as requirements for module implementation. The full suite of FPGA design and implementation tools is available for each module.

3. **Final Assembly**

   This is where the final design is assembled from the information created in the previous two phases. The information generated from each Active Module Implementation phase is used to "guide" the implementation of the associated logic in the final design. The advantage of this process is that it is fast, and allows the tools to treat the whole design as one entity for things like constraint resolution or buffer alignment.

Each of these phases is described in detail below.

Initial Budgeting Phase

The goal of the Initial Budgeting phase is threefold:

- To position any global logic in the top-level design that is outside of any particular module.
- To size and position each module previously defined in the HDL netlists on the target implementation chip.
- To position the input/output ports (nets that flow into or out of a module) of each module so that later implementations of these modules will correctly line up with one another.

The team leader or person responsible for the final design typically performs this phase of the flow. The importance of this phase cannot be overemphasized. Once this phase has been completed and module implementation begun, changes to the information provided by this phase will be difficult and time consuming. At this time, none of the logic within any given module is required, but a clear definition of the size and port connectivity is required. Once again, changes to this information will be costly to make at a later time. This phase requires the selection of a specific part (device, package and speed) that will be used later. Probably the most important thing to get correct at this time is the definition of what should be in each module. This information probably comes from the HDL tools, and changes to this will be difficult once the process has started. Time spent on up-front design at this phase will greatly benefit the rest of the implementation process.

It is suggested, though not required, that all "global logic and nets" be placed into the top-level design. This would include all I/Os, clock nets, DLLs, RAMs, and other resources that are not evenly distributed across the target chip. All resources that are part of the top-level design can be placed during this phase, then used in subsequent Active Module Implementation phases to provide those modules with a maximum amount of context. The main steps of this phase are creating an NGD file of the top-level design without any module implementation information, adding timing constraints, and using Floorplanner to position global logic, size/position each module on the target chip, and position the module ports for correct alignment.

The work for this phase should be performed in a directory that can eventually be accessed by each of the module implementers. Each of these steps is described below. This discussion assumes that the netlist for the top-level design is contained in the file "top.edf" after synthesis:

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

   This step is the first time a new command line switch is used to indicate to the tools that a Modular Design flow is being used. The "initial" argument is given to ngdbuild to indicate that we are in the Initial Budgeting phase of modular design. At this time the netlists describing each module should be unavailable to ngdbuild so that they will be treated as black boxes for future Active Module Implementation. The command used here is:

   ngdbuild -modular initial top.edf

   The output file of this step (top.ngo) will be directly used in the remaining steps of this phase, as well as in all Active Module Implementation phases.

2. **Add timing constraints.**

   It is at this point that any global clocks should be constrained with a period constraint. With respect to initial timing constraints, the user can place PERIOD and I/O timing constraints for all of the clocks and I/Os on the top level.

After completion of adding timing constraints ngdbuild will need to be rerun as above to put these constraints into the resultant top.ngd file. The command used here is:

constraints_editor top.ngd

These timing constraints will be saved in the file top.ucf , which will be used during all Active Module Implementation phases below.

> **NOTE:** If there are no clock loads at the top level of the design, the clock net will not appear in the Constraints Editor, and some other method must be used for defining PERIOD and I/O timing constraints, such as using NCF constraints from the front-end tool. For this reason, it may be worthwhile to add a dummy register at the top level, driven by the clock net. This will provide access to the clock net in the Constraints Editor. If the register has no load, it will be removed during the mapping phase.

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

   It is recommended that all logic present at the top level of the design be constrained to fixed locations in the top-level floorplan.

   As described in the Design Entry/Synthesis portion of this document, it is recommended that the HDL be written so that all 3-state drivers reside at the top level of the design. It is mandatory to ensure that all BUFT symbols driving the same long-line net be manually placed so that they are in the same row.

   The area-assignment step is used to size and position the target area for each module implementation on the target chip. It is also used to position the ports of each module so that they can be later implemented to have correct alignment with one another. Any net that connects into or out of a module is referred to as a *port net*. Each module/port pair will be indicated by a piece of logic "created" in the tools that can be positioned to "pull" the logic within a module to the desired locations. This "created logic" is referred to as *pseudo-logic* and will not be part of the final design, but is only used during module implementation. At this time there is no accurate way to tell how much logic is used by each module, so a guess must be made as to the size to be allocated for each module.

   > **NOTE:** The Floorplanner currently gives no indication of the required size of a module, so this information must be inferred from other sources.

   Each module should be positioned on the chip with the Floorplan → Assign Area Constraint menu item. Next, each port of each module should be placed so that correct alignment of buses can occur. A given net that connects module A and module B will have two separate ports that need to be placed to constrain it. First, 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. These ports, referred to as *pseudo-logic*, are not really part of the module and are used only to force alignment of the logic connected to them inside the module. They will be removed during the Final Assembly phase and will not affect the final design. For example, 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 acceptable to put the ports associated with Module A into the area defined for Module B rather than leave blank rows and columns between modules.

   The commands used to perform these tasks 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

Open "Module hierarchy"

Place each port (pseudo*) component

File->Write Constraints: top.ucf

Exit

These placement constraints will be saved into the file top.ucf. This file will be copied into each Active Module Implementation directory for use and further modifications for module specific constraints during the next phase.

In summary, these commands are used during Initial Budgeting phase of Modular Design:

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

At this point, all global logic has been placed, each module has been sized and positioned, and the ports for each module have been positioned. Extra time should be now be spent to make sure that as much as possible everything has been positioned correctly and that the modules have been correctly defined. Future changes, though possible, will be time consuming to make.

Active Module Phase

The goal of the Active Module implementation phase is to fully implement each module defined for this design using the sizes and locations defined in the Initial Budgeting phase (Figure 5). The Active Module phase is repeated for each module and should be performed in parallel in separate and distinct directories. The information generated by this phase will be a set of files that will be used as guide information during the Final Assembly phase. These files will not be visible to the team leader working on the top-level design until the module implementer explicitly publishes them. Typically, the module implementers will be distinct from the team leader who performed the initial budgeting, but this is not required. Once this phase has been completed for each module, the team leader can start the Final Assembly phase.

As each module is implemented during this phase, it will use the ngo and ucf files from the Initial Budgeting phase. Each module will be implemented within the context of the top-level design as if it were the only module that exists. The top-level design acts as a sort of testbench to provide added context during module implementation. It is important to understand this concept so that the names of input and output files can be understood. As an example, when a module is implemented, the output file will be named after the top-level design (top.ncd) rather than after the module itself. The module implementation information will be contained in this file along with the top-level design implementation information. During this phase the ngo file from the top-level design will be used in a read-only mode while the ucf file from top-level design will be used in a read/write mode. To simplify this, logic below the top.ngo file will be referenced directly with the top.ucf file, which will be copied into a module implementation directory to allow for modifications that are not needed by the top-level design.

The full suite of Xilinx implementation tools are available for individual module implementation, including the ability to simulate the module "independent" of the top-level design. This implies that designers are free to use any of the map and par command line switches and Floorplanner to achieve the desired implementation. The fpga_editor can also be used, but care must be taken not to violate any of the area constraints or placement information previously generated. The simulation capabilities can be used to verify that each module meets its desired specifications. The mapper and par have been modified to recognize the Active Module Implementation phase and to keep all logic and routing within the defined module boundaries. No logic optimization will occur across the module boundary from the top-level design to guarantee that this implementation will be compatible with other module implementations. The main steps of this phase are to create an NGD file for this module, add timing constraints specific to this module if needed, map the logic of the module, place and route the module,

floorplan placement information if needed, simulate the module if desired, and finally publish the module implementation information back to the team leader.

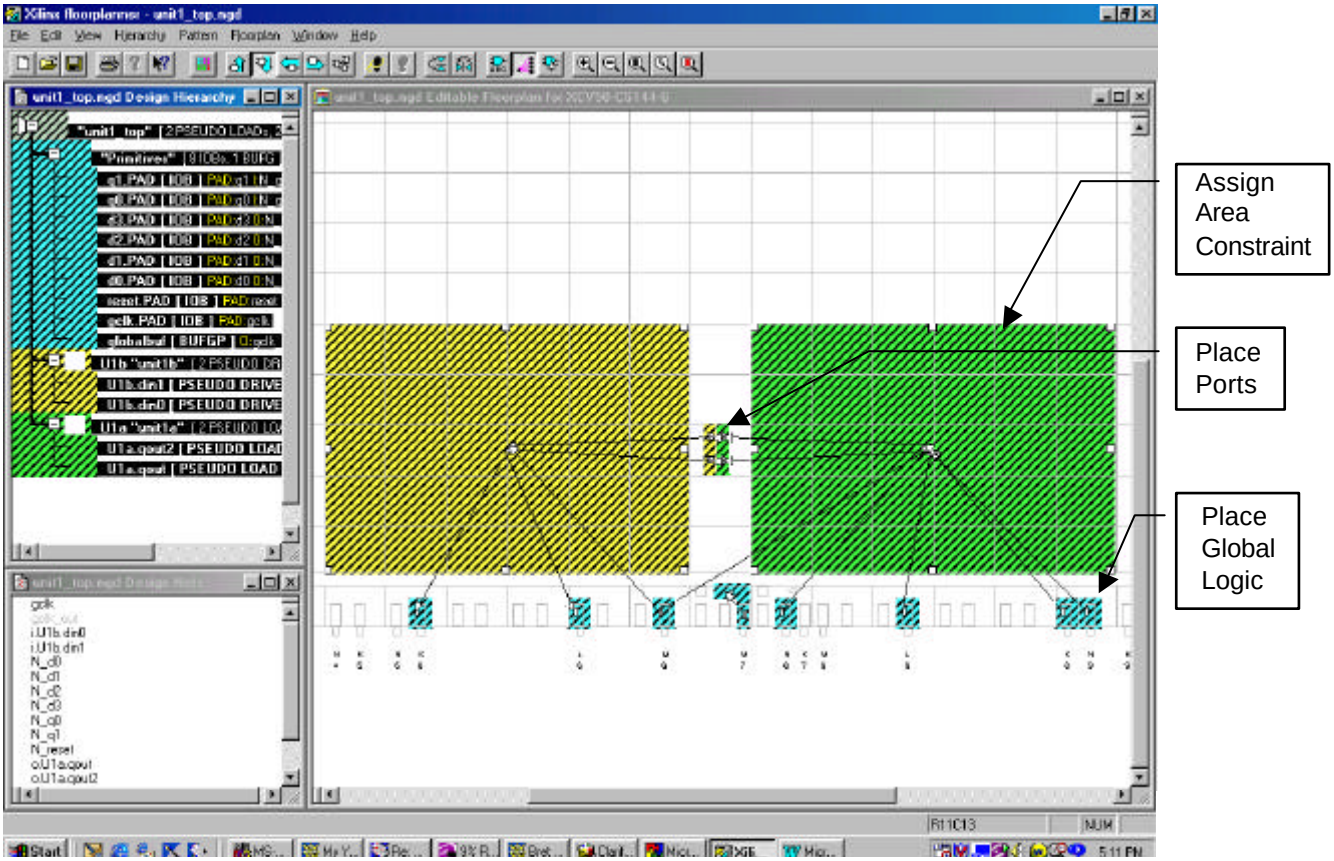


Figure 5: **Example of Floorplanner Usage During Initial Budget Mode**

The work for this phase for each module can be performed in any directory desired by the module implementers. It will only be accessible to the team leader for the Final Assembly when it has been explicitly published to a central directory. Each of these steps is described below; the discussion assumes that the netlist for the module is contained in the file "moda.edf" after synthesis, and that the information for the top-level design called top.* is contained in the directory "top_directory".

1. **Create an NGD file for this module.**

   As stated above, the ngd file for this module is actually an ngd file for the top-level design and this module. This file will be named after the top-level design as previously defined. The constraint information for the top-level design will be used during this phase, but will also possibly be appended with module-specific constraints so a copy of the top-level ucf file will be used.

   Create an empty directory and copy the top.ucf file into the directory. Now copy or create the EDIF file moda.edf that describes the logic of this module. Then run ngdbuild, giving the "module" argument to the -modular flag to indicate that we are in the Active Module Implementation phase of modular design. The name of the current module should also be given as an argument to the -active switch for future tools to use. The command used here (once moda.edf has been created and top.ucf has been copied here) is:

   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 of this step will again be top.ngd, but it will contain the implementation logic for the top-level design and Module A.

2. **Add timing constraints specific to this module.**

The constraint editor can be used to add any module-specific constraints (Figure 6). These are the timing requirements for the design that the module must meet. This step can often be skipped if no constraint information is needed at this time. Again, for module-level timing constraints, the user can enter a PERIOD for any local clocks and port timing using a modular version of the OFFSET constraint. The timing for ports allows the user to specify the path delay requirement for the data to propagate from the port to the setup at the register for inputs, or from the clock pin of the register to the port for outputs. The user can use the Constraints Editor that will show each port on the Ports Tab and create the appropriate OFFSET group/constraint. If the user is editing the UCF by hand, a TPSYNC point must be created using the port net name. Several ports with equivalent time requirements and clocks can use the same TPSYNC name. Then an OFFSET constraint is needed to correlate with each TPSYNC group. The syntax is similar to that of the OFFSET PAD GROUP syntax, except that the TPSYNC group name is used.

> **NOTE:** An OFFSET constraint on a module port must be relative to the actual clock pad net which is likely in the top-level design, and not to the clock port on the module.

Port to Port timing can be specified using he TPSYNC names in a FROM TO constraint.

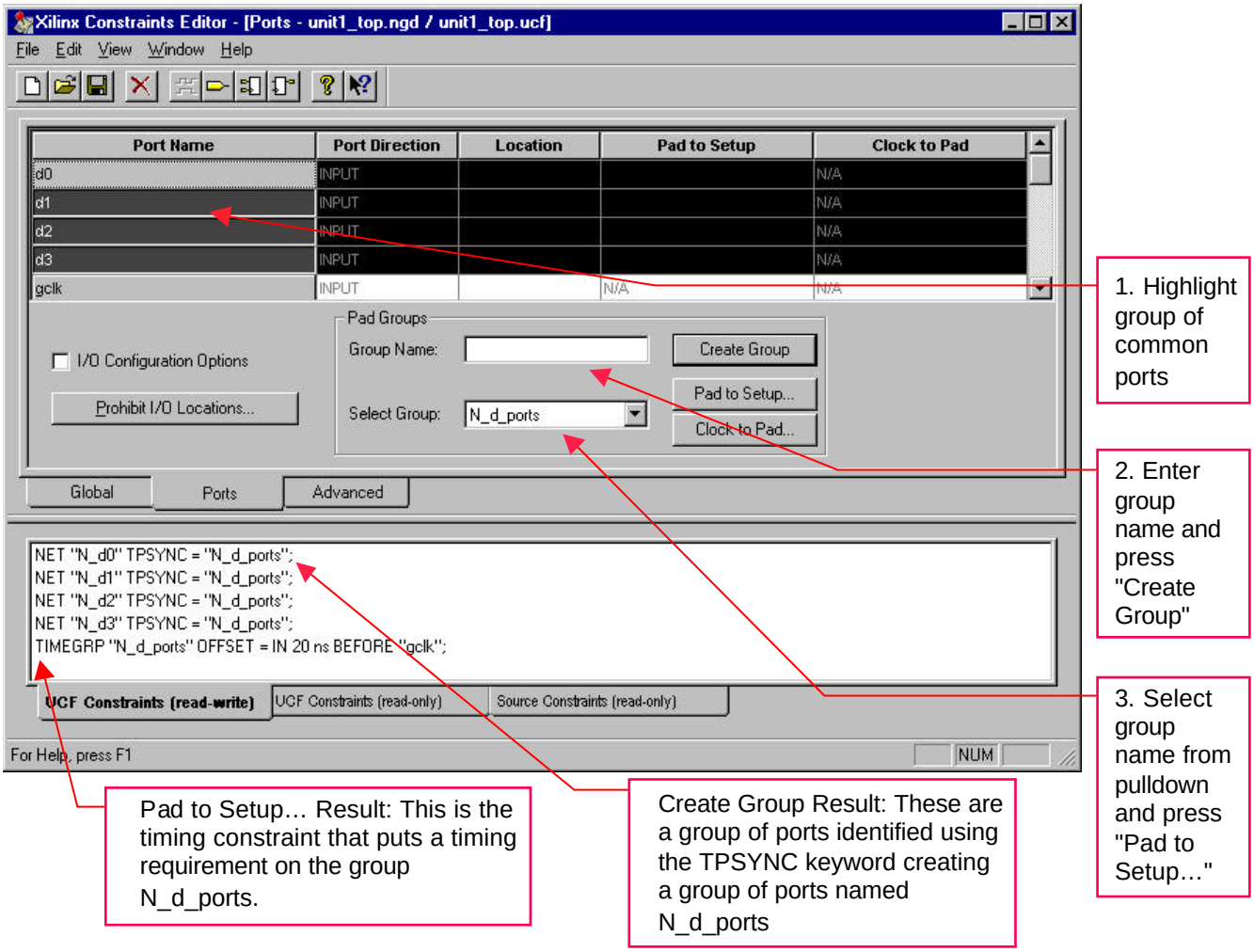


Figure 6: **Example of Constraints Editor Usage**

Remember that the file top.ngd contains the information which is being worked on. The commands used here are:

constraints_editor top.ngd

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

These timing constraints will be saved in the file top.ucf and then folded back into the file top.ngd for later use by the other implementation tool.

3. **Map the logic of the module.**

   Since the range constraints and active module information have been saved in the top.ngd file, no additional switches are required to map this design. All current mapping switches can be used to get the desired results. Map identifies the NGD file as a module implementation and scans the NGD to identify the logical block with the same name as the active module. It then scans each port on the active module and adds pseudo-logic to the port nets so that all will have drivers/loads. (This pseudo-logic will not be propagated into the final design.) The command used here is:

   map top.ngd <additional_map_switches>

   where <additional_map_switches> are any of the allowed switches for the mapper.

   The size and position of each module, referred to as a range constraint, are expanded as a range for each slice within the module and put into the output top.pcf file. All of the information for the active module, as well as the context information for the level design are placed into the output top.ncd file.

4. **Place and route the module.**

   Since the range constraints and active module information have been saved in the top.ncd file generated by the mapper above, no additional switches are required for par to place and route this design. All current par switches can be used to get the desired results. All pseudo-logic will be placed outside the range defined for the module boundary. The command used here is:

   par top.ncd output_ncd_filename <additional_par_switches>

   where <additional_par_switches> are any of the allowed switches for the par.

   The output_ncd_filename is the desired name of the placed and routed ncd file, as distinct from the mapped ncd file. This argument can be used below to speed up the Publish step. If the area described for this module is not large enough or is incorrectly sized to contain all of the physical logic for this module, the top.ucf generated by the Initial Budgeting phase will need to be regenerated. This will require a return to the Initial Budgeting phase and a possible reimplementation of all previously implemented modules.

5. **Floorplan placement information if needed.**

   If the module implementation does not meet timing as found in the reports or is otherwise not satisfactory, then Floorplanner can be used to explicitly reposition logic. All of the normal Floorplanner commands are available for working with this module. The Floorplanner will not notify the user of any attempt to violate any of the assigned range constraints for this module, however. After floorplanning is complete, the map and par steps will need to be rerun. The commands used during this step are :

   floorplanner top.ncd

   map top.ngd ...

   par top.ncd output_ncd_filename ...

   This step should not be required in most cases.

6. **Simulate the module if desired.**

   There are two modes of simulation available at this time. The first is normal simulation using the top-level design as context. The top-level design can be back-annotated and simulated completely. This would be done using the usual commands for correlated back-annotation:

   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 the inactive modules will be undefined, and the signals connected to their ports will be left dangling. Therefore, it will most likely be necessary to probe and/or stimulate these dangling signals in order to yield meaningful simulation results.

   The second mode is to simulate this module independent of the context design. This would be done using the 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 then be instantiated in a test bench that exercises just that module.

   Note that the use of the NGM file in module-only back-annotation is not supported at this time, so the original hierarchy within the module will not be visible in simulation. However, the instance names of registers and RAMs should be preserved, as well as the names of the nets connected to them.

   NOTE: Use of the NGM file in module-only back-annotation is not currently supported.

   Please see the **Simulation Flow** section below for more details on these modes and their operations.

7. **Publish the module implementation information back to the team leader.**

   The final step of an Active Module Implementation phase is to publish the files generated at this time back to the team leader for the Final Assembly phase. These files will need to be published into a directory previously created by the team leader referred to as a PIM directory. A PIM is a *Physically Implemented Module*. These files will be used during the Final Assembly phase as guide information. They will be renamed during the Publish step to accurately reflect the name of the module they are implementing. The new executable "pimcreate" is used to publish module implementation files back to the team leader. The -ncd switch is not required by pimcreate but will speed up its operation by not requiring the tool to load multiple ncd files to determine this information. The command used to publish files back to the team leader (assuming the path to the PIM directory is "pim_path") is :

   pimcreate pim_path -ncd output_ncd_filename

In summary, the following commands are used during Active Module Implementation phase of 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 this point, a single module has been implemented and published back to the team leader for the Final Assembly phase of Modular Design. These steps of the Active Module Implementation phase will need to be repeated for each module in the design.

Final Assembly Phase

The goal of the Final Assembly Phase is to produce a complete design from the information generated in the previous two phases. The team leader or person responsible for the overall design will be responsible for this phase of the design. It can be performed in the original directory where the Initial Budgeting phase was performed once all of the modules have been published back for implementation. It is at this point that all of the previous implementation comes together to create a complete design. If the overall design meets its timing constraints and simulation values, then the process is complete; otherwise, it will be necessary to return to either the Active Module Implementation phase or, in rare cases, the Initial Budgeting phase. This phase can be performed before all the modules have been published for assembly as a validation step, but is most useful when all of the referenced modules have been published.

This phase first builds a complete .ngd file from the .ngo file for the top-level design and the ngo files for each of the referenced modules. It then maps and routes the design using the .ngm and .ncd that were published for each of the referenced modules as guide information. The use of guide information at this step will greatly speed the overall implementation runtime. Since all of the top-level logic has been previously positioned and each module, along with its ports, has been sized and positioned, it should be possible to complete the design as specified. It is possible, though, that resource contention might arise between the modules, even though their assigned areas do not overlap due to their use of global logic or routing resources. It is also possible that the overall design will not meet its timing specifications, even though each module met its timing constraints due to additive delays. If either of these conditions occurs, or if the overall design is otherwise unacceptable, it will be necessary to re-implement one or more modules and repeat this phase. Though it is possible to use the tools during this phase to directly constrain or manipulate resources contained in a module, use of this technique should be minimized since it will render the published module information invalid.

The work for this phase can be performed in the directory where the Initial Budgeting phase was performed, but it can potentially change the top-level constraints file top.ucf such that it should not be used for future module implementation. To this end, it is recommended that a separate directory be used for this phase, with each of the files generated in the Initial Budgeting phase copied into it. Each of the steps of the Final Assembly phase is described below, assuming that information for the top-level design is contained in the files "top.*", the PIMs directory is located at pim_path and each module is named "moda", "modb", and so on.

1. **Create the NGD file for the whole design.**

   Since all of the NGO files for each referenced module have been published into the PIMs directory for this design, ngdbuild can be used to assemble all of these blocks into a complete design. None of the pseudo-logic used in the Active Module Implementation phase appears in any of the associated NGO files, so this extra logic will not be added to the full design. Ngdbuild is run using the "assemble" argument to the "-modular" switch, indicating that the Final Assembly phase of Modular Design has been entered. The path to the PIMs directory is also required, as well as the names of each module implementation to use.

   NOTE: Currently, all modules to be used must be specified, but a future enhancement will allow the absence of any module specification to imply that all published modules should be used.

   The command used here (assuming the use of two published modules "moda" and "modb") is :

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

This will read the previously created constraints file top.ucf in the current directory. The output of this command will be the file top.ngd that represents the logic of the full design. All of the specified PIM information will be written into the top.ngd file for future tool use.

> **NOTE:** If any PIM contains more than one ngo file, it will necessary at this time to also specify the directory of this PIM with the "-sd pim_path/module_name" flag to pick up all of the associated ngo files. This will be fixed in the first Modular Design update service pack.

2. **Map the full design.**

   The mapping of the full design will use the PIM information in the input ngd file to guide the mapping of all of the associated PIM logic. The use of Guide at this step allows any top-level constraints that might need to be pushed into module implementations to be correctly processed. Since the PIM information was placed into the top.ngd file by the previous step, no additional flags are needed to run mapping. All of the current map flags are available at this step to get the desired output. The command to use at this step is:

   map top.ngd <other_map_flags>

   where <other_map_flags> can be any of the currently support mapping flags.

   The output of this step will be mapped ncd file top.ncd and top.ngm. All of the PIM information found in the top.ngd file is also propagated into these files for later use. The map report top.mrp can be consulted to find out how the guide information for each module was processed.

   > **NOTE:** The guiding information in the .mrp file incorrectly takes into account the failure to guide the pseudo-logic that was found in the PIM guide files but not in the top.ngd file. This will be fixed later and should be ignored for now.

3. **Place and Route the full design.**

   The placement and routing of the full design can now be performed on the top-level design. This will use the PIM information previously generated as guide information that will greatly speed the implementation time. The use of Guide at this step enables certain tool optimizations to be performed (though not across module boundaries) which can increase the result quality of the overall design. Since all of the PIM information previously provided is contained in the top.ncd file, no additional flags need be specified for par. The command used here is:

   par top.ncd top_impl.ncd <additional_par_flags>

   where top_impl is a name assigned to the output placed and routed ncd file, and <additional_par_flags> are any of the currently supported par flags.

   These other flags can be used to get the desired result. The par report file top_impl.par can be consulted to view the performance of the overall design along with the associated guide information.

   > **NOTE:** The guiding information in the .par file incorrectly takes into account the failure to guide the pseudo-logic that was found in the PIM guide files but not in the top.ncd file. This will be fixed later and should be ignored for now.

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 matches its timing specifications. This process is described in the next section of this document.

Simulation Flow

Functional and timing simulation can be performed on the individual module during the active module implementation phase as well as to the entire design during the final assembly phase.

Simulating the Module

Simulating the module can be performed during the active module implementation phase. There are two methods to perform simulation:

1. **The module can be simulated directly, independent of the top-level design.** This would be done using 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:

   ngd2ver -tm <module_name> <module_name>.nga

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

   Notice the use of -te and -tm in the VHDL and Verilog flow, respectively. These options are used to rename the top-level entity to module_a, as indicated. Without these options, the resulting VHDL/Verilog entity/module will be "top".

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

   Current Limitations:

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

   - The use of the NGM file in module-only back-annotation is not supported at this time, so the original hierarchy within the module will not be visible in simulation. However, the instance names of registers and RAMs should be preserved, as well as the names of the nets connected to them.

   Timing simulation will reflect timing of components inside the module. However, delay and timing values of module ports should be ignored until a complete design simulation is performed. The boundary timing is meaningless when simulating a module because the loads and drivers of the ports are currently unknown.

2. **The top-level design with its active module can be back-annotated and simulated completely.** This would be done using the usual commands for correlated back-annotation:

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

   VHDL:

   ngd2vhdl top.nga

   Verilog:

   ngd2ver 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 the inactive modules will be undefined, and the

signals connected to their ports will be left dangling. Therefore, it will be necessary to probe and/or stimulate these dangling signals in order 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 the specific tool's documentation for more detail.

In Verilog, users can access internal signals from the testbench as well as from a script/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: Tool Limitations in the First Release

1. The Floorplanner currently gives no indication of the required size of a module, so this information must be inferred from other sources.
2. The Constraints Editor identifies clock signals by recognizing clock pin loads. Because of this, the Constraints Editor will not recognize a clock signal during Initial Budgeting mode unless there is a clock pin load at the top level of the design.
3. An OFFSET constraint on a module port must be relative to the actual clock pad net (which is likely in the top-level design), and not to the clock port on the module).
4. The NGM file is not valid when simulating using the -module switch.
5. Map guide information in report file refers to failure to guide pseudo-logic.
6. Par guide information in report file refers to failure 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.
06/19/00	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.