XAPP409 (v1.0) June 11, 2001

Simulating a Xilinx 3.1i CORE Generator VHDL Design

Summary

This application note provides an overview of the files that are generated from the Xilinx CORE Generator™ 3.1i for an HDL project and explains how and when each file is used. This application note briefly explains how to create simulator libraries, map the created libraries, and how to compile the XilinxCoreLib CORE Generator libraries to these user created libraries for the supported simulators. Also explained in this document is:

- The methodology for declaring and instantiating a CORE Generator macro, and different methodologies for simulating the macros
- How to load the design into the simulator
- How to run a functional and timing simulation
- How to download IP updates from the Xilinx website
- How to install the updates on the user's system

To obtain the VHDL code described in this document, go to section **VHDL Disclaimer and Download Instructions**, page 19 for instructions.

When using a Xilinx CORE Generator macro in the VHDL design, there are some basic steps to include the macro in the VHDL design.

1. Run the CORE Generator to create the macro of choice
2. Black box the CORE Generator module:
   - Use the VHO template for the component declaration (See **Instantiating a CORE Generator Macro**, page 5)
   - Use the VHO template for the component instantiation (See **Instantiating a CORE Generator Macro**, page 5)
3. Synthesize the design
4. Run the Xilinx implementation

Optional steps to run a functional simulation (in addition to the above steps):

1. Compile the Xilinx CORE Generator Libraries (See **Compiling the CORE Generator Libraries**, page 3)
2. Select and use a Configuration methodology in order to bind the component parameters selected with the precompiled model. A Configuration Declaration or a Configuration Specification methodology can be used. Use the VHO for the configuration template (See **Using Configurations for Simulation**, page 8)
3. Run the functional simulation. (See **Running a Functional Simulation**, page 15)

   Note: For Timing simulation, SIMPRIM libraries are used and no configurations are needed for simulation. (See **Running a Timing Simulation**, page 17)

The remainder of this application note describes the input/output files, steps needed to instantiate a CORE Generator macro and simulate it (functional and timing), and downloading IP Updates.

Input/Output Files

This section describes the input/output files for a created project and also the input/output files to the CORE Generator system.

Project Input Files

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

Project Input Files Description:

.COE ASCII data file. Defines the coefficient values for FIR Filters and initialization values for memory modules such as single/dual port RAM and ROM. See $XILINX/coregen/data for sample .COE files.

.XCO CORE Generator file containing the parameters used for regenerating a core. It can also be used as a logfile to determine the settings used to generate a particular core. This file is generated by the CORE Generator System along with any core that it creates in the current project directory. For details on the .xco file refer to the "XCO Files" section of the CORE Generator System User Guide 3.1i.

Project Output Files Overview

The .VEO and .VHO template files are for users to use to instantiate the cores into there existing HDL files. They are not used for implementation nor simulation directly.

The EDIF .EDN file is used for implementation only. The Xilinx implementation tool (NGDBuild) will pull this file in during the translate step, along with the other EDIF files for the entire design.

The .MIF file created by CORE Generator is used for simulation purposes only, and is generated from the .COE file. For distributed memory, the .MIF gets generated with initial default values when values are not specified.

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. Describes how the CORE is to be implemented. Used as input to the Xilinx Implementation Tools (NGDBuild).

.VEO Verilog Template file. The components in this file can be used as a guide to creating the core's Verilog instantiation and passing parameters to a Verilog behavioral model. For more details refer to the "Using the CORE Generator Verilog Design Flow Procedure" of 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 to pass parameters to the VHDL behavioral model via a configuration declaration. For more details, refer to the section "Using the CORE Generator VHDL Design Flow Procedure" of the CORE Generator System User Guide 3.1i.

.MIF Memory Initialization File which is automatically generated by the CORE Generator System for Virtex Block RAM modules when an HDL simulation flow is specified. A .MIF data file is used to specify the initialization values for the Block RAM modules during HDL functional simulation. To generate a MIF file you must direct CORE

Generator System to generate an EDIF Implementation Netlist and specify either a Verilog or a VHDL Behavioral Simulation output.

.PRJ The coregen.prj file contains a record of all installed COREs and their available versions. When a new project is created where all the options are chosen for the particular project the coregen.prj file is created. A link to the new project is also written to the known.prj file found in the $XILINX/coregen/preferences directory. This will allow users to load a previously created project from the CORE Generator GUI.

System Input/Output Files

The following files are defaultly located in the $XILINX/vhdl/src/XilinxCoreLib directory, except for the get_models.log file which is located in the $XILINX/vhdl/src directory. If you manually run get_models then the files will be located in the destination directory (get_models –dest *path_to_directory*), that you specify.

Note: If get_models is run manually and a destination directory is not specified, it will default to extract the files to the $XILINX/vhdl/src/XilinxCoreLib directory.

get_models.log

Log file containing all user visible messages displayed during a get_models run. The log file is written to the get_models destination directory.

verilog_analyze_order

This file lists the CORE Generator Verilog behavioral models in a suggested compilation order before performing a behavioral simulation in a compiled simulator. This applies to compiled Verilog simulators only, for example, Cadence NC-Verilog, MTI Verilog, and Synopsys VCS.

vhdl_analyze_order

This file lists the CORE Generator VHDL behavioral models in the order that they must be compiled for simulation. More than one compile 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 will describe how to create a library, map the library, and compile the XilinxCoreLib CORE Generator libraries to the created library.

Note: “$XILINX/vhdl/src/XilinxCoreLib” is the location where the XilinxCoreLib VHDL uncompiled libraries are located. get_models can still be run to extract all the files to a destination of choice, and compile the extracted models.

Note: The Xilinx 3.1i VHDL libraries are VHDL87 compliant.

MTI — VHDL

The following will pertain to ModelSim PE, EE, and SE. For ModelSim XE users, the CORE Generator libraries will be provided in precompiled format. If an IP Update has been downloaded and ModelSim XE is being used, the precompiled CORE Generator IP Update libraries will also need to be downloaded.

NOTE: For more information on compiling the Xilinx libraries (UNISIM, SIMPRIMS, etc.) refer to **http://support.xilinx.com/techdocs/2561.htm**

The Modelsim XE precompiled libraries can be downloaded from:

http://support.xilinx.com/support/mxelibs.htm

For information on how to compile the Xilinx CORE Generator libraries, refer to http://support.xilinx.com/techdocs/8066.htm. This solution also has a link to T_{CL} scripts that can be run from within MTI to compile the CORE Generator libraries.

1. Creating the library named “xilinxcorelib”. For VHDL, the library name must be xilinxcorelib, as this is the library referenced in the .VHO template file.

   ```
   vlib xilinxcorelib
   ```

2. Mapping the library such that the modelsim.ini is updated and ModelSim can identify the library.

   ```
   vmap xilinxcorelib complete/path/to/xilinxcorelib
   ```

   Note: If using the MODELSIM environment variable to point to a modelsim.ini, the user will need to make sure that write permissions are open to that particular modelsim.ini file. If not, the user will have to copy the modelsim.ini file to the local project directory, unset the MODELSIM variable, then run the vmap command.

3. Compiling the XilinxCoreLib libraries

   The vhdl_analyze_order file can be referenced for the order of compilation, located at $XILINX/vhdl/src/XilinxCoreLib/vhdl_analyze_order. This file is updated with every IP Update installed. This file can also be copied locally and modified to compile each line (vhdl file) in the file individually, but then the entire file can be executed all at once. The command would be similar to the following and would have to be added in front of every filename in the vhdl_analyze_order file:

   ```
   Vcom –work xilinxcorelib
   $XILINX/vhdl/src/XilinxCoreLib/filename.vhd
   ```

   Since there are a number of files that must be compiled, it is recommended to copy the vhdl_analyze_order file locally and add the above command to each line, instead of running the command manually for each file listed in the analyze_order file. Once this file is updated, save the file with a “.do” file extension. Then from ModelSim, execute the file by either running from the MTI command line “do analyze_order.do”, or from the pulldown menu Macro → Execute Macro and chose the file analyze_order.do.

VSS

To run simulation with VSS, the libraries must be precompiled and the CORE Generator libraries must be compiled by doing the following:

1. Create the physical library by creating a directory.

   ```
   mkdir xilinxcorelib
   ```

2. Map the created physical library to a library named “xilinxcorelib” in the .synopsys_vss.setup file.

   Example:

   ```
   xilinxcorelib : <path_to_directory>/xilinxcorelib
   ```

   The .synopsys_vss.setup needs to be in the same directory as where vhdlan is being run.

   The .synopsys_vss.setup must contain at least the following:

   ```
   TIMEBASE = PS
         WORK    > DEFAULT
         DEFAULT : .

         xilinxcorelib : <path_to_directory>/xilinxcorelib
   ```

```
-- VHDL library to UNIX dir mappings --
SYNOPSYS        : $SYNOPSYS/packages/synopsys/lib
IEEE               : $SYNOPSYS/packages/IEEE/lib
```

Note: Declare a working directory called xilinxcorelib to which the CORE Generator models will be compiled.

Note: An example .synopsys_vss.setup file is located at:

```
$XILINX/synopsys/examples/template.synopsys_vss.setup
```

3. Compile the XilinxCoreLib libraries.

   The vhdl_analyze_order file can be referenced for the order of compilation, located at $XILINX/vhdl/src/XilinxCoreLib/vhdl_analyze_order. This file is updated with every IP Update installed. This file can also be copied locally and modified to compile each line (vhdl file) in the file individually, but then the entire file can be executed all at once. The command would be similar to the following and would have to be added in front of every filename in the vhdl_analyze_order file:

   ```
   vhdlan -i -w xilinxcorelib
   $XILINX/vhdl/src/XilinxCoreLib/<filename>.vhd
   ```

   Since there are a number of files that must be compiled, it is recommended to copy the vhdl_analyze_order file locally and add the above command to each line, instead of running the command manually for each file listed in the analyze_order file.

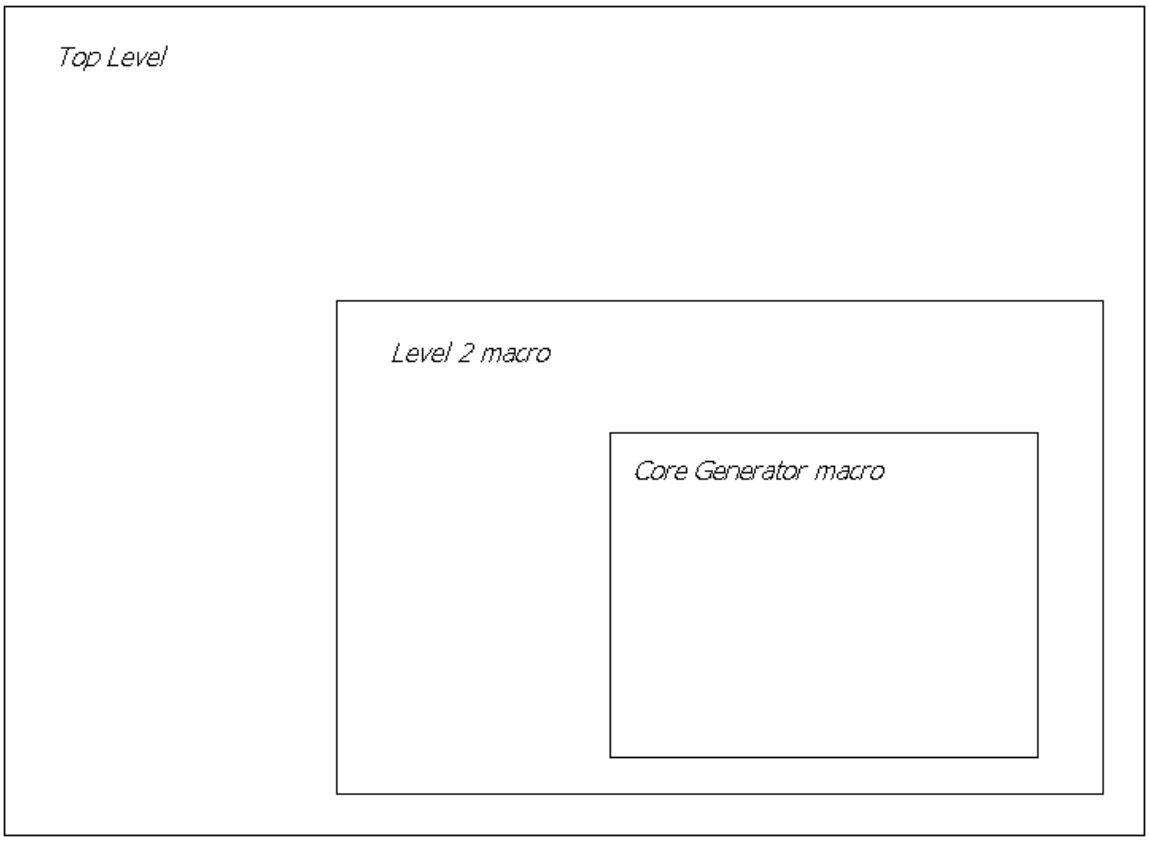
   See http://support.xilinx.com/techdocs/9755.htm for more information on compiling the Xilinx libraries (UNISIM, SIMPRIMS, etc.).

Instantiating a CORE Generator Macro

This section describes how to use the VHO template to cut and paste into existing VHDL files.

The design example used will have a CORE Generator macro down two levels in the hierarchy to illustrate how parameters are passed to the macros for functional simulation. The RAMB4_S16_S16 component is a 4096-bit dual-ported dedicated random access memory blocks with synchronous write capability. Each port is independent of the other while accessing the same set of 4096 memory cells. This example will not exercise the entire depth of the block RAM, but will be generating a 512-bit memory image.

The top-level instantiates a macro or second level, and the second level instantiates the CORE Generator macro.



In the testbench (tb.vhd), stimulus generation and response checking is done entirely in the code. Verifying results behaviorally is very efficient for simulation. If the design is not working properly, error messages are displayed as part of the text output.

VHDL

This section first describes how to use the VHO template to instantiate the core. Separate sections will explain how to use Configuration Declarations or Configuration Specifications to configure the CORE Generator core for simulation.

VHO file (Generated by CORE Generator)

The VHDL VHO template file can be used to cut and paste the library declaration, module declaration and instantiation, configuration declaration (to pass parameters to the simulation model), and example configuration declarations.

Top.vhd (Top-level that Instantiates the second level)

For this example, only the second level of hiearchy will be instantiated and bidirectional bus will be created. It is not necessary to cut and paste from the VHO into this file.

App Note Example for top.vhd:

The top level entity architecture pair is shown below.

```
Entity top is
…
end;

architecture struct of top is
   component level2 is
   (…
   );
   end component;
```

```
begin
   U0 : level2 port map
(…
);
   …
end struct;
```

Level2.vhd (second level of hierarchy that contains a CORE Generator macro)

The CORE Generator macro will be instantiated as a black-box in the VHDL code. Because of the "black-box" in the VHDL code, there will be a component declaration as well as a component instantiation in the HDL. For simulation, use a precompiled xilinxcorelib library, which will require a library declaration, and use a configuration declaration or specification to pass the appropriate parameters to the simulation model.

Library Declaration

For VHDL, declare the XilinxCoreLib library in the VHDL file that instantiates the CORE Generator macro. The CORE Generator macro is used for the simulator compiler since a precompiled CORE Generator simulation model (xilinxcorelib) is used to simulate the macro. The following can be added to the VHDL code that instantiates the macro.

```
-- synopsys translate_off

Library XilinxCoreLib;

-- synopsys translate_on
```

Component Declaration

The following section from the VHO file is used for the component declaration for the CORE Generator macro in the VHDL file. The component declaration must appear in the VHDL files architecture header, (before the architectures "BEGIN" statement).

```
component core32x16
   port (
   addra: IN std_logic_VECTOR(4 downto 0);
   clka: IN std_logic;
   addrb: IN std_logic_VECTOR(4 downto 0);
   clkb: IN std_logic;
   dia: IN std_logic_VECTOR(15 downto 0);
   wea: IN std_logic;
   dib: IN std_logic_VECTOR(15 downto 0);
   web: IN std_logic;
   ena: IN std_logic;
   enb: IN std_logic;
   rsta: IN std_logic;
   rstb: IN std_logic;
   doa: OUT std_logic_VECTOR(15 downto 0);
   dob: OUT std_logic_VECTOR(15 downto 0));
end component;
```

Component Instantiation

The following section from the VHO file is used for instantiating the CORE Generator macro into the VHDL file. The component declaration must appear in the VHDL files architecture body (after the architectures "BEGIN" statement). Change the instance name and the port connections to their own signal names.

```
your_instance_name : core32x16
     port map (
       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);
```

App Note Example for level2.vhd:

The second level (level2) of hierarchy in the design entity architecture pair and CORE Generator instantiation (core32x16) are shown below.

```
Entity level2 is
…
end;

Archtitecture struct of level2 is

   Component core32x16 port
    (…
    );
   end component;

Begin
U0 : core32x16 port map (…);
…
end struct;
```

Tb.vhd (Testbench)

The testbench can be the same for a “top level configuration” or with a “lower-level configuration”. The only difference is that with a lower level configuration methodology, there will be a configuration on each level of hierarchy all the way down to the level that contains the CORE generator macro, as opposed to having just one configuration at the top-level.

This testbench is only an example and will not fully stimulate the CORE Generator macro.

Using Configurations for Simulation

There are three methods to bind the simulation model to the component instantiation in order to simulate a CORE Generator macro.

1. Declared from the top-level and used to configure a coregen macro that is buried in hierarchy using a series of “For” statements to traverse the hierarchy.
2. The Configuration Declaration for the CORE Generator macro can be at the lower-level where the macro is at, and a Configuration Declaration is declared at this level.

The next level up in the hierarchy will have to have a configuration declaration pointing to the lower-level hierarchy configuration. The next level of hierarchy up from this will have to have a Configuration Declaration pointing to this levels configuration, and so on.

3. Use a Configuration Specification. This method uses a separate file used for simulation only that maps all the core parameters to the simulation model.

There are some advantages and disadvantages to using the different methods described above. Using Configuration Declarations, either a top-level or lower level, make sure to simulate

on the configuration, or else the parameters for the CORE Generator macros will not be mapped or bound. When compiling the file that instantiates the CORE Generator module, there will be warnings about an unbound instance, since the model for the macro has been precompiled, and gets mapped in the configuration which must be compiled last.

One advantage to using the lower level configuration, since there is a configuration on each level of hierarchy above it, is that it will allow a simulation to take place at any level in the hierarchy, as long the simulation is run on the configuration at that particular level. As opposed to the top-level configuration, the configuration would have to be modified to simulate at a particular point in the hierarchy. Another disadvantage to using configuration declarations is that they are used for functional simulation only.

For a functional simulation, simulate on the configuration and on the back-end for timing simulation while running the simulation on the testbench entity/architecture. A testbench that contains the configuration (from the functional simulation) can be used for a timing simulation. The simulation must take place on the entity/architecture and not on the configuration. Anytime the configuration changes using a top-level configuration, only the file (or testbench depending on where the configuration is), will have to be recompiled. If the testbench is large, have the configuration in a separate file. When using a lower-level configuration, the file that contains the configuration will have to be recompiled, which could be a disadvantage if that particular VHDL is large.

The configuration specification method allows simulation using the same testbench in the front and back-end. There will be no warnings about unbound instances since the (simulation only) macro file will be compiled first. Also the user will not have to keep track of all the entity-architecture pairs in the design hierarchy, as when using a top-level or lower level configuration declaration style. When using this method, anytime the configuration changes the entity/architecture will have to be recompiled to realize the changes. The advantage is that this file will only contain the configuration parameters, and will be relatively small. The following section describes the different methods for configurations.

Using a Top-level Configuration Declaration

Use a top-level Configuration Declartion when the Configuration Declaration is at the very top level (from the simulation stand-point), such as in the testbench. The configuration can also be in its own file, which can then be used for functional simulation. In doing this when running a timing simulation, the configuration will not have to be compiled, and the testbench can then still be used.

The entire hierarchy can be configured from a single declaration as in the following syntax:

```
Configuration config_name of entity_name is
    For architecture_name
   For instance_label:component_name
       Use entity
       Library_name.entity_name(architecture_name);
     For arch_name
         …
         lower-level configurations
         …
     end for;
    end for;
    end for;
end config_name;
```

Note: In the statement "use configuration library_name.entity_name", note that the library_name is going to be the name of the library that is being compiled to that particular VHDL file. Commonly used is the library "work".

Cfg.vhd (Configuration Declaration in a single file)

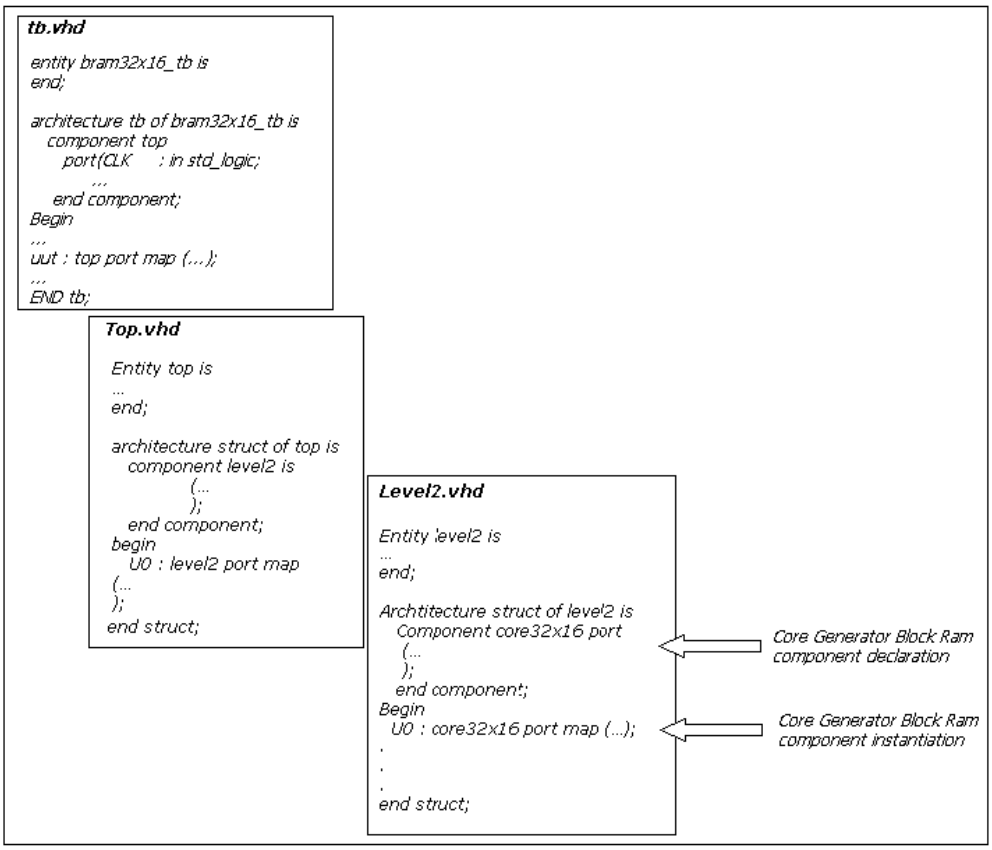
The Configuration Declaration can be in the testbench file. For this example, the configuration is put into its own file. When in its own file (for the backend timing simulation when the testbench is compiled), the configuration will not show up as a component since it will not be compiled.

When writing a top-level single configuration, keep track of the entity/architecture pairs and the component/s that are being configured in the configuration throughout the hierarchy that is being traversed.

App Note Example for cfg_top.vhd (complete configuration example):

The design entity/architecture pairs and component instantiations are shown below and include the testbench entity/architecture pair. The tb.vhd file instantiates top.vhd.

Note: The actual configuration can be in the testbench or in a separate file, but must configure the testbench entity/architecture pair.



The resulting configuration for the above example is shown below (in a separate file from the testbench):

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

Library XilinxCoreLib;

configuration cfg_top_lvl_config of bram32x16_tb is
for tb
      for uut: top  use entity work.top(struct);
         for struct
           for U0 : level2 use entity work.level2(struct);
```

```
                for struct
                   for all : core32x16 use entity \
             XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
                         generic map(
                           c_depth_b => 32,
                           c_depth_a => 32,
                           c_has_web => 1,
                           .
                            .
                            .
                           c_wea_polarity => 1);
                      end for;
               end for;
    end for;
       end for;
     end for;
end for;
end cfg_top_lvl_config;
```

Note: In the above example, “work” is the library that the VHDL files are being compiled to.

Using a Lower-level Configuration Declaration

All of the same VHDL files, tb.vhd, top.vhd, and level2.vhd can be used for a top-level configuration. In the top-level single configuration method, the configuration is in a single declaration; the testbench, top-level, and second level were not modified. For the lower-level configuration methodology, what needs to be added to the existing files will be described later. Again, the differences in the lower-level method configurations will be added to every level of hierarchy, as well as to the configuration for the testbench which will be different.

A Configuration Declaration is used at the level of hierarchy where the macro is instantiated at, and also has a configuration in each level above it.

The following is a configuration form for a tree of config declarations

```
configuration config_name of entity_name is
    for architecture
        for instance:component_name
        use configuration library_name.config_name;
        end for;
    end for;
end config_name;
```

Note: In the statement "use configuration library_name.config_name;" note that the library_name is going to be the name of the library that is being compiled to that particular VHDL file. Commonly used is the library "work".

Top.vhd (Top-level that Instantiates the second level)

For this example, only the second level of hierarchy will be instantiated and a bidirectional bus will be created; the template for the configuration declaration is used for this example. The following lines will have to be added to the top.vhd file in order for the configuration to be correctly compiled and used.

App Note Example for top.vhd

The following can be added to the existing top.vhd file. Refer to appropriate section in the “VHO file (Generated by CORE Generator)” section for the entity/architecture pairs that the configurations shown below are referencing.

```
configuration cfg_top of top is
for struct
    for U0 : level2
   use configuration work.cfg_level2;
```

```
    end for;
end for;
end cfg_top;
```

Note: In the above example, "work" is the library that the VHDL files are being compiled to.

Level2.vhd (second level of hierarchy that contains a CORE Generator Macro)

The CORE Generator macro will be instantiated as a black-box in the VHDL code. Because of the "black-box" in the VHDL code, there will be a component declaration as well as a component instantiation in the HDL. For simulation, use a precompiled xilinxcorelib library, which will require a library declaration, and use a configuration declaration or specification to pass the appropriate parameters to the simulation model.

App Note Example for level2.vhd:

The second level (level2) of hierarchy in the design entity architecture pair and CORE Generator instantiation (core32x16) is shown below.

The following can be added to the existing level2.vhd file. Refer to the appropriate section in the “VHO file (Generated by CORE Generator)” section for the entity/architecture pairs that the configurations shown below are referencing.

```
Library XilinxCoreLib;

Configuration cfg_level2 of level2 is
for struct
    for all : core32x16 use entity
XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
                generic map(
                        c_depth_b => 32,
                        c_depth_a => 32,
                        c_has_web => 1,
                        .
                         .
                         .
                        c_wea_polarity => 1);
    end for;
end for;
end cfg_level2;
```

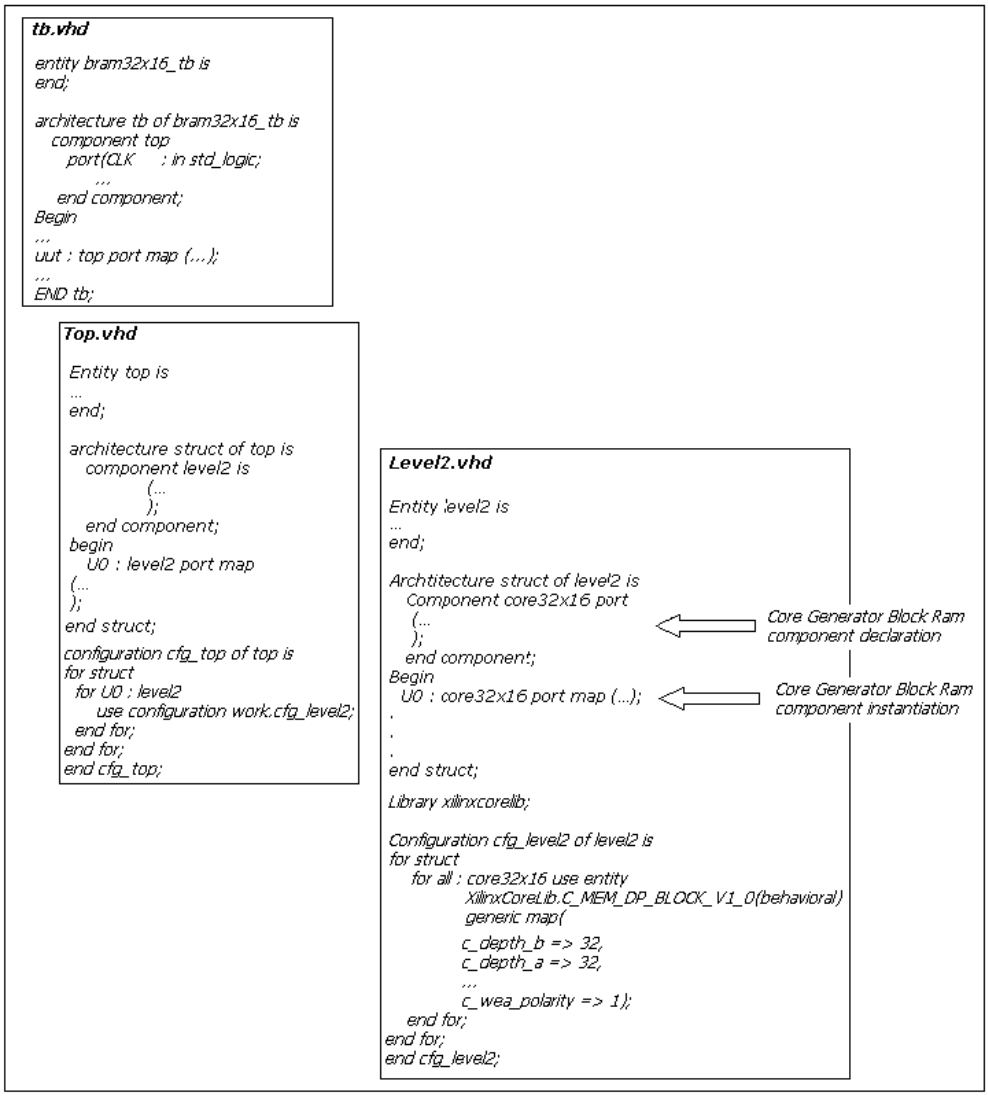
Tb.vhd (Testbench)

The testbench can be the same for a “top level configuration” or a “lower-level configuration”. The only difference is that with a lower level configuration methodology, there will be a configuration on each level of hierarchy all the way down to the level which contains the CORE generator macro, as opposed to having just one configuration at the top-level.

Cfg.vhd (Configuration Declaration file)

The Configuration Declaration can be in the testbench file. For this example, the configuration is put into its own file. When in its own file (for the backend timing simulation when the testbench is compiled), the configuration will not show up as a component since it will not be compiled.

App Note Example for cfg_low.vhd (complete configuration example):



The configuration to configure the testbench would then look like the following:

```
Configuration cfg_low_lvl_config of bram32x16_tb is
    for tb
        for uut : top
            use configuration work.cfg_top;
        end for;
    end for;
end cfg_low_lvl_config;
```

Note: In the above example, "work" is the library that the VHDL files are being compiled to.

Using a Configuration Specification

A Configuration Specification style can be utilized in a way such that a Configuration Declaration will not have to be used. Instead, a separate simulation file will be created for simulation purposes only and is used for a particular component's configuration parameters. There will be a separate VHDL file for each CORE Generator module. Using this method, components can be configured in the architecture that instances them, as opposed to being configured in a separate Configuration Declaration.

The same files that are used for "Using a Top-Level Configuration Declaration" can be used except for the cfg_top.vhd file, which is not required. Instead, a separate simulatable VHDL file

for the CORE Generator macro will be created from the VHO file. This will result in being able to simulate on the same testbench entity/architecture pair for functional and timing simulation.

App Note Example for core32x16.vhd (CORE Generator macro):

```
library IEEE;
use IEEE.std_logic_1164.all;

Library XilinxCoreLib;

entity core32x16 is
   port (
   addra: IN std_logic_VECTOR(4 downto 0);
   clka: IN std_logic;
   addrb: IN std_logic_VECTOR(4 downto 0);
   clkb: IN std_logic;
   dia: IN std_logic_VECTOR(15 downto 0);
   wea: IN std_logic;
   dib: IN std_logic_VECTOR(15 downto 0);
   web: IN std_logic;
   ena: IN std_logic;
   enb: IN std_logic;
   rsta: IN std_logic;
   rstb: IN std_logic;
   doa: OUT std_logic_VECTOR(15 downto 0);
   dob: OUT std_logic_VECTOR(15 downto 0));
end core32x16;

architecture core32x16_sim_arch of core32x16 is

BEGIN

  core32x16_SIM : entity
XilinxCoreLib.C_MEM_DP_BLOCK_V1_0(behavioral)
   generic map(
     c_depth_b => 32,
     c_depth_a => 32,
     c_has_web => 1,
     c_has_wea => 1,
     c_has_dib => 1,
     c_has_dia => 1,
     c_clka_polarity => 1,
     c_web_polarity => 1,
     c_address_width_b => 5,
     c_address_width_a => 5,
     c_width_b => 16,
     c_width_a => 16,
     c_clkb_polarity => 1,
     c_ena_polarity => 1,
     c_rsta_polarity => 1,
     c_has_rstb => 1,
     c_has_rsta => 1,
     c_read_mif => 1,
     c_enb_polarity => 1,
     c_pipe_stages => 0,
     c_rstb_polarity => 1,
     c_has_enb => 1,
     c_has_ena => 1,
```

```
      c_mem_init_radix => 16,
      c_default_data => "0",
      c_mem_init_file => "./core32x16.mif",
      c_has_dob => 1,
      c_generate_mif => 1,
      c_has_doa => 1,
      c_wea_polarity => 1)
    port map (
      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);

end core32x16_sim_arch;
```

Note: The path to the .MIF file may need to be modified.

Running a Functional Simulation

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

MTI - VHDL

1. Create a library to compile to, if not already created.

   ```
   vlib library_name
   ```

2. Compile the VHDL files.

   ```
   Vcom file1.vhd file2.vhd file3.vhd
   ```

3. Load the design on the Configuration Declaration in order for the parameters/generics specified in the configuration to be applied to the design.

   ```
   Vsim config_name
   ```

4. The GUI will start up and waves and signals can be added and the design can be simulated.

App Note Example:

```
vlib work
```

Using a Top-Level Configuaration Declaration:

```
vcom level2.vhd top.vhd
vcom –93 tb.vhd
vcom cfg_top.vhd
vsim –t ps cfg_top_lvl_config
add wave *
run -all
```

Using a Lower-Level Configuration Declaration:

```
vcom level2.vhd top.vhd
```

```
vcom –93 tb.vhd
vcom cfg_low.vhd
vsim –t ps cfg_low_lvl_config
add wave *
run -all
```

Using a Configuration Specification:

```
vcom -93 core32x16.vhd
vcom level2.vhd top.vhd
vcom top.vhd
vcom -93 tb.vhd
vsim -t ps ex_blkram_tb
add wave *
run -all
```

VSS

1. VSS requires a WORK directory to put the compiled data in. First, create the WORK directory in the project directory as follows:

   ```
   mkdir WORK
   ```

2. Make sure the following line is added to the .synopsys_vss.setup file:

   ```
   WORK: /path_to_directory/WORK
   ```

3. Compile the files:

   ```
   vhdlan -i file1.vhd file2.vhd
   ```

   ```
   vhdlan -i testbench.vhd
   ```

   ```
   vhdlan –i cfg_top.vhd (If the configuration is not in the testbench)
   ```

4. Load the simulation:

   ```
   vhdlsim -e commandfile.txt cfg_tstbench
   ```

The commandfile.txt contains the commands to start up the waveform viewer, add signals and stimulus. For more information on the commandfile.txt, consult the Synopsys VSS documentation.

App Note Example:

Using a Top-Level Configuaration Declaration:

```
Vhdlan -i level2.vhd top.vhd tb.vhd
Vhdlan –i cfg_top.vhd
vhdlsim –e commandfile.txt cfg_top_lvl_config
```

Using a Lower-Level Configuration Declaration:

```
Vhdlan -i level2.vhd top.vhd  tb.vhd
Vhdlan –i cfg_low.vhd
vhdlsim –e commandfile.txt cfg_low_lvl_config
```

Using a Configuration Specification:

```
Vhdlan –i core32x16.vhd level2.vhd top.vhd tb.vhd
Vhdlsim –e commandfile.txt cfg_tb.vhd
```

Note: VSS requires users to simulate on the configuration if it is used or not. In the above example, the configuration is not doing anything.

Functional Simulation Hints, Tips, and Common Problems

This section gives examples of common problems and situations to watch out for when running a functional simulation.

General

- Path to the MIF file: When using the VHO template file, be sure to modify the path to the MIF file when initializing any components contents.
- Many times troubleshooting "U" and "X" coming out of the coregen macro can be difficult to solve. The most common problems are related to the component and/or model binding:
  - Component not getting bound. Be sure that the particular component is getting loaded. Check the log files to see what is being loaded.
  - Make sure to simulate on the Configuration if using the Configuration Declaration method
- Be sure to declare the Xilinxcorelib library in the VHDL file that contains the configuration.
  - Make sure all inputs are at known levels.
- Make sure the xilinxcorelib library is compiled and also that it is mapped for the particular simulator being used. For instance, in MTI go to Design → Browse Libraries, then verify that the xilinxcorelib library is mapped, then verify the component being used exists in that library.

VHDL

- There will be warning messages about VHDL unbound instance when using any type of Configuration Declaration method. The warnings are normal and can be ignored as long as the following is done:
  - Be sure to load the simulation on the Configuration and not on the top level entity/architecture pair
  - Make sure that when the design is loading that the coregen model is loaded last
  - Be Sure to load the simulation on the configuration when using the Configuration Declaration methodology.
- Using a Configuration Declaration top-level method produces a lot of hiearchy. The Configuration “For” structure can be confusing. It may be easier to use a Configuration Specification method described in this App Note.

Running a Timing Simulation

This section describes how to load the backend timing simulation into the various simulators. See the appropriate simulator section in “Compiling the CORE Generator Libraries” for links to the appropriate solutions to compile the Xilinx Simprim libraries which are needed for the compiled simulator.

MTI - VHDL

1. Create a library to compile to, if not already created.

   ```
   vlib library_name
   ```

2. Compile the VHDL files.

   ```
   Vcom time_sim.vhd testbench.vhd
   ```

3. Load the design on testbench entity/architecture pair.

   ```
   Vsim –t ps testbench_arch
   ```

4. The GUI will start up and waves and signals can be added andthe design can be simulated.

App Note Example:

```
vlib work
vcom time_sim.vhd tb.vhd
vsim –t ps –sdfmax uut=./time_sim.sdf tb
add wave /*
run -all
```

VSS

1. VSS requires a WORK directory to put the compiled data in. First, create the WORK directory in the project directory as follows:

   ```
   mkdir WORK
   ```

2. Make sure the following line is added to the .synopsys_vss.setup file:

   ```
   WORK: /path_to_directory/WORK
   ```

3. Compile the files:

   ```
   vhdlan -i time_sim.vhd
   ```

   ```
   vhdlan -i testbench.vhd
   ```

4. Load the simulation:

   ```
   vhdlsim -e commandfile.txt cfg_tb_arch
   ```

The commandfile.txt contains the commands to start up the waveform viewer, add signals and stimulus. For more information on the commandfile.txt, consult the Synopsys VSS documentation.

> **Note:** VSS requires users to simulate on the configuration if it is used or not. In the above example, the configuration is not doing anything.

App Note Example:

```
Vhdlan –i time_sim.vhd tb.vhd
Vhdlsim –e commandfile.txt cfg_tb
```

Downloading and Installing IP Updates

The latest 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 #2 is described below, but refer to the latest documentation for the Update, either in the downloaded files or from the IP Update download page.

IP Update # 2 Install Instructions

- While not required, it is recommended the latest 3.2i Software Service Pack be installed.
- This IP Update is available both as a .zip fil, and as a .tar file, which has been compressed using "gzip". The Zip file can be unpacked using a recent release of WinZip (such as 7.0 SR-1 or later) on Windows. On Unix, some versions of Unzip as may be used to unpack this Zip file on UNIX, but see Xilinx Answer #7711 for information on some known issues with this. On UNIX platforms, it is recommended that the ".tar.gz" file be downloaded and unpacked using the UNIX command line utilities gzip and tar. (Note the problems seen with older UNIX "tar" commands below.)
- Quit the CORE Generator application if it is running.
- Download the zip file 32i_ip_update2.zip or 32i_ip_update2.tar.gz and save it to a temporary directory.
- Extract the zip file or tar.gz archive to the $XILINX directory, preserving the relative paths such as: coregen/ip/xilinx/baseblox_v1_0/com/xilinx/ip/baseblox_v1_0/
- Delete the corelib.xml file located in $XILINX/coregen/ip to force the CORE Generator to

regenerate this database during startup.

- Restart the Xilinx CORE Generator. The Xilinx CORE Generator will automatically detect that new IP has been added and will provide an opportunity to update "All" cores to the latest versions, add only "New" cores, or make only a "Custom" selection of cores available for the current CORE Generator project. Verify whether the installation has succeeded by verifying that the new cores can be seen in the CORE Generator GUI, such as "ADPCM for Virtex" in the "Communication & Networking/Telecommunication" folder.

VHDL Disclaimer and Download Instructions

Limited Warranty and Disclaimer. These designs are provided "as is". Xilinx and its licensors make and you receive no warranties or conditions, express, implied, statutory or otherwise, and Xilinx specifically disclaims any implied warranties of merchantability, non-infringement, or fitness for a particular purpose. Xilinx does not warrant that the functions contained in these designs will meet your requirements, or that the operation of these designs will be uninterrupted or error free, or that defects in the Designs will be corrected. Furthermore, Xilinx does not warrant or make any representations regarding use or the results of the use of the designs in terms of correctness, accuracy, reliability, or otherwise.

Limitation of Liability. In no event will Xilinx or its licensors be liable for any loss of data, lost profits, cost or procurement of substitute goods or services, or for any special, incidental, consequential, or indirect damages arising from the use or operation of the designs or accompanying documentation, however caused and on any theory of liability. This limitation will apply even if Xilinx has been advised of the possibility of such damage. This limitation shall apply not-withstanding the failure of the essential purpose of any limited remedies herein.

ftp://ftp.xilinx.com/pub/applications/xapp/XAPP409.zip

or

ftp://ftp.xilinx.com/pub/applications/xapp/XAPP409.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.