APPLICATION GUIDELINES

Integrating Xilinx System Generator with Simulink HDL Coder

The MathWorks™

Model-Based Design with Simulink, Simulink HDL Coder, and Xilinx System Generator

MathWorks tools for Model-Based Design provide signal, image, and video processing engineers with a development platform that spans design, modeling, simulation, code generation, and implementation.

Engineers who use Model-Based Design to target FPGAs or ASICs can design and simulate systems with MATLAB®, Simulink®, and Stateflow® and then generate bit-true, cycle-accurate, synthesizable Verilog® and VHDL® code using Simulink HDL Coder™.

Alternatively, engineers who specifically target Xilinx FPGAs can use a Xilinx library of bit- and cycle-true blocks to build a model in Simulink. They can then use Xilinx® System Generator for DSP, a plug-in to Simulink code generation software, to automatically generate synthesizable hardware description language (HDL) code mapped to pre-optimized Xilinx algorithms.

Used independently, each approach provides an effective FPGA design flow. Some projects, however, benefit from a mixture of approaches – a workflow that combines the native Simulink workflow, device-independent code, and code readability offered by Simulink HDL Coder, with the Xilinx FPGA-specific features and optimizations offered by Xilinx System Generator.

About this Guide

This guide describes an integrated workflow for engineers who want to:

- Integrate a Xilinx System Generator design into a Simulink model for HDL code generation
- Implement part of a Simulink design, intended for HDL code generation, in Xilinx System Generator

The workflow includes the following steps:

- Configuring a model for HDL code generation
- Generating HDL code
- Synthesis and implementation
- HDL simulation and verification

Appendix A, "Using Verilog for the Integration Workflow," describes changes to the workflow needed to generate Verilog code instead of VHDL.

Appendix B, "Synthesizing the Design Using Synplify Pro," explains how to use Synplicity® Synplify Pro® instead of Xilinx Synthesis Technology (XST) to synthesize the design.

For more information on Simulink HDL Coder, refer to the product documentation: *www.mathworks.com/products/slhdlcoder/*

Required Software

The example model and related scripts are distributed with this guide in a .zip file, `hdlcoderrecon_xsg_files_8a.zip`. Simulation and code generation from the model have been tested with the following versions of the software:

- MATLAB 7.6 (R2008a)
- Simulink 7.1
- Simulink HDL Coder 1.3 (requires Fixed-Point Toolbox™ 2.2 and Simulink Fixed Point™ 5.6)
- Signal Processing Blockset™ 6.7 (requires Signal Processing Toolbox™ 6.9)
- Image Processing Toolbox™ 6.1
- Xilinx® System Generator for DSP Version 10.1.3 (note: 10.1 Service Pack 3 or later is required for the integration)

To simulate, synthesize, and implement HDL code generated from the model, the following software is also required:

- Mentor Graphics ModelSim® SE 6.3c
- Xilinx® ISE Foundation™ Version 10.1 with Service Pack 3

The synthesis workflow described in Appendix B has been tested with the following version of software:

- Synplicity® Synplify Pro® 8.8.0.4

About the Example Model

Our example model is based on the "Image Reconstruction Using Cosimulation" demo in Simulink HDL Coder. The model performs image reconstruction from parallel-beam projection data using the Filtered Back-Projection algorithm.

The top level of the example model is shown in Figure 1. This guide focuses on the image reconstruction subsystem shown in Figure 2. This subsystem is the device under test (DUT), from which HDL code is to be generated. In the DUT, the counters and back projection subsystems are implemented using Simulink blocks, while the **`filtering_xil`** subsystem is implemented using Xilinx System Generator blocks.

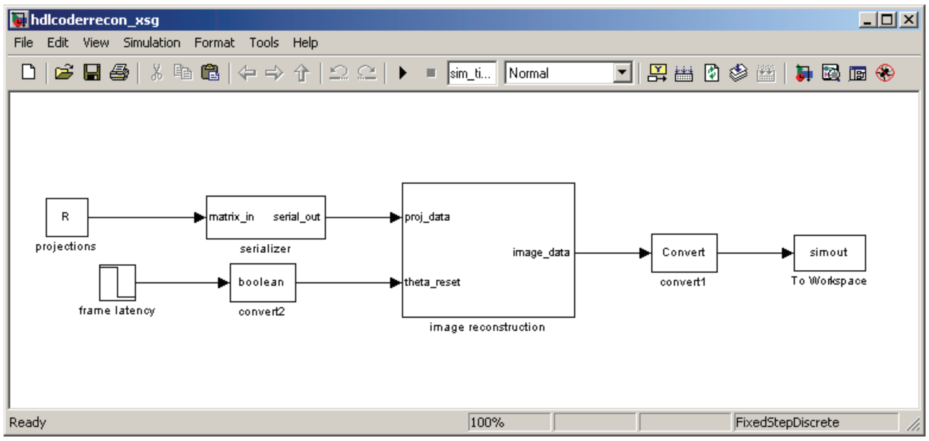


Figure 1. The top level of the example model `hdlcoderrecon_xsg`.

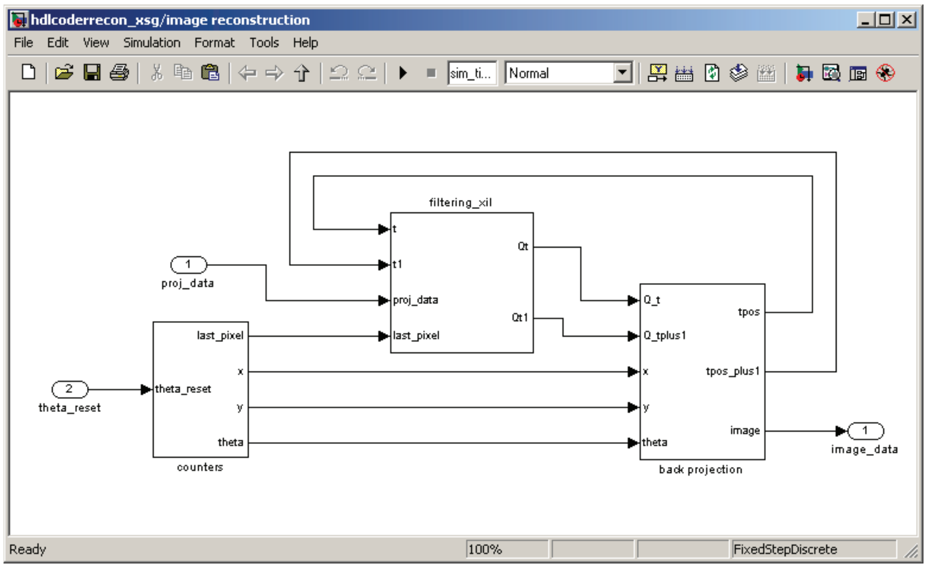


Figure 2. Image reconstruction subsystem.

The **filtering_xil** subsystem is shown in Figure 3. When Simulink HDL Coder generates code for the DUT, it implements the **filtering_xil** subsystem in HDL as a black box. Instead of generating code for the content of the **filtering_xil** subsystem, Simulink HDL Coder creates a black box interface — a VHDL component or Verilog module that includes only the HDL input/output port definitions for the subsystem.

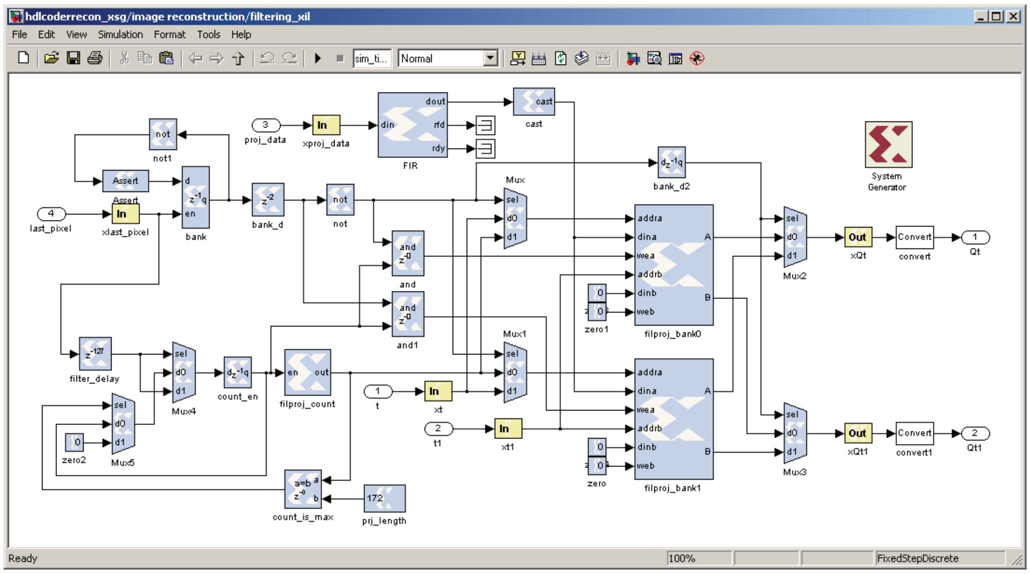


Figure 3. Filtering subsystem implemented using Xilinx System Generator.

Configuring a Model for HDL Code Generation

In a typical development workflow, engineers model and simulate a design in Simulink, using multiple iterations to identify and eliminate design problems in preparation for implementation. In the example used in this guide, the model hdlcoderrecon_xsg.mdl is ready to be implemented. Before using Simulink HDL Coder and Xilinx System Generator to generate code, however, you must prepare the model by:

- Matching port data types
- Setting code generation options for Xilinx System Generator
- Setting up a code generation control file for Simulink HDL Coder

Matching Port Data Types

To ensure the compatibility of data types between HDL code generated by Xilinx System Generator and the interface generated by Simulink HDL Coder, use the following guidelines:

- Fixed-point settings in Xilinx Gateway In blocks are independent of the Simulink fixed-point data type. When you route signals to and from the Xilinx subsystem through the Gateway blocks, you must verify manually that the data types of Simulink and Xilinx fixed-point signals are consistent. For example, in Figure 4, the data type of the Gateway In block, **xproj_data**, has been set to **Fix_17_11** to match the Simulink fixed-point type **sfix17_En11** (a 17-bit signed fixed-point number with 11 fractional bits) of the Inport block, **proj_data**.

- Because the output of a Gateway Out block is of type **`double`**, you must insert a Data Type Conversion block after the Gateway Out block to convert the data type back to Simulink fixed point. Place the Data Type Conversion block inside the black box subsystem to enable Simulink HDL Coder to generate a correct output interface for the subsystem. Figure 4 shows the correct location for a Data Type Conversion block.

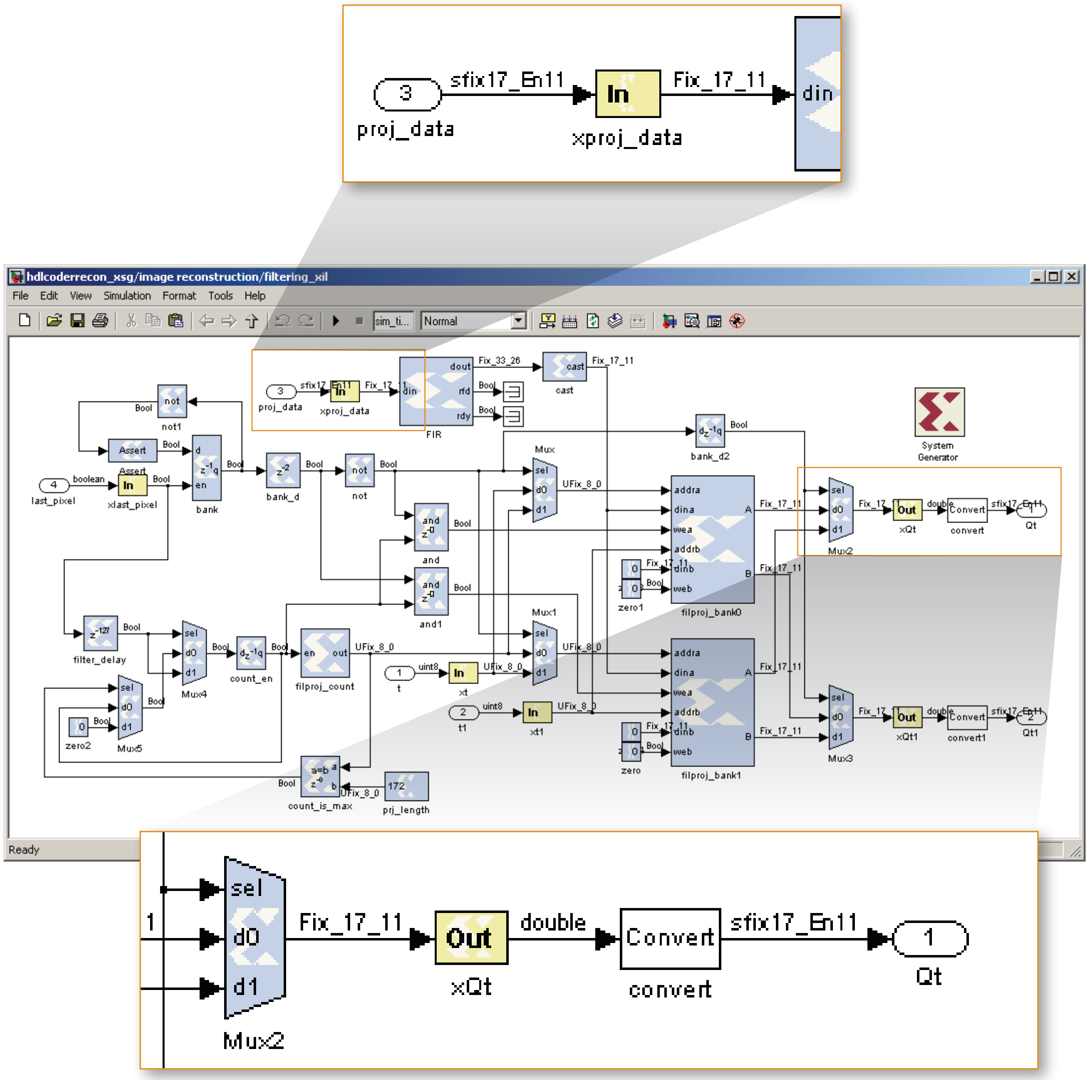
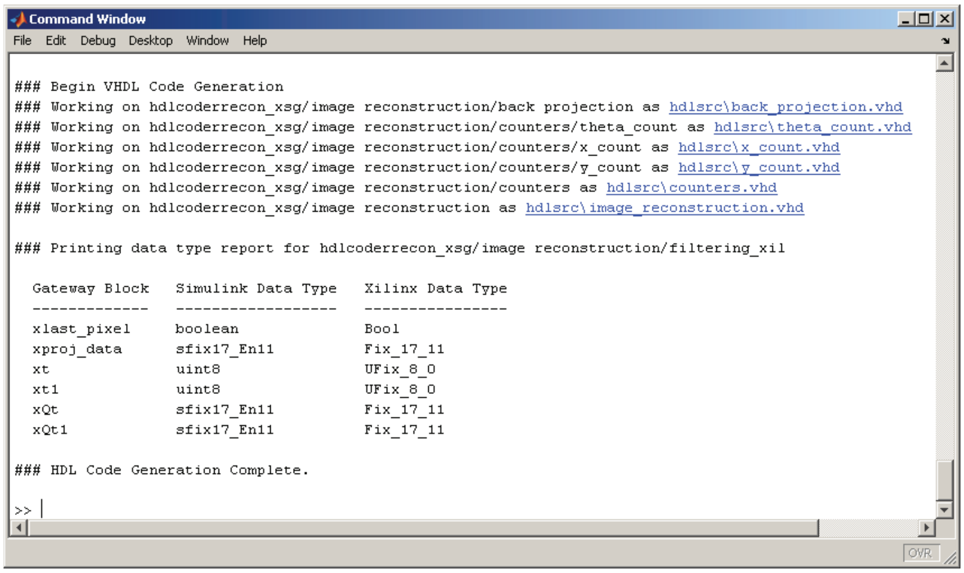


Figure 4. Matching Simulink and Xilinx fixed-point data types.

To help you verify that the Simulink and Xilinx data types are consistent across each Gateway block of the Xilinx subsystem, a data type report is printed in the command window during code generation. Figure 5 shows the report for the example model.

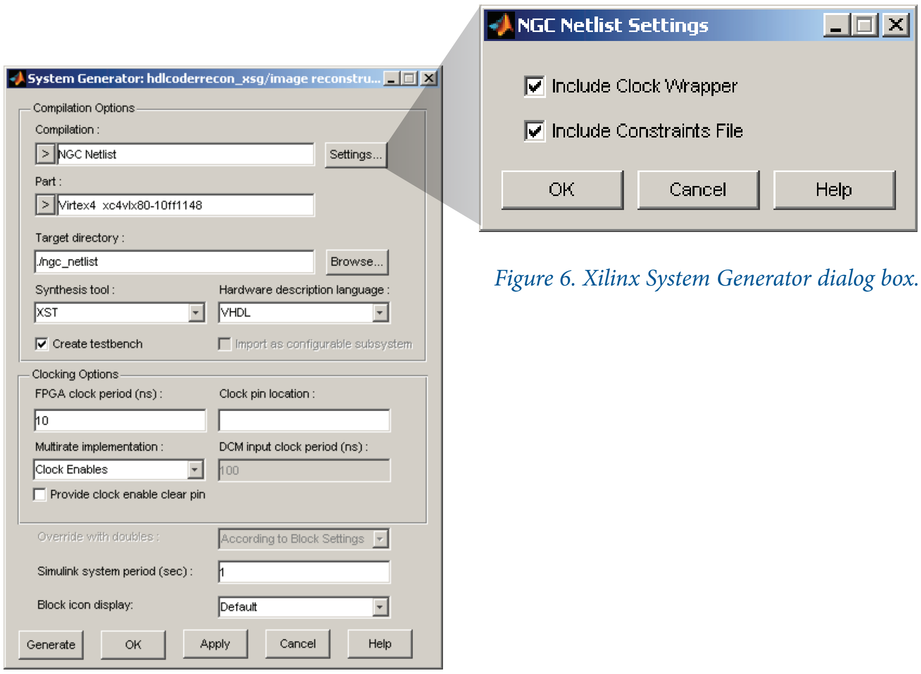


```
### Begin VHDL Code Generation
### Working on hdlcoderrecon_xsg/image reconstruction/back projection as hdlsrc\back_projection.vhd
### Working on hdlcoderrecon_xsg/image reconstruction/counters/theta_count as hdlsrc\theta_count.vhd
### Working on hdlcoderrecon_xsg/image reconstruction/counters/x_count as hdlsrc\x_count.vhd
### Working on hdlcoderrecon_xsg/image reconstruction/counters/y_count as hdlsrc\y_count.vhd
### Working on hdlcoderrecon_xsg/image reconstruction/counters as hdlsrc\counters.vhd
### Working on hdlcoderrecon_xsg/image reconstruction as hdlsrc\image_reconstruction.vhd

### Printing data type report for hdlcoderrecon_xsg/image reconstruction/filtering_xil

  Gateway Block    Simulink Data Type    Xilinx Data Type
  -------------    ------------------    ----------------
  xlast_pixel      boolean               Bool
  xproj_data       sfix17_En11           Fix_17_11
  xt               uint8                 UFix_8_0
  xt1              uint8                 UFix_8_0
  xQt              sfix17_En11           Fix_17_11
  xQt1             sfix17_En11           Fix_17_11

### HDL Code Generation Complete.

>> |
```

Figure 5. Data type report provided by Simulink HDL Coder during code generation.

Setting Code Generation Options for Xilinx System Generator

The System Generator block provides control of simulation and code generation parameters. Figure 6 shows the block dialog box in the example model.



Compilation : NGC Netlist
Part : Virtex4 xc4vlx80-10ff1148
Target directory : ./ngc_netlist
Synthesis tool : XST
Hardware description language : VHDL
[✓] Create testbench
[ ] Import as configurable subsystem
FPGA clock period (ns) : 10
Clock pin location :
Multirate implementation : Clock Enables
DCM input clock period (ns) : 100
[ ] Provide clock enable clear pin
Override with doubles : According to Block Settings
Simulink system period (sec) : 1
Block icon display: Default

[✓] Include Clock Wrapper
[✓] Include Constraints File

Figure 6. Xilinx System Generator dialog box.

Follow the settings described below to ensure proper configuration for the integration workflow:

1. Set **Compilation** to NGC Netlist. This setting creates a standalone Xilinx NGC binary netlist file, which contains all the generated HDL code, cores, and constraint file information.

   Use the default values (set to on) for the **Include Clock Wrapper** and **Include Constraints File** options in the Settings subdialog.

2. Set **Multirate implementation** in **Clocking Options** to Clock Enables. Currently only single rate models are supported for the integration workflow.

3. Set **Simulink system period** to match the value of the sample period of your Simulink model.

4. Select the **Create testbench** option (as shown) to generate test bench code for the Xilinx subsystem. See the "HDL Simulation and Verification" section of this guide for further information.

In addition, the settings in **Part, Synthesis tool,** and **Hardware description language** should match the corresponding Simulink HDL Coder property settings. A helper function is available in Simulink HDL Coder to help you match the settings. See "Setting Code Generation Properties Using Helper Function **`hdlsettingsxsg.m`**" in the next section for more details.

Setting Up a Code Generation Control File for Simulink HDL Coder

Code generation control files (referred to in this guide as control files) enable you to extend the HDL code generation process and direct its operation. In this case, you use the control file to make the code generator execute a specific block implementation method when generating HDL code for a selected block.

Specifying Xilinx Black Box Implementation

To incorporate Xilinx System Generator blocks in a Simulink HDL Coder design, you must encapsulate all Xilinx System Generator blocks in a subsystem, and specify the black box implementation **`XilinxBlackBoxHDLInstantiation`** for the subsystem in a control file. The implementation instructs Simulink HDL Coder software to create only an HDL interface to a subsystem, without generating any code for the contents of the subsystem. To match the HDL entity definition generated by Xilinx System Generator, Simulink HDL Coder does the following:

1. Automatically matches the HDL I/O port names and component name

   When generating a black box interface, the names of the Gateway blocks are used to create the HDL I/O port names. To match the clock wrapper entity created from NGC netlist compilation, a postfix "_cw" is added to the Xilinx System Generator HDL component name.

2. Automatically inserts black box attributes in VHDL

   Using NGC netlist, the subsystem implemented in Xilinx System Generator is presynthesized during code generation. Therefore the subsystem should be treated as a black box during synthesis. To specify that the Xilinx HDL component is a black box, VHDL attributes for XST and Synplify are inserted in the generated code.

3. Provides implementation parameters to match the HDL global ports, and turn off VHDL configuration statement

The Xilinx black box implementation provides a number of implementation parameters for customizing the generated HDL interface. You can add or remove global port(s), customize the HDL entity name and global port names, turn on or off VHDL configuration statement, etc. You can use the function **`hdlnewforeach`** to see all the available parameters.

To match the global ports of the Xilinx System Generator HDL entity, set the implementation parameters in Table 1 to the values shown.

Implementation Parameter	Default Value	Value to Match Xilinx System Generator HDL Entity
AddClockPort	on	on
ClockInputPort	clk	clk
AddClockEnablePort	on	on
ClockEnableInputPort	clk_enable	**ce**
AddResetPort	on	**off**
ResetInputPort	reset	Leave unspecified
InlineConfigurations	on	**off**

Table 1. Parameter settings for the XilinxBlackBoxHDLInstantiation implementation.

With the example model, a control file, **`hdlcoderrecon_xsg_control.m`**, is provided to select a Xilinx black box implementation for the **`filtering_xil`** subsystem, as illustrated in the code excerpt in Figure 7.

```
c.forEach('hdlcoderrecon_xsg/image reconstruction/filtering_xil', ...
           'built-in/SubSystem', {}, ...
           'hdldefaults.XilinxBlackBoxHDLInstantiation', ...
           {'AddClockPort', 'on', 'ClockInputPort', 'clk', ...
            'AddClockEnablePort', 'on', 'ClockEnableInputPort', 'ce', ...
            'AddResetPort', 'off', ...
            'InlineConfigurations', 'off'});
```

Figure 7. Specifying Xilinx black box implementation for the **`filtering_xil`** *subsystem.*

Setting Code Generation Properties Using Helper Function hdlsettingsxsg.m

After specifying the Xilinx black box implementation, you need to specify code generation properties in the control file. The values of these properties should be consistent with the settings in the Xilinx System Generator dialog box, and other settings in the generated HDL from System Generator. To help you select correct code generation settings, Simulink HDL Coder provides a helper function called **`hdlsettingsxsg.m`**. The function takes the path to the Xilinx System Generator block as input argument, programmatically obtains parameter settings in the block, and then specifies Simulink HDL Coder code generation properties that are compatible with System Generator. The function helps you do the following:

1. Match the target language

   The property **`TargetLanguage`** is set to match the **Hardware description language** setting in the Xilinx System Generator dialog box.

2. Specify synthesis script settings for the target synthesis tool and device

   The properties **`HDLSynthFilePostfix`**, **`HDLSynthInit`**, **`HDLSynthCmd`** and **`HDLSynthTerm`** are set according to the **Synthesis tool** setting in the Xilinx System Generator dialog box, so that the generated synthesis script can be used in the specified synthesis tool. The device specified in **Part** is incorporated to the **`HDLSynthTerm`** property setting.

3. Match hardware clock period

   When necessary, the test bench properties **`ClockHighTime`** and **`ClockLowTime`** are set according to the **FPGA clock period** setting in the Xilinx System Generator dialog box. Note that since the value of **`ClockHighTime`** and **`ClockLowTime`** must be integer, they are only set when **FPGA clock period** is an even integer. Otherwise, a warning is displayed.

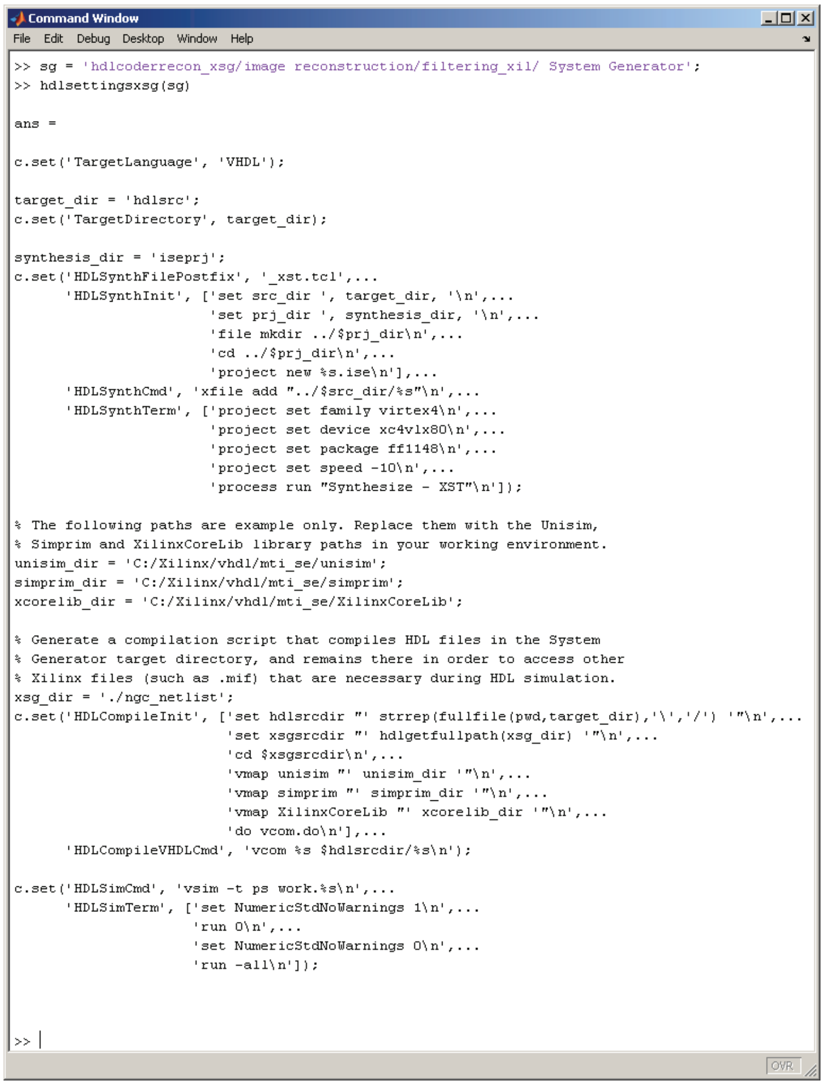
4. Specify test bench compilation and simulation script settings

   The properties **`HDLCompileInit`** and **`HDLCompileVHDLCmd`** / **`HDLCompileVerilogCmd`** are set to create a test bench compilation script that maps to Xilinx simulation libraries needed for HDL simulation, and to compile HDL files generated by both Xilinx System Generator and Simulink HDL Coder.

   In addition, the simulation script generated by Xilinx System Generator contains certain settings required for HDL simulation. The properties **`HDLSimCmd`** and **`HDLSimTerm`** are set to incorporate these settings to the simulation script generated by Simulink HDL Coder.

   See the “HDL Simulation and Verification” section of this guide for further information on these property settings.

After running the function **`hdlsettingsxsg.m`**, the code generation properties are printed to the MATLAB command window, as shown in Figure 8. You can then copy and paste the property settings to your control file, and attach the control file to your Simulink model.



```
>> sg = 'hdlcoderrecon_xsg/image reconstruction/filtering_xil/ System Generator';
>> hdlsettingsxsg(sg)

ans =

c.set('TargetLanguage', 'VHDL');

target_dir = 'hdlsrc';
c.set('TargetDirectory', target_dir);

synthesis_dir = 'iseprj';
c.set('HDLSynthFilePostfix', '_xst.tcl',...
      'HDLSynthInit', ['set src_dir ', target_dir, '\n',...
                       'set prj_dir ', synthesis_dir, '\n',...
                       'file mkdir ../$prj_dir\n',...
                       'cd ../$prj_dir\n',...
                       'project new %s.ise\n'],...
      'HDLSynthCmd', 'xfile add "../$src_dir/%s"\n',...
      'HDLSynthTerm', ['project set family virtex4\n',...
                       'project set device xc4vlx80\n',...
                       'project set package ff1148\n',...
                       'project set speed -10\n',...
                       'process run "Synthesize - XST"\n']);

% The following paths are example only. Replace them with the Unisim,
% Simprim and XilinxCoreLib library paths in your working environment.
unisim_dir = 'C:/Xilinx/vhdl/mti_se/unisim';
simprim_dir = 'C:/Xilinx/vhdl/mti_se/simprim';
xcorelib_dir = 'C:/Xilinx/vhdl/mti_se/XilinxCoreLib';

% Generate a compilation script that compiles HDL files in the System
% Generator target directory, and remains there in order to access other
% Xilinx files (such as .mif) that are necessary during HDL simulation.
xsg_dir = './ngc_netlist';
c.set('HDLCompileInit', ['set hdlsrcdir "' strrep(fullfile(pwd,target_dir),'\','/') '"\n',...
                         'set xsgsrcdir "' hdlgetfullpath(xsg_dir) '"\n',...
                         'cd $xsgsrcdir\n',...
                         'vmap unisim "' unisim_dir '"\n',...
                         'vmap simprim "' simprim_dir '"\n',...
                         'vmap XilinxCoreLib "' xcorelib_dir '"\n',...
                         'do vcom.do\n'],...
      'HDLCompileVHDLCmd', 'vcom %s $hdlsrcdir/%s\n');

c.set('HDLSimCmd', 'vsim -t ps work.%s\n',...
      'HDLSimTerm', ['set NumericStdNoWarnings 1\n',...
                     'run 0\n',...
                     'set NumericStdNoWarnings 0\n',...
                     'run -all\n']);



>> |
```

Figure 8. Output of helper function **`hdlsettingsxsg.m`** *from the example model.*

Generating HDL Code

After you have matched port data types in your model, set code generation options for Xilinx System Generator, and set up a code generation control file for Simulink HDL Coder, you are ready to generate HDL code from the model. To begin code generation, type **`makehdl`** in the MATLAB command window or use the Simulink Configuration Parameters GUI.

Simulink HDL Coder will not generate code for the Xilinx subsystem **`filtering_xil`**. Use the Xilinx System Generator dialog box to generate this code separately with Xilinx System Generator.

Note: If you have modified settings in the Xilinx System Generator block, you may need to re-run **`hdlsettingsxsg.m`** to create the latest code generation properties for Simulink HDL Coder, and apply those properties to your control file again.

Synthesis and Implementation

You can automatically synthesize generated HDL code from Simulink HDL Coder for Xilinx FPGAs using several synthesis tools, including Xilinx Synthesis Technology (XST) and Synplicity Synplify or Synplify Pro. This section describes synthesis of the design using XST. For details on using Synplify Pro for synthesis, see Appendix B.

When the **Synthesis tool** option in the Xilinx System Generator block is set to XST, the helper function `hdlsettingsxsg.m` sets the properties `HDLSynthFilePostfix`, `HDLSynthInit`, `HDLSynthCmd`, and `HDLSynthTerm` to create a Tcl script for use with XST. The generated Tcl script does the following:

1. Creates an ISE project in a new directory

2. Adds HDL files generated by Simulink HDL Coder to the ISE project

3. Sets target device according to settings in the Xilinx System Generator block

4. Runs XST synthesis

```
set src_dir hdlsrc
set prj_dir iseprj
file mkdir ../$prj_dir
cd ../$prj_dir
project new image_reconstruction.ise
xfile add "../$src_dir/back_projection.vhd"
xfile add "../$src_dir/theta_count.vhd"
xfile add "../$src_dir/x_count.vhd"
xfile add "../$src_dir/y_count.vhd"
xfile add "../$src_dir/counters.vhd"
xfile add "../$src_dir/image_reconstruction.vhd"
project set family virtex4
project set device xc4vlx80
project set package ff1148
project set speed -10
process run "Synthesize - XST"
```

Figure 9. `image_reconstruction_xst.tcl` *– synthesis script of the example model.*

Figure 9 shows the generated synthesis script `image_reconstruction_xst.tcl` with the example model.

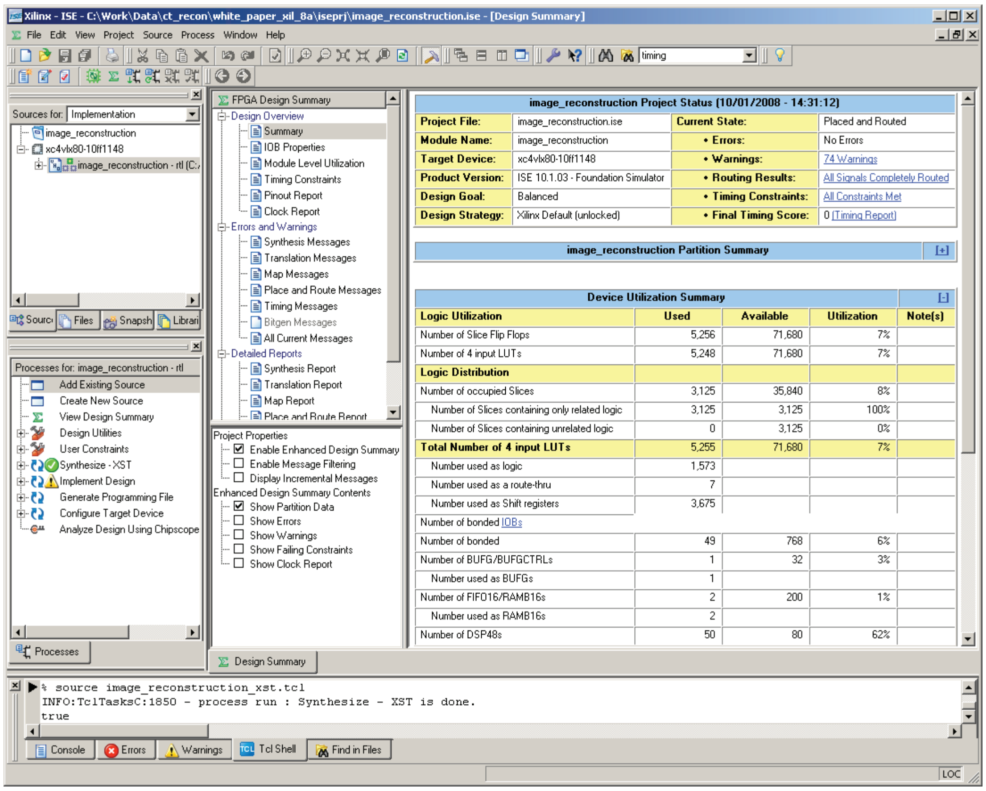
To synthesize the design, run the generated synthesis script in the Xilinx Tcl Shell (`xtclsh`), either from ISE Project Navigator or from the command line. For more details on the Xilinx Tcl Shell, refer to the Xilinx ISE documentation. You can also create a new project in ISE Project Navigator using the New Project Wizard, and synthesize the design manually.

Note that the resources reported by ISE after synthesis do not include the Xilinx component (`filtering_xil_cw`), because the component is synthesized as a black box.

To implement the design after synthesis, the netlist for the black box component **`filtering_xil_cw`** must be provided to ISE. You can either set the Translate property **Macro Search Path** to the location of the NGC netlist **`filtering_xil_cw.ngc`** generated by Xilinx System Generator, or copy the file to your ISE project directory.

When the design is ready to be implemented, run Implement Design to execute Translate, Map, and Place & Route.

The Xilinx ISE design summary shown in Figure 10 shows the logic utilization and performance of the example design after implementation.



Xilinx - ISE - C:\Work\Data\ct_recon\white_paper_xil_8a\iseprj\image_reconstruction.ise - [Design Summary]

image_reconstruction Project Status (10/01/2008 - 14:31:12)

Project File:	image_reconstruction.ise	**Current State:**	Placed and Routed
Module Name:	image_reconstruction	**• Errors:**	No Errors
Target Device:	xc4vlx80-10ff1148	**• Warnings:**	74 Warnings
Product Version:	ISE 10.1.03 - Foundation Simulator	**• Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	**• Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	**• Final Timing Score:**	0 (Timing Report)

image_reconstruction Partition Summary [+]

Device Utilization Summary [-]

Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	5,256	71,680	7%	
Number of 4 input LUTs	5,248	71,680	7%	
Logic Distribution				
Number of occupied Slices	3,125	35,840	8%	
Number of Slices containing only related logic	3,125	3,125	100%	
Number of Slices containing unrelated logic	0	3,125	0%	
Total Number of 4 input LUTs	5,255	71,680	7%	
Number used as logic	1,573			
Number used as a route-thru	7			
Number used as Shift registers	3,675			
Number of bonded IOBs				
Number of bonded	49	768	6%	
Number of BUFG/BUFGCTRLs	1	32	3%	
Number used as BUFGs	1			
Number of FIFO16/RAMB16s	2	200	1%	
Number used as RAMB16s	2			
Number of DSP48s	50	80	62%	

```
% source image_reconstruction_xst.tcl
INFO:TclTasksC:1850 - process run : Synthesize - XST is done.
true
```

Figure 10. The Xilinx ISE design summary window.

HDL Simulation and Verification

Before committing your design to hardware, it is a good practice to verify the HDL code. Simulink HDL Coder also generates HDL test benches that help you verify the generated HDL code using HDL simulation tools.

Generating HDL Test Bench and Simulation Scripts

Specifying Compilation and Simulation Script Settings Using hdlsettingsxsg.m

Simulink HDL Coder generates a test bench compilation script to compile the design for HDL simulation. The helper function **`hdlsettingsxsg.m`** adds the necessary settings to the compilation script by specifying the properties **`HDLCompileInit`** and **`HDLCompileVHDLCmd`** / **`HDLCompileVerilogCmd`**. The generated compilation script does the following:

1. Maps Xilinx simulation libraries

   To compile and simulate the portion of the design implemented with System Generator, the locations of the Xilinx libraries **`Unisim`**, **`Simprims`**, and **`XilinxCoreLib`** must be set in ModelSim via the **`vmap`** command. In the compilation script, these libraries are mapped to the default paths – for example, **`C:/Xilinx/vhdl/mti_se/unisim`** for the **`Unisim`** library in VHDL.

   To match the library paths to those in your working environment, copy the settings generated by **`hdlsettingsxsg.m`** to your control file, and modify the paths among those settings before generating code from your design.

2. Compiles HDL files generated by Xilinx System Generator

   The compilation script **`vcom.do`** generated by Xilinx System Generator is used to compile the Xilinx HDL files. Before **`vcom.do`** is run, the current directory is changed to the System Generator target directory, in order to access other Xilinx files (such as .mif and .coe) during simulation.

   The Simulink HDL Coder target directory is saved in the Tcl variable **`$hdlsrcdir`**.

3. Compiles HDL files generated by Simulink HDL Coder

   Lastly, the HDL files generated by Simulink HDL Coder are compiled. Because the current directory is the System Generator target directory, the path to the HDL files is added to the compile command, such as:

   ```
   vcom $hdlsrcdir/image_reconstruction.vhd
   ```

Figure 11 shows the generated test bench compilation script of the example design.

```
set hdlsrcdir "C:/Data/image_reconstruction/hdlsrc"
set xsgsrcdir "C:/Data/image_reconstruction/ngc_netlist"
cd $xsgsrcdir
vmap unisim "C:/Xilinx/vhdl/mti_se/unisim"
vmap simprim "C:/Xilinx/vhdl/mti_se/simprim"
vmap XilinxCoreLib "C:/Xilinx/vhdl/mti_se/XilinxCoreLib"
do vcom.do
vcom  $hdlsrcdir/back_projection.vhd
vcom  $hdlsrcdir/theta_count.vhd
vcom  $hdlsrcdir/x_count.vhd
vcom  $hdlsrcdir/y_count.vhd
vcom  $hdlsrcdir/counters.vhd
vcom  $hdlsrcdir/image_reconstruction.vhd
vcom  $hdlsrcdir/image_reconstruction_tb.vhd
```

Figure 11. `image_reconstruction_tb_compile.do.`

In addition to the compilation script, **`hdlsettingsxsg.m`** also incorporates settings necessary to simulate the Xilinx HDL component in the generated simulation script. The properties **`HDLSimCmd`** and **`HDLSimTerm`** are set to include the settings in **`vsim.do`**, the simulation script generated by Xilinx System Generator.

Figure 12 shows the generated test bench simulation script of the example design. The settings incorporated from **vsim.do** are highlighted.

```
onbreak resume
onerror resume
vsim -novopt -t ps work.image_reconstruction_tb
add wave sim:/image_reconstruction_tb/u_image_reconstruction/clk
add wave sim:/image_reconstruction_tb/u_image_reconstruction/reset
add wave sim:/image_reconstruction_tb/u_image_reconstruction/clk_enable
add wave sim:/image_reconstruction_tb/u_image_reconstruction/proj_data
add wave sim:/image_reconstruction_tb/u_image_reconstruction/theta_reset
add wave sim:/image_reconstruction_tb/u_image_reconstruction/ce_out
add wave sim:/image_reconstruction_tb/u_image_reconstruction/image_data
add wave sim:/image_reconstruction_tb/image_data_ref
set NumericStdNoWarnings 1
run 0
set NumericStdNoWarnings 0
run -all
```

Figure 12. **image_reconstruction_tb_sim.do**.

Additional Code Generation Property Settings for HDL Simulation

The function **hdlsettingsxsg.m** helps you setup many code generation properties that are necessary to simulate the HDL files generated by Simulink HDL Coder and Xilinx System Generator. In some cases, you may need to set additional properties to simulate the design successfully.

It is recommended to use the **-novopt** option with the ModelSim command **vsim** for HDL simulation. You can add the option to the **HDLSimCmd** setting generated by **hdlsettingsxsg.m**, as shown in Figure 13.

```
c.set('HDLSimCmd', 'vsim -novopt -t ps work.%s\n',...
      'HDLSimTerm', ['set NumericStdNoWarnings 1\n',...
                     'run 0\n',...
                     'set NumericStdNoWarnings 0\n',...
                     'run -all\n']);
```

Figure 13. Adding **-novopt** *flag to the simulation script property* **HDLSimCmd.**

By default, test bench inputs generated by Simulink HDL Coder are driven with the value 'X' before reset. If test bench inputs are directly connected to registers in the Xilinx subsystem, simulation mismatch may occur at the beginning of HDL simulation, until the initial undefined values are propagated out of the registers. Setting **InitializeTestBenchInputs** to **on** can prevent this temporary mismatch.

When `ResetLength` and `TestBenchClockEnableDelay` are set to `0`, HDL simulation begins at the first available clock edge. This matches the behavior of the HDL test bench generated by System Generator, and is important if the Xilinx subsystem of your Simulink model contains blocks that change states without input, such as counter, accumulator, etc.

Table 2 summarizes the additional property settings you may need for your design.

Code Generation Property	Default Value	Value to Match Xilinx System Generator HDL Simulation
HDLSimCmd	vsim work.%s\n	Add `-novopt` to the value generated by `hdlsettingsxsg.m`
InitializeTestBenchInputs	off	on
ResetLength	2	0
TestBenchClockEnableDelay	1	0

Table 2. Additional code generation properties settings for HDL simulation.

The example model requires a long simulation time, resulting in a large test bench file. To reduce the size of the test bench file, consider setting the total simulation time to a smaller value before generating HDL and test benches. In this example, a simulation time of 50,000 clock cycles (500 microseconds) is used for a good balance between run time and HDL functionality verification.

To generate a test bench from a Simulink model, type `makehdltb` in the MATLAB command window or use the Simulink Configuration Parameters GUI. If you modify the simulation time, you must run `makehdl` again (before running `makehdltb`) for the new simulation time to take effect in the test bench.

Generate the Xilinx subsystem test bench, simulation files and scripts separately using Xilinx System Generator. Select the **Create testbench** option, as shown previously in Figure 6.

Simulating the HDL Design in ModelSim

If you are using ModelSim SE or PE, you must compile the Xilinx simulation libraries using the `COMPXLIB` utility provided by Xilinx, either from Project Navigator or from the command line. For more details on how to use `COMPXLIB`, refer to the Xilinx ISE documentation.

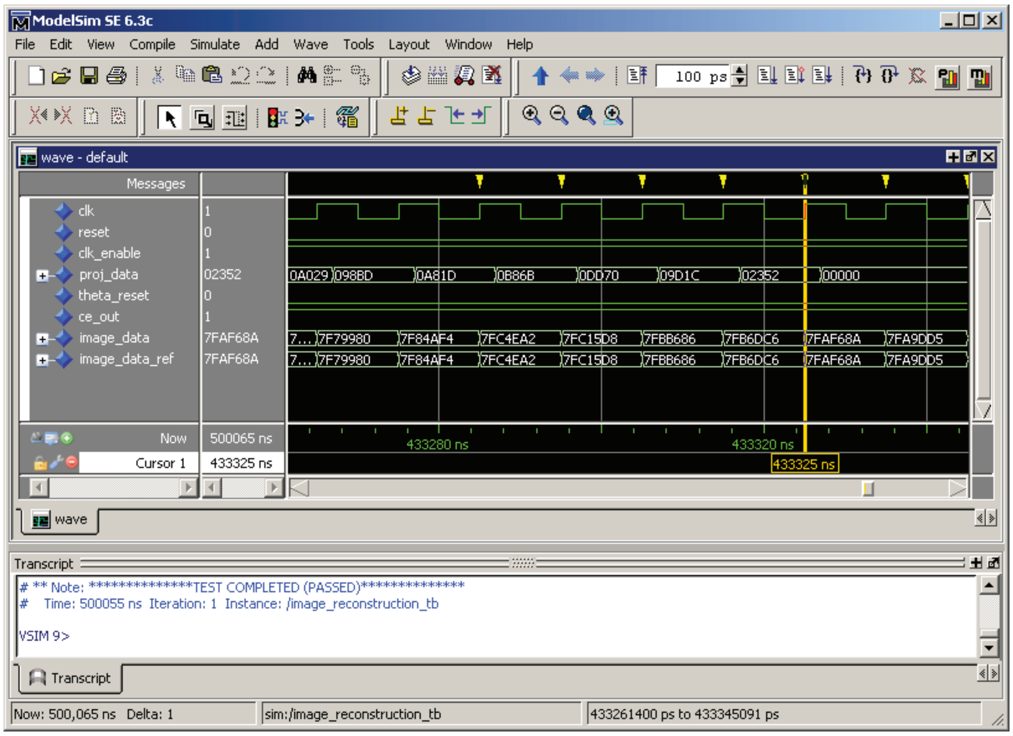
To compile the design for simulation, run the test bench compilation script in the ModelSim Transcript window:

```
do image_reconstruction_tb_compile.do
```

To simulate the design, run the test bench simulation script. Since the current directory has been changed to the Xilinx System Generator target directory during compilation, the path of the simulation script must be included when the script is run:

```
do $hdlsrcdir/image_reconstruction_tb_sim.do
```

Figure 14 shows the example model simulation results from ModelSim SE.



```
# ** Note: **************TEST COMPLETED (PASSED)**************
#    Time: 500055 ns  Iteration: 1  Instance: /image_reconstruction_tb

VSIM 9>
```

```
Now: 500,065 ns  Delta: 1    sim:/image_reconstruction_tb    433261400 ps to 433345091 ps
```

Figure 14. ModelSim simulation window.

For design efforts that can benefit from a combination of Simulink and Xilinx capabilities, the integration of Simulink HDL Coder and Xilinx System Generator enables an effective workflow that spans HDL code generation, synthesis and implementation, and simulation and verification.

Appendix A. Using Verilog in the Integration Workflow

The example in this guide focuses on a VHDL-centered workflow for integrating Xilinx System Generator with Simulink HDL Coder code generation software. You can use a similar workflow based on Verilog. The following sections describe the differences of a Verilog-based workflow.

A control file for code generation in Verilog (`hdlcoderrecon_xsg_control_verilog.m`) is included with the example files.

Generating and Editing HDL Code

For VHDL, black box attributes identify the Xilinx subsystem as a black box entity to the synthesis tool. For Verilog, if a module has an empty declaration, both XST and Synplify Pro automatically treat it as a black box during synthesis. Simulink HDL Coder, however, does not generate a module declaration for the Xilinx subsystem because the subsystem is implemented as a black box. Therefore you must complete the following steps manually:

1. Create an empty module declaration of the Xilinx subsystem in a wrapper file.

2. Add the wrapper file to the synthesis script.

The easiest way to create the empty module declaration is to copy the module declaration generated by Xilinx System Generator and then remove from it everything but the port definitions. In this example, an empty module declaration of the Xilinx subsystem `filtering_xil_cw` is needed. Figure 15 shows the contents of the wrapper file `filtering_xil_cw.v.`

```
module filtering_xil_cw (
  clk,
  ce,
  xlast_pixel,
  xproj_data,
  xqt,
  xqt1,
  xt,
  xt1
);

  input  clk;
  input  ce;
  input [0:0] xlast_pixel;
  input [16:0] xproj_data;
  input [7:0] xt;
  input [7:0] xt1;
  output [16:0] xqt;
  output [16:0] xqt1;

endmodule
```

Figure 15. Empty module declaration of `filtering_xil_cw` *in wrapper file* `filtering_xil_cw.v.`

After creating the wrapper file, you can add it to the synthesis script automatically using the control file property **`HDLSynthInit`**. Append the appropriate command to the **`HDLSynthInit`** setting generated by the helper function **`hdlsettingsxsg.m`**, as shown in Figure 16. Alternatively, you can manually add the wrapper file to the generated synthesis script.

Note that if the interface of the Xilinx subsystem in your Simulink model is changed, such as the name of the subsystem or I/O port definition, you must modify the **`HDLSynthInit`** setting and the wrapper file accordingly, and re-run code generation from Simulink HDL Coder and Xilinx System Generator if necessary.

```
synthesis_dir = 'iseprj';
c.set('HDLSynthFilePostfix', '_xst.tcl',...
       'HDLSynthInit', ['set src_dir ', target_dir, '\n',...
                        'set prj_dir ', synthesis_dir, '\n',...
                        'file mkdir ../$prj_dir\n',...
                        'cd ../$prj_dir\n',...
                        'project new %s.ise\n',...
                        'xfile add "../$src_dir/filtering_xil_cw.v"\n'],...
       'HDLSynthCmd', 'xfile add "../$src_dir/%s"\n',...
       'HDLSynthTerm', ['project set family virtex4\n',...
                        'project set device xc4vlx80\n',...
                        'project set package ff1148\n',...
                        'project set speed -10\n',...
                        'process run "Synthesize - XST"\n']);
```

Figure 16. Adding **`filtering_xil_cw.v`** *to the* synthesis **`.tcl`** *script using control file property* **`HDLSynthInit`**.

HDL Simulation and Verification

As with VHDL, it is recommended to use the **`-novopt`** option with the ModelSim command **`vsim`** for HDL simulation. You can add the option to the **`HDLSimCmd`** setting generated by **`hdlsettingsxsg.m`**, as shown in Figure 17.

```
c.set('HDLSimCmd', ['vsim -novopt +nowarnTFMPC -t ps work.%s ',...
                    '-L UNISIMS_VER -L SIMPRIMS_VER ',...
                    '-L XILINXCORELIB_VER work.glbl work.clock_pkg\n'],...
       'HDLSimTerm', 'run -all\n');
```

Figure 17. Adding **`-novopt`** *flag to the simulation script property* **`HDLSimCmd`** *for Verilog.*

Appendix B: Synthesizing the Design Using Synplify Pro

When the **Synthesis tool** option in the Xilinx System Generator block is set to Synplify Pro, the helper function **`hdlsettingsxsg.m`** sets the properties **`HDLSynthFilePostfix`**, **`HDLSynthInit`**, **`HDLSynthCmd`**, and **`HDLSynthTerm`** to create a Tcl script for use with Synplify Pro. The generated Tcl script does the following:

1. Creates a new project, and adds HDL files generated by Simulink HDL Coder to the project

2. Sets target device according to settings in the Xilinx System Generator block

3. Sets synthesis options, such as auto frequency, and turn off constraint file generation

4. Runs synthesis

The synthesis option **`write_apr_constraint`** is set to 0 in the synthesis Tcl script. When the option is set to 1, Synplify Pro creates a Xilinx **`.ncf`** constraint file together with the **`.edf`** output netlist. When the **`.edf`** netlist is added to an ISE project, ISE detects the **`.ncf`** file and automatically extracts the constraint information contained in it.

In some cases, the extracted constraints may conflict with the timing constraints embedded in the NGC netlist generated by System Generator. Setting **`write_apr_constraint`** to 0 prevents Synplify Pro from writing the **`.ncf`** constraint file, and avoids any conflict. During implementation, the clock period constraint embedded in the NGC netlist will be used by ISE to analyze sequential paths in the entire design.

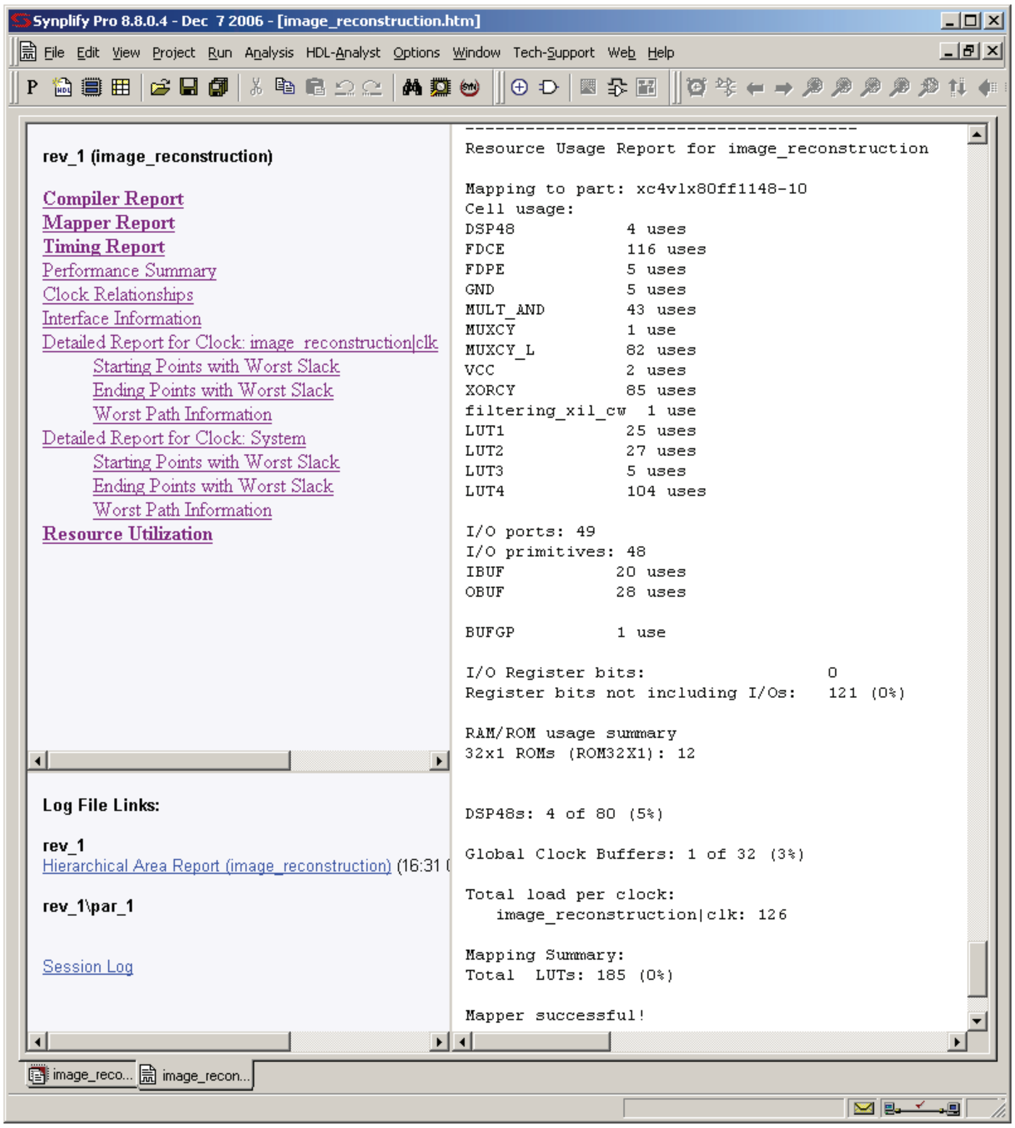
Figure 18 shows the generated synthesis script **`image_reconstruction_synplify.tcl`** with the example model.

```
project -new image_reconstruction.prj
add_file back_projection.vhd
add_file theta_count.vhd
add_file x_count.vhd
add_file y_count.vhd
add_file counters.vhd
add_file image_reconstruction.vhd
set_option -technology virtex4
set_option -part xc4vlx80
set_option -package ff1148
set_option -speed_grade -10
set_option -synthesis_onoff_pragma 0
set_option -frequency auto
set_option -write_apr_constraint 0
project -run synthesis
```

Figure 18. **`image_reconstruction_synplify.tcl`** *– synthesis script for Synplify Pro.*

To synthesize the design, run the generated synthesis script in Synplify Pro.

Figure 19 shows the Resource Usage report section from the Synplify Pro log file. Note that the resources needed by the Xilinx component (**`filtering_xil_cw`**) are not included because the component is synthesized as a black box.


Synplify Pro 8.8.0.4 - Dec 7 2006 - [image_reconstruction.htm]
File Edit View Project Run Analysis HDL-Analyst Options Window Tech-Support Web Help
image_reco...
image_recon...

rev_1 (image_reconstruction)

Compiler Report
Mapper Report
Timing Report
Performance Summary
Clock Relationships
Interface Information
Detailed Report for Clock: image_reconstruction|clk
    Starting Points with Worst Slack
    Ending Points with Worst Slack
    Worst Path Information
Detailed Report for Clock: System
    Starting Points with Worst Slack
    Ending Points with Worst Slack
    Worst Path Information
Resource Utilization

Log File Links:

rev_1
Hierarchical Area Report (image_reconstruction) (16:31

rev_1\par_1

Session Log

```
---------------------------------------
Resource Usage Report for image_reconstruction

Mapping to part: xc4vlx80ff1148-10
Cell usage:
DSP48           4 uses
FDCE            116 uses
FDPE            5 uses
GND             5 uses
MULT_AND        43 uses
MUXCY           1 use
MUXCY_L         82 uses
VCC             2 uses
XORCY           85 uses
filtering_xil_cw  1 use
LUT1            25 uses
LUT2            27 uses
LUT3            5 uses
LUT4            104 uses

I/O ports: 49
I/O primitives: 48
IBUF           20 uses
OBUF           28 uses

BUFGP          1 use

I/O Register bits:                  0
Register bits not including I/Os:   121 (0%)

RAM/ROM usage summary
32x1 ROMs (ROM32X1): 12


DSP48s: 4 of 80 (5%)

Global Clock Buffers: 1 of 32 (3%)

Total load per clock:
   image_reconstruction|clk: 126

Mapping Summary:
Total  LUTs: 185 (0%)

Mapper successful!
```

Figure 19. The Resource Usage report in the Synplify Pro log file window.

Implementing the Design

To implement the design in Xilinx ISE, create a new ISE project using the New Project Wizard, and add the netlist file `image_reconstruction.edf` generated by Synplify Pro as the project source file. After the project is created, set the Translate property **Macro Search Path** to the location of the NGC netlist `filtering_xil_cw.ngc` generated by Xilinx System Generator, or copy the file to the project directory. Run Implement Design to execute Translate, Map, and Place & Route.

The same can also be done programmatically using a `.tcl` script. The `image_recon_par.tcl` script in the example files accomplishes this, as shown in Figure 20.

```
# Create new directory
set prj_dir par
file mkdir $prj_dir
cd $prj_dir

# Create new project
project new image_recon.ise
project set family virtex4
project set device xc4vlx80
project set package ff1148
project set speed -10

# Add EDF netlist from Synplify Pro.
# Set the file path to match your project.
xfile add "../synthesis/rev_1/image_reconstruction.edf"

# Set search path for NGC subsystem.
# Set the file path to match your project.
project set "Macro Search Path" "../ngc_netlist"

# Implement the design
process run "Implement Design"
```

Figure 20. `image_recon_par.tcl` *– Implement the design in ISE using Tcl script.*

Revision History

Date	Document Version Number	Revision
02/09	2.0	Updated workflow for R2008a
02/08	1.0	Initial release for R2007a

Resources

VISIT
www.mathworks.com

TECHNICAL SUPPORT
www.mathworks.com/support

ONLINE USER COMMUNITY
www.mathworks.com/matlabcentral

DEMOS
www.mathworks.com/demos

TRAINING SERVICES
www.mathworks.com/training

THIRD-PARTY PRODUCTS AND SERVICES
www.mathworks.com/connections

WORLDWIDE CONTACTS
www.mathworks.com/contact

E-MAIL
info@mathworks.com

The MathWorks™
Accelerating the pace of engineering and science

91561v01 12/08