Method to Instantiate and Use a Core in *Warp*™ with Cypress CPLDs

Introduction

In order to meet the demand for increasingly complex designs, Cypress has formed IP Oasis – A partnership program with leading IP vendors to provide cores for Cypress CPLDs. Cores are blocks of IP that have been tested and optimized specifically for Cypress CPLDs. These cores represent a multitude of functions like PCI, SDRAM, UTOPIA, HDLC and more. Having pre-verified and optimized solutions for functional blocks enables designers to concentrate on developing the custom logic, resulting in a significant reduction in design cycle time. This application note describes in detail how customers can incorporate cores in their system level designs. The cores are available in the VIF file format, generated by *Warp*™ – Cypress's CPLD development system. This note contains a detailed description of the steps required to instantiate VIF files in both VHDL and Verilog designs.

How to Create VIF files in *Warp*

In most cases, the user will not need to perform the steps mentioned in this section as the VIF files and wrappers can be downloaded from the IP vendor's web page.

1. Create a new project in *Warp* and import the correct VHD or Verilog files into this new project.
2. Disable the 'Set Top' icon if it shows on any file by clicking the button shown below. This will cause *Warp* to compile the file to the VIF format, but not fit it. If there is only one file in the project, go to "Compile -> Selected File(s)" to create the VIF file for this one file. Do not compile using the "Compile -> Project" option as the '**Set Top**' icon is automatically placed on the file.

This is the '**Set Top**' button

3. Choose the device to fit by going to "Project -> Select Device...". The data sheet for the cores will state the device the downloaded core is targeted to. As an example, suppose the device selected is c39k100.
4. Compile the entire project. This will create VIF files for each of the files compiled in this project. If the 'Set Top' feature in step 2 is not turned off, no VIF file is created for the file with the 'Set Top' feature. The VIF files are placed in a directory that has the same name as the device selected with a 'lc' prefix. For this example, the directory name is 'lc39k100'. The path of this directory is the same as the path of the project created in step 1.
5. To generate VIF files for a different device, repeat step 3 – 4

Preparing VIF files for use in *Warp*

When a user receives a core package from a 3rd party IP Vendor, the cores are in the form of VIF files. VIF files are used by *Warp* to describe functional blocks such as cores. To use VIF files in a design, do the following:

1. In the same directory as the newly created *Warp* project, create a new folder. The name of this folder should be the same as the device the design is fit to, with a 'lc' prefix attached to it. For example, if the target device is c39k100, then the new folder name is 'lc39k100'.
2. Copy the entire folder containing the VIF files and the index file into this newly created folder.

How to Create a Wrapper for VIF files in VHDL

Though Cypress cores are supplied with wrappers, this section is included to illustrate how a wrapper is constructed.

1. Find the entity declaration of the original source file. For example, the VHDL code below contains the original Entity declaration of the source file from which the VIF was created.

```
library ieee;
use ieee.std_logic_1164.all;
entity VIFexample is
    port(
        rxclav_int: out std_logic;
        rx_tristate_clav: out std_logic;
        rxaddr: in std_logic;
        rxenb: in std_logic;
        tristate_all_outputs: out std_logic;
        rxsoc_int: out std_logic;
        rxdata_int: out std_logic;
        rxprty_int: out std_logic;
        underrun: out std_logic
    );
end VIFexample;
```

2. Create the entity of the wrapper with the same ports as the entity declaration of the original file. The user can choose a different name or the same name as the entity in the VIF file, but keep the mode and type the same. In the VHDL code below, the prefix 'top' was added to all port names.

```
entity VIFVHDLWrapper is
    port(
        toprxclav_int: out std_logic;
        toprx_tristate_clav: out std_logic;
        toprxaddr: in std_logic_vector(4 downto 0);
        toprxenb: in std_logic;
        toptristate_all_outputs: out     std_logic;
        toprxsoc_int: out std_logic;
        toprxdata_int: out std_logic_vector(15 downto 0);
        toprxprty_int: out std_logic;
        topunderrun: out std_logic
    );
end VIFVHDLWrapper;
```

3. After the architecture section of this wrapper before the 'begin' reserved word, create a component with the same name as the entity in the VIF file. In the code below, the component name is VIFExample, which is the same as the entity name declared in step 1.

```
.........
architecture arch_VHDLWrapper of VIFVHDLWrapper is
component VIFExample
port (
    rxclav_int: out std_logic;
    rx_tristate_clav: out std_logic;
    rxaddr: in std_logic_vector(4 downto 0);
    rxenb: in std_logic;
    tristate_all_outputs: out std_logic;
    rxsoc_int: out std_logic;
    rxdata_int: out std_logic_vector(15 downto 0);
    rxprty_int: out std_logic;
    underrun: out std_logic
);
end component;
begin
.....
```

4. In the architecture body after the 'begin' section, instantiate the component and map the component's ports to the top-level signals. In the example below, all the signals on the right side of the '=>' sign are signals declared in the entity section of this wrapper in step 2.

```
U1: VIFExample
    port map (
        rxclav_int => toprxclav_int,
        rx_tristate_clav => toprx_tristate_clav,
        rxaddr => toprxaddr,
        rxenb => toprxenb,
        tristate_all_outputs => toptristate_all_outputs,
        rxsoc_int => toprxsoc_int,
        rxdata_int => toprxdata_int,
        rxprty_int => toprxprty_int,
        underrun => topunderrun
    );
```

5. To compile, fit the design and create a simulation model, please see section "Compiling, Fitting, and Creating a Simulation Model".

How to Create a Wrapper for VIF files in Verilog

Though Cypress cores are supplied with wrappers, this section is included to illustrate how a wrapper is constructed.

In this section, assume that the entity declaration of the VIF file is the same as the code displayed in step 1 of section "How to Create a Wrapper for VIF files in VHDL".

1. Create a new project and a new .v file. This will be the wrapper file for the VIF file.
2. Create a module with the appropriate parameters. Usually, they are the same as the signal declaration of your module as shown below.

```
--This is the beginning of the Verilog wrapper
module VerilogVIFWrapper(toprxclav_int,
    toprx_tristate_clav, [4:0]toprxaddr, toprxenb,
    toptristate_all_outputs, toprxsoc_int,
    [15:0]toprxdata_int,toprxprty_int, underrun)
-- the below is the signal declaration to map to the VIF entity
output toprxclav_int;
output toprx_tristate_clav;
input [4:0]toprxaddr;
input toprxenb;
output toptristate_all_outputs;
output toprxsoc_int;
output [15:0]toprxdata_int;
output toprxprty_int;
output underrun;
```

-- the above are IO declarations for instantiating entity of VIF file.

3. Instantiate the entity of the VIF files in the verilog file created in step 1. Note: Verilog is case sensitive, so make sure that the component ports are the same as the entity declaration of the VIF file.

-- this code continues from the Verilog code above and it instantiates the entity in the VIF file.

```
VIFExample myVIF
    (.rxclav_int (toprxclav_int),
    .rx_tristate_clav (toprx_tristate_clav),
    .rxaddr (toprxaddr),
    .rxenb (toprxenb),
    .tristate_all_outputs (toptristate_all_outputs),
    .rxsoc_int (toprxsoc_int),
    .rxdata_int (toprxdata_int),
    .rxprty_int (toprxprty_int),
    .underrun (topunderrun)
    );
```

4. To compile, fit the design and create a simulation model, please see section “Compiling, Fitting and Creating a Simulation Model”.

How to use Downloaded Wrapper and VIF Files in VHDL

The steps described in this section are intended to help users who have received core VIF files and wrappers and want to use them in a VHDL design. Instantiating core with wrappers is the same as instantiating any other component in VHDL.

1. In the user’s design, create a component with the same name as the entity of the wrapper file. This component should be declared after the architecture section before the ‘begin’ reserved word. An example of component declaration for the VHDL wrapper described in section “How to Create a Wrapper for VIF files in VHDL” is shown below.

```
architecture arch_userdesign of userdesign is
-- the below is the component declaration placed between the architecture and the begin reserved words.
component VIFVHDLWrapper
port(
        toprxclav_int: out std_logic;
        toprx_tristate_clav: out std_logic;
        toprxaddr: in std_logic_vector(4 downto0);
        toprxenb: in std_logic;
        toptristate_all_outputs: out std_logic;
        toprxsoc_int: out std_logic;
        toprxdata_int: out std_logic_vector(15 downto 0);
        toprxprty_int: out std_logic;
        topunderrun: out std_logic
);
begin
.......
```

2. In the architecture body after the ‘begin’ section, instantiate the component and map the IOs to the signals of the wrapper. In the example below, all the signals on the right side of the “=>” sign are signals the user plans to use in the design with this wrapper component.

```
.....
begin
-- the following code is a continuation of the above code consisting of component declaration.
u1: VIFVHDLWrapper (
        toprxclav_int => “fill this section with proper signals”
        toprx_tristate_clav => insert proper signals here.
        toprxaddr => insert proper signals here
        toprxenb =>
        toptristate_all_outputs =>
        toprxsoc_int =>
        toprxdata_int =>
        toprxprty_int =>
        topunderrun =>
        );
```

After instantiating this wrapper, the user can link the signals in his/her own design.

How to use Downloaded Wrapper and VIF Files in Verilog

The steps described in this section are intended to help users who have received core VIF files and wrappers and want to use them in a Verilog design. Instantiating core with wrappers is the same as instantiating any other module in Verilog.

1. A Verilog example for the Verilog wrapper created in section “How to Create a Wrapper for VIF Files in Verilog” is shown below. This is instantiated in the body of the Verilog code.

```
module userdesign (.....
.....
output signal1 -- signal declaration
......
-- this is placed after the signal declaration in Verilog code
VerilogVIFWrapper myVerilogwrapper
    (.toprxclav_int(signal1),
    .toprx_tristate_clav(signal2),
    .toprxaddr(signal3),
    .toprxenb(signal4),
    .toptristate_all_outputs(signal5),
    .toprxsoc_int(signal6),
    .toprxdata_int(signal7),
    .toprxprty_int(signal8),
    .topunderrun(signal9)
    );
```

Signal1, Signal2, etc. are signals defined in the user’s design and are linked to the wrapper’s signals.

In Verilog compilation, make sure the wrapper is compiled before the user's design so that *Warp* can properly link the wrapper to the appropriate file that instantiate it.

Compiling, Fitting, and Creating a Simulation Model

1. Make sure that the "Enable Testbench Output" is set in *Warp* and the file is fitted. This is accomplished by going to "Project -> Compiler Options..." and click on the '**Synthesis**' Folder. An example of this is shown in *Figure 1*.
2. If the user is compiling a post-fit netlist to be simulated in Mentor Graphic's *ModelSim*, the user should choose the Timing Model to be "ModelT" as shown in *Figure 1*.
3. If the user is compiling a post-fit netlist to be simulated in *Aldec*™, the user should choose the Timing Model to be "Active - HDLSim / Active VHDL"

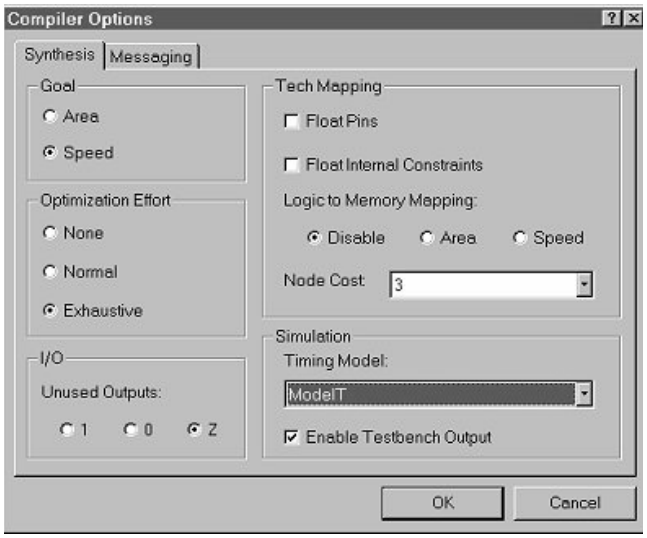

Figure 1. Setting Testbench Output for Simulation in *ModelSim*.

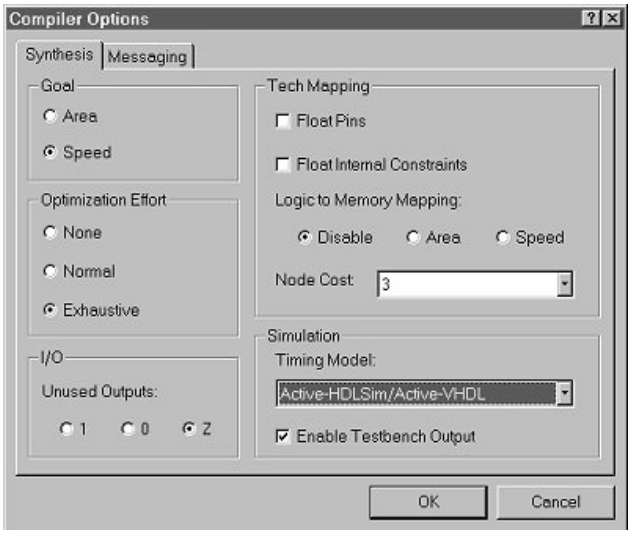

Figure 2. Setting Timing Model to Be Compatible with *Aldec* Simulation Tool

4. Go to "Compile -> Project" to generate post-fit netlists for *ModelSim*. The default output netlist is a VHDL file, but other output options can be chosen. In the default setting, netlists are located in a VHD folder having the same path as the *Warp* project.

For more information on importing and simulating the post-fit model in *ModelSim*, please refer to application note "Importing *Warp* post-fit netlists into Mentor Graphics' *ModelSim*". For more information on simulating post-fit model in *Aldec*, please refer to application note "An Introduction to Active-HDL™ Sim".

Making Core Input/Output Signal Visible in User's Design

When a core is instantiated in a design, Warp will optimize the Input/Output logic of the Core. This may result in some of the input and output signals not being visible in the simulation model. To make these signals visible, the "Synthesis_off" attribute should be set only on the signals that are non-visible. This however, may result in slower performance of the complete design. This is only recommended for understanding the behavior of the core and not for final implementation.

Conclusion

Through applying the procedure described in this application note, the user can synthesize and fit Cypress CPLDs in Cypress software, increasing speed and reducing design cycle time.