XAPP410 (v1.0) June 11, 2001

Simulating a Xilinx 3.1i CORE Generator Verilog Design

Summary

This application note begins with an overview of the steps necessary to include a Xilinx CORE Generator macro in a Verilog design. Next, the **Input/Output Files**, page 2 describes the Xilinx CORE Generator™ 3.1i files used in a Verilog HDL design project. The remaining sections describe the steps to simulate a Xilinx 3.li CORE Generator Verilog design, as follows:

- Compiling the CORE Generator libraries — how to create simulator libraries, map the created libraries, and compile the XilinxCoreLib CORE Generator libraries to user-created libraries.
- Instantiating a Core Generator Macro for Simulation — how to cut and paste VEO template information into Verilog HDL files to instantiate the macro.
- Running a Functional Simulation — how to compile and load a design into different simulators for functional simulation
- Running a Timing Simulation — how to compile and load a design into different simulators for functional simulation

To download and install IP Updates, go to section **Design Files Disclaimer and Download Instructions**, page 15 for instructions.

Overview

The basic steps necessary to include a Xilinx CORE Generator macro in a Verilog design are:

1. Run the CORE Generator to create the macro
2. Use the VEO template for the module declaration to black box the CORE Generator module (see **Instantiating a Core Generator Macro for Simulation**, page 6)
3. Synthesize the design
4. Run the Xilinx implementation tools

Additional steps to perform simulations are:

1. If the simulator uses precompiled libraries, compile the Xilinx CORE Generator Libraries (see **Compiling the CORE Generator Libraries**, page 3)
2. Use the VEO template to map the user-defined parameters for the macro to the simulation model (see **Instantiating a Core Generator Macro for Simulation**, page 6)
3. Run the functional or timing simulation (see Running a Functional Simulation and Running a Timing Simulation, below)

Input/Output Files

This section describes project and CORE Generator input and output files.

Project Input Files

	Coefficients	Parameter file
Verilog	.COE	.XCO
VHDL	.COE	.XCO

Project Input Files Description

.COE ASCII data file. It defines the coefficient values for FIR filters and initialization values for memory modules (see $XILINX/coregen/data for sample COE files).

.XCO CORE Generator system file placed in the current project directory. It contains the parameters used to generate a core, and can be referenced as a log file of the settings used to generate a core (for more information, see *XCO Files* in the Xilinx CORE Generator System User Guide 3.1i).

Project Output Files

	Implementation	Template	Memory Init	CoreGen Log	Coefficients	Project File
Verilog	.EDN	.VEO	.MIF	.LOG	.COE	.PRJ
VHDL	.EDN	.VHO	.MIF	.LOG	.COE	.PRJ

Project Output Files Description

.EDN EDIF implementation netlist for the core. It is an input file to the Xilinx Implementation Tools (NGDBuild) and specifies how the CORE will be implemented.

Note: The EDN file is used for implementation only. The Xilinx implementation tools read the EDN and other EDIF design files during the translate step (NGDBuild).

.VEO Verilog template file. The components in this file can be used as a guide to create the core's Verilog instantiation and pass parameters to a Verilog behavioral model (for further information, see "Using the CORE Generator Verilog Design Flow Procedure," in the CORE Generator System User Guide 3.1i).

.VHO VHDL template file. The components in this file can be used to instantiate a core and pass parameters to the VHDL behavioral model through a configuration declaration (for further information, see "Using the CORE Generator VHDL Design Flow Procedure," in the CORE Generator System User Guide 3.1i).

Note: The VEO and VHO template files are used to instantiate cores into HDL files. They are not used directly for implementation or simulation.

.MIF Memory Initialization File generated by the CORE Generator System. It specifies initialization values for block RAM modules during an HDL functional simulation. To produce an MIF file, direct the CORE Generator to produce an EDIF implementation netlist and either a Verilog or VDHL behavioral simulation output.

Note: The MIF file is generated by the CORE Generator from the COE file. It is used for simulation only. For distributed memory, the MIF file is generated with initial default values if the user does not specify values.

.PRJ The coregen.prj file contains a record of installed cores with current version information. It is produced when a project is created if all project options have been selected. To allow users to load the new project from the CORE Generator GUI, a link to the new project is written to the known.prj file, located in the $XLINX/coregen/preferences directory.

System Input/Output Files Description

Except for the get_models.log file, the default location for the files listed below is the $XILINX/verilog/src/XilinxCoreLib directory. The default location for the get_models.log file is the $XILINX/verilog/src directory.

get_models.log

Log file that contains user-visible messages displayed during a get_models run. The log file is written to the directory specified in the get_models command line (get_models –dest *path_to_directory).* If a destination directory is not specified, get_models extracts files to the $XILINX/verilog/src/XilinxCoreLib directory.

verilog_analyze_order

Contains a list of CORE Generator Verilog behavioral models in suggested compilation order for Verilog simulators (such as Synopsys VCS, Cadence NC-Verilog and MTI).

vhdl_analyze_order

Contains a list of CORE Generator VHDL behavioral models in compilation order. More than one compilation order may be valid for the library.

XilinxCoreLib/*.v

Verilog behavioral models extracted from the IP installed in the CORE Generator tree.

XilinxCoreLib/*.vhd

VHDL behavioral models extracted from the IP installed in the CORE Generator tree.

XilinxCoreLib/*_comp.vhd

VHDL component declaration files for each CORE Generator IP module extracted from the CORE Generator.

Compiling the CORE Generator Libraries

This section describes the steps necessary to create and map a library, then compile the XilinxCoreLib CORE Generator libraries to the created library.

Note: Uncompiled XilinxCoreLib Verilog libraries are located in the $XILINX/verilog/src/XilinxCoreLib directory. To extract library files to a destination directory, run get_models with the *-dest* parameter (see the get_models.log file description, above), then compile the extracted models.

MTI-Verilog

The information, below, on compiling the CORE Generator libraries applies to ModelSim PE, EE, and SE. ModelSim XE uses precompiled libraries, which can be downloaded from **http://support.xilinx.com/support/mxelibs.htm**. ModelSim XE users who download an IP update also must download the precompiled CORE Generator IP Update libraries.

For information on compiling Xilinx CORE Generator libraries and on how to use an MTI script to compile the libraries, go to **http://support.xilinx.com/techdocs/8066.htm**. This solution contains a link to T_{CL} scripts that can be run from MTI to compile the CORE Generator libraries.

Note: For additional information on compiling the Xilinx libraries (Unisim, Simprim, and others), go to **http://support.xilinx.com/techdocs/2561.htm**

To create, map, and compile the libraries, as follows:

1. Create the xilinxcorelib_ver library, as follows:

```
vlib xilinxcorelib_ver
```

2. Map the library to update the modelsim.ini file, which allows ModelSim to identify the library, as follows:

```
vmap xilinxcorelib_ver complete/path/to/xilinxcorelib_ver
```

Note: If the MODELSIM environment variable is used to point to the modelsim.ini file, write permission access to the modelsim.ini file must be enabled. If write permission access is not enabled, copy the modelsim.ini file to the local project directory, unset the MODELSIM variable, then run the vmap command, as shown above.

3. Compile the XilinxCoreLib libraries

   The verilog_analyze_order file, located in the $XILINX/verilog/src/XilinxCoreLib directory, contains a list of the Verilog behavioral model filenames in compilation order. Each file in the compilation list can be compiled separately by invoking the vlog command for each file in the list, as follows:

   ```
   vlog –work xilinxcorelib_ver
   \$XILINX/verilog/src/XilinxCoreLib/filename.v
   ```

To avoid performing repetitive manual compilations, do the following:

1. Copy the verilog_analyze_order file from the $XILINX/verilog/src/XilinxCoreLib directory to the local directory.
2. Insert the vlog command shown above at the beginning of each Verilog filename listed in the local verilog_analyze_order file.
3. Save the modified file as verilog_anaIyze_order.do in the local directory.
4. Run the verilog_anaIyze_order.do file from the MTI command line or select Macro -> Execute Macro from the ModelSim pulldown menu, then select the verilog_anaIyze_order.do file.

NC-Verilog

NC-Verilog uses shared precompiled libraries. Compile the libraries, as follows:

1. Create a library definitions file (if necessary—see the Note, below), named cds.lib.This file lists the names and locations of accessible libraries and includes statements that map logical library names to physical directory paths.

   Note: Cadence provides the nclaunch utility program to setup the necessary initialization files and compile the Verilog source libraries. The nclaunch program is included as part of Xilinx 3.1i and later releases. If nclaunch is unavailable, the cds.lib file must be created manually to map physical locations to logical names.

   The cds.lib can be created with any text editor, as follows:

   a. Use the UNIX mkdir command, as follows:

      ```
      mkdir -p <compile_dir>/xilinxcorelib_ver
      ```

   b. Use INCLUDE or SOFTINCLUDE to specify the location of the master cds.lib file (this makes logical library names available to all designs). For example:

      ```
      INCLUDE $CDS_INST_DIR/share/local/xilinx/cds.lib


      Edit $CDS_INST_DIR/share/local/xilinx/cds.lib to include


      DEFINE xilinxcorelib_ver <compile_dir>/xilinxcorelib_ver
      ```

   The following is a sample cds.lib file:

   ```
   DEFINE simprims /ncv/libs/verilog/simprims

   DEFINE uni3000 /ncv/libs/verilog/uni3000

   DEFINE uni5200 /ncv/libs/verilog/uni5200

   DEFINE uni9000 /ncv/libs/verilog/uni9000
   ```

```
DEFINE unisims /ncv/libs/verilog/unisims
DEFINE xilinxcorelib_ver /ncv/libs/verilog/xilinxcorelib_ver
```

2. Create a configuration variables file named hdl.var. This file defines variables that configure the user environment. The variables LIB_MAP, VIEW_MAP and WORK specify the search order of the libraries and views when the elaborator resolves instances.
   a. To make variable settings available to all designs, use INCLUDE or SOFTINCLUDE to specify the location of the master hdl.var file, for example:

   ```
   INCLUDE $CDS_INST_DIR/share/local/xilinx/hdl.var

   Edit $CDS_INST_DIR/share/local/xilinx/hdl.var

   SOFTINCLUDE $CDS_INST_DIR/tools/inca/files/hdl.var

   DEFINE LIB_MAP ( $LIB_MAP, \
            <compile_dir>/xilinxcorelib_ver => xilinxcorelib_ver)

   DEFINE VIEW_MAP ( $VIEW_MAP, .v => v)
   ```

   b. Edit the hdl.var file to list the search order of the simulation libraries according to the family to be simulated:

      The following is a sample hdl.var file:

   ```
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/simprims =>
   simprims)
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/uni3000 => uni3000)
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/uni5200 => uni5200)
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/uni9000 => uni9000)
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/simprims =>
   simprims)
   DEFINE LIB_MAP ($LIB_MAP, /ncv/libs/verilog/xilinxcorelib_ver=> \
   xilinxcorelib_ver)
   DEFINE VIEW_MAP ($VIEW_MAP, .v => v)
   ```

3. Parse and analyze the XilinxCoreLib simulation libraries using ncvlog (the libraries can be compiled in the order specified in the verilog_analyze_order file), as follows:

   ```
   ncvlog -messages -work xilinxcorelib_ver \
   $XILINX/verilog/src/XilinxCoreLib/*.v
   ```

For additional information on compiling the Xilinx libraries (Unisim, Simprim, and others), go to http://support.xilinx.com/techdocs/2554.htm

Note: Scripts for UNIX OS users are available in the Xilinx install that compile the Cadence's Affirma NC-Verilog libraries. They are located at $XILINX/bin/<platform>/compile_hdl.pl.

Note: Due to the presence of INCLUDE statements in libraries, it may be necessary to run the "ncvlog" command from the directory that contains the "XilinxCoreLib" uncompiled libraries.

VCS (5.0.1a and later)

VCS uses shared precompiled libraries. Compile the libraries, as follows:

```
vcs -Mdir=<compile_dir>/xilinxcorelib \
$XILINX/verilog/src/XilinxCoreLib/*.v
```

The -Mdir=<compile_dir> option specifies the pathname of the central directory where VCS creates the generated files. This option allows team members in a large design to share compiled files for new or improved modules. See http://support.xilinx.com/techdocs/6330.htm for additional information on compiling the Xilinx libraries (Unisim, Simprim, and others).

Note: VCS can be run in one of two ways: 1) by using library source files with compile-time options (similar to Verilog-XL) or 2) by using shared precompiled libraries as described above (see the appropriate sections, below, to run functional and timing simulations).

Verilog-XL

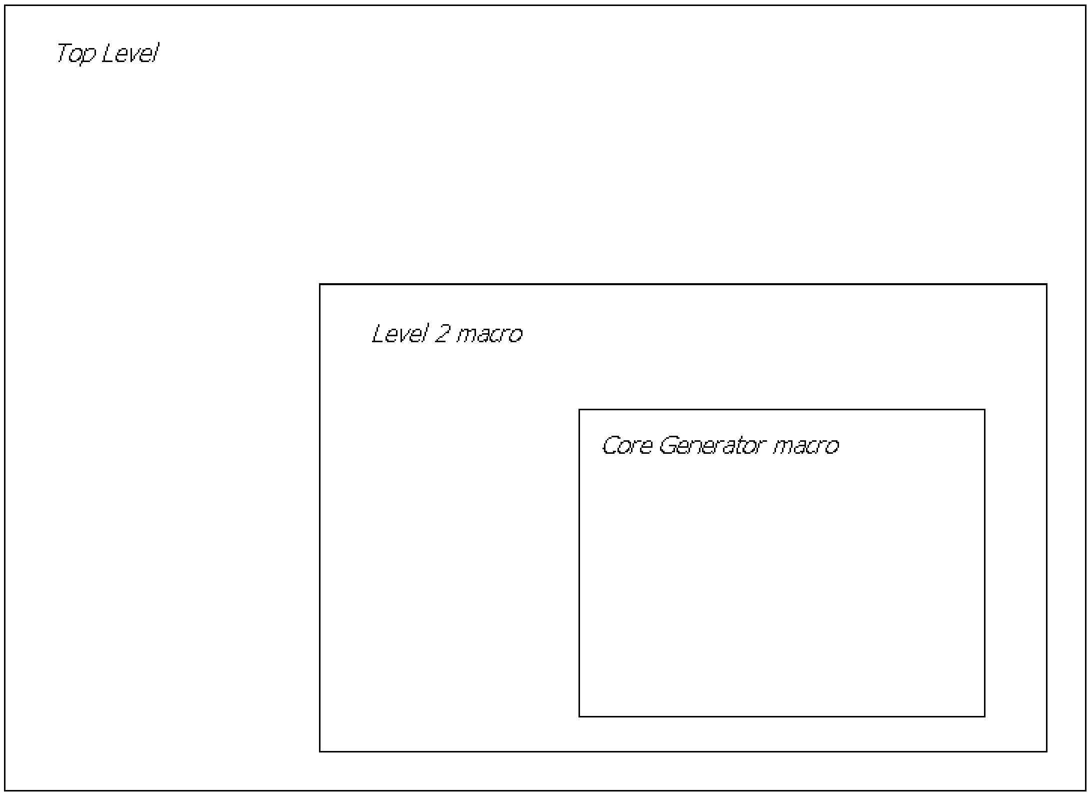
Verilog-XL uses the library source files with a compile-time option, so pre-compilation is not necessary. Please refer to **Running a Functional Simulation**, page 9 and **Running a Timing Simulation**, page 12 sections for information on running Verilog-XL simulations.

Instantiating a Core Generator Macro for Simulation

This section describes how cut and paste VEO template information into HDL files for simulation. More specifically, it shows how parameters are passed for Verilog simulations.

This design example uses a CORE Generator macro two levels down in the hierarchy to illustrate how parameters are passed to macros for functional simulation. The RAMB4_S16_S16 component is a 4096-bit dual-ported dedicated random access memory block with synchronous write capability. Each port independently accesses the same set of 4,096 memory cells. Since this example generates a 512-bit memory image, it does exercise the entire depth of the block RAM.

The top level instantiates a macro at the second level. The second level instantiates the CORE Generator macro.



In the test-fixture (tf_bram32x16.v), stimulus generation and response checking is performed entirely in code. Verifying results behaviorally in this manner is an extremely efficient simulation method. If the design does not function properly, text output error messages are displayed to alert the designer

VEO Template

This section explains how library inclusion, module declaration, and module instantiation information can be cut and pasted from the Verilog VEO template file.

top.v (Top level that Instantiates the Second Level)

The example in this section only instantiates the second level in the hierarchy to create a bidirectional bus. Therefore, it is not necessary to cut and paste information into the top.v file (see the complete design example files for top.v file information).

Bram32x16.v (Second Level in the hierarchy that contains a CORE Generator Macro)

A black-box methodology is used to instantiate the CORE Generator module (modules instantiated as black boxes are not elaborated or optimized). The following section from the VEO template file can be used to instantiate the CORE Generator macro in a Verilog HDL file. The instance name should be specified, as indicated, and the port connections changed to the appropriate signal names.

```
// Insert the marked text, below, into a Verilog file to instantiate
// this core.
// Change the instance name and port connections
// (in parentheses) to the appropriate signal names.

//----------- Begin Cut here for INSTANTIATION Template ---//
INST_TAG
core32x16 YourInstanceName (
   .ADDRA(ADDRA),
   .CLKA(CLKA),
   .ADDRB(ADDRB),
   .CLKB(CLKB),
   .DIA(DIA),
   .WEA(WEA),
   .DIB(DIB),
   .WEB(WEB),
   .ENA(ENA),
   .ENB(ENB),
   .RSTA(RSTA),
   .RSTB(RSTB),
   .DOA(DOA),
   .DOB(DOB));

// INST_TAG_END ------ End INSTANTIATION Template ---------
```

Core32x16.v (CORE Generator macro file—for the core only)

Library Inclusion

Some simulators allow libraries to be specified when a design is loaded. If this is not possible, the xilinxcorelib library can be specified with a Verilog directive in the HDL code that instantiates the CORE Generator macro. A section of the VEO template file can be used for this purpose, as shown in the following example:

```
// A Verilog compiler/interpreter may require the following
// option or it’s equivalent to help locate the Xilinx Core Library:
// +incdir+${XILINX}/verilog/src
// In this example, ${XILINX} refers to the XILINX software
// installation directory.
```

```
//----------- Begin Cut here for LIBRARY inclusion --------// LIB_TAG

// synopsys translate_off

‘include "XilinxCoreLib/C_MEM_DP_BLOCK_V1_0.v"

// synopsys translate_on

// LIB_TAG_END ------- End LIBRARY inclusion --------------
```

Module Declaration

The following CORE Generator module declaration section of the VEO template file can be cut and pasted into a Verilog HDL file. This section passes parameters to the simulation model for a particular implementation of a design.

Note: It may be necessary to change the path to the MIF file if a COE file is used to initialize the contents of memory.

```
// The following code must appear after the module in which it
// is to be instantiated. Make sure that the translate_off/_on
// compiler directives are correct for the synthesis tool(s).

//----------- Begin Cut here for MODULE Declaration -------// MOD_TAG
module core32x16 (ADDRA,CLKA, ADDRB, CLKB, DIA, WEA,
DIB, WEB, ENA, ENB, RSTA, RSTB, DOA, DOB);

input [4 : 0] ADDRA;
input CLKA;
input [4 : 0] ADDRB;
input CLKB;
input [15 : 0] DIA;
input WEA;
input [15 : 0] DIB;
input WEB;
input ENA;
input ENB;
input RSTA;
input RSTB;
output [15 : 0] DOA;
output [15 : 0] DOB;

// synopsys translate_off

   C_MEM_DP_BLOCK_V1_0 #(
     5,5,1,1,"0",32,32,1,1,1,1,1,1,1,1,1,1,1,1,
     "/complete/path/to/core32x16.mif",
     16,0,1,1,1,1,1,16,16)
   inst (
     .ADDRA(ADDRA),.CLKA(CLKA), .ADDRB(ADDRB),
     .CLKB(CLKB), .DIA(DIA), .WEA(WEA), .DIB(DIB),
     .WEB(WEB), .ENA(ENA), .ENB(ENB), .RSTA(RSTA),
     .RSTB(RSTB), .DOA(DOA), .DOB(DOB));

// synopsys translate_on

endmodule

// MOD_TAG_END ------- End MODULE Declaration ------------
```

Testfixture.v (Testfixture)

For the complete set of testfixture design example files, go to **Design Files Disclaimer and Download Instructions**, page 15. This testfixture example does not fully simulate the CORE Generator macro.

Running a Functional Simulation

This section describes how to compile and load a design into different simulators for functional simulation.

MTI-Verilog

1. If not already created, create a library to compile, as follows:

   ```
   vlib library_name
   ```

2. Compile the Verilog files, as follows:

   ```
   vlog $XILINX/verilog/src/glbl.v file1.v file2.v file3.v
   ```

3. Load the design in the testfixture module and glbl, specify the unisim_ver library for any instantiated Xilinx components, and specify the library xilinxcorelib, as follows:

   ```
   vsim –L library_name top_level_module glbl
   ```

4. After the GUI starts, add waves and signals, then simulate the design.

Application Note Example:

```
vlib work
vlog $XILINX/verilog/src/glbl.v tf_bram32x16.v top_bram32x16.v
\bram32x16.v core32x16.v
vsim -L unisims_ver -L xilinxcorelib_ver test glbl
add wave *
run -all
```

NC-Verilog

NC-Verilog simulations can be run in one of two ways:

1. Use library source files with compile-time options (similar to Verilog-XL):

   For RTL simulation, according to the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or Coregen components), use the following command lines:

   ```
   ncxlmode -y $XILINX/verilog/src/unisims -y \
   \$XILINX/verilog/src/simprims \

      +incdir+$XILINX/verilog/src +libext+.v \
   $XILINX/verilog/src/glbl.v \

    <testfixture>.v <design>.v
   ```

   The $XILINX/verilog/src/unisims directory contains the Unified components for RTL simulation and generic simulation primitives for any gate level or timing simulation.

   Application Note Examples:

   Using library source files with compile-time options:

   ```
   ncxlmode -y $XILINX/verilog/src/unisims –y \
   $XILINX/verilog/src/XilinxCoreLib +incdir+$XILINX/verilog/src \
   +libext+.v tf_bram32x16.v top_bram32x16.v bram32x16.v core32x16.v
   ```

2. Use shared precompiled libraries (see **Compiling the CORE Generator Libraries**, page 3 for instructions to compile the libraries for NC-Verilog—Unisim, Simprim, and others). Create a work directory where the design will be compiled, as follows:

```
Mkdir WORK
```

a. Map the WORK directory in the cds.lib and hdl.var files, as follows:

Add the following to the cds.lib file:

```
DEFINE WORK ./WORK
```

Add the following to the hdl.var file on the line before (above) the DEFINE VIEW_MAP line:

```
DEFINE LIB_MAP ($LIB_MAP, ./WORK => WORK)
```

b. Compile and simulate the design, as follows:

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v \
<testfixture>.v <design>.v
 ncelab -messages testfixture_name glbl
ncsim -messages testfixture_name
```

The -update option of ncvlog enables incremental compilation.

Application Note Examples:

Using shared precompiled libraries:

```
ncvlog -messages -WORK WORK tf_bram32x16.v top_bram32x16.v
bram32x16.v core32x16.v
ncelab -messages -timescale 1ns/1ps test glbl
ncsim -messages test
```

Note: Due to the presence of INCLUDE statements in libraries, it may be necessary to run the "ncvlog" command from the directory that contains the "XilinxCoreLib" uncompiled libraries.

Note: Due to the presence of the INCLUDE statement in the library files, it may be necessary to copy glbl.v to the current directory. As an alternative, use the –update $XILINX/verilog/src/glbl.v option with the ncvlog command.

Note: Use the –timescale option with ncelab if any HDL file does not contain the timescale directive.

VCS

VCS simulations can be run in one of two ways:

1. Use library source files with compile-time options (similar to Verilog-XL):

   For RTL simulation, according to the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or Coregen components), use the following command lines:

```
vcs -y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/simprims \
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \
-Mupdate -R <testfixture>.v <design>.v
```

The $XILINX/verilog/src/unisims directory contains the Unified components for RTL simulations and generic simulation primitives for gate-level and/or timing simulations.

The -R option automatically simulates the executable after compilation

The -Mupdate option enables incremental compilation. Modules are recompiled for the following reasons:

a. The target of a hierarchical reference has changed

b. A compile-time constant (for example, a parameter) has changed.

c. Ports instantiated in the module have changed.

d. Module inlining. For faster simulation, groups of module definitions are merged into one module definition in the VCS file. The affected modules are recompiled. Recompilation only occurs once.

Application Note Examples:

Using library source files with compile-time options:

```
vcs -y $XILINX/verilog/src/unisims -y $XILINX/verilog/src/XilinxCoreLib \
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v -Mupdate -R \
tf_bram32x16.v top_bram32x16.v bram32x16.v core32x16.v
```

2. Use shared precompiled libraries:

   Before using VCS, compile simulation libraries to <compiled_lib_dir> go to (**http://support.xilinx.com/techdocs/ 6330.htm**) for information on how to compile Xilinx Verilog libraries for VCS (Unisim, Simprim, and others).

   For RTL simulation, according to the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or Coregen components), use the following command lines:

```
vcs -Mupdate -Mlib=<compiled_dir>/unisims_ver -y $XILINX/verilog/src/unisims \
-Mlib=<compiled_dir>/simprims_ver -y $XILINX/verilog/src/simprims \
-Mlib=<compiled_dir>/xilinxcorelib_ver - +incdir+$XILINX/verilog/src \
+libext+.v $XILINX/verilog/src/glbl.v -R <testfixture>.v <design>.v
```

The unisims_ver directory contains the Unified components for RTL simulations. The simprims_ver directory contains generic simulation primitives for any gate-level or timing simulations. The XilinxCoreLib area directory contains the Coregen components for RTL simulations.

The -R option automatically simulates the executable after compilation

The -Mlib=<compiled_lib_dir> option provides VCS with a central location for descriptor information prior to compilation and a location for object files when it links the executable.

The -Mupdate option enables incremental compilation. Modules are recompiled for the following reasons:

a. The target of a hierarchical reference has changed

b. A compile-time constant (for example, a parameter) has changed

c. Ports instantiated in the module have changed

d. Module in-lining. For faster simulation, groups of module definitions are merged into one module definition in the VCS file. The affected modules are recompiled. Recompilation only occurs once.

Application Note Examples:

Using shared precompiled libraries:

```
vcs -Mupdate -Mlib=/vcs/lib/unisims_ver -y $XILINX/verilog/src/unisims \
-Mlib=/vcs/lib/xilinxcorelib_ver –y $XILINX/verilog/src/XilinxCoreLib  \
+incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v  \
-R tf_bram32x16.v top_bram32x16.v bram32x16.v core32x16.v
```

Verilog-XL

For RTL simulation, according to the makeup of the design (LogiBLOX, Xilinx instantiated primitives, or Coregen components), use the following command lines:

```
verilog -y $XILINX/verilog/src/unisims –y \
$XILINX/verilog/src/simprims \
 +incdir+$XILINX/verilog/src +libext+.v $XILINX/verilog/src/glbl.v \
 <testfixture>.v <design>.v
```

The $XILINX/verilog/src/unisims directory contains the Unified components for RTL simulations. The $XILINX/verilog/src/simprims directory contains generic simulation primitives for gate-level and/or timing simulations.

Application Note Examples:

```
verilog -y $XILINX/verilog/src/unisims –y $XILINX/verilog/src/XilinxCoreLib \
+incdir+$XILINX/verilog/src +libext+.v tf_bram32x16.v top_bram32x16.v bram32x16.v \
core32x16.v
```

Functional Simulation Hints, Tips, and Common Problems

The following are hints and cautions that apply to running functional simulations:

- Path to the MIF file — When using the VEO template file, be sure to modify the path to the MIF file must be modified to initialize components.
- INCLUDE path — If the INCLUDE statement from the VEO template is used, it may be necessary to change the "/"and the path to which it points.
- glbl.v errors — If glbl.v errors occur, copy the glbl.v to the project directory or, if applicable, set the appropriate simulator runtime option to point to the $XILINX/verilog/src/glbl.v file.

Running a Timing Simulation

This section describes how to load the backend timing simulations into different simulators (to compile Xilinx Simprim libraries required for compiled simulators, see the links to solutions contained in the appropriate simulator section in **Compiling the CORE Generator Libraries**, page 3).

MTI-Verilog

1. If not already created, create a library to compile, as follows:

   ```
   vlib library_name
   ```

2. Compile the Verilog files, as follows:

   ```
   vlog $XILINX/verilog/src/glbl.v file1.v file2.v file3.v
   ```

3. Load the design in the testfixture module and glbl, and specify the simprim_ver library, as the back annotated netlist contains all Xilinx primitive components, as follows

   ```
   Vsim –L simprims_ver top_level_module glbl
   ```

4. After the GUI starts, add waves and signals, then simulate the design

Application Note Example:

```
vlib work
vlog $XILINX/verilog/src/glbl.v tf_bram32x16.v time_sim.v
vsim -L simprims_ver test glbl
add wave *
run -all
```

NC-Verilog

NC-Verilog simulations can be run in one of two ways:

1. Use library source files with compile-time options:

   For timing simulations or post-Ngd2ver, the Simprim-based libraries are used. At the command line, specify the following:

```
ncxlmode -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
+libext+.v <testfixture>.v <design>.v
```

   Application Note Examples:

   Using library source files with compile-time options:

```
ncxlmode -y $XILINX/verilog/src/simprims +libext+.v $XILINX/verilog/src/glbl.v \
tf_bram32x16.v time_sim.v
```

2. Use shared precompiled libraries:

   For timing simulations or post-Ngd2ver, the Simprim-based libraries are used. At the command line, specify the following:

```
ncvlog -messages -update $XILINX/verilog/src/glbl.v <testfixture>.v time_sim.v
ncelab -messages -autosdf testfixture_name glbl
ncsim -messages testfixture_name
```

   See Xilinx Solution 947 (**http://support.xilinx.com/techdocs/947.htm**) for instructions to back annotate the SDF file for timing simulations.

Application Note Examples:

Using shared precompiled libraries:

```
ncvlog –messages –WORK WORK $XILINX/verilog/src/glbl.v tf_bram32x16.v time_sim.v
 ncelab -messages -autosdf test glbl
 ncsim -messages test
```

VCS

VCS simulations can be run in one of two ways:

1. Use library source files with compile-time options:

   For timing simulations or post-Ngd2ver, the Simprim-based libraries are used. At the command line, specify the following:

```
vcs +compsdf -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
+libext+.v -Mupdate -R <testfixture>.v time_sim.v
```

See Xilinx Solution 6349 (**http://support.xilinx.com/techdocs/6349.htm**) for instructions to back annotate the SDF file for timing simulations.

The -R option automatically simulates the executable after compilation

The -Mupdate option enables incremental compilation. Modules are recompiled for the following reasons:

a. The target of a hierarchical reference has changed.

b. A compile-time constant (for example, a parameter) has changed.

c. Ports instantiated in the module have changed.

d. Module in-lining. For faster simulation, groups of module definitions are merged into one module definition in the VCS file. The affected modules are recompiled. Recompilation only occurs once.

   Application Note Example:

Using library source files with compile-time options:

```
vcs +compsdf -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
+libext+.v -Mupdate -R tf_bram32x16.v time_sim.v
```

2. To use shared precompiled libraries

   For timing simulations or post-Ngd2ver, the Simprim-based libraries are used. At the command line, specify the following:

```
vcs +compsdf -Mupdate -Mlib=<compiled_lib_dir>/simprims_ver \
-y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v +libext+.v \
-R <testfixture>.v time_sim.v
```

   See Xilinx Solution 6349 (**http://support.xilinx.com/techdocs/6349.htm**) for instructions to back annotate the SDF file for timing simulations.

   The -R option automatically simulates the executable after compilation

   The -Mlib=<compiled_lib_dir> option provides VCS with a central location for descriptor information prior to compilation and a location for object files when it links the executable.

   The -M update option enables incremental compilation. Modules are recompiled for the following reasons:

   a. The target of a hierarchical reference has changed.

   b. A compile-time constant (for example, a parameter) has changed.

   c. Ports instantiated in the module have changed.

   d. Module in-lining. For faster simulation, groups of module definitions are merged into one module definition in the VCS file. The affected modules are recompiled. Recompilation only occurs once.

Application Note Example:

Using shared precompiled libraries:

```
vcs +compsdf -Mupdate -Mlib=/vcs/lib/simprims_ver \
-y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v +libext+.v \
-R tf_bram32x16.v time_sim.v
```

Verilog-XL

For timing simulations or post-Ngd2ver, the Simprim-based libraries are used.

At the command-line, specify the following:

```
verilog -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
 +libext+.v <testfixture>.v <design>.v
```

See Xilinx Solution 3167 (**http://support.xilinx.com/techdocs/3167.htm**) for instructions to specify Xilinx Simprim libraries using the -ul switch with Ngd2ver (instead of using the -y switch in Verilog-XL).

Application Note Example:

```
verilog -y $XILINX/verilog/src/simprims $XILINX/verilog/src/glbl.v \
 +libext+.v tf_bram32x16.v time_sim.v
```

Downloading and Installing IP Updates

IP Updates and instructions are available from the Xilinx website at:

http://support.xilinx.com/ipcenter/coregen/updates.htm

The general procedure for installing IP Update No. 2 is described below. For additional information, see the Update documentation (provided as part of the downloaded files and also available on the Xilinx IP Update download page at the link shown above).

IP Update No. 2 Install Instructions

- Although not required, Xilinx recommends installation of the latest 3.2i Software Service Pack.
- The IP Update is available in two formats: as a ZIP file and a TAR file. Windows users should unpack the ZIP file using WinZip (versions 7.0 SR-1 or later are recommended). Unix users should use the TAR file ("tar.gz", which is compressed with "gzip"). It can be unpacked with UNIX command line utilities gzip or tar (note the problems mentioned below that can occur with older versions of the UNIX "tar" commands). Unix users also may be able to use and unpack the ZIP file, but see Xilinx **Answer #7711** for issues related to this procedure.
- To install the Update No. 2:
  - Quit the CORE Generator application if is running.
  - Download 32i_ip_update2.zip (ZIP file) or 32i_ip_update2.tar.gz (TAR file), and save it in a temporary directory.
  - Extract the ZIP file or tar.gz archive to the $XILINX directory. Make sure to preserve relative paths (for example, `coregen/ip/xilinx/baseblox_v1_0/com/xilinx/ip/baseblox_v1_0/`).
- Delete the corelib.xml file located in the $XILINX/coregen/ip directory. This forces the CORE Generator to regenerate its database during startup.
- Restart the Xilinx CORE Generator. The Xilinx CORE Generator automatically detects that a new IP has been added and provides the user with three update options for the current CORE Generator project: 1) Update "All" cores to their latest versions, 2) Add only "New" cores, or 3) make a "Custom" selection of available cores. To confirm a successful installation, make sure the new cores are listed in the CORE Generator GUI (for example, confirm that "ADPCM for Virtex" appears in the "Communication & Networking/Telecommunication" folder).

Design Files Disclaimer and Download Instructions

Note: The application note design files are located at:

ftp://ftp.xilinx.com/pub/applications/xapp/XAPP410.zip

and

ftp://ftp.xilinx.com/pub/applications/xapp/XAPP410.tar.gz

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/11/01	1.0	Initial Xilinx release.