Chip-Level HDL Simulation Using the Xilinx Alliance Series

XAPP 108 May 21, 1998 (Version 1.0) Application Note

Summary

This application note describes the basic flow and some of the issues to be aware of for HDL simulation with Alliance Series software. The goal of this document is to familiarize the user with some of the concepts but should not be considered a replacement for the Xilinx or HDL simulator's documentation. Please refer to the *Xilinx HDL Design Guide* and other appropriate Xilinx documents for more detailed information on HDL simulation and synthesis. Also refer to the vendor documentation for specific information about your particular simulator or synthesis tool.

Introduction

HDL simulation is becoming increasingly popular in the design community due to increasing design size and complexity, as well as recent improvements in design synthesis and simulation tools. The two leading synthesis and simulation languages today are Verilog and VHDL. Both of these languages have been adopted standards by IEEE and both are becoming increasingly popular in the logic design community. The Xilinx Alliance Series software currently supports the Verilog IEEE 1364 Standard, VHDL IEEE Standard 1076.4 for Vital (Vital 95), and SDF version 2.1.

The new Xilinx Alliance Series was designed to be used with several HDL synthesis and simulation tools to provide a solution for programmable logic designs from beginning to end. The Xilinx Alliance Series provides libraries, netlist readers and netlist writers along with the powerful place and route software that integrates with your HDL design environment on PC and UNIX workstation platforms.

Overview of HDL Simulation Flow

A typical design flow includes these basic steps. (Figure 1)

1. HDL Code Entry
2. Behavioral or RTL (Pre-synthesis) Simulation
3. Synthesis of the design
4. Post-Synthesis simulation
5. Implementation (Place and Route) of the design
6. Timing simulation
7. Download of the design to the FPGA for in-circuit verification

It may take several iterations of some of these steps to get the desired result.

HDL Code Entry

Typically, the first step in beginning an HDL design is entering the HDL code, once the design specifications are finalized. During this design entry state there are many things to keep in mind in order to create an efficient design.

Coding Style

Generally it is suggested to pick a specific coding style for the design and adhere to it. This aids in reading and debugging the code. Some things to consider are the following:

- Capitalization style
- Indentation style
- Use of spaces in the code
- Code commenting, use, and style

Keep in mind that there are differences between sythesizable HDL code and simulatable HDL code. Generally the synthesizable code set is a sub-set of the simulatable HDL code. In other words, you can generally create HDL code that simulates properly but may or may not synthesize. The capabilities of the synthesis tool compared to the simulator are dependent on the particular tools that you are using. Review the documentation for your tools in order to determine the coding capabilities of each.

Behavioral Code

A general suggestion in the initial design creation is to keep your code behavioral. Avoid instantiating specific components unless necessary. This allows for more readable code, faster and simpler simulation, code portability (the ability to migrate to different device families) and code reuse (the ability to use the same code in future designs).

Instantiation of Components

At times it becomes necessary to describe components structurally in order to obtain the desired design structure rather than specifying behavioral code. In order to accomplish this, a method called instantiation is used to describe and connect these components to the existing design. A list of several of the most frequently instantiated components may be found in the appendix of the *Xilinx Alliance Series Quick Start Guide*. It is suggested to always use capital letters for all component names being instantiated as well as all component pin names. This is required for instantiated cells in behavioral simulation with Verilog Unisim library and may be also required by synthesis tools. Cadence schematics create lowercase names and must use the

Cadence interface libraries and not the uppercase UNISIM libraries. See Table 4 for an example of instantiating a Xilinx output register, OFD.

LogiBLOX

LogiBLOX modules may also be created and instantiated to simplify the design process. The LogiBLOX module generation can create an instantiation template that may be used to assist in the integration of the module into the design code. A simulation model may also be created to be used with behavioral simulation but should not be used in the synthesis of the design.

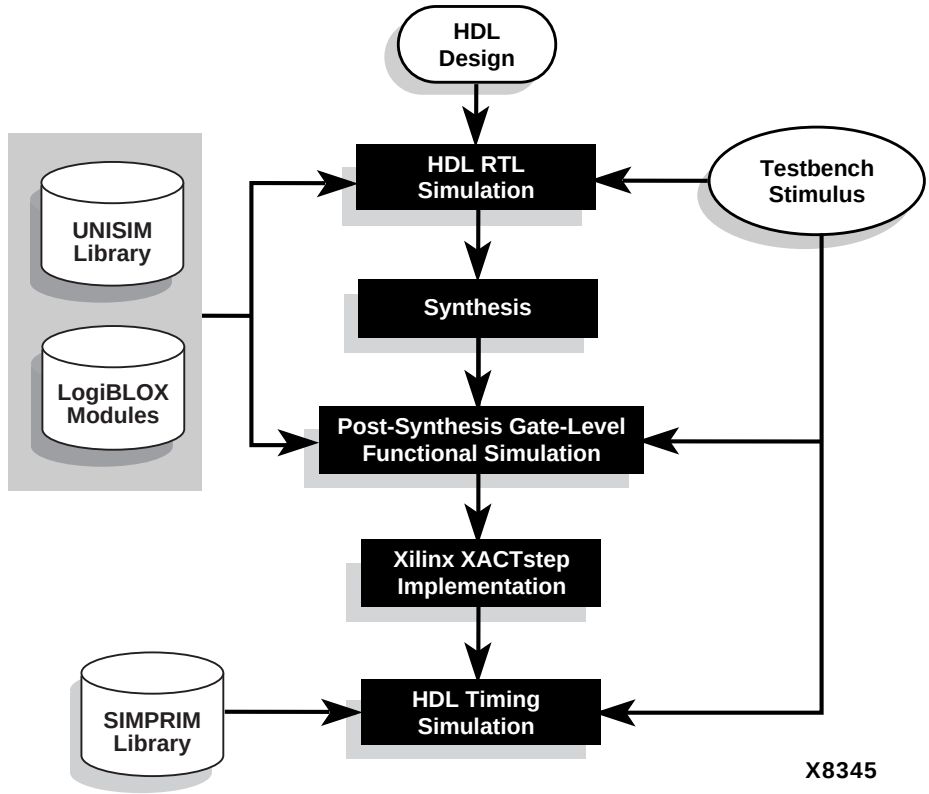

Figure 1: Typical Design Flow for Programmable Logic Device

RTL (Pre-synthesis) Simulation

RTL simulation is usually the second step in implementing a design after initially entering the code. This first pass simulation is typically performed to verify code syntax and to confirm that the functionality of the code is what is intended. At this step no timing information is provided and simulation should be performed in a unit-delay mode to help prevent the possibility of a race condition.

Testbench

Before simulation is performed, a testbench or test fixture is usually created to apply the stimulus to the design. A testbench is HDL code written for the simulator that instantiates the design netlist(s), initializes the design and then applies stimuli to verify the functionality of the design. The testbench can also be setup to display the desired simulation output to a file, waveform or screen. The testbench has many advantages over interactive simulation methods by allowing repeatable simulation throughout the design process and also provides documentation of the test conditions. There are several methods to create a testbench and simulate a design. A testbench may have a very simple structure that sequentially applies stimulus to specific inputs, or it may be very complex, including subroutine calls, stimulus read in from external files, conditional stimulus or other more complex structures. An example for Verilog and VHDL may be found in the $XILINX/synopsys/tutorial directory.

Within the Verilog test fixture, it is recommended that you also add a compiler directive ‘timescale to define the simulation timing unit and degree of precision. The syntax is: **‘timescale** <**time_unit**> / <**time_precision**>**.** For example:

‘timescale 1ns / 100 ps.

The above example specifies that the basic delay unit of the simulation is 1 nanosecond, but transitions may occur at intervals of 100 picosecond. This matches the resolution required by Xilinx Verilog libraries. There are no default timescale values in Verilog so it is always recommended that you specify the simulation units in the test fixture.

The test fixture generated by NGD2VER in later simulations instantiates the design in the test fixture and also declares the ‘timescale unit for you.

Clocking the Design

Almost every synchronous design requires a clock or several clocks to clock to the registers. A stimulus is typically provided in the testbench that mimics the oscillation of an external clock. If the FPGA internal oscillator is to be used, the simulation model may provide this stimulus but in some cases it may be desirable to manually apply the stimulus through a testbench to decrease simulation time and allow you to control the behavior directly. There are several ways to manufacture a clock in a test bench but Table 4 illustrates fairly efficient ways to create the oscillator.

UNISIM Library

If a Xilinx component is instantiated into the HDL code, a behavioral model must be available for this component before you can perform a RTL simulation of the design. The UNISIM library was created to address this need. To perform a behavioral simulation of the design containing instantiated components, you must point the simulator to the appropriate UNISIM library. For some simulators, the library may also need to be compiled. Compilation of the libraries is generally required for the VHDL Vital libraries however the Verilog libraries may or may not need to be compiled depending on the simulator being used. Consult your simulator documentation for more information on compiling simulation libraries. See Table 5 for more information about the UNISIM libraries. The source files for the Xilinx unified simulation libraries are located in the following directories:

Vital VHDL:

$XILINX/vhdl/src/unisims
The following 7 source files are provided:
unisim_VCOMP.vhd (component declaration file)
unisim_VCOMP52K.vhd (substitutional component declaration file for XC5200 designs)
unisim_VPKG.vhd (package file)
unisim_VITAL.vhd (model file)
unisim_VITAL52K.vhd (additional model file for XC5200 designs)
unisim_VCFG4K.vhd (configuration file for XC4000 edge decoders)
unisim_VCFG52K.vhd (configuration file for XC5200 internal decoders)

Verilog:

Each of the following directories contains individual files, one for each instantiable component.
$XILINX/verilog/src/UNI3000 (XC3000 family)
$XILINX/verilog/src/UNI4000E (XC4000E, XC4000L and SPARTAN families)
$XILINX/verilog/src/UNI4000X (XC4000EX, XC4000XL and XC4000XV families)
$XILINX/verilog/src/UNI5200 (XC5200 family)

In the above directories, $XILINX is the install directory for the Xilinx software.

For Verilog users, there are typically two common ways to point the simulator to the location of these libraries. You may either add the ‘uselib compiler directive pointing to the location of the UNISIM models on the system or you can specify the library location from command line switches. Consult your simulator documentation for details on setting up compiler libraries.

Many VHDL simulators specify library locations from a setup file or setting. Each simulator may have a different method to specify simulation library location so please consult the simulator documentation.

VHDL Global Set/Reset and Tri-State

If a global set/reset is desired for behavioral simulation, it must be included in the behavioral code. Any described register in the code must have a common signal that will asynchronously set or reset the register depending on the desired result. Similarly, if a global tri-state is desired for simulation, it should be described in the code as well. The UNISIM library also includes a few new models to assist in Vital VHDL simulation of the global set/reset and tri-state signals. Depending on the desired circuit, one or a combination of the following UNISIM models should be connected to the described global set/reset and/or tri-state signals in the design.

ROC (Reset On Configuration)

The ROC (Reset On Configuration) cell is a VHDL simulation-only component used to generate a single high pulse for simulating the global clearing or setting of registers that occurs in an implemented design during device configuration. This cell may be used in the simulations throughout the design process. Figure 2 and Figure 3 depict models for ROC cell simulation and implementation. The ROC cell is reinserted when the VHDL netlist for timing is written out.

The ROC cell must be instantiated into the code to connect to the design. The following is an example of such an instantiation:

<instance_name>: ROC port map (O =>GSR_NET);

The width of the ROC initialization pulse is generally passed in the design HDL testbench. The proper value for the reset pulse width may be found in the data book for the particular device you are targeting by finding the T_{POR} (Power On Reset) specification. See Figure 2 for an example configuration statement specifying a width of 100 ms.

ROCBUF (Reset On Configuration Buffer)

The ROCBUF (Reset On Configuration Buffer) cell is used in similar VHDL cases to the ROC cell but instead of providing a component to drive the net, it allows the user's testbench to drive the net without actually implementing it on chip. Therefore the user can issue the reset initialization pulse and subsequent reset pulses during the simulation via the simulation reset port if multiple configuration cycles are being simulated, however, the simulation reset port will not appear as an input pin in the implemented design.

As with the ROC cell, the ROCBUF cell must be instantiated into the code, however, this added port must now be driven from the testbench file. If this port is desired for the timing simulation as well, use the –gp switch for NGD2VHDL to recreate the simulation reset port.

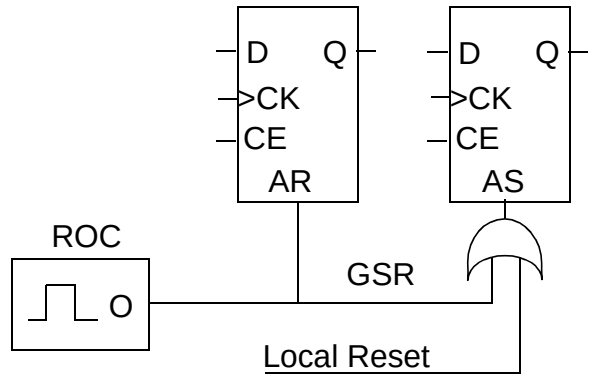

Figure 2: Model of ROC for Functional Simulation

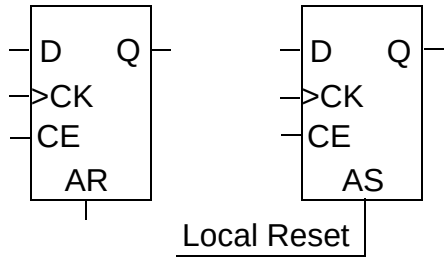

Figure 3: Model of ROC During Implementation After Logic Trimming

```
CONFIGURATION RTL_Simulation OF my_testbench_entity_name IS
        FOR my_testbench_architecture_name
                FOR design_instance_name:my_design
                        FOR design_architecture_name
                                FOR ALL:roc USE ENTITY unisim.roc(roc.v)
                                        Generic MAP (width => 100 ms)
                                END FOR;
                        END FOR;
                END FOR;
        END FOR;
END RTL_simulation;
```

Figure 2: Sample ROC Configuration Statement for RTL Simulation

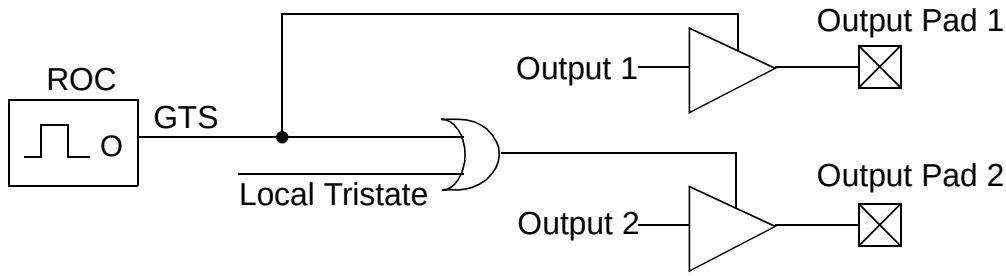

Figure 4: Model of TOC for Simulation

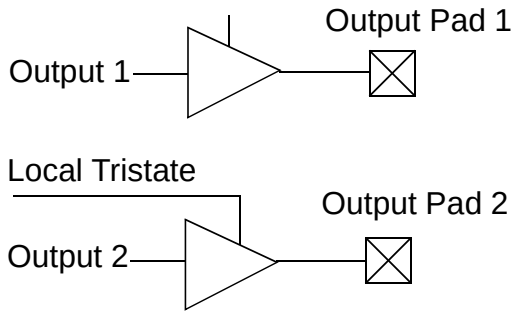

Figure 5: Model of TOC After Logic Trimming

TOC (Tri-state On Configuration)

The TOC (Tri-state On Configuration) cell is used in VHDL to generate a single high pulse for simulating the global tri-stating of the I/Os that occurs in an implemented design during device configuration. Figure 4 and Figure 5 depict the simulation and implementation models for the TOC cell.

The TOC cell is reinserted when the VHDL timing netlist is written. As with the ROC cell, the TOC cell must be instantiated into the code. The following is an example of this instantiation:

<instance_name>: TOC port map (O => GTS_NET);

To set the width of the TOC pulse, a configuration statement similar to that of the ROC must be specified in the simulation testbench. Typically the duration of the tri-state is similar to that of power on reset so the same value, T_{POR}, can be used for TOC cell. These times may change depending on the device, configuration method and options chosen. See Figure 6 for an example configuration statement for a TOC set for 100 ms.

TOCBUF (Tri-state On Configuration Buffer)

The TOCBUF (Tri-state On Configuration Buffer) cell allows VHDL test benches to access to the net connected to the tri-state pins of a design's I/Os via a user-defined input port referred to as the simulation tri-state port. Therefore the user can not only issue the reset initialization pulse, emulating the tri-stating of I/Os in the design during configuration, but can also issue subsequent initialization pulses via the simulation tri-state port if multiple configuration cycles are being simulated. Note, however, that the simulation tri-state port will not appear as an input pin in the implemented design.

As with the TOC cell, the TOCBUF cell must be instantiated into the code however this tri-state simulation port must now be driven by the testbench file. If this port is desired for

```
CONFIGURATION RTL_simulation OF my_testbench_entity_name IS
        FOR my_testbench_architecture_name
                FOR design_instance_name:my_design
                        FOR design_architecture_name
                                FOR ALL:toc USE ENTITY unisum.toc(toc.v)
                                        Generic MAP (width => 100 ms)
                                END FOR;
                        END FOR;
                END FOR;
        END FOR;
END RTL_Simulation;
```

Figure 6: SampleTOC Configuration Statement RTL Simulation

the timing simulation as well, use the –tp switch for NGD2VHDL to create the simulation tri-state port.

STARTBUF (Startup Buffer)

The STARTBUF (Startup Buffer) cell allows VHDL designs access to all of the input and output ports of the STARTUP cell, and adds 2 extra ports (GSROUT and GTSOUT) for simulation purposes. The port names for the STARTBUF cell differ slightly from that of the STARTUP cell. See Table 1 for details.

The input ports GSRIN and GTSIN can be connected either directly or indirectly via combinational logic to input ports of the design. The design input ports will appear as input pins in the implemented design. Note that use of the STARTBUF implies that the design has user-defined signals driving the global set/reset and/or tri-state resource(s) available in the implemented design. This is in addition to the automatic pulse that occurs during configuration.

Figure 7 and Figure 9 show how the STARTBUF is connected for Functional and Timing simulation while Figure 8 depicts the implemented circuit.

The STARTBUF is not reinserted, the GSR/GTS are directly wired to the ports.

To summarize, if it is desired to have either the Global Set/Reset (GSR) or Global Tri-state (GTS) signal as a port in the implemented design, it is suggested to use the STARTBUF cell for simulation and implementation. If it is desired to have access to the Global Set/Reset signal during simulation but not implementation, it is suggested to use the ROCBUF cell. Similarly, if it is desired to have access to the Global Tri-state signal during simulation but not implementation, use the TOCBUF cell. If it is not important to have access to these global signals but if the user still wishes to simulate the startup characteristics of the device, use the ROC and TOC cells.

Table 1: STARTBUF Pin Descriptions.

STARTBUF Pin Name	Connection Point	XC4000 STARTUP Pin Name	XC5200 STARTUP Pin Name
GSRIN	Global Set/Reset Port of design	GSR	GR
GTSIN	Global Tri-state Port of design	GTS	GTS
GSROUT	All registers asynchronous set/reset	Not available, for simulation only	Not available, for simulation only
GTSOUT	All output buffers tri-state control	Not available, for simulation only	Not available, for simulation only
CLKIN	Port or internal logic	CLK	CLK
Q2OUT	Port or internal logic	Q2	Q2
Q3OUT	Port or internal logic	Q3	Q3
Q1Q4OUT	Port or internal logic	Q1Q4	Q1Q4
DONEINOUT	Port or internal logic	DONEIN	DONEIN

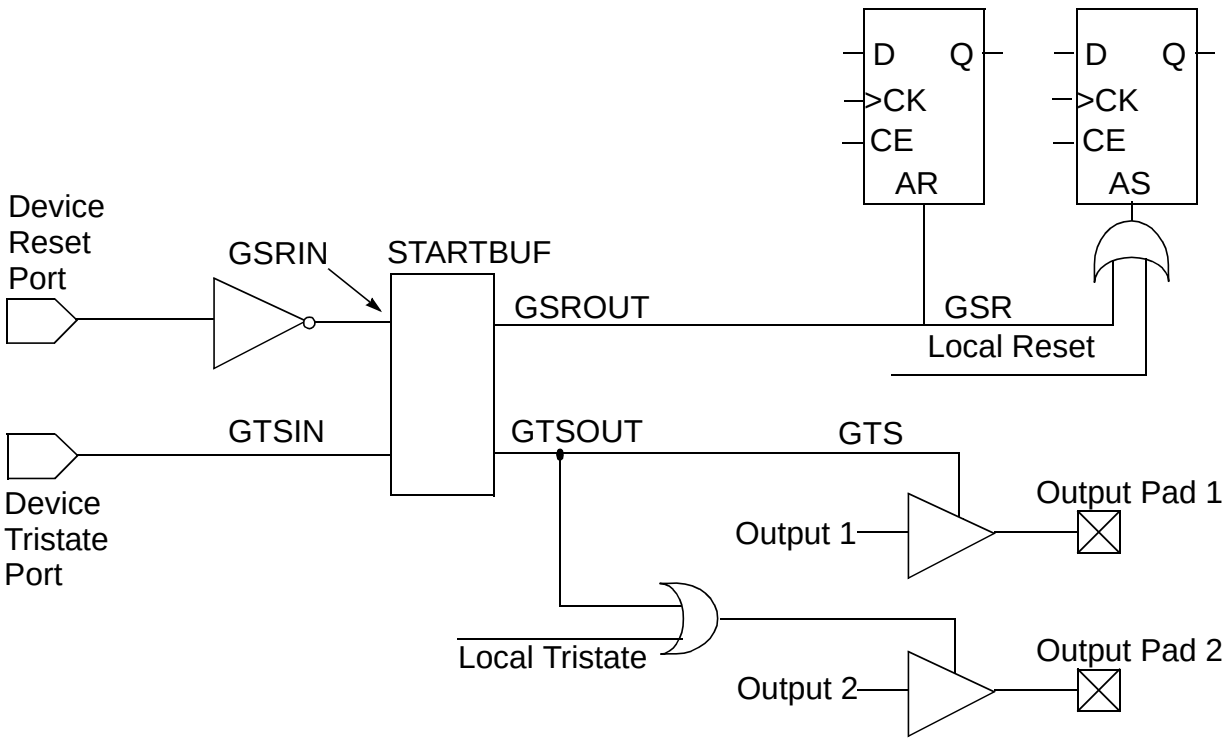

Figure 7: Model of STARTBUF Cell for Behavioral and Post-synthesis Simulation

Note: GSR has been configured as active low by the addition of an inverter to the STARTBUF

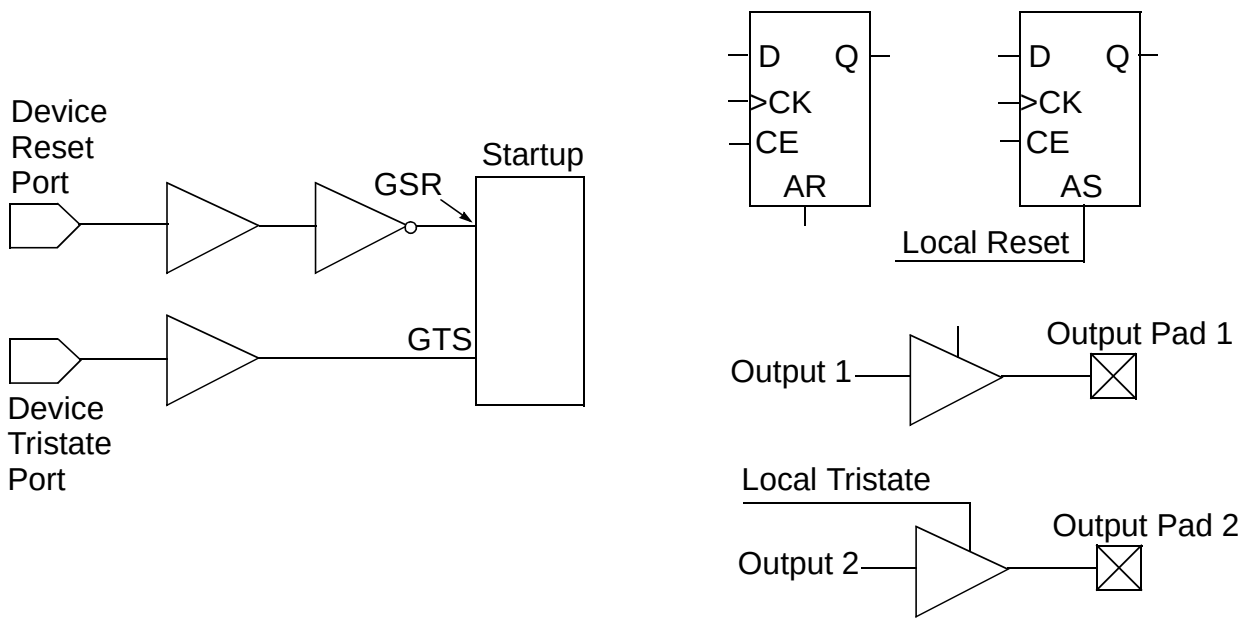

Figure 8: Diagram of Implementation Circuit After Logic Trimming

Note: GSR has been configured as active low by the addition of an inverter to the STARTBUF.

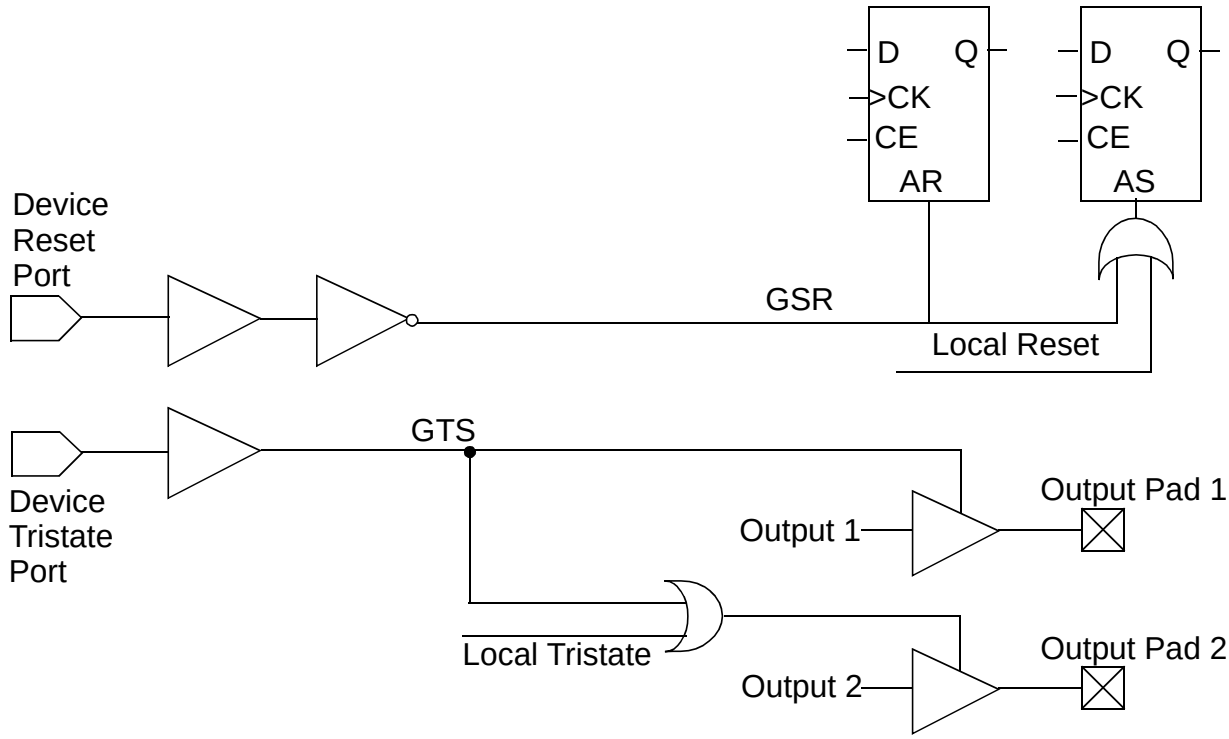

Figure 9: Model of Design Using STARTBUF for Timing Simulation

Note: GSR has been configured as active low by the addition of an inverted to the STARTBUF.

Verilog Global Set/Reset and Tri-State

For Verilog implementation, all behaviorally described (inferred) and instantiated registers should have a common signal which asynchronously sets or resets the register however the Verilog UNISIM library does not have ROC, ROCBUF, TOC, TOCBUF, or STARTBUF cells. The simulation of their function is rather simple from the testbench. Unlike VHDL VITAL 95, a global signal may be defined in the testbench to connect sub-module signals such as GSR or GTS.

For Verilog simulation without an instantiated STARTUP block, the following should be added to the design code and test fixture.

- Within the design Verilog code, GSR and/or GTS should be declared as Verilog wire or register within the design module.
- Then, within the test fixture file set two macros called:
  - GSR_SIGNAL
    to testfixture_module. design_instance.GSR (the name of the global set/reset signal, qualified by the name of the test fixture module) and design instance name instantiated in the test fixture.

  and the design instance name given in the test fixture file and
  - GTS_SIGNAL
    to testfixture_module design_instance.GTS (The name of the global tri-state signal, qualified by the name of the test fixture module and the design instance name given in the test fixture file.)

using the `define compiler directive. GSR and GTS should then be toggled High, then Low in an “initial” block. The duration of this reset and tri-state pulse should be obtained from the Xilinx Data book by acquiring the TPOR (Power-On Reset) specification for the device being used. This example illustrates a 100 ns Power-On Reset width (assuming a `timescale of 1 ns). This example illustrates an implementation targeting an XC4000 family device. See Figure 10.

```
module testfixture_name;

‘define GSR_SIGNAL testfixture_module.design_instance.GSR;
‘define GTS_SIGNAL testfixture_module.design_instance.GTS;

initial
 begin
      ‘GSR_SIGNAL = 1;       // reset the device
       ‘GTS_SIGNAL = 1;     // tri-state all outputs
       #100 ‘GSR_SIGNAL = 0;
      ‘GTS_SIGNAL = 0; // device now active
    <add simulation data here>
```

Figure 10: Verilog Global Set/Reset Test Fixture Implementation Targeting XC4000 or Spartan Device.

GSR and GTS are the signal names used for this example and active high is the polarity of these signals. See Table 2 for a listing of other Xilinx device's global signal names and polarities.

Asserting global set/reset and global tri-state when the STARTUP block is specified in the design is similar to asserting global set/reset and global tri-state without a STARTUP block in the design. There are two differences, however. The first difference is that the ‘define statement must now specify the name of the net attached to the GSR (for XC4000, see Table 2 for other devices) and/or GTS pin on the STARTUP block. The other difference is that the signals you toggle are now the external input ports that control the “global_set_reset_port” and/or “global_tri-state_port” on the STARTUP block. (Figure 11) After implementing the design in Alliance Series (post-NGDBUILD, post-MAP, or timing simulation), simply invoke the simulation using the same test fixture. The test fixture and simulation information may remain unchanged.

Table 2: Global Reset and Tri-state Names for Xilinx Devices

Device Family	Global Reset Name	Global Tri-state Name	Default Reset Polarity
XC3000	GR	Not Available	Low
XC4000	GSR	GTS	High
XC5000	GR	GTS	High
XC9500	PRLD	GTS	High
SPARTAN	GSR	GTS	High

```
module testfixture_name;
‘define GSR_SIGNAL port_connected_to_GSR_pin_of_STARTUP;
‘define GTS_SIGNAL port_connected_to_GTS_pin_of_STARTUP;
initial
 begin
    global_set_reset_port = 1;      // reset the device
    global_tri-state_port = 1;     // tri-state all outputs
    #100 global_set_reset_port = 0;
    global_tri-state_port = 0; // device now active
    <add simulation data here>
```

Figure 11: Asserting Global Set/reset and Global Tri-state for RTL Simulations

Note:The “define declaration can be REMed out for Post_NGDBuild, Post-MAP and Timing simulation.

LogiBLOX

As mentioned before, LogiBLOX can create a simulation model in order to allow behavioral simulation of the LogiBLOX created module. This model is not necessary for post-NGDBUILD, post-MAP or timing simulation. The process for this simulation is different for Verilog and VHDL users.

For Verilog, a structural netlist is created using a technology independent library called SIMPRIMs. In order to simulate this netlist, the simulator needs to be pointed to the location of the SIMPRIM models. The default location of these models are located in $XILINX/verilog/data although this location may be different on a particular system. The extensions of these libraries are .vmd. If the LogiBLOX module contains registers, these registers would need to be initialized by performing a global reset.

For VHDL designs, a behavioral netilst is created but a VITAL library needs to be compiled before simulating the LogiBLOX netlist. The location of the source code for the LogiBLOX library is $XILINX/vhdl/src/logiblox. Check with the simulator vendor’s documentation for details on how to compile this library

Synthesizing the Design

The method to synthesize a design is solely dependent on the synthesis tools being used. Familiarize yourself with the various synthesis options available to you. These options may have a dramatic effect on the resulting compiled design. Please refer to the *Xilinx Synthesis and Simulation Design Guide*, the synthesis tool’s documentation for instructions and hints on properly synthesizing the design to be targeted to a Xilinx device.

Post-Synthesis Simulation

Post synthesis simulation is used to verify the design functionality after the design has been synthesized and a structural representation of your design has been created. This may be done in any of three steps, after synthesis, after design netlist translation (NGDBUILD) or after Mapping the design.

After Synthesis

If the synthesis tool has the capability of writing out a post-synthesis HDL netlist of the design, this may be used to simulate the design and evaluate the synthesis results. This structural netlist will contain the synthesized gates created by the synthesis tool. Using the same UNISIM library used during behavioral simulation, a simulation may be performed on this structural netlist.

The same testbench used for the behavioral simulation may be used with this post-synthesis simulation, however the deign needs to be initialized by performing a global reset if this was not performed in the previous simulation. The methods for executing a global reset are different for Verilog and VHDL. See the previous Global Set/Reset and Tri-state section for details on performing a global reset for the design.

After Design Translation (Post NGDBUILD)

Simulation may also be performed after the netlist translation stage, NGDBUILD. Generally an HDL simulation may be performed here if the synthesis tool is not capable of writing out a structural simulation netlist. The .ngd file produced from NGDBUILD may be input into one of the simulation netlisters, NGD2VER or NGD2VHDL. NGD2VER and NGD2VHDL create a structural simulation netlist based on the SIMPRIM models. SIMPRIMs are “generic” technology independent models in which you perform sim-

ulation from the Alliance series software. Depending on the simulator being used, these SIMPRIM models may need to be compiled by the simulator before use.

In order to create the Verilog or VHDL simulation netlist, as shown in Figure 12, perform the following steps on the synthesis compiler produced EDIF or XNF netlist, referenced here by <*design*>.

```
ngdbuild <design>
ngd2ver  -tf  –ul <design>.ngd ngdbuild_sim.v
or
ngd2vhdl –tb <design>.ngd ngdbuild_sim.vhd
```

Figure 12: Creating a Verilog or VHDL Simulation Netlist

Note: The –ul switch for NGD2VER will automatically add the `uselib directive specifying the location of the SIMPRIM libraries. If your simulator does not accept this directive, omit this switch.

NGD2VER and NGD2VHDL may create a template testbench or test fixture which may simplify the simulation of the design. You may simply add your stimulus from the behavioral simulation or create new simulation stimulus to test the design for the desired outputs. The –tf and –tb switches will create the test fixture or testbench template. The Verilog test fixture file has a .tv extension and the VHDL test bench file has a .tvhd extension.

As with the previous simulations, the simulator needs to be pointed to the location of the simulation primitives, in this case SIMPRIM models. The location of the compiled libraries may be different on each system. See Table 5 for more information about the SIMPRIM libraries.

For Verilog users, the –ul switch will add a `uselib directive in the Verilog simulation netlist which points to the location of the SIMPRIM models on the system. Alternatively, this library location may be specified from command line switches.

For VHDL users, this is usually done in the setup file located in the simulation directory.

For more specific information, please consult the vendor documentation for details on specifying design libraries.

After Map

Simulation may also be performed after the maping of the design but before the place and route stage. This simulation will include the block delays for the design but not the routing delays. This is generally a good barometer to test whether the design is meeting the timing requirements before spending the time to fully place and route the design.

As with the previous simulation, NGD2VER or NGD2VHDL will create the structural simulation netlist based on SIMPRIM models. The following steps must be followed in order to create a post-MAP simulation model. (Figure 13)

A template file for the test fixture may be generated as mentioned in the previous simulation section.

The delays for the design are stored in an SDF, standard delay format, file which is created by the simulation netlister, NGD2VER or NGD2VHDL. This SDF file will contain all block or logic delays however will not contain any of the routing delays for the design since the design has not yet been placed and routed. All block delay values are worst case values. Actual device block delays are generally shorter under normal operating conditions.

For Verilog, the SDF file is automatically read when the simulator compiles the Verilog simulation netlist. Within the simulation netlist is the Verilog function $sdf_annotate which specifies the name of the SDF file to be read in.

The user specifies the SDF file for most VHDL simultors. The method for doing so is different depending on the simulator being used. Typically a command line or GUI switch is used to read in the SDF file.

The post-MAP simulation may also be back annotated to the original input netlist (EDIF or XNF file) to the Alliance Series software. This means that many of the internal signal and instance names to the design may be replaced in the simulation netlist to possibly aid in debugging internal nodes of the design. When executing NGDANNO, including the map.ngm file produced from the Map stage creates the back annotated simulation file. Since many of the internal signal and instance names of the design were most likely created by the synthesis tool and contain names that have no meaning to the designer, this step is not usually necessary and disabling this feature will decrease the run time of the Alliance Series software.

```
ngdbuild <design>
map <design>.ngd map.ncd
ngdanno map.ncd
ngd2ver –tf –ul map.nga map_sim.v
or
ngd2vhdl –tb map.ngd map_sim.vhd
```

Figure 13: Creating a Post-MAP Simulation Model

Placement and Routing of the Design

It is not until this stage of design implementation that we get the full picture as to how the design will behave in circuit. The overall functionality of the design has been defined from the beginning stages but it is not until the design has been placed and routed that all of the timing information of the design can be accurately calculated.

Timing Simulation

The timing simulation is the last step of chip-level verification before downloading the bitstream to the device. A timing simulation netlist may be created after the design has completed placement and routing in the Alliance Series tools. As with the previous simulations, NGD2VER or NGD2VHDL creates a structural netlist, however this netlist comes from the placed and routed .ncd file.

```
ngdbuild <design>
map <design>.ngd map.ncd
par <options> map.ncd <design>.ncd map.pcf
ngdanno <design>.ncd
ngd2ver –tf –ul <design>.nga time_sim.v
or
ngd2vhdl –tb <design>.ngd time_sim.vhd
```

Figure 14: Creating a Timing Simulation Model

SDF

For this timing simulation, an SDF file is created as with the after MAP simulation however this SDF file contains all block and routing delays for the design. As with the post-map version, all delays are worst case values.

Back Annotation

As with the post-map simulation, the Timing simulation can also be back annotated to the original design netlist. When implementing the design from Design Manager, this option is default. It is represented by the “Correlate Simulation Data to Input Design” radio button in the Simulation Data Options section of the Implementation Options window. Since many of the internal signal and instance names of the design were probably created by the synthesis tool and contain names that have no meaning to the designer, this step is not usually necessary and disabling this feature will decrease the run time of the Alliance Series software.

During timing simulation, there are usually two occurrences to look for. First, ensure that the design is functioning as expected after placement and routing. Second, look for timing violations. Timing violations from the simulation are represented by a setup or hold time error message issued by the simulator.

Setup and Hold Times

The simulator will issue a setup or hold time violation any time data changes at a register input (data or clock enable) within the setup or hold time window for the particular register. These are a few typical causes of a setup or hold time violation:

1. Data path delays are too long for clocking speeds.
2. Clock skew is unaccounted for before simulation.
3. Data paths cross out-of-phase or asynchronous clocks (to each other).
4. Input data changing at the wrong time.

Some questions for debugging setup or hold time issues are:

1. Did Trace or Timing Analyzer report the data path may run at speeds being clocked in simulation?
2. Is clock skew being accounted for in this path delay? Was the clock path analyzed by Trace or Timing Analyzer? Does subtracting the clock path delay from the data path delay still allow clocking speeds?
3. Will slowing down the clock speeds eliminate the setup/hold time violations?
4. Does this data path cross clock boundaries (from one clock frequency to another)? Are the clocks synchronous to each other? Is there appreciable clock skew between these clocks?
5. Is this path an input path to the device? Does changing the time at which the input stimulus is applied eliminate the setup/hold time violations?

Hopefully the answers to these questions help determine the cause of the setup or hold time violation. Based on these responses, possible design changes may need to be made in order to accommodate the simulation conditions.

While Xilinx data sheets state that there are zero hold-times on the internal registers and I/O registers with the default delay, it is still possible to receive a hold-time violation from the simulator. This hold-time violation is in reality a setup violation on the register but in order to get an accurate representation of the CLB delays, part of the setup time must be modeled as a hold time. For more information on this modeling please refer to http://www.xilinx.com/techdocs/782.htm.

Setup and hold violations may also occur on the write cycle of a RAM. Special care must be taken when writing to RAMs, especially asynchronous RAMs. It is suggested to use synchronous RAMs in your design whenever possible to simplify the write timing for a synchronous design.

Boundary Scan and Readback

At this time the Boundary Scan and Readback circuitry can not be simulated. Xilinx is looking into the creation of models for these components, however are not currently available.

Decreasing Simulation Times

As designs get larger and more complex, simulation times may also increase. Larger parts may take a long time to compile and simulate depending on utilization and machine in which the simulation is run. There are some techniques that may speed up simulation. For Verilog simulation, the ‘timescale resolution may be decreased in order to decrease simulation time with the trade-off of less precision.

Xilinx development system’s RAM, swap space and disk space requirements increase as design size increases. If these requirements are not met, simulation times may increase substantially or simulation may not be possible at all. Most modern HDL simulators require faster CPU computation speeds in order to manage shorter simulation times. Older CPU architectures generally will be significantly slower simulating a design.

Many simulators have other methods to speed up simulation times. Please consult the vendor simulation manual for addition suggestions for speeding up design simulation.

Downloading to the Device

Once the design is verified to behave as desired from a timing simulation, the design may be downloaded to the device to test the design in circuit. Under normal operating conditions, any synchronous path should function exactly as it did in the timing simulation. Asynchronous paths are typically slightly faster in the actual device since the delays are modeled for worst case conditions.

The Xilinx Alliance Series tools also includes a powerful tool for in circuit debugging of the design called Hardware Debugger. Hardware Debugger, when used with the Xchecker cable, allows the viewing of logic levels and the creation of waveforms of internal nodes within the FPGA. When this information is used along with the information received from the timing simulation, actual device behavior may be compared with that found from the simulation models. Please refer to the Xilinx Dynatext on-line manual *Hardware Debugger Reference/User Guide* for more detailed information on in-circuit debugging of a design.

Conclusion

As Xilinx devices continue to grow in density and complexity, it becomes increasingly important to create simple yet powerful design simulation and implementation tools. Xilinx recognizes this need and will continue to focus on the HDL design entry and verification as an important part of the design process and will continue its integration into this design methodology.

Please visit the Xilinx World Wide Web site for the latest information about Xilinx devices and software at http://www.xilinx.com. Other Xilinx related documents can be found in Table 3.

Table 3: Available Xilinx Documents

Synthesis and Simulation Design Guide
Cadence Interface/Tutorial Guide
Mentor Graphics Interface/Tutorial Guide
Synopsys (XSI) Interface/Tutorial Guide
Synopsys (XSI) Synthesis and Simulation Design Guide
Xilinx Development System User Guide

Table 4: Examples of common coding examples for Verilog and VHDL testbench creation. Most of these examples are not valid for synthesis however are useful for simulation testbench creation.

Description	Verilog	VHDL
Delay or wait 20 nS	‘**timescale** 1 ns/ 100 ps #20;	**wait for** 20 ns;

Creation of a free running clock	Initial
	begin
	clock = 0;
	#25 **forever** #25 clock = ~clock;
	end
		Loop
		wait for 25 ns;
		clock <= not (clock);
		end loop;

Print “Text.” to screen	**$display**(“Text.”);	**report** “Text.”
Print value of signal to screen whenever the value changes	**$monitor**(“%t”, $realtime, “%b”, clock,, “%b”, my_signal);	
Apply a binary value 1010 to an input bus X.	X = 4’b1010;	X <= “1010”;
Creation of a for loop 0 to 10.	**for**(x=0; x < 10; x=x+1) begin <*actions*> end	**for** x in 0 to 9 **loop** <*actions*> end loop;
Write “X = <*value*>” to an output file	**$dumpfile** (“filename.dmp”); **$dumpvars** (X);	**variable** TEMP; **write** (TEMP, “X = “); **write** (TEMP, X); **writeline** (filename, TEMP);
Wait until X is logic one.	**wait** (X == 1’b1);	**wait until** X = ‘1’;
Wait until X transitions to a logic one.	@(**posedge** X);	**wait on** X;

If-Else construct	**always** @ (X)
	begin
	if (X = 1)
	Y = 1’b0;
	else
	Y = 1’b1;
	end
		process (X)
		if (X = ‘1’) **then**
		Y = ‘0’;
		else
		Y = ‘1’;
		end if;
		end process;

Case construct	**always** @(X or A) **case** (X) 2’b00 : Y = 1’b0; 2’b01 : Y = 1’b1; default : Y = A; **endcase**	**process** (X, A) begin **case** X **is** **when** “00” => Y = ‘0’; **when** “01” => Y = ‘1’; **when others** => Y = A; end case; end process;
Example instantiation of an OFD	OFD U1 (.Q(D_OUT), .D(D_IN), .C(CLOCK));	U1: OFD **port map** (Q => D_OUT, D => D_IN, C => CLOCK);

Table 5: Xilinx Alliance Series HDL Simulation Library Information

Library	Simulation	Location of Source Code		Compilation of Libraries Required	
		Verilog	VITAL VHDL	Verilog	VITAL VHDL
UNISIM (XC4000 Family)	Behavioral and Post-Synthesis	$XILINX/ verilog/src/ UNI4000E *or* UNI4000X	$XILINX/ vhdl/src/ unisims	Not required for Verilog-XL. See vendor documentation for other simulators	Yes. Typical Compilation order: VCFG4K.vhd is optional: unisim_VCOMP.vhd unisim_VPKG.vhd unisim_VITAL.vhd *unisim_VCFG4K.vhd*
UNISIM (XC5200 Family)	Behavioral and Post-Synthesis	$XILINX/ verilog/src/ UNI5200	$XILINX/ vhdl/src/ unisims	Not required for Verilog-XL. See vendor documentation for other simulators	Yes. Typical compilation order: unisim_VCOMP52K.vhd unisim_VPKG.vhd unisim_VITAL.vh unisim_VITAL52K.vhd unisim_VCFG52K.vhd
LOGIBLOX (Device Independent)	Behavioral and Post-Synthesis	None	$XILINX/ vhdl/src/ logiblox	None	Yes. Typical compilation order: mvlutil.vhd mvlarith.vhd logiblox.vhd
SIMPRIM (Device Independent)	Post-NGDBUILD Post-MAP Timing	(for A1.4) $XILINX/ verilog/data (for A1.5) $XILINX/verilog/spc/simprims	$XILINX/ vhdl/src/ simprims	Not required for Verilog-XL. See vendor documentation for other simulators	Yes. Typical compilation order: simprim_Vcomponents.vhd simprim_Vpackage.vhd simprim_VITAL.vhd