CoCentric®
SystemC™ Compiler
RTL User and Modeling Guide

Version U-2003.06, June 2003

Comments?
E-mail your comments about Synopsys
documentation to doc@synopsys.com

SYNOPSYS®

Copyright Notice and Proprietary Information

Printed in the U.S.A.

Document Order Number: 37582-000 QA
CoCentric SystemC Compiler RTL User and Modeling Guide, Version U-2003.06

Contents

Appendix A. Compiler Directives

Preface

This preface includes the following sections:

What’s New in This Release

Information about new features, enhancements, and changes; known problems and limitations; and resolved Synopsys Technical Action Requests (STARs) is available in the *SystemC Compiler Release Notes* in SolvNet.

To see the *SystemC Compiler Release Notes*,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
3. Click Release Notes in the Main Navigation section, find the U-2003.06 Release Notes, then open the *CoCentric SystemC Compiler Release Notes*.

About This Guide

The *CoCentric SystemC Compiler RTL User and Modeling Guide* describes how to use SystemC Compiler for RTL synthesis. It also describes how to develop or refine a SystemC RTL model for synthesis with SystemC Compiler.

For information about SystemC, see the Open SystemC Community Web site at http://www.systemc.org.

Audience

The *CoCentric SystemC Compiler RTL User and Modeling Guide* is for designers with a basic knowledge of the SystemC Class Library, RTL design, and the C or C++ language and development environment.

Familiarity with one or more of the following Synopsys tools is helpful:

- Synopsys Design Compiler
- Synopsys HDL Compiler for VHDL
- Synopsys HDL Compiler (Presto Verilog)
- Synopsys Scirocco VHDL Simulator
- Synopsys Verilog Compiled Simulator (VCS)

Related Publications

For additional information about SystemC Compiler, see

- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

You might also want to refer to the following documentation:

- The *CoCentric SystemC Compiler Behavioral User and Modeling Guide*, which provides information about how to synthesize a refined SystemC hardware behavioral module into an RTL or a gate-level netlist. It also describes how to develop or refine a behavioral SystemC model for synthesis with SystemC Compiler.
- The *CoCentric System Studio HDL CoSim User Guide*, which provides information about cosimulating a system with mixed SystemC and HDL modules.
- The *CoCentric SystemC Compiler Quick Reference*, which provides a list of command with their options and a list of variables that affect the SystemC Compiler tool behavior.
- The SystemC documentation, available from the Open SystemC Community Web site at http://www.systemc.org.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* `[`*`pin2 ... pinN`*`]`
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center. Customer training is available through the Synopsys Customer Education Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services, including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click SolvNet Help in the Support Resources section.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required) and click “Enter a Call With the Support Center.”
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Using SystemC Compiler for RTL Synthesis

The CoCentric SystemC Compiler tool synthesizes a SystemC description with a behavioral module, RTL modules, or a mixed RTL-behavioral module into an HDL RTL module or a gate-level netlist. After synthesis, you can use the HDL RTL description or the netlist as input to other Synopsys products such as the Design Compiler and Physical Compiler tools.

This chapter describes the RTL synthesis process and the commands you typically use, in the following sections:

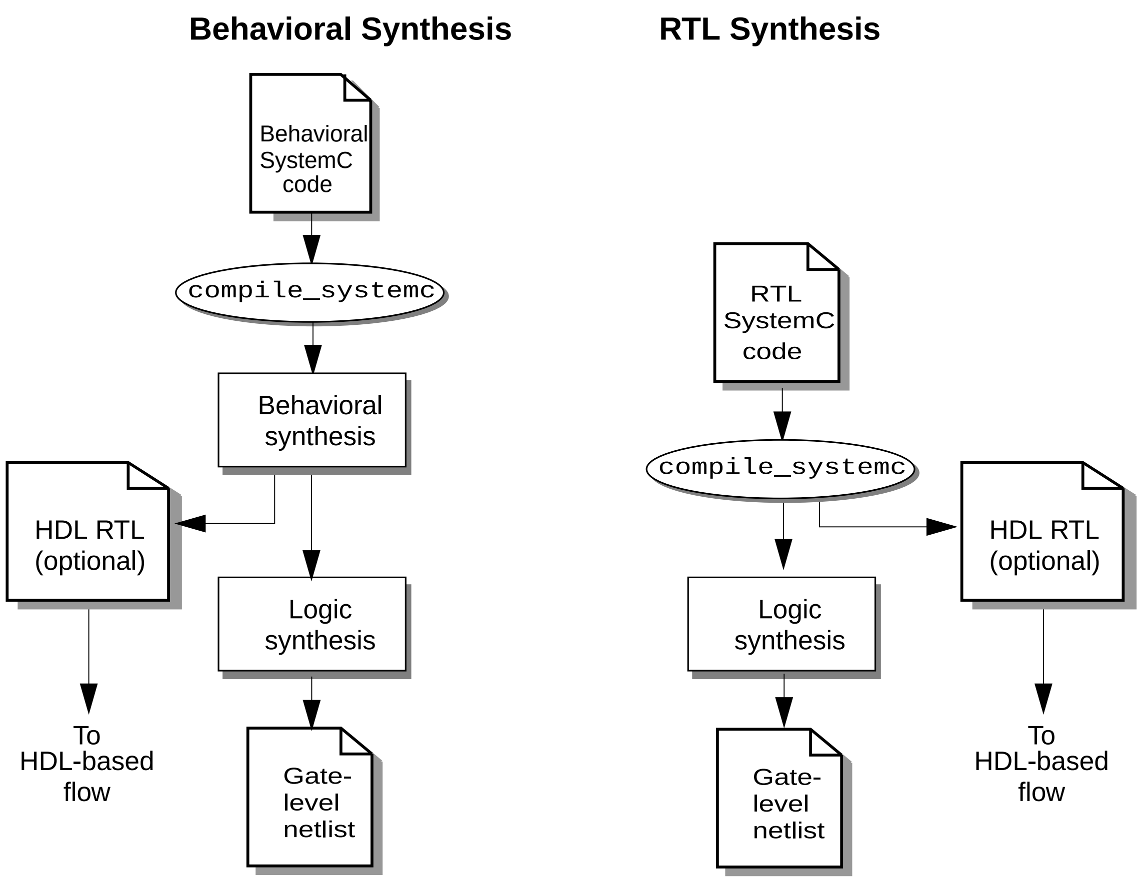
- Synthesis With SystemC Compiler
- RTL Design for Synthesis Overview
- Inputs and Outputs for RTL Synthesis
- Synthesizing a SystemC Design in a Single File
- Synthesizing a Design With Multiple RTL Files

- Synthesizing a Design With Integrated Behavioral and RTL Modules
- Passing Parameters to a Module
- Synthesizing a Design With an Instantiated HDL Model
- Synthesizing a Design With an Instantiated DesignWare Component

Synthesis With SystemC Compiler

SystemC Compiler is a tool that can accept RTL and behavioral SystemC descriptions and perform behavioral or RTL synthesis, as required, to create a gate-level netlist. You can also use SystemC Compiler to create an RTL HDL description for simulation or to use with other HDL tools in your flow. Figure 1-1 shows the behavioral and RTL synthesis paths to gate-level netlists.

Figure 1-1   Behavioral Synthesis Compared to RTL Synthesis



Choosing the Right Abstraction for Synthesis

You can implement a hardware module by using RTL or behavioral-level synthesis. An RTL model describes registers in your design and the combinational logic between them. You specify the functionality of your system as a finite state machine (FSM) and a datapath. Because register updates are tied to a clock, the model is cycle accurate, both at the interfaces and internally. Internal cycle accuracy means that you specify the clock cycle in which each operation is performed.

A behavioral model is an algorithmic description like a software program. Unlike a pure software program, however, the I/O behavior of the model is described in a cycle-accurate fashion. Therefore, wait statements are inserted into the algorithmic description to clearly delineate clock cycle boundaries and when I/O happens. Unlike RTL descriptions, the behavior is described algorithmically rather than in terms of an FSM and a datapath.

Evaluate each design module by module, and consider each module's attributes, described in the following sections, to determine whether RTL or behavioral synthesis is applicable.

Identifying Attributes Suitable for RTL Synthesis

Look for the following design attributes when identifying a hardware module that is suitable for RTL synthesis with SystemC Compiler:

- The design is asynchronous.
- It is easier to conceive the design as an FSM and a datapath than as an algorithm—for example, it is a microprocessor.

- The design is very high performance, and the designer, therefore, needs complete control over the architecture.
- The design contains complex memory such as SDRAM or RAMBUS.

Identifying Attributes Suitable for Behavioral Synthesis

Look for the following design attributes when identifying a hardware module that is suitable for behavioral synthesis with SystemC Compiler:

- It is easier to conceive the design as an algorithm than as an FSM and a datapath—for example, it is a fast Fourier transform, filter, an inverse quantization, or a digital signal processor.
- The design has a complex control flow—for example, it is a network processor.
- The design has memory accesses, and you need to synthesize access to synchronous memory.

For information about behavioral synthesis and modeling, see the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

RTL Design for Synthesis Overview

A pure C/C++ model of your hardware describes only what the hardware is intended to do, without providing information about the hardware structure or architecture. Starting with a C/C++ model, first modify the design to create the hardware structure. To do this,

- Define the I/O ports for the hardware module
- Specify the internal structure as modules
- Specify the internal communication between the modules

For each block in the design, you start with a functional-level SystemC model and change it into an RTL model for synthesis with SystemC Compiler. To modify the high-level model into an RTL model for synthesis, you

- Define the I/O in a cycle-accurate fashion
- Separate the control logic and datapath
- Determine the data-path architecture
- Define an explicit FSM for the control logic

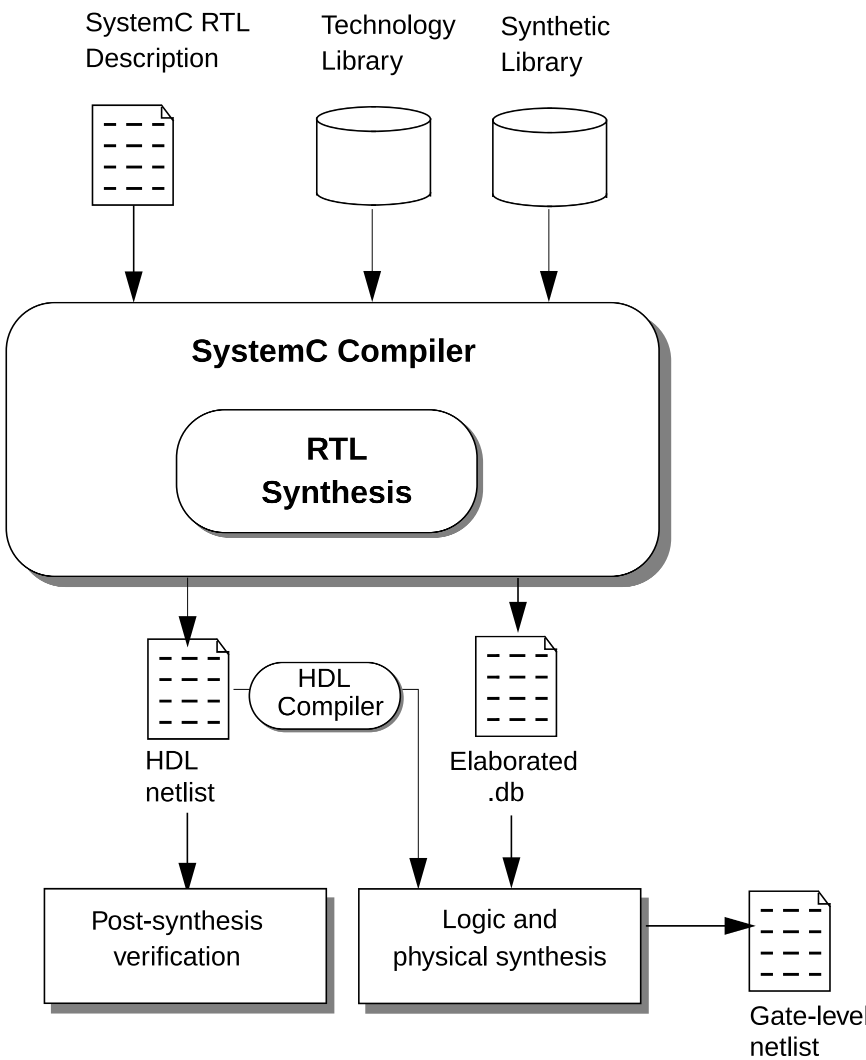
A high-level SystemC model can contain abstract ports, which are types that are not readily translated to hardware. For each abstract port, define a port or a set of ports to replace each terminal of the abstract port, and replace all accesses to the abstract ports or terminals with accesses to the newly defined ports. For information about abstract ports, see the http://www.systemc.org web site.

Further information on designing for synthesis is provided in “Modifying Data for Synthesis” on page 3-8 and “Recommendations About Modification for Synthesis” on page 3-20.

Inputs and Outputs for RTL Synthesis

SystemC Compiler requires a SystemC RTL description and libraries defining the components and technology that will be used to implement the hardware. Figure 1-2 shows the flow into and out of SystemC Compiler.

Figure 1-2 SystemC Compiler Input and Output Flow for RTL Synthesis



RTL Description

Write the SystemC RTL description, using the SystemC Class Library according to the guidelines in Chapter 2, "Creating SystemC Modules for Synthesis," Chapter 3, "Using the Synthesizable Subset," and Chapter 4, "RTL Coding Guidelines."

The RTL description is independent of the technology. Using SystemC Compiler, you can change the target technology library without modifying the RTL description.

The example designs used in this manual are described in Appendix B, "Examples." The files for these examples are available in the SystemC Compiler installation in the $SYNOPSYS/doc/syn/ccsc/ccsc_examples directory.

Technology Library

A technology library is provided by ASIC vendors in Synopsys .db database format. It provides the area, timing, wire load models, and operating conditions. You provide the path to your chosen technology library for your design by defining the `target_library` variable in dc_shell.

Sample technology libraries are provided in the SystemC Compiler installation at $SYNOPSYS/libraries/syn.

Synthetic Library

The DesignWare synthetic library is a technology-independent library of logic components such as adders and multipliers. SystemC Compiler maps your design operators to the synthetic library logical components. You provide the path to your chosen synthetic libraries for your design by defining the `synthetic_library` variable in dc_shell.

The DesignWare synthetic libraries are provided in the SystemC Compiler installation at $SYNOPSYS/libraries/syn. The synthetic libraries have names such as standard.sldb, dw01.sldb, and dw02.sldb. For information about the DesignWare libraries, see the DesignWare online documentation.

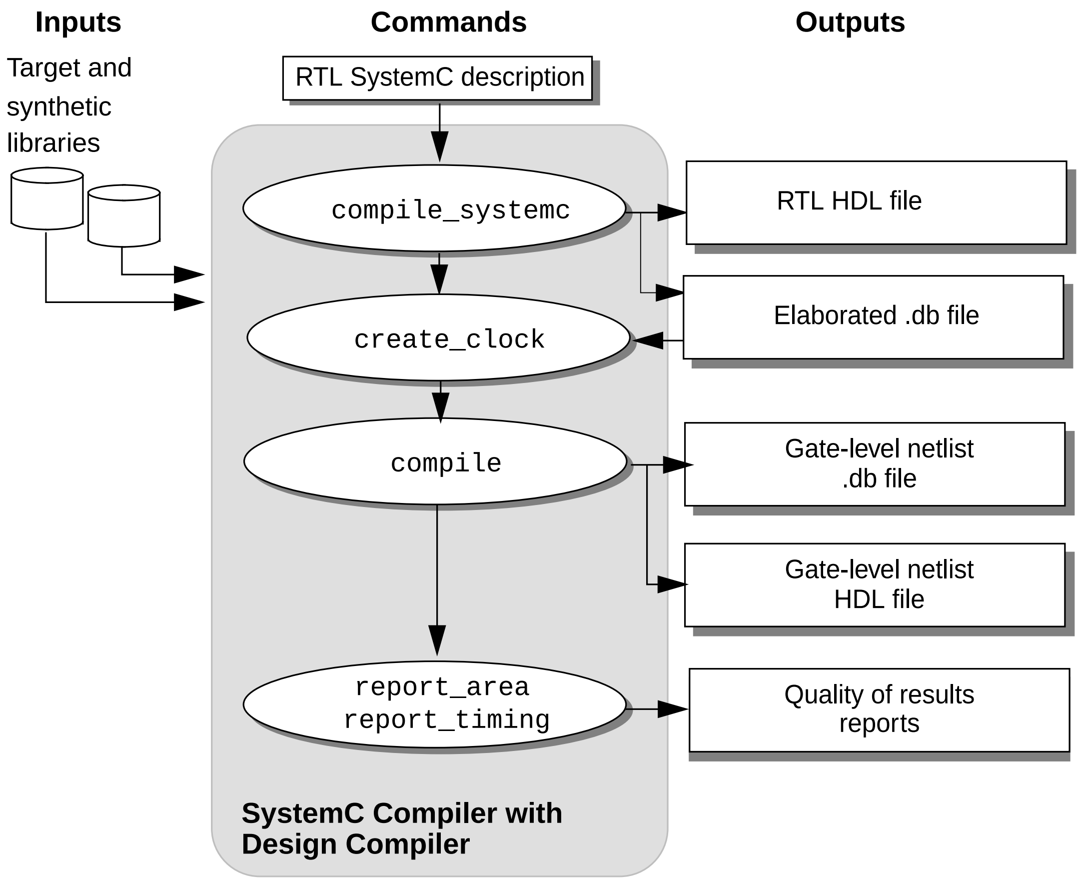
Outputs From SystemC Compiler

SystemC Compiler generates an elaborated .db file for input into the Design Compiler tool. It also generates RTL HDL files that can be used in HDL-based flows.

Synthesizing a SystemC Design in a Single File

Figure 1-3 illustrates the primary commands you use to perform synthesis of a SystemC RTL design in a single file with SystemC Compiler and compile the design into gates (using Design Compiler). The diagram also shows the inputs you provide and the outputs produced at various stages.

Figure 1-3   Single RTL Module Command Flow



The commands used in this chapter show the typical options you use. For a full description of a command and all its options, see the Synopsys online man pages. How to access and use man pages is described in Appendix A in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

Starting SystemC Compiler

SystemC Compiler is integrated into Design Compiler. Enter the SystemC Compiler commands at the dc_shell prompt or use the `include` command to run a script that contains the commands. To start dc_shell or dc_shell-t, enter the following at a UNIX prompt:

```
unix% dc_shell
```

or

```
unix% dc_shell-t
```

If this is the first time you are using SystemC Compiler, see Appendix A in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide* for information about setting up your environment, entering commands, and using scripts.

Elaborating Your Design

Use the `compile_systemc` command to read your SystemC source code and check it for compliance with synthesis policy, C++ syntax, and C++ semantics. If there are no errors, it produces an internal database ready for synthesis. This process is called analysis and elaboration.

The `compile_systemc` command, using the default settings for options, does the following:

- Checks C++ syntax and semantics
- Replaces source code arithmetic operators with DesignWare components

- Performs optimizations such as constant propagation, constant folding, dead code elimination, function inlining, and algebraic simplification
- For a behavioral module, performs the necessary elaboration steps to prepare the SystemC description for timing analysis, scheduling, and logic synthesis
- For a mixed RTL-behavioral module, creates a behavioral submodule that contains all the behavioral processes

The `compile_systemc` command and the other SystemC Compiler commands respond with a 1 if no errors were encountered or a 0 if an error was encountered. It also displays explanatory messages for errors and warnings.

Analyzing and Elaborating a Design With the compile_systemc Command

If your design has one or more modules with one or more RTL processes, use the `compile_systemc` command with the `-rtl` option to elaborate the design. For example, to elaborate the count zeros sequential design, enter

```
dc_shell> compile_systemc -rtl count_zeros_cseq.cc
```

Use the `compile_systemc` command without the `-rtl` option to elaborate a design with a behavioral module or a mixed RTL-behavioral module. Behavioral synthesis and elaboration of a mixed RTL-behavioral module are described in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

For information about issuing C++ compiler preprocessor options with the `compile_systemc` command, see Appendix A in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

Creating an Elaborated .db File for Synthesis

To create an internal database of your SystemC RTL module for synthesis with Design Compiler, enter

```
dc_shell> compile_systemc -rtl -format db
            design_name.cc
```

The `-format` option arguments are `db`, `verilog`, and `vhdl`. The default is `db`. You can also specify an argument list with a combination of the arguments. For example,

```
dc_shell> compile_systemc -rtl
          -format {db, verilog} design_name.cc
```

To write out an elaborated database as a .db file, for example to use with Physical Compiler, use the `write` command with the `-output` option.

Enter

```
dc_shell> write
      -hierarchy
      -output ./WORK/design_name_elab.db
```

This command writes the elaborated .db file into the ./WORK directory.

Creating an RTL HDL Description

In certain cases, you might want to convert a SystemC RTL description into a Verilog or VHDL RTL description.

Creating a Verilog Netlist. To create a Verilog netlist of your SystemC RTL design,

1. Execute the `compile_systemc` command with the following options:

   ```
   dc_shell> compile_systemc -rtl
            -format verilog
            design_name.cc
   ```

   When you execute the `compile_systemc` command with the `-format verilog` option, SystemC Compiler creates a separate Verilog .v file in the current working directory for each module named *module_name*.v.

2. To analyze and elaborate the Verilog *module_name*.v file created in step 1 with HDL Compiler, enter

   ```
   dc_shell> analyze
             -format verilog
             module_name.v
   dc_shell> elaborate module_name
   ```

   The `analyze` command translates the Verilog file into an internal database format, and the `elaborate` command creates and optimizes the circuit that corresponds to the RTL description.

Creating a VHDL Netlist. To compile and create a VHDL netlist of your SystemC RTL design,

1. Execute the `compile_systemc` command with the following option:

   ```
   dc_shell> compile_systemc -rtl
            -format vhdl
            design_module.cc
   ```

When you execute the `compile_systemc` command with the `-format vhdl` option, SystemC Compiler creates a separate VHDL .vhd file in the current working directory for each module named *module_name*.vhd.

2. To use the HDL Compiler tool to analyze and elaborate the VHDL *design_name*.vhd file created in step 1, enter

```
dc_shell> analyze
          -format vhdl
          module_name.vhd
dc_shell> elaborate module_name
```

The `analyze` command translates the VHDL file into an internal database format, and the `elaborate` command creates and optimizes the circuit that corresponds to the RTL description.

The *design_module*.vhd file contains a Library statement and Use statements to define the standard VHDL libraries used by the design.

Creating a Single HDL Netlist for Multiple RTL Modules. By default, SystemC Compiler creates a separate RTL file for each RTL module synthesized with the `compile_systemc` command. To direct SystemC Compiler to create a single HDL file with multiple RTL modules, use the `compile_systemc` command with the `-single` option.

For example,

```
dc_shell> compile_systemc -rtl
           -format verilog
          -single  design_name.cc
```

Designating an HDL File Name. By default, SystemC Compiler creates an HDL file named *module_name*.v or *module_name*.vhd. To direct SystemC Compiler to create the HDL file with a different name, use the `compile_systemc` command with the `-output` option.

For example,

```
dc_shell> compile_systemc -rtl
           -format verilog
          -output ./my_new_name.v
         design_name.cc
```

Designating a Directory and Library for the Design

By default, SystemC Compiler writes the intermediate files it creates while executing the `compile_systemc` command to the WORK library. By default, the WORK library is mapped to the current working directory.

To map the WORK library or a design library you create to a physical UNIX directory other than the default current working directory, use the `define_design_lib` command. You need to create the directory before you can map a library to it.

To create a WORK directory and map the WORK library to it so SystemC Compiler writes the intermediate files into the WORK directory instead of the current working directory, enter

```
dc_shell> mkdir ./WORK
dc_shell> define_design_lib WORK
         -path ./WORK
```

To create a new library named *my_design_library* and a new directory named *my_design_lib* for the intermediate files, enter

```
dc_shell> mkdir  /usr/design_libs/my_design_lib
dc_shell> define_design_lib my_design_library
         -path /usr/design_libs/my_design_lib
```

After you create a new library and map it to a directory, you can designate a design library during synthesis other than the default WORK library for the design by using the `compile_systemc` command `-library` option.

For example,

```
dc_shell> compile_systemc  -library my_design_library
         -rtl -format db design_name.cc
```

You can also use the `compile_systemc` command `-work` option instead of the `-library` option. The `-work` option is an alias for the `-library` option.

Setting the Clock Period

If your design has a clock port, use the `create_clock` command to set the clock period for the clock port. The clock period uses the same unit that is defined in the target technology library.

For example, to create a clock for the port in your design named *clk* with a period of 10 units, enter

```
dc_shell> create_clock clk -period 10
```

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

Compiling and Writing the Gate-Level Netlist

Use the `compile` command to create the gate-level netlist. This command performs logic synthesis and optimization of the current design.

```
dc_shell> compile
          -map_effort low | medium | high
```

Use the following command to write the gate-level netlist in .db format:

```
dc_shell> write
          -hierarchy
          -output my_netlist.db
```

For verification at the gate level, write a Verilog or VHDL gate-level netlist file by entering the following command:

```
dc_shell> write
          -format verilog
          -hierarchy
         -output my_netlist.v
```

or

```
dc_shell> write
          -format vhdl
          -hierarchy
         -output my_netlist.vhd
```

Generating Summary Reports

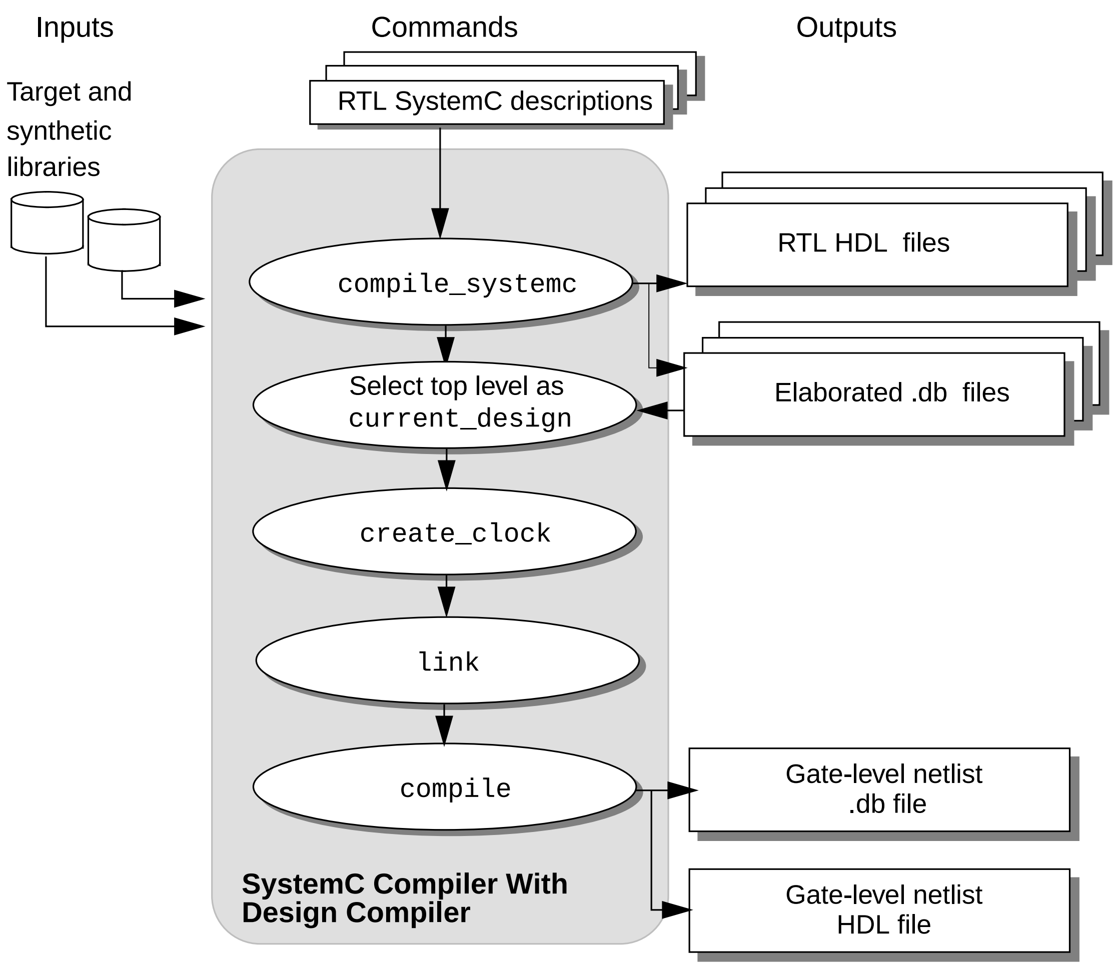
To generate summary reports of a design after it is compiled to gates, use one or both of the following commands:

```
dc_shell> report_area
dc_shell> report_timing
```

Synthesizing a Design With Multiple RTL Files

Figure 1-4 illustrates the primary commands you use to perform synthesis of a design with multiple RTL SystemC files and compile the design into gates. The diagram also shows the inputs you provide and the outputs SystemC Compiler can provide.

Figure 1-4 Command Flow for Multiple RTL Files



Analyzing and Elaborating Multiple RTL Files

If your design is hierarchical and has multiple RTL files, use the `compile_systemc` command to elaborate each file separately, and then use the `link` command to link the internal databases before compiling the design to gates.

To create internal database files of your separate SystemC RTL files for synthesis,

- Analyze and elaborate each file in your design, using the following command:

```
dc_shell> compile_systemc -rtl -format db
          module1_name.cc
dc_shell> compile_systemc -rtl -format db
          module2_name.cc
dc_shell> compile_systemc -rtl -format db
          top_module_name.cc
```

  The `compile_systemc` command with the `-rtl -format db` options creates a separate internal database for each file.

  If you want to compile your entire design to gates, the current design must be the top-level RTL module that instantiates all other RTL modules. To ensure that the current design in SystemC Compiler memory is the top-level RTL module, execute the `compile_systemc` command with the file containing the top RTL module last. Or you can use the `current_design` command to make the top-level module the current design. For example, to change the current design to *my_design_abc*, enter

```
dc_shell> current_design  my_design_abc
```

Setting the Clock Period

Use the `create_clock` command to set the clock period if your design contains a clock port. If your design does not have a clock port, you can skip this command. Enter

```
dc_shell> create_clock -name clk -period 10
```

The clock period uses the same unit that is defined in the target technology library.

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

Linking the .db Files

After analyzing and elaborating the modules in your design and before compiling the design to gates, use the `link` command to connect all the library components and subdesigns that your design references. Enter

```
dc_shell> link
```

The `link` command removes existing links to all library components and subdesigns before it starts the linking process. If you do not enter the `link` command to manually link the design references, the `compile` command performs linking but does not remove existing links.

Compiling and Writing the Gate-Level Netlist

Use the `compile` command to create the gate-level netlist of the hierarchical RTL design. This command performs logic synthesis and optimization of the current design.

```
dc_shell> compile
          -map_effort low | medium | high
```

Use the following command to write the gate-level netlist in .db format:

```
dc_shell> write
          -hierarchy
          -output top_module_netlist.db
```

Synthesizing a Design With Integrated Behavioral and RTL Modules

To perform synthesis of a design with integrated RTL and behavioral modules, synthesize the RTL and behavioral modules to gates before linking the integrated design, using the following steps:

1. Create the gate-level internal databases of your SystemC RTL modules, as described in "Synthesizing a Design With Multiple RTL Files" on page 1-19.
2. Create the gate-level internal databases of your SystemC behavioral modules, as described in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

   If your design has more than one behavioral module, instantiate the multiple behavioral modules in an RTL module.

3. Analyze and elaborate the top-level RTL module that combines the RTL and behavioral modules. Enter

```
dc_shell> compile_systemc -rtl -rtl_format db  all_top.h
```

4. Use the `read` command to read in the RTL and behavioral modules (.db files) that you previously compiled to gates, if the RTL and behavioral databases are not already in memory. Enter

```
dc_shell> read rtl_gates.db
dc_shell> read behavioral_gates.db
```

5. Use the `link` command to link the RTL, behavioral, and library .db into a single design.

```
dc_shell> link
```

6. Compile the integrated hierarchical RTL and behavioral design to gates and write the gate-level netlist, as described in "Compiling and Writing the Gate-Level Netlist" on page 1-22.

Elaborating a mixed RTL-behavioral module is described in the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

Passing Parameters to a Module

When you have an RTL module with one or more parameters, you can pass the parameter values from the command line with the `compile_systemc` command `-param` option. If the file *module_top.cc*, for example, has the following parameterized module definitions

```
// a, b, and c are parameters
M1 (const sc_module_name &name_,
         int a, int b, int c);

// d is a parameter
M2 (const sc_module_name &name_,
         int d);
```

You can pass parameter values by position to *module_top.cc* by entering

```
dc_shell> compile_systemc -rtl -param "M1 (5, 6, 7);M2 (8);"
          module_top.cc
```

This creates M1 with a = 5, b = 6, and c = 7 and M2 with d = 8.

Do not enter the module name, because it is not used for synthesis.

For more information about creating parameterized modules and setting default parameter values, see "Defining a Constructor With the SC_HAS_PROCESS Macro" on page 2-18.

Limitations for Passing Parameters

When you pass module parameters with the `-param` option, each parameter value must be a constant.

Names of Parameterized Modules

SystemC Compiler creates a unique module for each distinct module parameterization, and the parameter values are propagated into the module. It appends the parameter values to the module name during elaboration to create a unique module name. For example,

```
dc_shell> compile_systemc -rtl -param "M1 (5, 6, 7);M2 (8);"
          module_top.cc
```

This command creates the following module names:

```
M1_5_6_7
M2_8
```

When you also use the `-format verilog` option, the following Verilog file names are created:

```
M1_5_6_7.v
M2_8.v
```

If the module contains a loop that creates more than one instance of a module, SystemC Compiler appends the loop iteration count to the module instance name to create a unique name.

For a particular situation, you can direct SystemC Compiler not to rename the modules with the `compile_systemc` command `-dont_rename` option. For example,

```
dc_shell> compile_systemc -rtl -param "M1 (5, 6, 7);M2 (8);"
         -dont_rename "M1, M2"
         module_top.cc
```

This command creates the following module names without appending the parameter values to the module name:

```
M1   M2
```

If you use the `-format verilog` option, the Verilog file names created are the following:

```
M1.v
M2.v
```

Synthesizing a Design With an Instantiated HDL Model

To instantiate a Verilog or VHDL model in your SystemC RTL design, create a dummy SystemC module with the same module name and port names as those of the HDL model that you want to instantiate. This provides the SystemC design with the interface to your HDL model. The module and port names are case-sensitive and must exactly match the HDL names.

You do not need to describe the module's function in the dummy module, because Design Compiler replaces it with the actual HDL internal database .db file. You can treat the SystemC and HDL models separately and then link them together with the `link` command.

Example 1-1 shows a dummy module for the *simple* HDL model. An instance of the *simple* module named *m_pSimple* is created in the SystemC *inst* module.

Example 1-1 Instantiating an HDL .db in a SystemC Design

```
/****simple.h****/
#include <systemc.h>
// Dummy module for the VHDL .db
SC_MODULE(simple){
   sc_in<sc_logic> a;
   sc_in<sc_lv<2> > b;
   sc_out<sc_logic> z;
   SC_CTOR(simple){ }
};
/****inst.cpp****/
#include <systemc.h>
#include "simple.h"
SC_MODULE(inst){
   sc_in<sc_logic> pi_a;
   sc_in<sc_lv<2> > pi_b;
   sc_out<sc_logic> po_z;

   simple *m_pSimple;
   ... // Functionality of inst.

   SC_CTOR(inst){
```

```
      m_pSimple = new simple("simple_systemc_wrapper");
      m_pSimple->a(pi_a);
      m_pSimple->b(pi_b);
      m_pSimple->z(po_z);
   }
};
```

You need to use the `analyze`, `elaborate`, and `compile` commands with the HDL model before you use the `compile_systemc` command with the SystemC design. Then use the `link` command to link the internal databases before compiling the design to gates.

To create internal database files of your separate HDL and SystemC RTL models and synthesize the design to gates, enter

```
/* Elaborate the HDL design into memory */
dc_shell> analyze -format [vhdl simple.vhd
          | verilog simple.v]
dc_shell> elaborate simple
dc_shell> compile
/* Elaborate the SystemC design into memory */
dc_shell> compile_systemc  -rtl inst.rtl.cpp
dc_shell> current_design = inst
/* Link the SystemC and HDL designs */
dc_shell> link
dc_shell> compile
```

Depending on your design, you can compile the HDL and SystemC modules separately before linking them.

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

Synthesizing a Design With an Instantiated DesignWare Component

Instantiating a DesignWare component in your SystemC RTL design is similar to instantiating an HDL model. You need to create a dummy SystemC module with the same module name and port names as those of the DesignWare component that you want to instantiate. This provides the SystemC design with the interface to the DesignWare component. The module and port names are case-sensitive and must exactly match the DesignWare component names.

You do not need to describe the module's function in the dummy module, because Design Compiler replaces it with the DesignWare equivalent from the technology library during synthesis.

If the DesignWare component has a parameterized port width, you can specify the port width as a constructor parameter, as described in "Passing Parameters to a Module" on page 1-23. Example 1-2 shows a dummy module for the DW01_add component with a constructor parameter to pass the port width.

Example 1-2   Dummy SystemC Module For a DesignWare Component

```
/****dw01_add.h****/
#include "systemc.h"

/*
 *   This dummy header matches the pinout
 *   of the DW01_add block.
 *   This module does not require functionality
 *   for synthesis; you need to provide the
 *   DW01_add functionality for simulation.
 */

SC_MODULE(DW01_add) {
    sc_in< sc_uint<8> > A, B;
    sc_in<bool> CI;
    sc_out< sc_uint<8> > SUM;
    sc_out< bool > CO;
```

```
    SC_HAS_PROCESS(DW01_add);

    DW01_add(sc_module_name &_name, sc_uint<5> width) {   }
};
```

Example 1-3 shows creating an instance of the DW01_add module named *DWAdder* in the SystemC *adder* module. A constructor parameter specifies a bit-width of 8 for the DesignWare component.

Example 1-3 Instantiating a DesignWare Component in a SystemC Module

```
/****adder.h****/
#include "systemc.h"
#include "dw01_add.h"

SC_MODULE(adder) {
  sc_in<bool>  clk, reset;
  sc_in< bool >       carry;
  sc_in< sc_uint<8> > data1;
  sc_in< sc_uint<8> > data2;

  sc_out< sc_uint<8> > dataOut;

  DW01_add *DWAdder;

  sc_signal<bool> open;

  SC_CTOR(adder) {
    DWAdder = new DW01_add("DWAdder", 8);
    DWAdder->A(data1);
    DWAdder->B(data2);
    DWAdder->CI(carry);
    DWAdder->SUM(dataOut);
    DWAdder->CO(open);
  }
};
```

You need to use the `compile_systemc` command with the `-dont_rename` option for the SystemC design before using the `analyze` and `elaborate` commands with the intermediate HDL model of the DesignWare component. Then use the `link` command to link the internal databases before compiling the design to gates.

To create internal databases of the SystemC RTL model and synthesize the design to gates,

```
/* Elaborate the SystemC design into memory
 * and create an intermediate HDL file */
dc_shell> compile_systemc  -rtl -format [verilog | vhdl]
          -dont_rename "DW01_add" adder.h
/* Elaborate the HDL design into memory */
dc_shell> analyze -format [vhdl simple.vhd
          | verilog simple.v]
dc_shell> elaborate simple
/* Link the SystemC and HDL designs */
dc_shell> link
dc_shell> compile
```

Implementing functions with DesignWare components is described in “Specifying Preserved Functions and Implementing DesignWare Components” on page 2-43.

2

Creating SystemC Modules for Synthesis

This chapter explains the SystemC and C/C++ language elements that are important for RTL synthesis with SystemC Compiler. It contains the following sections:

Defining Modules and Processes

This modeling guide explains how to develop SystemC RTL modules for synthesis with SystemC Compiler. It assumes that you are knowledgeable about the C/C++ language and the SystemC Class Library available from the Open SystemC Community Web site at http://www.systemc.org.

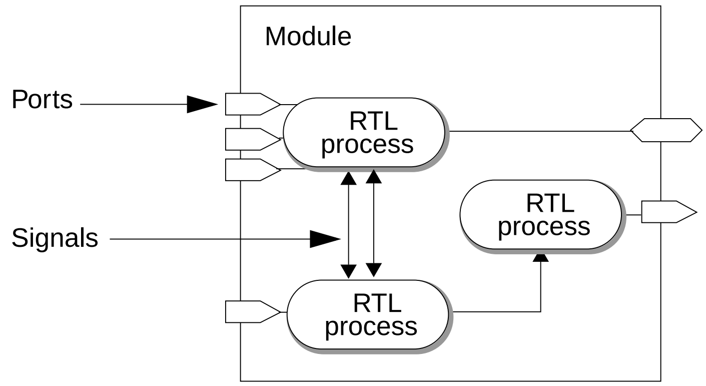
Modules

The basic building block in SystemC is the module. A SystemC module is a container in which processes and other modules are instantiated. For synthesis with SystemC Compiler, modules are either RTL or behavioral. A typical module can have

- Single or multiple RTL processes to specify combinational or sequential logic
- Single or multiple behavioral processes
- Multiple instantiated modules to specify hierarchy
- One or more member functions that are called from within an instantiated process

Figure 2-1 illustrates a module with several RTL processes. The processes within a module are concurrent, and they execute whenever one of their sensitive inputs changes.

Figure 2-1   Module



Processes

SystemC provides processes for describing the parallel behavior of hardware systems. This means processes execute concurrently, rather than sequentially like C++ functions. The code within a process, however, executes sequentially.

You can declare more than one process in a module, but processes cannot contain other processes or modules.

Registering a Process

Defining a process is similar to defining a C++ function. You declare a process as a member function of a module class and register it as a process in the module's constructor, which is described in "Creating a Process in a Module" on page 2-11. When you register a process, it is recognized as a SystemC process rather than as an ordinary member function.

You can register multiple processes, but it is an error to register more than one instance of the same process. To create multiple instances of the same process, enclose the process in a module and instantiate the module multiple times.

Triggering Execution of a Process

You define a sensitivity list that identifies which input ports and signals trigger the execution of the code within a process. You can define level-sensitive inputs to specify combinational logic or edge-sensitive inputs to specify sequential logic, which is described in “Defining the Sensitivity List” on page 2-12.

Reading and Writing in a Process

A process can read from and write to ports, signals, and internal variables, as described in “Reading and Writing Ports and Signals” on page 2-28.

Processes use signals to communicate with each other. One process can cause another process to execute by assigning a new value to a signal that connects them. Do not use data variables for communication between processes, because the processes execute in random order and it can cause nondeterminism (order dependencies) during simulation.

Types of Processes

SystemC provides three process types—SC_METHOD, SC_CTHREAD, and SC_THREAD—that execute whenever their sensitive inputs change. A process has a sensitivity list that identifies which inputs trigger the code within the process to execute when the value on one of its sensitive inputs changes.

For simulation and testbenches, you can use any of the process types.

SC_METHOD Process. The SC_METHOD process is used to describe a hierarchical design or RTL hardware. It is level sensitive, meaning it is sensitive to changes in the signal values, or it is edge sensitive, meaning it is sensitive to particular transitions (edges) of the signal, and it executes when one of its sensitive inputs changes.

SC_CTHREAD Process. The SC_CTHREAD clocked thread process is sensitive to one edge of one clock. Use a clocked thread process to describe functionality for behavioral synthesis with SystemC Compiler.

The SC_CTHREAD process models the behavior of a sequential logic circuit with nonregistered inputs and registered outputs. A registered output comes directly from a register (flip-flop) in the synthesized circuit.

For information about creating behavioral processes, see the *CoCentric SystemC Compiler Behavioral User and Modeling Guide*.

Thread Process. The SC_THREAD process is not used for synthesis. For more information about the SC_THREAD process, see the SystemC documentation at the Open SystemC Community Web site http://www.systemc.org.

Creating a Module

As a recommended coding practice, describe a module by using a separate header file (*module_name*.h) and an implementation file (*module_name*.cpp or *module_name*.cc).

Module Header File

Each module header file contains

- Port declarations
- Internal signal variable declarations
- Internal data variable declarations
- Process declarations
- Member function declarations
- A module constructor

Module Syntax

Declare a module by using the syntax shown in bold in the following example:

```
#include "systemc.h"
SC_MODULE (module_name) {
      //Module port declarations
      //Signal variable declarations
      //Data variable declarations
      //Member function declarations
      //Method process declarations

      //Module constructor
      SC_CTOR (module_name) {
                  //Register processes
                  //Declare sensitivity list
      }
```

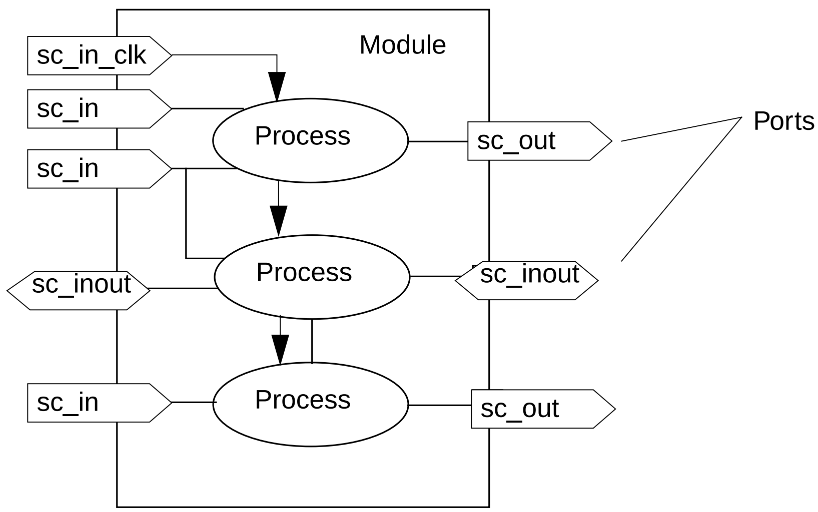
```
};
```

SC_MODULE and SC_CTOR are C++ macros defined in the SystemC Class library.

Module Ports

Each module has any number of ports that determine the direction of data into or out of the module, as shown in Figure 2-2.

Figure 2-2 Module Ports



A port is a data member of SC_MODULE. You can declare any number of sc_in, sc_out, and sc_inout ports.

Note:

> The `compile_systemc -rtl -format verilog` command converts an sc_inout port to a Verilog out port, not an inout port. You can read from and write to a Verilog out port. Verilog inout ports have restrictions for synthesis, as described in the *HDL Compiler (Verilog Presto) Reference Manual*.

For VHDL, the `compile_systemc -rtl -format vhdl` command treats an sc_inout port as a VHDL inout port. It treats an sc_out port as an out port and a signal or an out port, depending on the situation.

Port Syntax

Declare ports by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
      //Module port declarations
      sc_in<port_data_type> port_name;
      sc_out<port_data_type> port_name;
      sc_inout<port_data_type> port_name;
      sc_in<port_data_type> port_name;


      //Module constructor
      SC_CTOR (module_name) {
                  //Register processes
                  //Declare sensitivity list
      }
};
```

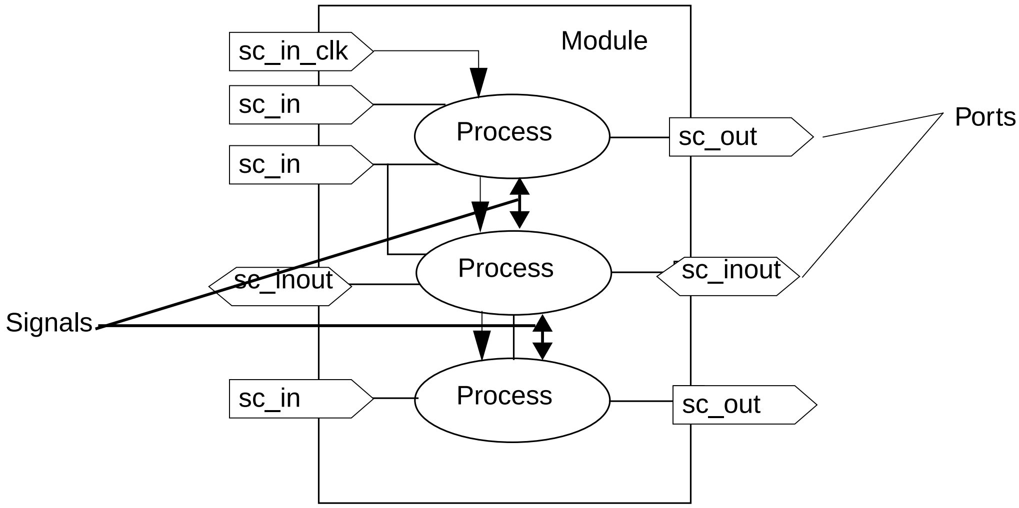
Port Data Types

Ports connect to signals and have a data type associated with them. For synthesis, declare each port as one of the synthesizable data types described in "Converting to a Synthesizable Subset" on page 3-2.

Signals

Modules use ports to communicate with other modules. In hierarchical modules, use signals to communicate between the instantiated modules. Use internal signals for peer-to-peer communication between processes within the same module, as shown in Figure 2-3.

Figure 2-3 Processes and Signals



Signal Syntax

Declare signals by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
      //Module port declarations
      sc_in<port_type> port_name;
      sc_out<port_type> port_name;
      sc_in<port_type>port_name;

      //Internal signal variable declarations
      sc_signal<signal_type> signal_name;
      sc_signal<signal_type> signal1, signal2;

      //Data variable declarations
      //Process declarations
      //Member function declarations

      //Module constructor
      SC_CTOR (module_name) {
                  //Register processes
                  //Declare sensitivity list
      }
};
```

Signal Data Types

A signal's bit-width is determined by its corresponding data type. Specify the data type as any of the synthesizable SystemC or C++ data types listed in "Converting to a Synthesizable Subset" on page 3-2. Signals and the ports they connect must have the same data types.

Data Member Variables

Inside a module, you can define data member variables of any synthesizable SystemC or C++ type. These variables can be used for internal storage in the module. Recommendations about using data member variables for synthesis are provided in "Data Members of a Module" on page 3-18. Declare internal data variables by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
      //Module port declarations
      sc_in<port_type> port_name;
      sc_out<port_type> port_name;
      sc_in port_name;

      //Internal signal variable declarations
      sc_signal<signal_type> signal_name;

      //Data member variable declarations
      int count_val;        //Internal counter
      sc_int<8> mem[1024];  //Array of sc_int

      //Process declarations
      //Member function declaration

      //Module constructor
      SC_CTOR (module_name) {
                  //Register processes
                  //Declare sensitivity list
      }
};
```

Note:

Do not use data variables for peer-to-peer communication in a module. This can cause pre-synthesis and post-synthesis simulation mismatches and nondeterminism (order dependency) in your design.

Assigning to Data Members in the Constructor

You can make assignments to data members from within the constructor. These assignments are treated as constants for synthesis. You cannot reassign the data members within any process in the module.

Creating a Process in a Module

You declare a process as a member function of a module class and register it as a process in the module's constructor. You must declare a process with a return type of void and no arguments, as shown in bold in Example 2-1.

To register a function as an SC_METHOD process, use the SC_METHOD macro that is defined in the SystemC class library. The SC_METHOD macro takes one argument, the name of the process.

Example 2-1 Creating a Method Process in a Module

```
SC_MODULE(my_module){
     // Ports
     sc_in<int> a;
     sc_in<bool> b;
     sc_out<int> x;
     sc_out<int> y;
     // Internal signals
     sc_signal<bool>c;
     sc_signal<int> d;
     // process declaration
     void my_method_proc();
     // module constructor
     SC_CTOR(my_module) {
          // register process
          SC_METHOD(my_method_proc);
          // Define the sensitivity list
     }
};
```

Defining the Sensitivity List

An SC_METHOD process reacts to a set of signals called its sensitivity list. You can use the sensitive(), sensitive_pos(), or sensitive_neg() functions or the sensitive, sensitive_pos, or sensitive_neg streams in the sensitivity declaration list.

Defining a Level-Sensitive Process

For combinational logic, define a sensitivity list that includes all input ports, inout ports, and signals used as inputs to the process. Use the sensitive() method to define the level-sensitive inputs. Example 2-2 shows in bold a stream-type declaration and a function-type declaration. Specify any number of sensitive inputs for the stream-type declaration, and specify only one sensitive input for the function-type declaration. You can call the sensitive function multiple times with different inputs.

Example 2-2 Defining a Level-Sensitive Sensitivity List

```
SC_MODULE(my_module){
     // Ports
     sc_in<int> a;
     sc_in<bool> b;
     sc_out<int> x;
     sc_out<int> y;
     // Internal signals
     sc_signal<bool>c;
     sc_signal<int> d;
     sc_signal<int> e;
     // process declaration
     void my_method_proc();
     // module constructor
     SC_CTOR(my_module) {
          // register process
          SC_METHOD(my_method_proc);
          // declare level-sensitive sensitivity list
          sensitive << a << c << d; // Stream declaration
          sensitive(b);  //Function declaration
          sensitive(e);  //Function declaration
     }
};
```

Incomplete Sensitivity Lists

To eliminate the risk of pre-synthesis and post-synthesis simulation mismatches, include all the inputs to the combinational logic process in the sensitivity list of the method process. Example 2-3 shows an incomplete sensitivity list.

Example 2-3 Incomplete Sensitivity List

```
//method process
void comb_proc () {
    out_x = in_a & in_b & in_c;
}

SC_CTOR( comb_logic_complete ) {
  // Register method process
  SC_METHOD( comb_proc);
  sensitive << in_a << in_b;  // missing in_c
}
```

SystemC Compiler issues a warning if your sensitivity list is incomplete, but it proceeds to build a 3-input AND gate for the description in Example 2-3. When you simulate this description, however, out_x is not recalculated when in_c changes, because in_c is not in the sensitivity list. The simulated behavior, therefore, is not that of a 3-input AND gate.

Defining an Edge-Sensitive Process

For sequential logic, define a sensitivity list of the input ports and signals that trigger the process. Use the sensitive_pos(), the sensitive_neg(), or both the sensitive_pos() and sensitive_neg() methods to define the edge-sensitive inputs that trigger the process. Declare ports and the edge-sensitive inputs as type sc_in<bool>.

For edge-sensitive inputs, SystemC Compiler tests for the rising or falling edge of the signal. It infers flip-flops for variables that are assigned values in the process.

Define the sensitivity list by using either the function or the stream syntax. Example 2-4 shows in bold an example of a stream-type declaration for two inputs and a function-type declaration for the clock input.

Example 2-4 Defining an Edge-Sensitive Sensitivity List

```
SC_MODULE(my_module){
     // Ports
     sc_in<int> a;
     sc_in<bool> b;
     sc_in<bool> clock;
     sc_out<int> x;
     sc_out<int> y;
     sc_in<bool> reset, set;
     // Internal signals
     sc_signal<bool>c;
     sc_signal<int> d;
     // process declaration
     void my_method_proc();
     // module constructor
     SC_CTOR(my_module) {
          // register process
          SC_METHOD(my_method_proc);
          // declare sensitivity list
          sensitive_pos (clock); //Function delaration
          sensitive_neg << reset << set; // Stream declaration
     }
};
```

Limitations for Sensitivity Lists

When you define a sensitivity list, adhere to the following limitations:

- You cannot specify both edge-sensitive and level-sensitive inputs in the same process for synthesis.
- You cannot declare an sc_logic type for the clock or other edge-sensitive inputs. You can declare only an sc_in<bool> data type.

Member Functions

You can declare member functions in a module that are not processes. This type of member function is not registered as a process in the module’s constructor. It can be called from a process. Member functions can contain any synthesizable C++ or SystemC statement allowed in an SC_METHOD process.

A member function that is not a process can return any synthesizable data type.

Implementing the Module

In the module implementation file, define the functionality of each SC_METHOD process and member function. Example 2-5 shows a minimal implementation file.

Example 2-5 Module Implementation File

```
#include "systemc.h"
#include "my_module.h"
void my_module::my_method_proc() {
   // describe process functionality as C++ code
}
```

Module Constructor

For each module, you need to create a constructor, which is used for synthesis to

- Register processes
- Define a sensitivity list for each SC_METHOD process
- Define optional parameters for the module
- Make optional assignments to data members, which are treated as constants for synthesis

Defining a Constructor With the SC_CTOR Macro

The SC_CTOR macro provides a simple way to define a constructor with a single argument, which is the name of the module. You need to define the SC_CTOR in the header file, not in the implementation file. Within the constructor's body, you register each process for the module. For synthesis, other statements are not allowed in the SC_CTOR constructor.

Example 2-6 shows in bold a constructor defined with an SC_CTOR macro.

Example 2-6 Module Constructor

```
// my_module.h header file
SC_MODULE (my_module) {
     // Declare ports
     sc_in<bool> reset;
     sc_in<sc_int<8> > data_in;
     sc_in_clk clk;
     sc_out<sc_int<16> > real_out;
     sc_out<sc_int<16> > imaginary_out;

     // Declare internal variables and signals

     // Declare processes in the module
     void my_method_proc();

     // Constructor
     SC_CTOR (my_module){
          // Register processes
          ...
          // Define the sensitivity lists
          ...
     }
};
```

Registering a Process

To register a function as a process, use the SC_METHOD macro for an RTL process and the SC_CTHREAD macro for a behavioral process. These macros are defined in the SystemC library.

The SC_METHOD macro takes a single argument, the name of a process to register. In addition, you need to define one or more sensitivity lists for each process.

Example 2-7 shows in bold a module with an SC_CTOR constructor that registers an SC_METHOD process and defines two sensitivity lists for the process.

Example 2-7 Registering a Process and Defining a Sensitivity List

```
SC_MODULE(my_module){
      // Ports
      sc_in<int> a;
      sc_in<bool> b;
      sc_in<bool> clock;
      sc_out<int> x;
      sc_out<int> y;
      sc_in<bool> reset, set;
      // Internal signals
      sc_signal<bool>c;
      sc_signal<int> d;
      // process declaration
      void my_method_proc();
      // module constructor
      SC_CTOR(my_module) {
            // register process
            SC_METHOD(my_method_proc);
            // declare sensitivity lists
            sensitive_pos (clock); //Function delaration
            sensitive_neg << reset << set; // Stream declaration
      }
};
```

Defining a Constructor With the SC_HAS_PROCESS Macro

You can use the SC_HAS_PROCESS macro, introduced in SystemC 2.0, instead of the SC_CTOR macro to define a constructor with standard C++ syntax and any number of parameters. You might want to define a constructor with multiple parameters, for example, to specify values when instantiating the module, to pass a unique identification to a block, or to change the number of iterations performed for a certain algorithm.

Using the SC_HAS_PROCESS macro, you can define the constructor in the header file or in the implementation file. Moving the constructor definition into the implementation lets you hide some of the module's functionality when providing an IP to an end user.

Defining the Constructor Parameters. When you use the SC_HAS_PROCESSES macro to define a constructor, do not define a return type for the constructor. Define the first argument of the constructor as an sc_module_name or a char * type. You need to define the module name parameter even though it is not used for synthesis.

You can then define any number of integral parameter arguments. A parameter can have a default value, which you assign in the constructor. The module receives parameter values when you instantiate it or pass values with the `compile_systemc` command. Inside the module, the parameters are constant values.

Example 2-8 shows a constructor with parameters. The related code is highlighted in bold.

Example 2-8 Module Constructor With Parameters

```
/****parm2.h****/
#include "systemc.h"

SC_MODULE(parm2) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in< sc_uint<8> > data1, data2;
  sc_out< sc_uint<16> > data_out;

  SC_HAS_PROCESS(parm2);

  void mult1();

  bool const_var1;
  sc_uint<9> const_var2;
  // Constructor with parameters without default values
  parm2( const sc_module_name& name_,
         bool const1, sc_uint<9> const2 );
```

```
};
/****parm2.cc****/
#include "parm2.h"
parm2::parm2(const sc_module_name& name_,
             bool const1, sc_uint<9> const2){
  const_var1 = const1;
  const_var2 = const2;
  SC_METHOD(mult1);
  sensitive_pos << clk << reset;
}
void parm2::mult1() {
  if (reset.read() == 1) {
    data_out.write(4);
  } else {
    sc_uint<16> tmp1 = (data1.read() * data2.read());
    sc_uint<10> tmp2 = const_var1 + const_var2;
    sc_uint<16> tmp3 = tmp1 + tmp2;
    data_out.write(tmp3);
  }
}
```

Instantiating a Module With Parameters. When you instantiate a module that has parameters, pass the constructor parameters by position. The passed parameters must be constant values at compile time. Example 2-9 shows instantiation of the *parm1* module in the *use_parms* module. The related code is highlighted in bold.

Example 2-9 Instantiating a Module With Parameters

```
/****use_parms.h****/
#include "systemc.h"
#include "parm2.h"
SC_MODULE(use_parms){
   parm2 *parm;
   parm = new parm2("my_name", 1, 6);
  ...
};
```

You can also pass parameter values to a module from the command line with the `compile_systemc` command `-param` option, as described in “Passing Parameters to a Module” on page 1-23.

Setting and Using Default Parameter Values. It is recommended that you initialize all parameters by assigning default values.

Example 2-10 shows in bold a constructor with two parameters that are assigned default values of 0 and 7. The parameter default values are used unless you instantiate the module with parameter values or you pass parameter values with the `compile_systemc` command `-param` option.

Example 2-10 Module Constructor With Parameter Default Values

```
/****parm2a.h****/
#include "systemc.h"

SC_MODULE(parm2a) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in< sc_uint<8> > data1, data2;
  sc_out< sc_uint<16> > data_out;
  SC_HAS_PROCESS(parm2a);

  void mult1();
  bool const_var1;
  sc_uint<9> const_var2;

  // Parameters with default values
  parm2a( const sc_module_name& name_,
         bool const1 = 0, sc_uint<9> const2 = 7 );
};
/****parm2a.cc****/
#include "parm2a.h"

parm2a::parm2a(const sc_module_name& name_,
             bool const1, sc_uint<9> const2){
  const_var1 = const1;
  const_var2 = const2;
  SC_METHOD(mult1);
  sensitive_pos << clk << reset;
}

void parm2a::mult1() {
  if (reset.read() == 1) {
    data_out.write(4);
  } else {
    sc_uint<16> tmp1 = (data1.read() * data2.read());
    sc_uint<10> tmp2 = const_var1 + const_var2;
    sc_uint<16> tmp3 = tmp1 + tmp2;
```

```
    data_out.write(tmp3);
  }
}
```

If a module defines default parameter values of 0 and 7 as in Example 2-10, when you instantiate the module without parameter values or use the `compile_systemc` command without the `-parm` option, the default values are used.

For example,

```
dc_shell> compile_systemc -rtl -format verilog parm2a.cc
```

This command uses the default values 0 and 7.

When you provide parameter values, it overrides the default values. For example,

```
dc_shell> compile_systemc -rtl -format verilog
      -param "parm2a(1,6);" parm2a.cc
```

This command uses the values 1 and 6.

You can also specify a partial parameter list. Any parameter not specified with the `-param` option uses the default values. For example,

```
dc_shell> compile_systemc -rtl -format verilog
      -param "parm2a(1);" parm2a.cc
```

This command uses the values 1 and 7 because the default value of *const2* is 7.

Parameters are passed by position, and default parameter values are substituted only for the missing *trailing* arguments. Arrange the parameter list so the parameters that are most likely to take user-specified values are specified first.

Initializing in the Constructor. To make it easier to create sensitivity lists, assign to signals, and perform other repetitive initialization tasks, you can include loops in the constructor.

Use simple assignments for initialization. All for loops used in a constructor must be unrollable. Do not use a data member as a loop counter variable.

Creating a Sensitivity List With a Loop. Example 2-11 shows a module that uses loops to create the sensitivity list in two processes.

Example 2-11 Loop to Define Sensitivity List

```
/****sens_loop.h****/
#include "systemc.h"

#define DESIGN sens_loop

SC_MODULE(sens_loop) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in< sc_uint<8> > data[4];
  sc_out< sc_uint<16> > data_out[2];

  SC_HAS_PROCESS(sens_loop);

  sc_signal< sc_uint<8> > int_indata[4];
  sc_signal< sc_uint<16> > int_outdata[2];

  void mult_reg();
  void read_inputs();
  void assign_outputs();

  sens_loop( const sc_module_name& name_,
             int const1 = 4, int const2 = 2);
};
/****sens_loop.cpp****/
#include "sens_loop.h"

sens_loop::sens_loop( const sc_module_name& name_,
                      int const1, int const2) {

  SC_METHOD(mult_reg);
  sensitive_pos << clk;

  SC_METHOD(read_inputs);
```

```
  for(int i = 0; i < const1; i++){
   /* synopsys unroll */
   sensitive << data[i];
  }
  SC_METHOD(assign_outputs);
  for(int i = 0; i < const2; i++) {
      /* synopsys unroll */
    sensitive << int_outdata[i];
  }
}

void sens_loop::mult_reg() {
  sc_uint<16> tmp1 = data[0].read() * data[1].read();
  sc_uint<16> tmp2 = data[2].read() * data[3].read();
  int_outdata[0] = tmp1;
  int_outdata[1] = tmp2;
}

void sens_loop::read_inputs() {
  for(int i =0; i < const1; i++){
      /* synopsys unroll */
    int_indata[i] = data[i].read();
  }
}

void sens_loop::assign_outputs() {
  for(int i = 0; i< const2; i++){
      /* synopsys unroll */
    data_out[i].write(int_outdata[i].read());
  }
}
#include "systemc.h"

#define DESIGN sens_loop

SC_MODULE(sens_loop) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in< sc_uint<8> > data[4];
  sc_out< sc_uint<16> > data_out[2];

  SC_HAS_PROCESS(sens_loop);

  sc_signal< sc_uint<8> > int_indata[4];
  sc_signal< sc_uint<16> > int_outdata[2];

  void mult_reg();
  void read_inputs();
  void assign_outputs();

  sens_loop( const sc_module_name& name_,
             int const1 = 4, int const2 = 2);
};
```

```
#include "sens_loop.h"

sens_loop::sens_loop( const sc_module_name& name_,
                      int const1, int const2) {

  SC_METHOD(mult_reg);
  sensitive_pos << clk;

  SC_METHOD(read_inputs);
  for(int i = 0; i < const1; i++){
   /* synopsys unroll */
   sensitive << data[i];
  }
  SC_METHOD(assign_outputs);
  for(int i = 0; i < const2; i++) {
      /* synopsys unroll */
    sensitive << int_outdata[i];
  }
}

void sens_loop::mult_reg() {
  sc_uint<16> tmp1 = data[0].read() * data[1].read();
  sc_uint<16> tmp2 = data[2].read() * data[3].read();
  int_outdata[0] = tmp1;
  int_outdata[1] = tmp2;
}

void sens_loop::read_inputs() {
  for(int i =0; i < const1; i++){
      /* synopsys unroll */
    int_indata[i] = data[i].read();
  }
}

void sens_loop::assign_outputs() {
  for(int i = 0; i< const2; i++){
      /* synopsys unroll */
    data_out[i].write(int_outdata[i].read());
  }
}
```

Instantiating Multiple Modules With a Loop. Example 2-12 defines an asynchronous reset D flip-flop. The has_loop_inst module instantiates back-to-back pairs of the D flip-flop, which are typically called synchronizers. The TMP signal is an intermediate node that connects between each flip-flop pair.

This example uses the NUM_INSTS macro to set the parameter that defines the number of pairs to instantiate and the width of the IN and OUT ports. This value is set to a default of 4.

Example 2-12 Instantiating Multiple Modules With a Loop

```
/****loopinst.h****/
#include "systemc.h"

// This module defines a typical asynchronous
// reset D flip-flop, which is sensitive to
// the positive clock edge.

SC_MODULE(dff_pos_module) {
  sc_in<bool>in_data;
  sc_out<bool> out_q;
  sc_in_clkclock;
  sc_in<bool>reset;

  void dff_pos_function() {
      if (reset){
           out_q = 0;
      }else{
          out_q = in_data;
      }
  }

  SC_CTOR(dff_pos_module) {
     SC_METHOD(dff_pos_function);
     sensitive_pos << clock << reset;
  }
};

// This loop instantiates two-level
// (back-to-back) D flip-flops, which
// is often called synthronizers.
// The signal TMP is an intermediate node that
// connects between the two D flip-flops.

SC_MODULE(has_loop_inst) {

   // By default, the number of instances and
   // the IN and OUT port widths are set to 4.
   // Specify a different number of instances
   // in the synthesis script or the Makefile.
   #ifndef NUM_INSTS
      #define NUM_INSTS 4
   #endif

    sc_in<bool>   clock;
    sc_in<bool>   reset;
```

```
    sc_in<bool>   IN [NUM_INSTS];
    sc_out<bool>  OUT [NUM_INSTS];

    SC_HAS_PROCESS(has_loop_inst);

    sc_signal<bool> TMP [NUM_INSTS];

    dff_pos_module   *dff_instance_a [NUM_INSTS];
    dff_pos_module   *dff_instance_b [NUM_INSTS];

    has_loop_inst(const sc_module_name& name_,
                  int const1 = NUM_INSTS);
};
/****loopinst.cc****/
#include "systemc.h"
#include "loopinst.h"

has_loop_inst::has_loop_inst
       (const sc_module_name& name_,
        int const1) {

  #ifndef SYN
     char *dffname_a [NUM_INSTS] =
           {"dff1a", "dff2a", "dff3a", "dff4a"};
     char *dffname_b [NUM_INSTS] =
           {"dff1b", "dff2b", "dff3b", "dff4b"};
  #endif

  char *name1, *name2;

  for (int i=0; i<const1; i++) {  // snps unroll
    #ifndef SYN
       name1 = dffname_a [i];
       name2 = dffname_b [i];
    #endif

    // Instantiate the first column of D flip-flops.
    dff_instance_a [i] = new dff_pos_module(name1);
    (*dff_instance_a [i]) (IN[i], TMP[i], clock, reset);

    // Instantiate the second column of D flip-flops.
    dff_instance_b [i] = new dff_pos_module(name2);
    (*dff_instance_b [i]) (TMP[i], OUT[i], clock, reset);
  }
}
```

Limitation of Using Constructor Parameters. Because of C++ restrictions, you cannot define a constructor parameter to

- Specify the bit-width of data types
- Change anything defined outside the constructor, such as memory size, the number of ports, or function prototypes

SystemC Compiler has the following restrictions for using a constructor defined with the SC_HAS_PROCESSES macro:

- A parameter cannot be a struct type.
- Use only simple data member assignment statements in the constructor.
- Use only unrollable for loops.

Reading and Writing Ports and Signals

In the module implementation description, you can read from or write to a port or a signal by using the read and write methods or by assignment. An sc_out port and an sc_inout port have read() and write() methods to allow you to read from or write to the port. An sc_in port has only a read() method.

When you read from or write to a port or a signal, a recommended coding practice is to use the read() and write() methods to distinguish port and signals from variable assignments. The read() and write() methods perform any necessary data conversion. Use the assignment operator for variables. Example 2-13 shows in bold how to use the read and write methods for ports and signals, and it shows assignment operators for variables.

Example 2-13 Using Assignment and read() and write() Methods

```
// read method
address = into.read();      // get address
// assignment
temp1 = address;                  // save address
data_tmp = memory[address];  // get data from memory
// write method
outof.write(data_tmp);     // write out
// assignment
temp2 = data_tmp;
// save data_tmp
//...
```

Reading and Writing Bits of Ports and Signals

You read or write all bits of a port or signal. You cannot read or write the individual bits, regardless of the type. To do a bit-select on a port or signal, read the value into a temporary variable and do a bit-select on the temporary variable. Example 2-14 shows in bold how to read from or write bits to a temporary variable.

Example 2-14 Reading and Writing Bits of a Variable

```
//...
sc_signal <sc_int<8> > a;
sc_int<8> b;
bool c;
b = a.read();
c = b[0];

// c = a[0]; // Will not work in SystemC
```

Example 2-14 reads the value of signal a into temporary variable b and writes bit 0 of b into variable c. You cannot read a bit from signal a, because this operation is not allowed in SystemC.

Signal and Port Assignments

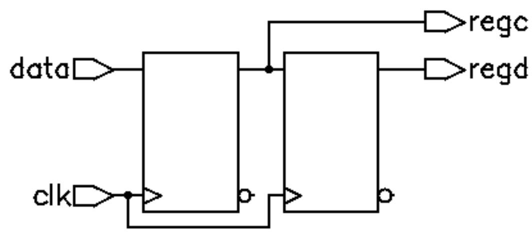
When you assign a value to a signal or a port, the value on the right side of the assignment statement is not transferred to the left side until the next simulation delta cycle (see the SystemC documentation for SystemC simulation semantics). This means the signal values seen by other processes are not updated immediately, but deferred.

Example 2-15 shows a serial register implementation with signal assignment, and Figure 2-4 shows the resulting schematic.

Example 2-15 Signal Assignment

```
#include "systemc.h"

SC_MODULE(rtl_nb) {
  sc_in<bool> clk;
  sc_in<bool> data;
  sc_inout<bool> regc, regd;

  void reg_proc() {
    regc.write(data.read());
    regd.write(regc.read());
  }

  SC_CTOR(rtl_nb) {
    SC_METHOD(reg_proc);
    sensitive_pos << clk;
  }
};
```

Figure 2-4 Signal Assignment Schematic



Variable Assignment

When you assign a value to a variable, SystemC Compiler considers that the value on the right side is transferred immediately to the left side of the assignment statement.

Example 2-16 includes a variable assignment, in which the implementation assigns the value of data to rega and regb, as the resulting schematic in Figure 2-5 indicates.

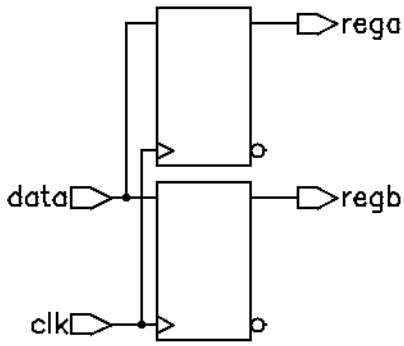
Note:

> This example is only an illustration of variable assignment. You can write the same behavior more efficiently by removing the rega_v and regb_v variables and writing the ports directly.

Example 2-16 Variable Assignment

```
#include "systemc.h"

SC_MODULE(rtl_b) {
  sc_in<bool> clk;
  sc_in<bool> data;
  sc_out<bool> rega, regb;

  bool rega_v, regb_v;

  void reg_proc() {
    rega_v = data.read();
    regb_v = rega_v;
    rega.write(rega_v);
    regb.write(regb_v);
  }

  SC_CTOR(rtl_b) {
    SC_METHOD(reg_proc);
    sensitive_pos << clk;
  }
};
```

Figure 2-5   Variable Assignment Schematic



Creating a Module With a Single SC_METHOD Process

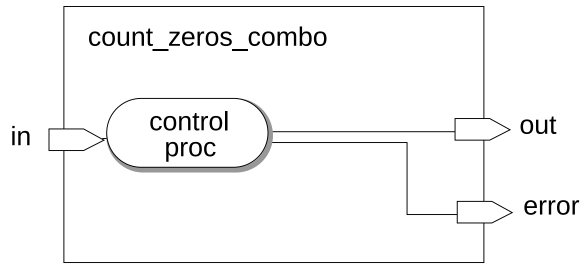
Example 2-17 is an RTL description of a count zeros circuit that contains one SC_METHOD process, control_proc( ), and two member functions, legal( ) and zeros( ). The circuit determines in one cycle if an 8-bit value on the input port is valid (having no more than one sequence of zeros) and how many zeros the value contains. The circuit produces two outputs, the number of zeros found and an error indication. Figure 2-6 illustrates the module and its ports. The design description and the complete set of files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_examples.

Example 2-17   Count Zeros Combinational Version

```
/****count_zeros_comb.h file***/
#include "systemc.h"

SC_MODULE(count_zeros_comb) {
  sc_in<sc_uint<8> >  in;
  sc_out<sc_uint<4> >  out;
  sc_out<bool> error;

  bool legal(sc_uint<8> x);
  sc_uint<4> zeros(sc_uint<8> x);
  void control_proc();

  SC_CTOR(count_zeros_comb) {
```

```
    SC_METHOD(control_proc);
    sensitive << in;
  }
};

/****count_zeros_comb.cpp file****/
#include "count_zeros_comb.h"

void count_zeros_comb::control_proc() {
  sc_uint<4> tmp_out;
  bool is_legal = legal(in.read());
  error.write(is_legal != 1);
  is_legal ? tmp_out = zeros(in.read()) : tmp_out = 0;
  out.write(tmp_out);
}

bool count_zeros_comb::legal(sc_uint<8> x) {
  bool is_legal = 1;
  bool seenZero = 0;
  bool seenTrailing = 0;
  for (int i=0; i <=7; ++i) {
    if ((seenTrailing == 1) && (x[i] == 0)) {
      is_legal = 0;
      break;
    } else if ((seenZero == 1) && (x[i] == 1)) {
      seenTrailing = 1;
    } else if (x[i] == 0) {
      seenZero = 1;
    }
  }
  return is_legal;
}

sc_uint<4> count_zeros_comb::zeros(sc_uint<8> x) {
  int count = 0;
  for (int i=0; i <= 7; ++i) {
    if (x[i] == 0)
      ++count;
  }
  return count;
}
```

Figure 2-6   Count Zeros Combinational Module



To synthesize a design similar to this example, use the commands in "Synthesizing a SystemC Design in a Single File" on page 1-9.

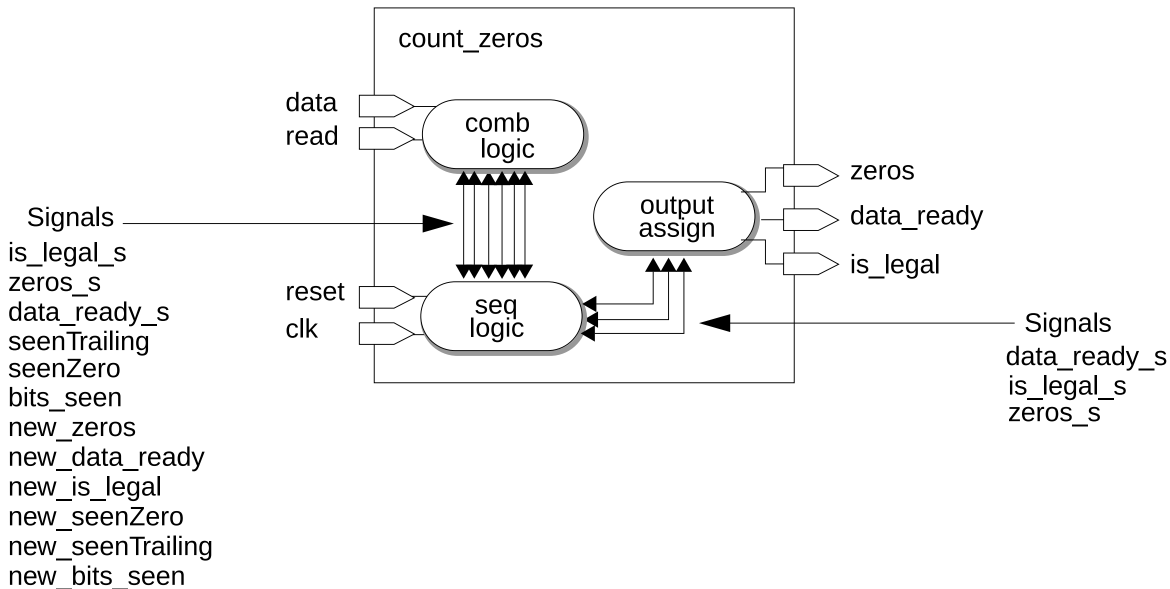
Creating a Module With Multiple SC_METHOD Processes

Example 2-18 is a sequential description of the count zeros circuit described in "Creating a Module With a Single SC_METHOD Process" on page 2-32. The complete set of files is available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_examples.

In this sequential version, there are three SC_METHOD processes and several signals for communication between the processes, as shown in Figure 2-7. The comb_logic( ) and output_assign( ) processes are level-sensitive, and the seq_logic( ) process is sensitive to the positive edge of the clk and reset inputs. The set_defaults( ) member function is called at the beginning of the comb_logic( ) process.

This example does not show typical simulation-specific code you might include for debugging purposes.

Figure 2-7 Count Zeros Sequential Module



Example 2-18 Count Zeros Sequential Version

```
/****count_zeros_seq.h file****/
#include "systemc.h"

#define ZEROS_WIDTH 4
#define MAX_BIT_READ 7

SC_MODULE(count_zeros_seq) {
  sc_in<bool> data, reset, read, clk;
  sc_out<bool> is_legal, data_ready;
  sc_out<sc_uint<ZEROS_WIDTH> > zeros;

  sc_signal<bool> new_data_ready, new_is_legal, new_seenZero, new_seenTrailing;
  sc_signal<bool> seenZero, seenTrailing;
  sc_signal<bool> is_legal_s, data_ready_s;
  sc_signal<sc_uint<ZEROS_WIDTH> > new_zeros, zeros_s;
  sc_signal<sc_uint<ZEROS_WIDTH - 1> > bits_seen, new_bits_seen;

  // Processes
  void comb_logic();
  void seq_logic();
  void assign_outputs();

  // Helper functions
  void set_defaults();
```

```
  SC_CTOR(count_zeros_seq) {
    SC_METHOD(comb_logic);
    sensitive << data << read << is_legal_s << data_ready_s;
    sensitive << seenTrailing << seenZero << zeros_s << bits_seen;

    SC_METHOD(seq_logic);
    sensitive_pos << clk << reset;

    SC_METHOD(assign_outputs);
    sensitive << is_legal_s << data_ready_s << zeros_s;
  }
};


/****count_zeros_seq.cpp file****/
#include "count_zeros_seq.h"

/*
 *  SC_METHOD: comb_logic()
 *    finds a singular run of zeros and counts them
 */
void count_zeros_seq::comb_logic() {
  set_defaults();
  if (read.read()) {
    if (seenTrailing && (data.read() == 0)) {
      new_is_legal = false;
      new_zeros = 0;
      new_data_ready = true;
    } else if (seenZero && (data.read() == 1)) {
      new_seenTrailing = true;
    } else if (data.read() == 0) {
      new_seenZero = true;
      new_zeros = zeros_s.read() + 1;
    }


    if (bits_seen.read() == MAX_BIT_READ){
      new_data_ready = true;
    }else{
      new_bits_seen = bits_seen.read() + 1;
    }
  }
}

/*
 *  SC_METHOD: seq_logic()
 *    All registers have asynchronous resets
```

```
 */
void count_zeros_seq::seq_logic() {
  if (reset) {
    zeros_s = 0;
    bits_seen = 0;
    seenZero = false;
    seenTrailing = false;
    is_legal_s = true;
    data_ready_s = false;
  } else {
    zeros_s = new_zeros;
    bits_seen = new_bits_seen;
    seenZero = new_seenZero;
    seenTrailing = new_seenTrailing;
    is_legal_s = new_is_legal;
    data_ready_s = new_data_ready;
  }
}

/*
 *  SC_METHOD: assign_outputs()
 *  Zero time assignments of signals to their associated outputs
 */
void count_zeros_seq::assign_outputs() {
  zeros = zeros_s;
  is_legal = is_legal_s;
  data_ready = data_ready_s;
}

/*
 *   method: set_defaults()
 *     sets the default values of the new_* signals for the comb_logic
 *     process.
 */
void count_zeros_seq::set_defaults() {
  new_is_legal = is_legal_s;
  new_seenZero = seenZero;
  new_seenTrailing = seenTrailing;
  new_zeros = zeros_s;
  new_bits_seen = bits_seen;
  new_data_ready = data_ready_s;
}
```

To synthesize a design similar to this example, use the commands in "Synthesizing a SystemC Design in a Single File" on page 1-9.

Creating a Hierarchical RTL Module

You can create a hierarchical module with multiple instantiated modules. The lower-level modules can contain either SC_METHOD processes or an SC_CTHREAD behavioral process. The design description and the complete set of files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_examples.

The Basics of Hierarchical Module Creation

To create a hierarchical module,

1. Create data members in the top-level module that are pointers to the instantiated modules.
2. Allocate the instantiated modules inside the constructor of the top-level module, giving each instance a unique name.
3. Bind the ports of the instantiated modules to the ports or signals of the top-level module. Use either binding by position or binding by name coding style.

Example 2-19 shows the partial source code of two modules, fir_fsm and fir_data, instantiated in the fir_top module. The relevant code is highlighted in bold.

Example 2-19 Hierarchical Module With Multiple RTL Modules

```
/****fir_top.h****/
#include <systemc.h>
#include "fir_fsm.h"
#include "fir_data.h"

SC_MODULE(fir_top) {

  sc_in_clk          CLK;
  sc_in<bool>        RESET;
  sc_in<bool>        IN_VALID;
  sc_in<int>         SAMPLE;
```

```
  sc_out<bool>      OUTPUT_DATA_READY;
  sc_out<int>       RESULT;

  sc_signal<unsigned> state_out; //Communication between
                                    //two peer modules

  // Create data members - pointers to instantiated
  // modules
  fir_fsm  *fir_fsm1;
  fir_data *fir_data1;
  SC_CTOR(fir_top) {
     // Create new instance of fir_fsm module
      fir_fsm1 = new fir_fsm("FirFSM");

      // Binding by name
      fir_fsm1->clock(CLK);
      fir_fsm1->reset(RESET);
      fir_fsm1->in_valid(IN_VALID);
      fir_fsm1->state_out(state_out);

     // Binding by position alternative
     //fir_fsm1 (CLK, RESET, IN_VALID, state_out);

     // Create new instance
     // of fir_data module and bind by name
      fir_data1 = new fir_data("FirData");
      fir_data1->reset(RESET);
      fir_data1->state_out(state_out);
      fir_data1->sample(SAMPLE);
      fir_data1->result(RESULT);
      fir_data1->output_data_ready(OUTPUT_DATA_READY);
      fir_data1->clk(CLK);
        ...
    }
};

/****fir_fsm.h****/
SC_MODULE(fir_fsm) {

  sc_in<bool>      clock;
  sc_in<bool>      reset;
  sc_in<bool>      in_valid;
  sc_out<unsigned> state_out;
     ...

/****fir_data.h****/
SC_MODULE(fir_data) {

  sc_in<bool>      clk;
  sc_in<bool>      reset;
  sc_in<unsigned>  state_out;
  sc_in<int>       sample;
  sc_out<int>      result;
```

```
  sc_out<bool>     output_data_ready;
     ...
```

Creating an Integrated RTL and Behavioral Module

Creating an integrated RTL and behavioral module is similar to creating a hierarchical RTL module. Example 2-20 shows an integrated module, all_top, that contains an instance of the hierarchical RTL fir_rtl module in Example 2-19 on page 2-38 and an instance of a behavioral version fir_beh of the FIR filter shown in Example 2-21 on page 2-41. The design description and complete set of files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_examples.

Example 2-20 FIR Top-Level Integrated Module

```
/****all_top.h file****/

#include <systemc.h>

#include "fir_rtl.h"
#include "fir_beh.h"

SC_MODULE(all_top) {

  sc_in<bool>       reset;
  sc_in<bool>       input_valid;

  sc_in<int>        sample;

  sc_out<int>       sample_out_rtl;
  sc_out<bool>      output_ready_rtl;
  sc_out<int>       sample_out_syn;
  sc_out<bool>      output_ready_syn;

  sc_in<bool>       clk1;

 // Instantiates RTL and behavioral models
  fir_rtl *fir_rtl1;
  fir_beh *fir_beh1;

  SC_CTOR(all_top) {

      fir_rtl1 = new fir_rtl("firTOP");
```

```
        fir_rtl1->reset(reset);
        fir_rtl1->in_valid(input_valid);
        fir_rtl1->sample(sample);
        fir_rtl1->result(sample_out_rtl);
        fir_rtl1->output_data_ready(output_ready_rtl);
        fir_rtl1->clk(clk1);

        fir_beh1 = new fir_beh("FIR");
        fir_beh1->reset(reset);
        fir_beh1->input_valid(input_valid);
        fir_beh1->sample(sample);
        fir_beh1->result(sample_out_syn);
        fir_beh1->output_data_ready(output_ready_syn);
        fir_beh1->CLK(clk1);
        };
}
```

Example 2-21 FIR Behavioral Module

```
/****fir_beh.h file****/

C_MODULE(fir_beh) {
  sc_in<bool>  reset;
  sc_in<bool>  input_valid;
  sc_in<int>   sample;
  sc_out<bool> output_data_ready;
  sc_out<int>  result;
  sc_in_clk    CLK;

  SC_CTOR(fir_beh){
      SC_CTHREAD(entry, CLK.pos());
      watching(reset.delayed() == true);
  }
  void entry();
};


/****fir_beh.cpp file****/

include <systemc.h>
#include "fir_beh.h"
#include "fir_const.h"

void fir_beh::entry() {

  sc_int<8>  sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];
```

```
    // reset watching
    for (int i=0; i<=15; i++){
      // synopsys unroll
      shift[i] = 0;
    }
    result.write(0);
    output_data_ready.write(false);
    wait();

    // main functionality
    fir_loop:while(1) {

      output_data_ready.write(false);
      wait_until(input_valid.delayed() == true);

      sample_tmp = sample.read();
      acc = sample_tmp*coefs[0];

      for(int i=14; i>=0; i--) {
        // synopsys unroll
        acc += shift[i]*coefs[i+1];
      }


      for(int i=14; i>=0; i--) {
        // synopsys unroll
        shift[i+1] = shift[i];
      }
      shift[0] = sample_tmp;
      // write output values
      result.write(acc);
      output_data_ready.write(true);
      wait();
    }
  }
```

To synthesize a design similar to this example, use the commands in “Synthesizing a Design With Integrated Behavioral and RTL Modules” on page 1-22.

Specifying Preserved Functions and Implementing DesignWare Components

Functions increase the readability of your source code. By default, SystemC Compiler inlines functions, which makes the HDL created by SystemC Compiler difficult to understand. To improve the readability of the generated HDL, you can direct SystemC Compiler to preserve a function instead of creating inline code. Or you can direct SystemC Compiler to map a function to a synthetic library operator to be implemented by a DesignWare component.

Note:

> SystemC Compiler version U-2003.06 supports these features only for the RTL Verilog flow, not the VHDL flow.

Defining a Preserved Function

To preserve a function, insert the `preserve_function` compiler directive in your code as the first line in the function body. Example 2-22 shows a preserved function that passes two input parameters and returns a single output.

Example 2-22 Preserved Function With a Single Output Return Value

```
...
sc_uint<8> gpf_modulus(sc_uint<8> A, sc_uint<8> B){
  // synopsys preserve_function
  return A % B;
}
```

The function parameters and return types can be any synthesizable type (Table 3-3 on page 3-11) or a struct of synthesizable types. You can read and write module variables, ports, and signals without passing them as function parameters.

You can define one or more outputs for a preserved function. Depending on your design requirements, you can return one output and pass the other outputs as nonconstant references (Example 2-23), or pass all outputs as nonconstant references with a void return (Example 2-24).

Example 2-23 Preserved Function With a Return Value and a Passed Reference

```
...
sc_int<8> foo (sc_int<8> a, sc_int<8> b,
                          sc_int<8> &div_num) {
  // synopsys preserve_function

  sc_int<8> mod_num;

  if (b==0) {
    if (a>0)
      div_num = 127;
    else
      div_num = -128;
    mod_num = a;
  }
  else {
    div_num = a/b;
    mod_num = a%b;
  }
  return (mod_num);
}
```

Example 2-24 Preserved Function With a Void Return and Multiple Passed References

```
...
void data::foo (sc_int<8> a, sc_int<8> b,
                        sc_int<8> &mod_num,
                        sc_int<8> &div_num) {
  // synopsys preserve_function

  if (b==0) {
    if (a>0)
      div_num = 127;
    else
      div_num = -128;
    mod_num = a;
  }
  else {
    div_num = a/b;
```

```
    mod_num = a%b;
  }
}
```

When you use reference parameters, you need to ensure that you are not creating an alias by mistake. You create an alias by passing the same object by reference to different parameters. An alias between two outputs creates a short circuit between the outputs.

For example, this problem occurs in the following:

```
//Definition
void abc(int a, const int& b, int& c) {
   /* synopsys map_to_operator XXX_OP */
   ...
}

void xyz () {
   //function call that causes alias
   abc(x, y, y);
   ...
}
```

In the above example, parameters b and c are bound to the same y variable, causing an error.

Another more subtle alias can result from the following function call:

```
abc(x, a[i], a[j]);
```

In the above function call, a potential alias occurs, based on the value of i and j. In such a situation, use a temporary variable to avoid the problem; for example,

```
abc(x, a[i], temp);
a[j] = temp;
```

Verilog HDL From a Preserved Function

When you execute the `compile_systemc` command with the `-output verilog` option for a design with a preserved function, SystemC Compiler creates a Verilog output parameter for a nonvoid return type and any nonconstant reference parameters. It creates Verilog input parameters for constant reference parameters and all other parameters. Inout parameters are not supported.

SystemC Compiler creates a Verilog function or task for a preserved function. It creates a Verilog function if the preserved function meets the following criteria:

- Has a single output returned by the function and no other output parameters
- Is used in an expression
- Has at least one input parameter
- Does not read from or write to a signal or port
- Does not call another function that fails to meet the above criteria

Example 2-25 shows the Verilog HDL created by SystemC Compiler from the SystemC code in Example 2-22. Because the function meets the above criteria, a Verilog function is created.

Example 2-25 Verilog HDL Function From a Preserved Function

```
...
function [7:0] gpf_modulus;
    input [7:0] A;
    input [7:0] B;
    begin
        gpf_modulus = A % B;
    end
    endfunction
```

Otherwise, SystemC Compiler creates a Verilog task for the preserved function.

Example 2-26 shows the Verilog HDL created by SystemC Compiler from the SystemC code in Example 2-23. Because the function has two outputs (a return and a nonconstant reference), a task is created.

Example 2-26 Verilog HDL Task From a Preserved Function

```
...
task foo;
    output signed [7:0] foo_RETURN_PORT;
    input signed [7:0] a;
    input signed [7:0] b;
    output signed [7:0] div_num;
    reg signed [7:0] mod_num;
    reg signed [7:0] __tmp181;
    begin
        if (b == 8'sb00000000)
        begin
            if (a > 8'sb00000000)
                div_num = 8'sb01111111;
            else
                div_num = -128;
            mod_num = a;
        end
        else
        begin
            __tmp181 = a / b;
            div_num = __tmp181;
            mod_num = a % b;
        end
        foo_RETURN_PORT = mod_num;
    end
    endtask
```

Mapping a Function to a Synthetic Operator

You can direct SystemC Compiler to map a function to a synthetic library operator to be implemented by a DesignWare component. To map to a function to a specified synthetic operator, insert the `map_to_operator` compiler directive as the first line in the function body.

Example 2-27 shows code that instructs SystemC Compiler to use a square root (SQRT_TC_OP) synthetic operator for synthesis. In this example, the SQRT_TC_OP operator has an input port A and returns the output on the default return port named Z.

Example 2-27 Specifying a Synthetic Operator

```
...
sc_uint<8> example::dw_sqrt(sc_int<16> A ) {
  // synopsys map_to_operator SQRT_TC_OP

#ifdef SIM
  double temp_d;
  sc_uint<8> temp;

  temp_d = sqrt(fabs(double(A)));
  root = temp_d;
#endif
}
```

You do not need to describe the component's functionality for synthesis. After you execute the SystemC Compiler `compile_systemc` command, this function is replaced by the SQRT_TC_OP operator, provided that it exists in a synthetic library specified in your synthetic library path.

The function body is ignored for synthesis. For simulation, you can describe the functionality and enclose it in #ifdef and #endif directives to indicate the code is excluded for synthesis.

Your function inputs and outputs and their names must match the synthetic operator ports, which are case-sensitive. Use the `report_synlib` command to generate a synthetic library report showing the synthetic operator ports and their names.

The function parameters and return types can be any synthesizable type (see Table 3-3 on page 3-11).

You can define more than one output for a DesignWare component. For example, you can return one output and pass the other outputs as nonconstant references, as shown in Example 2-28.

Example 2-28 A map_to_operator Function With a Return and a Reference Parameter

```
...
sc_int<8>  data::foo (sc_int<8> A, sc_int<8> B,
                      sc_int<8> &QUOTIENT) {

  // synopsys map_to_operator DIV_TC_OP
  // synopsys return_port_name REMAINDER

  sc_int<8> mod_num;
  if (B==0) {
    if (A>0)
      QUOTIENT = 127;
    else
      QUOTIENT = -128;
    mod_num = A;
  }
  else {
    QUOTIENT = A/B;
    mod_num = A%B;
  }
  return mod_num;
}
```

You can also pass all outputs as nonconstant references with a void return, as shown in Example 2-29.

Example 2-29 A map_to_operator Function With Multiple Reference Parameters

```
void data::foo (sc_int<8> A, sc_int<8> B,
            sc_int<8> &REMAINDER,
            sc_int<8> &QUOTIENT) {

    // synopsys map_to_operator DIV_TC_OP

  if (B==0) {
    if (A>0)
      QUOTIENT = 127;
    else
      QUOTIENT = -128;
    REMAINDER = A;
```

```
  }
  else {
    QUOTIENT = A/B;
    REMAINDER = A%B;
  }
}
```

When mapping an output port with a nonvoid return, use the `return_port_name` compiler directive to specify the output port you want returned by the function, as shown in Example 2-28. Otherwise, it returns the default return port named Z.

Verilog HDL From a Function Mapped to a DesignWare Component

When you use the `compile_systemc` command with the `-output verilog` option, the function body is converted to Verilog, but it is ignored for synthesis.

SystemC Compiler creates a Verilog function or task for a function that is mapped to a DesignWare component, using the same criteria described in “Verilog HDL From a Preserved Function” on page 2-46.

3

Using the Synthesizable Subset

This chapter explains the subsets of the SystemC and C/C++ language elements and data types that are used for RTL synthesis with SystemC Compiler. It contains the following sections:

Converting to a Synthesizable Subset

To prepare for synthesis, you need to convert all nonsynthesizable code into synthesizable code. This is required only for functionality that is to be synthesized, and not for the testbench or the software part of the system.

Although you can use any SystemC class or C++ construct for simulation and other stages of the design process, only a subset of the language can be used for synthesis. SystemC Compiler does not recognize nonsynthesizable constructs, and it displays an error message if it encounters any of these constructs in your code. You can use #ifdef and #endif to comment out code that is needed only for simulation. For example, you can exclude trace and print statements with these compiler directives.

Excluding Nonsynthesizable Code

SystemC Compiler provides compiler directives you can use in your code

- To include synthesis-specific directives
- To exclude or comment out nonsynthesizable and simulation-specific code so it does not interfere with synthesis

You can isolate nonsynthesizable code or simulation-specific code with a compiler directive, either the C language #ifdef and #endif (recommended) or a comment starting with the words `synopsys` and `synthesis_off`. Example 3-1 shows compiler directives in bold that exclude simulation code for simulation or synthesis.

Example 3-1 Excluding Simulation-Only and Synthesis-Only Code

```
//C directive  (recommended style)
 #ifdef SIM
 ...//Simulation-only code
 #endif

//SystemC Compiler directive
//(using #ifdef instead is recommended)
 /* synopsys synthesis_off */
 ... //Simulation-only code
 /* synopsys synthesis_on */
```

For this example, if the symbol SIM is defined, the additional code is compiled with the intent of doing a simulation.

You can define the SIM symbol with a #define directive, or you can provide it in the compiler command line for simulation purposes.

SystemC and C++ Synthesizable Subsets

The synthesizable subsets of SystemC and C++ are provided in the sections that follow. Wherever possible, a recommended corrective action is indicated for converting nonsynthesizable constructs into synthesizable constructs. For many nonsynthesizable constructs, there is no obvious recommendation for converting them into synthesizable constructs or there are numerous ways to convert them. In such cases, a recommended corrective action is not indicated. Familiarize yourself with the synthesizable subset, and use it as much as possible in your pure C/C++ or high-level SystemC models to minimize the modification effort for synthesis.

You can use any SystemC or C++ construct for a testbench. You do not need to restrict your code to the synthesizable subset in the testbench.

Nonsynthesizable SystemC Constructs

SystemC Compiler does not support the SystemC constructs listed in Table 3-1 for RTL synthesis.

Table 3-1 Nonsynthesizable SystemC Constructs for RTL Synthesis

Category	Construct	Comment	Corrective action
Thread process	SC_THREAD	Used for modeling a testbench, simulation, and modeling at the behavioral level.	
CTHREAD process	SC_CTHREAD	Used for simulation and modeling at the behavioral level.	
Main function	sc_main()	Used for simulation.	
Clock generation	sc_start()	Used for simulation.	Use only in sc_main( ).
Communication	sc_interface, sc_port, sc_mutex, sc_fifo	Used for modeling communication.	Comment out for synthesis.
Global watching	watching()	Not supported for RTL synthesis.	
Local watching	W_BEGIN, W_END, W_DO, W_ESCAPE	Not supported.	
Synchronization	Master-slave library of SystemC	Used for synchronization of events.	Comment out for synthesis.
Tracing	sc_trace, sc_create* trace_file	Creates waveforms of signals, channels, and variables for simulation.	Comment out for synthesis.

Nonsynthesizable C/C++ Constructs

SystemC Compiler does not support the C and C++ constructs listed in Table 3-2 for RTL synthesis.

Table 3-2 Nonsynthesizable C/C++ Constructs

Category	Construct	Comment	Corrective action
Local class declaration		Not allowed.	Replace.
Nested class declaration		Not allowed.	Replace.
Derived class		Only SystemC modules and processes are supported.	Replace.
Dynamic storage allocation	malloc(), free(), new, new[], delete, delete[]	malloc(), free(), new, new[], delete, and delete[] are not supported. The new construct is allowed only to instantiate a module to create hierarchy.	Use static memory allocation.
Exception handling	try, catch, throw	Not allowed.	Comment out.
Recursive function call		Not allowed.	Replace with iteration.
Function overloading		Not allowed (except the classes overloaded by SystemC).	Replace with unique function calls.
C++ built-in functions		Math library, I/O library, file I/O, and similar built-in C++ functions not allowed.	Replace with synthesizable functions or remove.
Virtual function		Not allowed.	Replace with a nonvirtual function.

Table 3-2 Nonsynthesizable C/C++ Constructs (Continued)

Category	Construct	Comment	Corrective action
Inheritance		Not allowed.	Create an independent SC_MODULE.
Multiple inheritance		Not allowed.	Create independent modules.
Member access control specifiers	public, protected, private, friend	Allowed in code but ignored for synthesis. All member access is public.	
Accessing struct members with the (->) operator	-> operator	Not allowed, except for module instantiation.	Replace with access using the period (.) operator.
Static member		Not allowed.	Replace with nonstatic member variable.
Dereference operator	* and & operators	Not allowed.	Replace dereferencing with array accessing.
for loop comma operator	, operator	The comma operator is not allowed in a for loop definition.	Remove the comma operators.
Unbounded loop		Not allowed.	Replace with a bounded loop, such as a for loop.
Out-of-bound array access		Not allowed.	Replace with in-bound array access.
Operator overloading		Not allowed (except the classes overloaded by SystemC).	Replace overloading with unique function calls.

Table 3-2 Nonsynthesizable C/C++ Constructs (Continued)

Category	Construct	Comment	Corrective action
Operator, sizeof	sizeof	Not allowed.	Determine size statically for use in synthesis.
Pointer	*	Pointers are allowed only in hierarchical modules to instantiate other modules.	Replace all other pointers with access to array elements or individual elements.
Pointer type conversions		Not allowed.	Do not use pointers. Use explicit variable reference.
this pointer	this	Not allowed.	Replace.
Reference, C++	&	Allowed only for passing parameters to functions.	Replace in all other cases.
Reference conversion		Reference conversion is supported for implicit conversion of signals only.	Replace in all other cases.
Static variable		Not allowed in functions.	
User-defined template class		Only SystemC templates classes such as sc_int<> are supported.	Replace.
Explicit user-defined type conversion		The C++ built-in types and SystemC types are supported only for explicit conversion.	Replace in all other cases.
Type casting at runtime		Not allowed.	Replace.
Type identification at runtime		Not allowed.	Replace.
Unconditional branching	goto	Not allowed.	Replace.

Table 3-2 Nonsynthesizable C/C++ Constructs (Continued)

Category	Construct	Comment	Corrective action
Unions		Not allowed.	Replace with structs.
Global variable		Not supported for synthesis.	Replace with local variables.
Member variable		Member variables accessed by two or more SC_METHOD processes are not supported. However, access to member variables by only one process is supported.	Use signals instead of variables for communication between processes.
Volatile variable		Not allowed.	Use only nonvolatile variables.

Modifying Data for Synthesis

A pure C/C++ model or a high-level SystemC model typically uses native C++ types or aggregates (structures) of such types. Native C++ types such as int, char, bool, and long have fixed, platform-dependent widths, which are often not the correct width for efficient hardware. For example, you might need only a 6-bit integer for a particular operation, instead of the native C++ 32-bit integer. In addition, C++ does not support four-valued logic vectors, operations such as concatenation, and other features that are needed to efficiently describe hardware operations.

SystemC provides a set of limited-precision and arbitrary-precision data types that allows you to create integers, bit vectors, and logic vectors of any length. SystemC also supports all common operations on these data types.

To modify a SystemC model for synthesis, you need to evaluate all variable declarations, formal parameters, and return types of all functions to determine the appropriate data type and the appropriate widths of each data type. The following sections provide recommendations about the appropriate data type to use and when. Selecting the data widths is a design decision, and it is typically a tradeoff between the cost of hardware and the required precision. This decision is, therefore, up to you.

Synthesizable Data Types

C++ is a strongly typed language. Every constant, port, signal, variable, function return type, and parameter is declared as a data type, such as bool or sc_int<*n*>. Therefore, it is important that you use the correct data types in expressions.

Nonsynthesizable Data Types

All SystemC and C++ data types, except the following types, can be used for RTL synthesis:

- Floating-point types such as float and double
- Fixed-point types sc_fixed, sc_ufixed, sc_fix, and sc_ufix
- Access types such as pointers
- File types such as FILE
- I/O streams such as stdout and cout

Recommended Data Types for Synthesis

For best synthesis, use appropriate data types and bit-widths so SystemC Compiler does not build unnecessary hardware.

The following are some general recommendations about data type selection:

- For a single-bit variable, use the native C++ type bool.
- For variables with a width of 64 bits or less, use the sc_int or sc_uint data type. Use sc_uint for all logic and unsigned arithmetic operations. Use sc_int for signed arithmetic operations as well as for logic operations. These types produce the fastest simulation runtimes of the SystemC types.
- For variables larger than 64 bits, use sc_bigint or sc_biguint if you want to do arithmetic operations with these variables.
- Use sc_logic or sc_lv only when you need to model three-state signals or buses. When you use these data types, avoid comparison with X and Z values in your synthesizable code, because such comparisons are not synthesizable. Examples of three-state inference are provided in "Three-State Inference" on page 4-47.
- Use native C++ integer types for loop counters. Recommendations about loops are provided in "Loops" on page 4-53.
- Use the native C++ data types with caution, because their size is platform dependent. For example, on most platforms, a char is 8 bits wide, a short is 16 bits wide, and both an int and a long are 32 bits wide. An int, however, can be 16, 32, or 64 bits wide.

To restrict bit size for synthesis, use the recommended SystemC data types summarized in Table 3-3 in place of the equivalent C++ native type. For example, change an int type to an sc_int<*n*> type.

Table 3-3 Synthesizable Data Types

SystemC and C++ type	Description
sc_bit	A single-bit true or false value. Supported but not recommended. Use the bool data type.
sc_bv<*n*>	An arbitrary-length bit vector. Use sc_uint<*n*> when possible.
sc_logic	A single-bit 0, 1, X, or Z.
sc_lv<*n*>	An arbitrary-length logic vector.
sc_int<*n*>	Fixed-precision integers with a maximum size of 64 bits and 64 bits of precision during operations.
sc_uint<*n*>	Fixed-precision integers with a maximum size of 64 bits and 64 bits of precision during operations, unsigned.
sc_bigint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision.
sc_biguint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision, unsigned.
bool	A single-bit true or false value.
int	A signed integer, typically 32 or 64 bits, depending on the platform.
unsigned int	An unsigned integer, typically 32 or 64 bits, depending on the platform.
long	A signed integer, typically 32 bits or longer, depending on the platform.
unsigned long	An unsigned integer, typically 32 bits or longer, depending on the platform.

Table 3-3 Synthesizable Data Types (Continued)

SystemC and C++ type	Description
char	8-bit signed character, platform-dependent.
unsigned char	8-bit unsigned character, platform-dependent.
short	A signed short integer, typically 16 bits, depending on the platform.
unsigned short	An unsigned short integer, typically 16 bits, depending on the platform.
struct	A user-defined aggregate of synthesizable data types.
enum	A user-defined enumerated data type associated with an integer constant.

SystemC to VHDL Data Type Conversion

The `compile_systemc` command with the `-rtl -format vhdl` option converts the SystemC data types to VHDL data types, as listed in Table 3-4.

Table 3-4 SystemC to VHDL Data Type Conversion

SystemC data type	VHDL data type
bool	std_logic
sc_int	signed
sc_uint	unsigned

Using SystemC Data Types

Use the SystemC data type operators to access individual bits of a value.

Fixed-Precision and Arbitrary-Precision Data Type Operators

Table 3-5 lists the operators available for the SystemC sc_int and sc_uint fixed-precision and sc_bigint and sc_biguint arbitrary-precision integer data types.

Table 3-5 SystemC Integer Data Type Operators

Operators
Bitwise &(and), |(or), ^(xor), and ~(not)
Bitwise <<(shift left) and >>(shift right)
Assignment =, &=, |=, ^=, +=, -=, *=, /=, and %=
Equality ==, !=
Relational <, <=, >, and >=
Autoincrement ++ and autodecrement --
Bit selection [x]
Part selection range (x,y)
Concatenation (x,y)
Type conversion: to_uint( ) and to_int( )

Note:

The reduction and_reduce(), or_reduce(), and xor_reduce() operators are not available for the fixed- and arbitrary-precision data types.

Concatenating Variables

Variables must be of the same SystemC data type to use the concatenation operator (,). SystemC Compiler reports an error if your code concatenates variables of different SystemC data types. For example, the following code produces an error, because the data type of the parity variable is not the same as the data types of a and b:

```
...
sc_uint<16> a = 0;
sc_uint<15> b = 0;
bool parity;
sc_uint<32> c = 0;
...
c = (a, b, parity);
...
```

To correct this coding error, you must use the same data types of variables b, parity, and c. For example,

```
...
sc_uint<16> a = 0;
sc_uint<15> b = 0;
sc_uint<1> parity;
sc_uint<32> c = 0;
...
c = (a, b, parity);
...
```

Or you can cast the parity variable type. For example,

```
...
sc_uint<16> a = 0;
sc_uint<15> b = 0;
bool parity;
sc_uint<32> c = 0;
...
c = (a, b, sc_uint<1>(parity))
...
```

The equality operator (=) has a higher precedence than the concatenation operator (,). Enclose concatenation operations in an expression within parentheses to ensure that the expression is evaluated correctly. For example, in the following expression, a = b is evaluated before b and c are concatenated:

```
a = b, c;
```

To ensure that b and c are concatenated before the result is assigned to c, enclose (b, c) within parentheses, as follows:

```
a = (b, c);
```

Using a Variable to Read and Write Bits

You can read or write all bits of a port or signal. You cannot read or write the individual bits, regardless of the data type, because this operation is not allowed in SystemC.

To do a bit-select on a port or signal, read the value into a temporary variable and do a bit-select on the temporary variable. Example 3-2 shows reading from a port into a temporary variable and writing selected bits to an output port.

Example 3-2   Reading and Writing Bits With a Variable

```
#include "systemc.h"

SC_MODULE(bit_range) {
   sc_in<sc_int<8> > in;
   sc_out<sc_int<5> > out;

   sc_int<8> var_i;
   sc_signal<sc_int<5> > sig_i;

   void entry() {
     var_i = in.read();
     sig_i = var_i.range(6,2);
     out.write(var_i.range(1,5));
   }

   SC_CTOR(bit_range) {
```

```
    SC_METHOD(entry);
    sensitive << in;
  }
};
```

Example 3-2 reads the value of port in into temporary variable var_i and writes bits 2 through 6 of var_i into signal sig_i. Then it writes bits 1 through 5 to port out.

Using Constants

SystemC Compiler supports constant variables local to a function. It supports static constants only at the global level. Example 3-3 shows some examples of using constants in your design.

Example 3-3 Defining a Bit-Width at the Global Level

```
#include <systemc.h>

// The keyword static is allowed only for
// constants in the global namespace.
static const sc_uint<8> my_array1[2] = {1,2};
#define BITWIDTH1 4
#define BITWIDTH2 8

SC_MODULE(rtl_const) {

  sc_in<sc_uint<BITWIDTH1> >  addr;
  sc_out<sc_uint<BITWIDTH2> > data1;
  sc_out<sc_uint<BITWIDTH2> > data2;

  sc_uint<8> my_array2[2];

  void my() {
    const sc_uint<8> my_array2[2] = {3,4};
    const int const2 = 4;

    data1 = my_array1[0];
    data2 = my_array2[addr.read()];
  }

  SC_CTOR(rtl_const) {
    SC_METHOD(my);
    sensitive << addr;
  }
};
```

Using Enumerated Data Types

SystemC Compiler supports enumerated (enum) data types and interprets an enum data type the same way a C++ compiler interprets it. Example 3-4 shows an enum data type definition.

Example 3-4 Enumerated Data Type

```
enum command_t{
    NONE,
    RED,
    GREEN,
    YELLOW
};
```

Using Aggregate Data Types

To group data types into a convenient aggregate type, define them as a struct type (Example 3-5 or Example 3-6). You need to use all synthesizable data types in a struct in order for it to be synthesizable. SystemC Compiler splits the struct type into individual elements for synthesis.

For synthesis, do not nest a struct inside a struct, and do not include an array in the struct.

Example 3-5 Aggregate struct Data Type

```
struct package {
    sc_uint<8> command;
    sc_uint<8> address;
    sc_uint<12> data;
}
```

Example 3-6 Aggregate typedef Data Type

```
typedef struct {
    sc_uint<8> command;
    sc_uint<8> address;
    sc_uint<12> data;
```

```
} package_s;
package_s package;
```

Data Members of a Module

Do not use data members for interprocess communication, because it can lead to nondeterminism (order dependencies) during simulation and can cause mismatches between the results of pre-synthesis and post-synthesis simulation. Instead of a data member for interprocess communication, use an sc_signal for this purpose.

Example 3-7 shows (in bold) a data member variable named count that is incorrectly used to communicate between the do_count( ) and outregs( ) processes. A value is written to the *count* variable in the do_count( ) process, and a value is read from the same variable in the outregs( ) process. The order in which the two processes execute cannot be predicted—therefore, you cannot determine whether writing to the *count* variable is happening before or after count increments.

Example 3-7 Incorrect Use of a Data Member Variable for Interprocess Communication

```
/****mem_var_bad.h****/
#include "systemc.h"
SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> reset_z;
  sc_out<sc_uint<4> > count_out;
  sc_uint<4> count;            // Member Variable
  SC_CTOR(counter) {
    SC_METHOD(do_count);
    sensitive_pos << clk;
    sensitive_neg << reset_z;

    SC_METHOD(outregs);
    sensitive_pos << clk;
    sensitive_neg << reset_z;
  }
  void do_count() {
```

```
    if (reset.read() == 0) {
      count = 0;
    }else{
      count++;
    }
  }
  void outregs() {
    if (reset.read() == 0){
      count_out.write(0);
    }else{
      count_out.write(count);
    }
  }

};
```

To eliminate the nondeterminism of *count* in Example 3-7, change count to an sc_signal, as shown in bold in Example 3-8. Notice that the only change in the code is the type declaration of *count*.

Example 3-8 Correct Use of a Signal for Interprocess Communication

```
/****mem_var_good.h****/
#include "systemc.h"

SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> reset_z;
  sc_out<sc_uint<4> > count_out;

  // Signal for interprocess communication
  sc_signal<sc_uint<4> > count;
  SC_CTOR(counter) {
    SC_METHOD(do_count);
    sensitive_pos << clk;
    sensitive_neg << reset_z;

    SC_METHOD(outregs);
    sensitive_pos << clk;
    sensitive_neg << reset_z;
  }
  void do_count() {
    if (reset_z.read() == 0){
      count = 0;
    }else{
      count.read() +1;
    }
  }
  void outregs() {
    if (reset_z.read() == 0){
```

```
      count_out.write(0);
    }else{
      count_out.write(count);
    }
  }

};
```

Assigning to Data Members in the Constructor

You can make assignments to data members from within the constructor. These assignments are treated as constants for synthesis.

Recommendations About Modification for Synthesis

The following practices are recommended during modification for synthesis:

- After each modification step, reverify your design to ensure that you did not introduce errors during that step.
- Although it is recommended that you thoroughly define for synthesis at each modification stage, you might prefer a different technique. For example, during data modification, you can change one data type at a time and evaluate the impact on synthesizability and the quality of results with SystemC Compiler. Similarly, you might want to replace one nonsynthesizable construct with a synthesizable construct and reverify the design before replacing the next nonsynthesizable construct.

4

RTL Coding Guidelines

This chapter provides SystemC RTL coding guidelines. The examples in this chapter use the lsi_10k sample target library provided in the $SYNOPSYS/libraries/syn directory.

It contains the following sections:

Register Inference

Register inference allows you to use sequential logic in your designs and keep your designs technology independent. A register is an array of 1-bit memory devices. A latch is a level-sensitive memory device, and a flip-flop is an edge-triggered memory device. Use the coding guidelines in this section to control flip-flop and latch inference.

As a recommended design practice, whenever you infer registers, make certain that the clock and data inputs to the registers can be directly controlled from the ports of the design. This ensures that you can initialize your design easily during simulation as well as in the actual circuit. You can, of course, infer registers with a set and a reset, which makes the task of register initialization easier and is highly recommended.

Flip-Flop Inference

SystemC Compiler can infer D flip-flops, JK flip-flops, and toggle flip-flops. The following sections provide details about each of these flip-flop types.

Simple D Flip-Flop

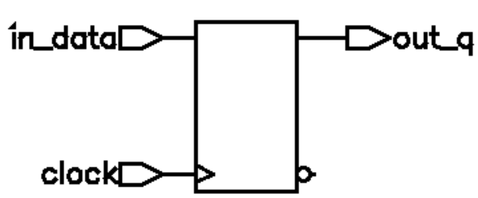
To infer a simple D flip-flop, make the SC_METHOD process sensitive to only one edge of the clock signal. To infer a rising-edge-triggered flip-flop, make the process sensitive to the positive edge of the clock, and make the process sensitive to the negative edge to infer a falling-edge-triggered flip-flop.

SystemC Compiler creates flip-flops for all the variables that are assigned values in the process. Example 4-1 is a common SC_METHOD process description that infers a flip-flop. Figure 4-1 shows the inferred flip-flop.

Example 4-1 Inferring a Rising-Edge-Triggered Flip-Flop

```
/* Rising-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff1) {
  sc_in<bool> in_data;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  // Method for D-flip-flop
  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff1) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock;
  }
};

void dff1::do_dff_pos(){
  out_q.write(in_data.read());
}
```

Figure 4-1 Inferred Rising-Edge-Triggered Flip-Flop



D Flip-Flop With an Active-High Asynchronous Set or Reset

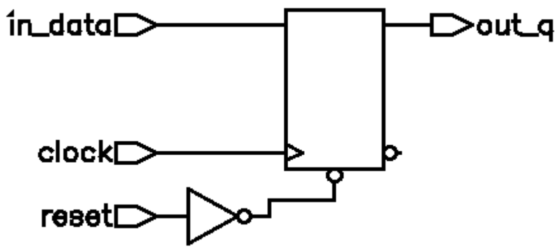
To infer a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the SC_METHOD process constructor. Specify the asynchronous signal conditions with an if statement in the SC_METHOD process definition. Example 4-2 shows a typical asynchronous specification. Specify the asynchronous branch conditions before you specify the synchronous branch conditions.

Example 4-2 is the SystemC description for a D flip-flop with an active-high asynchronous reset. Figure 4-2 shows the inferred flip-flop.

Example 4-2 D Flip-Flop With an Active-High Asynchronous Reset

```
/* Rising-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff3) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff3) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock << reset;
  }
};

void dff3::do_dff_pos () {
   if (reset.read()){
     out_q.write(0);
   }else{
     out_q.write(in_data.read());
   }
}
```

Figure 4-2 D Flip-Flop With an Active-High Asynchronous Reset



D Flip-Flop With an Active-Low Asynchronous Set or Reset

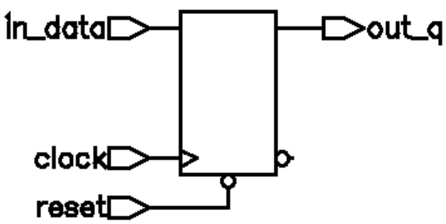
Example 4-3 is a SystemC description for a D flip-flop with an active-low asynchronous reset. Figure 4-3 shows the inferred flip-flop.

Example 4-3 D Flip-Flop With an Active-Low Asynchronous Reset

```
/* Rising-edge-triggered DFF
   with active-low reset */

#include "systemc.h"

SC_MODULE (dff3a) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff3a) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock;
    sensitive_neg << reset;
  }
};

void dff3a::do_dff_pos () {
   if (reset.read() == 0){
     out_q.write(0);
```

```
    }else{
      out_q.write(in_data.read());
    }
}
```

Figure 4-3   D Flip-Flop With an Active-Low Asynchronous Reset



D Flip-Flop With Active-High Asynchronous Set and Reset

Example 4-4 is a SystemC description for a D flip-flop with active-high asynchronous set and reset. Figure 4-4 shows the inferred flip-flop.

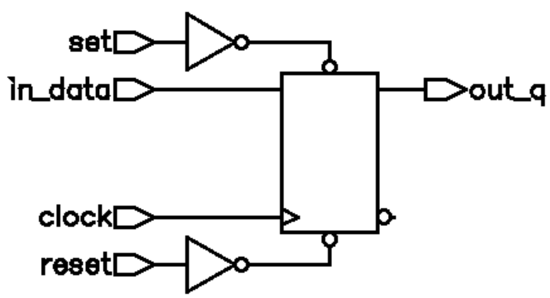
An implied priority exists between set and reset, and reset has priority. This priority is not guaranteed, because it can be implemented differently in various technology libraries. To ensure the correct behavior, assign a high value to either the set or reset at one time, but not to both at the same time.

Example 4-4   Flip-Flop With Asynchronous Set and Reset

```
/* Rising-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff4) {
  sc_in<bool> in_data, reset, set;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port
```

```
  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff4) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock << reset << set;
  }
};

void dff4::do_dff_pos () {
   if (reset.read()){
     out_q.write(0);
   }else if (set.read()){
     out_q.write(1);
   }else{
     out_q.write(in_data.read());
   }
}
```

Figure 4-4   Flip-Flop With Asynchronous Set and Reset



D Flip-Flop With Synchronous Set or Reset

The previous examples illustrated how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set a flip-flop.

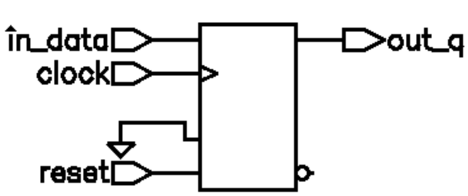
If the target technology library does not have a D flip-flop with a synchronous reset, a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop is inferred. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

To specify a synchronous set or reset input, do not include it in the sensitivity list. Describe the synchronous set or reset test and action in an if statement. Example 4-5 is a SystemC description for a D flip-flop with synchronous reset. Figure 4-5 shows the inferred flip-flop.

Example 4-5 D Flip-Flop With Synchronous Reset

```
/* Rising-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff5) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  // Method for D-flip-flop
  void dff ();

  // Constructor
  SC_CTOR (dff5) {
    SC_METHOD (dff);
    sensitive_pos << clock;
  }
};

void dff5::dff()
{
  if (reset.read()){
    out_q.write(0);
  }else{
    out_q.write(in_data.read());
  }
}
```

Figure 4-5 D Flip-Flop With Synchronous Reset



Inferring JK Flip-Flops

Use a switch…case statement to infer JK flip-flops.

JK Flip-Flop With Synchronous Set and Reset. Example 4-6 shows the SystemC code that implements the JK flip-flop truth table in Table 4-1. In the JK flip-flop, the J and K signals are similar to active-high synchronous set and reset. Figure 4-6 shows the inferred flip-flop.

Table 4-1 Rising-Edge-Triggered JK Flip-Flop Truth Table

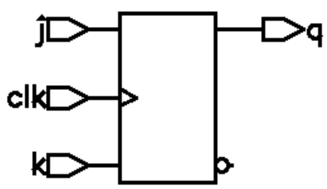
J	K	CLK	Qn+1
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	$\overline{Qn}$
X	X	Falling	Qn

Example 4-6 JK Flip-Flop

```
/* Rising-edge-triggered JK FF */

#include "systemc.h"

SC_MODULE (jkff1) {
  sc_in<bool> j, k;
  sc_inout<bool> q;  // inout to read q for toggle
  sc_in<bool> clk;   // clock port

  // Method for D-flip-flop
  void jk_flop ();

 // Constructor
  SC_CTOR (jkff1) {
    SC_METHOD (jk_flop);
    sensitive_pos << clk;
  }
};
```

```
void jkff1::jk_flop() {
    sc_uint<2> temp;         //temp to create vector
    temp[1] = j.read( );
    temp[0] = k.read( );
    switch(temp) {
    case 0x1: q.write(0);     // write a zero
      break;
    case 0x2: q.write(1);     // write a 1
      break;
    case 0x3:                 // toggle
      q.write(!q.read());
      break;
    default: break;          // no change
    }
}
```

Figure 4-6 JK Flip-Flop



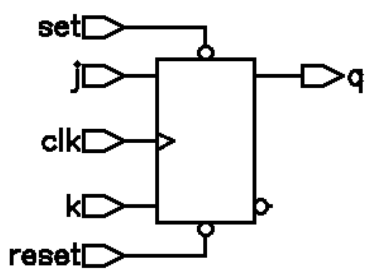
JK Flip-Flop With Asynchronous Set and Reset. Example 4-7 is a SystemC description for a JK flip-flop with an active-low asynchronous set and reset. To specify an asynchronous set or reset, specify the signal in the sensitivity list as shown in Example 4-7. Figure 4-7 shows the inferred flip-flop.

Example 4-7 JK Flip-Flop With Asynchronous Set and Reset

```
/* Rising-edge-triggered JKFF */

#include "systemc.h"

SC_MODULE (jkff2) {
  sc_in<bool> j, k, set, reset;
  sc_inout<bool> q;   // inout to read q for toggle
  sc_in<bool> clk;    // clock port

  // Method for D-flip-flop
  void jk_flop ();
```

```
  // Constructor
  SC_CTOR (jkff2) {
    SC_METHOD (jk_flop);
    sensitive_pos << clk;
    sensitive_neg << set << reset;
  }
};
void jkff2::jk_flop() {
      sc_uint<2> temp;  //temp to create vector
      if (reset.read()==0){
           q.write(0);  // reset
      }else if (set.read()==0){
           q.write(1);  // set
      }else {
           temp[1] = j.read();
           temp[0] = k.read();
           switch(temp) {
             case 0x1: q.write(0);  // write zero
               break;
             case 0x2: q.write(1);  // write a 1
               break;
             case 0x3:              // toggle
               q.write(!q.read());
               break;
             default: break;        // no change
           }
      }
}
```

Figure 4-7   JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

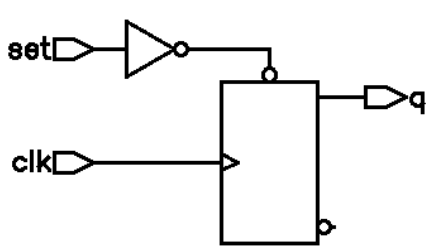
This section describes the toggle flip-flop with an asynchronous set and the toggle flip-flop with an asynchronous reset.

Toggle Flip-Flop With Asynchronous Set. Example 4-8 is a description for a toggle flip-flop with asynchronous set. The asynchronous set signal is specified in the sensitivity list. Figure 4-8 shows the flip-flop.

Example 4-8 Toggle Flip-Flop With Asynchronous Set

```
#include "systemc.h"

SC_MODULE( tff1 ) {
  sc_in<bool> set, clk;
  sc_inout<bool> q;    // inout to read q for toggle

  void t_async_set_fcn ();

 SC_CTOR( tff1 ) {
    SC_METHOD( t_async_set_fcn);
    sensitive_pos << clk << set;
  }
};

void tff1::t_async_set_fcn () {
  if (set.read()){
    q.write(1);
  }else{
    q.write(!q.read());
  }
}
```

Figure 4-8 Toggle Flip-Flop With Asynchronous Set

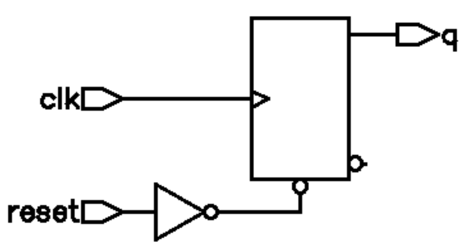


Toggle Flip-Flop With Asynchronous Reset. Example 4-9 is a SystemC description for a toggle flip-flop with asynchronous reset. The asynchronous reset signal is specified in the sensitivity list. Figure 4-9 shows the inferred flip-flop.

Example 4-9 Toggle Flip-Flop With Asynchronous Reset

```
#include "systemc.h"

SC_MODULE( tff2 ) {
  sc_in<bool> reset, clk;
  sc_inout<bool> q;  // to read q for toggle

  void t_async_reset_fcn();

 SC_CTOR( tff2 ) {
    SC_METHOD( t_async_reset_fcn);
    sensitive_pos << clk << reset;
  }
};

void tff2::t_async_reset_fcn () {
      if (reset.read()){
       q.write(0);
      }else{
       q.write(!q.read());
      }
}
```

Figure 4-9 Toggle Flip-Flop With Asynchronous Reset



Latch Inference

In simulation, a signal or a variable holds its value until that value is reassigned. A latch implements the ability to hold a state in hardware. SystemC Compiler supports inference of set/reset (SR) and delay (D) latches.

You can unintentionally infer latches from your SystemC code, which can add unnecessary hardware. SystemC Compiler infers a D latch when your description has an incomplete assignment in an if…else or switch…case statement. To avoid creating a latch, specify all conditions in if…else and switch…case statements and assign all variables in each branch.

Inferring a D Latch From an If Statement

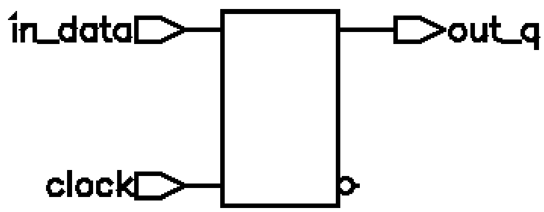
An if statement infers a D latch when there is no else clause, as shown in Example 4-10. The SystemC code specifies a value for output out_q only when the clock has a logic 1 value, and it does not specify a value when the clock has a logic 0 value. As a result, output out_q becomes a latched value. Figure 4-10 shows the schematic of the inferred latch.

Example 4-10 D Latch Inference Using an if Statement

```
#include "systemc.h"

SC_MODULE( d_latch1 ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    if (clock.read())
       {out_q.write(in_data.read());}
  }

  // Constructor
  SC_CTOR( d_latch1 ) {
```

```
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-10   D Latch Inferred From an if Statement



Inferring an SR Latch. SR latches are difficult to test, so use them with caution. If you use SR latches, verify that the inputs are hazard free and do not generate glitches. During synthesis, SystemC Compiler does not ensure that the logic driving the inputs is hazard free.

Example 4-11 is the SystemC code that implements the truth table in Table 4-2 for an SR latch. Figure 4-11 shows the inferred SR latch.

Output y is unstable when both inputs are at a logic 0 value, so you need to include a check in the SystemC code to detect this condition during simulation. SystemC Compiler does check for these conditions.

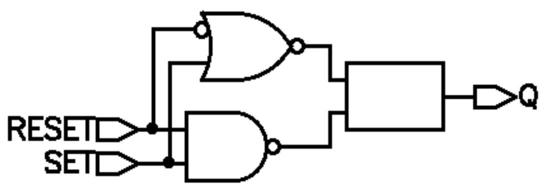
Table 4-2   Truth Table for the SR Latch (NAND Type)

set	**reset**	**Q**
0	0	Not stable
0	1	1
1	0	0
1	1	Q

Example 4-11 SR Latch

```
/***sr_latch.cc***/

#include "systemc.h"

SC_MODULE( sr_latch ) {
  sc_in<bool> RESET, SET;
  sc_out<bool> Q;

  void sr_latch_fcn () {
    if (RESET.read() == 0){
      Q.write(0);
    }else if (SET.read() == 0){
      Q.write(1);
    }
  }

  SC_CTOR( sr_latch ) {
    SC_METHOD( sr_latch_fcn);
    sensitive << RESET << SET;
  }
};
```

Figure 4-11 SR Latch

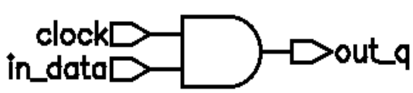


Avoiding Latch Inference. To avoid latch inference, assign a value to a signal for all cases in a conditional statement. Example 4-12 shows addition of an else clause to avoid the latch inferred by the if statement in Example 4-10, and Figure 4-12 shows the resulting schematic.

Example 4-12   Adding an Else Clause to Avoid Latch Inference

```
#include "systemc.h"

SC_MODULE( d_latch1a ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    if (clock.read()){
       out_q.write(in_data.read());
    }else{
       out_q.write(false);
    }
  }

  // Constructor
  SC_CTOR( d_latch1a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-12   Avoiding Latch Inference by Adding Else Clause

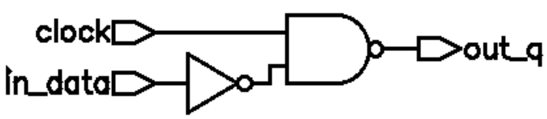


You can also avoid latch inference by assigning a default value to the output port. Example 4-13 shows the setting of a default value to avoid the latch inferred by the if statement in Example 4-10, and Figure 4-13 shows the resulting schematic.

Example 4-13 Setting a Default Value to Avoid Latch Inference

```
#include "systemc.h"

SC_MODULE( d_latch1b ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    out_q.write(1);     // set a default
    if (clock.read())
       {out_q.write(in_data.read());}
  }

  // Constructor
  SC_CTOR( d_latch1b ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-13 Avoiding Latch Inference by Setting a Default Value



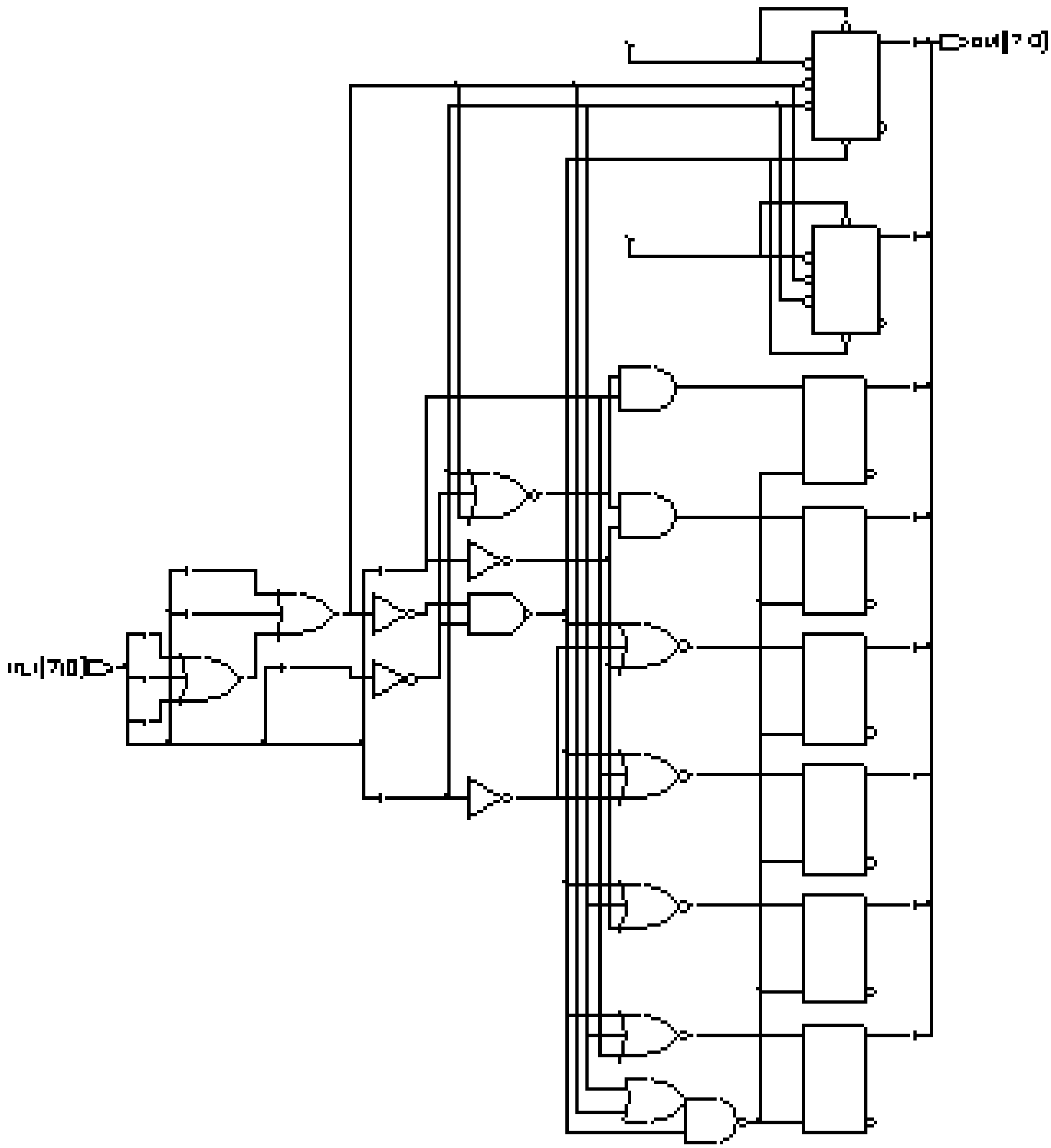
Inferring a Latch From a Switch Statement

Example 4-14 shows a switch statement that infers D latches because it does not provide assignments to the out port for all possible values of the in_i input. Figure 4-14 shows the inferred latches.

Example 4-14 Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2 ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2 ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-14 Latch Inference From a switch Statement

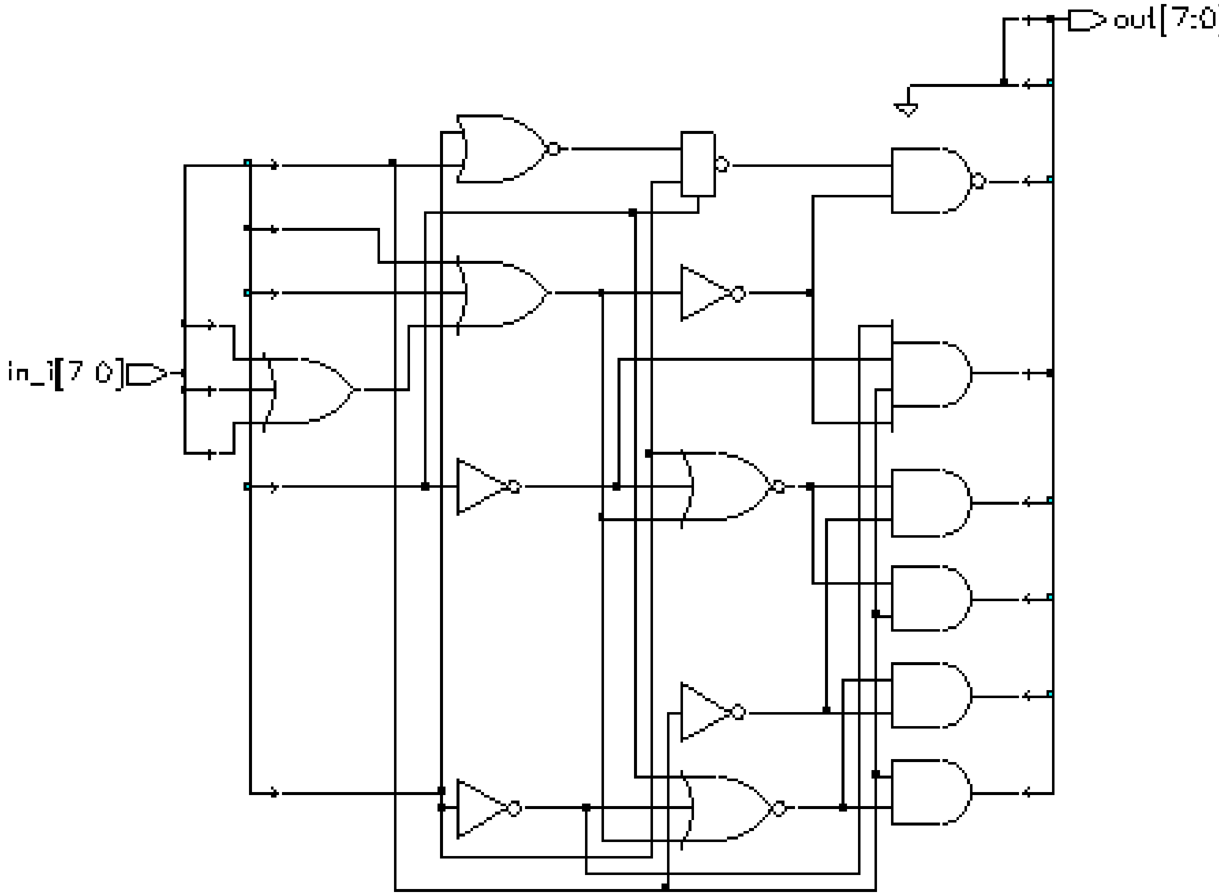


To avoid latch inference caused by the incomplete switch statement in Example 4-14, add a default case statement, as shown in Example 4-15. Figure 4-15 shows the resulting schematic.

Example 4-15 Avoiding Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2a ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    default: out.write(0x01);
    }
  }

  // Constructor
  SC_CTOR( d_latch2a ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-15 Avoiding Latch Inference by Adding a Default Case to a switch Statement

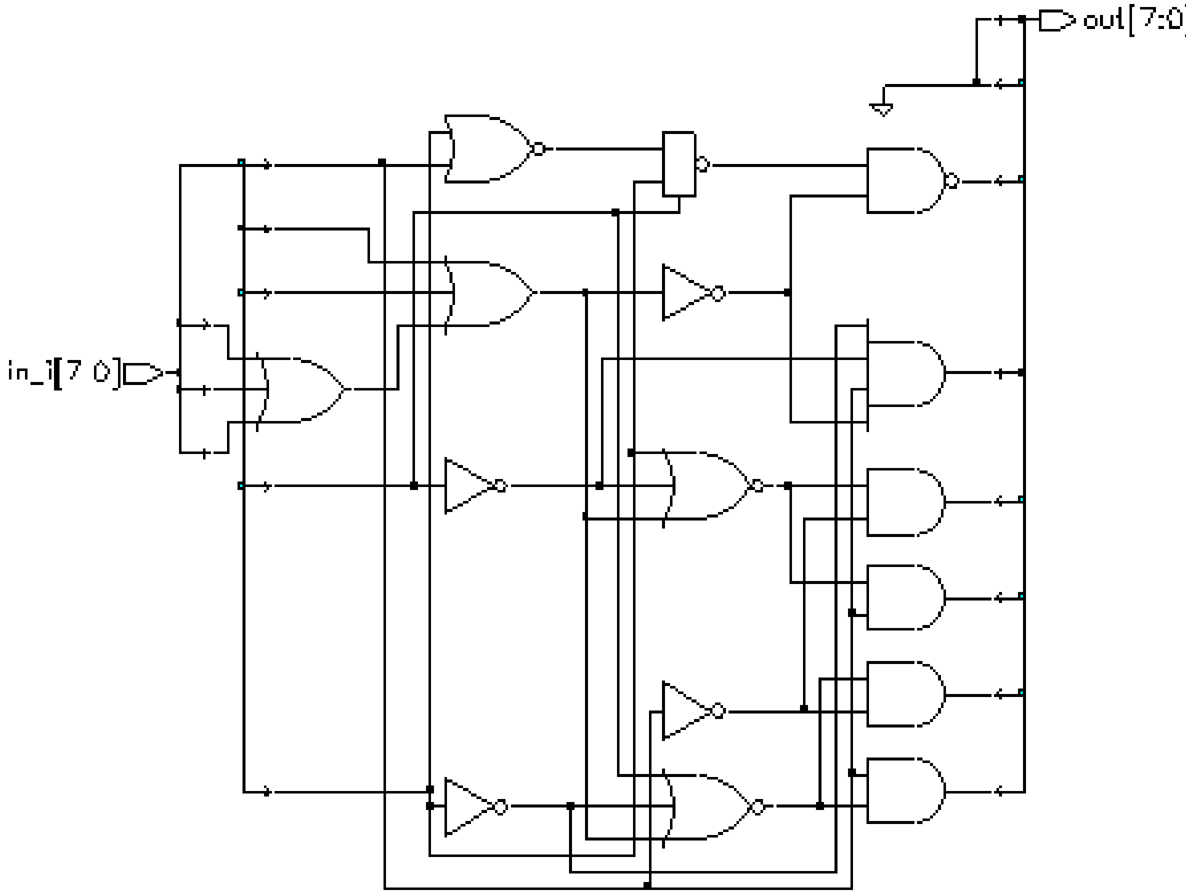


You can also avoid latch inference caused by the incomplete switch statement in Example 4-14 by writing a default value to the output port, as shown in Example 4-16. Figure 4-16 shows the resulting schematic.

Example 4-16 Set a Default Value to Avoid Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2b ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    out.write(1);  // Set default value
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2b ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-16 Avoiding Latch Inference by Setting a Default Case Before a switch Statement



Priority Encoding

Switch…case and if…else conditional statements are priority-encoded in simulation. Priority-encoded hardware is rarely needed, and it can add unnecessary gates and time to the synthesized design.

SystemC Compiler defaults to priority-encoded logic for a switch statement when

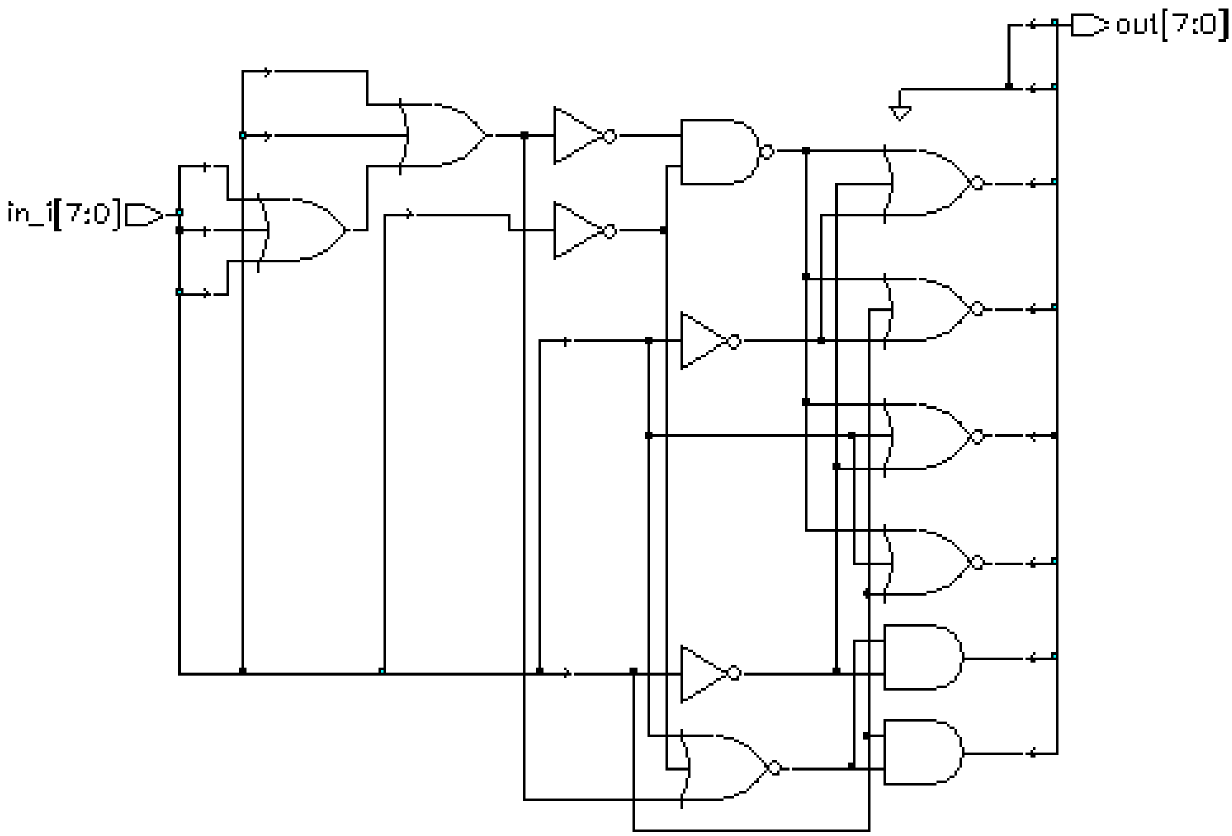
- You do not define all the cases in a switch statement
- All the cases are not mutually exclusive

In addition to defining a default case (Example 4-15 on page 4-21) and setting a default value (Example 4-16 on page 4-23), you can instruct SystemC Compiler that other cases are not necessary for a switch statement by adding the `full_case` compiler directive in your code. Example 4-17 uses the `full_case` directive, and Figure 4-17 shows the resulting schematic.

Example 4-17 Using the full_case Compiler Directive With a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2c ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    // synopsys full_case
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2c ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-17 Using the full_case Compiler Directive With a switch Statement



Active-Low Set and Reset

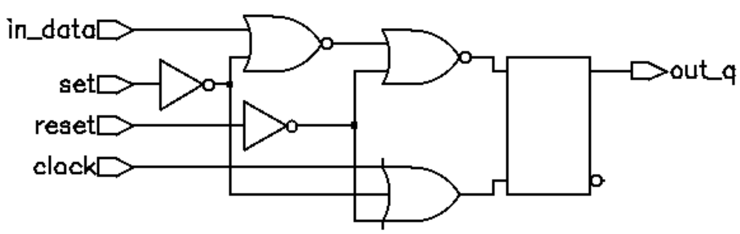
To instruct SystemC Compiler to implement all the signals in a group as active-low, add a check to the SystemC code to ensure that the group of signals has only one active-low signal at a given time. SystemC Compiler does not produce any logic to check this assertion.

Example 4-18 shows a latch with an active-low set and reset. Figure 4-18 shows the resulting schematic.

Example 4-18 Latch With Active-Low Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch6a ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
    infer_latch: {
      if (reset.read() == 0){
        out_q.write(0);
      }else if (set.read() == 0){
        out_q.write(1);
      }else if (clock.read()){
        out_q.write(in_data.read());
      }
    }
  }
  // Constructor
  SC_CTOR( d_latch6a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-18 Latch With Active-Low Set and Reset



Active-High Set and Reset

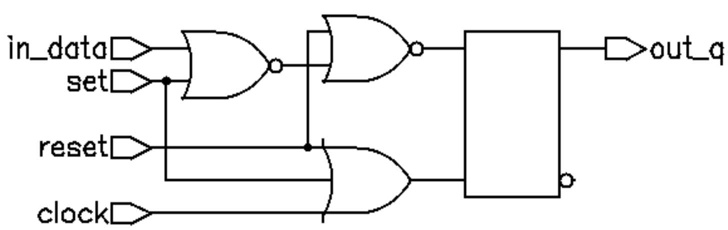
To instruct SystemC Compiler to implement all the signals in a group as active-high, add a check to the SystemC code to ensure that the group of signals has only one active-high signal at a given time. SystemC Compiler does not produce any logic to check this assertion.

Example 4-19 shows a latch with the set and reset specified as active-high. Figure 4-19 shows the resulting schematic.

Example 4-19 Latch With Active-High Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch7a ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
    infer_latch: {
      if (reset.read()){
        out_q.write(0);
      }else if (set.read()){
        out_q.write(1);
      }else if (clock.read()){
        out_q.write(in_data.read());
      }
    }
  }
  // Constructor
  SC_CTOR( d_latch7a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-19 Latch With Active-High Set and Reset



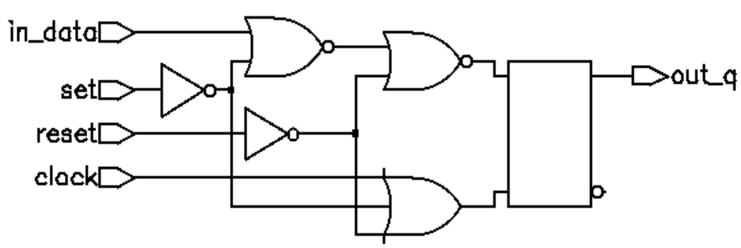
D Latch With an Asynchronous Set and Reset

Example 4-20 is a SystemC description for a D latch with an active-low asynchronous set and reset. Figure 4-20 shows the inferred latch.

Example 4-20 D Latch With Asynchronous Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch6 ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
      if (reset.read() == 0){
        out_q.write(0);
      }else if (set.read() == 0){
        out_q.write(1);
      }else if (clock.read()){
        out_q.write(in_data.read());
      }
  }
  // Constructor
  SC_CTOR( d_latch6 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-20 Latch With Asynchronous Set and Reset



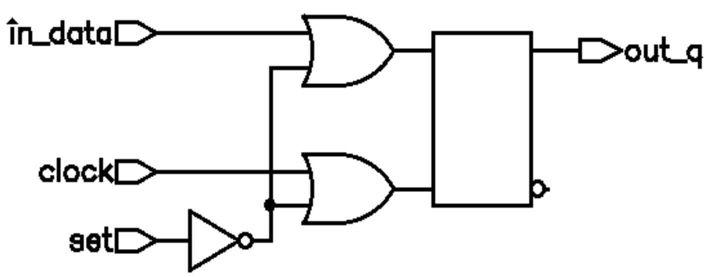
D Latch With an Asynchronous Set

Example 4-21 is a SystemC description for a D latch with an asynchronous set. Figure 4-21 shows the inferred latch.

Example 4-21 D Latch With Asynchronous Set

```
#include "systemc.h"

SC_MODULE( d_latch4 ) {
  sc_in<bool> in_data, set;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn () {
    if (set.read() == 0){
      out_q.write( 1 );
    }else if (clock.read()){
      out_q.write(in_data.read());
    }
  }

  // Constructor
  SC_CTOR( d_latch4 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set;
  }
};
```

Figure 4-21 Latch With Asynchronous Set



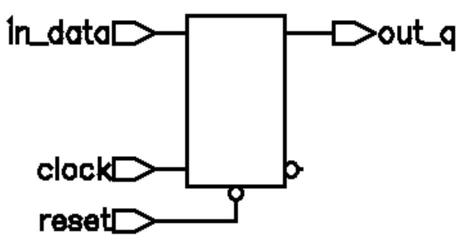
D Latch With an Asynchronous Reset

Example 4-22 is a SystemC description for a D latch with an asynchronous reset. Figure 4-22 shows the inferred latch.

Example 4-22 D Latch With Asynchronous Reset

```
#include "systemc.h"

SC_MODULE( d_latch5 ) {
  sc_in<bool> in_data, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn () {
    if (reset.read() == 0){
      out_q.write(0);
    }else if (clock.read()){
      out_q.write(in_data.read());
    }
  }
  // Constructor
  SC_CTOR( d_latch5 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << reset;
  }
};
```

Figure 4-22 Latch With Asynchronous Reset



Understanding the Limitations of Register Inference

SystemC Compiler cannot infer the following components:

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flips with multiple clock inputs
- Multiport latches

You can instantiate these components in your SystemC description. SystemC Compiler interprets these flip-flops and latches as black boxes.

Instantiating a Component as a Black Box

To instantiate a flip-flop or a latch as a black box in your SystemC RTL netlist, create a dummy SystemC module with the same module name and port names as those of the cell in the technology library that you want to instantiate. The module and port names are case-sensitive and must exactly match the cell names. You do not need to describe the module's function, because Design Compiler replaces it with the actual library cell.

Example 4-23 shows a dummy module for the or2c1 cell from the tc6a_cbacore sample technology library. An instance of the or2c1 module named my_gate is created in the gate module.

Example 4-23 Instantiating a Register That Cannot Be Inferred

```
/****gate.h****/


/*****************************************
 *  This example shows how to instantiate
 *  an or2c1 gate from the tc6a_cbacore
 *  library in a SystemC RTL netlist.
 *****************************************/
#include "systemc.h"

SC_MODULE(or2c1) {
  sc_in<bool>  A, B;
  sc_out<bool> Y;

  SC_CTOR(or2c1) {}
};

SC_MODULE(gate) {
  sc_in<bool>         clk, reset;
  sc_in<sc_uint<8> >  data;
  sc_out<bool>        match;

  sc_signal<bool> a_match, b_match;

  /*
   *   RTL processes
   */
  void match_a();
  void match_b();

  /*
   *   Pointer for block allocation
   */
  or2c1 *my_gate;

  SC_CTOR(gate) {
    /*
     *   Instantiate and hook up the gate
     */
    my_gate = new or2c1("my_gate");
    my_gate->A(a_match);
    my_gate->B(b_match);
    my_gate->Y(match);

    SC_METHOD(match_a);
    sensitive_pos << clk;
    sensitive_neg << reset;

    SC_METHOD(match_b);
    sensitive_pos << clk;
    sensitive_neg << reset;
```

```
  }
};

/****gate.cc****/
#include "gate.h"

void gate::match_a() {
  if (reset.read() == 0) {
    a_match = 0;
  } else {
    if (data.read() == 3) {
      a_match = 1;
    } else {
      a_match = 0;
    }
  }
}

void gate::match_b() {
  if (reset.read() == 0) {
    b_match = 0;
  } else {
    if (data.read() == 7) {
      b_match = 1;
    } else {
      b_match = 0;
    }
  }
}
```

To perform RTL synthesis and instantiate the or2c1 cell,

1. Elaborate the gate module that contains the dummy or2c1 module.

   ```
   dc_shell>  compile_systemc -rtl -format db gate.cc
   ```

2. Remove the dummy module.

   ```
   dc_shell> remove_design or2c1
   ```

3. Set the current design to be the gate module.

   ```
   dc_shell> current_design gate
   ```

4. Remove the current links, and create new links for the design. This links the or2c1 cell from the technology library.

```
dc_shell> link
```

5. Instruct Design Compiler not to touch the my_gate instantiation of the or2c1 cell in the design.

```
dc_shell> set_dont_touch find(cell, my_gate)
```

6. Compile the design to gates.

```
dc_shell> compile
```

Multibit Inference

A multibit component (MBC), such as a 16-bit register, reduces the area and power in a design. The primary benefit of MBCs is to create a more uniform structure for layout during place and route.

Multibit inference allows you to map registers, multiplexers, and three-state cells to regularly structured logic or multibit library cells. Multibit library cells (macro cells, such as 16-bit banked flip-flops) have these advantages:

- Smaller area and delay, due to shared transistors (as in select or set/reset logic) and optimized transistor-level layout. With single-bit components, the select or set/reset logic is repeated in each single-bit component.
- Reduced clock skew in sequential gates, because the clock paths are balanced internally in the hard macro implementing the MBC.
- Lower power consumption by the clock in sequential banked components, due to reduced capacitance driven by the clock net.
- Better performance, due to the optimized layout within the MBC.

- Improved regular layout of the datapath.

To direct SystemC Compiler to infer multibit components,

- Add the `infer_multibit` or `dont_infer_ multibit` compiler directive (see “Multibit Inference Compiler Directives” on page A-3) to individual ports or signals in the SystemC description.
- Or define the dc_shell `hdlin_infer_multibit` variable, which specifies that multibit inference is allowed for the entire design. The allowed values for `hdlin_infer_multibit` are `default_all`, `default_none`, and `never`. See the `hdlin_infer_multibit` man page for additional information.

Inferring Multibit

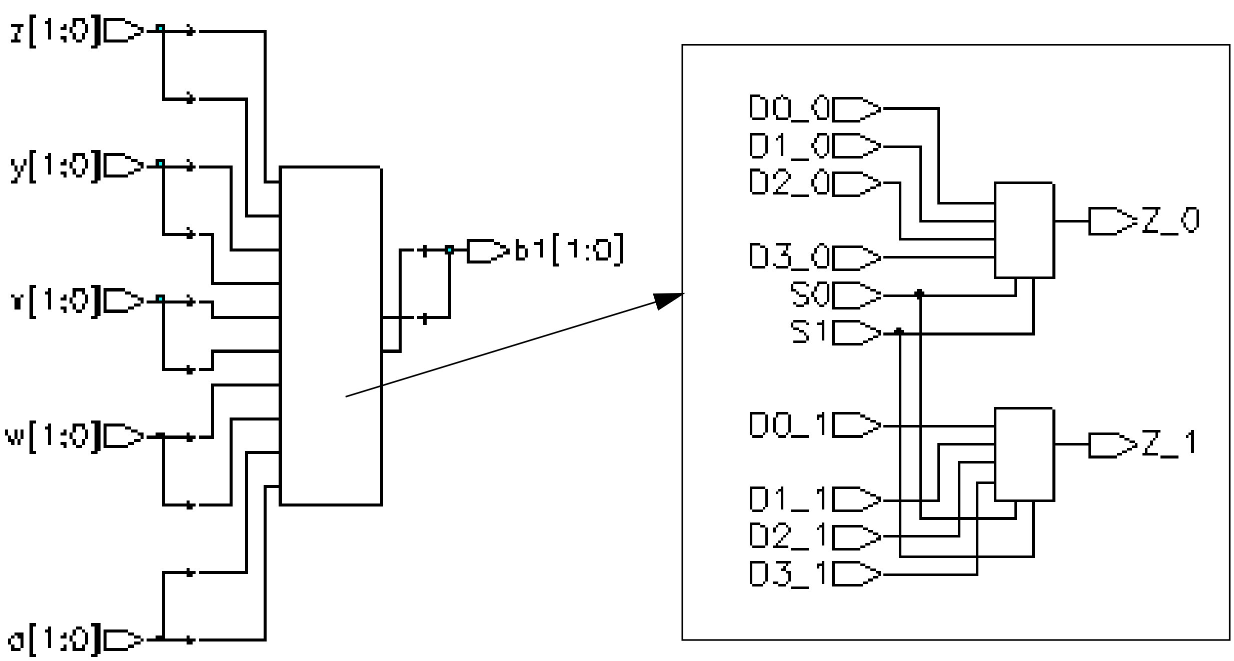
Example 4-24 shows inference of a 2-bit multiplexer, resulting in the schematic in Figure 4-23.

Example 4-24 Inferring a 2-Bit 4-to-1 Multiplexer

```
#include "systemc.h"

SC_MODULE( infer_multibit ) {

  sc_in<sc_uint<2> > a;
  sc_in<sc_int<2> > w;
  sc_in<sc_int<2> > x;
  sc_in<sc_int<2> > y;
  sc_in<sc_int<2> > z;

  sc_out<sc_int<2> > b1; // synopsys infer_multibit "b1"

  void f1 ();

  SC_CTOR( infer_multibit ) {
    SC_METHOD( f1);
    sensitive << a << w << x << y << z;
  }
};

void infer_multibit:: f1 ()
```

```
{
  switch (a.read ()) {
  case 3:
    b1.write(w);
    break;
  case 2:
    b1.write(x);
    break;
  case 1:
    b1.write(y);
    break;
  case 0:
    b1.write(z);
    break;
  }
}
```

Figure 4-23   Inferring a 2-Bit 4-to-1 Multiplexer



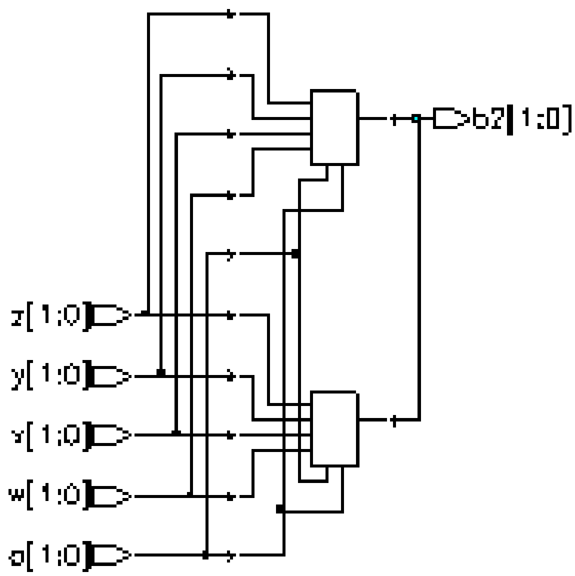
Preventing Multibit Inference

Example 4-25 shows restriction of the description to prevent inference of a 2-bit multiplexer. This restriction results in the schematic in Figure 4-24.

Example 4-25 Preventing Inference of a 2-Bit 4-to-1 Multiplexer

```
#include "systemc.h"

SC_MODULE( infer_multibit2 ) {

  sc_in<sc_uint<2> > a;
  sc_in<sc_int<2> > w;
  sc_in<sc_int<2> > x;
  sc_in<sc_int<2> > y;
  sc_in<sc_int<2> > z;

  sc_out<sc_int<2> > b2; // synopsys dont_infer_multibit "b2"

  void f1 ();

  SC_CTOR( infer_multibit2 ) {
    SC_METHOD( f1);
    sensitive << a << w << x << y << z;
  }
};

void infer_multibit2:: f1 ()
{
  switch (a.read ()) {
  case 3:
    b2.write(w);
    break;
  case 2:
    b2.write(x);
    break;
  case 1:
    b2.write(y);
    break;
  case 0:
    b2.write(z);
    break;
  }
}
```

Figure 4-24   Preventing Inference of a 2-Bit 4-to-1 Multiplexer



Multiplexer Inference

SystemC Compiler can infer a generic multiplexer cell (MUX_OP) from switch statements and if-then-else statements in your SystemC description. SystemC Compiler maps inferred MUX_OPs to multiplexer cells in the target technology library.

The size of the inferred MUX_OP depends on the number of unique cases in the switch statement. If you want to use the multiplexer inference feature, the target technology library must contain at least a 2-to-1 multiplexer.

MUX_OPs are hierarchical cells similar to Synopsys DesignWare components. SystemC Compiler passes the multiplexer inference information to Design Compiler, and Design Compiler determines the MUX_OP implementation during logic synthesis, based on the design constraints. For information about how Design Compiler maps MUX_OPs to multiplexers in the target technology library, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Inferring Multiplexers From a Block of Code

Use the `infer_mux` compiler directive to instruct SystemC Compiler to infer MUX_OPs for all switch statements inside a block of code. In Example 4-26, the `infer_mux` compiler directive is attached to the code block labeled tt, which contains two switch statements. The code block can contain any number of switch statements.

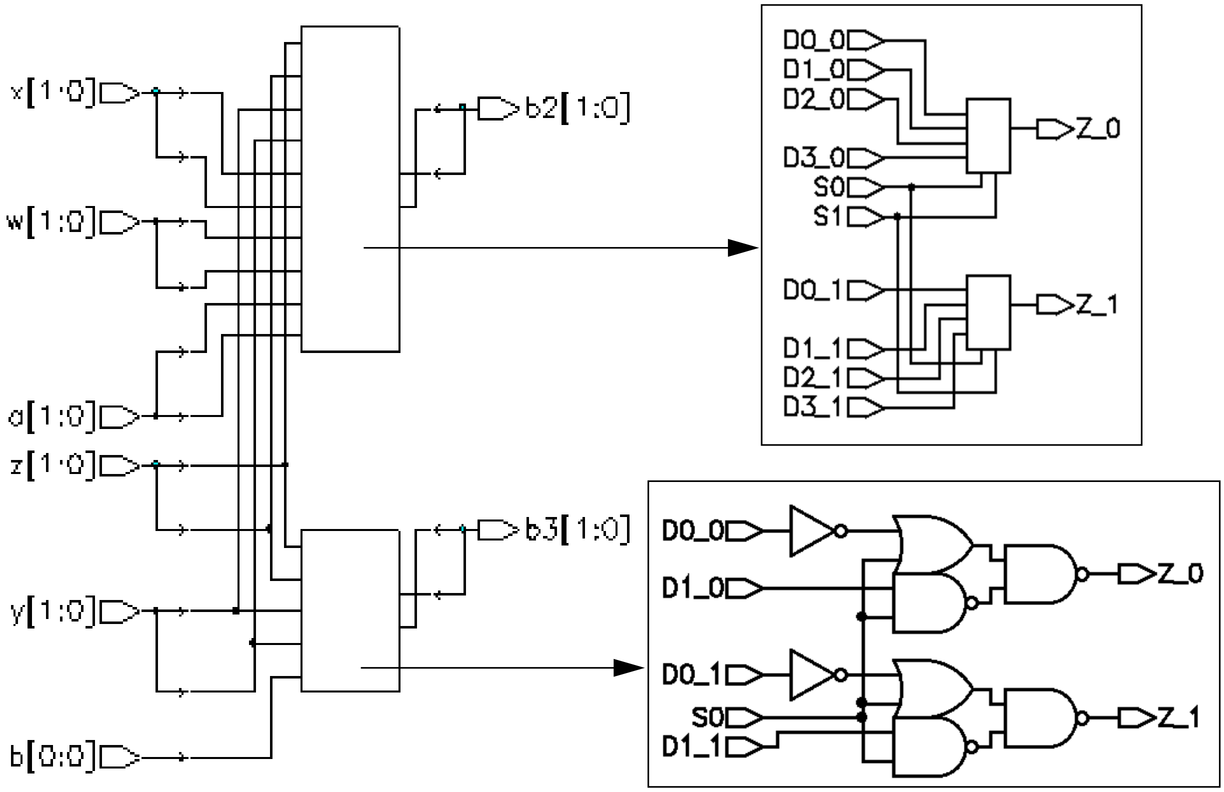
SystemC Compiler infers a MUX_OP for each case in the switch statement. The first switch statement has four unique cases and infers a 4-to-1 MUX_OP. The second switch statement has two unique cases and infers a 2-to-1 MUX_OP. Figure 4-25 shows the inferred multiplexers.

Example 4-26   Multiplexer Inference From a Block of Code

```
#include "systemc.h"

SC_MODULE( infer_mux_blk ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();

  SC_CTOR( infer_mux_blk ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

//  infer mux for all switch statements in block ’tt’
void infer_mux_blk:: f2 ()
```

```
{
  // synopsys infer_mux "tt"
  tt: {
    switch (a.read ()) {
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
    case 1:
      b2.write(y);
      break;
    case 0:
      b2.write(z);
      break;
    }
    switch (b.read ()) {
    case 1:
      b3.write(y);
      break;
    case 0:
      b3.write(z);
      break;
    }
  }
}
```

Figure 4-25   Inferring a Multiplexer for a Block



Preventing Multiplexer Inference

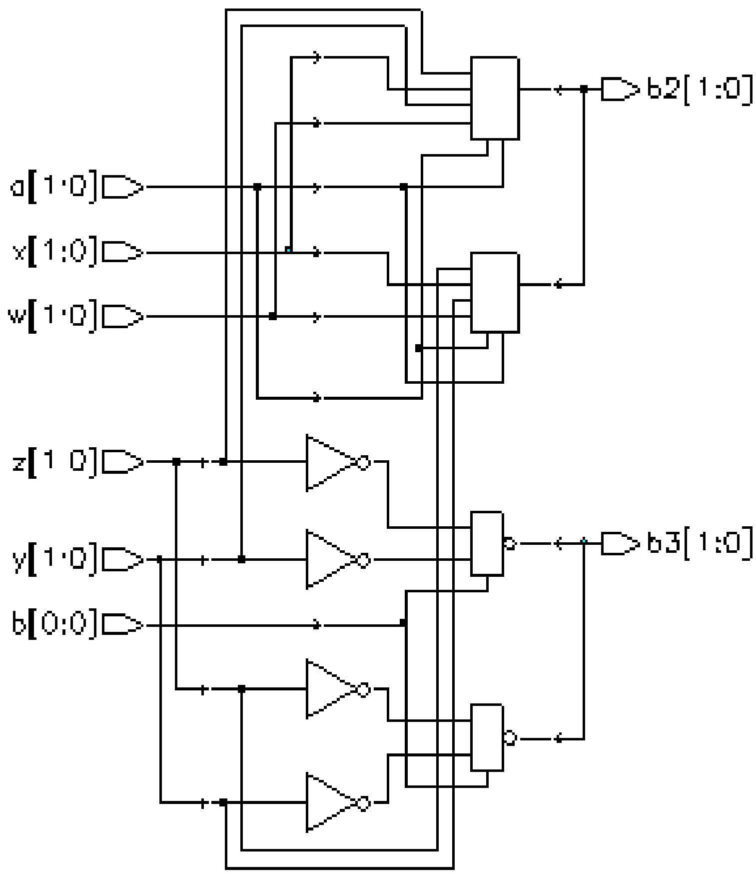
Example 4-27 shows the code from Example 4-26 without the `infer_mux` compiler directive, and Figure 4-26 shows the resulting schematic.

Example 4-27   No Multiplexer Inference From a Block of Code

```
#include "systemc.h"

SC_MODULE( infer_mux_blk ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();
```

```
  SC_CTOR( infer_mux_blk ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

/*  Do not use the infer mux for all switch
    statements in block ’tt’ */

void infer_mux_blk:: f2 ()
{
  tt: {
    switch (a.read ()) {
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
    case 1:
      b2.write(y);
      break;
    case 0:
      b2.write(z);
      break;
    }
    switch (b.read ()) {
    case 1:
      b3.write(y);
      break;
    case 0:
      b3.write(z);
      break;
    }
  }
}
```

Figure 4-26   Block of Code Without Multiplexer Inference



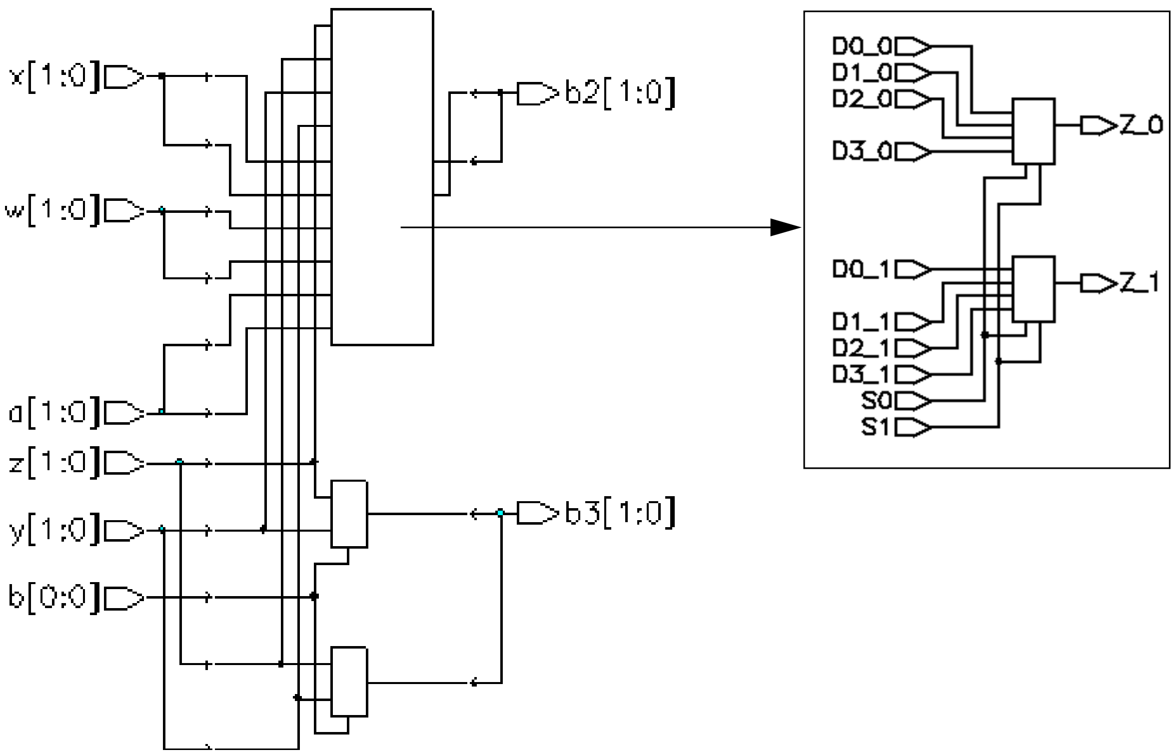
Inferring a Multiplexer From a Specific Switch Statement

You can also specify the `infer_mux` compiler directive from a single switch statement by placing the compiler directive as the first line inside the switch statement, as shown in Example 4-28. This switch statement reads four unique values, and SystemC Compiler infers a 4-to-1 MUX_OP. Figure 4-27 shows the inferred multiplexer.

Example 4-28 Multiplexer Inference From a Specific switch Statement

```
#include "systemc.h"

SC_MODULE( infer_mux_blk3 ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();

  SC_CTOR( infer_mux_blk3 ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

/*  Infer mux for only the first switch statement
    in block ’tt’ */
void infer_mux_blk3:: f2 ()
{
  tt: {
    switch (a.read ()) { //synopsys infer_mux
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
    case 1:
      b2.write(y);
      break;
    case 0:
      b2.write(z);
      break;
    }
    switch (b.read ()) {
    case 1:
      b3.write(y);
      break;
    case 0:
      b3.write(z);
      break;
    }
  }
}
```

Figure 4-27 Inferring a Multiplexer From a Specific switch Statement



Understanding the Limitations of Multiplexer Inference

SystemC Compiler does not infer MUX_OPs for

- if...else statements
- switch statements in while loops

SystemC Compiler infers MUX_OPs for incompletely specified switch statements, but the resulting logic might not be optimal. SystemC Compiler considers the following types of switch statements incompletely specified:

- A switch statement that has a missing case statement branch
- A switch statement that contains an if statement
- A switch statement that contains other switch statements

Three-State Inference

A three-state driver is inferred when you assign the value Z to a variable. The value Z represents the high-impedance state. You can assign high-impedance values to single-bit or bused variables. The assignment must occur in a conditional statement (if or switch) or with the conditional operator (?:). Note that only the sc_logic and sc_lv data types support the value Z.

Simple Three-State Inference

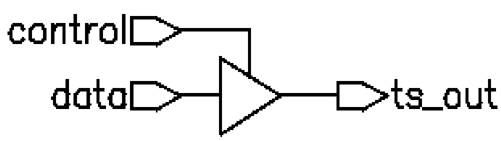
Example 4-29 is a SystemC description for a simple three-state driver. Figure 4-28 shows the schematic the code generates.

Example 4-29 Three-State Buffer Inference From a Block of Code

```
// simple three-state buffer inference
#include "systemc.h"
SC_MODULE( tristate_ex1 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;

  // Method for three-state driver
  void tristate_fcn () {
```

```
    if (control.read()){
      ts_out.write(data.read());
    }else{
      ts_out.write('Z');
    }
  }

  // Constructor
  SC_CTOR( tristate_ex1 ) {
    SC_METHOD( tristate_fcn);
    sensitive << control << data;
  }
};
```

Figure 4-28   Schematic of a Simple Three-State Driver



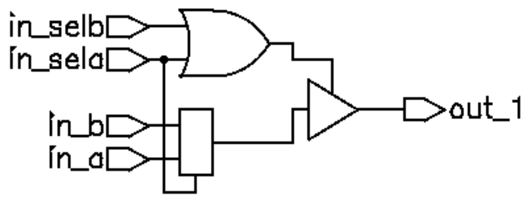
Example 4-30 shows a different coding style for three-state inference. In this case, SystemC Compiler infers a single three-state driver. Figure 4-29 shows the schematic the code generates.

Example 4-30   Inferring One Three-State Driver

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex2 ) {
  sc_in<bool> in_sela, in_selb;
  sc_in<sc_logic> in_a, in_b;
  sc_out<sc_logic> out_1;

  // Method for single three-state driver
  void tristate_fcn () {
    out_1.write('Z');  //default value
    if (in_sela.read()){
      out_1.write(in_a.read());
    }else if (in_selb.read()){
      out_1.write(in_b.read());
    }
  }

  // Constructor
  SC_CTOR( tristate_ex2 ) {
    SC_METHOD( tristate_fcn);
```

```
        sensitive << in_sela <<in_selb << in_a << in_b;
      }
    };
```

Figure 4-29   Three-State Driver With Gated Data



Three-State Driver for Bus

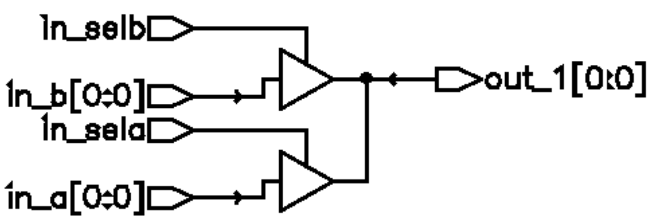
To infer a three-state driver to resolve bus contention, use a port of type sc_out_rv, as shown in Example 4-31. Figure 4-30 shows the resulting schematic.

Example 4-31   Three-State Driver for Bus

```
// Three-state buffer inference
// with resolved logic output
#include "systemc.h"

SC_MODULE( tristate_ex3 ) {
  sc_in<bool> in_sela, in_selb;
  sc_in<sc_logic> in_a, in_b;
  sc_out_rv<1> out_1;

  // Method for first three-state driver
  void tristate_a();

  // Method for second three-state driver
  void tristate_b();

  // Constructor
  SC_CTOR( tristate_ex3 ) {
    SC_METHOD( tristate_a);
    sensitive << in_sela << in_a;
    SC_METHOD( tristate_b);
    sensitive << in_selb << in_b;
  }
};
```

```
void tristate_ex3::tristate_a() {
    if (in_sela.read()){
      out_1.write(in_a.read());
    }else{
      out_1.write("Z");
    }
}

void tristate_ex3::tristate_b() {
    if (in_selb.read()){
      out_1.write(in_b.read());
   }else{
      out_1.write("Z");
   }
}
```

Figure 4-30 Three-State Bus Driver Schematic



Registered Three-State Drivers

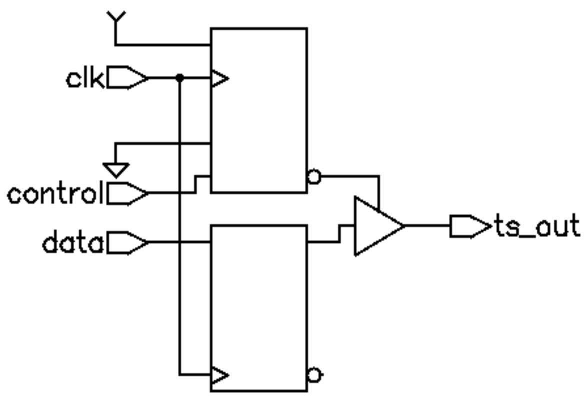
When a variable is registered in the same process in which it is inferred as three-state, SystemC Compiler also registers the enable pin of the three-state gate. Example 4-32 is an example of this type of code. Figure 4-31 shows the schematic generated by the code.

Example 4-32 Three-State Driver With Registered Enable

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex4 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;
```

```
  sc_in_clk clk;

  // Method for three-state driver
  void tristate_fcn () {
    if (control.read()){
      ts_out.write(data.read());
    }else{
      ts_out.write('Z');
    }
  }

  // Constructor
  SC_CTOR( tristate_ex4 ) {
    SC_METHOD( tristate_fcn);
    sensitive_pos << clk; // note inferred seq logic
  }
};
```

Figure 4-31   Three-State Driver With Registered Enable

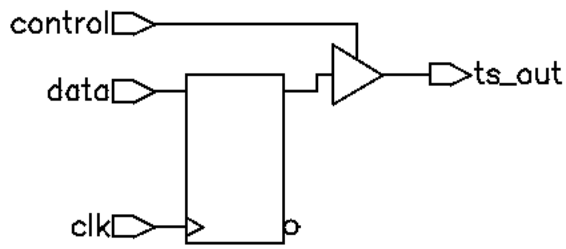


To avoid registering the enable pin, separate the three-state driver inference from the sequential logic inference, using two SC_METHOD processes. Example 4-33 uses two methods to instantiate a three-state gate, with a flip-flop only on the input. Note that the sc_signal temp is used to communicate between the two SC_METHOD processes. Figure 4-32 shows the schematic the code generates.

Example 4-33 Three-State Driver Without Registered Enable

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex5 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;
  sc_in_clk clk;

  sc_signal<sc_logic> temp;

  // Method for three-state driver
  void tristate_fcn () {
    if (control.read()){
      ts_out.write(temp);
    }else{
      ts_out.write('Z');
    }
  }

  // Method for sequential logic
  void flop () {
    temp = data.read();
  }

  // Constructor
  SC_CTOR( tristate_ex5 ) {
    SC_METHOD( tristate_fcn);
    sensitive << control << temp ;
    SC_METHOD( flop );
    sensitive_pos << clk;
  }
};
```

Figure 4-32 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

The value of Z is valid only for the sc_logic and sc_lv data types. You can use the Z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You cannot use the Z value in an expression, except for comparison with Z. Be careful when using expressions that compare with the Z value. SystemC Compiler always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, SystemC Compiler issues a warning when it synthesizes such comparisons. The following example shows incorrect use of the Z value in an expression:

```
OUT_VAL = (’Z’ && IN_VAL);
```

The following example shows correct use of the Z value:

```
IN_VAL =  ’Z’;
```

Loops

SystemC Compiler supports for loops, while loops, and do-while loops for synthesis. For RTL synthesis, SystemC Compiler keeps all loops rolled, but they are automatically unrolled by Design Compiler. Therefore, all loops must be unrollable.

Loop Unrolling Criteria

To make a loop unrollable, adhere to the following criteria for creating loops:

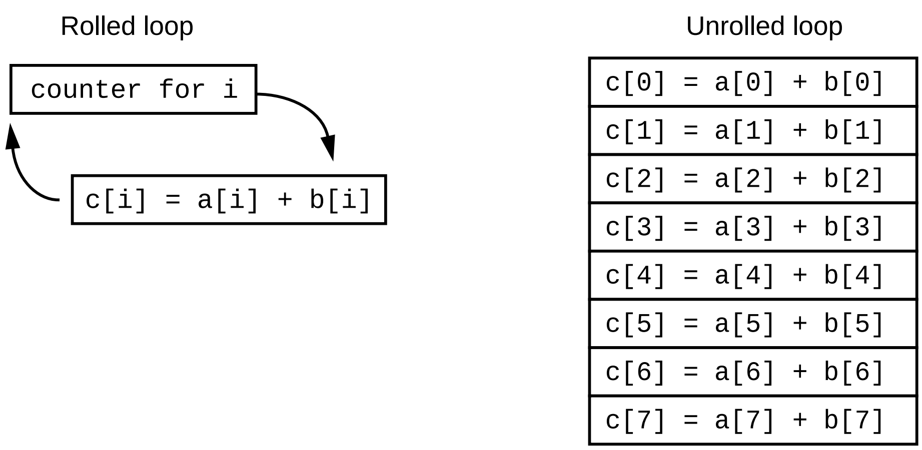
- A loop index must be an integer type. Valid types are char, short, int, long, sc_int, sc_bigint, and the unsigned version of these types.
- The loop index initial value must resolve to a constant at compile time.
- The loop index initial assignment cannot be in a conditional branch that may or may not be executed.
- The valid loop index operations are add, subtract, increment, and decrement.
- The valid loop condition test relational operators are <, <=, >, >=, and !=. The equality operator == is not useful for a loop condition test and is not supported for synthesis.
- The loop condition test limit must be a constant value or an expression that resolves to a constant at compile time.
- Loops cannot contain switch statements that have a continue statement.
- The loop condition cannot be null or empty.

Unrolled Loop

An unrolled loop replicates the code body of each loop iteration. Unrolled loops can cause longer runtimes. Figure 4-33 shows a representation of a rolled and an unrolled for loop. For RTL synthesis, all loops are unrolled.

Figure 4-33 Rolled and Unrolled for Loops

```
rolled_loop:
  for (int i=0; i<=7; i++) {
      c[i] = a[i] + b[i];
      ...
  } // end rolled_loop
```



for Loop Comma Operator

The comma (,) operator in the for loop definition in Example 4-34 is not supported for synthesis.

Example 4-34 Comma (,) Operator Is Not Supported in a for Loop

```
for (i=0, j=0; i < 6; i++, j++)
```

Dead Loops

A dead loop is a loop that never executes, and SystemC Compiler issues an error message if your code contains a dead loop. Example 4-35 shows a dead loop.

Example 4-35 Dead Loop

```
for (int i = 0; i > 0; i++) { ... }
```

Infinite Loops

SystemC Compiler issues an error message if your code contains an infinite loop. Example 4-36 shows various infinite loops.

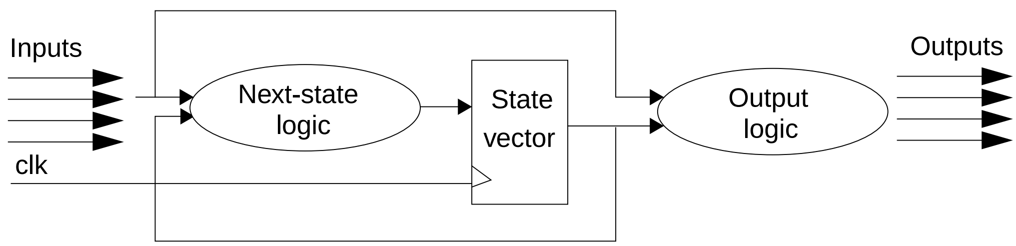
Example 4-36 Infinite Loops

```
for (int i = 1; i <= 127; i = i + 0) { ... }

for (char i = 0; i <= 127; i++) { ... }

for (char i = 0; i <= 127; i += 74) { ... }
```

State Machines

Explicitly describe state machines for RTL synthesis. Figure 4-34 shows a Mealy state machine structure.

Figure 4-34 Mealy State Machine



The diagram in Figure 4-34 has one sequential element—the state vector—and two combinational elements, the output logic and the next-state logic. Although the output logic and the next-state logic are separate in this diagram, you can merge them into one logic block in which gates can be shared for a smaller design area.

The output logic is always a function of the current state (state vector) and optionally a function of the inputs. If inputs are included in the output logic, the state machine is a Mealy state machine. If inputs are not included, the state machine is a Moore state machine.

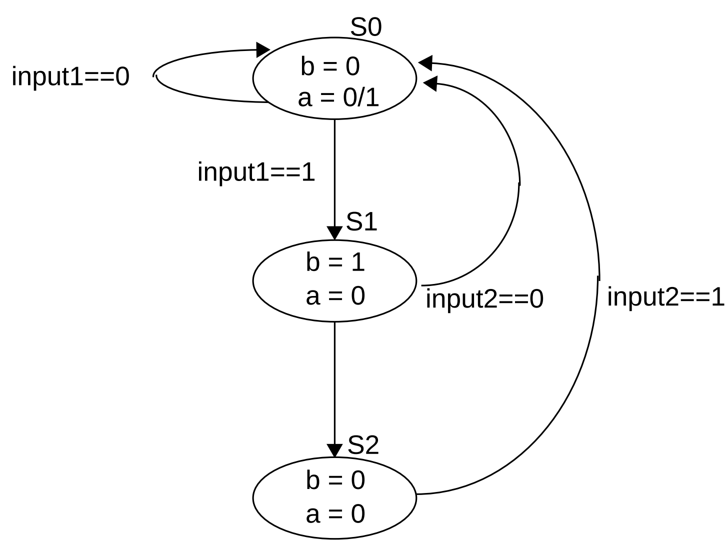
The next-state logic is always a function of the current state (state vector) and optionally a function of the inputs.

The common implementations of state machines are

- An SC_METHOD process for updating the state vector and a single common SC_METHOD process for both the output and the next-state logic
- An SC_METHOD process for the state vector, an SC_METHOD process for the output logic, and a separate SC_METHOD process for the next-state logic
- A Moore machine with a single process for computing and updating the next-state vector and outputs

Figure 4-35 shows a state diagram that represents a state machine, where a and b are outputs.

Figure 4-35 Finite State Machine State Diagram



State Machine With a Common Computation Process

Example 4-37 shows the state machine represented in Figure 4-35 with a common SC_METHOD process for computing the output and next-state logic.

Example 4-37 State Machine With Common Computation Process

```
/**ex_fsm_a.h**/
SC_MODULE(ex_fsm_a){

  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state, next_state;

  void ns_logic();
  void update_state();

  SC_CTOR(ex_fsm_a){
    SC_METHOD(update_state);
     sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << input1 << input2;
    }
  };


/**ex_fsm_a.cpp**/
#include "systemc.h"
#include "fsm_types.h"
#include "ex_fsm_a.h"

void ex_fsm_a::update_state()  {
  if (rst.read() == true){
      state = S0;
  }else{
      state = next_state;
  }
}

void ex_fsm_a::ns_logic()  {
// Determine next state
  switch(state)  {
    case S0:
         b.write(0);
         if (input1.read() || input2.read()){
```

```
            a.write(1);
         }else{
            a.write(0);
         }if (input1.read() == 1){
            next_state = S1;
         }else{
            next_state = S0;
          }
         break;
    case S1:
         a.write(0);
         b.write(1);
         if (input2.read() == 1){
            next_state = S2;
         }else{
            next_state = S0;
          }
         break;
    case S2:
         a.write(0);
         b.write(0);
         next_state = S0;
         break;
    default:
         a.write(0);
         b.write(0);
         next_state = S0;
         break;
  }
}
```

State Machine With Separate Computation Processes

Example 4-38 shows the state machine represented in Figure 4-35 with separate SC_METHOD processes for computing the output and next-state logic.

Example 4-38   State Machine With Separate Processes

```
/**ex_fsm_b.h**/
SC_MODULE(ex_fsm_b){

  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state, next_state;

  void ns_logic();
```

```
  void output_logic();
  void update_state();

  SC_CTOR(ex_fsm_b){
    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << input1 << input2;
    SC_METHOD(output_logic);
    sensitive << state << input1 << input2;
  }
};

/**ex_fsm_b.cpp**/
#include "systemc.h"
#include "fsm_types.h"
#include "ex_fsm_b.h"

void ex_fsm_b::update_state()  {
    if (rst.read() == true){
      state = S0;
    }else{
      state = next_state;
    }
}

void ex_fsm_b::ns_logic()  {

// Determine next state
  switch(state)  {
    case S0:
         if (input1.read())
            next_state = S1;
         else
            next_state = S0;
         break;
    case S1:
         if (input2.read())
            next_state = S2;
         else
            next_state = S0;
         break;
    case S2:
         next_state = S0;
         break;
    default:
         next_state = S0;
         break;
  }
}

void ex_fsm_b::output_logic(){
// determine outputs
```

```
  a.write(state == S0 && (input1.read() || input2.read() ) );
  b.write(state == S1);
}
```

Moore State Machine

Example 4-39 shows a Moore state machine with a single SC_METHOD process for computing and updating the output and next-state logic.

Example 4-39 Moore State Machine

```
/**ex_fsm_c.h**/
SC_MODULE(ex_fsm_c){

  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state;

  void update_state();

  SC_CTOR(ex_fsm_c){
    SC_METHOD(update_state);
    sensitive_pos << clk;
  }
};
/**ex_fsm_c.cpp**/
#include "systemc.h"
#include "fsm_types.h"
#include "ex_fsm_c.h"

void ex_fsm_c::update_state()  {
  if (rst.read() == true) {
    b.write(0);
    a.write(0);
    state = S0;
  } else {
    switch(state) {
    case S0:
      b.write(0);
      if (input1.read() || input2.read())
     a.write(1);
      else
     a.write(0);
      if (input1.read() == 1)
     state = S1;
      break;
```

```
    case S1:
      a.write(0);
      b.write(1);
      if(input2.read() == 1)
     state = S2;
      break;
    case S2:
      a.write(0);
      b.write(0);
      state = S0;
      break;
    }
  }
}
```

Defining a State Vector Variable

You can use the `state_vector` compiler directive to label a variable in your SystemC description as the state vector for a finite state machine. This allows SystemC Compiler to extract the labeled state vector from the SystemC description to use in reports and other output. For details about using this compiler directive, see “State Vector Compiler Directive” on page A-6.

A

Compiler Directives

This appendix provides a list of the compiler directives you can use for RTL synthesis with SystemC Compiler. It contains the following sections:

Synthesis Compiler Directives

To specify a compiler directive (also known as a pragma) in your SystemC code, insert a comment in which the first word is `synopsys`. You can use either a multiple-line comment enclosed in /* and */ characters or a single-line comment beginning with two slash (//) characters.

Table A-1 lists the compiler directives.

Table A-1 SystemC Compiler Compiler Directives

Compiler directive	Details
/* synopsys line_label *string* */	page A-3
/* synopsys infer_multibit *signal_name_list* */	page A-3
/* synopsys dont_infer_multibit *signal_name_list* */	page A-3
/* synopsys infer_mux *signal_name_list* */ /* synopsys infer_mux */	page A-4
/* synopsys dont_infer_mux *signal_name_list* */	page A-4
/* synopsys unroll */	page A-5
/* synopsys full_case */	page A-5
/* synopsys parallel_case */	page A-6
/* synopsys state_vector string */	page A-6
/* synopsys enum */	page A-8
/*synopsys synthesis_off */ and /* synopsys synthesis_on */	page A-8

Line Label Compiler Directive

Use the `line_label` compiler directive to label a loop or a line of code. In SystemC Compiler-generated reports, the label is reflected in the report hierarchy. You can also use a label with a command that sets contingencies, such as the `set_cycles` command. For example,

```
my_module2 :: entry {
         // Synopsys compiler directive
     while (true) { //synopsys line_label reset_loop2
       ...
       wait();
       ...
       wait();
     }
}
```

Instead of the `line_label` compiler directive, you can use the C/C++ line label, described in “C/C++ Line Label” on page A-9.

Multibit Inference Compiler Directives

To infer multibit implementation for individual ports or signals, add the `infer_multibit` compiler directive to individual port or signal declarations in the SystemC description, using the following syntax:

```
sc_out<sc_int<n> > port_name; /*synopsys infer_multibit "port_name"*/
```

Example 4-24 on page 4-36 is a code example that uses this compiler directive.

To prevent multibit inference for individual ports or signals, add the `dont_infer_multibit` compiler directive to individual port or signal declarations in the SystemC description, using the following syntax:

```
sc_out<sc_int<n> > port_name; /*synopsys dont_infer_multibit
"port_name"*/
```

Example 4-25 on page 4-38 shows a code example that uses this compiler directive.

Multiplexer Inference Compiler Directives

To infer a multiplexer for a switch…case statement, add the `infer_mux` compiler directive as the first line of code in the switch statement, using the following syntax:

```
switch (var) {
  //synopsys infer_mux
  ...
}
```

Example 4-28 on page 4-45 shows a code example that uses this compiler directive.

To infer a multiplexer for one or more switch…case statements within a block of code, add a line label to the block of code and use the `infer_mux` compiler directive, using the following syntax:

```
//synopsys infer_mux "label_1"

label_1: switch (var) {
   ...
}
```

Example 4-26 on page 4-40 shows a code example that uses this compiler directive.

Loop Unrolling Compiler Directive

Loops are unrolled by default for RTL synthesis with SystemC Compiler. Therefore, the `unroll` compiler directive used for behavioral synthesis with SystemC Compiler is not necessary, and it is ignored for RTL synthesis.

switch…case Compiler Directives

You can create multiple branching paths in logic with a switch…case statement.

Full Case

If you do not specify all possible branches of a switch…case statement but you know that one or more branches can never occur, you can declare a switch statement as full case with the `full_case` compiler directive. For example,

```
switch(state)  { //synopsys full_case
  case S0:
       if (input1.read())
          next_state = S1;
       else
          next_state = S0;
       break;
  case S1:
       if (input2.read())
          next_state = S2;
       else
          next_state = S0;
       break;
  case S2:
       next_state = S0;
       break;
  default:
       next_state = S0;
       break;
}
```

Example 4-17 on page 4-25 shows a code example that uses this compiler directive.

Parallel Case

SystemC Compiler automatically determines whether a switch statement is full or parallel.

All cases of a switch statement are, by definition, mutually exclusive (parallel) in C/C++. Because of this, the `parallel_case` compiler directive used by Design Compiler and other Synopsys tools is redundant. No cases overlap, by design, and a priority encoder is not necessary, so SystemC Compiler synthesizes a multiplexer.

State Vector Compiler Directive

The `state_vector` directive allows you to define the state vector of a finite state machine (and its encoding) in a SystemC description. It labels a variable in a SystemC description as the state vector for a finite state machine.

The syntax for the `state_vector` directive is

```
// synopsys state_vector vector_name
```

where *vector_name* is the variable for the state vector. This declaration allows SystemC Compiler to extract the labeled state vector from the SystemC description. Example A-1 shows one way to use the `state_vector` directive.

Note:

Do not define two `state_vector` directives in one module. Although SystemC Compiler does not issue an error message, it recognizes only the first `state_vector` directive and ignores the second.

Example A-1   Using the state_vector Compiler Directive

```
#include "systemc.h"
SC_MODULE( state_vector) {
  sc_in<sc_uint<2> > in1;
  sc_in_clk clock;
  sc_out<sc_uint<2> > out;

  sc_signal<sc_uint<2> > state;//snps state_vector state
  sc_signal<sc_uint<2> > next_state;

  void f1 ();
  void f2 ();
  void state_register ();

  SC_CTOR( state_vector ) {
    SC_METHOD( f1);
    sensitive (in1);
    sensitive (state);
    SC_METHOD( f2);
    sensitive (state);
    SC_METHOD(state_register);
    sensitive_pos << clock;
  }
};

void state_vector:: f1 ()
{
  switch (state.read ().to_uint ()) {
  case 0:
    next_state = (in1.read () +1) % 4;
    break;
  case 1:
    next_state = (in1.read ()+2) % 4;
    break;
  case 2:
    next_state = (in1.read ()+4) % 4;
    break;
  case 3:
    next_state = (in1.read ()+8) % 4;
    break;
  }
}
```

```
void state_vector:: state_register ()
{
  state = next_state;
}

void state_vector:: f2 ()
{
  out = state;
}
```

Enumerated Data Type Compiler Directive

The `enum` compiler directive is not used by SystemC Compiler. Use the C/C++ enum construct instead, as described in "Using Enumerated Data Types" on page 3-17.

Synthesis Off and On

The `synthesis_off` and `synthesis_on` compiler directives can be used to isolate simulation-specific code and prevent the code from being interpreted for synthesis. For example,

```
/* synopsys synthesis_off */
... //Simulation-only code
/* synopsys synthesis_on */
```

Use the C language #ifdef directive, described in "C/C++ Conditional Compilation" on page A-9, instead of the `synthesis_off` and `synthesis_on` directives.

C/C++ Compiler Directives

You can use C/C++ compiler directives instead of or in addition to the equivalent `synopsys` compiler directives.

C/C++ Line Label

Use the C line label instead of the `line_label` compiler directive. For example,

```
my_module1 :: entry {
     // C-style line label
     reset_loop1: while (true) {
       ...
       wait();
       ...
       wait();
    }
}
```

C/C++ Conditional Compilation

Use the C/C++ language #if, #ifdef, #ifndef, #elif, #else, and #endif conditional compilation directives to isolate blocks of code and prevent them from inclusion during synthesis or simulation.

For example,

```
    //C directive
     #ifdef SIM
     ...//Simulation-only code
     #else
     ...//Synthesis-only code
     #endif
```

B

Examples

This appendix describes several examples that demonstrate the basic concepts of RTL synthesis with SystemC Compiler. The files for these examples are available in the SystemC Compiler installation in the $SYNOPSYS/doc/syn/ccsc/ccsc_examples directory.

This appendix describes the following examples:

- Count Zeros Combinational Version
- Count Zeros Sequential Version
- FIR RTL Version
- FIR RTL and Behavioral Integrated Version
- Drink Machine

Count Zeros Combinational Version

This circuit is a combinational specification of a design problem. The circuit is given an 8-bit value, and it determines

- That the value contains exactly one sequence of zeros
- That the number of zeros in the sequence (if any)

The circuit must complete this computation in a single clock cycle. The input to the circuit is an 8-bit value. The circuit produces two outputs, the number of zeros found and an error indication.

A valid value contains only one sequence of zeros. If more than one sequence of zeros is detected, the value is invalid. A value consisting of all ones is a valid value. If a value is invalid, the count of zeros is set to zero and an error is indicated.

RTL description files are available in $SYNOPSYS/doc/syn/ccsc/ccsc_examples/count_zeros/count_zeros_comb. Table B-1 provides a list of the files.

Table B-1 RTL Count Zeros Combinational Files

File name	File description
readme_czero_combo.txt	Description of the count zeros combination version.
count_zeros_comb.h, count_zeros_comb.cc	RTL model. The RTL description has one SC_METHOD process and two member functions (legal and zeros).
count_zero_run_rtl.scr	RTL synthesis to gates command script.

Count Zeros Sequential Version

The sequential implementation of the count zeros design is slightly different from the specification in the combinational version. The circuit now accepts the 8-bit string serially, 1 bit per clock cycle, using the data and clk inputs. The other two inputs are

- The reset input, which resets the circuit by calling the defaults member function
- The read input, which causes the circuit to begin accepting data

The three outputs from the circuit are

- The is_legal output, which is true if the data is a valid value
- The data_ready output, which is true when all 8 bits are processed or at the first invalid bit
- The zeros output, which is the integer value of zeros if the is_legal output is true

RTL description files are available in $SYNOPSYS/doc/syn/ccsc/ccsc_examples/count_zeros/count_zeros_seq. Table B-2 provides a list of the files.

Table B-2 RTL Count Zeros Sequential Files

File name	File description
readme_czero_seq.txt	Description of the count zeros sequential version.
count_zeros_seq.h, count_zeros_cseq.cc	RTL model. The RTL description has three SC_METHOD processes.
count_zero_seq_run_rtl.scr	RTL synthesis to gates command script.

FIR RTL Version

The FIR filter design is a hierarchical RTL module (fir_rtl) that contains the FSM module (fir_fsm) and a data module (fir_data). Figure B-1 illustrates the modules, the port binding, and their interconnecting signals.

RTL description files are available in $SYNOPSYS/doc/syn/ccsc/ccsc_examples/fir/fir_rtl. Table B-3 provides a list of the files.

Figure B-1 FIR RTL Modules

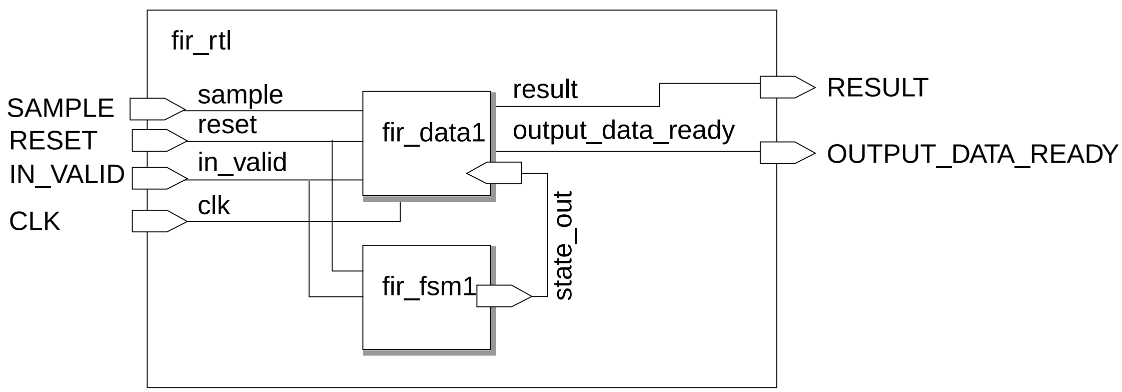


Table B-3 FIR RTL Files

File name	File description
readme_fir_rtl.txt	Description of the FIR RTL version.
fir_rtl.h, fir_rtl.cpp	RTL model, which instantiates the FSM and data modules.
fir_fsm.h, fir_fsm.cpp	FIR FSM module.
fir_data.h, fir_data.cpp	FIR data module.
fir_const_rtl.h	FIR constant coefficients.
fir_rtl_run.scr	RTL synthesis to gates command script.

FIR RTL and Behavioral Integrated Version

The FIR integrated RTL and behavioral design top-level module (all_top) contains both a behavioral module (fir_beh) and the hierarchical RTL module (fir_rtl). The inputs (sample, reset, in_valid, and clk) feed into both the RTL and behavioral modules. Figure B-2 illustrates the modules and the port bindings.

RTL and behavioral integrated description files are available in $SYNOPSYS/doc/syn/ccsc/ccsc_examples/fir/fir_integrated. Table B-4 provides a list of the files.

Figure B-2 FIR RTL and Behavioral Integrated Modules

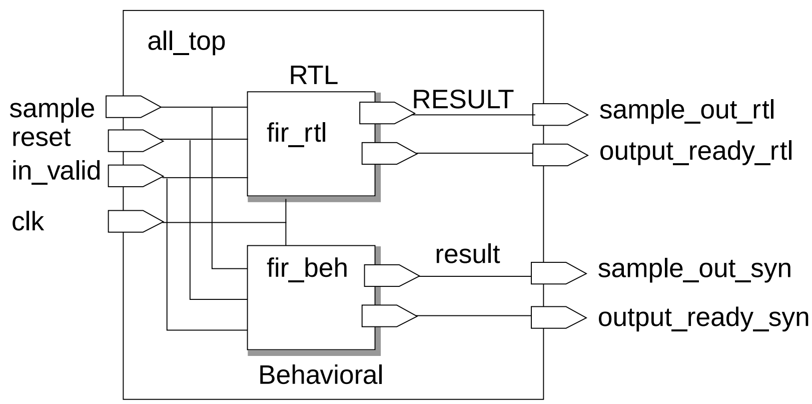


Table B-4 FIR RTL Files

File name	File description
readme_fir_int.txt	Description of the FIR RTL version.
all_top.h	Instantiations of RTL and behavioral FIR modules.
fir_beh.h, fir_beh.cpp	Behavioral model.
fir_rtl.h, fir_rtl.cpp	RTL model, which instantiates the FSM and data modules.
fir_fsm.h, fir_fsm.cpp	FIR FSM module.
fir_data.h, fir_data.cpp	FIR data module.
fir_const.h	FIR constant coefficients.
fir_const_rtl.h	FIR constant coefficients.
all_run.scr	Synthesis to gates command script.

Drink Machine

The drink machine circuit is a vending machine that dispenses drinks. It contains a state machine that counts money as input. The drink machine waits for a deposit of 35 cents and signals the vending machine to dispense a drink. If change is owed, the machine returns it when it dispenses the drink.

RTL description files are available in $SYNOPSYS/doc/syn/ccsc/ccsc_examples/drink_machine. Table B-5 provides a list of the files.

Table B-5 RTL Drink Machine Files

File name	File description
readme_drink.txt	Description of the drink machine.
drink_machine.h, drink_machine.cc	RTL model. The RTL description has two SC_METHOD processes.
drink.scr	RTL synthesis to gates command script.

Index

D

E

T

U

V

W