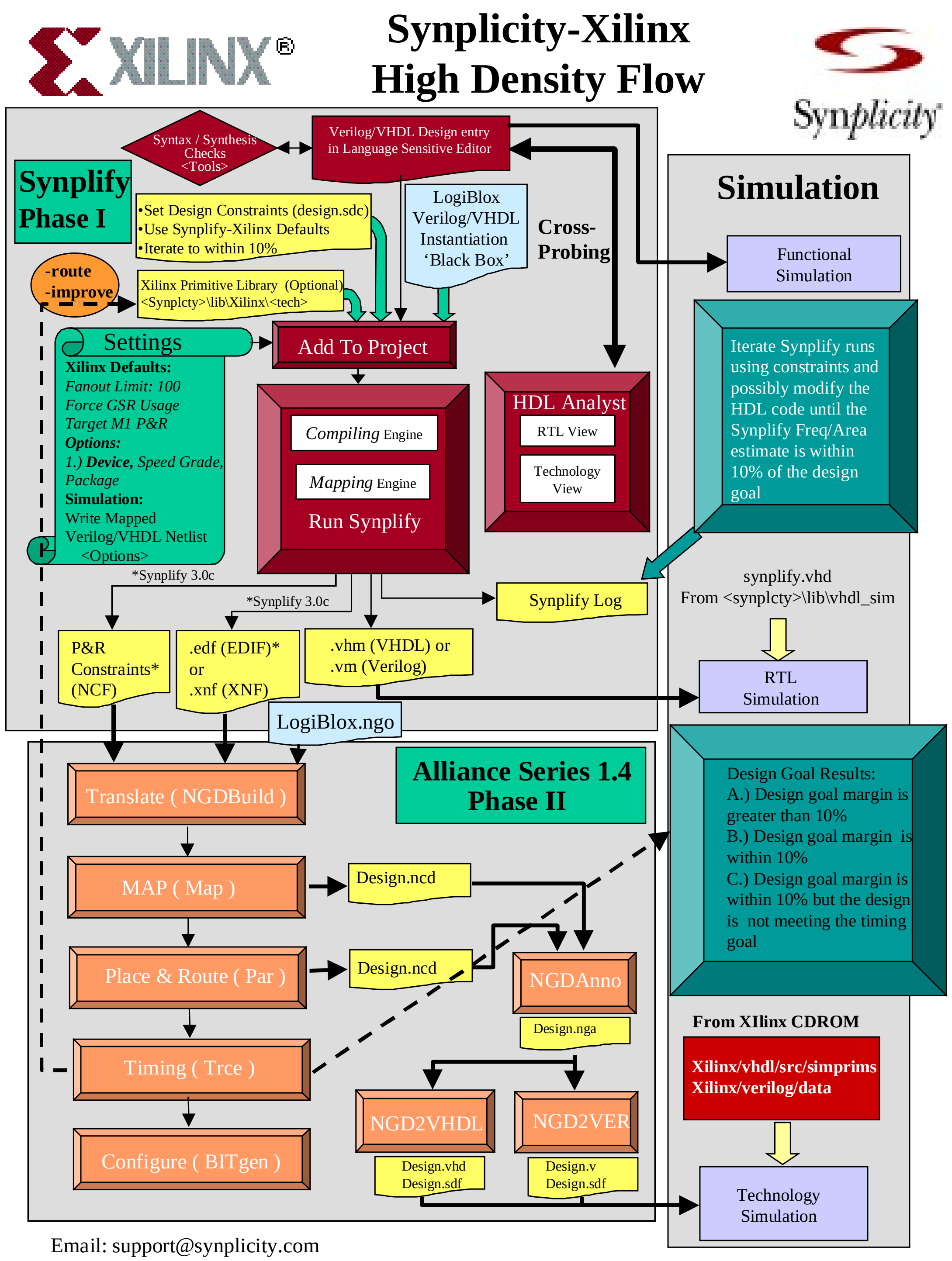
Synplicity-Xilinx High Density Methodology

This High Density Methodology note is intended to assist designers who are using Synplicity and Xilinx to a design high density FPGA (125K –150K gates). The recommended settings and flow phases are based on the assumption that the user would like to tune the circuit performance (area/speed) from Synplify and A1.4 (M1) for high density FPGA designs. Phase I describes synthesis specific techniques and Phase II describes implementation specific (P&R) techniques for optimizing a design for speed.

Flow Overview


Synplicity-Xilinx
High Density Flow
XILINX
Synplicity
Syntax / Synthesis Checks <Tools>
Verilog/VHDL Design entry in Language Sensitive Editor
Synplify Phase I
•Set Design Constraints (design.sdc)
•Use Synplify-Xilinx Defaults
•Iterate to within 10%
LogiBlox Verilog/VHDL Instantiation 'Black Box'
Cross-Probing
-route
-improve
Xilinx Primitive Library (Optional) <Synplcty>\lib\Xilinx\<tech>
Settings
Xilinx Defaults:
Fanout Limit: 100
Force GSR Usage
Target M1 P&R
Options:
1.) Device, Speed Grade, Package
Simulation:
Write Mapped Verilog/VHDL Netlist
<Options>
Add To Project
Compiling Engine
Mapping Engine
Run Synplify
HDL Analyst
RTL View
Technology View
Simulation
Functional Simulation
Iterate Synplify runs using constraints and possibly modify the HDL code until the Synplify Freq/Area estimate is within 10% of the design goal
*Synplify 3.0c
*Synplify 3.0c
Synplify Log
synplify.vhd
From <synplcty>\lib\vhdl_sim
P&R Constraints* (NCF)
.edf (EDIF)* or .xnf (XNF)
.vhm (VHDL) or .vm (Verilog)
RTL Simulation
LogiBlox.ngo
Alliance Series 1.4 Phase II
Translate (NGDBuild)
MAP (Map)
Design.ncd
Place & Route (Par)
Design.ncd
NGDAnno
Design.nga
Timing (Trce)
Configure (BITgen)
NGD2VHDL
Design.vhd
Design.sdf
NGD2VER
Design.v
Design.sdf
Design Goal Results:
A.) Design goal margin is greater than 10%
B.) Design goal margin is within 10%
C.) Design goal margin is within 10% but the design is not meeting the timing goal
From XIlinx CDROM
Xilinx/vhdl/src/simprims
Xilinx/verilog/data
Technology Simulation

Email: support@synplicity.com

Phase I: Synthesis

Phase I focuses the designer's efforts on tuning Synplify results for area and/or speed. Designer's can direct the Synplify optimization engine to concentrate efforts on the timing critical potions of designs by setting design specific constraints. Synplify supports user defined design constraints that can be applied to clocks, registers, inputs, outputs, and multi-cycle paths. Synplicity recommends that designers set design constraints to closely match the design goal within +- 5%. By setting realistic design constraints, the user insures that the implementation engine (P&R) will not be "over constrained" when Synplify supports forward annotation of P&R constraints (Synplify 3.0c). Designers should be aware that over-constraining the design can decrease routeablity and increase P&R run times.

Phase I - Three Step Approach to Achieve the Highest Quality of Results from Synplify

- Step 1 - Specify Design Constraints
- Step 2 – Set Synplify-Xilinx Device Options
- Step 3 - Iterate through the Synplify flow to achieve +- 10% of the design goal for area and/or speed

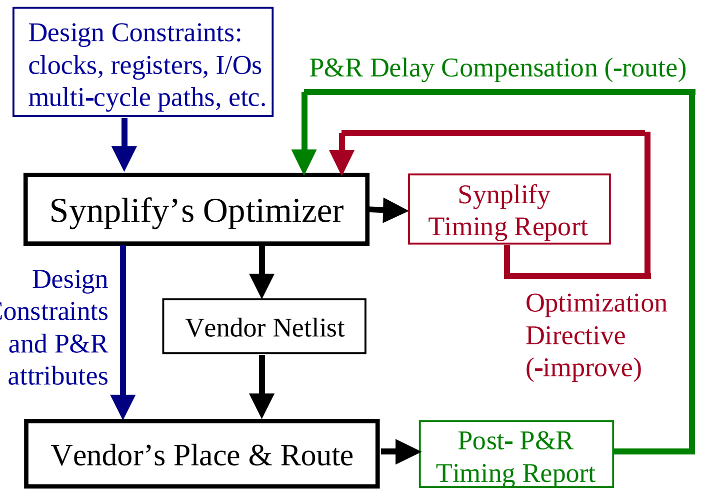
Step1 : Set Design Constraints

Design constraints should be defined in a Synplify constraint file (*.sdc) and included along with your HDL source files in the Synplify project window. The design constraints described in detail below are used to direct the Synplify optimization engine and will be passed on to P&R via a netlist constraint file (NCF) automatically created by Synplify 3.0c or with a user constraint file (UCF) generated by the user for Synplify 3.0b/3.0a/3.0. Please refer to the A1.4 (M1) Basic UCF Timing Constraints in Appendix II for details on defining Place and Route constraints.

Synthesis Constraints Optimization Environment

SCOPE™

- **Design Constraints**
  - **clocks**
  - **registers**
  - **I/Os**
  - **Multi-cycle paths**
- **Optimization Directives**
  - **-improve**
- **P&R Delay Compensation**
  - **-route**



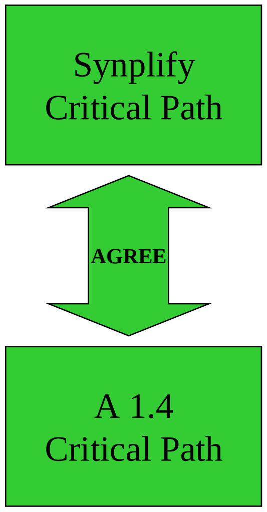
Design Constraints: clocks, registers, I/Os multi-cycle paths, etc.
P&R Delay Compensation (-route)
Synplify's Optimizer
Synplify Timing Report
Design Constraints and P&R attributes
Vendor Netlist
Optimization Directive (-improve)
Vendor's Place & Route
Post- P&R Timing Report

SCOPE:
In Synplify 3.0, there is a new class of constraint options, -improve and –route, that are intended to focus critical path re-optimization and factor in actual place and route delay into the design optimization process. These constraint options are not passed onto the place and route engine since their primary purpose is to 'nudge' the Synplify optimization engine. The –improve and -route constraint options allow the user to fine tune a timing critical design without over constraining the place and route engine.

Using the "-improve <ns>" option forces Synplify to restructure the design during optimization to reduce the timing paths by the specified number of nanoseconds to try and meet the clock frequency goal. The -improve option will not affect the Synplify timing reports because these delays are exclusively an optimization 'nudge'.

Using the "-route <ns>" option forces Synplify to use that additional route delay in its calculations to try and meet the clock frequency goal. The -route option will include additional delay in the Synplify timing reports because the specified delays represent the actual P&R delays. Please note that the –route argument is determined only after the designer has taken a design through Phase I and Phase II of the High Density Flow. The A1.4 (M1) Timing (Trce) output determines the amount of 'route delay' (ns) that should be used as the argument to the –route option.

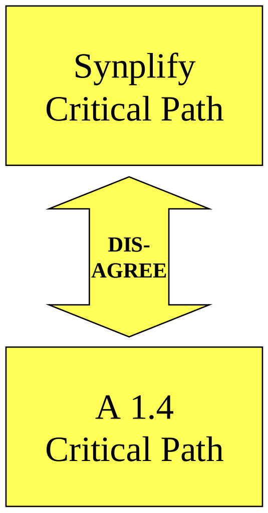
-improve



The critical path is the same in Synplify and Alliance 1.4 (P&R) and the design is not meeting the performance goal.

Use -improve to force Synplify to restructure the design during optimization to try and meet the clock frequency goal.

-route



The critical path is different in Synplify and Alliance 1.4 (P&R) and the design is not meeting the performance goal.

Use -route to force Synplify to add route delay in its calculations to try and meet the clock frequency goal.

Graphical User Interface Controls:

The Synplify graphical user interface allows designers to set a global clock frequency and enable the Symbolic FSM compiler.

Frequency (MHz)

By entering the desired clock frequency goal in the "Frequency (MHz)" control box , the designer can specify a global clock frequency constraint .The frequency will be applied to all the clocks found in the design. For designs which contain multiple clocks, the designer should set an individual constraint per clock in a Synplify constraint file (*.sdc). When using a Synplify constraint file (design.sdc), it is recommended that the user additionally set the Frequency (MHz) control in the Synplify project window. Some optimization decisions are based on the default timing value (MHz) specified in the UI. Setting the Frequency (MHz) control in the Synplify project window helps to improve your synthesis results. If there is more than one clock in the design set the Synplify "Frequency (MHz)" control in the Project form to your desired Global Default Clock frequency. Please reference the Clock Constraint sub-section of this application note for further details on how to set constraints for multiple clocks.

Symbolic FSM Compiler

By enabling the Symbolic FSM Compiler, Synplify will automatically recognize and extract the state machines in the design and perform the Symbolic FSM Compiler optimizations. The optimizations include Reachability Analysis, Next State Logic Optimization, and State Machine Re-Encoding (One Hot). Please reference Synplify On-Line Help or the Synplify User's Guide for details about the Symbolic FSM Compiler.

The traditional methods used to generate state machine logic result in highly, encoded states. State machines with highly, encoded state variables typically have a minimum number of flip-flops and wide combinatorial functions. These characteristics are acceptable for PAL and gate array architectures. However, because FPGAs have many flip-flops and narrow function generators, highly, encoded state variables can result in inefficient implementation in terms of speed and density.

One-hot encoding allows you to create state machine implementations that are more efficient for FPGA architectures. You can create state machines with one flip-flop per state and decreased width of combinatorial logic. One-hot encoding is usually the preferred method for large FPGA-based state machine implementation. For small state machines (fewer than 8 states), binary encoding may be more efficient. To improve design performance, you can divide large (greater than 32 states) state machines into several small state machines and use the appropriate encoding style for each.

Clock Constraints

Clock Constraints allow the user to specify a specific frequency goal for synthesis. All clocks contained with in a design should be constrained regardless of the relative speed.

```
define_clock <clock_name> {-freq <MHz> | -period <ns>}
```

Examples:

```
define_clock CLK1 -freq 33.0
define_clock CLK2 -freq 15.0
define_clock CLK3 -period  100.0
```

Register Constraints:

Use the "define_reg_input_delay" timing constraint to speed up paths feeding into a register by a given number of nanoseconds.

```
define_reg_input_delay <register_name> [-improve <ns>]  [-route <ns>]
```

Use the "define_reg_output_delay" timing constraint to speed up paths coming from a register by a given number of nanoseconds.

```
define_reg_output_delay <register_name>  [-improve <ns>]  [-route <ns>]
```

Example:

```
# Try harder and reduce the clock period on register_a by 1.0 ns and factor in a 2.0 ns route delay.
define_reg_input_delay register_a   -improve 1.0 –route 2.0
```

Input Constraints:

Use the "define_input_delay" timing constraint to specify the input delay, i.e. the delay before the signal arrives at the input pin.

```
define_input_delay {<input_port_name> | -default } <ns>  [-improve <ns> ]  [-route <ns> ]
```

Examples:

```
# Set the default input delay on all inputs to 3.0 ns.
define_input_delay -default 3.0
# Set the input delay on input_a to 10.0 ns.
define_input_delay input_a 10.0
```

Output Constraints:

Use the "define_output_delay" timing constraint to specify the output delay, i.e. the delay of the logic outside the FPGA that is driven by the design outputs.

```
define_output_delay { <output_port_name> | -default } <ns>  [-improve <ns> ]  [-route <ns> ]
```

Examples:

```
# The output delay for output_a is 5.0. Also, try harder and reduce the clock period by 3.0 ns.
define_output_delay output_a 5.0 -improve 3.0
# The routing delay turned out to be 1.8 ns more than predicted by Synplify.
# Also, in this example, output_a has a 10.0 ns output delay.
# Rerun synthesis trying to improve results by 1.8, by adding 1.8 ns to the timing calculations.
define_output_delay output_a 10.0 -route 1.8
```

Multicycle Path Constraints:

Use the "define_multicycle_path" timing constraint to specify that certain paths use multiple clock cycles and that Synplify should not try to report those paths as a violation of the design frequency timing goal.

```
define_multicycle_path  { -from <register_or_input> | -to <register_or_output> } <num_clock_cycles>
```

Example:

```
#paths to register_c use 2 clock cycles
define_multicycle_path  -to register_c 2
```

Please reference the Synplify On Line Help or the Synplify User’s Guide for more details about setting timing constraints.

Step 2 – Set Synplify-Xilinx Device Options

The Synplify-Xilinx device option settings default to the following:

Technology: (User Selectable)
Part: (User Selectable)
Package: (User Selectable)
Speed Grade: (User Selectable – Defaults to the fastest speed grade)
Fanout Limit: 100
Force GSR Usage
Disable I/O Insertion (Turned Off)
Target M1 P&R

It is recommended the user set the Xilinx Technology, Part, Package, and Speed Grade option for the target device based on design goals. Synplify automatically sets intelligent default options and the tool provides flexibility to modify these settings on a per design basis.

Fanout Limit:

Synplify automatically maintains reasonable fanout limits, and it also provides you the flexibility to specify maximum fanout guidelines. To set the fanout limit, choose Set Device Options from the Target menu. Type an integer to represent the guideline for the number of fanouts for a given driver. If you do not set the fanout limit, it defaults to one hundred. (The minimum you can set the fanout limit to is eight.)

Large fanouts can cause large delays and routability problems. During technology mapping, Synplify tries to keep the fanout under the fanout limit. Synplify uses the fanout limit as a guideline, and not as a hard limit. Synplify first reduces fanout by replicating the driver of the high fanout net and splitting the net into segments. This replication may affect the number of register bits in your design. If replication is not possible, then Synplify buffers the signals. Buffering is more expensive both in terms of intrinsic delay and consumption of resources, and is therefore not used until a slightly higher fanout limit than specified.

For extremely large fanout nets you may want to consider inserting a global buffer (BUFGS) in your design. In addition to reducing delay for a large fanout net, it can free up routing resources for other signals.

The Log File contains a Net Buffering Report that shows how many nets were buffered or had their source replicated, and the number of segments created for the net.

Click the View Log button to display the Net Buffering Report.

Force GSR Usage:

The XC4000 and XC5200 families have a start up block for global set and reset. Synplify will automatically create a startup block to access the GSR resource (Global Set Reset) if it is appropriate for your design. The GSR resource is a pre-routed signal that goes to the reset input of each flip-flop in the FPGA. Using this resource instead of general routing for a reset signal can have a significant positive impact on the routability and performance of your design.

By default, if there is a single reset used in your design, then Synplify will connect that reset signal to a STARTUP block. It will do this even if some flip-flops have no reset at all. Usually, flip-flops without set or reset may be safely initialized because the reset is only used when the device is turned on. If this is not the case, then you will need to turn off the Force GSR Usage option. To turn off the Force GSR Usage option, choose Set Device Options from the Target menu, and deselect the checkbox. When Synplify sees this option turned off, Synplify requires that all flip-flops have resets and that the resets are all the same before it uses GSR.

If multiple resets are used in the design, then Synplify will not automatically create a startup block for GSR. If you still want one of the reset signals to be used for GSR then you will need to manually instantiate a STARTUP_GSR module in your design source file.

To assist with instantiation of startup blocks for the XC4000 series, Synplify includes files "synplcty\lib\xilinx\xc4000.v", and "synplcty\lib\xilinx\xc4000.vhd". These files contain modules or components called STARTUP_GSR, STARTUP_GTS, and STARTUP_CLK that you can directly instantiate. Synplify will merge these three startup blocks to form a single STARTUP block in the resulting XNF file, so you can use them independently. You can use these files as templates for startup blocks for the XC5200 series. For more information on the startup block, please see the XACT Libraries Guide.

Disable I/O Insertion :
Synplify automatically inserts input/output (I/O) pads into your design. You may manually insert I/Os; in that case, Synplify will insert I/Os only for the pins that need them.

If you do not want Synplify to automatically insert any I/Os into your design, click the "Disable I/O Insertion" checkbox in the Synplify Set Device Options form. This is useful to see how much area your blocks of logic take up, before synthesizing to an entire FPGA.

Note: If you disable automatic I/O insertion, you will not get any I/Os in your design unless you manually instantiate them yourself.

Please refer to the Synplify On Line Help or the Synplify User's Guide for details about the Xilinx device option settings.

Step 3 – Iterate through the Synplify flow to achieve +-10% of the design goal for area and/or speed

Once the designer has specified the input source files, set timing constraints, and selected the target architecture and options, the design can be synthesized by clicking on the Synplify RUN button. The area and speed results of the Synplify run are saved into the Synplify log file (design.srr). The Synplify log file contains an estimate of the operating frequency of your design and an estimated number of device specific resources used. If the log file estimate of area and/or speed differs by more than 10% from the design goal, the designer should iterate Phase I Synplify runs adjusting constraints to improve timing and possibly modifying the HDL code (LogiBLOX) to improve area/timing based on critical path feedback from the RTL and Technology viewer until the Synplify log file estimate is within 10% of the design goal. If the log file estimate of area and/or speed is within 10% of the design goal, the designer should continue onto Phase II of the High Density Flow.

LogiBLOX
LogiBLOX is a graphical interactive design tool, from Xilinx, that is used for creating high-level modules such as counters, shift registers and multiplexers. LogiBLOX includes both a library of generic modules and a set of tools for customizing them. The modules created with LogiBLOX can be instantiated in Verilog/VHDL source code as a "black boxes" for use with Synplify. Please see Appendix I for additional information regarding Xilinx LogiBLOX implementation details.

Synplify-Xilinx Macro Support:
You can create an instance of any externally defined macro (LogiBLOX: arithematic operators & memories) , including a user-defined macro or Xilinx macro such as an I/O or flip-flop, by instantiating what Synplify calls a "black box" in your Verilog or VHDL source files. These black boxes are Verilog empty module descriptions or VHDL component declarations.

For instantiating Xilinx macros like gates, counters, flip-flips, or I/Os, you can use the Synplicity-supplied Xilinx Macro Libraries that pre-define the Xilinx macro "black boxes".

Using the Xilinx Macro Libraries With Verilog Designs

The Synplify Xilinx Macro Libraries contain pre-defined black boxes for the Xilinx macros so that you can manually instantiate them into your design. Simply add the appropriate Xilinx Macro Library file name in the Source Files list box at the top of the list. The macro libraries are located in the "synplcty\lib\xilinx" directory, and are called xc3000.v, and xc4000.v. Use the macro library file that corresponds to your target architecture.

Using the Xilinx Macro Libraries With VHDL Designs

The Synplify Xilinx Macro Libraries contain pre-defined black boxes for the Xilinx macros so that you can manually instantiate them into your design. Simply add the appropriate library and use clauses to the top of your files that instantiate the macros.

Syntax

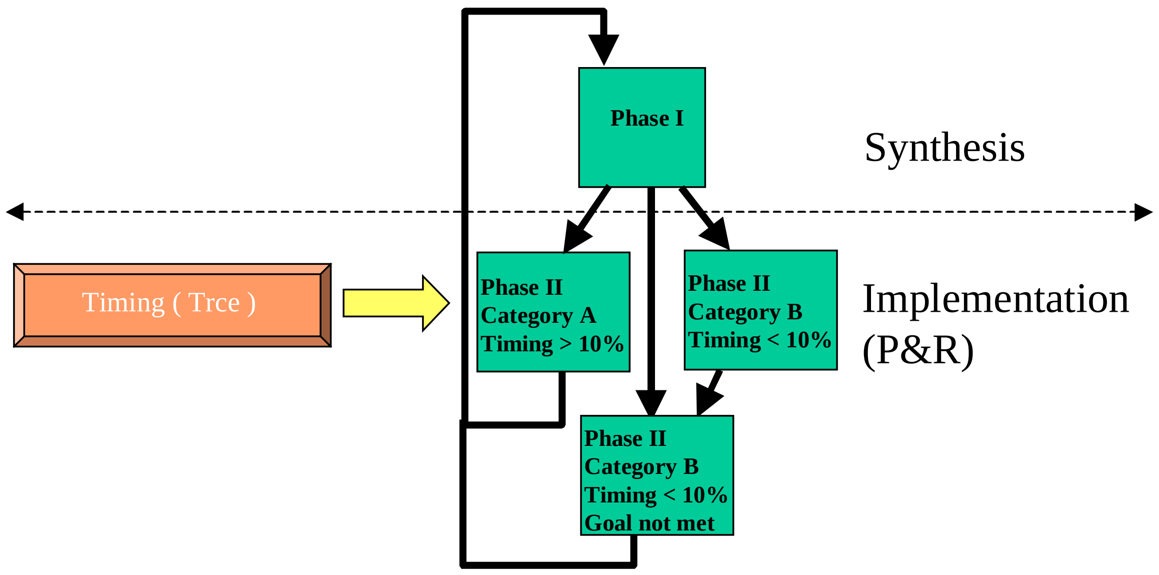
```
library <family> ;
use <family>.components.all;
where <family> is xc3000, xc4000 or xc5200.
```

Example

```
library xc4000;
use xc4000.components.all ;
```

<u>Phase II: Implementation (P&R)</u>

Phase II focuses the designer's efforts on tuning A1.4 (M1) results for speed. The designer should review the output results from Timing (Trce) to determine which Phase II result category A, B, or C correlates to the design.


Phase I
Synthesis
Timing (Trce)
Phase II Category A Timing > 10%
Phase II Category B Timing < 10%
Implementation (P&R)
Phase II Category B Timing < 10% Goal not met

Implementation Result Categories:
There are three Implementation Result Categories that will assist the designer with 'next steps' based on the post place and route frequency and area results for the design. Please refer to the Better Managing Large XL Designs in M1 in Appendix III for recommended PAR specific strategies.

A.) The design goal margin after P&R timing is greater than 10%

1.) The designer should iterate through Phase I* to try and achieve +/-5% margin after synthesis and then proceed onto Phase II.
[*Iterate Phase I Synplify runs adjusting constraints (-improve, -route) to improve timing and possibly modifying the HDL code to improve area/timing based on critical path feedback from the RTL and Technology viewer until the Synplify log file estimate is within 10% of the design goal.]

B.) The design goal margin after P&R timing is within 10%.

1.) Set the PAR effort level* to 3 in order to achieve a faster result and the designer should expect a normal P&R run time.
To run PAR in this manner from the command line, use:
```
par -l 3 <design.ncd> <design_r.ncd>
```
From the Design Manager, flow engine GUI, the same thing can be accomplished:
Design-> Implement -> Options -> Edit Implementation Template -> Place and Route
2. Move the bar to the 3rd position from the left most 'tic'. (toward Best Results)

2.) Set the PAR effort level* to 4 in order to achieve a faster result and the designer should expect a longer P&R run time.
To run PAR in this manner from the command line, use:
```
par -l 4 <design.ncd> <design_r.ncd>
```
From the Design Manager, flow engine GUI, the same thing can be accomplished:
Design-> Implement -> Options -> Edit Implementation Template -> Place and Route
Move the bar to the 4th position from the left most 'tic'. (toward Best Results)

[*The Runtime (PAR effort level) default is 2]

Please note that Implementation results after Result Category B may direct the designer to Result Category C.

C.) The design goal margin is within 10% but the design is not meeting the required timing goal.

1.) Optionally change the Synplify device options speed grade in Phase I and run Synplify and A1.4 to establish the design speed and area.
2.) Optionally run Multi-Pass Place and Route .

To run PAR in this manner from the command line, use:
```
par -s 3 -n 10 -t 2 -i 3 <design.ncd> <design_r.ncd> <design.pcf>
```
From the Design Manager, flow engine GUI, the same thing can be accomplished:
Step1:
Design-> New Version (Open a new version of the design to run Multi-Pass P&R)
Design-> FPGA Multi-Pass Place & Route…
Set the following:
Starting Strategy: 2
Iterations to Attempt: 10
Iterations to Save: 3
Step 2
Design -> Implement->Options -> Edit Implementation Template -> Place and Route
Set Router Options to Run 3 Routing Passes.

These settings will direct P&R to start at Cost Table 2, iterate through 10 attempts, and save the top 3 scores. It is recommended the designer save the top 3 Cost Table scores in the event that the design requires additional P&R runs to meet the timing goals. When the P&R run is completed the designer should use re-entrant routing on the design with the lowest design score in the PAR report to complete the design.

3.) Return to Phase I and adjusting constraints and/or possibly modify the HDL code based on critical path feedback from the RTL and Technology viewer until the Synplify log file estimate is within 5%.

APPENDIX I
<u>LogiBLOX Instantiation</u>

Creating the LogiBLOX Module

1. Launch the LogiBLOX Gui

-In WinNT or Win95, select the LogiBLOX icon from the Xilinx Group
-In Unix, execute lbgui

2. In the LogiBLOX Setup window make the following selections

-Vendor: Other
-Bus Notation: B<I>
-Browse for project directory
-Choose desired device family
-Select NGO generatation
-Select HDL template

3. After Pressing OK, create desired module.

RAM Module Example

A. Make the following Selections

-Module Name: ramblx
-Module Type: Memories
-Data Bus Width: 8
-Memory Depth: 16

B. Select OK.
C. Instantiation

Verilog

-Cut and paste the module stub definition from the .vei file.
The .vei is the Verilog template for the module produced by LogiBLOX.
-Add /* synthesis black_box */ to the module declaration just before the semicolon ";"

example

```
module raminst (Addr, Daddr, Sout, Dout, Din, Wren, Wclk);

input [3:0] Addr, Daddr;
input [7:0] Din;
input Wren, Wclk;
output [7:0] Sout, Dout;


ramblx modram(.A(Addr),.DPRA(Daddr),.SPO(Sout),.DPO(Dout),.DI(Din),
         .WR_EN(Wren),.WR_CLK(Wclk));

endmodule

module ramblx(A, SPO, DI, WR_EN, WR_CLK, DPO, DPRA) /* synthesis black_box */;
input [3:0] A;
output [7:0] SPO;
input [7:0] DI;
input WR_EN;
input WR_CLK;
output [7:0] DPO;
input [3:0] DPRA;
endmodule
-------------------------------------------------------------------------------
```

VHDL

-Cut and paste the component declaration from the .vhi file. The .vhi is the template for the module created by LogiBLOX.

-Declare black_box as an attribute with boolean value then set the black_box attribute on the LogiBLOX module as true.

example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity raminst is
port (
  Addr,Daddr            : in std_logic_vector(3 downto 0);
  Din                   : in std_logic_vector(7 downto 0);
  Wren, Wclk            : in std_logic;
  Sout, Dout            : out std_logic_vector(7 downto 0));
end raminst;

architecture structural of raminst is

attribute black_box:boolean;

component ramblx
  PORT(
    A: IN std_logic_vector(3 DOWNTO 0);
    DI: IN std_logic_vector(7 DOWNTO 0);
    WR_EN: IN std_logic;
    WR_CLK: IN std_logic;
    DPRA: IN std_logic_vector(3 DOWNTO 0);
    SPO: OUT std_logic_vector(7 DOWNTO 0);
    DPO: OUT std_logic_vector(7 DOWNTO 0));
end component;

attribute black_box of ramblx: component is true;
begin
Modram: ramblx port map (A=>Addr, DPRA=>Daddr, DI=>Din, WR_EN=>Wren,
      WR_CLK=>Wclk, SPO=>Sout, DPO=>Dout);
end structural;
```

D. Synthesize

For more information about LogiBLOX, refer to the dynatext online manual LogiBLOX Reference/User Guide.
For more information on black box instantiation in Synplify, please refer to the Synplify User’s Guide Page 337.

APPENDIX II
<u>Basic UCF Timing constraints</u>

Place and Route timing constrains should realistic +-5% of the design goal. The User Constraint File is passed as an argument to the NGDBuild executable or specified in the Flow Engine GUI (Design Manager) menu bar and form selections :Design-> Implement -> Options, User Constraints.

The User Constraints File contains the design specific constraints which can be passed to the P&R engine. The UCF file is read by ngdbuild by using the command line argument [–uc <ucf_file[.ucf]>].

You can define timing specifications for your design in the User Constraints File (UCF). The UCF gives you tight control of the overall specifications by giving you access to more types of constraints; the ability to define precise timing paths; and the ability to prioritize signal constraints. Furthermore, you can group signals together to simplify timing specifications. For more information on timing specifications in the UCF file, refer to the Quickstart Guide for Xilinx Alliance Series, the Libraries Guide, and the Answers Database on the Xilinx Web site.

1. Clock Constraints

The PERIOD timespec covers all timing paths that start or end at a register, latch, or synchronous RAM which are clocked by the reference net (excluding pad destinations). Also covered is the setup requirement of the synchronous element relative to other elements (ex. flip flops, pads, etc.).

NOTE: The default unit for time is nanoseconds.

Examples:

```
NET clk20MHz PERIOD = 50;
NET clk50mhz TNM = TNM_Name;
TIMESPEC TS01 = PERIOD : clk50mhz : 20 ;
```

For single clock designs or designs that don't share data between clock domains , use the following clock definition, or the source to destination can use the destination PERIOD delay:

```
NET clock PERIOD = 50 ;
```

For designs that have more than one clock and data is passed between clock domains and the path delay is not the destination PERIOD value, use the following method:

```
NET clock1 TNM = clock1 ;
TIMESPEC TS_clock1 = PERIOD clock1 50 ;
NET clock2 TNM = clock2 ;
TIMESPEC TS_clock2 = PERIOD clock2 40 ;
TIMESPEC TS_clock1_clock2 = FROM clock1 TO clock2 100 ;
TIMESPEC TS_clock2_clock1 = FROM clock2 TO clock1 80 ;
```

The PERIOD constraint is the easiest for global clock TIMESPEC'ing. The OFFSET spec for the I/O accounts for the global clock delay, but it requires careful naming of the I/O and produces a separate reportable constraint for each I/O. This can lead to long reports that are difficult to read from the Xilinx Trce program.

The PAD to FFS and FFS to PAD constraints don't take the clock delay into account. The OFFSET constraint would be more appropriate for the remaining pad related constraints. The general recommendation is to use a PERIOD constraint with some OFFSET constraints.

One note though, it seems to take longer to process the PERIOD and OFFSET combination through the Trce timing analysis tools. A FFS to PAD and PAD to FFS constraints, and subtract off or add in the clock delay to create the spec, this takes less time to process.

So, because we are talking about system performance, you need to add the minimum clock delay to your PADS to FFS specs and subtract the maximum clock delay from your FFS to PADS specs.
It sounds opposite, but it is correct. An example would help. Let's say,

25 MHz internal timing is easy - just use a period constraint of 50ns.

Then, lets say that you have a BUFGLS that is distributing the clock. If the max time in the data book is about 5ns, the minimum time might be somewhere around 3ns. It is a guess, but an educated one.

The PADS to FFS spec should be set to 50ns + 3ns. This is because when you look at the overall timing from a system point of view, you will take the number reported by the tools and subtract off the minimum clock delay. So if a path from a PAD to a FFS was reported as 53ns by the tools, it would actually meet the 50ns desired timing.

In the FFS to PADS case, you would use a 50ns - 5ns spec. This would result in possible worst case paths of 45ns. Then, when you add in the worst case clock delay, you end up with a system clock to out of 50ns. Again, this is the desired timing.

Naming and Grouping Signals Together

You can name and group signals with TNMs (Timing Names) or with TIMEGRPs (Time groups). TNMs and TIMEGRPs are placed on these start and endpoints: ports, registers, latches, or synchronous RAMs. The new specification, TPSYNC, allows you to define an asynchronous node for a timing specification.

2. TNMs

Timing Names are used to identify a port, register, latch, RAM or groups of these components for timing specifications. TNMs are specified from a UCF with the following syntax.

```
INST Instance_Name TNM=TNM_Name;
```

Instance_Name is the name given to the port, register, latch or RAM in your design. The instance names for any port or instantiated component are provided by you in your HDL code. Inferred flip-flops and latch names can usually be determined from the log files. TNM_Name is the arbitrary name you give the timing group.

You can include several of these statements in the UCF file with a common TNM_NAME to group elements for a timing specification as follows.

```
NET DATA TNM=INPUT_PORTS;
NET SELECT TNM=INPUT_PORTS;
```

The above example takes two ports, DATA and SELECT, and gives
them the common timing name INPUT_PORTS.

3. TIMEGRPs

Time Groups are another method for specifying a group of components for timing specifications. Time groups use existing TNMs or TIMEGRPs to create new groups or to define new groups based on the output net that the group sources.   There are several methods to create TIMEGRPs in the UCF file, as follows.

```
TIMEGRP TIMEGRP_Name=TNM1:TNM2;
TIMEGRP TIMEGRP_Name=TNM3:EXCEPT:TNM_4;
```

The Xilinx software recognizes the following global timing names.

- FFS - All flips-flop in your design
- PADS - All external ports in your design
- RAMS - All synchronous RAMs in your design
- LATCHES - All Latches in your design

The following time group specifies the group name, FAST_FFS, which consists of all flip-flops in your design except for the ones with the TNM or TIMEGRP SLOW_FFS attribute.

```
TIMEGRP FAST_FFS=FFS EXCEPT SLOW_FFS;
```

4. TPSYNC Specification

In the latest version of the Xilinx software, you can define any node as a source or destination for a timing specification with the TPSYNC keyword.   In synthesis designs, it is usually difficult to identify the net names for asynchronous paths of inferred logic. These net names can change from compile to compile, so it is not recommended to use this specification with inferred logic. However, with instantiated logic, the declared SIGNAL or WIRE name usually remains intact in the netlist and does not change from compile to compile. The UCF syntax is as follows.

```
NET Net_Name TPSYNC=TPSYNC_Name;
```

In the NET statement shown above, the TPSYNC is attached to the output net of a 3-state buffer called BUS3STATE. If a TPSYNC is attached to a net, then the source of the net is considered to be the endpoint (in this case, the 3-state buffer itself). The subsequent TIMESPEC statement can use the TPSYNC name just as it uses a TNM name.

In the following NET statement, the TPSYNC is attached to the output net of a 3-state buffer, BUS3STATE. If a TPSYNC is attached to a net, then the source of the net is considered to be the endpoint (in this case, the 3-state buffer itself). The subsequent TIMESPEC statement can use the TPSYNC name just as it uses a TNM name.

```
NET BUS3STATE TPSYNC=bus3;
TIMESPEC TSNewSpc3=FROM:PAD(ENABLE_BUS):TO:bus3:20ns;
```

5. FROM:TO Specification

FROM:TO style timespecs can be used to constrain paths between time groups.

NOTE: Keywords: RAMS, FFS, PADS, and LATCHES are predefined time groups used to specify all elements of each type in a design.

```
TIMESPEC TS02 = FROM : PADS : TO : FFS  : 36 ;
TIMESPEC TS03 = FROM : FFS  : TO : PADS : 36 ns ;
TIMESPEC TS04 = FROM : PADS : TO : PADS : 66 ;
TIMESPEC TS05 = FROM : PADS : TO : RAMS : 36 ;
TIMESPEC TS06 = FROM : RAMS : TO : PADS : 35.5 ;
```

5. OFFSET timespec

To automatically include clock buffer/routing delay in your "PADS:TO: synchronous element> or <synchronous element>:TO:PADS timing specifications, use OFFSET constraints instead of FROM:TO constraints.

Examples:
For an input where the maximum clock-to-out (Tco) of the driving device is 10 ns:

```
NET in_net_name OFFSET = IN : 10 : AFTER : clk_net_name ;
```

For an output where the minimum setup time (Tsu) of the device being driven is 5 ns:

```
NET out_net_name OFFSET = OUT : 5 : BEFORE : clk_net_name ;
```

6. Mulicycle paths

If a FF clock-enable is used on all flip flops of a multi-cycle path, you can attach TNM to the clock enable net. NOTE: TNM attached to a net "forward traces" to any FF, LATCH, RAM, or PAD attached to the net.

```
NET clock PERIOD = 40ns;
NET multi_cycle_ce TNM = slowffs;
TIMESPEC TS01 = FROM : slowffs : TO : slowffs : 80ns;
```

APPENDIX III
<u>Better Managing Large XL Designs in M1</u>

Here are some suggestions for compilation flows and strategies when compiling large designs in M1 so that you can avoid excessively long runtimes while still realizing the bulk of a design's performance potential.

1. Recommended Compile Strategies in PAR

Some strategies that can be used to more efficiently place and route designs are:

* *Non Timing-Driven Placement and Routing, + Delay-based Cleanup Routing Pass*
* *Non Timing-Driven Placement + "Restricted" Timing Driven Routing*
* *Timing Driven Placement + "Restricted" Timing Driven Routing*

These strategies are ordered to progress from quickest runtime (good for initial design evaluation and non-timing critical designs) towards increasingly better design performance results (for those designs with timespecs that are increasingly tougher to meet.)

<u>Strategy 1:</u> **Non-Timing Driven Placement & Routing + Delay-based Cleanup Pass**

This strategy will typically produce circuit performance that is anywhere from 55% to 75% of that which could be ultimately obtained with a "full-out" timing-driven run, with a runtime that is usually many times faster.

To run PAR in this manner from the command line, use:
par -x -d 1 <*design.ncd*> <*design_r.ncd*>
When running non-timing driven PAR, it is important to also run at least one pass of a cleanup router - preferably the Delay-Based cleanup router (hence the **–d 1** in the above command line). This will work to minimize route delays.

From the Flow Engine GUI, the same thing can be accomplished by de-selecting the "Use <u>T</u>iming Constraints During Place and Route" check-box, and setting "Run **<u>1</u> Delay-based Cleanup Passes" under the Place and Route tab of the Implementation Options dialog box.**

<u>Strategy 2:</u> Non-Timing Driven Placement + "Restricted" Timing Driven Routing

This strategy will typically produce a result that has significantly better circuit performance than a fully non-timing driven run (Strategy 1), but have faster runtime than a fully timing driven run.

To do the non-timing driven placement from use the command line:
par -x -r <*design.ncd*> <*design_r.ncd*>
From the GUI, the same thing can be accomplished by specifying the par -r **option from template customization form in the Design Manager (use <u>U</u>tilities -> <u>T</u>emplate Manager -> Cus<u>t</u>omize to access this form), and de-selecting the "Use <u>T</u>iming Constraints During Place and Route" check-box in the PAR implementation options dialog box.**

The next step would be to run a re-entrant routing phase in timing-driven mode to route the design. Testing has shown that 75% to 85% of a given design's ultimate circuit performance is achieved relatively early in the PAR routing process. Therefore it is recommended that you initially limit the number of router iterations to three **or** four **passes.**

To run the timing-driven re-entrant routing phase with one delay-based cleanup pass, use the command line:

par -k -i 3 -d 1 -w <*design_r.ncd*> <*design_r.ncd*>

From the Flow Engine GUI select the Setup -> Advanced pulldown menus. Select the "Allow Re-entrant Routing" check-box, set "Run **3 Re-entrant Route Passes", set "Run** 1 Delay-based Cleanup Passes", and check the box labeled "Use Timespecs During Re-entrant Place and Route".

Strategy 3: Timing Driven Placement + "Restricted" Routing

By invoking the timing analysis engine during the placement phase, a better result will be produced than would be obtained using timing-driven routing alone. Again, we limit the runtime by explicitly setting number of attempted router iterations to a relatively low count (three or four.)

For timing driven placement, with a limited number of router iterations, the command line usage is :
par -i 3 –d 1 *<design.ncd> <design_r.ncd> <design.pcf>*
From the Flow Engine GUI, the same thing can be accomplished by selecting the "Use Timing Constraints During Place and Route" check-box and setting "Run **3 Router Iterations" under the Place and Route tab of the Implementation Options dialog box.**

As with the earlier strategies, further re-entrant routing passes can be used to obtain incremental gains in performance.

Further PAR Usage Techniques:

The above strategies are geared towards minimizing compilation runtimes, while still achieving the bulk of a design's performance potential. If an incremental gain (e.g. 3% to 10%) in circuit performance is required after initially using one of the above strategies, one following PAR usage techniques can be used.

* **Run one or two more delay-based cleanup passes if running non-timing driven PAR (Strategy 1)**
* **Run more iterations of the re-entrant route process for the timing-driven runs (Strategies 2, 3).**
* **Use higher PAR effort levels**

Reduce Levels of Logic:

Often times it is best to change the source design when significant gains in circuit performance are required. Reducing the numbers of levels of logic between synchronous elements will most effectively increase circuit performance - as well as often resulting in a faster runtimes.

MPPR:

As always, the Multi-Pass Place & Route (MPPR) capability can be used with any of the above three strategies if still more performance is required. Testing has shown 15% to 20% differences in ultimate performance using different placement seeds/cost tables. Typically, running MPPR is a task that's best suited for overnight or weekend runs.

2. Understand Both Design and Device.

Though the XL family is quite good at maintaining circuit performance throughout the entire device range, the large (e.g. 4085XL) devices, because they are bigger and have correspondingly longer routing segments, can exhibit longer delays than would be found in a smaller (e.g. 4020XL) device.

With the large FPGA devices, it is much more important to take advantage of running pre-PAR timing analysis runs

Better Estimation of Routing v. Block Delays

Many designers have traditionally used the a "50/50" measure as a rough rule of thumb to evaluate pre-route timing reports. Here, the designer will allocate roughly 50% of the timing budget to the logic blocks (LUTs, TBUFs, IOBs, etc) and the rule was that the remaining 50% should be sufficient to account for general routing delays. Of course, this is only a rough estimate; *carry chains* have much faster routing, *high-fanout nets* can cause routing delays to become more dominant. In reality, the 50/50 rule is perhaps more realistically a "30/70 - 70/30" *range*.

3. Use Hierarchy in your design

By effectively partitioning your designs, you can significantly reduce compile time and improve synthesis results. This section provides recommendations for partitioning your designs.

Restrict Shared Resources to Same Hierarchy Level

Resources that can be shared should be on the same level of hierarchy. If these resources are not on the same level of hierarchy, the synthesis tool cannot determine if these resources should be shared.

Compile Multiple Instances Together

You may want to compile multiple occurrences of the same instance together to reduce the gate count. However, to increase design speed, do not compile a module in a critical path with other instances.

Restrict Related Combinatorial Logic to Same Hierarchy Level

Keep related combinatorial logic in the same hierarchical level to allow synplify tool to optimize an entire critical path in a single operation. Boolean optimization does not operate across hierarchical boundaries. Therefore, if a critical path is partitioned across boundaries, logic optimization is restricted. Also, constraining modules is difficult if combinatorial logic is not restricted to the same level of hierarchy.

Separate Speed Critical Paths from Non-critical Paths

To achieve satisfactory synthesis results, locate design modules with different functions at different levels of
the hierarchy. Design speed is the first priority of optimization algorithms. To achieve a design that efficiently utilizes device area, remove timing constraints from design modules.

Restrict Combinatorial Logic that Drives a Register to Same Hierarchy Level

To reduce the number of CLBs used, restrict combinatorial logic that drives a register to the same hierarchical block.

Restrict Module Size

Restrict module size to 100 - 200 CLBs. This range varies based on your computer configuration; the time
required to complete each optimization run; if the design is worked on by a design team; and the target FPGA routing resources. Although smaller blocks give you more control, you may not always obtain the most efficient design.

Register All Outputs

Arrange your design hierarchy so that registers drive the module output in each hierarchical block. Registering outputs makes your design easier to constrain because you only need to constrain the clock period and the ClockToSetup of the previous module. If you have multiple combinatorial blocks at different levels of the hierarchy, you must manually calculate the delay for each module. Also, registering the outputs of your design hierarchy can eliminate any possible problems with logic optimization across hierarchical boundaries.

Restrict One Clock to Each Module or to Entire Design

By restricting one clock to each module, you only need to describe the relationship between the clock at the
top level of the design hierarchy and each module clock. By restricting one clock to the entire design, you only need to describe the clock at the top level of the design hierarchy.

4. Using Global Low Skew Buffers on high fanout signals

For designs with global signals, use global clock buffers to take advantage of the low-skew, high-drive capabilities of the dedicated global buffer tree of the target device. Synplify will automatically inserts a BUFG generic clock buffer on the clock signals with the highest fanout. The Xilinx implementation software automatically selects the clock buffer that is appropriate for your specified design architecture. If you want to use a specific global buffer, you must instantiate it.

You can instantiate an architecture-specific buffer if you understand the architecture and want to specify how the resources should be used. Each XC4000E/L and Spartan device contains four primary and four secondary global buffers that share the same routing resources. XC4000EX/XL/XV devices have sixteen global buffers; each buffer has its own routing resources. XC5200 devices have four dedicated global buffers in each corner of the device.

XC4000 EX/XL/XV devices have two different types of global buffer, Global Low-Skew Buffers (BUFGLS) and Global Early Buffers (BUFGE). Global Low-Skew buffers are standard global buffers that should be used for most internal clocking or high fanout signals that must drive a large portion of the device. There are eight BUFGLS buffers available, two in each corner of the device. The Global Early buffers are designed to provide faster clock access, but CLB access is limited to one quadrant of the device. I/O access is also limited. Similarly, there are eight BUFGEs, two in each corner of the device.

Because Global early and Global Low-Skew Buffers share a single pad, a single IPAD can drive a BUFGE, BUFGLS or both in parallel. The parallel configuration is especially useful for clocking the fast capture latches of the device. Since the Global Early and Global Low-Skew Buffers share a common input, they cannot be driven by two unique signals.

You can use the following criteria to help select the appropriate global
buffer for a given design path.

- The simplest option is to use a Global Low-Skew Buffer.
- If you want a faster clock path, use a BUFG. Initially, the software will try to use a Global Low-Skew Buffer. If timing requirements are not met, a BUFGE is automatically used if possible.
- If a single quadrant of the chip is sufficient for the clocked logic, and timing requires a faster clock than the Global Low-Skew buffer, use a Global Early Buffer.

Note: For more information on using the XC4000 EX/XL/XV device family global buffers, refer to the online Xilinx Data Book or the Xilinx web site at http://www.xilinx.com.

For XC4000E/L and Spartan devices, you can use secondary global buffers (BUFGS) to buffer high-fanout, low-skew signals that are sourced from inside the FPGA. To access the secondary global clock buffer for an internal signal, instantiate the BUFGS cell. You can use primary global buffers (BUFGP) to distribute signals applied to the FPGA from an external source. Internal signals can be globally distributed with a
primary global buffer, however, the signals must be driven by an external pin.

XC4000E devices have four primary (BUFGP) and four secondary (BUFGS) global clock buffers that share four global routing lines. These global routing resources are only available for the eight global buffers. The
eight global nets run horizontally across the middle of the device and can be connected to one of the four vertical longlines that distribute signals to the CLBs in a column. Because of this arrangement only four of the eight global signals are available to the CLBs in a column. These routing resources are "free" resources because they are outside of the normal routing channels. Use these resources whenever possible. You may want to use the secondary buffers first because they have more flexible routing capabilities.

You should use the global buffer routing resources primarily for high-fanout clocks that require low skew, however, you can use them to drive certain CLB pins, as shown in the following figure. In addition, you can use these routing resources to drive high-fanout clock enables, clear lines, and the clock pins (K) of CLBs and IOBs.

If your design does not contain four high-fanout clocks, use these routing resources for signals with the next highest fanout. To reduce routing congestion, use the global buffers to route high-fanout signals. These
high-fanout signals include clock enables and reset signals (not global reset signals). Use global buffer routing resources to reduce routing congestion; enable routing of an otherwise unroutable design; and ensure that routing resources are available for critical nets.

Xilinx recommends that you assign up to four secondary global clock buffers to the four signals in your design with the highest fanout (such as clock nets, clock enables, and reset signals). Clock signals that require low skew have priority over low-fanout non-clock signals. You can source the signals with an input buffer or a gate internal to the design. Generate internally sourced clock signals with a register to avoid unwanted glitches.

5. Increasing performance with GSR/GR net

Many designs contain a net that initializes most of the flip-flops in the design. If this signal can initialize all the flip-flops, you can use the GSR/GR net. You should always include a net that initializes your design to a known state.

To ensure that your HDL simulation results at the RTL level match the synthesis results, write your code so that every flip-flop and latch is preset or cleared when the GSR signal is asserted. By default, Synplify can infer the GSR/GR net if there is a single reset net used through out your HDL.

6. Implementing Multiplexers with Tri-state buffers

A 4-to-1 multiplexer is efficiently implemented in a single XC4000 or Spartan family CLB. The six input signals (four inputs, two select lines) use the F, G, and H function generators. Multiplexers that are larger than 4-to-1 exceed the capacity of one CLB. For example, a 16-to-1 multiplexer requires five CLBs and has two logic levels. These additional CLBs increase area and delay. Xilinx recommends that you use internal tristate buffers (BUFTs) to implement large multiplexers.

Large multiplexers built with BUFTs have the following advantages.

- Can vary in width with only minimal impact on area and delay
- Can have as many inputs as there are tristate buffers per horizontal longline in the target device
- Have one-hot encoded selector inputs

7. Comparing If Statement and Case Statement

The If statement generally produces priority-encoded logic and the Case statement generally creates balanced logic. An If statement can contain a set of different expressions while a Case statement is evaluated against a common controlling expression. In general, use the Case statement for complex decoding and use the If statement for speed critical paths.

APPENDIX IV
<u>Design Size and Performance</u>

Your design should meet the following requirements.

- Design must function at the specified speed
- Design must fit in the targeted device

After your design is compiled, you can determine preliminary device utilization and performance with Synplify's reporting options. After your design is mapped, you can determine the actual device utilization. At this point in the design flow, you should verify that your design is large enough to incorporate any future changes or additions. Also, at this point, it is important to verify that your design will perform as specified.

Determining Actual Device Utilization and Pre-routed Performance

To determine if your design fits the specified device, you must map it with the Map program. The generated report file design_name.mrp contains the implemented device utilization information. You can run the Map program from the Design Manager or from the command line.

Improving Implementation Results

Conversely, you can select options that increase the run time, but produce a better design within the Xilinx environment. These options generally produce a faster design at the cost of a longer run time. These options are useful when you run your designs for an extended period of time (overnight or over the weekend).

Multi-Pass Place and Route Option

Use this option to place and route your design with several different cost tables (seeds) to find the best possible placement for your design. This optimal placement results in shorter routing delays and faster designs. This option works well when the router passes are limited. Once an optimal cost table is selected, use the re-entrant routing feature to finish the routing of your design. Select this option from the Design menu in the Design Manager, and specify this option at the command line with the -n switch.

Turns Engine Option

This option is a Unix-only feature that works with the Multi-Pass Place and Route option to allow parallel processing of placement and routing on several Unix machines. The only limitation to how many cost tables are concurrently tested is the number of workstations you have available. To use this option in the Design Manger, specify a node list when selecting the Multi-Pass Place and Route option. To use this feature at the command line, use the -m switch to specify a node list, and the -n switch to specify the number of place and route iterations.

Re-entrant Routing Option

Use the re-entrant routing option to further route an already routed design. The router reroutes some connections to improve the timing or to finish routing unrouted nets. You must specify a placed and routed design (.ncd) file for the implementation tools. This option is best used when router iterations are initially limited, or when your design timing goals are close to being achieved.

Cost-Based Clean-up Option

This option specifies clean-up passes after routing is completed to substitute more appropriate routing options available from the initial routing process. For example, if several local routing resources are used to transverse the chip and a longline is available, the longline is substituted in the clean-up pass. The default value of cost-based cleanup passes is 1. To change the default value, use the Template Manager in the Design Manager, or the -c switch at the command line.

Delay-Based Clean-up Option

This option specifies clean-up passes after routing is completed to substitute more appropriate routing options to reduce delays. The default number of passes for delay-based clean-up is 0. You can change the default in the Design Manager in the Implementation Options window, or at the command line with the -d switch.

Guide Option (not recommended)

This option is generally not recommended for synthesis-based designs. Re-synthesizing modules can cause the signal and instance names in the resulting netlist to be significantly different from those in earlier synthesis runs. This can occur even if the source level code (Verilog or VHDL) contains only a small change. Because the guide process is dependent on the names of signals and comps, synthesis designs often result in a low match rate during the guiding process.   Generally, this option does not improve implementation results.

How to Reach Synplicity:
Synplicity, Inc.
624 East Evelyn Ave.
Sunnyvale, CA 94086   USA
Phone: 408 617-6000
Fax: 408 617-6001
Internet: http://www.synplicity.com
Support email: support@synplicity.com
Sales email: sales@synplicity.com
To give comments
on the documentation: info@synplicity.com

How to Reach Xilinx:
Xilinx
2100 Logic Drive
San Jose, CA 95124-3450
Phone: 408 559-7778
Fax: 408 879-4780
Internet: http://www.xilinx.com
Support email: hotline@xilinx.com
Sales email: FPGA@xilinx.com