XST User Guide

About This Guide

This manual describes Xilinx® Synthesis Technology (XST) support for HDL languages, Xilinx® devices, and constraints for the ISE™ software. The manual also discusses FPGA and CPLD optimization techniques and explains how to run XST from Project Navigator Process window and command line.

Guide Contents

This manual contains the following chapters and appendixes.

- Chapter 1, "Introduction," provides a basic description of XST and lists supported architectures.

- Chapter 2, "HDL Coding Techniques," describes a variety of VHDL and Verilog coding techniques that can be used for various digital logic circuits, such as registers, latches, tristates, RAMs, counters, accumulators, multiplexers, decoders, and arithmetic operations. The chapter also provides coding techniques for state machines and black boxes.

- Chapter 3, "FPGA Optimization," explains how constraints can be used to optimize FPGAs and explains macro generation. The chapter also describes the Virtex™ primitives that are supported.

- Chapter 4, "CPLD Optimization," discusses CPLD synthesis options and the implementation details for macro generation.

- Chapter 5, "Design Constraints," describes constraints supported for use with XST. The chapter explains which attributes and properties can be used with FPGAs, CPLDs, VHDL, and Verilog. The chapter also explains how to set options from the Process Properties dialog box in Project Navigator.

- Chapter 6, "VHDL Language Support," explains how VHDL is supported for XST. The chapter provides details on the VHDL language, supported constructs, and synthesis options in relationship to XST.

- Chapter 7, "Verilog Language Support," describes XST support for Verilog constructs and meta comments.

- Chapter 8, "Mixed Language Support,"describes how to run an XST project that mixes Verilog and VHDL designs.

- Chapter 9, "Log File Analysis," describes the XST log file, and explains what it contains.

- Chapter 10, "Command Line Mode," describes how to run XST using the command line. The chapter describes the XST, run and set commands and their options.

- Appendix A, "XST Naming Conventions," discusses net naming and instance naming conventions.

Additional Resources

For additional information, go to http://www.xilinx.com/support. The following table lists some of the resources you can access from this website. You can also directly access these resources using the provided URLs.

Resource	Description/URL
Tutorials	Tutorials covering Xilinx® design flows, from design entry to verification and debugging http://www.xilinx.com/support/techsup/tutorials/index.htm
Answer Browser	Database of Xilinx® solution records http://www.xilinx.com/xlnx/xil_ans_browser.jsp
Application Notes	Descriptions of device-specific design techniques and approaches http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp?category=Application+Notes
Data Sheets	Pages from The Programmable Logic Data Book, which contains device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp
Problem Solvers	Interactive tools that allow you to troubleshoot your design issues http://www.xilinx.com/support/troubleshoot/psolvers.htm
Tech Tips	Latest news, design tips, and patch information for the Xilinx® design environment http://www.xilinx.com/xlnx/xil_tt_home.jsp

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
Courier font	Messages, prompts, and program files that the system displays	speed grade: – 100
Courier bold	Literal commands that you enter in a syntactical statement	**ngdbuild** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**

Convention	Meaning or Use	Example
Italic font	Variables in a syntax statement for which you must supply values	**ngdbuild** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr =**{**on**│**off**}
Vertical bar │	Separates items in a list of choices	**lowpwr =**{**on**│**off**}
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis . . .	Repetitive material that has been omitted	**allow block** *block_name loc1 loc2... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current file or in another file in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II™ Platform FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Table of Contents

Preface: About This Guide

Chapter 1: Introduction

Chapter 2: HDL Coding Techniques

Chapter 3: FPGA Optimization

Chapter 4: CPLD Optimization

Chapter 5: Design Constraints

Chapter 6: VHDL Language Support

Chapter 7: Verilog Language Support

Chapter 8: Mixed Language Support

Chapter 9: Log File Analysis

Chapter 10: Command Line Mode

Chapter 1

Introduction

This chapter contains the following sections.

- *"Architecture Support"*
- *"XST Flow"*

Architecture Support

The software supports the following architecture families in this release.

- Virtex™/-E/-II/-II Pro/-II Pro X/-4
- Spartan™-II/-IIE/-3
- CoolRunner™ XPLA3/-II
- XC9500™/XL/XV

XST Flow

XST is a Xilinx® tool that synthesizes HDL designs to create Xilinx® specific netlist files called NGC files. The NGC file is a netlist that contains both logical design data and constraints that takes the place of both EDIF and NCF files. This manual describes XST support for Xilinx® devices, HDL languages and design constraints. The manual also explains how to use various design optimization and coding techniques when creating designs for use with XST.

What's New

The following is a list of the major changes to XST for release 7.1i.

HDL Language Support

VHDL

- Support for configurations and nested configurations. See *"VHDL Language Support" in Chapter 6*.
- Support for shared variables (dual-write block RAM inference only). See *"VHDL Language Support" in Chapter 6*.
- Support for NULL arrays.
- Improved support for File Read. See *"File Type Support" in Chapter 6*.
  - Support for hexadecimal values in read operations.

♦ Support for std_ulogic type (limited to values "1" and "0").

♦ Support for character and string type read functions.

Verilog

- Improved support for while loops. See "While Loops" in Chapter 7.

- Support for variable part selects. See "Variable Part Selects" in Chapter 7.

- Support of defparams for passing attributes for Xilinx® library primitives. See "Virtex™ Primitive Support" in Chapter 3.

Macro Inference

- Support for Dynamic Shift Registers without size limitations.

- Inference of dual-write port block RAM (VHDL only). See "Dual-Port Block RAM with Two Write Ports" in Chapter 2.

- Support for parity bits usage for block RAM implementation (limited to Virtex-4™ devices). See "RAMs/ROMs" in Chapter 2.

- Support for Block RAM initialization via signal declaration mechanism. See "Initializing Block RAM" in Chapter 2.

  ♦ Initialization of dual-port block RAMs in VHDL.

  ♦ Initialization of single and dual-port block RAMs in Verilog.

- Support for Block RAM/Block ROM initialization from an external text file. See "Initializing Block RAM" in Chapter 2.

- Finite State Machine (FSM) Processing.

  ♦ Support for Safe FSM implementation. See "Safe FSM Implementation" in Chapter 2.

  ♦ Improved automatic FSM encoding selection.

  ♦ Introduced new speed oriented encoding method, called Speed1. See "State Encoding Techniques" in Chapter 2, and "Constraints Summary" in Chapter 5.

  ♦ Report of original and final FSM encoding. See Chapter 9, "Log File Analysis."

- Introduced Automatic choice between standard and KCM implementation for multiplication with constants. See "Multiplication with Constant" in Chapter 2.

- Support for CSD implementation for multiplication with constants. See "Multiplication with Constant" in Chapter 2.

- Improved register balancing of multipliers; XST can move registers with reset signal. See "Pipelined Multipliers" in Chapter 2.

- Support for Macro inference for Virtex-4™ devices and its control via USE_DSP48 constraint/command line switch. See "Using DSP48 Block Resources" in Chapter 3.

- Register balancing of DSP48 based multipliers for Virtex-4™ devices. See "Pipelined Multipliers" in Chapter 2.

- HDL Synthesis Report section of XST LOG file is now displayed after Advanced HDL Synthesis step. See "Introduction," and "Dynamic Shift Register" in Chapter 2 and "Introduction" in Chapter 9.

Design Constraints

- Removed support for old XST constraint file and private XST timing constraints.

- Support for acceptance of LOC/RLOC constraints on inferred block RAM, when block RAM requires only a single primitive. See "Specifying INITs and RLOCs in HDL Code" in Chapter 3.

- Introduced new *csd* and *pipe_block* options for Multiplier Style constraint (MULT_STYLE). See "Multiplier Style" and "Constraints Summary" in Chapter 5.

- Introduced new Safe Implementation constraint (SAFE_IMPLEMENTATION). See "Safe Implementation" and "Constraints Summary" in Chapter 5; and "Run Command" in Chapter 10.

- Introduced new Recovery State constraint (SAFE_RECOVERY_STATE) for Safe FSM Implementation. See "Recovery State" and "Constraints Summary" in Chapter 5.

- New value, *speed1*, for FSM Encoding (FSM_ENCODING) constraint. See "FSM Encoding Algorithm" in Chapter 5.

- Support for control of macro implementation on DSP48 in Virtex-4™ devices via USE_DSP48 constraint. See "Using DSP48 Block Resources" in Chapter 3.

- Multiplier Style menu in Process Properties replaced by Use DSP48 menu when working with Virtex-4™ devices. See "Considerations for Virtex-4™ Devices" in Chapter 2, and "Setting Global Constraints and Options" in Chapter 5.

- Changed behavior for Maximum Fanout (MAX_FANOUT) constraint when it applies to a specific signal. See "Max Fanout" and "Constraints Summary" in Chapter 5.

- Convert Tristates To Logic constraint (TRISTATE2LOGIC) is now active by default for new projects in Xilinx Specific Process Properties. See "Convert Tristates to Logic" and "Constraints Summary" in Chapter 5.

- Buffer Type constraint (BUFFER_TYPE) can now be applied on internal clocks. See "Virtex™ Primitive Support" in Chapter 3, "Buffer Type" and "Constraints Summary" in Chapter 5.

- Introduced new value, *bufr*, for Buffer Type constraint (BUFFER_TYPE). See "Buffer Type" in Chapter 5.

- Introduced new command line switch, –bufr, to control the number of available regional clock buffers for Virtex-4™ devices. See "Number of Regional Clock Buffers" in Chapter 5.

- Clock Signal constraint (CLOCK_SIGNAL) can now be applied to internal clocks. See "Clock Signal" and "Constraints Summary" in Chapter 5.

- Improved Box Type constraint (BOX_TYPE) support. See "Box Type" and "Constraints Summary" in Chapter 5.
    - New value, *user_black_box*, implemented.
    - BOX_TYPE is now supported via XCF file.

- Introduced new Duplication Suffix command line switch (DUPLICATION_SUFFIX) to control name generation of replicated flip-flops. See "Duplication Suffix" and "Constraints Summary" in Chapter 5; "Run Command" in Chapter 10; and "Name Generation Control" in Appendix A.

- The default value for the Hierarchy Separator command line option (HIERARCHY_SEPARATOR) for newly created projects in ISE™ Synthesis Properties is now "/". See "Hierarchy Separator" and "Constraints Summary" in Chapter 5.

- Introduced new Use Clock Enable (USE_CLOCK_ENABLE), Use Synchronous Set (USE_SYNC_SET) and Use Synchronous Reset (USE_SYNC_RESET) constraints to

prevent the usage in the final netlist flip-flops with clock enable, synchronous set and synchronous reset. See "Use Clock Enable," "Use Synchronous Set," "Use Synchronous Reset" and "Constraints Summary" in Chapter 5.

- Support for Period constraint on internal clocks.

- Improved automatic support for third party constraints (Synplicity). See "Third Party Constraints" in Chapter 5.

- Introduced new Enable Automatic Floorplanning command line switch, (ENABLE_AUTO_FLOORPLANNING), that affects Incremental Synthesis flow. See "Enable Auto Floorplanning" in Chapter 5.

- New syntax for switches supporting multiple directories. This change is related to the support of file and directory names with spaces. See "Names with Spaces" in Chapter 10.

FPGA Flow

- Support for automatic inference of BUFGs and BUFRs on most critical internal clocks controlled by BUFFER_TYPE constraint.

- Improved support for Incremental Synthesis. INCREMTENAL_SYNTHESIS constraint can now be applied to blocks that are instantiated more than once. See "Incremental Synthesis Flow" in Chapter 3.

- Improved detection and reporting of multi-source problems.

- Reporting the number of analyzed or failed paths and ports in Detailed Timing Report. See "Timing Detail" in Chapter 3.

- Detection and reporting of combinatorial loops.

- Improved quality of results under high optimization effort for speed oriented optimization.

Log File

- New Message filtering mechanism.

XST in Project Navigator

Before synthesizing your design, you can set a variety of options for XST. Following are the instructions to set the options and run XST from Project Navigator. All of these options can also be set from the command line. See Chapter 5, "Design Constraints," and Chapter 10, "Command Line Mode" for details.

1. Select your top-level design in the Source window.
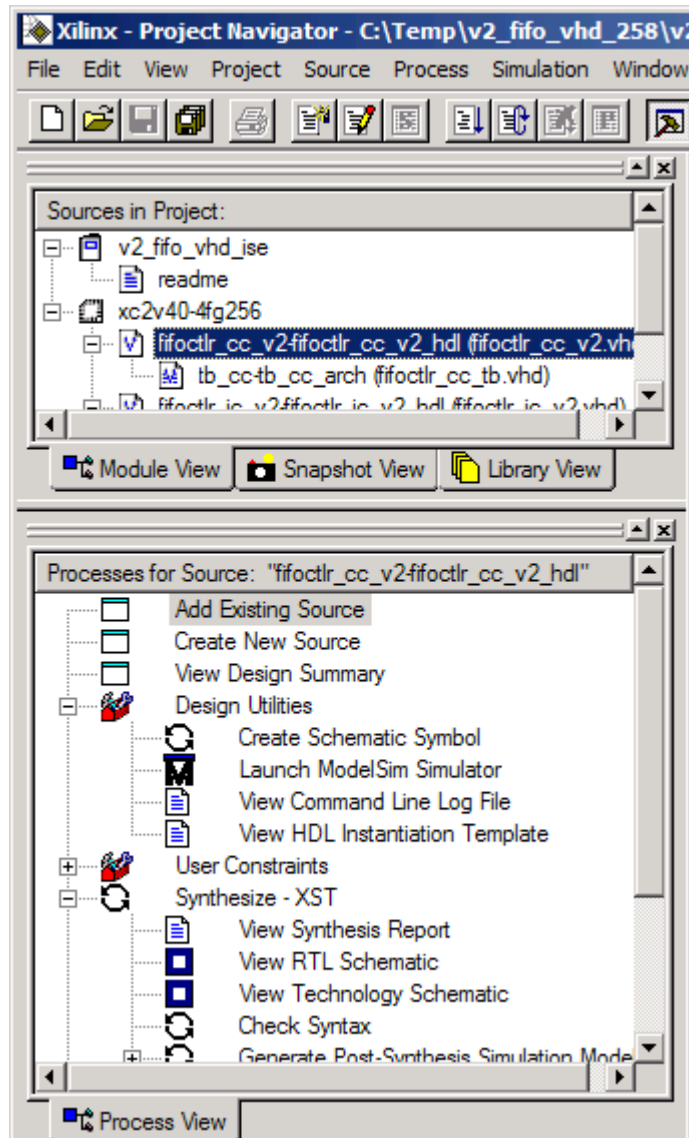
2. To set the options, right-click **Synthesize - XST** in the Process window; select **Properties** to display the Process Properties dialog box.
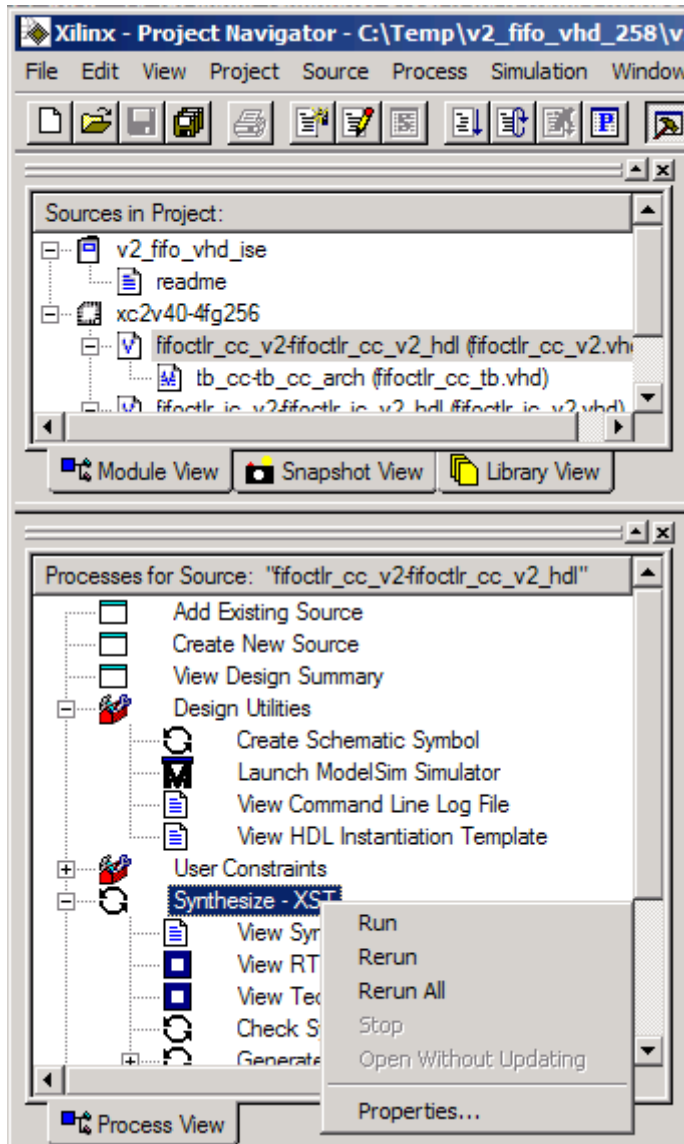
3. Set the desired Synthesis, HDL, and Xilinx® Specific Options in the Process Properties dialog box. For a complete description of these options, refer to "General Constraints" in Chapter 5.
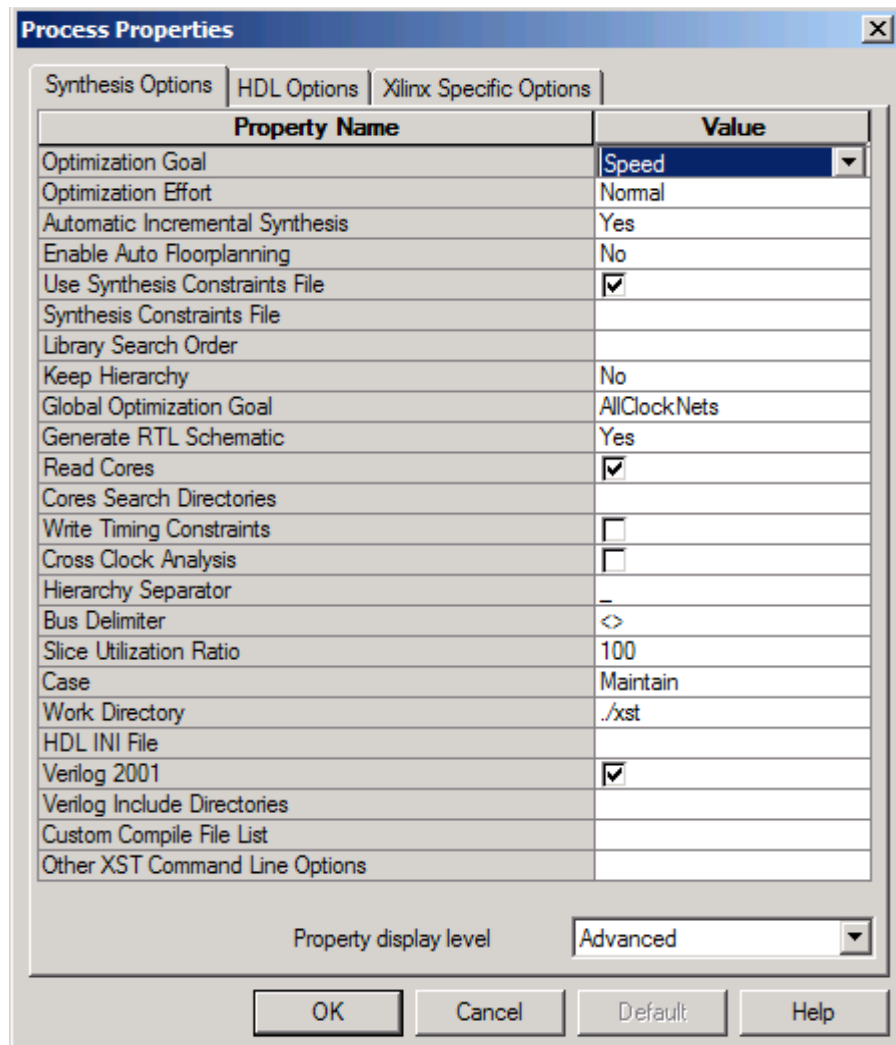
4. When a design is ready to synthesize, you can invoke XST in Project Navigator. With the top-level source file selected, double-click **Synthesize - XST** in the Process window.
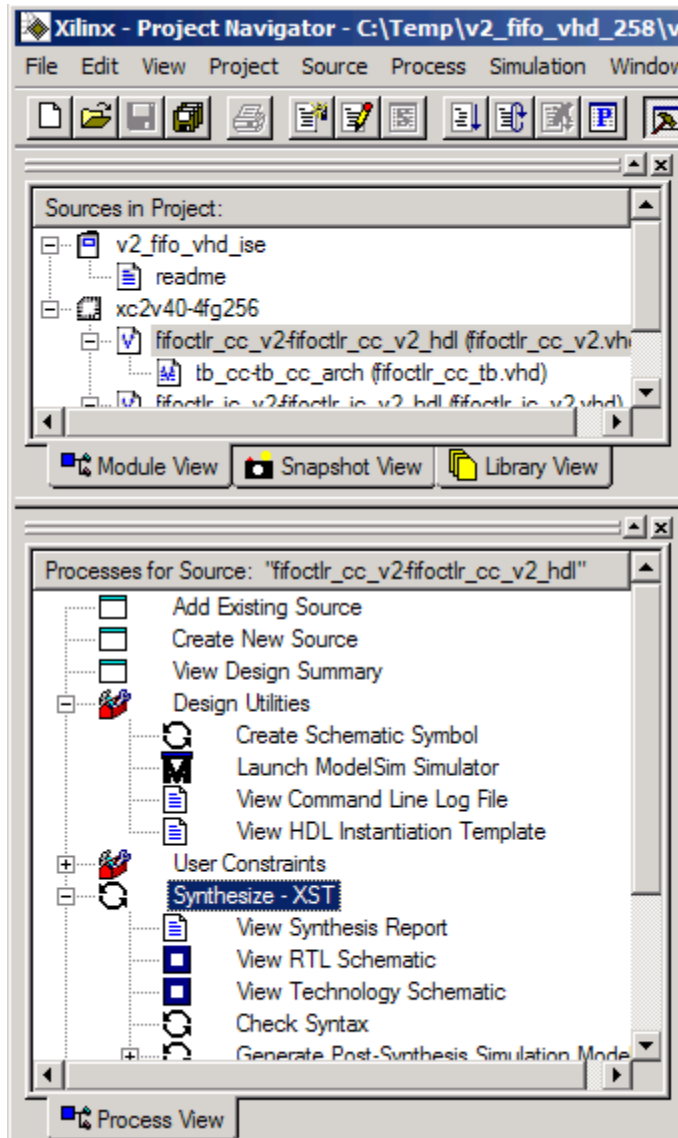
Note: To run XST from the command line, refer to Chapter 10, "Command Line Mode" for details.

5. When synthesis is complete, view the results by double-clicking **View Synthesis Report**. Following is a portion of a sample report.
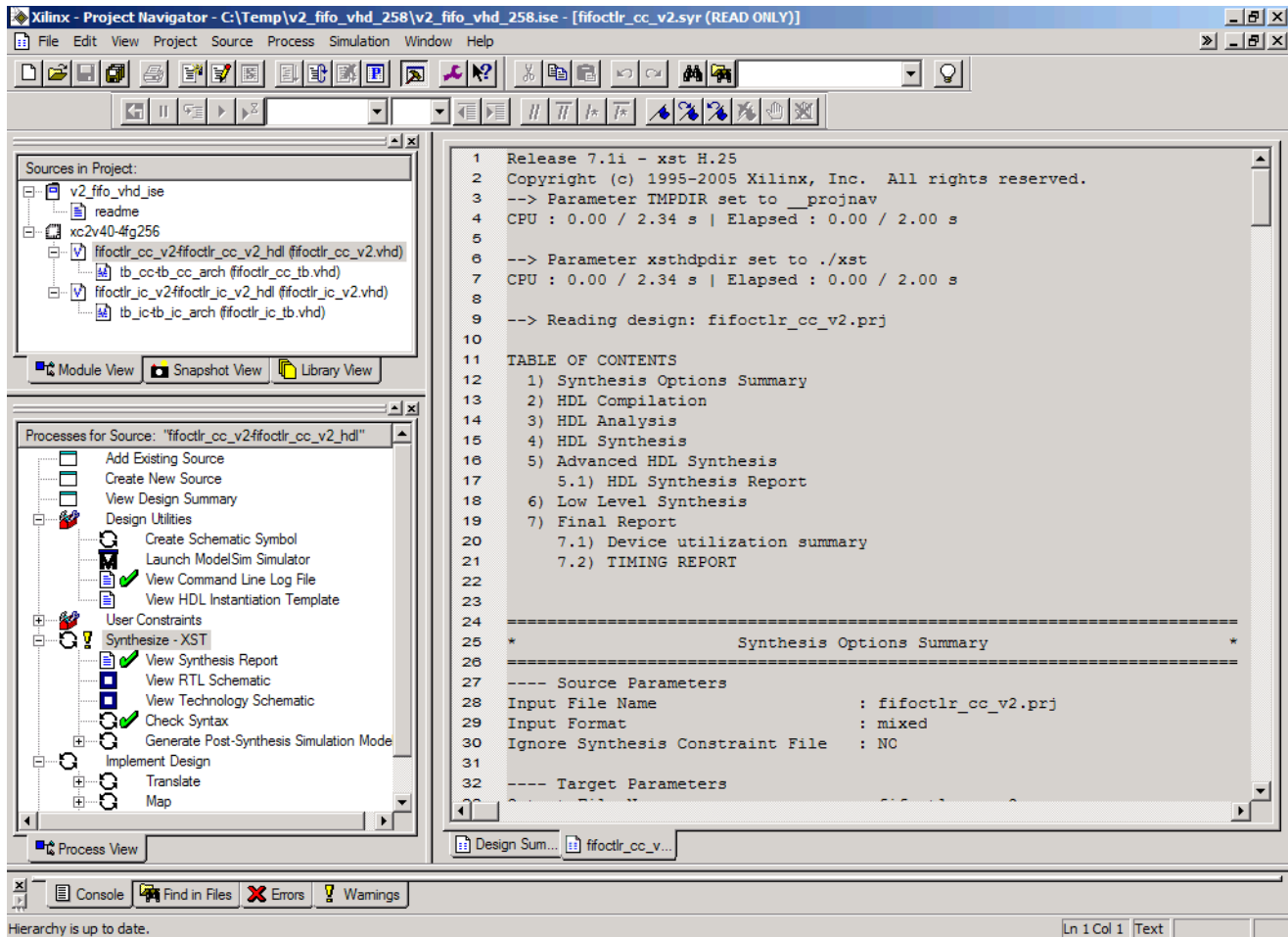
Figure 1-1: **View Synthesis Report**

Chapter 2

HDL Coding Techniques

This chapter contains the following sections:

- "Introduction"
- "Signed/Unsigned Support"
- "Registers"
- "Latches"
- "Tristates"
- "Counters"
- "Accumulators"
- "Shift Registers"
- "Dynamic Shift Register"
- "Multiplexers"
- "Decoders"
- "Priority Encoders"
- "Logical Shifters"
- "Arithmetic Operations"
- "RAMs/ROMs"
- "State Machine"
- "Safe FSM Implementation"
- "Black Box Support"

Introduction

Designs are usually made up of combinatorial logic and macros (for example, flip-flops, adders, subtractors, counters, FSMs, RAMs). The macros greatly improve performance of the synthesized designs. Therefore, it is important to use some coding techniques to model the macros so that they are optimally processed by XST.

During its run, XST first tries to recognize (infer) as many macros as possible. Then all of these macros are passed to the Low Level Optimization step, either preserved as separate blocks or merged with surrounded logic in order to get better optimization results. This filtering depends on the type and size of a macro (for example, by default, 2-to-1 multiplexers are not preserved by the optimization engine). You have full control of the processing of inferred macros through synthesis constraints.

Note: Please refer to Chapter 5, "Design Constraints," for more details on constraints and their utilization.

There is detailed information about the macro processing in the XST LOG file. It contains the following:

- The set of macros and associated signals, inferred by XST from the VHDL/Verilog source on a block by block basis.

  The macro inference is done in two steps: HDL Synthesis and Advanced HDL Synthesis. In the HDL Synthesis step, XST recognizes as many simple macro blocks as possible, such as adders, subtractors, registers, etc. In the Advanced HDL Synthesis step, XST does additional macro processing by improving the macros (for example, pipelining of multipliers) recognized at the HDL synthesis step or by creating the new, more complex ones, such as dynamic shift registers.

- The overall statistics of recognized macros.

- The number and type of macros preserved by low level optimization.

The following log sample displays the set of recognized macros on a block by block basis.

```
Synthesizing Unit <timecore>.
     Related source file is timecore.vhd.
     Found finite state machine <FSM_0> for signal <state>.
     ...
     Found 7-bit subtractor for signal <fsm_sig1>.
     Found 7-bit subtractor for signal <fsm_sig2>.
     Found 7-bit register for signal <min>.
     Found 4-bit register for signal <points_tmp>.
...
     Summary:
          inferred   1 Finite State Machine(s).
          inferred  18 D-type flip-flop(s).
          inferred  10 Adder/Subtracter(s).
Unit <timecore> synthesized.
...
Synthesizing Unit <divider>.
     Related source file is divider.vhd.
     Found 18-bit up counter for signal <counter>.
     Found 1 1-bit 2-to-1 multiplexers.
     Summary:
          inferred   1 Counter(s).
          inferred   1 Multiplexer(s).
Unit <divider> synthesized. ...
```

The following log sample displays the additional macro processing done during the Advanced HDL Synthesis step.

```
=====================================================
*              Advanced HDL Synthesis                *
=====================================================

Implementing FSM <FSM_0> on signal <current_state> on BRAM.

INFO:Xst - Data output of ROM <Mrom_tmp_one_hot> in block <decode> is
tied to register <one_hot> in block <decode>.

INFO:Xst - The register is removed and the ROM is implemented as read-
only block RAM.
...
```

The following log sample displays the overall statistics of recognized macros.

```
...
=================================================
HDL Synthesis Report

 Macro Statistics
 # FSMs                           : 1
 # ROMs                           : 4
   16x7-bit ROM                   : 4
 # Registers                      : 3
   7-bit register                 : 2
   4-bit register                 : 1
 # Counters                       : 1
   18-bit up counter              : 1
 # Multiplexers                   : 1
   2-to-1 multiplexer             : 1
 # Adders/Subtractors             : 10
   7-bit adder                    : 4
   7-bit subtractor               : 6
=================================================
 ...
```

The following log sample displays the number and type of macros preserved by low level optimization.

```
...
================================================
Final Results
 ...
 Macro Statistics
 # FSMs                               : 1
 # ROMs                               : 4
   16x7-bit ROM                       : 4
 # Registers                          : 7
   7-bit register                     : 2
   1-bit register                     : 4
   18-bit register                    : 1
 # Adders/Subtractors                 : 11
   7-bit adder                        : 4
   7-bit subtractor                   : 6
   18-bit adder                       : 1
...
================================================
...
```

This chapter discusses the following Macro Blocks:

- Registers
- Tristates
- Counters
- Accumulators
- Shift Registers
- Dynamic Shift Registers
- Multiplexers
- Decoders
- Priority Encoders
- Logical Shifters
- Arithmetic Operators (Adders, Subtractors, Adders/Subtractors, Comparators, Multipliers, Dividers)
- RAMs
- State Machines
- Black Boxes

For each macro, both VHDL and Verilog examples are given. There is also a list of constraints you can use to control the macro processing in XST.

Note: For macro implementation details please refer to Chapter 3, "FPGA Optimization" and Chapter 4, "CPLD Optimization".

Table 2-1 provides a list of all the examples in this chapter, as well as a list of VHDL and Verilog synthesis templates available from the Language Templates in Project Navigator.

To access the synthesis templates from Project Navigator:

1. Select **Edit** → **Language Templates...**
2. Click the **+** sign for either VHDL or Verilog.
3. Click the **+** sign next to Synthesis Templates.

Table 2-1:   **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Registers	Flip-flop with Positive-Edge Clock	D Flip-Flop
	Flip-flop with Negative-Edge Clock and Asynchronous Clear	D Flip-flop with Asynchronous Reset
	Flip-flop with Positive-Edge Clock and Synchronous Set	D Flip-Flop with Synchronous Reset
	Flip-flop with Positive-Edge Clock and Clock Enable	D Flip-Flop with Clock Enable
	Latch with Positive Gate	D Latch
	Latch with Positive Gate and Asynchronous Clear	D Latch with Reset
	4-bit Latch with Inverted Gate and Asynchronous Preset	
	4-bit Register with Positive-Edge Clock, Asynchronous Set and Clock Enable	
Tristates	Description Using Combinatorial Process and Always Block	Process Method (VHDL) Always Method (Verilog) Standalone Method (VHDL and Verilog)
	Description Using Concurrent Assignment	

Table 2-1: **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Counters	4-bit Unsigned Up Counter with Asynchronous Clear 4-bit Unsigned Down Counter with Synchronous Set 4-bit Unsigned Up Counter with Asynchronous Load from Primary Input 4-bit Unsigned Up Counter with Synchronous Load with a Constant 4-bit Unsigned Up Counter with Asynchronous Clear and Clock Enable 4-bit Unsigned Up/Down counter with Asynchronous Clear 4-bit Signed Up Counter with Asynchronous Reset	4-bit asynchronous counter with count enable, asynchronous reset and synchronous load
Accumulators	4-bit Unsigned Up Accumulator with Asynchronous Clear	None

Table 2-1: **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Shift Registers	8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Serial Out	4-bit Loadable Serial In Serial Out Shift Register
	8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable, Serial In and Serial Out	4-bit Serial In Parallel out Shift Register
	8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Clear, Serial In and Serial Out	4-bit Serial In Serial Out Shift Register
	8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Set, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Parallel Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out	
	8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock, Serial In and Parallel Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Serial Out	

Table 2-1: **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Shift Registers (continued)	8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Clear, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Set, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Parallel Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out	
	8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out	
	8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock, Serial In and Parallel Out	
Multiplexers	4-to-1 1-bit MUX using IF Statement	
	4-to-1 MUX Using Case Statement	4-to-1 MUX Design with CASE Statement
	4-to-1 MUX Using Tristate Buffers	4-to-1 MUX Design with Tristate Construct
	No 4-to-1 MUX	

Table 2-1: **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Decoders	VHDL (One-Hot) Verilog (One-Hot) VHDL (One-Cold) Verilog (One-Cold)	1-of-8 Decoder, Synchronous with Reset
Priority Encoders	3-Bit 1-of-9 Priority Encoder	8-to-3 encoder, Synchronous with Reset
Logical Shifters	Example 1 Example 2 Example 3	None
Dynamic Shifters	16-bit Dynamic Shift Register with Positive-Edge Clock, Serial In and Serial Out	None

Table 2-1: **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
Arithmetic Operators	Unsigned 8-bit Adder Unsigned 8-bit Adder with Carry In Unsigned 8-bit Adder with Carry Out Unsigned 8-bit Adder with Carry In and Carry Out Simple Signed 8-bit Adder Unsigned 8-bit Subtractor Unsigned 8-bit Adder/Subtractor Unsigned 8-bit Greater or Equal Comparator Unsigned 8x4-bit Multiplier Division By Constant 2 Resource Sharing	N-Bit Comparator, Synchronous with Reset

Table 2-1:   **VHDL and Verilog Examples and Templates**

Macro Blocks	Chapter Examples	Language Templates
RAMs	Single-Port RAM with Asynchronous Read	Single-Port Block RAM
	Single-Port RAM with "False" Synchronous Read	Single-Port Distributed RAM
	Single-Port RAM with Synchronous Read (Read Through)	
	Dual-Port RAM with Asynchronous Read	Dual-Port Block RAM
	Dual-Port RAM with False Synchronous Read	Dual-Port Distributed RAM
	Dual-Port RAM with Synchronous Read (Read Through)	
	Dual-Port Block RAM with Different Clocks	
	Block RAM with Reset	
	Multiple-Port RAM Descriptions	
State Machines	FSM with 1 Process	Binary State Machine
	FSM with 2 Processes	
	FSM with 3 Processes	One-Hot State Machine
Black Boxes	VHDL Code	None
	Verilog Code	

Signed/Unsigned Support

When using Verilog or VHDL in XST, some macros, such as adders or counters, can be implemented for signed and unsigned values.

For Verilog, to enable support for signed and unsigned values, you must enable Verilog-2001. You can enable it by selecting the Verilog 2001 option under the Synthesis Options tab in the Process Properties dialog box in Project Navigator, or by setting the –verilog2001 command line option to *yes*. See the "VERILOG2001" section in the *Constraints Guide* for details.

For VHDL, depending on the operation and type of the operands, you must include additional packages in your code. For example, in order to create an unsigned adder, you can use the following arithmetic packages and types that operate on unsigned values:

PACKAGE	TYPE
numeric_std	unsigned
std_logic_arith	unsigned
std_logic_unsigned	std_logic_vector

To create a signed adder you can use arithmetic packages and types that operate on signed values.

PACKAGE	TYPE
numeric_std	signed
std_logic_arith	signed
std_logic_signed	std_logic_vector

Please refer to the IEEE VHDL Manual for details on available types.

Registers

XST recognizes flip-flops with the following control signals:

- Asynchronous Set/Clear
- Synchronous Set/Clear
- Clock Enable

Log File

The XST log file reports the type and size of recognized flip-flops during the Macro Recognition step.

```
 ...
 Synthesizing Unit <flop>.
     Related source file is ff_1.vhd.
     Found 1-bit register for signal <q>.
     Summary:
         inferred   1 D-type flip-flop(s).
 Unit <flop> synthesized.
 ...
 ==============================
 HDL Synthesis Report

 Macro Statistics
 # Registers                     : 1
    1-bit register               : 1
 ==============================
 ...
```

Related Constraints

Related constraints are IOB, REGISTER_DUPLICATION, EQUIVALENT_REGISTER_REMOVAL, REGISTER_BALANCING.

Flip-flop with Positive-Edge Clock

The following figure shows a flip-flop with positive-edge clock.
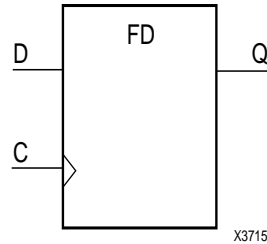
X3715

The following table shows pin definitions for a flip-flop with positive edge clock.

IO Pins	Description
D	Data Input
C	Positive Edge Clock
Q	Data Output

VHDL Code

Following is the equivalent VHDL code sample for the flip-flop with a positive-edge clock.

```
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(
    C, D : in std_logic;
    Q : out std_logic);
end flop;
architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

When using VHDL, for a positive-edge clock instead of using

```
if (C'event and C='1') then
```

you can also use

```
if (rising_edge(C)) then
```

Verilog Code

Following is the equivalent Verilog code sample for the flip-flop with a positive-edge clock.

```verilog
module flop (C, D, Q);
   input C, D;
   output Q;
   reg Q;

always @(posedge C)
   begin
     Q <= D;
   end
endmodule
```

Flip-flop with Negative-Edge Clock and Asynchronous Clear

The following figure shows a flip-flop with negative-edge clock and asynchronous clear.

X3847
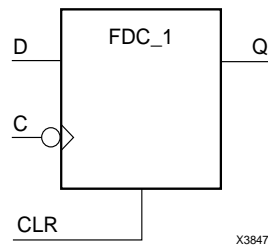
The following table shows pin definitions for a flip-flop with negative-edge clock and asynchronous clear.

IO Pins	Description
D	Data Input
C	Negative-Edge Clock
CLR	Asynchronous Clear (active High)
Q	Data Output

VHDL Code

Following is the equivalent VHDL code for a flip-flop with a negative-edge clock and asynchronous clear.

```
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(
    C, D, CLR: in std_logic;
    Q : out std_logic);
end flop;
architecture archi of flop is
begin
  process (C, CLR)
  begin
    if (CLR = '1')then
      Q <= '0';
    elsif (C'event and C='0')then
      Q <= D;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for a flip-flop with a negative-edge clock and asynchronous clear.

```
module flop (C, D, CLR, Q);
  input C, D, CLR;
  output Q;
  reg Q;

always @(negedge C or posedge CLR)
  begin
    if (CLR)
      Q <= 1'b0;
    else
      Q <= D;
  end
endmodule
```

Flip-flop with Positive-Edge Clock and Synchronous Set

The following figure shows a flip-flop with positive-edge clock and synchronous set.
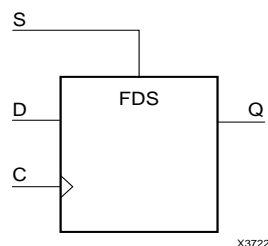
The following table shows pin definitions for a flip-flop with positive-edge clock and synchronous set.

IO Pins	Description
D	Data Input
C	Positive-Edge Clock
S	Synchronous Set (active High)
Q	Data Output

VHDL Code

Following is the equivalent VHDL code for the flip-flop with a positive-edge clock and synchronous set.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(
    C, D, S : in std_logic;
    Q : out std_logic);
end flop;
architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C='1') then
      if (S='1') then
        Q <= '1';
      else
        Q <= D;
      end if;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for the flip-flop with a positive-edge clock and synchronous set.

```verilog
module flop (C, D, S, Q);
  input C, D, S;
  output Q;
  reg Q;

always @(posedge C)
  begin
    if (S)
      Q <= 1'b1;
    else
      Q <= D;
  end
endmodule
```

Flip-flop with Positive-Edge Clock and Clock Enable

The following figure shows a flip-flop with positive-edge clock and clock enable.

X8361
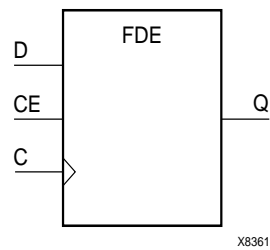
The following table shows pin definitions for a flip-flop with positive-edge clock and clock enable.

IO Pins	Description
D	Data Input
C	Positive-Edge Clock
CE	Clock Enable (active High)
Q	Data Output

VHDL Code

Following is the equivalent VHDL code for the flip-flop with a positive-edge clock and clock enable.

```
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(
    C, D, CE : in std_logic;
    Q : out std_logic);
end flop;
architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C='1') then
      if (CE='1') then
        Q <= D;
      end if;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for the flip-flop with a positive-edge clock and clock enable.

```
module flop (C, D, CE, Q);
   input C, D, CE;
   output Q;
   reg Q;

always @(posedge C)
   begin
     if (CE)
        Q <= D;
   end
endmodule
```

4-bit Register with Positive-Edge Clock, Asynchronous Set and Clock Enable

The following figure shows a 4-bit register with positive-edge clock, asynchronous set and clock enable.
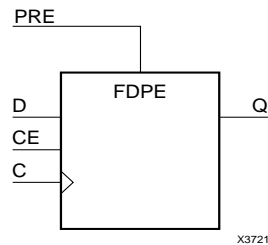
The following table shows pin definitions for a 4-bit register with positive-edge clock, asynchronous set and clock enable.

IO Pins	Description
D[3:0]	Data Input
C	Positive-Edge Clock
PRE	Asynchronous Set (active High)
CE	Clock Enable (active High)
Q[3:0]	Data Output

VHDL Code

Following is the equivalent VHDL code for a 4-bit register with a positive-edge clock, asynchronous set and clock enable.

```
library ieee;
use ieee.std_logic_1164.all;

entity flop is
  port(
    C, CE, PRE : in std_logic;
    D : in std_logic_vector (3 downto 0);
    Q : out std_logic_vector (3 downto 0));
end flop;

architecture archi of flop is
begin
  process (C, PRE)
  begin
    if (PRE='1') then
      Q <= "1111";
    elsif (C'event and C='1')then
      if (CE='1') then
        Q <= D;
      end if;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for a 4-bit register with a positive-edge clock, asynchronous set and clock enable.

```
module flop (C, D, CE, PRE, Q);
  input C, CE, PRE;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] Q;

always @(posedge C or posedge PRE)
  begin
    if (PRE)
      Q <= 4'b1111;
    else if (CE)
    Q <= D;
  end
endmodule
```

Latches

XST can recognize latches with the asynchronous set/clear control signals.

Latches can be described using:

- Process (VHDL) and always block (Verilog).
- Concurrent state assignment.

Note: XST does not support Wait statements (VHDL) for latch descriptions.

Log File

The XST log file reports the type and size of recognized latches during the Macro Recognition step.

```
   ...
   Synthesizing Unit <latch>.
       Related source file is latch_1.vhd.
   WARNING:Xst:737 - Found 1-bit latch for signal <q>.
       Summary:
             inferred   1 Latch(s).
   Unit <latch> synthesized.

   =======================================
   HDL Synthesis Report

   Macro Statistics
   # Latches                      : 1
     1-bit latch                  : 1
   =======================================
   ...
```

Related Constraints

A related constraint is IOB.

Latch with Positive Gate

The following figure shows a latch with a positive gate.

X3740
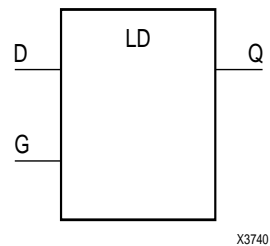
The following table shows pin definitions for a latch with a positive gate.

IO Pins	Description
D	Data Input
G	Positive Gate
Q	Data Output

VHDL Code

Following is the equivalent VHDL code for a latch with a positive gate.

```
library ieee;
use ieee.std_logic_1164.all;

entity latch is
  port(
    G, D : in std_logic;
    Q : out std_logic);
end latch;
architecture archi of latch is
begin
  process (G, D)
  begin
    if (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for a latch with a positive gate.

```
module latch (G, D, Q);
  input G, D;
  output Q;
  reg Q;

always @(G or D)
  begin
    if (G)
      Q <= D;
  end
endmodule
```

Latch with Positive Gate and Asynchronous Clear

The following figure shows a latch with a positive gate and an asynchronous clear.
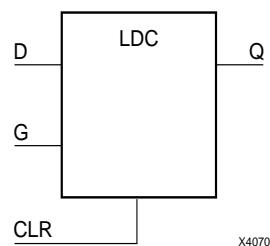
The following table shows pin definitions for a latch with a positive gate and an asynchronous clear.

IO Pins	Description
D	Data Input
G	Positive Gate
CLR	Asynchronous Clear (active High)
Q	Data Output

VHDL Code

Following is the equivalent VHDL code for a latch with a positive gate and an asynchronous clear.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity latch is
  port(
    G, D, CLR : in std_logic;
    Q : out std_logic);
end latch;

architecture archi of latch is
begin
  process (CLR, D, G)
  begin
    if (CLR='1') then
      Q <= '0';
    elsif (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for a latch with a positive gate and an asynchronous clear.

```verilog
module latch (G, D, CLR, Q);
  input G, D, CLR;
  output Q;
  reg Q;

always @(G or D or CLR)
  begin
    if (CLR)
      Q <= 1'b0;
    else if (G)
      Q <= D;
  end
endmodule
```

4-bit Latch with Inverted Gate and Asynchronous Preset

The following figure shows a 4-bit latch with an inverted gate and an asynchronous preset.

X8376
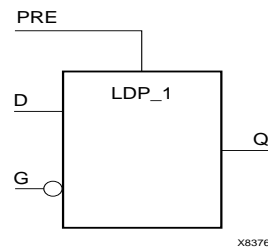
The following table shows pin definitions for a latch with an inverted gate and an asynchronous preset.

IO Pins	Description
D[3:0]	Data Input
G	Inverted Gate
PRE	Asynchronous Preset (active High)
Q[3:0]	Data Output

VHDL Code

Following is the equivalent VHDL code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
library ieee;
use ieee.std_logic_1164.all;

entity latch is
  port(
    D : in std_logic_vector(3 downto 0);
    G, PRE : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end latch;

architecture archi of latch is
begin
  process (PRE, G)
  begin
    if (PRE='1') then
      Q <= "1111";
    elsif (G='0') then
      Q <= D;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the equivalent Verilog code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
module latch (G, D, PRE, Q);
  input G, PRE;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] Q;

always @(G or D or PRE)
  begin
    if (PRE)
      Q <= 4'b1111;
    else if (~G)
      Q <= D;
  end
endmodule
```

Tristates

Tristate elements can be described using the following:

- Combinatorial process (VHDL) and always block (Verilog).
- Concurrent assignment.

Log File

The XST log reports the type and size of recognized tristates during the Macro Recognition step.

```
   ...
   Synthesizing Unit <three_st>.
        Related source file is tristates_1.vhd.
        Found 1-bit tristate buffer for signal <o>.
        Summary:
                 inferred   1 Tristate(s).
    Unit <three_st> synthesized.

   =============================
    HDL Synthesis Report

    Macro Statistics
    # Tristates                      : 1
       1-bit  tristate  buffer       : 1
   =============================
   ...
```

Related Constraints

A related constraint is TRISTATE2LOGIC.

Description Using Combinatorial Process and Always Block

The following figure shows a tristate element using a combinatorial process and always block.
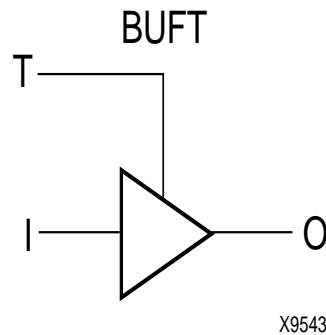
The following table shows pin definitions for a tristate element using a combinatorial process and always block.

IO Pins	Description
I	Data Input
T	Output Enable (active Low)
O	Data Output

VHDL Code

Following is VHDL code for a tristate element using a combinatorial process and always block.

```
library ieee;
use ieee.std_logic_1164.all;

entity three_st is
  port(
    T : in std_logic;
    I : in std_logic;
    O : out std_logic);
end three_st;

architecture archi of three_st is
begin
  process (I, T)
  begin
    if (T='0') then
      O <= I;
    else
      O <= 'Z';
    end if;
  end process;
end archi;
```

Verilog Code

Following is Verilog code for a tristate element using a combinatorial process and always block.

```
module three_st (T, I, O);
  input T, I;
  output O;
  reg O;

always @(T or I)
  begin
    if (~T)
      O = I;
    else
      O = 1'bZ;
  end
endmodule
```

Description Using Concurrent Assignment

In the following two examples, note that comparing to 0 instead of 1 infers a BUFT primitive instead of a BUFE macro. (The BUFE macro has an inverter on the E pin.)

VHDL Code

Following is VHDL code for a tristate element using a concurrent assignment.

```
library ieee;
use ieee.std_logic_1164.all;

entity three_st is
  port(
    T : in std_logic;
    I : in std_logic;
    O : out std_logic);
end three_st;

architecture archi of three_st is
begin
  O <= I when (T='0') else 'Z';
end archi;
```

Verilog Code

Following is the Verilog code for a tristate element using a concurrent assignment.

```
module three_st (T, I, O);
  input T, I;
  output O;

  assign O = (~T) ? I: 1'bZ;
endmodule
```

Counters

XST is able to recognize counters with the following control signals.

- Asynchronous Set/Clear
- Synchronous Set/Clear
- Asynchronous/Synchronous Load (signal and/or constant)
- Clock Enable
- Modes (Up, Down, Up/Down)
- Mixture of all of the above

HDL coding styles for the following control signals are equivalent to the ones described in "Registers" in this chapter.

- Clock
- Asynchronous Set/Clear
- Synchronous Set/Clear
- Clock Enable

Moreover, XST supports both unsigned and signed counters.

Log File

The XST log file reports the type and size of recognized counters during the Macro Recognition step.

```
...
Synthesizing Unit <counter>.
     Related source file is counters_1.vhd.
     Found 4-bit up counter for signal <tmp>.
     Summary:
          inferred   1 Counter(s).
 Unit <counter> synthesized.


==============================
 HDL Synthesis Report

 Macro Statistics
 # Counters                       : 1
   4-bit  up  counter             : 1
==============================
...
```

Note: During synthesis, XST decomposes Counters on Adders and Registers if they do not contain synchronous load signals. This is done to create additional opportunities for timing optimization. Because of this, counters reported during the Macro Recognition step and in the overall statistics of recognized macros may not appear in the final report. Adders/registers are reported instead.

Related Constraints

There are no related constraints available.

4-bit Unsigned Up Counter with Asynchronous Clear

The following table shows pin definitions for a 4-bit unsigned up counter with an asynchronous clear.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
Q[3:0]	Data Output

VHDL Code

Following is VHDL code for a 4-bit unsigned up counter with an asynchronous clear.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port(
    C, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;
architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        tmp <= "0000";
      elsif (C'event and C='1') then
        tmp <= tmp + 1;
      end if;
    end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up counter with asynchronous clear.

```verilog
module counter (C, CLR, Q);
  input C, CLR;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C or posedge CLR)
  begin
    if (CLR)
      tmp <= 4'b0000;
    else
      tmp <= tmp + 1'b1;
  end
  assign Q = tmp;
endmodule
```

4-bit Unsigned Down Counter with Synchronous Set

The following table shows pin definitions for a 4-bit unsigned down counter with a synchronous set.

IO Pins	Description
C	Positive-Edge Clock
S	Synchronous Set (active High)
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned down counter with a synchronous set.

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

entity counter is
  port(
    C, S : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        if (S='1') then
          tmp <= "1111";
        else
          tmp <= tmp - 1;
        end if;
      end if;
  end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned down counter with synchronous set.

```
module counter (C, S, Q);
   input C, S;
   output [3:0] Q;
   reg [3:0] tmp;

always @(posedge C)
   begin
     if (S)
       tmp <= 4'b1111;
     else
       tmp <= tmp - 1'b1;
   end
   assign Q = tmp;
endmodule
```

4-bit Unsigned Up Counter with Asynchronous Load from Primary Input

The following table shows pin definitions for a 4-bit unsigned up counter with an asynchronous load from the primary input.

IO Pins	Description
C	Positive-Edge Clock
ALOAD	Asynchronous Load (active High)
D[3:0]	Data Input
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

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

entity counter is
  port(
    C, ALOAD : in std_logic;
    D : in std_logic_vector(3 downto 0);
    Q : out std_logic_vector(3 downto 0));
end counter;
```

```
architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C, ALOAD, D)
    begin
      if (ALOAD='1') then
        tmp <= D;
      elsif (C'event and C='1') then
        tmp <= tmp + 1;
      end if;

  end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

```
module counter (C, ALOAD, D, Q);
  input C, ALOAD;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C or posedge ALOAD)
  begin
    if (ALOAD)
      tmp <= D;
    else
      tmp <= tmp + 1'b1;
  end
  assign Q = tmp;
endmodule
```

4-bit Unsigned Up Counter with Synchronous Load with a Constant

The following table shows pin definitions for a 4-bit unsigned up counter with a synchronous load with a constant.

IO Pins	Description
C	Positive-Edge Clock
SLOAD	Synchronous Load (active High)
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned up counter with a synchronous load with a constant.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port(
    C, SLOAD  : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        if (SLOAD='1') then
          tmp <= "1010";
        else
          tmp <= tmp + 1;
        end if;
      end if;
    end process;
    Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up counter with a synchronous load with a constant.

```verilog
module counter (C, SLOAD, Q);
  input C, SLOAD;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C)
  begin
    if (SLOAD)
      tmp <= 4'b1010;
    else
      tmp <= tmp + 1'b1;
  end
  assign Q = tmp;
endmodule
```

4-bit Unsigned Up Counter with Asynchronous Clear and Clock Enable

The following table shows pin definitions for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
CE	Clock Enable
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

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

entity counter is
  port(
    C, CLR, CE : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        tmp <= "0000";
      elsif (C'event and C='1') then
        if (CE='1') then
          tmp <= tmp + 1;
        end if;
      end if;
    end process;
    Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

```
module counter (C, CLR, CE, Q);
  input C, CLR, CE;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C or posedge CLR)
  begin
    if (CLR)
      tmp <= 4'b0000;
    else if (CE)
      tmp <= tmp + 1'b1;
  end
  assign Q = tmp;
endmodule
```

4-bit Unsigned Up/Down counter with Asynchronous Clear

The following table shows pin definitions for a 4-bit unsigned up/down counter with an asynchronous clear.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
UP_DOWN	up/down count mode selector
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned up/down counter with an asynchronous clear.

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

entity counter is
  port(
    C, CLR, UP_DOWN : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;
```

```
architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        tmp <= "0000";
      elsif (C'event and C='1') then
        if (UP_DOWN='1') then
          tmp <= tmp + 1;
        else
          tmp <= tmp - 1;
        end if;
      end if;
    end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up/down counter with an asynchronous clear.

```
module counter (C, CLR, UP_DOWN, Q);
  input C, CLR, UP_DOWN;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C or posedge CLR)
  begin
    if (CLR)
      tmp <= 4'b0000;
    else if (UP_DOWN)
      tmp <= tmp + 1'b1;
    else
      tmp <= tmp - 1'b1;
    end
  assign Q = tmp;
endmodule
```

4-bit Signed Up Counter with Asynchronous Reset

The following table shows pin definitions for a 4-bit signed up counter with an asynchronous reset.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit signed up counter with an asynchronous reset.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity counter is
  port(
    C, CLR : in std_logic;
    Q : out std_logic_vector(3 downto 0));
end counter;

architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR = '1') then
        tmp <= "0000";
      elsif (C'event and C='1') then
        tmp <= tmp + 1;
      end if;
    end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset.

```
module counter (C, CLR, Q);
  input C, CLR;
  output signed [3:0] Q;
  reg signed [3:0] tmp;

always @ (posedge C or posedge CLR)
  begin
    if (CLR)
      tmp = 4'b0000;
    else
      tmp = tmp + 1'b1;
  end
  assign Q = tmp;
endmodule
```

4-bit Signed Up Counter with Asynchronous Reset and Modulo Maximum

The following table shows pin definitions for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
Q[7:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit signed up counter with an asynchronous reset and a maximum using the VHDL mod function.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity counter is
  generic (MAX : integer := 16);
  port(
    C, CLR : in std_logic;
    Q : out integer range 0 to MAX-1);
end counter;

architecture archi of counter is
  signal cnt : integer range 0 to MAX-1;
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        cnt <= 0;
      elsif (rising_edge(C)) then
        cnt <= (cnt + 1) mod (MAX) ;
      end if;
    end process;
  Q <= cnt;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

```
module counter (C, CLR, Q);
  parameter
  MAX_SQRT = 4,
  MAX = (MAX_SQRT*MAX_SQRT);

  input C, CLR;
  output [MAX_SQRT-1:0] Q;

  reg [MAX_SQRT-1:0] cnt;

always @ (posedge C or posedge CLR)
  begin
    if (CLR)
      cnt <= 0;
    else
      cnt <= (cnt + 1) %MAX;
  end
  assign Q = cnt;
endmodule
```

Related Constraints

There are no related constraints available.

Accumulators

An accumulator differs from a counter in the nature of the operands of the add and subtract operation:

- In a counter, the destination and first operand is a signal or variable and the other operand is a constant equal to 1: A <= A + 1.

- In an accumulator, the destination and first operand is a signal or variable, and the second operand is either:

  - a signal or variable: A <= A + B.

  - a constant not equal to 1: A <= A + Constant.

An inferred accumulator can be up, down or updown. For an updown accumulator, the accumulated data may differ between the up and down mode:

```
...
if updown = '1' then
  a <= a + b;
else
  a <= a - c;
...
```

XST can infer an accumulator with the same set of control signals available for counters. (Refer to "Counters" in this chapter for more details.)

Log File

The XST log file reports the type and size of recognized accumulators during the Macro Recognition step.

```
   ...
   Synthesizing Unit <accum>.
        Related source file is accumulators_1.vhd.
        Found 4-bit up accumulator for signal <tmp>.
        Summary:
            inferred   1 Accumulator(s).
   Unit <accum> synthesized.


   ==============================
    HDL Synthesis Report

    Macro Statistics
    # Accumulators                    : 1
       4-bit up accumulator           : 1
   ==============================
   ...
```

Note: During synthesis, XST decomposes Accumulators on Adders and Registers if they do not contain synchronous load signals. This is done to create additional opportunities for timing optimization. Because of this, Accumulators reported during the Macro Recognition step and in the overall statistics of recognized macros may not appear in the final report. Adders/registers are reported instead.

Related Constraints

Related constraints are USE_DSP48 and KEEP which are available for Virtex-4™ devices.

4-bit Unsigned Up Accumulator with Asynchronous Clear

The following table shows pin definitions for a 4-bit unsigned up accumulator with an asynchronous clear.

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Clear (active High)
D[3:0]	Data Input
Q[3:0]	Data Output

VHDL Code

Following is the VHDL code for a 4-bit unsigned up accumulator with an asynchronous clear.

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

entity accum is
  port(
    C, CLR : in std_logic;
    D : in std_logic_vector(3 downto 0);
    Q : out std_logic_vector(3 downto 0));
end accum;

architecture archi of accum is
  signal tmp : std_logic_vector(3 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        tmp <= "0000";
      elsif (C'event and C='1') then
        tmp <= tmp + D;
      end if;
    end process;
  Q <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-bit unsigned up accumulator with an asynchronous clear.

```
module accum (C, CLR, D, Q);
  input C, CLR;
  input [3:0] D;
  output [3:0] Q;
  reg [3:0] tmp;

always @(posedge C or posedge CLR)
  begin
    if (CLR)
      tmp = 4'b0000;
    else
      tmp = tmp + D;
  end
  assign Q = tmp;
endmodule
```

Considerations for Virtex-4™ Devices

The Virtex-4™ family enables accumulators to be implemented on DSP48 resources. XST can push up to 2 levels of input registers into DSP48 blocks.

XST can implement an accumulator in a DSP48 block if its implementation requires only a single DSP48 resource. If an accumulator macro does not fit in a single DSP48, XST will implement the entire macro using slice logic.

Macro implementation on DSP48 resources is controlled by the USE_DSP48 constraint/command line option, with a default value of *auto.* In this mode, XST implements accumulators taking into account the number of available DSP48 resources on the device.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers as possible in the DSP48. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if you want to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

As with other families, for Virtex-4™, XST reports the details of inferred accumulators at the HDL Synthesis step. But in the Final Synthesis Report, accumulators are no longer visible, because they are implemented within the MAC implementation mechanism.

Shift Registers

In general, a shift register is characterized by the following control and data signals, which are fully recognized by XST.

- clock
- serial input
- asynchronous set/reset
- synchronous set/reset
- synchronous/asynchronous parallel load
- clock enable
- serial or parallel output. The shift register output mode may be:
  - ♦ serial: only the contents of the last flip-flop are accessed by the rest of the circuit
  - ♦ parallel: the contents of one or several flip-flops, other than the last one, are accessed
- shift modes: left, right, etc.

There are different ways to describe shift registers. For example, in VHDL you can use:

- concatenation operator

```
shreg <= shreg (6 downto 0) & SI;
```

- "for loop" construct

```
for i in 0 to 6 loop
  shreg(i+1) <= shreg(i);
end loop;
shreg(0) <= SI;
```

- predefined shift operators; for example, SLL or SRL

Consult the VHDL/Verilog language reference manuals for more information.

FPGAs:

Virtex™/-E/-II/-II Pro/-II Pro X/4 and Spartan™-II/-IIE/-3 have specific hardware resources to implement shift registers: SRL16 for Virtex™ /-E/-II/-II Pro/-II Pro X and Spartan™-II/-IIE/-3 and SRLC16 for Virtex™-II/-II Pro/-II Pro X/4 and Spartan-3™. Both are available with or without a clock enable. The following figure shows the pin layout of SRL16E.
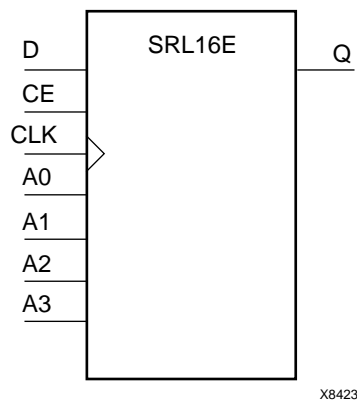
The following figure shows the pin layout of SRLC16.
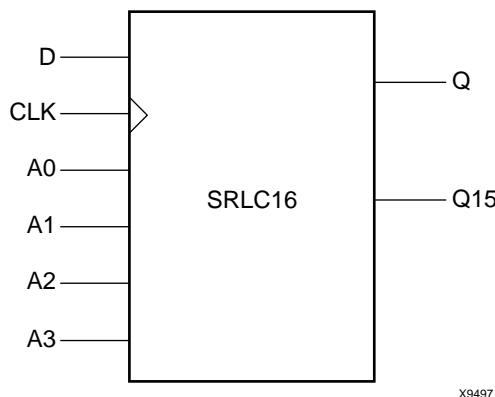
Note: Synchronous and asynchronous control signals are not available in the SLRC16x primitives.

SRL16 and SRLC16 support only LEFT shift operation for a limited number of IO signals.

- clock
- clock enable
- serial data in
- serial data out

This means that if your shift register *does have*, for instance, a synchronous parallel load, no SRL16 is implemented. XST uses specific internal processing which enables it to produce the best final results.

The XST log file reports recognized shift registers when it can be implemented using SRL16.

Log File

The XST log file reports the type and size of recognized shift registers during the Macro Recognition step.

```
   ...
   Synthesizing Unit <shift>.
        Related source file is shift_registers_1.vhd.
        Found 8-bit shift register for signal <tmp<7>>.
        Summary:
             inferred   1 Shift register(s).
    Unit <shift> synthesized.


   ==============================
    HDL Synthesis Report

    Macro Statistics
    # Shift Registers                    : 1
      8-bit shift register               : 1
   ==============================
   ...
```

Related Constraints

A related constraint is SHREG_EXTRACT.

8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Serial Out

Note: For this example, XST infers an SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
SO	Serial Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, a serial in and a serial out.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI : in std_logic;
    SO : out std_logic);
end shift;

architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        for i in 0 to 6 loop
          tmp(i+1) <= tmp(i);
        end loop;
        tmp(0) <= SI;
      end if;
    end process;
  SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, serial in and serial out.

```verilog
module shift (C, SI, SO);
  input C,SI;
  output SO;
  reg [7:0] tmp;

always @(posedge C)
  begin
    tmp <= tmp << 1;
    tmp[0] <= SI;
  end
  assign SO = tmp[7];
endmodule
```

8-bit Shift-Left Register with Negative-Edge Clock, Clock Enable, Serial In and Serial Out

Note: For this example, XST infers an SRL16E_1.

The following table shows pin definitions for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

IO Pins	Description
C	Negative-Edge Clock
SI	Serial In
CE	Clock Enable (active High)
SO	Serial Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI, CE : in std_logic;
    SO : out std_logic);
end shift;

architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='0') then
        if (CE='1') then
          for i in 0 to 6 loop
            tmp(i+1) <= tmp(i);
          end loop;
          tmp(0) <= SI;
        end if;
      end if;
    end process;
  SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```verilog
module shift (C, CE, SI, SO);
   input C, SI, CE;
   output SO;
   reg [7:0] tmp;

always @(negedge C)
   begin
      if (CE)
         begin
            tmp <= tmp << 1;
            tmp[0] <= SI;
         end
   end
   assign SO = tmp[7];
endmodule
```

8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Clear, Serial In and Serial Out

Note: Because this example includes an asynchronous clear, XST does **not** infer an SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, an asynchronous clear, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
CLR	Asynchronous Clear (active High)
SO	Serial Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, an asynchronous clear, a serial in and a serial out.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift is
   port(
      C, SI, CLR : in std_logic;
      SO : out std_logic);
end shift;
```

```
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C, CLR)
    begin
      if (CLR='1') then
        tmp <= (others => '0');
      elsif (C'event and C='1') then
        tmp <= tmp(6 downto 0) & SI;
      end if;
    end process;
  SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, asynchronous clear, serial in and serial out.

```
module shift (C, CLR, SI, SO);
  input C, SI, CLR;
  output SO;
  reg [7:0] tmp;

always @(posedge C or posedge CLR)
  begin
    if (CLR)
      tmp <= 8'b00000000;
    else
      begin
        tmp <= {tmp[6:0], SI};
      end
  end
  assign SO = tmp[7];
endmodule
```

8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Set, Serial In and Serial Out

Note: For this example, XST does **not** infer an SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
S	Synchronous Set (active High)
SO	Serial Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI, S : in std_logic;
    SO : out std_logic);
end shift;

architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C, S)
    begin
      if (C'event and C='1') then
        if (S='1') then
          tmp <= (others => '1');
        else
          tmp <= tmp(6 downto 0) & SI;
        end if;
      end if;
    end process;
  SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

```verilog
module shift (C, S, SI, SO);
  input C, SI, S;
  output SO;
  reg [7:0] tmp;

always @(posedge C)
  begin
    if (S)
      tmp <= 8'b11111111;
    else
      begin
        tmp <= {tmp[6:0], SI};
      end
  end
  assign SO = tmp[7];
endmodule
```

8-bit Shift-Left Register with Positive-Edge Clock, Serial In and Parallel Out

Note: For this example, XST does **not** infer SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
PO[7:0]	Parallel Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI : in std_logic;
    PO : out std_logic_vector(7 downto 0));
end shift;

architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        tmp <= tmp(6 downto 0)& SI;
      end if;
    end process;
  PO <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
module shift (C, SI, PO);
  input C, SI;
  output [7:0] PO;
  reg [7:0] tmp;

always @(posedge C)
  begin
    tmp <= {tmp[6:0], SI};
  end
  assign PO = tmp;
endmodule
```

8-bit Shift-Left Register with Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out

Note: For this example, XST does **not** infer SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
ALOAD	Asynchronous Parallel Load (active High)
D[7:0]	Data Input
SO	Serial Output

VHDL Code

Following is VHDL code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
library ieee;
use ieee.std_logic_1164.all;
entity shift is
  port(
    C, SI, ALOAD : in std_logic;
    D  : in std_logic_vector(7 downto 0);
    SO : out std_logic);
end shift;
architecture archi of shift is
  signal tmp : std_logic_vector(7 downto 0);
  begin
    process (C, ALOAD, D)
    begin
      if (ALOAD='1') then
        tmp <= D;
      elsif (C'event and C='1') then
        tmp <= tmp(6 downto 0) & SI;
      end if;
    end process;
  SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
module shift (C, ALOAD, SI, D, SO);
  input C, SI, ALOAD;
  input [7:0] D;
  output SO;
  reg [7:0] tmp;

always @(posedge C or posedge ALOAD)
  begin
    if (ALOAD)
      tmp <= D;
    else
      tmp <= {tmp[6:0], SI};
  end
  assign SO = tmp[7];
endmodule
```

8-bit Shift-Left Register with Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out

Note: For this example, XST does **not** infer SRL16.

The following table shows pin definitions for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
SLOAD	Synchronous Parallel Load (active High)
D[7:0]	Data Input
SO	Serial Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left register with a positive-edge clock, synchronous parallel load, serial in and serial out.

```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI, SLOAD : in std_logic;
    D  : in std_logic_vector(7 downto 0);
    SO : out std_logic);
end shift;
```

```
architecture archi of shift is
  signal tmp: std_logic_vector(7 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        if (SLOAD='1') then
          tmp <= D;
        else
          tmp <= tmp(6 downto 0) & SI;
        end if;
      end if;
    end process;
    SO <= tmp(7);
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

```
module shift (C, SLOAD, SI, D, SO);
  input C, SI, SLOAD;
  input [7:0] D;
  output SO;
  reg [7:0] tmp;

always @(posedge C)
  begin
    if (SLOAD)
      tmp <= D;
    else
      tmp <= {tmp[6:0], SI};
  end
  assign SO = tmp[7];
endmodule
```

8-bit Shift-Left/Shift-Right Register with Positive-Edge Clock, Serial In and Parallel Out

Note: For this example, XST does **not** infer an SRL16.

The following table shows pin definitions for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
LEFT_RIGHT	Left/right shift mode selector
PO[7:0]	Parallel Output

VHDL Code

Following is the VHDL code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
library ieee;
use ieee.std_logic_1164.all;

entity shift is
  port(
    C, SI, LEFT_RIGHT : in std_logic;
    PO : out std_logic_vector(7 downto 0));
end shift;

architecture archi of shift is
  signal tmp : std_logic_vector(7 downto 0);
  begin
    process (C)
    begin
      if (C'event and C='1') then
        if (LEFT_RIGHT='0') then
          tmp <= tmp(6 downto 0) & SI;
        else
          tmp <= SI & tmp(7 downto 1);
        end if;
      end if;
    end process;
    PO <= tmp;
end archi;
```

Verilog Code

Following is the Verilog code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
module shift (C, SI, LEFT_RIGHT, PO);
  input C, SI, LEFT_RIGHT;
  output PO;
  reg [7:0] tmp;

always @(posedge C)
  begin
    if (LEFT_RIGHT == 1'b0)
      tmp <= {tmp[6:0], SI};
    else
      tmp <= {SI, tmp[7:1]};
  end
  assign PO = tmp;
endmodule
```

Dynamic Shift Register

XST can infer Dynamic shift registers. Once a dynamic shift register has been identified, its characteristics are handed to the XST macro generator for optimal implementation using SRL16x primitives available in Spartan™-II/-IIE/-3, Virtex™/-II/-II Pro/-II Pro X/-4 or SRLC16x in Virtex™-II/-II Pro/-II Pro X/-4 and Spartan-3™.

16-bit Dynamic Shift Register with Positive-Edge Clock, Serial In and Serial Out

The following table shows pin definitions for a dynamic register. The register can be either serial or parallel; be left or right; have a synchronous or asynchronous clear; and have a depth up to **16 bits**.

IO Pins	Description
Clk	Positive-Edge Clock
SI	Serial In
AClr	Asynchronous Clear (optional)
SClr	Synchronous Clear (optional)
SLoad	Synchronous Parallel Load (optional)
Data	Parallel Data Input Port (optional)
ClkEn	Clock Enable (optional)
LeftRight	Direction selection (optional)
SerialInRight	Serial Input Right for Bidirectional Shift Register (optional)
PSO[x:0]	Serial or Parallel Output

LOG File

The recognition of dynamic shift registers happens in the Advanced HDL Synthesis step. The XST log file reports the size of recognized dynamic shift registers during the Macro Recognition step:

```
...
...
=============================================
*              HDL Synthesis               *
=============================================

Synthesizing Unit <dynamic_srl>.
   Related source file is vlg.v.
   Found 1-bit 16-to-1 multiplexer for signal <Q>.
   Found 16-bit register for signal <data>.
   Summary:
 inferred   16 D-type flip-flop(s).
 inferred    1 Multiplexer(s).
Unit <dynamic_srl> synthesized.

=============================================
*          Advanced HDL Synthesis          *
=============================================
...
Dynamic shift register inference ...

=============================================
HDL Synthesis Report

Macro Statistics
# Shift Registers                   : 1
 16-bit dynamic shift register      : 1

=============================================
...
```

Related Constraints

A related constraint is SHREG_EXTRACT.

VHDL Code

Following is the VHDL code for a 16-bit dynamic shift register.

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

entity shiftregluts is
  port(
    CLK  : in std_logic;
    DATA : in std_logic;
    CE   : in std_logic;
    A    : in std_logic_vector(3 downto 0);
    Q    : out std_logic);
end shiftregluts;
```

```
architecture rtl of shiftregluts is
  constant DEPTH_WIDTH : integer := 16;
  type SRL_ARRAY is array (0 to DEPTH_WIDTH-1) of std_logic;
-- The type SRL_ARRAY can be array
-- (0 to DEPTH_WIDTH-1) of
-- std_logic_vector(BUS_WIDTH downto 0)
-- or array (DEPTH_WIDTH-1 downto 0) of
-- std_logic_vector(BUS_WIDTH downto 0)
-- (the subtype is forward (see below))

  signal SRL_SIG : SRL_ARRAY;

  begin
    PROC_SRL16 : process (CLK)
    begin
      if (CLK'event and CLK = '1') then
        if (CE = '1') then
          SRL_SIG <= DATA & SRL_SIG(0 to DEPTH_WIDTH-2);
        end if;
      end if;
    end process;
    Q <= SRL_SIG(conv_integer(A));
end rtl;
```

Verilog Code

Following is the Verilog code for a 16-bit dynamic shift register.

```
module dynamic_srl (Q,CE,CLK,D,A);
  input CLK, D, CE;
  input [3:0] A;
  output Q;
  reg [15:0] data;
  assign Q = data[A];

always @(posedge CLK)
  begin
    if (CE == 1'b1)
      data <= {data[14:0], D};
  end

endmodule
```

Multiplexers

XST supports different description styles for multiplexers (MUXs), such as If-Then-Else or Case. When writing MUXs, you must pay particular attention in order to avoid common traps. For example, if you describe a MUX using a Case statement, and you do not specify all values of the selector, you may get latches instead of a multiplexer. Writing MUXs you can also use "don't cares" to describe selector values.

During the Macro Inference step, XST makes a decision to infer or not infer the MUXs. For example, if the MUX has several inputs that are the same, then XST can decide not to infer it. If you do want to infer the MUX, you can force XST by using the design constraint called MUX_EXTRACT.

If you use Verilog, then you must be aware that Verilog Case statements can be full or not full, and they can also be parallel or not parallel. A Case statement is:

- FULL if all possible branches are specified.
- PARALLEL if it does not contain branches that can be executed simultaneously.

The following tables gives three examples of Case statements with different characteristics.

Full and Parallel Case

```
module full (sel, i1, i2, i3, i4, o1);
input [1:0] sel;
input [1:0] i1, i2, i3, i4;
output [1:0] o1;

  reg [1:0] o1;

always @(sel or i1 or i2 or i3 or i4)
  begin
    case (sel)
      2'b00:  o1 = i1;
      2'b01:  o1 = i2;
      2'b10:  o1 = i3;
      2'b11:  o1 = i4;
      endcase
    end
endmodule
```

Not Full but Parallel

```
module notfull (sel, i1, i2, i3, o1);
   input [1:0] sel;
   input [1:0] i1, i2, i3;
   output [1:0] o1;

   reg [1:0] o1;

always @(sel or i1 or i2 or i3)
  begin
    case (sel)
      2'b00:  o1 = i1;
      2'b01:  o1 = i2;
      2'b10:  o1 = i3;
    endcase
  end
endmodule
```

```
Neither Full nor Parallel

   module notfull_notparallel (sel1, sel2, i1, i2, o1);
      input [1:0] sel1, sel2;
      input [1:0] i1, i2;
      output [1:0] o1;

      reg [1:0] o1;

   always @(sel1 or sel2)
      begin
        case (2'b00)
          sel1:  o1 = i1;
          sel2:  o1 = i2;
        endcase
      end
   endmodule
```

XST automatically determines the characteristics of the Case statements and generates logic using multiplexers, priority encoders and latches that best implement the exact behavior of the Case statement.

This characterization of the Case statements can be guided or modified by using the Case Implementation Style parameter. Please refer to the Chapter 5, "Design Constraints" for more details. Accepted values for this parameter are *none*, *full*, *parallel* and *full-parallel*.

- If none is used (the default), XST implements the exact behavior of the Case statements.
- If full is used, XST considers that Case statements are complete and avoids latch creation.
- If parallel is used, XST considers that the branches cannot occur in parallel and does not use a priority encoder.
- If full-parallel is used, XST considers that Case statements are complete and that the branches cannot occur in parallel, therefore saving latches and priority encoders.

The following table indicates the *resources* used to synthesize the three examples above using the four Case Implementation Styles. The term "resources" means the functionality. For example, if you code the Case statement neither full nor parallel with Case Implementation Style set to *none*, from the functionality point of view, XST implements a priority encoder + latch. But, it does not inevitably mean that XST infers the priority encoder during the Macro Recognition step.

Parameter Value	Case Implementation		
	Full	Not Full	Neither Full nor Parallel
none	MUX	Latch	Priority Encoder + Latch
parallel	MUX	Latch	Latch
full	MUX	MUX	Priority Encoder
full-parallel	MUX	MUX	MUX

Note: Specifying full, parallel or full-parallel may result in an implementation with a behavior that may differ from the behavior of the initial model.

Log File

The XST log file reports the type and size of recognized MUXs during the Macro Recognition step.

```
...
Synthesizing Unit <mux>.
    Related source file is multiplexers_1.vhd.
    Found 1-bit 4-to-1 multiplexer for signal <o>.
    Summary:
        inferred   1 Multiplexer(s).
Unit <mux> synthesized.

==============================
 HDL Synthesis Report

 Macro Statistics
 # Multiplexers                    : 1
    1-bit 4-to-1 multiplexer       : 1
==============================
...
```

Related Constraints

Related constraints are MUX_EXTRACT and MUX_STYLE.

4-to-1 1-bit MUX using IF Statement

The following table shows pin definitions for a 4-to-1 1-bit MUX using an If statement.

IO Pins	Description
a, b, c, d	Data Inputs
s[1:0]	MUX selector
o	Data Output

VHDL Code

Following is the VHDL code for a 4-to-1 1-bit MUX using an If statement.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
  port (
    a, b, c, d : in std_logic;
    s : in std_logic_vector (1 downto 0);
    o : out std_logic);
end mux;
```

```
architecture archi of mux is
  begin
    process (a, b, c, d, s)
    begin
      if (s = "00") then
        o <= a;
      elsif (s = "01") then
        o <= b;
      elsif (s = "10") then
        o <= c;
      else
        o <= d;
      end if;
    end process;
end archi;
```

Verilog Code

Following is the Verilog code for a 4-to-1 1-bit MUX using an If statement.

```
module mux (a, b, c, d, s, o);
  input a,b,c,d;
  input [1:0] s;
  output o;
  reg o;

always @(a or b or c or d or s)
  begin
    if (s == 2'b00)
      o = a;
    else if (s == 2'b01)
      o = b;
    else if (s == 2'b10)
      o = c;
    else
      o = d;
  end
endmodule
```

4-to-1 MUX Using Case Statement

The following table shows pin definitions for a 4-to-1 1-bit MUX using a Case statement.

IO Pins	Description
a, b, c, d	Data Inputs
s[1:0]	MUX selector
o	Data Output

VHDL Code

Following is the VHDL code for a 4-to-1 1-bit MUX using a Case statement.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
  port (
    a, b, c, d : in std_logic;
    s : in std_logic_vector (1 downto 0);
    o : out std_logic);
end mux;

architecture archi of mux is
  begin
    process (a, b, c, d, s)
    begin
      case s is
        when "00" => o <= a;
        when "01" => o <= b;
        when "10" => o <= c;
        when others => o <= d;
      end case;
  end process;
end archi;
```

Verilog Code

Following is the Verilog Code for a 4-to-1 1-bit MUX using a Case statement.

```
module mux (a, b, c, d, s, o);
  input a, b, c, d;
  input [1:0] s;
  output o;
  reg o;

always @(a or b or c or d or s)
  begin
    case (s)
      2'b00   : o = a;
      2'b01   : o = b;
      2'b10   : o = c;
      default : o = d;
    endcase
  end
endmodule
```

4-to-1 MUX Using Tristate Buffers

The following table shows pin definitions for a 4-to-1 1-bit MUX using tristate buffers.

IO Pins	Description
a, b, c, d	Data Inputs
s[3:0]	MUX Selector
o	Data Output

VHDL Code

Following is the VHDL code for a 4-to-1 1-bit MUX using tristate buffers.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
  port (
    a, b, c, d : in std_logic;
    s : in std_logic_vector (3 downto 0);
    o : out std_logic);
end mux;

architecture archi of mux is
begin
  o <= a when (s(0)='0') else 'Z';
  o <= b when (s(1)='0') else 'Z';
  o <= c when (s(2)='0') else 'Z';
  o <= d when (s(3)='0') else 'Z';
end archi;
```

Verilog Code

Following is the Verilog Code for a 4-to-1 1-bit MUX using tristate buffers.

```
module mux (a, b, c, d, s, o);
  input a, b, c, d;
  input [3:0] s;
  output o;

  assign o = s[3] ? a :1'bz;
  assign o = s[2] ? b :1'bz;
  assign o = s[1] ? c :1'bz;
  assign o = s[0] ? d :1'bz;
endmodule
```

No 4-to-1 MUX

The following example does not generate a 4-to-1 1-bit MUX, but a 3-to-1 MUX with 1-bit latch. The reason is that not all selector values were described in the If statement. It is supposed that for the s=11 case, "O" keeps its old value, and therefore a memory element is needed.

The following table shows pin definitions for a 3-to-1 1-bit MUX with a 1-bit latch.

IO Pins	Description
a, b, c, d	Data Inputs
s[1:0]	Selector
o	Data Output

VHDL Code

Following is the VHDL code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
  port (
    a, b, c, d : in std_logic;
    s : in std_logic_vector (1 downto 0);
    o : out std_logic);
end mux;

architecture archi of mux is
begin
  process (a, b, c, d, s)
  begin
    if (s = "00") then
      o <= a;
    elsif (s = "01") then
      o <= b;
    elsif (s = "10") then
      o <= c;
    end if;
  end process;
end archi;
```

Verilog Code

Following is the Verilog code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
module mux (a, b, c, d, s, o);
  input a, b, c, d;
  input [1:0] s;
  output o;
  reg    o;

always @(a or b or c or d or s)
  begin
    if (s == 2'b00)
      o = a;
    else if (s == 2'b01)
      o = b;
    else if (s == 2'b10)
      o = c;
  end
endmodule
```

Decoders

A decoder is a multiplexer whose inputs are all constant with distinct one-hot (or one-cold) coded values. Please refer to "Multiplexers" in this chapter for more details. This section shows two examples of 1-of-8 decoders using One-Hot and One-Cold coded values.

Log File

The XST log file reports the type and size of recognized decoders during the Macro Recognition step.

```
  Synthesizing Unit <dec>.
      Related source file is decoders_1.vhd.
      Found 1-of-8 decoder for signal <res>.
      Summary:
          inferred   1 Decoder(s).
  Unit <dec> synthesized.
 =============================
  HDL Synthesis Report

  Macro Statistics
  # Decoders                        : 1
    1-of-8 decoder                  : 1
 =============================
  ...
```

The following table shows pin definitions for a 1-of-8 decoder.

IO pins	Description
s[2:0]	Selector
res	Data Output

Related Constraints

A related constraint is DECODER_EXTRACT.

VHDL (One-Hot)

Following is the VHDL code for a 1-of-8 decoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity dec is
  port (
    sel: in std_logic_vector (2 downto 0);
    res: out std_logic_vector (7 downto 0));
end dec;
```

```
architecture archi of dec is
begin
  res <= "00000001" when sel = "000" else
         "00000010" when sel = "001" else
         "00000100" when sel = "010" else
         "00001000" when sel = "011" else
         "00010000" when sel = "100" else
         "00100000" when sel = "101" else
         "01000000" when sel = "110" else
         "10000000";
end archi;
```

Verilog (One-Hot)

Following is the Verilog code for a 1-of-8 decoder.

```
module mux (sel, res);
  input [2:0] sel;
  output [7:0] res;
  reg [7:0] res;

always @(sel or res)
  begin
    case (sel)
      3'b000 : res = 8'b00000001;
      3'b001 : res = 8'b00000010;
      3'b010 : res = 8'b00000100;
      3'b011 : res = 8'b00001000;
      3'b100 : res = 8'b00010000;
      3'b101 : res = 8'b00100000;
      3'b110 : res = 8'b01000000;
      default : res = 8'b10000000;
    endcase
  end
endmodule
```

VHDL (One-Cold)

Following is the VHDL code for a 1-of-8 decoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity dec is
  port (
    sel: in std_logic_vector (2 downto 0);
    res: out std_logic_vector (7 downto 0));
end dec;
```

```
                    architecture archi of dec is
                    begin
                      res <= "11111110" when sel = "000" else
                             "11111101" when sel = "001" else
                             "11111011" when sel = "010" else
                             "11110111" when sel = "011" else
                             "11101111" when sel = "100" else
                             "11011111" when sel = "101" else
                             "10111111" when sel = "110" else
                             "01111111";
                    end archi;
```

Verilog (One-Cold)

Following is the Verilog code for a 1-of-8 decoder.

```
module mux (sel, res);
  input [2:0] sel;
  output [7:0] res;
  reg [7:0] res;

always @(sel)
  begin
    case (sel)
      3'b000 : res = 8'b11111110;
      3'b001 : res = 8'b11111101;
      3'b010 : res = 8'b11111011;
      3'b011 : res = 8'b11110111;
      3'b100 : res = 8'b11101111;
      3'b101 : res = 8'b11011111;
      3'b110 : res = 8'b10111111;
      default : res = 8'b01111111;
    endcase
  end
endmodule
```

Decoders with Unselected Outputs

In the current version, XST does not infer decoders if one or several of the decoder outputs are not selected, except when the unused selector values are consecutive and at the end of the code space. Following is an example:

IO pins	Description
s[2:0]	Selector
res	Data Output

VHDL Code (No Decoder Inference)

For the following VHDL code, XST does not infer a decoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity dec is
  port (
    sel: in std_logic_vector (2 downto 0);
    res: out std_logic_vector (7 downto 0));
end dec;
architecture archi of dec is
begin
  res <=
    "00000001" when sel = "000" else
    "XXXXXXXX" when sel = "001" else -- unused decoder output
    "00000100" when sel = "010" else
    "00001000" when sel = "011" else
    "00010000" when sel = "100" else
    "00100000" when sel = "101" else
    "01000000" when sel = "110" else
    "10000000";
end archi;
```

Verilog Code (No Decoder Inference)

For the following Verilog code, XST does not infer a decoder.

```
module mux (sel, res);
  input [2:0] sel;
  output [7:0] res;
  reg [7:0] res;

always @(sel)
  begin
    case (sel)
      3'b000 : res = 8'b00000001;
      3'b001 : res = 8'bxxxxxxxx; // unused decoder output
      3'b010 : res = 8'b00000100;
      3'b011 : res = 8'b00001000;
      3'b100 : res = 8'b00010000;
      3'b101 : res = 8'b00100000;
      3'b110 : res = 8'b01000000;
      default : res = 8'b10000000;
    endcase
  end
endmodule
```

VHDL Code (Decoder Inference)

The following VHDL code leads to the inference of a 1-of-8 decoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity dec is
  port (
    sel: in std_logic_vector (2 downto 0);
    res: out std_logic_vector (7 downto 0));
end dec;
architecture archi of dec is
begin
  res <= "00000001" when sel = "000" else
         "00000010" when sel = "001" else
         "00000100" when sel = "010" else
         "00001000" when sel = "011" else
         "00010000" when sel = "100" else
         "00100000" when sel = "101" else
-- 110 and 111 selector values are unused
         "XXXXXXXX";
end archi;
```

Verilog Code (Decoder Inference)

The following Verilog code leads to the inference of a 1-of-8 decoder.

```
module mux (sel, res);
  input [2:0] sel;
  output [7:0] res;
  reg [7:0] res;

always @(sel or res)
  begin
    case (sel)
      3'b000 : res = 8'b00000001;
      3'b001 : res = 8'b00000010;
      3'b010 : res = 8'b00000100;
      3'b011 : res = 8'b00001000;
      3'b100 : res = 8'b00010000;
      3'b101 : res = 8'b00100000;
// 110 and 111 selector values are unused
      default : res = 8'bxxxxxxxx;
    endcase
  end
endmodule
```

Priority Encoders

XST can recognize a priority encoder, but in most cases XST does not infer it. To force priority encoder inference, use the PRIORITY_EXTRACT constraint with the value *force*. Xilinx® strongly suggests that you use this constraint on a signal-by-signal basis; otherwise, the constraint may guide you towards sub-optimal results.

Log File

The XST log file reports the type and size of recognized priority encoders during the Macro Recognition step.

```
  ...
  Synthesizing Unit <priority>.
      Related source file is priority_encoders_1.vhd.
      Found 3-bit 1-of-9 priority encoder for signal <code>.
      Summary:
          inferred   3 Priority encoder(s).
   Unit <priority> synthesized.


   ==============================
   HDL Synthesis Report

   Macro Statistics
   # Priority Encoders              : 1
     3-bit 1-of-9 priority encoder  : 1
   ==============================
   ...
```

3-Bit 1-of-9 Priority Encoder

Note: For this example XST may infer a priority encoder. You must use the PRIORITY_EXTRACT constraint with a value `force` to force its inference.

Related Constraint

A related constraint is PRIORITY_EXTRACT.

VHDL Code

Following is the VHDL code for a 3-bit 1-of-9 Priority Encoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity priority is
port (
    sel : in std_logic_vector (7 downto 0);
    code :out std_logic_vector (2 downto 0));
end priority;
```

```
                    architecture archi of priority is
                    begin
                      code <= "000" when sel(0) = '1' else
                              "001" when sel(1) = '1' else
                              "010" when sel(2) = '1' else
                              "011" when sel(3) = '1' else
                              "100" when sel(4) = '1' else
                              "101" when sel(5) = '1' else
                              "110" when sel(6) = '1' else
                              "111" when sel(7) = '1' else
                              "---";
                    end archi;
```

Verilog Code

Following is the Verilog code for a 3-bit 1-of-9 Priority Encoder.

```
module priority (sel, code);
  input [7:0] sel;
  output [2:0] code;
  reg [2:0] code;

always @(sel)
  begin
    if (sel[0]) code = 3'b000;
    else if (sel[1]) code = 3'b001;
    else if (sel[2]) code = 3'b010;
    else if (sel[3]) code = 3'b011;
    else if (sel[4]) code = 3'b100;
    else if (sel[5]) code = 3'b101;
    else if (sel[6]) code = 3'b110;
    else if (sel[7]) code = 3'b111;
    else             code = 3'bxxx;
  end
endmodule
```

Logical Shifters

Xilinx® defines a logical shifter as a combinatorial circuit with 2 inputs and 1 output:

- The first input is a data input that is shifted.
- The second input is a selector whose binary value defines the shift distance.
- The output is the result of the shift operation.

Note: *All* of these I/Os are mandatory; otherwise, XST does *not* infer a logical shifter.

Moreover, you must adhere to the following conditions when writing your HDL code:

- Use only logical, arithmetic and rotate shift operations. Shift operations that fill vacated positions with values from another signal are not recognized.
- For VHDL, you can only use predefined shift (SLL, SRL, ROL, etc.) or concatenation operations. Please refer to the IEEE VHDL language reference manual for more information on predefined shift operations.
- Use only one type of shift operation.
- The *n* value in the shift operation must be incremented or decremented only by 1 for each consequent binary value of the selector.

- The *n* value can be only positive.
- All values of the selector must be presented.

Log File

The XST log file reports the type and size of a recognized logical shifter during the Macro Recognition step.

```
...
Synthesizing Unit <lshift>.
    Related source file is Logical_Shifters_1.vhd.
    Found 8-bit shifter logical left for signal <so>.
    Summary:
        inferred   1 Combinational logic shifter(s).
 Unit <lshift> synthesized.
...
==============================
 HDL Synthesis Report

 Macro Statistics
 # Logic shifters                 : 1
   8-bit shifter logical left     : 1
==============================
...
```

Related Constraints

A related constraint is SHIFT_EXTRACT.

Example 1

The following table shows pin descriptions for a logical shifter.

IO pins	Description
D[7:0]	Data Input
SEL	Shift Distance Selector
SO[7:0]	Data Output

VHDL Code

Following is the VHDL code for a logical shifter.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lshift is
  port(
    DI  : in unsigned(7 downto 0);
    SEL : in unsigned(1 downto 0);
    SO  : out unsigned(7 downto 0));
end lshift;
architecture archi of lshift is
begin
  with SEL select
    SO <= DI when "00",
      DI sll 1 when "01",
      DI sll 2 when "10",
      DI sll 3 when others;
end archi;
```

Verilog Code

Following is the Verilog code for a logical shifter.

```
module lshift (DI, SEL, SO);
  input  [7:0] DI;
  input  [1:0] SEL;
  output [7:0] SO;
  reg    [7:0] SO;

always @(DI or SEL)
  begin
    case (SEL)
      2'b00 : SO = DI;
      2'b01 : SO = DI << 1;
      2'b10 : SO = DI << 2;
      default : SO = DI << 3;
    endcase
  end
endmodule
```

Example 2

XST does *not* infer a logical shifter for this example, as not all of the selector values are presented.

IO pins	Description
D[7:0]	Data Input
SEL	Shift Distance Selector
SO[7:0]	Data Output

VHDL Code

Following is the VHDL code.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lshift is
  port(
    DI  : in unsigned(7 downto 0);
    SEL : in unsigned(1 downto 0);
    SO  : out unsigned(7 downto 0));
end lshift;
architecture archi of lshift is
begin
  with SEL select
    SO <= DI when "00",
      DI sll 1 when "01",
      DI sll 2 when others;
end archi;
```

Verilog Code

Following is the Verilog code.

```
module lshift (DI, SEL, SO);
  input  [7:0] DI;
  input  [1:0] SEL;
  output [7:0] SO;
  reg    [7:0] SO;

always @(DI or SEL)
  begin
    case (SEL)
      2'b00 : SO = DI;
      2'b01 : SO = DI << 1;
      default : SO = DI << 2;
    endcase
  end
endmodule
```

Example 3

XST does *not* infer a logical shifter for this example, as the value is not incremented by 1 for each consequent binary value of the selector.

IO pins	Description
D[7:0]	Data Input
SEL	shift distance selector
SO[7:0]	Data Output

VHDL Code

Following is the VHDL code.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lshift is
  port(
    DI  : in unsigned(7 downto 0);
    SEL : in unsigned(1 downto 0);
    SO  : out unsigned(7 downto 0));
end lshift;
architecture archi of lshift is
begin
  with SEL select
    SO <= DI when "00",
      DI sll 1 when "01",
      DI sll 3 when "10",
      DI sll 2 when others;
end archi;
```

Verilog Code

Following is the Verilog code.

```verilog
module lshift (DI, SEL, SO);
  input  [7:0] DI;
  input  [1:0] SEL;
  output [7:0] SO;
  reg    [7:0] SO;

always @(DI or SEL)
  begin
    case (SEL)
      2'b00 : SO = DI;
      2'b01 : SO = DI << 1;
      2'b10 : SO = DI << 3;
      default : SO = DI << 2;
    endcase
  end
endmodule
```

Arithmetic Operations

XST supports the following arithmetic operations:

- Adders with:
  - ◆ Carry In
  - ◆ Carry Out
  - ◆ Carry In/Out
- Subtractors
- Adders/Subtractors
- Comparators (=, /=,<, <=, >, >=)
- Multipliers
- Dividers

Adders, subtractors, comparators and multipliers are supported for signed and unsigned operations.

Please refer to "Signed/Unsigned Support" in this chapter for more information on the signed/unsigned operations support in VHDL.

Moreover, XST performs resource sharing for adders, subtractors, adders/subtractors and multipliers.

Adders, Subtractors, Adders/Subtractors

This section provides HDL examples of adders and subtractors.

Log File

The XST log file reports the type and size of recognized adder, subtractor and adder/subtractor during the Macro Recognition step.

```
   ...
 Synthesizing Unit <adder>.
     Related source file is arithmetic_operations_1.vhd.
     Found 8-bit adder for signal <sum>.
     Summary:
         inferred   1 Adder/Subtracter(s).
 Unit <adder> synthesized.


 ==============================
 HDL Synthesis Report

 Macro Statistics
 # Adders/Subtractors              : 1
   8-bit adder                     : 1
 ==============================
```

Related Constraints

Related constraints are USE_DSP48 and KEEP, which are available for Virtex-4™ devices.

Unsigned 8-bit Adder

This subsection contains a VHDL and Verilog description of an unsigned 8-bit adder.

The following table shows pin descriptions for an unsigned 8-bit adder.

IO pins	Description
A[7:0], B[7:0]	Add Operands
SUM[7:0]	Add Result

VHDL Code

Following is the VHDL code for an unsigned 8-bit adder.

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

entity adder is
  port(
    A, B : in std_logic_vector(7 downto 0);
    SUM  : out std_logic_vector(7 downto 0));
end adder;
architecture archi of adder is
begin
  SUM <= A + B;
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit adder.

```
module adder(A, B, SUM);
  input  [7:0] A;
  input  [7:0] B;
  output [7:0] SUM;

  assign SUM = A + B;
endmodule
```

Unsigned 8-bit Adder with Carry In

This section contains VHDL and Verilog descriptions of an unsigned 8-bit adder with carry in.

The following table shows pin descriptions for an unsigned 8-bit adder with carry in.

IO pins	Description
A[7:0], B[7:0]	Add Operands
CI	Carry In
SUM[7:0]	Add Result

VHDL Code

Following is the VHDL code for an unsigned 8-bit adder with carry in.

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

entity adder is
  port(
    A, B : in std_logic_vector(7 downto 0);
    CI   : in std_logic;
    SUM  : out std_logic_vector(7 downto 0));
end adder;
architecture archi of adder is
begin
  SUM <= A + B + CI;
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit adder with carry in.

```
module adder(A, B, CI, SUM);
  input  [7:0] A;
  input  [7:0] B;
  input  CI;
  output [7:0] SUM;

  assign SUM = A + B + CI;
endmodule
```

Unsigned 8-bit Adder with Carry Out

This section contains VHDL and Verilog descriptions of an unsigned 8-bit adder with carry out.

If you use VHDL, then before writing a "+" operation with carry out, please examine the arithmetic package you are going to use. For example, "std_logic_unsigned" does not allow you to write "+" in the following form to obtain Carry Out:

```
Res(9-bit) = A(8-bit) + B(8-bit)
```

The reason is that the size of the result for "+" in this package is equal to the size of the longest argument, that is, 8 bits.

- One solution, for the example, is to adjust the size of operands A and B to 9-bits using concatenation.

```
Res <= ("0" & A) + ("0" & B);
```

    In this case, XST recognizes that this 9-bit adder can be implemented as an 8-bit adder with carry out.

- Another solution is to convert A and B to integers and then convert the result back to the std_logic vector, specifying the size of the vector equal to 9.

The following table shows pin descriptions for an unsigned 8-bit adder with carry out.

IO pins	Description
A[7:0], B[7:0]	Add Operands
SUM[7:0]	Add Result
CO	Carry Out

VHDL Code

Following is the VHDL code for an unsigned 8-bit adder with carry out.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
  port(
    A, B : in std_logic_vector(7 downto 0);
    SUM  : out std_logic_vector(7 downto 0);
    CO   : out std_logic);
end adder;

architecture archi of adder is
  signal tmp: std_logic_vector(8 downto 0);
  begin
  tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)),9);
  SUM <= tmp(7 downto 0);
  CO  <= tmp(8);
end archi;
```

In the preceding example, two arithmetic packages are used:

- std_logic_arith. This package contains the integer to std_logic conversion function, that is, conv_std_logic_vector.

- std_logic_unsigned. This package contains the unsigned "+" operation.

Verilog Code

Following is the Verilog code for an unsigned 8-bit adder with carry out.

```
module adder(A, B, SUM, CO);
  input [7:0] A;
  input [7:0] B;
  output [7:0] SUM;
  output CO;
  wire [8:0] tmp;

  assign tmp = A + B;
  assign SUM = tmp [7:0];
  assign CO = tmp [8];
endmodule
```

Unsigned 8-bit Adder with Carry In and Carry Out

The following table shows pin descriptions for an unsigned 8-bit adder with carry in and carry out.

IO pins	Description
A[7:0], B[7:0]	Add Operands
CI	Carry In
SUM[7:0]	Add Result
CO	Carry Out

VHDL Code

Following is the VHDL code for an unsigned 8-bit adder with carry in and carry out.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity adder is
  port(
    A, B : in std_logic_vector(7 downto 0);
    CI : in std_logic;
    SUM : out std_logic_vector(7 downto 0);
    CO : out std_logic);
end adder;
architecture archi of adder is
  signal tmp: std_logic_vector(8 downto 0);
  begin
    tmp <= conv_std_logic_vector((conv_integer(A) + conv_integer(B)
           + conv_integer(CI)),9);
    SUM <= tmp(7 downto 0);
    CO <= tmp(8);
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit adder with carry in and carry out.

```
module adder(A, B, CI, SUM, CO);
  input  CI;
  input  [7:0] A;
  input  [7:0] B;
  output [7:0] SUM;
  output CO;
  wire [8:0] tmp;
  assign tmp = A + B + CI;
  assign SUM = tmp [7:0];
  assign CO  = tmp [8];
endmodule
```

Simple Signed 8-bit Adder

The following table shows pin descriptions for a simple signed 8-bit adder.

IO pins	Description
A[7:0], B[7:0]	Add Operands
SUM[7:0]	Add Result

VHDL Code

Following is the VHDL code for a simple signed 8-bit adder.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity adder is
  port(
    A, B : in std_logic_vector(7 downto 0);
    SUM  : out std_logic_vector(7 downto 0));
end adder;
architecture archi of adder is
  begin
    SUM <= A + B;
end archi;
```

Verilog Code

Following is the Verilog code for a simple signed 8-bit adder.

```
module adder (A,B,SUM)
  input signed [7:0] A;
  input signed [7:0] B;
  output signed [7:0] SUM;
  wire signed [7:0] SUM;

  assign SUM = A + B;
endmodule
```

Unsigned 8-bit Subtractor

The following table shows pin descriptions for an unsigned 8-bit subtractor.

IO pins	Description
A[7:0], B[7:0]	Sub Operands
RES[7:0]	Sub Result

VHDL Code

Following is the VHDL code for an unsigned 8-bit subtractor.

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

entity subtr is
  port(
    A, B : in std_logic_vector(7 downto 0);
    RES  : out std_logic_vector(7 downto 0));
end subtr;
architecture archi of subtr is
  begin
    RES <= A - B;
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit subtractor.

```
module subtr(A, B, RES);
  input  [7:0] A;
  input  [7:0] B;
  output [7:0] RES;
  assign RES = A - B;
endmodule
```

Unsigned 8-bit Adder/Subtractor

The following table shows pin descriptions for an unsigned 8-bit adder/subtractor.

IO pins	Description
A[7:0], B[7:0]	Add/Sub Operands
OPER	Add/Sub Select
SUM[7:0]	Add/Sub Result

VHDL Code

Following is the VHDL code for an unsigned 8-bit adder/subtractor.

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

entity addsub is
  port(
    A, B : in std_logic_vector(7 downto 0);
    OPER : in std_logic;
    RES  : out std_logic_vector(7 downto 0));
end addsub;
architecture archi of addsub is
  begin
    RES <= A + B when OPER='0'
    else A - B;
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit adder/subtractor.

```verilog
module addsub(A, B, OPER, RES);
  input  OPER;
  input  [7:0] A;
  input  [7:0] B;
  output [7:0] RES;
  reg    [7:0] RES;

always @(A or B or OPER)
  begin
    if (OPER==1'b0)
      RES <= A + B;
    else
      RES <= A - B;
  end
endmodule
```

Considerations for Virtex-4™ Devices

The Virtex-4™ family allows adders/subtractors to be implemented on DSP48 resources. XST supports the registered version of these macros and can push up to 2 levels of input registers and 1 level of output registers into DSP48 blocks. If the Carry In or Add/Sub operation selectors are registered, XST pushes these registers into the DSP48 as well.

XST can implement an adder/subtractor in a DSP48 block if its implementation requires only a single DSP48 resource. If an accumulator macro does not fit in a single DSP48, XST implements the entire macro using slice logic.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint/command line option with a default value of *auto.* In this mode XST implements adders/subtractors using LUTs. You must set the value of this constraint to *yes* to force XST to push these macros into a DSP48.

Please note that when implementing adders/subtractors on DSP48 blocks, XST does not perform any automatic DSP48 resource control, and as a consequence, the number of generated DSP48 blocks in the NGC netlist may exceed the number of available DSP48 blocks in a target device.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if you want to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

Comparators (=, /=,<, <=, >, >=)

This section contains a VHDL and Verilog description for an unsigned 8-bit greater or equal comparator.

Log File

The XST log file reports the type and size of recognized comparators during the Macro Recognition step.

```
  ...
  Synthesizing Unit <compar>.
    Related source file is comparators_1.vhd.
    Found 8-bit comparator greatequal for signal <$n0000> created at
  line 10.
      Summary:
          inferred    1 Comparator(s).
  Unit <compar> synthesized.


  ==============================
  HDL Synthesis Report

  Macro Statistics
  # Comparators                      : 1
     8-bit comparator greatequal     : 1
  ==============================
  ...
```

Unsigned 8-bit Greater or Equal Comparator

The following table shows pin descriptions for a comparator.

IO pins	Description
A[7:0], B[7:0]	Comparison Operands
CMP	Comparison Result

VHDL Code

Following is the VHDL code for an unsigned 8-bit greater or equal comparator.

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

entity compar is
  port(
    A, B : in std_logic_vector(7 downto 0);
    CMP  : out std_logic);
end compar;
architecture archi of compar is
  begin
    CMP <= '1' when A >= B else '0';
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8-bit greater or equal comparator.

```
module compar(A, B, CMP);
  input  [7:0] A;
  input  [7:0] B;
  output CMP;

  assign CMP = (A >= B) ? 1'b1 : 1'b0;
endmodule
```

Multipliers

When implementing a multiplier, the size of the resulting signal is equal to the sum of 2 operand lengths. If you multiply A (8-bit signal) by B (4-bit signal), then the size of the result must be declared as a 12-bit signal.

Large Multipliers Using Block Multipliers

XST can generate large multipliers using an 18x18 bit block multiplier available in Virtex™-II/-II Pro/-II Pro X. For multipliers larger than this, XST can generate larger multipliers using multiple 18x18 bit block multipliers.

Registered Multiplier

For Virtex™-II/-II Pro/-II Pro X, in instances where a multiplier would have a registered output, XST infers a unique registered multiplier. This registered multiplier is 18x18 bits.

Under the following conditions, a registered multiplier is not used, and a multiplier + register is used instead.

- Output from the multiplier goes to any component other than the register.
- The MULT_STYLE constraint is set to *lut.*
- The multiplier is asynchronous.
- The multiplier has control signals other than synchronous reset or clock enable.
- The multiplier does not fit in a single 18x18 bit block multiplier.

The following pins are optional for a registered multiplier.

- clock enable port
- synchronous and asynchronous reset, and load ports

Considerations for Virtex-4™ Devices

The Virtex-4™ family allows multipliers to be implemented on DSP48 resources. XST supports the registered version of these macros and can push up to 2 levels of input registers and 2 levels of output registers into DSP48 blocks.

If a multiplier implementation requires multiple DSP48 resources, XST automatically decomposes it onto multiple DSP48 blocks. Depending on the operand size and to get the best performance, XST may implement most of a multiplier using DSP48 blocks and use slice logic for the rest of the macro. For example, it is not sufficient to use a single DSP48 to implement an 18x18 unsigned multiplier. In this case, XST implements most of the logic in one DSP48 and the rest in LUTs.

For Virtex-4™ devices, XST can infer pipelined multipliers, not only for the LUT implementation, but for the DSP48 implementation as well. Please see "Pipelined Multipliers" for details.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint/command line option, with a default value of *auto*. In this mode, XST implements multipliers taking into account the number of available DSP48 resources in the device.

Please note that XST can automatically recognize the MULT_STYLE constraint with values *lut* and *block* and then convert internally to USE_DSP48. Xilinx strongly recommends the use of the USE_DSP48 constraint for Virtex-4™ designs to define FPGA resources used for multiplier implementation. Xilinx recommends the use of the MULT_STYLE constraint to define the multiplier implementation method on the selected FPGA resources. This means that if USE_DSP48 is set to *auto* or *yes*, you may use `mult_style=pipe_block` to pipeline the DSP48 implementation if the multiplier implementation requires multiple DSP48 blocks. If USE_DSP48 is set to *no*, you can use `mult_style=pipe_lut|KCM|CSD` to define the multiplier implementation method on LUTs.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if you want to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

Multiplication with Constant

When one of the arguments is a constant, XST can create efficient dedicated implementations of a multiplier with constant. There are two methods: KCM and CSD.

Please note that dedicated implementations do not always provide the best results for multiplication with constants. Starting in release 6.2i, XST can automatically choose between KCM or standard multiplier implementation. The CSD method cannot be automatically chosen. Use the MULT_STYLE constraint to force CSD implementation.

Note: XST does not support KCM or CSD implementation for signed numbers.

Limitations:

If the either of the arguments is larger than 32 bits, XST does not use KCM or CSD implementation, even if it is specified with the MULT_STYLE constraint.

Log File

The XST log file reports the type and size of recognized multipliers during the Macro Recognition step.

```
  ...
 Synthesizing Unit <mult>.
     Related source file is multipliers_1.vhd.
     Found 8x4-bit multiplier for signal <res>.
     Summary:
         inferred   1 Multiplier(s).
 Unit <mult> synthesized.


 =============================
  HDL Synthesis Report

  Macro Statistics
  # Multipliers                    : 1
    8x4-bit multiplier             : 1
 =============================
  ...
```

Related Constraints

Related constraints are MULT_STYLE, USE_DSP48 and KEEP.

Unsigned 8x4-bit Multiplier

The following table shows pin descriptions for an unsigned 8x4-bit multiplier.

IO pins	Description
A[7:0], B[3:0]	MULT Operands
RES[7:0]	MULT Result

VHDL Code

Following is the VHDL code for an unsigned 8x4-bit multiplier.

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

entity mult is
  port(
    A   : in std_logic_vector(7 downto 0);
    B   : in std_logic_vector(3 downto 0);
    RES : out std_logic_vector(11 downto 0));
end mult;
architecture archi of mult is
  begin
    RES <= A * B;
end archi;
```

Verilog Code

Following is the Verilog code for an unsigned 8x4-bit multiplier.

```verilog
module compar(A, B, RES);
   input  [7:0] A;
   input  [3:0] B;
   output [11:0] RES;

   assign RES = A * B;
endmodule
```

Pipelined Multipliers

To increase the speed of designs with large multipliers, XST is capable of inferring pipelined multipliers. By interspersing registers between the stages of large multipliers, pipelining can significantly increase the overall frequency of your design. The effect of pipelining is similar to flip-flop retiming which is described in "Flip-Flop Retiming" in Chapter 3.

To insert pipeline stages, describe the necessary registers in your HDL code and place them after any multipliers, then set the MULT_STYLE constraint to *pipe_lut*. If the target family is Virtex-4™ and implementation of a multiplier requires multiple DSP48 blocks, XST can pipeline this implementation as well. You must set the MULT_STYLE constraint for this instance to *pipe_block*.

When XST detects valid registers for pipelining and MULT_STYLE is set to *pipe_lut* or *pipe_block*, XST uses the maximum number of available registers to reach the maximum multiplier speed. XST automatically calculates the maximum number of registers for each multiplier to get the best frequency.

If you have not specified sufficient register stages, and MULT_STYLE is coded directly on a signal, XST guides you via the HDL Advisor to specify the optimum number of register stages. XST does this during the Advanced HDL Synthesis step. If the number of registers placed after the multiplier exceeds the maximum required, and shift register extraction is activated, then XST implements the unused stages as shift registers.

Limitations:

- XST cannot pipeline hardware Multipliers (implementation using MULT18X18S resource).

- XST cannot pipeline multipliers if registers contain asynch set/reset or synch reset signals. XST *can* pipeline if registers contain synch reset signals.

Log File

.

```
=====================================================================
*                          HDL Synthesis                          *
=====================================================================

Synthesizing Unit <my_mult>.
    Related source file is pipe_mult_1.vhd.
    Found 36-bit register for signal <MULT>.
    Found 18-bit register for signal <a_in>.
    Found 18-bit register for signal <b_in>.
    Found 18x18-bit multiplier for signal <mult_res>.
    Found 36-bit register for signal <pipe_1>.
    Summary:
  inferred 108 D-type flip-flop(s).
  inferred   1 Multiplier(s).
Unit <my_mult> synthesized.
...
=====================================================================
*                       Advanced HDL Synthesis                    *
=====================================================================

Found pipelined multiplier on the signal <mult_res> with 1 pipeline
level(s).
INFO:Xst - HDL ADVISOR - You can improve the performance of this
multiplier by adding 3 register level(s).
```

VHDL Code

Use the following templates to implement pipelined multipliers in VHDL.

The following VHDL template shows the multiplication operation placed outside the process block and the pipeline stages represented as single registers.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult is
  generic(
    A_port_size : integer := 18;
    B_port_size : integer := 18);
  port(
    clk  : in std_logic;
    A    : in unsigned (A_port_size-1 downto 0);
    B    : in unsigned (B_port_size-1 downto 0);
    MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));
end mult;
```

```
architecture beh of mult is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res : unsigned ( (A_port_size+B_port_size-1) downto 0);
  signal pipe_1,
         pipe_2,
         pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);

begin
  mult_res <= a_in * b_in;
  process (clk)
    begin
      if (clk'event and clk='1') then
        a_in <= A; b_in <= B;
        pipe_1 <= mult_res;
        pipe_2 <= pipe_1;
        pipe_3 <= pipe_2;
        MULT   <= pipe_3;
      end if;
  end process;
end beh;
```

The following VHDL template shows the multiplication operation placed inside the
process block and the pipeline stages represented as single registers.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult is
  generic(
    A_port_size: integer := 18;
    B_port_size: integer := 18);
  port(
    clk  : in std_logic;
    A    : in unsigned (A_port_size-1 downto 0);
    B    : in unsigned (B_port_size-1 downto 0);
    MULT : out unsigned ((A_port_size+B_port_size-1) downto 0));
end mult;

architecture beh of mult is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res : unsigned ((A_port_size+B_port_size-1) downto 0);
  signal pipe_2,
         pipe_3 : unsigned ((A_port_size+B_port_size-1) downto 0);

begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      a_in <= A; b_in <= B;
      mult_res <= a_in * b_in;
      pipe_2   <= mult_res;
      pipe_3   <= pipe_2;
      MULT     <= pipe_3;
    end if;
  end process;
end beh;
```

The following VHDL template shows the multiplication operation placed outside the process block and the pipeline stages represented as shift registers.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mult is
  generic(
    A_port_size: integer := 18;
    B_port_size: integer := 18);

port(
    clk  : in std_logic;
    A    : in unsigned (A_port_size-1 downto 0);
    B    : in unsigned (B_port_size-1 downto 0);
    MULT : out unsigned ( (A_port_size+B_port_size-1) downto 0));
end mult;

architecture beh of mult is
  signal a_in, b_in : unsigned (A_port_size-1 downto 0);
  signal mult_res   : unsigned ((A_port_size+B_port_size-1) downto 0);

  type pipe_reg_type is array (2 downto 0) of unsigned
      ((A_port_size+B_port_size-1) downto 0);
  signal pipe_regs : pipe_reg_type;

begin
  mult_res <= a_in * b_in;
  process (clk)
    begin
      if (clk'event and clk='1') then
        a_in <= A; b_in <= B;
        pipe_regs <= mult_res & pipe_regs(2 downto 1);
        MULT <= pipe_regs(0);
      end if;
  end process;
end beh;
```

Verilog Code

Use the following templates to implement pipelined multipliers in Verilog.

The following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```verilog
module mult(clk, A, B, MULT);
  input clk;
  input [17:0] A;
  input [17:0] B;
  output [35:0] MULT;

  reg [35:0] MULT;
  reg [17:0] a_in, b_in;
  wire [35:0] mult_res;
  reg [35:0] pipe_1, pipe_2, pipe_3;

  assign mult_res = a_in * b_in;

  always @(posedge clk)
```

```
      begin
        a_in <= A; b_in <= B;
        pipe_1 <= mult_res;
        pipe_2 <= pipe_1;
        pipe_3 <= pipe_2;
        MULT   <= pipe_3;
      end
    endmodule
```

The following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, A, B, MULT);
  input clk;
  input [17:0] A;
  input [17:0] B;
  output [35:0] MULT;

  reg [35:0] MULT;
  reg [17:0] a_in, b_in;
  reg [35:0] mult_res;
  reg [35:0] pipe_2, pipe_3;

  always @(posedge clk)
    begin
      a_in <= A; b_in <= B;
      mult_res <= a_in * b_in;
      pipe_2 <= mult_res;
      pipe_3 <= pipe_2;
      MULT <= pipe_3;
    end
endmodule
```

The following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as shift registers.

```
module mult3(clk, A, B, MULT);
  input clk;
  input [17:0] A;
  input [17:0] B;
  output [35:0] MULT;

  reg [35:0] MULT;

  reg [17:0] a_in, b_in;
  wire [35:0] mult_res;
  reg [35:0] pipe_regs [3:0];

  assign mult_res = a_in * b_in;

  always @(posedge clk)
    begin
      a_in <= A; b_in <= B;
      {pipe_regs[3],pipe_regs[2],pipe_regs[1],pipe_regs[0]} <=
      {MULT, pipe_regs[3],pipe_regs[2],pipe_regs[1]};
    end
endmodule
```

Multiply Adder/Subtractor

The Multiply Adder/Subtractor macro is a complex macro consisting of several basic macros such as multipliers, adder/subtractors and registers. The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources available in Virtex-4™ devices.

Log File

In the Log file, XST reports the details of inferred multipliers, adder/subtractors and registers at the HDL Synthesis step. The composition of multiply adder/subtractor macros happens at the Advanced HDL Synthesis step. XST reports information about inferred MACs, because they are implemented within the MAC implementation mechanism.

```
=====================================================================
*                          HDL Synthesis                           *
=====================================================================

Synthesizing Unit <m_multaddsub1>.
    Related source file is "m_multaddsub1.vhd".
   Found 8x8-bit multiplier for signal <$n0000> created at line 20.
    Found 8-bit register for signal <A_reg1>.
    Found 8-bit register for signal <A_reg2>.
    Found 8-bit register for signal <B_reg1>.
    Found 8-bit register for signal <B_reg2>.
    Found 16-bit adder for signal <multaddsub>.
    Summary:
  inferred  32 D-type flip-flop(s).
  inferred   1 Adder/Subtractor(s).
  inferred   1 Multiplier(s).
Unit <m_multaddsub1> synthesized.
...
=====================================================================
*                     Advanced HDL Synthesis                       *
=====================================================================
...
INFO:Xst:1952 - Multiplier <Mmult__n0000> in block <m_mult1> and
adder/subtractor <Madd_mult> in block <m_mult1> are combined into a
MAC.
INFO:Xst:1953 - The following registers are also absorbed by the MAC:
<A_reg2> in block <m_mult1>, <A_reg1> in block <m_mult1>, <B_reg2> in
block <m_mult1>, <B_reg1> in block <m_mult1>, <R_mult1> in block
<m_mult1>.
```

Related Constraints

Related constraints are USE_DSP48 and KEEP which are available for Virtex-4™ devices.

Considerations for Virtex-4™ Devices

XST supports the registered version of this macro and can push up to 2 levels of input registers on multiplier inputs, 1 register level on the Adder/Subtractor input and 1 level of output register into the DSP48 block. If the Carry In or Add/Sub operation selectors are registered, XST pushes these registers into the DSP48. In addition, the multiplication operation could be registered as well.

XST can implement a multiply adder/subtractor in a DSP48 block if its implementation requires only a single DSP48 resource. If the macro exceeds the limits of a single DSP48, XST processes it as two separate Multiplier and Adder/Subtractor macros, making independent decisions on each macro. Please refer to the "Multipliers" and "Adders, Subtractors, Adders/Subtractors" sections for more information.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint/command line option, with default value of *auto*. Please note that when implementing multiply adders/subtractors on DSP48 blocks, XST does not perform any automatic DSP48 resource control, and as a consequence, the number of generated DSP48 blocks in the NGC netlist may exceed the number of available DSP48 blocks in a target device.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if you want to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

In the Log file, XST reports the details of inferred multipliers, adder/subtractors and registers at the HDL Synthesis step. XST reports about inferred MACs during the Advanced HDL Synthesis Step where the MAC implementation mechanism takes place.

Multiplier Adder with 2 Register Levels on Multiplier Inputs

VHDL Code

Use the following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity m_multaddsub1 is
  generic (p_width: integer:=8);
  port (
  clk     : in std_logic;
  A, B, C : in std_logic_vector(p_width-1 downto 0);
  RES     : out std_logic_vector(p_width*2-1 downto 0));
end m_multaddsub1;
```

```
architecture beh of m_multaddsub1 is
  signal A_reg1, A_reg2,
         B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
  signal multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin

  multaddsub <= A_reg2 * B_reg2 + C;

  process (clk)
  begin
    if (clk'event and clk='1') then
      A_reg1 <= A; A_reg2 <= A_reg1;
      B_reg1 <= B; B_reg2 <= B_reg1;
    end if;
  end process;

  RES <= multaddsub;
end beh;
```

Verilog Code

Use the following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in Verilog.

```
module mvl_multaddsub1(clk, A, B, C, RES);
  input clk;
  input [7:0] A;
  input [7:0] B;
  input [7:0] C;
  output [15:0] RES;

  reg  [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
  wire [15:0] multaddsub;

  always @(posedge clk)
    begin
      A_reg1 <= A; A_reg2 <= A_reg1;
      B_reg1 <= B; B_reg2 <= B_reg1;
    end

  assign multaddsub = A_reg2 * B_reg2 + C;
  assign RES = multaddsub;
endmodule
```

Multiplier Adder/Subtractor with 2 Register Levels on Multiplier Inputs

VHDL Code

Use the following templates to implement Multiplier Adder/Subtractor with 2 Register Levels on Multiplier Inputs in VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity m_multaddsub2 is
  generic (p_width: integer:=8);
  port (
    clk,add_sub: in std_logic;
    A, B, C: in std_logic_vector(p_width-1 downto 0);
    RES: out std_logic_vector(p_width*2-1 downto 0));
end m_multaddsub2;

architecture beh of m_multaddsub2 is
  signal A_reg1, A_reg2,
         B_reg1, B_reg2 : std_logic_vector(p_width-1 downto 0);
  signal mult, multaddsub : std_logic_vector(p_width*2-1 downto 0);
begin
  mult <= A_reg2 * B_reg2;
  multaddsub <= mult + C when add_sub = '1' else mult - C;

  process (clk)
  begin
    if (clk'event and clk='1') then
      A_reg1 <= A; A_reg2 <= A_reg1;
      B_reg1 <= B; B_reg2 <= B_reg1;
    end if;
  end process;

  RES <= multaddsub;
end beh;
```

Verilog Code

Use the following templates to implement Multiplier Adder/Subtractor with 2 Register Levels on Multiplier Inputs in Verilog.

```verilog
module mvl_multaddsub2(clk, add_sub, A, B, C, RES);
   input clk,add_sub;
   input [7:0] A;
   input [7:0] B;
   input [7:0] C;
   output [15:0] RES;

   reg  [7:0] A_reg1, A_reg2, B_reg1, B_reg2;
   wire [15:0] mult, multaddsub;

   always @(posedge clk)
     begin
       A_reg1 <= A; A_reg2 <= A_reg1;
       B_reg1 <= B; B_reg2 <= B_reg1;
     end

   assign mult = A_reg2 * B_reg2;
   assign multaddsub = add_sub ? mult + C : mult - C;
   assign RES = multaddsub;
endmodule
```

Multiply Accumulate (MAC)

The Multiply Accumulate macro is a complex macro which consists of several basic macros such as multipliers, accumulators and registers. The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources available on Virtex-4™ devices.

Log File

In the Log file, XST reports the details of inferred multipliers, accumulators and registers at the HDL Synthesis step. The composition of multiply accumulate macros happens at the Advanced HDL Synthesis step.

```
======================================================================
*                         HDL Synthesis                               *
======================================================================
...
Synthesizing Unit <m_mac1>.
    Related source file is "m_mac1.vhd".
    Found 16-bit up accumulator for signal <accum>.
    Found 8x8-bit multiplier for signal <mult>.
    Summary:
  inferred   1 Accumulator(s).
  inferred   1 Multiplier(s).
Unit <m_mac1> synthesized.
...
======================================================================
*                     Advanced HDL Synthesis                          *
======================================================================
...
MAC inference ...
INFO:Xst:1951 - Multiplier <Mmult_mult> in block <m_mac1> and
accumulator <accum> in block <m_mac1> are combined into a MAC.
```

Related Constraints

Related constraints are USE_DSP48 and KEEP which are available for Virtex-4™ devices.

Considerations for Virtex-4™ Devices

The Multiply Accumulate macro is a complex macro which consists of several basic macros as multipliers, accumulators and registers. The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources available in Virtex-4™ devices.

XST supports the registered version of this macro, and can push up to 2 levels of input registers into the DSP48 block. If Adder/Subtractor operation selectors are registered, XST pushes these registers into the DSP48. In addition, the multiplication operation could be registered as well.

XST can implement a multiply accumulate in a DSP48 block if its implementation requires only a single DSP48 resource. If the macro exceeds the limits of a single DSP48, XST processes it as a 2 separate Multiplier and Accumulate macros, making independent decisions on each macro. Please refer to the "Multipliers" and "Accumulators" sections for more information.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint/command line option, with default value of *auto.* Please note that when implementing multiply accumulate macros on DSP48 blocks, XST does not perform any automatic DSP48 resource control, and as consequence the number of generated DSP48 blocks in the NGC netlist may exceed the number of available DSP48 blocks in the target device.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if you want to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

In the Log file, XST reports the details of inferred multipliers, accumulators and registers at the HDL Synthesis step. The composition of multiply accumulate macros happens at Advanced HDL Synthesis step.

Multiplier Up Accumulate with Register After Multiplication

VHDL Code

Use the following templates to implement Multiplier Up Accumulate with Register After Multiplication in VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity m_mac1 is
  generic (p_width: integer:=8);
  port (
    clk, reset: in std_logic;
    A, B: in std_logic_vector(p_width-1 downto 0);
    RES: out std_logic_vector(p_width*2-1 downto 0));
end m_mac1;

architecture beh of m_mac1 is
  signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      if (reset = '1') then
        accum <= (others => '0');
        mult <= (others => '0');
      else
        accum <= accum + mult;
        mult <= A * B;
      end if;
    end if;
  end process;

  RES <= accum;
end beh;
```

Verilog Code

Use the following templates to implement Multiplier Up Accumulate with Register After Multiplication in Verilog.

```
module mvl_mac1(clk, reset, A, B, RES);
  input clk, reset;
  input [7:0] A;
  input [7:0] B;
  output [15:0] RES;
  reg [15:0] mult, accum;

  always @(posedge clk)
    begin
      if (reset)
        mult <= 16'b0000000000000000;
      else
        mult <= A * B;
    end

  always @(posedge clk)
    begin
      if (reset)
        accum <= 16'b0000000000000000;
      else
        accum <= accum + mult;
    end

  assign RES = accum;
endmodule
```

Multiplier Up/Down Accumulate with Register After Multiplication.

VHDL Code

Use the following templates to implement Multiplier Up/Down Accumulate with Register After Multiplication in VHDL.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity m_mac2 is
  generic (p_width: integer:=8);
  port (
    clk, reset, add_sub: in std_logic;
    A, B: in std_logic_vector(p_width-1 downto 0);
    RES: out std_logic_vector(p_width*2-1 downto 0));
end m_mac2;
```

```
architecture beh of m_mac2 is
  signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk='1') then
      if (reset = '1') then
        accum <= (others => '0');
        mult <= (others => '0');
      else
        if (add_sub = '1') then
          accum <= accum + mult;
        else
          accum <= accum - mult;
        end if;
        mult <= A * B;
      end if;
    end if;
  end process;

  RES <= accum;
end beh;
```

Verilog Code

Use the following templates to implement Multiplier Up/Down Accumulate with Register After Multiplication in Verilog.

```
module mvl_mac2(clk, reset, add_sub, A, B, RES);
  input clk, reset, add_sub;
  input [7:0] A;
  input [7:0] B;
  output [15:0] RES;
  reg [15:0] mult, accum;

  always @(posedge clk)
    begin
      if (reset)
        mult <= 16'b0000000000000000;
      else
        mult <= A * B;
    end

  always @(posedge clk)
    begin
      if (reset)
        accum <= 16'b0000000000000000;
      else
        if (add_sub)
          accum <= accum + mult;
        else
          accum <= accum - mult;
    end

  assign RES = accum;
endmodule
```

Dividers

Dividers are only supported when the divisor is a constant and is a power of 2. In that case, the operator is implemented as a shifter; otherwise XST issues an error message.

Log File

When you implement a divider with a constant with the power of 2, XST does not issue any message during the Macro Recognition step. In case your divider does not correspond to the case supported by XST, the following error message displays:

```
...
ERROR:Xst:719 - file1.vhd (Line 172).
Operator is not supported yet : 'DIVIDE'
...
```

Related Constraints

There are no related constraints available.

Division By Constant 2

This section contains VHDL and Verilog descriptions of a Division By Constant 2 divider.

The following table shows pin descriptions for a Division By Constant 2 divider.

IO pins	Description
DI[7:0]	Division Operands
DO[7:0]	Division Result

VHDL

Following is the VHDL code for a Division By Constant 2 divider.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity divider is
  port(
    DI : in unsigned(7 downto 0);
    DO : out unsigned(7 downto 0));
end divider;

architecture archi of divider is
  begin
    DO <= DI / 2;
end archi;
```

Verilog

Following is the Verilog code for a Division By Constant 2 divider.

```verilog
module divider(DI, DO);
   input [7:0] DI;
   output [7:0] DO;

   assign DO = DI / 2;
endmodule
```

Resource Sharing

The goal of resource sharing (also known as folding) is to minimize the number of operators and the subsequent logic in the synthesized design. This optimization is based on the principle that two similar arithmetic resources may be implemented as one single arithmetic operator if they are never used at the same time. XST performs both resource sharing and, if required, reduces the number of multiplexers that are created in the process.

XST supports resource sharing for adders, subtractors, adders/subtractors and multipliers.

If the optimization goal is SPEED, then the disabling of resource sharing may lead to better results. XST advises you to try to deactivate resource sharing at the Advance HDL Synthesis step in order to improve clock frequency.

Log File

The XST log file reports the type and size of recognized arithmetic blocks and multiplexers during the Macro Recognition step.

```
    ...
  Synthesizing Unit <addsub>.
      Related source file is resource_sharing_1.vhd.
      Found 8-bit addsub for signal <res>.
      Found 8 1-bit 2-to-1 multiplexers.
      Summary:
          inferred   1 Adder/Subtracter(s).
          inferred   8 Multiplexer(s).
  Unit <addsub> synthesized.

 ==============================
  HDL Synthesis Report

  Macro Statistics
  # Multiplexers                    : 1
    2-to-1 multiplexer              : 1
  # Adders/Subtractors              : 1
    8-bit addsub                    : 1
 ==============================
 ...
 =====================================================================
 *                       Advanced HDL Synthesis                     *
 =====================================================================

 INFO:Xst - HDL ADVISOR - Resource sharing has identified that some
 arithmetic operations in this design can share the same physical
 resources for reduced device utilization. For improved clock
 frequency you may try to disable resource sharing.
 ...
```

Related Constraint

The related constraint is RESOURCE_SHARING.

Example

For the following VHDL/Verilog example, XST gives the following solution.

X8984
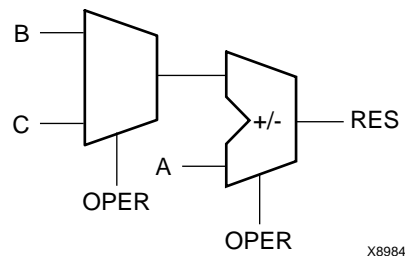
The following table shows pin descriptions for the example.

IO pins	Description
A[7:0], B[7:0], C[7:0]	Operands
OPER	Operation Selector
RES[7:0]	Data Output

VHDL Code

Following is the VHDL example for resource sharing.

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

entity addsub is
  port(
    A, B, C : in std_logic_vector(7 downto 0);
    OPER : in std_logic;
    RES : out std_logic_vector(7 downto 0));
end addsub;

architecture archi of addsub is
  begin
    RES <= A + B when OPER='0' else A - C;
end archi;
```

Verilog Code

Following is the Verilog code for resource sharing.

```
module addsub(A, B, C, OPER, RES);
  input OPER;
  input [7:0] A;
  input [7:0] B;
  input [7:0] C;
  output [7:0] RES;
  reg [7:0] RES;

  always @(A or B or C or OPER)
  begin
    if (OPER==1'b0)
      RES <= A + B;
    else
      RES <= A - C;
  end
endmodule
```

RAMs/ROMs

If you do not want to instantiate RAM primitives to keep your HDL code technology independent, XST offers an automatic RAM recognition capability. XST can infer distributed as well as block RAM. It covers the following characteristics, offered by these RAM types.

- Synchronous write
- Write enable
- RAM enable
- Asynchronous or synchronous read
- Reset of the data output latches
- Data output reset
- Single, dual or multiple-port read
- Single-port/Dual-port write
- Parity bits (Virtex-4™ devices only)

Note: XST does not support RAMs/ROMs with negative addresses.

The type of inferred RAM depends on its description.

- RAM descriptions with an asynchronous read generate a distributed RAM macro.
- RAM descriptions with a synchronous read generate a block RAM macro. In some cases, a block RAM macro can actually be implemented with distributed RAM. The decision on the actual RAM implementation is done by the macro generator.

Following is the list of VHDL/Verilog templates that are described below.

- Virtex-II™ RAM Read/Write modes
    - Read-First Mode
    - Write-First Mode
    - No-Change Mode
- Single-Port RAM with Asynchronous Read
- Single-Port RAM with "False" Synchronous Read
- Single-Port RAM with Synchronous Read (Read Through)
- Single-Port RAM with Enable
- Dual-Port RAM with Asynchronous Read
- Dual-Port RAM with False Synchronous Read
- Dual-Port RAM with Synchronous Read (Read Through)
- Dual-Port RAM with One Enable Controlling Both Ports
- Dual-Port RAM with Enable Controlling Each Port
- Dual-Port RAM with Different Clocks
- Dual-Port Block RAM with two write ports
- Multiple-Port RAM Descriptions
- Block RAM with Reset
- Initializing Block RAM
- ROMs Using Block RAM Resources

If a given template can be implemented using Block and Distributed RAM, XST implements BLOCK ones. You can use the RAM_STYLE constraint to control RAM implementation and select a desirable RAM type. Please refer to Chapter 5, "Design Constraints" for more details.

Please note that the following features specifically available with block RAM are *not* yet supported.

- Dual write port
- Parity bits (except Virtex-4™ devices)
- Different aspect ratios on each port

Please refer to Chapter 3, "FPGA Optimization" for more details on RAM implementation.

Note: Note that XST can implement State Machines (see "State Machine") and map general logic (see "Mapping Logic onto Block RAM" in Chapter 3) on block RAMs.

Log File

The XST log file reports the type and size of recognized RAM as well as complete information on its I/O ports during the Macro Recognition step.

```
...
Synthesizing Unit <raminfr>.
     Related source file is rams_1.vhd.
     Found 128-bit single-port distributed RAM for signal <ram>.
      -------------------------------------------------------
     | aspect ratio  | 32-word x 4-bit        |      |
     | clock         | connected to signal <clk>  | rise  |
     | write enable  | connected to signal <we>   | high  |
     | address       | connected to signal <a>    |      |
     | data in       | connected to signal <di>   |      |
     | data out      | connected to signal <do>   |      |
     | ram_style     | Auto                   |      |
      -------------------------------------------------------
 INFO:Xst - For optimized device usage and improved timings, you
     may take advantage of available block RAM resources by
     registering the read address.
     Summary:
         inferred   1 RAM(s).
 Unit <raminfr> synthesized.


 ====================================
 HDL Synthesis Report

 Macro Statistics
 # RAMs                              : 1
    128-bit single-port distributed RAM : 1
 ====================================
...
```

Related Constraints

Related constraints are RAM_EXTRACT, RAM_STYLE, ROM_EXTRACT and ROM_STYLE.

Beginning in release 7.1i, XST accepts LOC and RLOC constraints on inferred RAMs that can be implemented in a single block RAM primitive. The LOC and RLOC constraints are propagated to the NGC netlist.

Virtex-II™/Spartan-3™ RAM Read/Write Modes

Block RAM resources available in Virtex™-II/-II Pro/-II Pro X and Spartan-3™ offer different read/write synchronization modes. This section provides coding examples for all three modes that are available: write-first, read-first and no-change.

The following examples describe a simple single-port block RAM. You can deduce descriptions of dual-port block RAMs from these examples. Dual-port block RAMs can be configured with a different read/write mode on each port. Inference supports this capability.

The following table summarizes support for read/write modes according to the targeted family and how XST handles it.

Family	Inferred Modes	Behavior
Spartan-3™ Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X	write-first, read-first, no-change	• Macro inference and generation • Attach adequate WRITE_MODE, WRITE_MODE_A, WRITE_MODE_B constraints to generated block RAMs in NCF
Virtex™, Virtex-E,™ Spartan-II™ Spartan-IIE™	write-first	• Macro inference and generation • No constraint to attach on generated block RAMs
CPLD	none	RAM inference completely disabled

Read-First Mode

The following templates show a single-port RAM in read-first mode.

VHDL Code

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

entity raminfr is
  port (
    clk : in std_logic;
    we : in std_logic;
    en : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di : in std_logic_vector(3 downto 0);
    do : out std_logic_vector(3 downto 0));
end raminfr;
```

```vhdl
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if clk'event and clk = '1' then
        if en = '1' then
          if we = '1' then
            RAM(conv_integer(addr)) <= di;
          end if;
          do <= RAM(conv_integer(addr));
        end if;
      end if;
    end process;
end syn;
```

Verilog Code

```verilog
module raminfr (clk, en, we, addr, di, do);
  input clk;
  input we;
  input en;
  input [4:0] addr;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] RAM [31:0];
  reg [3:0] do;

  always @(posedge clk)
    begin
      if (en)
        begin
          if (we)
            RAM[addr] <= di;
          do <= RAM[addr];
        end
    end
endmodule
```

Write-First Mode

The following templates show a single-port RAM in write-first mode.

VHDL Code

The following template shows the recommended configuration coded in VHDL.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity raminfr is
  port (
    clk : in std_logic;
    we : in std_logic;
    en : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di : in std_logic_vector(3 downto 0);
    do : out std_logic_vector(3 downto 0));
end raminfr;
```

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if clk'event and clk = '1' then
        if en = '1' then
          if we = '1' then
            RAM(conv_integer(addr)) <= di;
            do <= di;
          else
            do <= RAM(conv_integer(addr));
          end if;
        end if;
      end if;
  end process;
end syn;
```

The following templates show an alternate configuration of a single-port RAM in write-first mode with a registered read address coded in VHDL.

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

entity raminfr is
  port (
    clk : in std_logic;
    we : in std_logic;
    en : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di : in std_logic_vector(3 downto 0);
    do : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_addr : std_logic_vector(4 downto 0);
begin
  process (clk)
    begin
      if clk'event and clk = '1' then
        if en = '1' then
          if we = '1' then
            mem(conv_integer(addr)) <= di;
          end if;
          read_addr <= addr;
        end if;
      end if;
  end process;
  do <= ram(conv_integer(read_addr));
end syn;
```

Verilog Code

The following template shows the recommended configuration coded in Verilog.

```verilog
module raminfr (clk, we, en, addr, di, do);
  input clk;
  input we;
  input en;
  input [4:0] addr;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] RAM [31:0];
  reg [4:0] read_addr;

always @(posedge clk)
  begin
    if (en)
      begin
        if (we)
          RAM[addr] <= di;
        read_addr <= addr;
      end
  end
  assign do = RAM[read_addr];
endmodule
```

No-Change Mode

The following templates show a single-port RAM in no-change mode.

VHDL Code

The following template shows the recommended configuration coded in VHDL.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity raminfr is
  port (
    clk : in std_logic;
    we : in std_logic;
    en : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di : in std_logic_vector(3 downto 0);
    do : out std_logic_vector(3 downto 0));
end raminfr;
```

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  begin
    process (clk)
      begin
        if clk'event and clk = '1' then
          if en = '1' then
            if we = '1' then
              RAM(conv_integer(addr)) <= di;
            else
              do <= RAM(conv_integer(addr));
            end if;
          end if;
        end if;
    end process;
end syn;
```

The following templates show an alternate configuration of a single-port RAM in no-change mode with a registered read address coded in VHDL.

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

entity raminfr is
  port (
    clk  : in std_logic;
    we   : in std_logic;
    en   : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    do   : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_addr : std_logic_vector(4 downto 0);

  begin
  process (clk)
    begin
      if clk'event and clk = '1' then
        if en = '1' then
          if we = '1' then
            RAM(conv_integer(addr)) <= di;
          else
            read_addr <= addr;
          end if;
        end if;
      end if;
    end process;
  do <= RAM(read_addr);
end syn;
```

Verilog Code

The following template shows the recommended configuration coded in Verilog.

```verilog
module raminfr (clk, we, en, addr, di, do);
   input clk;
   input we;
   input en;
   input [4:0] addr;
   input [3:0] di;
   output [3:0] do;
   reg [3:0] RAM [31:0];
   reg [3:0] do;

   always @(posedge clk)
   begin
     if (en)
       begin
         if (we)
           RAM[addr] <= di;
         else
           do <= RAM[addr];
       end
   end
endmodule
```

Single-Port RAM with Asynchronous Read

The following descriptions are directly mappable onto *distributed RAM only.*

X8976

The following table shows pin descriptions for a single-port RAM with asynchronous read.

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

VHDL Code

Following is the VHDL code for a single-port RAM with asynchronous read.

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

entity raminfr is
  port (
    clk : in std_logic;
    we  : in std_logic;
    a   : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (we = '1') then
          RAM(conv_integer(a)) <= di;
        end if;
      end if;
  end process;
  do <= RAM(conv_integer(a));
end syn;
```

Verilog Code

Following is the Verilog code for a single-port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);
  input clk;
  input we;
  input  [4:0] a;
  input  [3:0] di;
  output [3:0] do;
  reg    [3:0] ram [31:0];

  always @(posedge clk)
    begin
      if (we)
        ram[a] <= di;
    end
  assign do = ram[a];
endmodule
```

Single-Port RAM with "False" Synchronous Read

The following descriptions do not implement true synchronous read access as defined by the Virtex™ block RAM specification, where the read address is registered. They are *only mappable onto distributed RAM* with an additional buffer on the data output, as shown below.
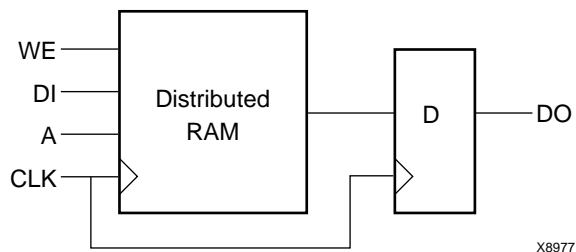
The following table shows pin descriptions for a single-port RAM with "false" synchronous read.

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

VHDL

Following is the VHDL code for a single-port RAM with "false" synchronous read.

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

entity raminfr is
  port (
    clk : in std_logic;
    we  : in std_logic;
    a   : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do  : out std_logic_vector(3 downto 0));
end raminfr;
```

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (we = '1') then
          RAM(conv_integer(a)) <= di;
        end if;
        do <= RAM(conv_integer(a));
      end if;
  end process;
end syn;
```

Verilog

Following is the Verilog code for a single-port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);
  input clk;
  input we;
  input [4:0] a;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] ram [31:0];
  reg [3:0] do;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    do <= ram[a];
  end
endmodule
```

The following descriptions, featuring an additional reset of the RAM output, are also *only mappable onto Distributed RAM* with an additional resetable buffer on the data output as shown in the following figure:
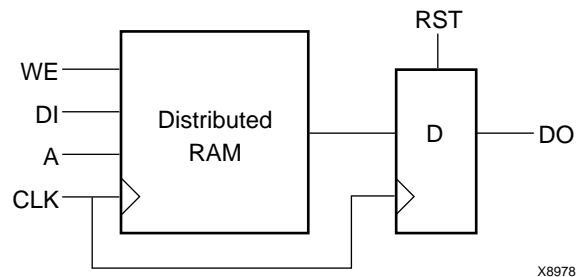
X8978

The following table shows pin descriptions for a single-port RAM with "false" synchronous read and reset on the output.

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
rst	Synchronous Output Reset (active High)
a	Read/Write Address
di	Data Input
do	Data Output

VHDL Code

Following is the VHDL code.

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

entity raminfr is
  port (
    clk : in std_logic;
    we  : in std_logic;
    rst : in std_logic;
    a   : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (we = '1') then
          RAM(conv_integer(a)) <= di;
        end if;
        if (rst = '1') then
          do <= (others => '0');
        else
          do <= RAM(conv_integer(a));
        end if;
      end if;
  end process;
end syn;
```

Verilog Code

Following is the Verilog code.

```verilog
module raminfr (clk, we, rst, a, di, do);
   input clk;
   input we;
   input rst;
   input [4:0] a;
   input [3:0] di;
   output [3:0] do;
   reg [3:0] ram [31:0];
   reg [3:0] do;

   always @(posedge clk) begin
     if (we)
       ram[a] <= di;
     if (rst)
       do <= 4'b0;
     else
       do <= ram[a];
   end
endmodule
```

Single-Port RAM with Synchronous Read (Read Through)

The following description implements a true synchronous read. A true synchronous read is the synchronization mechanism available in Virtex™ block RAMs, where the read address is registered on the RAM clock edge. Such descriptions are *directly mappable onto block RAM*, as shown below. (The same descriptions can also be mapped onto *Distributed RAM*).

X8979

The following table shows pin descriptions for a single-port RAM with synchronous read (read through).

IO pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

VHDL Code

Following is the VHDL code for a single-port RAM with synchronous read (read through).

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity raminfr is
  port (
    clk : in std_logic;
    we  : in std_logic;
    a   : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_a : std_logic_vector(4 downto 0);
begin
  process (clk)
    begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      read_a <= a;
    end if;
  end process;
  do <= RAM(conv_integer(read_a));
end syn;
```

Verilog Code

Following is the Verilog code for a single-port RAM with synchronous read (read through).

```verilog
module raminfr (clk, we, a, di, do);
  input clk;
  input we;
  input [4:0] a;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] ram [31:0];
  reg [4:0] read_a;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
  end
  assign do = ram[read_a];
endmodule
```
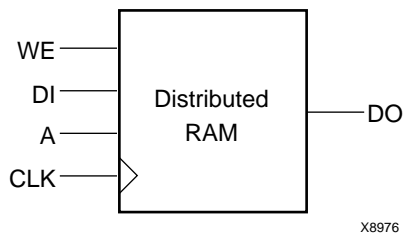
Single-Port RAM with Enable

The following description implements a single-port RAM with a global enable.

X9478
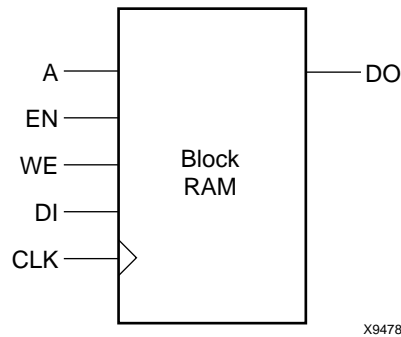
The following table shows pin descriptions for a single-port RAM with enable.

IO pins	Description
clk	Positive-Edge Clock
en	Global Enable
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

VHDL Code

Following is the VHDL code for a single-port block RAM with enable.

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

entity raminfr is
  port (
    clk : in std_logic;
    en  : in std_logic;
    we  : in std_logic;
    a   : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do  : out std_logic_vector(3 downto 0));
end raminfr;
```

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_a : std_logic_vector(4 downto 0);
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (en = '1') then
          if (we = '1') then
            RAM(conv_integer(a)) <= di;
          end if;
          read_a <= a;
        end if;
      end if;
  end process;
  do <= RAM(conv_integer(read_a));
end syn;
```

Verilog Code

Following is the Verilog code for a single-port block RAM with enable.

```
module raminfr (clk, en, we, a, di, do);
  input clk;
  input en;
  input we;
  input [4:0] a;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] ram [31:0];
  reg [4:0] read_a;

  always @(posedge clk) begin
    if (en)
      begin
        if (we)
          ram[a] <= di;
          read_a <= a;
        end
      end
  assign do = ram[read_a];
endmodule
```

Dual-Port RAM with Asynchronous Read

The following example shows where the two output ports are used. It is directly mappable onto *Distributed RAM only.*

X8980

The following table shows pin descriptions for a dual-port RAM with asynchronous read.

IO pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
a	Write Address/Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with asynchronous read.

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

entity raminfr is
  port (
    clk  : in std_logic;
    we   : in std_logic;
    a    : in std_logic_vector(4 downto 0);
    dpra : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    spo  : out std_logic_vector(3 downto 0);
    dpo  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (we = '1') then
          RAM(conv_integer(a)) <= di;
        end if;
      end if;
  end process;
  spo <= RAM(conv_integer(a));
  dpo <= RAM(conv_integer(dpra));
end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with asynchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
  input clk;
  input we;
  input [4:0] a;
  input [4:0] dpra;
  input [3:0] di;
  output [3:0] spo;
  output [3:0] dpo;
  reg [3:0] ram [31:0];

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
  end
  assign spo = ram[a];
  assign dpo = ram[dpra];
endmodule
```

Dual-Port RAM with False Synchronous Read

The following description is mapped onto Distributed RAM with additional registers on the data outputs. Please note that this template *does not* describe dual-port block RAM.

X8981

The following table shows pin descriptions for a dual-port RAM with false synchronous read.

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
a	Write Address/Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with false synchronous read.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity raminfr is
  port (
    clk  : in std_logic;
    we   : in std_logic;
    a    : in std_logic_vector(4 downto 0);
    dpra : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    spo  : out std_logic_vector(3 downto 0);
    dpo  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      spo <= RAM(conv_integer(a));
      dpo <= RAM(conv_integer(dpra));
    end if;
  end process;
end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with false synchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
  input clk;
  input we;
  input [4:0] a;
  input [4:0] dpra;
  input [3:0] di;
  output [3:0] spo;
  output [3:0] dpo;
  reg [3:0] ram [31:0];
  reg [3:0] spo;
  reg [3:0] dpo;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    spo = ram[a];
    dpo = ram[dpra];
  end
endmodule
```

Dual-Port RAM with Synchronous Read (Read Through)

The following descriptions are *directly mappable onto block RAM*, as shown in the following figure. (They may also be implemented with *Distributed RAM.*).
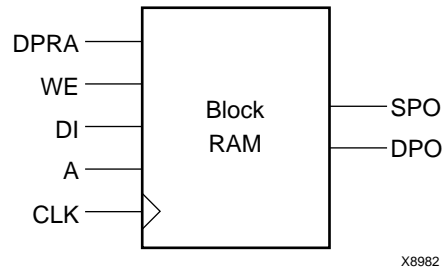
X8982

The following table shows pin descriptions for a dual-port RAM with synchronous read (read through).

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Write Address/Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with synchronous read (read through).

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

entity raminfr is
  port (
    clk  : in std_logic;
    we   : in std_logic;
    a    : in std_logic_vector(4 downto 0);
    dpra : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    spo  : out std_logic_vector(3 downto 0);
    dpo  : out std_logic_vector(3 downto 0));
end raminfr;
```

```
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_a : std_logic_vector(4 downto 0);
  signal read_dpra : std_logic_vector(4 downto 0);
begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (we = '1') then
          RAM(conv_integer(a)) <= di;
        end if;
        read_a <= a;
        read_dpra <= dpra;
      end if;
  end process;
  spo <= RAM(conv_integer(read_a));
  dpo <= RAM(conv_integer(read_dpra));
end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
  input clk;
  input we;
  input [4:0] a;
  input [4:0] dpra;
  input [3:0] di;
  output [3:0] spo;
  output [3:0] dpo;
  reg [3:0] ram [31:0];
  reg [4:0] read_a;
  reg [4:0] read_dpra;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
    read_dpra <= dpra;
  end
  assign spo = ram[read_a];
  assign dpo = ram[read_dpra];
endmodule
```

Using More than One Clock

The two RAM ports may be synchronized on distinct clocks, as shown in the following description. In this case, only a block RAM implementation is applicable.

The following table shows pin descriptions for a dual-port RAM with synchronous read (read through) and two clocks.

IO pins	Description
clk1	Positive-Edge Write/Primary Read Clock
clk2	Positive-Edge Dual Read Clock

IO pins	Description
we	Synchronous Write Enable (Active High)
add1	Write/Primary Read Address
add2	Dual Read Address
di	Data Input
do1	Primary Output Port
do2	Dual Output Port

VHDL Code

Following is the VHDL code.

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

entity raminfr is
  port (
    clk1 : in std_logic;
    clk2 : in std_logic;
    we   : in std_logic;
    add1 : in std_logic_vector(4 downto 0);
    add2 : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    do1  : out std_logic_vector(3 downto 0);
    do2  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_add1 : std_logic_vector(4 downto 0);
  signal read_add2 : std_logic_vector(4 downto 0);
begin
  process (clk1)
    begin
      if (clk1'event and clk1 = '1') then
        if (we = '1') then
          RAM(conv_integer(add1)) <= di;
        end if;
        read_add1 <= add1;
      end if;
  end process;
  do1 <= RAM(conv_integer(read_add1));

  process (clk2)
    begin
      if (clk2'event and clk2 = '1') then
        read_add2 <= add2;
      end if;
  end process;
  do2 <= RAM(conv_integer(read_add2));
end syn;
```

Verilog Code

Following is the Verilog code.

```verilog
module raminfr (clk1, clk2, we, addra, addrb, di, doa, dob);
  input clk1, clk2;
  input we;
  input [4:0] addra;
  input [4:0] addrb;
  input [3:0] di;
  output [3:0] doa;
  output [3:0] dob;
  reg [3:0] ram [31:0];
  reg [4:0] read_addra;
  reg [4:0] read_addrb;

  always @(posedge clk1) begin
    if (we == 1)
      begin
        ram[addra] <= di;
      end
      read_addra <= addra;
  end
  assign doa = ram[read_addra];

  always @(posedge clk2) begin
    read_addrb <= addrb;
  end
  assign dob = ram[read_addrb];
endmodule
```

Dual-Port RAM with One Enable Controlling Both Ports

The following descriptions are *directly mappable onto block RAM,* as shown in the following figure.
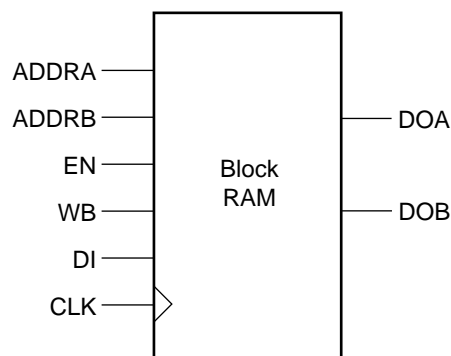
X9477

The following table shows pin descriptions for a dual-port RAM with one enable controlling both ports.

IO Pins	Description
clk	Positive-Edge Clock
en	Primary Global Enable (active High)
we	Primary Synchronous Write Enable (active High)
addra	Write Address/Primary Read Address
addrb	Dual Read Address
di	Primary Data Input
doa	Primary Output Port
dob	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with one global enable controlling both ports.

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

entity raminfr is
  port (
    clk   : in std_logic;
    en    : in std_logic;
    we    : in std_logic;
    addra : in std_logic_vector(4 downto 0);
    addrb : in std_logic_vector(4 downto 0);
    di    : in std_logic_vector(3 downto 0);
    doa   : out std_logic_vector(3 downto 0);
    dob   : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_addra : std_logic_vector(4 downto 0);
  signal read_addrb : std_logic_vector(4 downto 0);
```

```
                    begin
                      process (clk)
                        begin
                          if (clk'event and clk = '1') then
                            if (en = '1') then
                              if (we = '1') then
                                RAM(conv_integer(addra)) <= di;
                              end if;
                              read_addra <= addra;
                              read_addrb <= addrb;
                            end if;
                          end if;
                      end process;
                      doa <= RAM(conv_integer(read_addra));
                      dob <= RAM(conv_integer(read_addrb));
                    end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with one global enable controlling both ports.

```
                    module raminfr (clk, en, we, addra, addrb, di, doa, dob);
                      input clk;
                      input en;
                      input we;
                      input [4:0] addra;
                      input [4:0] addrb;
                      input [3:0] di;
                      output [3:0] doa;
                      output [3:0] dob;

                      reg [3:0] ram [31:0];
                      reg [4:0] read_addra;
                      reg [4:0] read_addrb;

                      always @(posedge clk)
                      begin
                        if (ena)
                          begin
                            if (wea)
                              ram[addra] <= di;
                            read_aaddra <= addra;
                            read_aaddrb <= addrb;
                          end
                      end
                      assign doa = ram[read_addra];
                      assign dob = ram[read_addrb];
                    endmodule
```
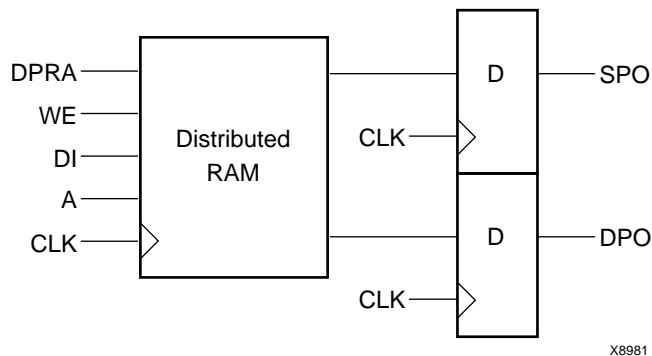
Dual-Port RAM with Enable on Each Port

The following descriptions are *directly mappable onto block RAM,* as shown in the following figure.
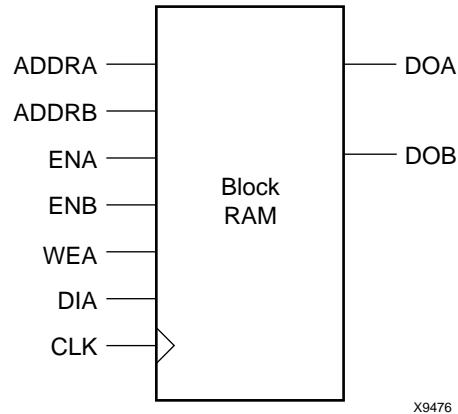
The following table shows pin descriptions for a dual-port RAM with enable on each port.

IO Pins	Description
clk	Positive-Edge Clock
ena	Primary Global Enable (Active High)
enb	Dual Global Enable (Active High)
wea	Primary Synchronous Write Enable (Active High)
addra	Write Address/Primary Read Address
addrb	Dual Read Address
dia	Primary Data Input
doa	Primary Output Port
dob	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with enable on each port.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity raminfr is
port (
  clk   : in std_logic;
  ena, enb : in std_logic;
  wea   : in std_logic;
  addra : in std_logic_vector(4 downto 0);
  addrb : in std_logic_vector(4 downto 0);
  dia   : in std_logic_vector(3 downto 0);
  doa   : out std_logic_vector(3 downto 0);
  dob   : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_addra : std_logic_vector(4 downto 0);
  signal read_addrb : std_logic_vector(4 downto 0);
  begin

    process (clk)
    begin
      if (clk'event and clk = '1') then
        if (ena = '1') then
          if (wea = '1') then
            RAM(conv_integer(addra)) <= dia;
          end if;
          read_addra <= addra;
        end if;
      end if;
    end process;

    process (clk)
    begin
      if (clk'event and clk = '1') then
        if (enb = '1') then
          read_addrb <= addrb;
        end if;
      end if;
    end process;

    doa <= RAM(conv_integer(read_addra));
    dob <= RAM(conv_integer(read_addrb));
end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with enable on each port.

```verilog
module raminfr (clk, ena, enb, wea, addra, addrb, dia, doa, dob);

    input clk, ena, enb, wea;
    input [4:0] addra, addrb;
    input [3:0] dia;
    output [3:0] doa, dob;

    reg [3:0] ram [31:0];
    reg [4:0] read_addra, read_addrb;

always @(posedge clk) begin
    if (ena) begin
      if (wea) begin
        ram[addra] <= dia;
      end
    end
end

always @(posedge clk) begin
    if (enb) begin
      read_addrb <= addrb;
    end
end

assign doa = ram[read_addra];
assign dob = ram[read_addrb];
endmodule
```

Dual-Port Block RAM with Different Clocks

The following example shows where the two clocks are used.
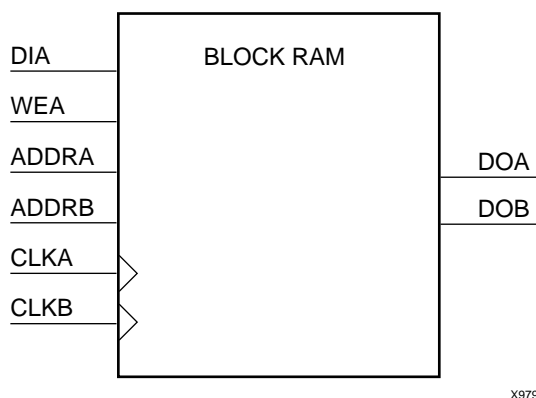
X9799

The following table shows pin descriptions for a dual-port RAM with different clocks.

IO Pins	Description
clka	Positive-Edge Clock
clkb	Positive-Edge Clock

IO Pins	Description
wea	Primary Synchronous Write Enable (Active High)
addra	Write Address/Primary Read Address
addrb	Dual Read Address
dia	Primary Data Input
doa	Primary Output Port
dob	Dual Output Port

VHDL Code

Following is the VHDL code for a dual-port RAM with different clocks.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity raminfr is
  port (
    clka  : in std_logic;
    clkb  : in std_logic;
    wea   : in std_logic;
    addra : in std_logic_vector(4 downto 0);
    addrb : in std_logic_vector(4 downto 0);
    dia   : in std_logic_vector(3 downto 0);
    doa   : out std_logic_vector(3 downto 0);
    dob   : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_addra : std_logic_vector(4 downto 0);
  signal read_addrb : std_logic_vector(4 downto 0);
  begin
    process (clka)
    begin
      if (clka'event and clka = '1') then
        if (wea = '1') then
          RAM(conv_integer(addra)) <= dia;
        end if;
        read_addra <= addra;
      end if;
    end process;
```

```
        process (clkb)
        begin
          if (clkb'event and clkb = '1') then
            read_addrb <= addrb;
          end if;
        end process;
        doa <= RAM(read_addra);
        dob <= RAM(read_addrb);
      end syn;
```

Verilog Code

Following is the Verilog code for a dual-port RAM with different clocks.

```
module raminfr (clka, clkb, wea, addra, addrb, dia, doa, dob);
  input clka;
  input clkb;
  input wea;
  input [4:0] addra;
  input [4:0] addrb;
  input [3:0] dia;
  output [3:0] doa;
  output [3:0] dob;

  reg [3:0] RAM [31:0];
  reg [4:0] read_addra;
  reg [4:0] read_addrb;

always @(posedge clka)
  begin
    if (wea == 1'b1)
      RAM[addra] <= dia;
    read_addra <= addra;
  end

always @(posedge clkb)
  begin
    read_addrb <= addrb;
  end

  assign doa = RAM[read_addra];
  assign dob = RAM[read_addrb];
endmodule
```

Dual-Port Block RAM with Two Write Ports

Starting with release 7.1i, XST supports dual-port block RAMs with two write ports. Please note that this feature supports for VHDL only.

The notion of dual-write ports implies not only distinct data ports, but also possibly distinct write clocks and write enables. Note that distinct write clocks also means distinct read clocks since the dual-port block RAM offers two clocks, one shared by the primary read and write port, the other shared by the secondary read and write port.

The description of this type of block RAM is based on the usage of shared variables. Until 7.1i release, shared variables were not supported and XST would error out with a specific message during VHDL parsing. In release 7.1i, the XST VHDL analyser accepts all shared

variables, but errors out in the HDL Synthesis step if a shared variable does not describe a valid RAM macro.

The following is the most general example; it has different clocks, enables and write enables.

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

entity raminfr is
  port(
    clka  : in  std_logic;
    clkb  : in  std_logic;
    ena   : in  std_logic;
    enb   : in  std_logic;
    wea   : in  std_logic;
    web   : in  std_logic;
    addra : in  std_logic_vector(7 downto 0);
    addrb : in  std_logic_vector(7 downto 0);
    dia   : in  std_logic_vector(15 downto 0);
    dib   : in  std_logic_vector(15 downto 0);
    doa   : out std_logic_vector(15 downto 0);
    dob   : out std_logic_vector(15 downto 0));
end raminfr;

architecture syn of raminfr is
  type RAMtype is array (0 to 255) of std_logic_vector(15 downto 0);
  shared variable RAM : RAMtype;
  begin

    process (CLKA)
    begin
      if CLKA'event and CLKA = '1' then
        if ENA = '1' then
          if WEA = '1' then
            RAM(conv_integer(ADDRA)) := DIA;
          end if;
          DOA <= RAM(conv_integer(ADDRA));
        end if;
      end if;
  end process;

    process (CLKB)
    begin
      if CLKB'event and CLKB = '1' then
        if ENB = '1' then
          if WEB = '1' then
            RAM(conv_integer(ADDRB)) := DIB;
          end if;
          DOB <= RAM(conv_integer(ADDRB));
        end if;
      end if;
    end process;
end syn;
```

Because of the shared variable, the description of the different read/write synchronizations may be different from templates recommended for single-write RAMs. Note that the order of appearance of the different lines of code becomes important.

Write-First

Write-first synchronization may be described using either of the following two templates.

Alternative 1

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    if WEA = '1' then
      RAM(conv_integer(ADDRA)) := DIA;
      DOA <= DIA;
    else
      DOA <= RAM(conv_integer(ADDRA));
    end if;
  end if;
end process;
```

Alternative 2

In this example, the read statement necessarily comes after the write statement.

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    if WEA = '1' then
      RAM(conv_integer(ADDRA)) := DIA;
    end if;
    DOA <= RAM(conv_integer(ADDRA));   -- The read statement must come
                                       -- AFTER the write statement
  end if;
end process;
```

Although they may look the same except for the signal/variable difference, it is also important to understand the functional difference between this template and the following well known template which describes a read-first synchronization in a single-write RAM:

```
signal RAM : RAMtype;

process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    if WEA = '1' then
      RAM(conv_integer(ADDRA)) <= DIA;
    end if;
    DOA <= RAM(conv_integer(ADDRA));
  end if;
end process;
```

Read-First

A read-first synchronization is described as follows, where the read statement must come BEFORE the write statement:

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    DOA <= RAM(conv_integer(ADDRA));  -- The read statement must come
                                      -- BEFORE the write statement
    if WEA = '1' then
      RAM(conv_integer(ADDRA)) := DIA;
    end if;
  end if;
end process;
```

No-Change

The following is a description for a no-change synchronization:

```
process (CLKA)
begin
  if CLKA'event and CLKA = '1' then
    if WEA = '1' then
      RAM(conv_integer(ADDRA)) := DIA;
    else
      DOA <= RAM(conv_integer(ADDRA));
    end if;
  end if;
end process;
```

Multiple-Port RAM Descriptions

XST can identify RAM descriptions with two or more read ports that access the RAM contents at addresses different from the write address. However, there can only be one write port. XST implements the following descriptions by replicating the RAM contents for each output port, as shown:
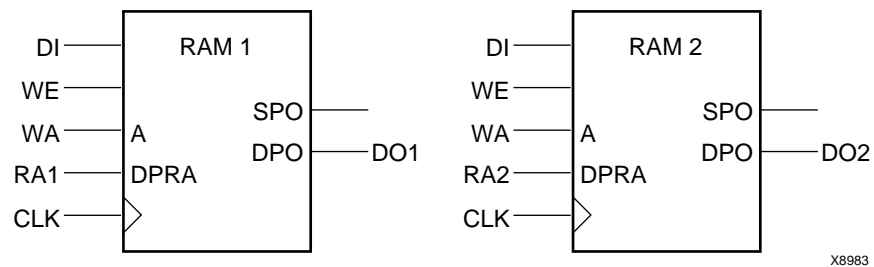
The following table shows pin descriptions for a multiple-port RAM.

IO pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
wa	Write Address

IO pins	Description
ra1	Read Address of the First RAM
ra2	Read Address of the Second RAM
di	Data Input
do1	First RAM Output Port
do2	Second RAM Output Port

VHDL Code

Following is the VHDL code for a multiple-port RAM.

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

entity raminfr is
  port (
    clk : in std_logic;
    we  : in std_logic;
    wa  : in std_logic_vector(4 downto 0);
    ra1 : in std_logic_vector(4 downto 0);
    ra2 : in std_logic_vector(4 downto 0);
    di  : in std_logic_vector(3 downto 0);
    do1 : out std_logic_vector(3 downto 0);
    do2 : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(wa)) <= di;
      end if;
    end if;
  end process;
  do1 <= RAM(conv_integer(ra1));
  do2 <= RAM(conv_integer(ra2));
end syn;
```

Verilog Code

Following is the Verilog code for a multiple-port RAM.

```
module raminfr (clk, we, wa, ra1, ra2, di, do1, do2);
  input clk;
  input we;
  input [4:0] wa;
  input [4:0] ra1;
  input [4:0] ra2;
  input [3:0] di;
  output [3:0] do1;
  output [3:0] do2;
  reg [3:0] ram [31:0];

always @(posedge clk)
  begin
    if (we)
      ram[wa] <= di;
    end
  assign do1 = ram[ra1];
  assign do2 = ram[ra2];
endmodule
```

Block RAM with Reset

XST supports block RAM with reset on the data outputs, as offered with Virtex™, Virtex-II™ and related block RAM resources. Optionally, you can include a synchronously controlled initialization of the RAM data outputs.

Block RAM with the following synchronization modes can have resetable data ports.

- Read-First Block RAM with Reset

- Write-First Block RAM with Reset

- No-Change Block RAM with Reset

- Registered ROM with Reset

- Supported Dual-Port Templates

  Note: Because XST does not support block RAMs with dual-write in a dual-read block RAM description, both data outputs may be reset, but the various read-write synchronizations are only allowed for the primary data output. The dual output may only be used in read-first mode.

The following example shows a Read-First Block RAM with reset.

X10019
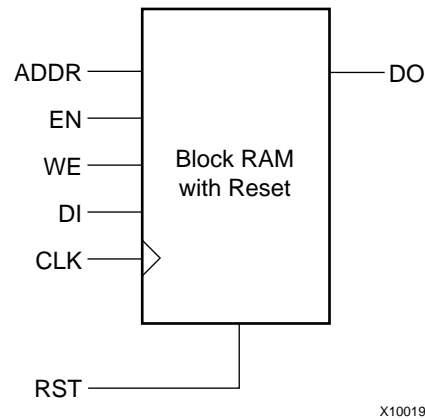
The following table shows pin descriptions for a block RAM with reset.

IO pins	Description
clk	Positive-Edge Clock
en	Global Enable
we	Write Enable (active High)
addr	Read/Write Address
rst	Reset for data output
di	Data Input
do	RAM Output Port

VHDL Code

Following is the VHDL code for a read-first RAM with reset.

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

entity ramrst is
  port (
    clk  : in std_logic;
    en   : in std_logic;
    we   : in std_logic;
    rst  : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    di   : in std_logic_vector(3 downto 0);
    do   : out std_logic_vector(3 downto 0));
end ramrst;
```

```
architecture syn of ramrst is
  type ram_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  signal ram : ram_type;

begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then  -- optional enable
        if we = '1' then  -- write enable
          ram(conv_integer(addr)) <= di;
        end if;
        if rst = '1' then -- optional reset
          do <= (others => '0');
        else
          do <= ram(conv_integer(addr)) ;
        end if;
      end if;
    end if;
  end process;
end syn;
```

Verilog Code

Following is the Verilog code for a read-first RAM with reset.

```
module raminfr (clk, en, we, rst, addr, di, do);
  input clk;
  input en;
  input we;
  input rst;
  input [4:0] addr;
  input [3:0] di;
  output [3:0] do;
  reg [3:0] ram [31:0];
  reg [3:0] do;
always @(posedge clk)
  begin
    if en // optional enable
      begin
        if we // write enable
          ram[addr] <= di;
        if rst // optional reset
          do <= reset_value;
        else
          do <= ram[addr];
      end
  end
endmodule
```

Block RAM with Optional Output Registers

For Virtex-4™ devices, XST supports block RAMs with optional output registers on the data outputs.

VHDL Code

Following is the VHDL code for a Block Ram with optional output registers.

```vhdl
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity raminfr is
  port (
    clk1, clk2   : in std_logic;
    we, en1, en2 : in std_logic;
    addr1        : in std_logic_vector(6 downto 0);
    addr2        : in std_logic_vector(6 downto 0);
    di           : in std_logic_vector(7 downto 0);
    res1         : out std_logic_vector(7 downto 0);
    res2         : out std_logic_vector(7 downto 0));
end raminfr;

architecture beh of raminfr is
  type mem_type is array (127 downto 0) of
         std_logic_vector (7 downto 0);
  signal mem    : mem_type;
  signal do1    : std_logic_vector(7 downto 0);
  signal do2    : std_logic_vector(7 downto 0);
begin
  process (clk1)
  begin
    if rising_edge(clk1) then
      if we = '1' then
        mem(conv_integer(addr1)) <= di;
      end if;
      do1 <= mem(conv_integer(addr1));
    end if;
  end process;

  process (clk2)
  begin
    if rising_edge(clk2) then
      do2 <= mem(conv_integer(addr2));
    end if;
  end process;

  process (clk1)
  begin
    if rising_edge(clk1) then
      if en1 = '1' then
        res1 <= do1;
      end if;
    end if;
  end process;
```

```
process (clk2)
begin
  if rising_edge(clk2) then
    if en2 = '1' then
      res2 <= do2;
    end if;
  end if;
end process;

end beh;
```

Verilog Code

Following is the Verilog code for a Block Ram with optional output registers.

```verilog
module raminfr (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);
  input clk1;
  input clk2;
  input we, en1, en2;
  input [6:0] addr1;
  input [6:0] addr2;
  input [7:0] di;
  output [7:0] res1;
  output [7:0] res2;

  reg [7:0] res1;
  reg [7:0] res2;

  reg [7:0] RAM [127:0];
  reg [7:0] do1;
  reg [7:0] do2;

always @(posedge clk1)
  begin
    if (we == 1'b1)
      RAM[addr1] <= di;
    do1 <= RAM[addr1];
  end

always @(posedge clk2)
  begin
    do2 <= RAM[addr2];
  end

always @(posedge clk1)
  begin
    if (en1 == 1'b1)
      res1 <= do1;
  end

always @(posedge clk2)
  begin
    if (en2 == 1'b1)
      res2 <= do2;
  end
endmodule
```

During the HDL Synthesis process XST recognizes BRAM and 2 optional output registers as separate objects from BRAM. It is at the Advanced HDL Synthesis step that XST pushes the optional registers to BRAM:.

```
=======================================================================
*                  HDL Synthesis                                   *
=======================================================================
...
   Found 128x8-bit dual-port block RAM for signal <mem>.
   -----------------------------------------------------------------
   | mode             | read-first                    |         |
   | dual mode        | read-first                    |         |
   | aspect ratio     | 128-word x 8-bit              |         |
   | clock            | connected to signal <clk1>    | rise    |
   | dual clock       | connected to signal <clk2>    | rise    |
   | write enable     | connected to signal <we>      | high    |
   | address          | connected to signal <addr1>|          |
   | dual address     | connected to signal <addr2>|          |
   | data in          | connected to signal <di>      |         |
   | data out         | connected to signal <do1>     |         |
   | dual data out    | connected to signal <do2>     |         |
   | ram_style        | Auto                          |         |
   -----------------------------------------------------------------
   Found 8-bit register for signal <res1>.
   Found 8-bit register for signal <res2>.

=======================================================================
*              Advanced HDL Synthesis                               *
=======================================================================
...
INFO:Xst:1954 - Data output of block RAM <Mram_mem> in block
<m_bram_1> is tied to register <res1> in block <m_bram_1>. The
register is absorbed by the block RAM and provides isolation from the
interconnect routing delay for maximal operating frequency.
INFO:Xst:1955 - Dual data output of block RAM <Mram_mem> in block
<m_bram_1> is tied to register <res2> in block <m_bram_1>. The
register is absorbed by the block RAM and provides isolation from the
interconnect routing delay for maximal operating frequency.
...
```

Initializing Block RAM

Block RAM initial contents can be specified by initialization of the signal describing the memory array in your HDL code. You can do this directly in your HDL code, or you can specify a file containing the initialization data.

Starting from release 7.1i, XST supports initialization for dual-port block RAMs. Please note that this mechanism is supported only for Virtex-II™, Virtex-II Pro™, Spartan-3™ and Virtex-4™ device families.

Note: XST does not support block RAM initialization in Verilog.

VHDL Code

- Specify block RAM initial contents by initializing the signal describing the memory array in your VHDL code as in the following example:

```
...
type ram_type is array (0 to 63) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
  (
  X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
  X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
  X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
  X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
  X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
  X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
  X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
  X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
  X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
  X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
  X"0030D", X"02341", X"08201", X"0400D");
...
process (clk)
begin
  if rising_edge(clk) then
    if we = '1' then
      RAM(conv_integer(a)) <= di;
    end if;
    ra <= a;
  end if;
end process;
...
do <= RAM(conv_integer(ra));
```

The block RAM initial contents can be specified in hexadecimal, as in the previous example, or in binary as shown in the following example:

```
...
type ram_type is array (0 to SIZE-1) of std_logic_vector(15 downto 0);
signal RAM : ram_type :=
  (
  "0111100100000101",
  "0000010110111101",
  "1100001101010000",
  ...
  "0000100101110011");
...
```

- Specify a dual-port block RAM initial contents using the following VHDL code:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity raminfr is
  port (
    clk1  : in std_logic;
    clk2  : in std_logic;
    we    : in std_logic;
    addr1 : in std_logic_vector(7 downto 0);
    addr2 : in std_logic_vector(7 downto 0);
    di    : in std_logic_vector(15 downto 0);
    do1   : out std_logic_vector(15 downto 0);
    do2   : out std_logic_vector(15 downto 0));
end raminfr;

architecture syn of raminfr is
  type mem_type is array (255 downto 0)
        of std_logic_vector (15 downto 0);
  signal mem  : mem_type:= (255 downto 100 => X"B8B8",
                            99 downto 0   => X"8282");
begin
  process (clk1)
  begin
    if rising_edge(clk1) then
      if we = '1' then
        mem(conv_integer(addr1)) <= di;
      end if;
      do1 <= mem(conv_integer(addr1));
    end if;
  end process;

  process (clk2)
  begin
    if rising_edge(clk2) then
      do2 <= mem(conv_integer(addr2));
    end if;
  end process;
end syn;
```

- To initialize block RAM from values contained in an external file, use a read function in your VHDL code. Please refer to the "File Type Support" in Chapter 6 for more information. Set up the initialization file as follows.

  - Use each line of the initialization file to represent the initial contents of a given row in the RAM.

  - RAM contents can be represented in binary or hexadecimal.

  - There should be as many lines in the file as there are rows in the block RAM array.

  - Below is a sample of the contents of a file initializing an 8 x 32-bit RAM with binary values:

    ```
    00001111000011110000111100001111
    01001010001000001100000010000100
    00000000001111100000000001000001
    11111101010000011100010000100100
    00001111000011110000111100001111
    01001010001000001100000010000100
    00000000001111100000000001000001
    11111101010000011100010000100100
    ```

Example:

In the following example, the loop that generates the initial value is controlled by testing that we are in the RAM address range.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity raminitfile1 is

  port( clk : in std_logic;
    we   : in std_logic;
    addr : in std_logic_vector(2 downto 0);
    din  : in std_logic_vector(31 downto 0);
    dout : out std_logic_vector(31 downto 0));

end raminitfile1;
```

```
architecture archi of raminitfile1 is

   type RamType is array(0 to 7) of bit_vector(31 downto 0);

   type RamType is array(0 to 63) of bit_vector(15 downto 0);

   function InitRamFromFile (RamFileName : in string) return
     RamType is
       FILE RamFile : text is in RamFileName;
         variable RamFileLine : line;
         variable RAM : RamType;
       begin
         for I in RamType'range loop
           readline (RamFile, RamFileLine);
           read (RamFileLine, RAM(I));
         end loop;
     return RAM;
end function;

   signal RAM : RamType := InitRamFromFile("raminitfile1.data");

     begin
       process (clk)
       begin
         if clk'event and clk = '1' then
           if we = '1' then
             RAM(conv_integer(addr)) <= to_bitvector(din);
           end if;
             dout <= to_stdlogicvector(RAM(conv_integer(addr)));
         end if;
       end process;
end archi;
```

Note: If there are not enough lines in the external data file, XST will issue the following message.

```
ERROR:Xst – raminitfile1.vhd line 40: Line <RamFileLine> has not enough
elements for target <RAM<63>>.
```

Verilog Code

XST does not support block RAM initialization in Verilog.

Limitations

- Initialization is only valid for block RAM resources. If you attempt to initialize distributed RAM, XST ignores the initialization, and issues a warning message.

- Initialization of inferred RAMs from RTL code is not supported via INIT constraints. Use of INIT constraints is only supported if RAM primitives are directly instantiated from the UNISIM library.

ROMs Using Block RAM Resources

XST can use block RAM resources to implement ROMs with synchronous outputs or address inputs. These ROMs are implement as single-port block RAMs. The use of block RAM resources to implement ROMs is controlled by the ROM_STYLE constraint. Please see Chapter 5, "Design Constraints" for details about the ROM_SYTLE attribute. Please see Chapter 3, "FPGA Optimization" for details on ROM implementation.

Here is a list of VHDL/Verilog templates described below.

- ROM with registered output
- ROM with registered address

The following table shows pin descriptions for a registered ROM.

IO Pins	Description
clk	Positive-Edge Clock
en	Synchronous Enable (active High)
addr	Read Address
data	Data Output

VHDL Code

Following is the recommended VHDL code for a ROM with registered output.

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

entity rominfr is
  port (
    clk  : in std_logic;
    en   : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    data : out std_logic_vector(3 downto 0));
end rominfr;
architecture syn of rominfr is
  type rom_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  constant ROM : rom_type :=(
    "0001","0010","0011","0100","0101","0110","0111","1000","1001",
    "1010","1011","1100","1101","1110","1111","0001","0010","0011",
    "0100","0101","0110","0111","1000","1001","1010","1011","1100",
    "1101","1110","1111");
  begin
    process (clk)
    begin
      if (clk'event and clk = '1') then
        if (en = '1') then
          data <= ROM(conv_integer(addr);
        end if;
      end if;
    end process;
end syn;
```

Following is alternate VHDL code for a ROM with registered output.

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

entity rominfr is
  port (
    clk  : in std_logic;
    en   : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    data : out std_logic_vector(3 downto 0));
end rominfr;
architecture syn of rominfr is
  type rom_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  constant ROM : rom_type :=(
    "0001","0010","0011","0100","0101","0110","0111","1000","1001",
    "1010","1011","1100","1101","1110","1111","0001","0010","0011",
    "0100","0101","0110","0111","1000","1001","1010","1011","1100",
    "1101","1110","1111");
  signal rdata : std_logic_vector(3 downto 0);

  begin
    rdata <= ROM(conv_integer(addr));

    process (clk)
    begin
      if (clk'event and clk = '1') then
        if (en = '1') then
          data <= rdata;
        end if;
      end if;
    end process;
end syn;
```

Following is VHDL code for a ROM with registered address.

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

entity rominfr is
  port (
    clk  : in std_logic;
    en   : in std_logic;
    addr : in std_logic_vector(4 downto 0);
    data : out std_logic_vector(3 downto 0));
end rominfr;
architecture syn of rominfr is
  type rom_type is array (31 downto 0) of std_logic_vector (3 downto 0);
  constant ROM : rom_type :=
("0001","0010","0011","0100","0101","0110","0111","1000","1001","1010"
,"1011","1100","1101","1110","1111","0001","0010","0011","0100","0101"
,"0110","0111","1000","1001","1010","1011","1100","1101","1110","1111"
  );
  signal raddr : std_logic_vector(4 downto 0);

begin
  process (clk)
    begin
      if (clk'event and clk = '1') then
        if (en = '1') then
          raddr <= addr;
        end if;
      end if;
    end process;
  data <= ROM(conv_integer(raddr));
end syn;
```

Verilog Code

Following is Verilog code for a ROM with registered output.

```verilog
module rominfr (clk, en, addr, data);
  input clk;
  input en;
  input [4:0] addr;
  output reg [3:0] data;

  always @(posedge clk) begin
    if (en)
      case(addr)
        4'b0000: data <= 4'b0010;
        4'b0001: data <= 4'b0010;
        4'b0010: data <= 4'b1110;
        4'b0011: data <= 4'b0010;
        4'b0100: data <= 4'b0100;
        4'b0101: data <= 4'b1010;
        4'b0110: data <= 4'b1100;
        4'b0111: data <= 4'b0000;
        4'b1000: data <= 4'b1010;
        4'b1001: data <= 4'b0010;
        4'b1010: data <= 4'b1110;
        4'b1011: data <= 4'b0010;
        4'b1100: data <= 4'b0100;
        4'b1101: data <= 4'b1010;
        4'b1110: data <= 4'b1100;
        4'b1111: data <= 4'b0000;
        default: data <= 4'bXXXX;
      endcase
  end
endmodule
```

Following is Verilog code for a ROM with registered address.

```verilog
module rominfr (clk, en, addr, data);
  input clk;
  input en;
  input  [4:0] addr;
  output reg [3:0] data;
  reg    [4:0] raddr;
  always @(posedge clk) begin
    if (en)
      raddr <= addr;
  end
  always @(raddr) begin
    if (en)
      case(raddr)
        4'b0000: data = 4'b0010;
        4'b0001: data = 4'b0010;
        4'b0010: data = 4'b1110;
        4'b0011: data = 4'b0010;
        4'b0100: data = 4'b0100;
        4'b0101: data = 4'b1010;
        4'b0110: data = 4'b1100;
        4'b0111: data = 4'b0000;
        4'b1000: data = 4'b1010;
        4'b1001: data = 4'b0010;
        4'b1010: data = 4'b1110;
        4'b1011: data = 4'b0010;
        4'b1100: data = 4'b0100;
        4'b1101: data = 4'b1010;
        4'b1110: data = 4'b1100;
        4'b1111: data = 4'b0000;
        default: data = 4'bXXXX;
      endcase
  end
endmodule
```

State Machine

XST proposes a large set of templates to describe Finite State Machines (FSMs). By default, XST tries to distinguish FSMs from VHDL/Verilog code, and apply several state encoding techniques (it can re-encode the user's initial encoding) to get better performance or less area. However, you can disable FSM extraction by using the FSM_EXTRACT design constraint.

Note: XST can handle only synchronous state machines.

There are many ways to describe FSMs. A traditional FSM representation incorporates Mealy and Moore machines, as in the following figure. Please note that XST supports both of these models:

Only for Mealy Machine
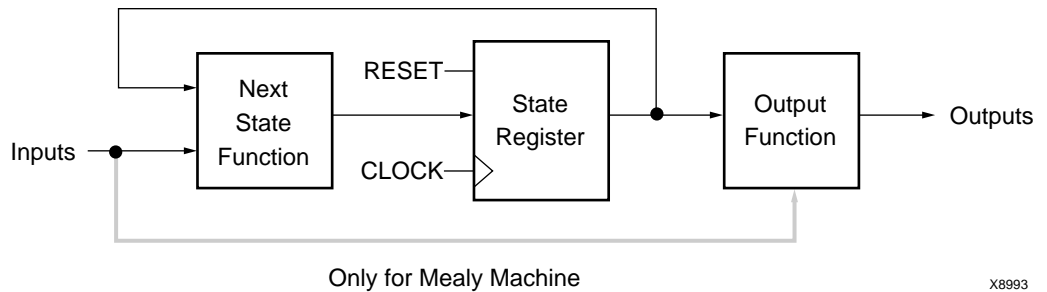
X8993

For HDL, process (VHDL) and always blocks (Verilog) are the most suitable ways for describing FSMs. (For description convenience Xilinx® uses "process" to refer to both: VHDL processes and Verilog always blocks.)

You may have several processes (1, 2 or 3) in your description, depending upon how you consider and decompose the different parts of the preceding model. Following is an example of the Moore Machine with Asynchronous Reset, "RESET".

- 4 states: s1, s2, s3, s4
- 5 transitions
- 1 input: "x1"
- 1 output: "outp"

This model is represented by the following bubble diagram:
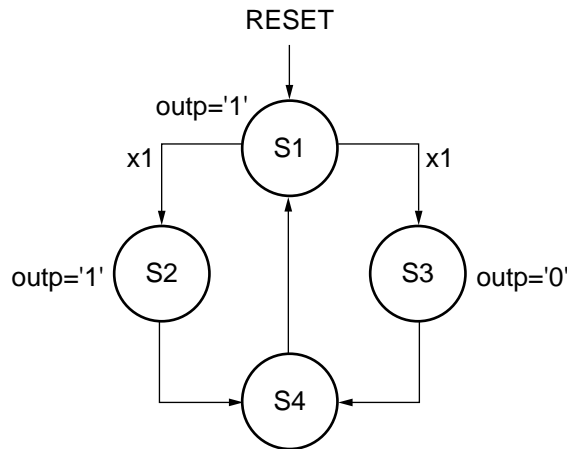
FSM with 1 Process

Please note, in this example output signal "outp" is a *register*.

VHDL Code

Following is the VHDL code for an FSM with a single process.

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

entity fsm is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic);
end entity;

architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type;
begin
  process (clk, reset)
  begin
    if (reset ='1') then
      state <= s1;
      outp <= '1';
    elsif (clk='1' and clk'event) then
      case state is
        when s1 =>
          if x1='1' then
            state <= s2;
            outp <= '1';
          else
            state <= s3;
            outp <= '0';
          end if;
        when s2 => state <= s4; outp <= '0';
        when s3 => state <= s4; outp <= '0';
        when s4 => state <= s1; outp <= '1';
      end case;
    end if;
  end process;
end beh1;
```

Verilog Code

Following is the Verilog code for an FSM with a single process.

```verilog
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  parameter s1 = 2'b00; parameter s2 = 2'b01;
  parameter s3 = 2'b10; parameter s4 = 2'b11;

  always@(posedge clk or posedge reset)
  begin
    if (reset)
      begin
        state <= s1; outp <= 1'b1;
      end
    else
      begin
        case (state)
          s1: begin
            if (x1 == 1'b1)
              begin
                state <= s2;
                outp <= 1'b1;
              end
            else
              begin
                state <= s3;
                outp <= 1'b1;
              end
          end
          s2: begin
            state <= s4; outp <= 1'b0;
          end
          s3: begin
            state <= s4; outp <= 1'b0;
          end
          s4: begin
            state <= s1; outp <= 1'b1;
          end
        endcase
      end
  end
endmodule
```

FSM with 2 Processes

To eliminate a register from the "outputs", you can remove all assignments "outp <=…" from the Clock synchronization section.

This can be done by introducing two processes as shown in the following figure.
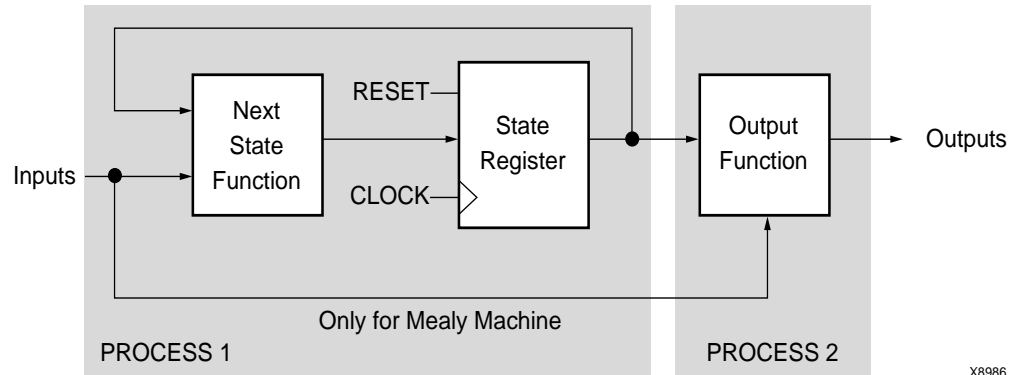
VHDL Code

Following is VHDL code for an FSM with two processes.

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

entity fsm is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic);
end entity;
architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type;
begin
  process1: process (clk, reset)
  begin
    if (reset ='1') then
      state <= s1;
    elsif (clk='1' and clk'Event) then
      case state is
        when s1 =>
          if x1='1' then
            state <= s2;
          else
            state <= s3;
          end if;
        when s2 => state <= s4;
        when s3 => state <= s4;
        when s4 => state <= s1;
      end case;
    end if;
  end process process1;
```

```
                    process2 : process (state)
                      begin
                        case state is
                          when s1 => outp <= '1';
                          when s2 => outp <= '1';
                          when s3 => outp <= '0';
                          when s4 => outp <= '0';
                        end case;
                    end process process2;
                    end beh1;
```

Verilog Code

Following is the Verilog code for an FSM with two processes.

```verilog
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  parameter s1 = 2'b00; parameter s2 = 2'b01;
  parameter s3 = 2'b10; parameter s4 = 2'b11;

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
      begin
        case (state)
          s1: if (x1 == 1'b1)
                  state <= s2;
                else
                  state <= s3;
          s2: state <= s4;
          s3: state <= s4;
          s4: state <= s1;
        endcase
      end
  end

  always @(state)
  begin
    case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
    endcase
  end
endmodule
```

FSM with 3 Processes

You can also separate the NEXT State function from the state register:
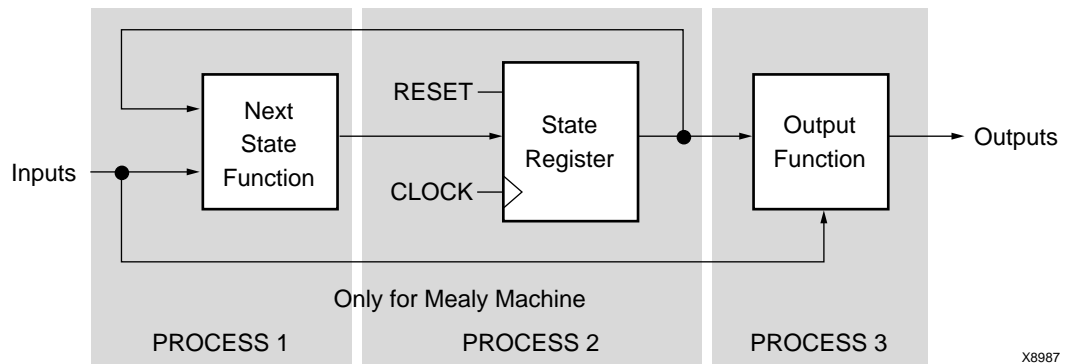
Separating the NEXT State function from the state register provides the following description:

VHDL Code

Following is the VHDL code for an FSM with three processes.

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

entity fsm is
  port (
    clk, reset, x1 : IN std_logic;
    outp : OUT std_logic);
end entity;

architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state, next_state: state_type;
begin

  process1: process (clk, reset)
  begin
    if (reset ='1') then
      state <= s1;
    elsif (clk = '1' and clk'Event) then
      state <= next_state;
    end if;
  end process process1;
```

```
                    process2 : process (state, x1)
                    begin
                      case state is
                        when s1 =>
                          if x1='1' then
                            next_state <= s2;
                          else
                            next_state <= s3;
                          end if;
                        when s2 => next_state <= s4;
                        when s3 => next_state <= s4;
                        when s4 => next_state <= s1;
                      end case;
                    end process process2;
                    process3 : process (state)
                      begin
                        case state is
                          when s1 => outp <= '1';
                          when s2 => outp <= '1';
                          when s3 => outp <= '0';
                          when s4 => outp <= '0';
                        end case;
                    end process process3;
                  end beh1;
```

Verilog Code

Following is the Verilog code for an FSM with three processes.

```
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  reg [1:0] next_state;
  parameter s1 = 2'b00; parameter s2 = 2'b01;
  parameter s3 = 2'b10; parameter s4 = 2'b11;

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state <= s1;
    else
      state <= next_state;
  end

  always @(state or x1)
  begin
    case (state)
      s1:
        if (x1 == 1'b1)
          next_state = s2;
        else
      s2: next_state = s4;
      s3: next_state = s4;
      s4: next_state = s1;
    endcase
  end
```

```
      always @(state)
      begin
        case (state)
          s1: outp = 1'b1;
          s2: outp = 1'b1;
          s3: outp = 1'b0;
          s4: outp = 1'b0;
        endcase
      end
    endmodule
```

State Registers

State registers must be initialized with an asynchronous or synchronous signal. XST does not support FSM without initialization signals. Please refer to "Registers" in this chapter for templates on how to write Asynchronous and Synchronous initialization signals.

In VHDL, the type of a state register can be a different type: integer, bit_vector, std_logic_vector, for example. But it is common and convenient to define an enumerated type containing all possible state values and to declare your state register with that type.

In Verilog, the type of state register can be an integer or a set of defined parameters. In the following Verilog examples the state assignments could have been made like this:

```
    parameter [3:0]
      s1 = 4'b0001,
      s2 = 4'b0010,
      s3 = 4'b0100,
      s4 = 4'b1000;
    reg [3:0] state;
```

These parameters can be modified to represent different state encoding schemes.

Next State Equations

Next state equations can be described directly in the sequential process or in a distinct combinational process. The simplest template is based on a Case statement. If using a separate combinational process, its sensitivity list should contain the state signal and all FSM inputs.

Unreachable States

XST can detect unreachable states in an FSM. It lists them in the log file in the HDL Synthesis step.

FSM Outputs

Non-registered outputs are described either in the combinational process or in concurrent assignments. Registered outputs must be assigned within the sequential process.

FSM Inputs

Registered inputs are described using internal signals, which are assigned in the sequential process.

State Encoding Techniques

XST supports the following state encoding techniques.

- Auto
- One-Hot
- Gray
- Compact
- Johnson
- Sequential
- User
- Speed1

Auto

In this mode, XST tries to select the best suited encoding algorithm for each FSM.

One-Hot

One-hot encoding is the default encoding scheme. Its principle is to associate one code bit and also one flip-flop to each state. At a given clock cycle during operation, one and only one bit of the state variable is asserted. Only two bits toggle during a transition between two states. One-hot encoding is very appropriate with most FPGA targets where a large number of flip-flops are available. It is also a good alternative when trying to optimize speed or to reduce power dissipation.

Gray

Gray encoding guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In addition, this coding technique minimizes hazards and glitches. Very good results can be obtained when implementing the state register with T flip-flops.

Compact

Compact encoding consists of minimizing the number of bits in the state variables and flip-flops. This technique is based on hypercube immersion. Compact encoding is appropriate when trying to optimize area.

Johnson

Like Gray, Johnson encoding shows benefits with state machines containing long paths with no branching.

Sequential

Sequential encoding consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.

Speed1

Speed1 encoding is oriented for speed optimization. The number of bits for a state register depends on the particular FSM, but generally it is greater than the number of FSM states.

User

In this mode, XST uses original encoding, specified in the HDL file. For example, if you use enumerated types for a state register, then in addition you can use the ENUM_ENCODING constraint to assign a specific binary value to each state. Please refer to Chapter 5, "Design Constraints" for more details.

Log File

The XST log file reports the full information of recognized FSM during the Macro Recognition step. Moreover, if you allow XST to choose the best encoding algorithm for your FSMs, it reports the one it chose for each FSM.

As soon as encoding is selected, XST reports the original and final FSM encoding. Please note that if the target family is an FPGA, then XST reports this encoding at the HDL

Synthesis step. If the target family is a CPLD, then XST reports this encoding at the Low Level Optimization step.

```
...
Synthesizing Unit <fsm>.
    Related source file is state_machines_1.vhd.
    Found finite state machine <FSM_0> for signal <state>.
    -------------------------------------------------------------
    | States             | 4                                     |
    | Transitions        | 5                                     |
    | Inputs             | 1                                     |
    | Outputs            | 1                                     |
    | Clock              | clk (rising_edge)                     |
    | Reset              | reset (positive)                      |
    | Reset type         | asynchronous                          |
    | Reset State        | s1                                    |
    | Power Up State     | s1                                    |
    | Encoding           | automatic                             |
    | Implementation     | LUT                                   |
    -------------------------------------------------------------
 ...
    Summary:
        inferred   1 Finite State Machine(s).
 ...
  Unit <fsm> synthesized.

=========================================================
*                 Advanced HDL Synthesis               *
=========================================================
 ...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <FSM_0> on signal <state[1:4]> with Speed1 encoding.
------------------
State | Encoding
------------------
s1    | 1010
s2    | 0011
s3    | 0001
s4    | 0100
------------------
===============================================================
  HDL Synthesis Report

  Macro Statistics
  # FSMs                               : 1
  # Registers                          : 1
    1-bit register                     : 1

===============================================================
 ...
```

RAM-based FSM Synthesis

Large FSMs can be made more compact and faster by implementing them in the block RAM resources provided in Virtex™ and later technologies. You can direct XST to use block RAM resources for FSMs by using the FSM_STYLE constraint. Values for FSM_STYLE are *lut* and *bram*. The *lut* option is the default and it causes XST to map the FSM using LUTs. The *bram* option directs XST to map the FSM onto block RAM.

In Project Navigator, invoke this constraint by choosing either **LUT** or **Bram** from the drop down list to the right of FSM Style under the HDL Options tab of the Process Properties dialog box. From the command line, use the –fsm_style command line switch. You can also use the FSM_STYLE constraint in your HDL code. See the *Constraints Guide* for more information.

If it cannot implement a state machine on block RAM, XST:

- generates a warning message with the reason for the warning in the Advanced HDL Synthesis step of the log file.

- automatically implements the state machine using LUTs.

For example, if FSM has a asynchronous reset, it cannot be implemented using block RAM. In this case XST informs the user:

```
...
=======================================================================
*                         Advanced HDL Synthesis                      *
=======================================================================

WARNING:Xst - Unable to fit FSM <FSM_0> in BRAM (reset is
asynchronous).
Selecting encoding for FSM_0 ...
Optimizing FSM <FSM_0> on signal <current_state> with one-hot
encoding.
...
```

Safe FSM Implementation

XST can add logic to your FSM implementation that will let your state machine recover from an invalid state. If during its execution, a state machine gets into an invalid state, the logic added by XST will bring it back to a known state, called a recovery state. This is known as Safe Implementation mode. To activate Safe FSM implementation, select the Safe Implementation option from the Synthesis Properties dialog box in Project Navigator or apply the SAFE_IMPLEMENTATION constraint to the hierarchical block or signal that represents the state register. See "Safe Implementation" in Chapter 5 for details about the SAFE_IMPLEMENTATION constraint.

By default, XST automatically selects a reset state as the recovery state. If the FSM does not have an initialization signal, XST selects a power-up state as the recovery state. You can manually define the recovery state by applying the RECOVERY_STATE constraint. See "Recovery State" in Chapter 5 for details about the RECOVERY_STATE constraint.

Black Box Support

Your design may contain EDIF or NGC files generated by synthesis tools, schematic editors or any other design entry mechanism. These modules must be instantiated in your code to be connected to the rest of your design. You can do this in XST by using black box instantiation in the VHDL/Verilog code. The netlist is propagated to the final top-level netlist without being processed by XST. Moreover, XST enables you to attach specific constraints to these black box instantiations, which are passed to the NGC file.

In addition, you may have a design block for which you have an RTL model, as well as your own implementation of this block in the form of an EDIF netlist. The RTL model is only valid for simulation purposes, but by using the BOX_TYPE constraint you can direct XST to skip synthesis of this RTL code and create a black box. The EDIF netlist is linked to the synthesized design during NGDBuild. Please see "General Constraints" in Chapter 5 for more information. Also see the *Constraints Guide* for details.

Note: Remember that once you make a design a black box, each instance of that design is a black box. While you can attach constraints to the instance, XST ignores any constraint attached to the original design.

Log File

From the flow point of view, the recognition of black boxes in XST is done before the macro inference process. Therefore the LOG file differs from the one generated for other macros.

```
...
Analyzing Entity <black_b> (Architecture <archi>).

 WARNING:Xst:766 - black_box_1.vhd (Line 15). Generating a Black Box
for component <my_block>.
 Entity <black_b> analyzed. Unit <black_b> generated
....
```

Related Constraints

XST has a BOX_TYPE constraint that can be applied to black boxes. However, it was introduced essentially for Virtex™ Primitive instantiation in XST. Please read "Virtex™ Primitive Support" in Chapter 3 in before using this constraint.

VHDL Code

Following is the VHDL code for a black box.

```
library ieee;
use ieee.std_logic_1164.all;

entity black_b is
  port(
    DI_1, DI_2 : in std_logic;
    DOUT : out std_logic);
end black_b;
```

```
architecture archi of black_b is
  component my_block
    port (
       I1 : in std_logic;
       I2 : in std_logic;
       O : out std_logic);
  end component;

begin
  inst: my_block port map (
    I1=>DI_1,
    I2=>DI_2,
    O=>DOUT);
end archi;
```

Verilog Code

Following is the Verilog code for a black box.

```
module my_block (in1, in2, dout);
  input in1, in2;
  output dout;
endmodule

module black_b (DI_1, DI_2, DOUT);
  input DI_1, DI_2;
  output DOUT;
  my_block inst (
    .in1(DI_1),
    .in2(DI_2),
    .dout(DOUT)
    );
endmodule
```

Note: Please refer to the VHDL/Verilog language reference manuals for more information on component instantiation.

FPGA Optimization

This chapter contains the following sections:

- *"Introduction"*
- *"Virtex™ Specific Synthesis Options"*
- *"Macro Generation"*
- *"Using DSP48 Block Resources"*
- *"Mapping Logic onto Block RAM"*
- *"Flip-Flop Retiming"*
- *"Incremental Synthesis Flow"*
- *"Speed Optimization Under Area Constraint"*
- *"Log File Analysis"*
- *"Implementation Constraints"*
- *"Virtex™ Primitive Support"*
- *"Cores Processing"*
- *"Specifying INITs and RLOCs in HDL Code"*
- *"PCI Flow"*

Introduction

XST performs the following steps during FPGA synthesis and optimization:

- Mapping and optimization on an entity/module by entity/module basis.
- Global optimization on the complete design.

The output of this process is an NGC file.

This chapter describes the following:

- Constraints that can be applied to tune the synthesis and optimization process.
- Macro generation.
- Information in the log file.
- Timing model used during the synthesis and optimization process.
- Constraints available for timing-driven synthesis.
- Information on the generated NGC file.
- Information on support for primitives.

Virtex™ Specific Synthesis Options

XST supports a set of options that allows the tuning of the synthesis process according to the user constraints. This section lists the options that relate to the FPGA-specific optimization of the synthesis process. For details about each option, see "FPGA Constraints (non-timing)" in Chapter 5.

Following is a list of FPGA options.

- BUFGCE
- Buffer Type
- Clock Buffer Type
- Decoder Extraction
- FSM Style
- Global Optimization Goal
- Incremental Synthesis
- Keep Hierarchy
- Logical Shifter Extraction
- Map Logic on BRAM
- Max Fanout
- Move First Stage
- Move Last Stage
- Multiplier Style
- Mux Style
- Number of Global Clock Buffers
- Optimize Instantiated Primitives
- Pack I/O Registers into IOBs
- Priority Encoder Extraction
- RAM Style
- Register Balancing
- Register Duplication
- Resynthesize
- Shift Register Extraction
- Signal Encoding
- Slice Packing
- Use Carry Chain
- Write Timing Constraints
- XOR Collapsing

Macro Generation

The Virtex™ Macro Generator module provides the XST HDL Flow with a catalog of functions. These functions are identified by the inference engine from the HDL description; their characteristics are handed to the Macro Generator for optimal implementation. The set of inferred functions ranges in complexity from simple arithmetic operators such as adders, accumulators, counters and multiplexers to more complex building blocks such as multipliers, shift registers and memories.

Inferred functions are optimized to deliver the highest levels of performance and efficiency for Virtex™ architectures and then integrated into the rest of the design. In addition, the generated functions are optimized through their borders depending on the design context.

This section categorizes, by function, all available macros and briefly describes technology resources used in the building and optimization phase.

Macro Generation can be controlled through attributes. These attributes are listed in each subsection. For general information on attributes see Chapter 5, "Design Constraints".

XST uses dedicated carry chain logic to implement many macros. In some situations carry chain logic may lead to sub-optimal optimization results. Use the USE_CARRY_CHAIN constraint to direct XST to deactivate this feature. Please refer to Chapter 5, "Design Constraints" for more information.

Arithmetic Functions

For Arithmetic functions, XST provides the following elements:

- Adders, Subtracters and Adder/Subtracters
- Cascadable Binary Counters
- Accumulators
- Incrementers, Decrementers and Incrementer/Decrementers
- Signed and Unsigned Multipliers

XST uses fast carry logic (MUXCY) to provide fast arithmetic carry capability for high-speed arithmetic functions. The sum logic formed from two XOR gates is implemented using LUTs and the dedicated carry-XORs (XORCY). In addition, XST benefits from a dedicated carry-ANDs (MULTAND) resource for high-speed multiplier implementation.

Loadable Functions

For Loadable functions XST provides the following elements.

- Loadable Up, Down and Up/Down Binary Counters
- Loadable Up, Down and Up/Down Accumulators

XST can provide synchronously loadable, cascadable binary counters and accumulators inferred in the HDL flow. Fast carry logic is used to cascade the different stages of the macros. Synchronous loading and count functions are packed in the same LUT primitive for optimal implementation.

For Up/Down counters and accumulators, XST uses the dedicated carry-ANDs to improve the performance.

Multiplexers

For multiplexers, the Macro Generator provides the following two architectures.

- MUXF*x* based multiplexers
- Dedicated Carry-MUXs based multiplexers

For Virtex-E™, MUXF*x* based multiplexers are generated by using the optimal tree structure of MUXF5, MUXF6 primitives, which allows compact implementation of large inferred multiplexers. For example, XST can implement an 8:1 multiplexer in a single CLB. In some cases dedicated carry-MUXs are generated; these can provide more efficient implementations, especially for very large multiplexers.

For Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X and Virtex-4™ devices, XST can implement a 16:1 multiplexer in a single CLB using a MUXF7 primitive, and it can implement a 32:1 multiplexer across two CLBs using a MUXF8.

To have better control of the implementation of the inferred multiplexer, XST offers a way to select the generation of either the MUXF5/MUXF6 or Dedicated Carry-MUXs architectures. The attribute MUX_STYLE specifies that an inferred multiplexer be implemented on a MUXF*x* based architecture if the value is MUXF, or a Dedicated Carry-MUXs based architecture if the value is MUXCY.

You can apply this attribute to either a signal that defines the multiplexer or the instance name of the multiplexer. This attribute can also be global.

The attribute MUX_EXTRACT with, respectively, the value *no* or *force* can be used to disable or force the inference of the multiplexer.

Priority Encoder

The if/elsif structure described in the "Priority Encoders" in Chapter 2 is implemented with a 1-of-n priority encoder.

XST uses the MUXCY primitive to chain the conditions of the priority encoder, which results in its high-speed implementation.

You can enable/disable priority encoder inference using the PRIORITY_EXTRACT constraint.

Generally, XST does not infer and so does not generate a large number of priority encoders. Therefore, Xilinx recommends that you use the PRIORITY_EXTRACT constraint with the *force* option if you would like to use priority encoders.

Decoder

A decoder is a demultiplexer whose inputs are all constant with distinct one-hot (or one-cold) coded values. An n-bit or 1-of-m decoder is mainly characterized by an m-bit data output and an n-bit selection input, such that $n^{**}(2-1) < m <= n^{**}2$.

Once XST has inferred the decoder, the implementation uses the MUXF5 or MUXCY primitive depending on the size of the decoder.

You can enable/disable decoder inference using the DECODER_EXTRACT property.

Shift Register

Two types of shift register are built by XST:

- Serial shift register with single output.
- Parallel shift register with multiple outputs.

The length of the shift register can vary from 1 bit to 16 bits as determined from the following formula:

Width = (8*A3)+(4*A2)+(2*A1)+A0+1

If A3, A2, A1 and A0 are all zeros (0000), the shift register is one-bit long. If they are all ones (1111), it is 16-bits long.

For serial shift register SRL16, flip-flops are chained to the appropriate width.

For a parallel shift register, each output provides a width of a given shift register. For each width a serial shift register is built, it drives one output, and the input of the next shift register.

You can enable/disable shift register inference using the SHREG_EXTRACT constraint.

RAMs

Two types of RAM are available in the inference and generation stages: distributed and block RAMs.

- If the RAM is asynchronous READ, Distributed RAM is inferred and generated.
- If the RAM is synchronous READ, block RAM is inferred. In this case, XST can implement block RAM or distributed RAM. The default is block RAM.

For Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X, Virtex-4™, Spartan-II™, Spartan-IIE™ and Spartan-3™ devices, XST uses the following primitives.

- RAM16X1S and RAM32X1S for Single-Port Synchronous Distributed RAM
- RAM16X1D primitives for Dual-Port Synchronous Distributed RAM

For Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X, Virtex-4™ and Spartan-3™ devices, XST uses the following primitives.

- For Single-Port Synchronous Distributed RAM:
    - For Distributed Single-Port RAM with *positive* clock edge:

      RAM16X1S, RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S,RAM64X2S, RAM128X1S,

    - For Distributed Single-Port RAM with *negative* clock edge:

      RAM16X1S_1, RAM16X2S_1, RAM16X4S_1, RAM16X8S_1, RAM32X1S_1, RAM32X2S_1, RAM32X4S_1, RAM32X8S_1, RAM64X1S_1,RAM64X2S_1, RAM128X1S_1,

- For Dual-Port Synchronous Distributed RAM:
    - For Distributed Dual-Port RAM with *positive* clock edge:

      RAM16X1D, RAM32X1D, RAM64X1D

    - For Distributed Dual-Port RAM with *negative* clock edge:

      RAM16X1D_1, RAM32X1D_1, RAM64X1D_1

For block RAM XST uses:

- RAMB4_Sn primitives for Single-Port Synchronous Block RAM
- RAMB4_Sm_Sn primitives for Dual-Port Synchronous Block RAM

In order to have better control of the implementation of the inferred RAM, XST offers a way to control RAM inference, and to select the generation of distributed RAM or block RAMs (if possible).

The RAM_STYLE attribute specifies that an inferred RAM be generated using:

- Block RAM if the value is *block*.
- Distributed RAM if the value is *distributed.*

You can apply the RAM_STYLE attribute either to a signal that defines the RAM or the instance name of the RAM. This attribute can also be global.

If the RAM resources are limited, XST can generate additional RAMs using registers. To do this use the RAM_EXTRACT attribute with the value set to *no*.

ROMs

A ROM can be inferred when all assigned contexts in a Case or If...else statement are constants. Macro inference only considers ROMs of at least 16 words with no width restriction. For example, the following HDL equation can be implemented with a ROM of 16 words of 4 bits.

```
data = if address = 0000 then 0010
       if address = 0001 then 1100
       if address = 0010 then 1011
       ...
       if address = 1111 then 0001
```

A ROM can also be inferred from an array composed entirely of constants, as in the following HDL example.

```
type ROM_TYPE is array(15 downto 0)of std_logic_vector(3 downto 0);
constant ROM : rom_type := ("0010", "1100", "1011", ..., "0001");
...
data <= ROM(conv_integer(address));
```

The ROM_EXTRACT attribute can be used to disable the inference of ROMs. Use the value *yes* to enable ROM inference, and *no* to disable ROM inference. The default is *yes.*

Two types of ROM are available in the inference and generation stages: Distributed ROM and Block ROM.

- Distributed ROMs are generated by using the optimal tree structure of LUT, MUXF5, MUXF6, MUXF7 and MUXF8 primitives which allows compact implementation of large inferred ROMs.
- Block ROMs are generated by using block RAM resources. When a synchronous ROM is identified, it can be inferred either as a distributed ROM plus a register, or it can be inferred using block RAM resources.

The ROM_STYLE attribute specifies what kind of synchronous ROM XST infers as follows.

- If set to *block*, and the ROM fits entirely on a single block of RAM, XST infers the ROM using block RAM resources.
- If set to *distributed*, XST infers a distributed ROM plus register.

- If set to *auto*, XST determines the most efficient method to use and infers the ROM accordingly. *Auto* is the default.

You can apply ROM_STYLE as a VHDL attribute or a Verilog meta comment to an individual signal, or to the entity/module of the ROM. This attribute can also be applied globally from the Process Properties dialog box in Project Navigator, or from the command line.

Using DSP48 Block Resources

XST can automatically implement several macros on a DSP48 block. Supported macros are the following:

- adders/subtractors
- accumulators
- multipliers
- multiply adder/subtractors
- multiply accumulate (MAC)

XST supports the registered versions of these macros as well.

Macro implementation on DSP48 blocks is controlled by the USE_DSP48 constraint/command line option with a default value of *auto*.

In *auto* mode, XST attempts to implement accumulators, multipliers, multiply adder/subtractors and MACs on DSP48 resources. XST does not implement adders/subtractors on DSP48 resources in *auto* mode. To push adder/subtractors into a DSP48, set the USE_DSP48 constraint/command line option value to *yes*.

XST performs automatic resource control in *auto* mode for multipliers and accumulators, using the following general method:

- First, XST implements all possible multiply adder/subtractors and MACs on DSP48 without using DSP48 resource control.
- Next, XST uses the remainder of DSP48 resources to implement multipliers and accumulators. The biggest are processed first, and the smaller ones later.

To deliver the best performance, XST by default tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. If you want to shape a macro in a specific way, you must use the KEEP constraint. For example, if your design has a multiplier with 2 register levels on each input, and you would like to exclude the first register stage from the DSP48, you must place KEEP constraints on the outputs of these registers.

Please refer to Chapter 2, "HDL Coding Techniques" for detailed information on individual macro processing.

If your design contains several interconnected macros, where each macro can be implemented on DSP48, XST attempts to interconnect DSP48 blocks using fast BCIN/BCOUT and PCIN/PCOUT connections. Such situations are typical in filter and complex multiplier descriptions.

Mapping Logic onto Block RAM

If there are unused block RAM resources and your design does not fit into your target device, you can place some of your design logic into block RAM. To do this, you must decide what part of the HDL design is to be placed in block RAM and put this part of the RTL description in a separate hierarchical block. Attach a BRAM_MAP constraint to this separate block either directly in HDL code or via the XCF file.

Please note that in the current release XST cannot automatically decide what logic could be placed in block RAM.

When placing logic into a separate block it must satisfy the following criteria.

- All outputs must be registered.
- The block may contain only one level of registers, which are output registers.
- All output registers must have the same control signals.
- The output registers must have a Synchronous Reset signal.
- The block cannot contain multisources or tristate busses.
- The KEEP attribute is not allowed on intermediate signals.

XST attempts to map the logic onto block RAM during the Advanced Synthesis step. If any of the listed requirements are not satisfied, XST does not map the logic onto block RAM, and generates a warning message with the reason for the warning. If the logic cannot be placed in a single block RAM primitive, XST spreads it over several block RAMs.

The following example places 8-bit adders with constant in a single block RAM primitive:

VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity logic_bram_1 is
  port (
    clk, rst : in  std_logic;
    A,B       : in  unsigned (3 downto 0);
    RES       : out unsigned (3 downto 0));
  attribute bram_map: string;
  attribute bram_map of logic_bram_1: entity is "yes";
end logic_bram_1;

architecture beh of logic_bram_1 is
  begin
    process (clk)
      begin
        if (clk'event and clk='1') then
          if (rst = '1') then
            RES <= "0000";
          else
            RES <= A + B + "0001";
          end if;
        end if;
    end process;
end beh;
```

VERILOG

```
module vlogic_bram_1 (clk, rst, A, B, RES);
  input  clk, rst;
  input  [3:0] A, B;
  output [3:0] RES;
  reg    [3:0] RES;

// synthesis attribute bram_map of vlogic_bram_1 is yes

  always @(posedge clk)
    begin
      if (rst)
        RES = 4'b0000;
      else
        RES = A + B + 8'b0001;
    end

endmodule
```

LOG

.

```
...
=====================================================================
*                       HDL Synthesis                             *
=====================================================================

Synthesizing Unit <logic_bram_1>.
    Related source file is bram_map_1.vhd.
    Found 4-bit register for signal <RES>.
    Found 4-bit adder for signal <$n0001> created at line 21.
    Summary:
 inferred   4 D-type flip-flop(s).
 inferred   1 Adder/Subtracter(s).
Unit <logic_bram_1> synthesized.


=====================================================================
*                     Advanced HDL Synthesis                      *
=====================================================================
...
Entity <logic_bram_1> mapped on BRAM.
...

Device utilization summary:
---------------------------

Selected Device : 2v40cs144-6

 Number of bonded IOBs:                13  out of    88    14%
 Number of BRAMs:                       1  out of     4    25%
 Number of GCLKs:                       1  out of    16     6%
...
```

In the following example, an asynchronous reset is used instead of a synchronous one and so, the logic is not mapped onto block RAM:

VHDL

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity no_logic_bram is
  port (
    clk, rst : in  std_logic;
    A,B      : in  unsigned (3 downto 0);
    RES      : out unsigned (3 downto 0));
  attribute bram_map : string;
  attribute bram_map of no_logic_bram : entity is "yes";
end no_logic_bram;
```

```
architecture beh of no_logic_bram is
  begin
    process (clk, rst)
      begin
        if (rst='1') then
          RES <= "0000";
        elsif (clk'event and clk='1') then
          RES <= A + B + "0001";
        end if;
      end process;
end beh;
```

VERILOG

```
module no_vlogic_bram (clk, rst, A, B, RES);
  input  clk, rst;
  input  [3:0] A, B;
  output [3:0] RES;
  reg    [3:0] RES;

// synthesis attribute bram_map of no_vlogic_bram is yes

  always @(posedge clk or posedge rst)
    begin
      if (rst)
        RES <= 4'b0000;
      else
        RES <= A + B + 8'b0001;
    end
endmodule
```

LOG

.

```
...
====================================================================
*                      Advanced HDL Synthesis                      *
====================================================================
...
INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.
               Output FF <RES> must have a synchronous reset.
```

Flip-Flop Retiming

Flip-flop Retiming is a technique that consists of moving flip-flops and latches across logic for the purpose of improving timing, thus increasing clock frequency. Flip-flop retiming can be either forward or backward. Forward retiming moves a set of flip-flops that are the input of a LUT to a single flip-flop at its output. Backward retiming moves a flip-flop that is at the output of a LUT to a set of flip-flops at its input. Flip-flop retiming can significantly increase the number of flip-flops in the design, and it may remove some flip-flops.

Nevertheless, the behavior of the designs remains the same. Only timing delays are modified.

Flip-flop Retiming is part of global optimization, and it respects the same constraints as all the other optimization techniques. Retiming is an iterative process, therefore a flip-flop that is the result of a retiming can be moved again in the same direction (forward or backward) if it results in better timing. The only limit for the retiming is when the timing constraints are satisfied, or if no more improvements in timing can be obtained.

For each flip-flop moved, a message is printed specifying the original and new flip-flop names, and if it is a forward or backward retiming.

Note the following limitations.

- Flip-flop retiming is not applied to flip-flops that have the IOB=TRUE property.
- Flip-flops are not moved forward if the flip-flop or the output signal has the KEEP property.
- Flip-flops are not moved backward if the input signal has the KEEP property.
- Instantiated flip-flops are not moved.
- Flip-flops with both a set and a reset are not moved.

Flip-flop retiming can be controlled by applying the REGISTER_BALANCING, MOVE_FIRST_STAGE and MOVE_LAST_STAGE constraints.

Incremental Synthesis Flow

The main goal of Incremental Synthesis flow is to reduce the overall time that the designer spends in completing a project. This can be achieved by allowing you to re-synthesize only the modified portions of the design instead of the entire design. We may consider two main categories of incremental synthesis:

- Block Level: The synthesis tool re-synthesizes the entire block if at least one modification was made inside this block.
- Gate or LUT Level: The synthesis tool tries to identify the exact changes made in the design and generates the final netlist with minimal changes.

XST supports block level incremental synthesis with some limitations.

Incremental Synthesis is implemented using two constraints: INCREMENTAL_SYNTHESIS and RESYNTHESIZE.

INCREMENTAL_SYNTHESIS:

Use the INCREMENTAL_SYNTHESIS constraint to control the decomposition of the design on several logic groups.

- If this constraint is applied to a specific block, this block with all its descendents is considered as one logic group, until the next INCREMENTAL_SYNTHESIS constraint is found. During synthesis, XST generates a single NGC file for the logic group.
- Beginning in release 7.1i, you can apply the INCREMENTAL_SYNTHESIS constraint to a block that is instantiated a multiple number of times.
- If a single block is changed then the entire logic group is resynthesized and a new NGC file(s) is generated.
- Beginning in release 7.1i, XST can propagate INCREMENTAL_SYNTHESIS constraints to the NGC netlist in the form of rangeless area groups. This enables

automatic floorplanning in the implementation flow. You can do this by setting the Enable Auto Floorplanning option to *incremental design* under the Synthesis Options tab of the Process Properties dialog box. Please note that by default XST does not propagate INCREMENTAL_SYNTHESIS constraints to the final netlist.

Example

Figure 3-1 shows how blocks are grouped by use of the INCREMENTAL_SYNTHESIS constraint. Consider the following:

- LEVA, LEVA_1, LEVA_2, my_add, my_sub as one logic group.
- LEVB, my_and, my_or and my_sub as another logic group.
- TOP is considered separately as a single logic group.

X9858
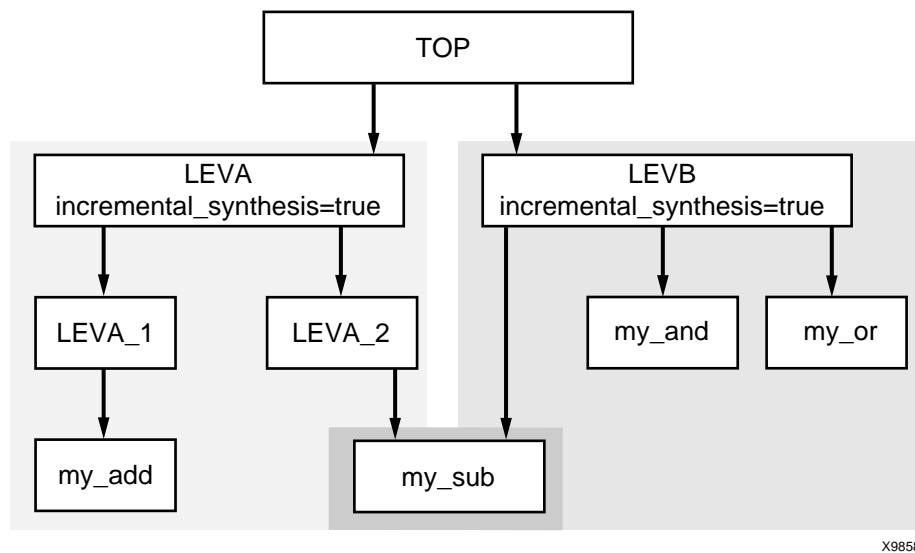
Figure 3-1: **Grouping through Incremental Synthesis**

RESYNTHESIZE

XST is able to automatically recognize what blocks were changed and to resynthesize only changed ones. This detection is done at the file level. This means that if an HDL file contains two blocks, both blocks are considered modified. If these two blocks belong to the same logic group then there is no impact on the overall synthesis time. If the HDL file contains two blocks that belong to different logic groups, both logic groups are considered changed and so are resynthesized. Xilinx recommends that you only keep different blocks in the a single HDL file if they belong to the same logic group.

Use the RESYNTHESIZE constraint to force resynthesis of the blocks that were not changed.

Note: In the current release, XST runs HDL synthesis on the entire design. However, during low level optimization XST re-optimizes modified blocks only.

In this example, XST generates three NGC files as shown in the following log file segment:.

```
...
=================================================================
*
*                          Final Report
*
=================================================================


 Final Results
 Top Level Output File Name        : c:\users\incr_synt\new.ngc
 Output File Name                  : c:\users\incr_synt\leva.ngc
 Output File Name                  : c:\users\incr_synt\levb.ngc


=================================================================
 ...
```

If you made changes to block "LEVA_1", XST automatically resynthesizes the entire logic group, including LEVA, LEVA_1, LEVA_2 and my_add, my_sub as shown in the following log file segment.

```
...
=================================================================
*
*                       Low Level Synthesis
*
=================================================================


 Final Results
 Incremental synthesis         Unit <my_and> is up to date ...
 Incremental synthesis         Unit <my_and> is up to date ...
 Incremental synthesis         Unit <my_and> is up to date ...
 Incremental synthesis         Unit <my_and> is up to date ...

Optimizing unit <my_sub> ...
Optimizing unit <my_add> ...
Optimizing unit <leva_1> ...
Optimizing unit <leva_2> ...
Optimizing unit <leva> ...


=================================================================
 ...
```

If you make no changes to the design, during Low Level synthesis, XST reports that all blocks are up to date and the previously generated NGC files are kept unchanged, as shown in the following log file segment.

```
   ...
   ================================================================
   *
   *                        Low Level Synthesis
   *
   ================================================================


    Incremental synthesis: Unit <my_and> is up to date ...
    Incremental synthesis: Unit <my_or> is up to date ...
    Incremental synthesis: Unit <my_sub> is up to date ...
    Incremental synthesis: Unit <my_add> is up to date ...
    Incremental synthesis: Unit <levb> is up to date ...
    Incremental synthesis: Unit <leva_1> is up to date ...
    Incremental synthesis: Unit <leva_2> is up to date ...
    Incremental synthesis: Unit <leva> is up to date ...
    Incremental synthesis: Unit <top> is up to date ...


   ================================================================
    ...
```

If you changed one timing constraint, then XST cannot detect this modification. To force XST to resynthesize the required blocks, use the RESYNTHESIZE constraint. For example, if "LEVA" must be resynthesized, then apply the RESYNTHESIZE constraint to this block. All blocks included in the <leva> logic group are re-optimized and new NGC files are generated as shown in the following log file segment.

```
   ...
   ================================================================
   *
   *                        Low Level Synthesis
   *
   ================================================================


    Incremental synthesis: Unit <my_and> is up to date ...
    Incremental synthesis: Unit <my_or> is up to date ...
    Incremental synthesis: Unit <levb> is up to date ...
    Incremental synthesis: Unit <top> is up to date ...
   ...
    Optimizing unit <my_sub> ...
    Optimizing unit <my_add> ...
    Optimizing unit <leva_1> ...
    Optimizing unit <leva_2> ...
    Optimizing unit <leva> ...


   ================================================================
    ...
```

If you have previously run XST in non-incremental mode and then switched to incremental mode, or the decomposition of the design has changed, you must delete all previously generated NGC files before continuing. Otherwise XST issues an error.

In the previous example, adding "incremental_synthesis=true" to the block LEVA_1, XST gives the following error:

```
ERROR:Xst:624 - Could not find instance <inst_leva_1> of cell <leva_1>
in <leva>
```

The problem most likely occurred because the design was previously run in non-incremental synthesis mode. To fix the problem, remove the existing NGC files from the project directory.

Please note that if you modified the HDL in the top level block of the design, and at the same time changed the name of top level block, XST cannot detect design modifications and resynthesize the top-level block. Force resynthesis by using the RESYNTHESIZE constraint.

Speed Optimization Under Area Constraint

XST performs timing optimization under area constraint. This option, "Slice Utilization Ratio," is available under the XST Synthesis Options in the Process Properties dialog box in Project Navigator. By default this constraint is set to 100% of the selected device size.

This constraint has influence at low level synthesis only (it does not control the inference process). If this constraint is specified, XST makes an area estimation, and if the specified constraint is met, XST continues timing optimization trying not to exceed the constraint. If the size of the design is more than requested, then XST tries to reduce the area first and if the area constraint is met, then starts timing optimization. In the following example the area constraint was specified as 100% and initial estimation shows that in fact it occupies 102% of the selected device. XST starts optimization and reaches 95%.

```
    ...
   ================================================================
   *
   *                      Low Level Synthesis
   *
   ================================================================


    Found area constraint ratio of 100 (+ 5) on block tge,
     actual ratio is 102.
    Optimizing block <tge> to meet ratio 100 (+ 5) of 1536 slices :
    Area constraint is met for block <tge>, final ratio is 95.


   ==============================================================
    ...
```

If the area constraint cannot be met, then XST ignores it during timing optimization and runs low level synthesis in order to reach the best frequency. In the following example, the

target area constraint is set to 70%. XST was not able to satisfy it and so gives the corresponding warning message.

```
...
===================================================================
*
*                      Low Level Synthesis
*
===================================================================


 Found area constraint ratio of 70 (+ 5) on block fpga_hm, actual
  ratio is 64.
 Optimizing block <fpga_hm> to meet ratio 70 (+ 5) of 1536 slices :
 WARNING:Xst - Area constraint could not be met for block <tge>, final
ratio is 94
...



===================================================================
...
```

Note: "(+5)" stands for the max margin of the area constraint. This means that if the area constraint is not met, but the difference between the requested area and obtained area during area optimization is less or equal then 5%, then XST runs timing optimization taking into account the achieved area, not exceeding it.

In the following example the area was specified as 55%. XST achieved only 60%. But taking into account that the difference between requested and achieved area is not more than 5%, XST considers that the area constraint was met.

```
...
===================================================================
*
*                      Low Level Synthesis
*
===================================================================


 Found area constraint ratio of 55 (+ 5) on block fpga_hm, actual
  ratio is 64.
 Optimizing block <fpga_hm> to meet ratio 55 (+ 5) of 1536 slices :
 Area constraint is met for block <fpga_hm>, final ratio is 60.



===================================================================
...
```

The SLICE_UTILIZATION_RATIO constraint can be attached to a specific block of a design Please refer to the *Constraint Guide* for more information.

Log File Analysis

The XST log file related to FPGA optimization contains the following sections.

- Design optimization
- Resource usage report
- Timing report

Design Optimization

During design optimization, XST reports the following.

- Potential removal of equivalent flip-flops

  Two flip-flops (latches) are equivalent when they have the same data and control pins.

- Register replication

  Register replication is performed either for timing performance improvement or for satisfying MAX_FANOUT constraints. Register replication can be turned off using the REGISTER_DUPLICATION constraint.

Following is a portion of the log file.

```
Starting low level synthesis ...
Optimizing unit <down4cnt> ...
Optimizing unit <doc_readwrite> ...
 ...
Optimizing unit <doc> ...
Building and optimizing final netlist ...
Register doc_readwrite_state_D2 equivalent to doc_readwrite_cnt_ld has
been removed
Register I_cci_i2c_wr_l equivalent to wr_l has been removed
Register doc_reset_I_reset_out has been replicated 2 time(s)
Register wr_l has been replicated 2 time(s)
```

Resource Usage

In the Final Report, the Cell Usage section reports the count of all the primitives used in the design. These primitives are classified in the following groups:

- BELS

  This group contains all the logical cells that are basic elements of the Virtex™ technology, for example, LUTs, MUXCY, MUXF5, MUXF6, MUXF7, MUXF8.

- Flip-flops and Latches

  This group contains all the flip-flops and latches that are primitives of the Virtex™ technology, for example, FDR, FDRE, LD.

- RAMS

  This group contains all the RAMs.

- SHIFTERS

  This group contains all the shift registers that use the Virtex™ primitives. They are SRL16, SRL16_1, SRL16E, SRL16E_1, and SRLC*.

- Tristates

  This group contains all the tristate primitives, namely the BUFT.

- Clock Buffers

    This group contains all the clock buffers, namely BUFG, BUFGP, BUFGDLL.

- IO Buffers

    This group contains all the standard I/O buffers, except the clock buffer, namely IBUF, OBUF, IOBUF, OBUFT, IBUF_GTL ...

- LOGICAL

    This group contains all the logical cells primitives that are not basic elements, namely AND2, OR2, ...

- OTHER

    This group contains all the cells that have not been classified in the previous groups.

The following section is an example of an XST report for cell usage:

```
==================================================
...
Cell Usage :
# BELS                             : 70
#        LUT2                      : 34
#        LUT3                      : 3
#        LUT4                      : 34
# FlipFlops/Latches                : 9
#        FDC                       : 8
#        FDP                       : 1
# Clock Buffers                    : 1
#        BUFGP                     : 1
# IO Buffers                       : 24
#        IBUF                      : 16
#        OBUF                      : 8
==================================================
```

Device Utilization summary

Where XST estimates the number of slices, gives the number of flip-flops, IOBs, BRAMS, etc. This report is very close to the one produced by MAP.

Clock Information

A short table gives information about the number of clocks in the design, how each clock is buffered and how many loads it has.

Timing Report

At the end of the synthesis, XST reports the timing information for the design. The report shows the information for all four possible domains of a netlist: "register to register", "input to register", "register to outpad" and "inpad to outpad".

The following is an example of a timing report section in the XST log:

```
NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
----------------------------------+-----------------------+-------+
Clock Signal                      | Clock buffer(FF name) | Load  |
----------------------------------+-----------------------+-------+
clk                               | BUFGP                 | 9     |
----------------------------------+-----------------------+-------+

Timing Summary:
---------------
Speed Grade: -6

   Minimum period: 7.523ns (Maximum Frequency: 132.926MHz)
   Minimum input arrival time before clock: 8.945ns
   Maximum output required time after clock: 14.220ns
   Maximum combinational path delay: 10.889ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)
========================================================================
Timing constraint: Default period analysis for Clock 'clk'
  Clock period: 7.523ns (frequency: 132.926MHz)
  Total number of paths / destination ports: 63 / 22
------------------------------------------------------------------------
Delay:              7.523ns (Levels of Logic = 2)
  Source:           sdstate_FFD1
  Destination:      sdstate_FFD2
  Source Clock:     clk rising
  Destination Clock: clk rising

  Data Path: sdstate_FFD1 to sdstate_FFD2
                        Gate   Net
  Cell:in->out   fanout Delay  Delay    Logical Name(NetName)
  ---------------------------------   ----------------------
  FDC:C->Q         15   1.372  2.970    state_FFD1 (state_FFD1)
  LUT3:I1->O        1   0.738  1.26     LUT_54 (N39)
  LUT3:I1->O        1   0.738  0.00     I_next_state_2 (N39)
  FDC:D                 0.440           state_FFD2
  ---------------------------------
  Total                 7.523ns (3.288ns logic, 4.235ns route)
                               (43.7% logic, 56.3% route)
                        Gate   Net
  Cell:in->out    fanout  Delay  Delay  Logical Name (Net Name)
  ---------------------------------   ----------------------
  FDC:C->Q         15   1.372  2.970    I_state_2
  begin scope: 'block1'
  LUT3:I1->O        1   0.738  1.265    LUT_54
  end scope: 'block1'
  LUT3:I0->O        1   0.738  0.000    I_next_state_2
  FDC:D                 0.440           I_state_2
  ---------------------------------
  Total                 7.523ns
```

Timing Summary

The Timing Summary section gives a summary of the timing paths for all 4 domains:

- The path from any clock to any clock in the design:

      Minimum period: 7.523ns (Maximum Frequency: 132.926MHz)

- The maximum path from all primary inputs to the sequential elements:

      Minimum input arrival time before clock: 8.945ns

- The maximum path from the sequential elements to all primary outputs:

      Maximum output required time before clock: 14.220ns

- The maximum path from inputs to outputs:

      Maximum combinational path delay: 10.899ns

If there is no path in the domain concerned, "No path found" is then printed instead of the value.

Timing Detail

The Timing Detail section describes the most critical path in detail for each region:

The start point and end point of the path, the maximum delay of this path, and the slack. The start and end points can be: **Clock** (with the phase: rising/falling) or **Port**:

    Path from Clock 'sysclk' rising to Clock 'sysclk' rising : 7.523ns
    (Slack: -7.523ns)

The detailed path shows the cell type, the input and output of this gate, the fanout at the output, the gate delay, the net delay estimated and the name of the instance. When entering a hierarchical block, **begin scope** is printed, and similarly **end scope** is printed when exiting a block.

The preceding report corresponds to the following schematic:
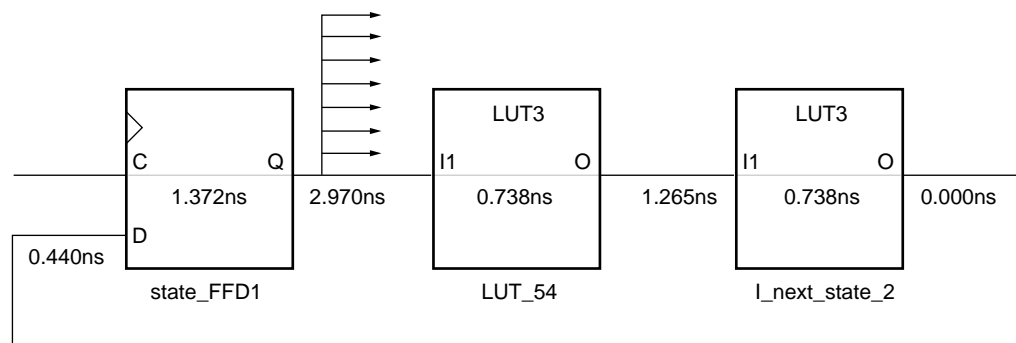
X9554

In addition, the Timing Report section shows the number of analyzed paths and ports. If XST is run with timing constraints, it displays the number of failed paths and ports as well. The number of analyzed and failed paths shows you how many timing problems there are in the design. The number of analyzed and failed ports may show you how they are spread in the design. The number of ports in a timing report represent the number of destination elements for a timing constraint.

For example, if you use the following timing constraints:

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group" value units;
```

then the number of ports corresponds to the number of elements in the destination group.

For a given timing constraint, XST may report that the number of failed paths is 100. But the number of failed destination ports is only two flip-flops. This means that it is sufficient to only analyze the design description for these two flip-flops to detect what should be changed in order to meet timing.

Implementation Constraints

XST writes all implementation constraints generated from HDL or constraint file attributes (LOC, ...) into the output NGC file.

KEEP properties are generated by the buffer insertion process (for maximum fanout control or for optimization purposes).

Virtex™ Primitive Support

XST enables you to instantiate Virtex™ primitives directly in your VHDL/Verilog code. Virtex™ primitives such as MUXCY_L, LUT4_L, CLKDLL, RAMB4_S1_S16, IBUFG_PCI33_5, and NAND3b2 can be manually inserted in your HDL design through instantiation. These primitives are not by default optimized by XST and are available in the final NGC file. Use the Optimize Instantiated Primitives synthesis option to optimize instantiated primitives and obtain better results. Timing information is available for most of the primitives, allowing XST to perform efficient timing-driven optimization.

Some of these primitives can be generated through attributes.

- BUFFER_TYPE (CLOCK_BUFFER) can be assigned to the primary input or internal signal to force the use of BUFGDLL, IBUFG, BUFR or BUFGP. The same constraints can be used to disable buffer insertion.

- IOSTANDARD can be used to assign an I/O standard to an I/O primitive. For example:

  ```
  // synthesis attribute IOSTANDARD of in1 is PCI33_5
  ```

  assigns PCI33_5 I/O standard to the I/O port.

The primitive support is based on the notion of the black box. Refer to "Safe FSM Implementation" in Chapter 2 for the basics of the black box support.

There is a significant difference between black box and primitive support. Assume you have a design with a submodule called MUXF5. In general, the MUXF5 can be your own functional block or a Virtex™ primitive. So, to avoid confusion about how XST interprets this module, use a special constraint, called BOX_TYPE. This attribute must be attached to the component declaration of MUXF5.

- If the BOX_TYPE attribute is attached to the MUXF5 with a value of:
  - *primitive*, or *black_box*, XST tries to interpret this module as a Virtex™ primitive and use its parameters, for instance, in critical path estimation.

♦ *user_black_box*, XST processes it as a regular user black box. If the name of the user black box is the same as that of a Virtex™ primitive, XST renames it to a unique name and generates a warning message with the reason for the warning. For example, MUX5 could be renamed to MUX51 as in the following log sample:

```
...
================================================================
*                       Low Level Synthesis                    *
================================================================

WARNING:Xst:79 - Model 'muxf5' has different characteristics in
destination library
WARNING:Xst:80 - Model name has been changed to 'muxf51'
...
```

- If the BOX_TYPE attribute is not attached to the MUXF5. Then XST processes this block as a user hierarchical block. If the name of the user black box is the same as that of a Virtex™ primitive, XST renames it to a unique name and then generates a warning message with the reason for the warning.

To simplify the instantiation process, XST comes with VHDL and Verilog Virtex™ libraries. These libraries contain the complete set of Virtex™ primitives declarations with a BOX_TYPE constraint attached to each component. If you use:

- VHDL—You must declare library "unisim" with its package "vcomponents" in your source code.

```
library unisim;
use unisim.vcomponents.all;
```

The source code of this package can be found in the `vhdl\src\unisims_vcomp.vhd` file of the XST installation.

- Verilog— Starting in release 6.1i, the "unisim" library is already precompiled and XST automatically links it with your design.

Some primitives, like LUT1, enable you to use an INIT during instantiation. There are two ways to pass an INIT to the final netlist.

- Attach an INIT attribute to the instantiated primitive.
- Pass the INIT via the generics mechanism in VHDL, or the parameters mechanism in Verilog. Xilinx recommends this method, as it allows you to use the same code for synthesis and simulation.

VHDL Code

Following is the VHDL code for passing an INIT value via the INIT constraint.

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity lut2_attribute is
  port(
    I0,I1 : in std_logic;
    O : out std_logic);
end lut2_attribute;
```

```
architecture beh of lut2_attribute is
  attribute INIT: string;
  attribute INIT of inst: label is "1";
  begin
    inst: LUT2 port map (I0=>I0,I1=>I1,O=>O);
end beh;
```

Following is the VHDL code for passing an INIT value via the generics mechanism.

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity lut2_generics is
  port(
    I0,I1 : in std_logic;
    O : out std_logic);
end lut2_generics;

architecture beh of lut2_generics is
  begin
    inst: LUT2 generic map (INIT=>"1")
    port map (I0=>I0,I1=>I1,O=>O);
end beh;
```

Verilog Code

Following is the Verilog code for passing an INIT value via the INIT constraint.

```
module vlut2_attribute (I0,I1,O);
  input I0,I1;
  output O;
  LUT2 inst (.I0(I0), .I1(I1), .O(O));
// synthesis attribute INIT of inst is "2"

endmodule
```

Following is the Verilog code for passing an INIT value via the parameters mechanism.

```
module vlut2_parameter (I0,I1,O);
  input I0,I1;
  output O;

  LUT2 #(4'h2) inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

Following is the Verilog code for passing an INIT value via the defparam mechanism.

```
module vlut2_defparam (I0,I1,O);
  input I0,I1;
  output O;

  LUT2 inst (.I0(I0), .I1(I1), .O(O));
defparam inst.INIT = 4'h2;

endmodule
```

Log File

XST does not issue any message concerning instantiation of Virtex™ primitives during HDL synthesis because the BOX_TYPE attribute with its value, *primitive*, is attached to each primitive in the UNISIM library.

Please note that if you instantiate a block (non primitive) in your design and the block has no contents (no logic description) or the block has a logic description, but you attach a BOX_TYPE constraint to it with a value of *user_black_box*, XST issues a warning message as in the following log file sample:

```
...
Analyzing Entity <black_b> (Architecture <archi>).
WARNING : (VHDL_0103). c:\jm\des.vhd (Line 23).
Generating a Black Box for component <my_block>.
Entity <black_b> analyzed. Unit <black_b> generated.
...
```

Related Constraints

Related constraints are BOX_TYPE and the various PAR constraints that can be passed from HDL to NGC without processing.

Cores Processing

If a design contains cores, represented by an EDIF or an NGC file, XST can automatically read them for timing estimation and area utilization control. The Read Cores option in the Synthesis Options in the Process Properties dialog box in Project Navigator allows you to enable or disable this feature. Using the read_cores option of the run command from the command line, you can also specify *optimize*. This enables cores processing, and allows XST to integrate the core netlist into the overall design.

By default, XST reads cores. In the following VHDL example, the block "my_add" is an adder, which is represented as a black box in the design whose netlist was generated by CORE Generator™.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

entity read_cores is
  port(
    A, B : in std_logic_vector (7 downto 0);
    a1, b1 : in std_logic;
    SUM : out std_logic_vector (7 downto 0);
    res : out std_logic);
end read_cores;

architecture beh of read_cores is
  component my_add
  port (
    A, B : in std_logic_vector (7 downto 0);
    S : out std_logic_vector (7 downto 0));
  end component;
```

```
begin
  res <= a1 and b1;
  inst: my_add port map (A => A, B => B, S => SUM);
end beh;
```

If Read Cores is disabled, XST estimates Maximum Combinational Path Delay as 6.639ns (critical path goes through a simple AND function) and an area of one slice.

If Read Cores is enabled then XST displays the following messages during Low Level Synthesis.

```
...
=======================================================================
*
*                      Low Level Synthesis
*
=======================================================================


Launcher: Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 6.1i - edif2ngd G.21
INFO:NgdBuild - Copyright (c) 1995-2003 Xilinx, Inc.  All rights
reserved.
Writing the design to "my_add.ngo"...
Loading core <my_add> for timing and area information for instance
<inst>.

=======================================================================
...
```

Estimation of Maximum Combinational Path Delay is 8.281ns with an area of five slices. Please note that by default, XST reads EDIF/NGC cores from the current (project) directory. If the cores are not in the project directory, you must use the Cores Search Directories synthesis option to specify which directory the cores are in.

Specifying INITs and RLOCs in HDL Code

Using the UNISIM library allows you to directly instantiate LUT components in your HDL code. To specify a function that a particular LUT must execute, apply an INIT constraint to the instance of the LUT. If you want to place an instantiated LUT or register in a particular slice of the chip, then attach an RLOC constraint to the same instance.

It is not always convenient to calculate INIT functions and different methods that can be used to achieve this. Instead, you can describe the function that you want to map onto a single LUT in your VHDL or Verilog code in a separate block. Attaching a LUT_MAP constraint (XST is able to automatically recognize the XC_MAP constraint supported by Synplicity) to this block indicates to XST that this block must be mapped on a single LUT. XST automatically calculates the INIT value for the LUT and preserves this LUT during optimization. In the following VHDL example, the "top" block contains the instantiation of two AND gates, described in "and_one" and "and_two" blocks. XST generates two LUT2s and does not merge them. Please refer to the LUT_MAP constraint description in the *Constraints Guide* for details.

```
library ieee;
use ieee.std_logic_1164.all;
entity and_one is
  port (
    A, B : in std_logic;
    REZ  : out std_logic);
  attribute LUT_MAP: string;
  attribute LUT_MAP of and_one: entity is "yes";
end and_one;

architecture beh of and_one is
begin
  REZ <= A and B;
end beh;

library ieee;
use ieee.std_logic_1164.all;
entity and_two is
  port(
    A, B : in std_logic;
    REZ  : out std_logic);
  attribute LUT_MAP: string;
  attribute LUT_MAP of and_two: entity is "yes";
end and_two;

architecture beh of and_two is
begin
  REZ <= A or B;
end beh;

library ieee;
use ieee.std_logic_1164.all;
entity top is
  port(
    A,B,C : in std_logic;
    REZ   : out std_logic);
end top;

architecture beh of top is

  component and_one
    port(
      A, B  : in std_logic;
      REZ   : out std_logic);
  end component;

  component and_two
    port(
      A, B : in std_logic;
      REZ  : out std_logic);
  end component;

  signal tmp: std_logic;
begin
  inst_and_one: and_one port map (A => A, B => B, REZ => tmp);
  inst_and_two: and_two port map (A => tmp, B => C, REZ => REZ);
end beh;
```

If a function cannot be mapped on a single LUT, XST issues an Error and interrupts the synthesis process. If you would like to define an INIT value for a flip-flop, described at RTL level, you can assign its initial value in the signal declaration stage. This value is not ignored during synthesis and is propagated to the final netlist as an INIT constraint attached to the flip-flop. This feature is supported for registers only. It is not supported for RAM descriptions. In the following VHDL example, a 4-bit register is inferred for signal "tmp". An INIT value equal "1011" is attached to the inferred register and propagated to the final netlist.

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
port (
    CLK : in std_logic;
    DO : out std_logic_vector(3 downto 0));
end test;

architecture beh of test is signal
  tmp: std_logic_vector(3 downto 0):="1011";
  begin
    process (CLK)
      begin
        if (clk'event and clk='1') then
          tmp <= DI;
        end if;
    end process;
  DO <= tmp;
end beh;
```

To infer a register in the previous example, and place it in a specific location of a chip, attach an RLOC constraint to the "tmp" signal as in the following VHDL example. XST propagates it to the final netlist. Please note that this feature is supported for registers, and is supported for inferred block RAM as well if it can be implemented on a single block RAM primitive.

```
library ieee;
use ieee.std_logic_1164.all;

entity test is
  port (
    CLK : in std_logic;
    DI : in std_logic_vector(3 downto 0);
    DO : out std_logic_vector(3 downto 0));
end test;
```

```
architecture beh of test is
  signal tmp: std_logic_vector(3 downto 0):="1011";

  attribute RLOC: string;
  attribute RLOC of tmp: signal is "X3Y0 X2Y0 X1Y0 X0Y0";

  begin
  process (CLK)
    begin
      if (clk'event and clk='1') then
        tmp <= DI;
      end if;
    end process;
    DO <= tmp;
end beh;
```

PCI Flow

To successfully use PCI flow with XST (i.e. to satisfy all placement constraints and meet timing requirements) set the following options.

- For VHDL designs, ensure that the names in the generated netlist are all in uppercase. Note that by default, the case for VHDL synthesis flow is *lower*. Specify the case by selecting the Case option under the Synthesis Options tab in the Process Properties dialog box within Project Navigator.

- For Verilog designs, ensure that Case is set to *maintain*, which is a default value. Specify Case as described above.

- Preserve the hierarchy of the design. Specify the Keep Hierarchy setting by selecting the Keep Hierarchy option under the Synthesis Options tab in the Process Properties dialog box within Project Navigator.

- Preserve equivalent flip-flops, which XST removes by default. Specify the Equivalent Register Removal setting by selecting the Equivalent Register Removal option under the Xilinx® Specific Options tab in the Process Properties dialog box within Project Navigator.

- Prevent logic and flip-flop replication caused by high fanout flip-flop set/reset signals. Do this by:

  ♦ Setting a high maximum fanout value for the entire design via the Max Fanout menu in the Synthesis Options tab in the Process Properties dialog box within Project Navigator.

    or

  ♦ Setting a high maximum fanout value for the initialization signal connected to the RST port of PCI core by using the MAX_FANOUT attribute (for example: max_fanout=2048).

- Prevent XST from automatically reading PCI cores for timing and area estimation. In reading PCI cores, XST may perform some logic optimization in the user's part of the design that does not allow the design to meet timing requirements or might even lead to errors during MAP. Disable Read Cores by unchecking the Read Cores option under the Synthesis Options tab in the Process Properties dialog box in Project Navigator.

Note: By default, XST reads cores for timing and area estimation.

Chapter 4

CPLD Optimization

This chapter contains the following sections.

- *"CPLD Synthesis Options"*
- *"Implementation Details for Macro Generation"*
- *"Log File Analysis"*
- *"Constraints"*
- *"Improving Results"*

CPLD Synthesis Options

This section describes the CPLD-supported families and their specific options.

Introduction

XST performs device specific synthesis for CoolRunner™ XPLA3/-II and XC9500™/XL/XV families, and generates an NGC file ready for the CPLD fitter.

The general flow of XST for CPLD synthesis is the following:

1. HDL synthesis of VHDL/Verilog designs
2. Macro inference
3. Module optimization
4. NGC file generation

Global CPLD Synthesis Options

This section describes supported CPLD families and lists the XST options related *only* to CPLD synthesis that can only be set from the Process Properties dialog box in Project Navigator.

Families

Five families are supported by XST for CPLD synthesis:

- CoolRunner XPLA3™
- CoolRunner™ -II
- XC9500™
- XC9500XL™
- XC9500XV™

The synthesis for the CoolRunner™, XC9500XL™, and XC9500XV™ families includes clock enable processing; you can allow or invalidate the clock enable signal (when invalidating, it is replaced by equivalent logic). Also, the selection of the macros which use the clock enable (counters, for instance) depends on the family type. A counter with clock enable is accepted for the CoolRunner™, XC9500XL™ and XC9500XV™ families, but rejected (replaced by equivalent logic) for XC9500™ devices.

List of Options

Following is a list of CPLD synthesis options that you can set from the Process Properties dialog box in Project Navigator. For details about each option, refer to "CPLD Constraints (non-timing)" in Chapter 5.

- Keep Hierarchy
- Macro Preserve
- XOR Preserve
- Equivalent Register Removal
- Clock Enable
- WYSIWYG
- No Reduce

Implementation Details for Macro Generation

XST processes the following macros:

- adders
- subtractors
- add/sub
- multipliers
- comparators
- multiplexers
- counters
- logical shifters
- registers (flip-flops and latches)
- XORs

The macro generation is decided by the Macro Preserve option, which can take two values:

yes — macro generation is allowed.

no — macro generation is inhibited.

The general macro generation flow is the following:

1. HDL infers macros and submits them to the low-level synthesizer.
2. Low-level synthesizer accepts or rejects the macros depending on the resources required for the macro implementations.

An accepted macro is generated by an internal macro generator. A rejected macro is replaced by equivalent logic generated by the HDL synthesizer. A rejected macro may be decomposed by the HDL synthesizer into component blocks so that one component may be a new macro requiring fewer resources than the initial one, and another smaller macro

may be accepted by XST. For instance, a flip-flop macro with clock enable (CE) cannot be accepted when mapping onto the XC9500™. In this case the HDL synthesizer submits two new macros:

- a flip-flop macro without clock enable signal.
- a MUX macro implementing the clock enable function.

A generated macro is optimized separately and then merged with surrounded logic because the optimization process gives better results for larger components.

Log File Analysis

XST messages related to CPLD synthesis are located after the following message:

```
========================================================
*                  Low Level Synthesis                 *
========================================================
```

The log file produced by XST contains:

- Tracing of progressive unit optimizations:

    ```
     Optimizing unit unit_name ...
    ```

- Information, warnings or fatal messages related to unit optimization:

    - When equation shaping is applied (XC9500™ devices only):

        ```
        Collapsing ...
        ```

    - Removing equivalent flip-flops:

        ```
        Register ff1 equivalent to ff2 has been removed
        ```

    - User constraints fulfilled by XST:

        ```
        implementation constraint: constraint_name[=value]: signal_name
        ```

- Final results statistics:

    ```
    Final Results
       Top Level Output file name : file_name
       Output format : ngc
       Optimization goal : {area | speed}
       Target Technology : {9500 | 9500xl | 9500xv | xpla3 | xbr | cr2s}
       Keep Hierarchy : {yes | soft | no}
       Macro Preserve : {yes | no}
       XOR Preserve : {yes | no}

    Design Statistics
       NGC Instances: nb_of_instances
       I/Os: nb_of_io_ports
    ```

```
Macro Statistics
  # FSMs: nb_of_FSMs
  # Registers: nb_of_registers
  # Tristates: nb_of_tristates
  # Comparators: nb_of_comparators
    n-bit comparator {equal | not equal | greater| less | greatequal
        | lessequal}:
    nb_of_n_bit_comparators
  # Multiplexers: nb_of_multiplexers
    n-bit m-to-1 multiplexer :
    nb_of_n_bit_m_to_1_multiplexers
  # Adders/Subtractors: nb_of_adds_subs
    n-bit adder: nb_of_n_bit_adds
    n-bit subtractor: nb_of_n_bit_subs
  # Multipliers: nb_of_multipliers
  # Logic Shifters: nb_of_logic_shifters
  # Counters: nb_of_counters
  n-bit {up | down | updown} counter:
  nb_of_n_bit_counters
  # XORs: nb_of_xors
Cell Usage :
  # BELS: nb_of_bels
  #       AND...: nb_of_and...
  #       OR...: nb_of_or...
  #       INV: nb_of_inv
  #       XOR2: nb_of_xor2
  #       GND: nb_of_gnd
  #       VCC: nb_of_vcc
  # FlipFlops/Latches: nb_of_ff_latch
  #       FD...: nb_of_fd...
  #       LD...: nb_of_ld...
  # Tri-States: nb_of_tristates
  #       BUFE: nb_of_bufe
  #       BUFT: nb_of_buft
  # IO Buffers: nb_of_iobuffers
  #       IBUF: nb_of_ibuf
  #       OBUF: nb_of_obuf
  #       IOBUF: nb_of_iobuf
  #       OBUFE: nb_of_obufe
  #       OBUFT: nb_of_obuft
  # Others: nb_of_others
```

Constraints

The constraints (attributes) specified in the HDL design or in the constraint files are written by XST into the NGC file as signal properties.

Improving Results

XST produces optimized netlists for the CPLD fitter, which fits them in specified devices and creates the download programmable files. The CPLD low-level optimization of XST consists of logic minimization, subfunction collapsing, logic factorization, and logic decomposition. The result of the optimization process is an NGC netlist corresponding to Boolean equations, which are reassembled by the CPLD fitter to fit the best of the macrocell capacities. A special XST optimization process, known as equation shaping, is applied for XC9500™/XL/XV devices when the following options are selected:

- Keep Hierarchy: **No**
- Optimization Effort: **2** or **High**
- Macro Preserve: **No**

The equation shaping processing also includes a critical path optimization algorithm, which tries to reduce the number of levels of critical paths.

The CPLD fitter multi-level optimization is still recommended because of the special optimizations done by the fitter (D to T flip-flop conversion, De Morgan Boolean expression selection).

How to Obtain Better Frequency?

The frequency depends on the number of logic levels (logic depth). In order to reduce the number of levels, the following options are recommended.

- Optimization Effort: **2** or **High** — this value implies the calling of the collapsing algorithm, which tries to reduce the number of levels without increasing the complexity beyond certain limits.
- Optimization Goal: **Speed** — the priority is the reduction of number of levels.

The following tries, in this order, may give successively better results for frequency:

Try 1: Select only optimization effort 2 and speed optimization. The other options have default values:

- Optimization effort: **2** or **High**
- Optimization Goal: **Speed**

Try 2: Flatten the user hierarchy. In this case the optimization process has a global view of the design, and the depth reduction may be better:

- Optimization effort: **1**/**Normal** or **2**/**High**
- Optimization Goal: **Speed**
- Keep Hierarchy: **no**

Try 3: Merge the macros with surrounded logic. The design flattening is increased:

- Optimization effort: `1` or **Normal**
- Optimization Goal: **Speed**
- Keep Hierarchy: **no**
- Macro Preserve **no**

Try 4: Apply the equation shaping algorithm. Options to be selected:

- Optimization effort: `2` or **High**
- Macro Preserve: **no**
- Keep Hierarchy: **no**

The CPU time increases from Try 1 to Try 4.

Obtaining the best frequency depends on the CPLD fitter optimization. Xilinx recommends running the multi-level optimization of the CPLD fitter with different values for the –pterms options, starting with 20 and finishing with 50 with a step of 5. Statistically the value 30 gives the best results for frequency.

How to Fit a Large Design?

If a design does not fit in the selected device, exceeding the number of device macrocells or device P-Term capacity, you must select an area optimization for XST. Statistically, the best area results are obtained with the following options:

- Optimization effort: `1`∕**Normal** or `2`∕**High**
- Optimization Goal: **Area**
- Default values for other options

Another option that you can try is "–wysiwyg yes". This option may be useful when the design cannot be simplified by the optimization process and the complexity (in number of P-Terms) is near the device capacity. It may be that the optimization process, trying to reduce the number of levels, creates larger equations, therefore increasing the number of P-Terms and so preventing the design from fitting. By validating this option, the number of P-Terms is not increased, and the design fitting may be successful.

XILINX®

Chapter 5

Design Constraints

This chapter describes constraints, options, and attributes supported for use with XST.

This chapter contains the following sections.

Introduction

Constraints are essential to help you meet your design goals or obtain the best implementation of your circuit. Constraints are available in XST to control various aspects of the synthesis process itself, as well as placement and routing. Synthesis algorithms and heuristics have been tuned to automatically provide optimal results in most situations. In some cases, however, synthesis may fail to initially achieve optimal results; some of the available constraints allow you to explore different synthesis alternatives to meet your specific needs.

The following mechanisms are available to specify constraints.

- Options provide global control on most synthesis aspects. They can be set either from within the Process Properties dialog box in Project Navigator or by setting options of the run command from the command line.

- VHDL attributes can be directly inserted into your VHDL code and attached to individual elements of the design to control both synthesis, and placement and routing.

- Constraints can be added as Verilog meta comments in your Verilog code.

- Constraints can be specified in a separate constraint file.

Typically, global synthesis settings are defined within the Process Properties dialog box in Project Navigator or with command line arguments, while VHDL attributes or Verilog meta comments can be inserted in your source code to specify different choices for individual parts of the design. Note that the local specification of a constraint overrides its global setting. Similarly, if a constraint is set both on a node (or an instance) and on the enclosing design unit, the former takes precedence for the considered node (or instance).

Setting Global Constraints and Options

This section explains how to set global constraints and options from the Process Properties dialog box within Project Navigator.

For a description of each constraint that applies generally — that is, to FPGAs, CPLDs, VHDL, and Verilog — refer to the *Constraints Guide*.

Note: Except for the Value fields with check boxes, there is a pull-down arrow or browse button in each Value field. However, you cannot see the arrow until you click in the Value field.

Synthesis Options

To specify the HDL synthesis options from Project Navigator:

1. Select a source file from the Source file window.

2. Right-click on **Synthesize - XST** in the Process window.

3. Select **Properties**.

4. When the Process Properties dialog box displays, click the Synthesis Options tab.

Depending on the device family you have selected (FPGA or CPLD), one of two dialog boxes displays:
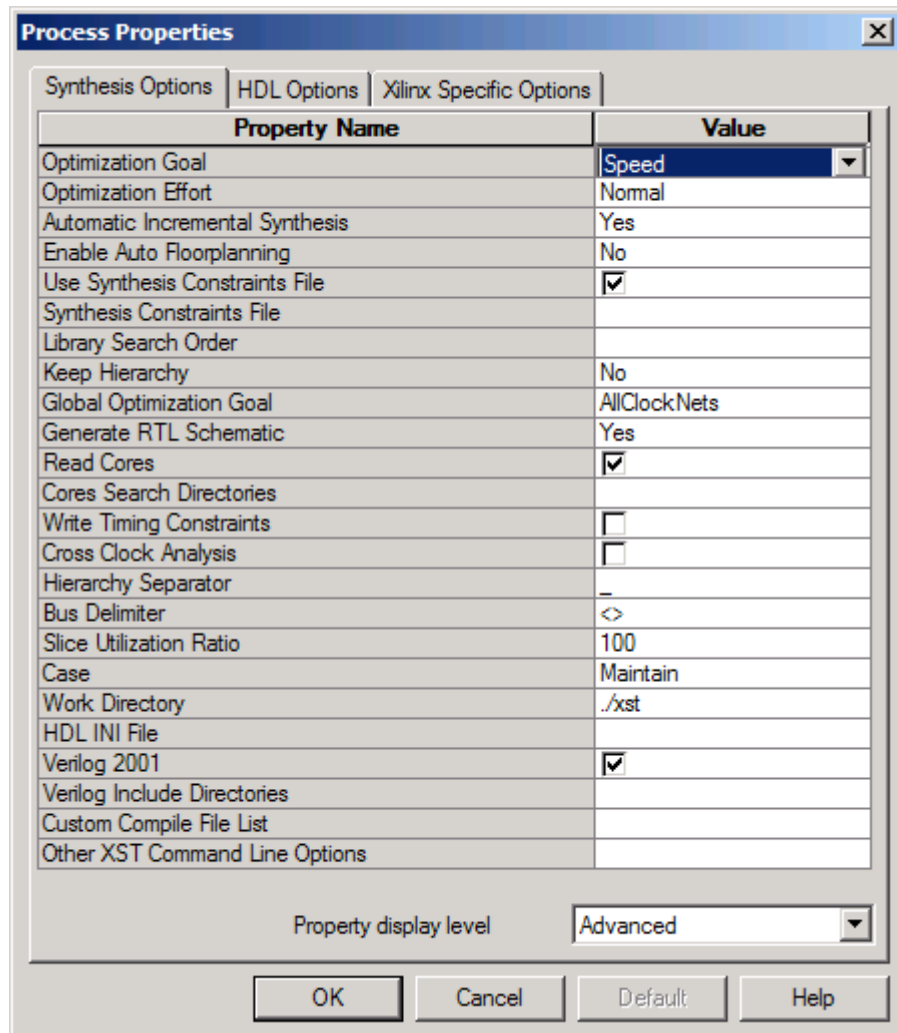
Figure 5-1: **Synthesis Options (FPGA)**
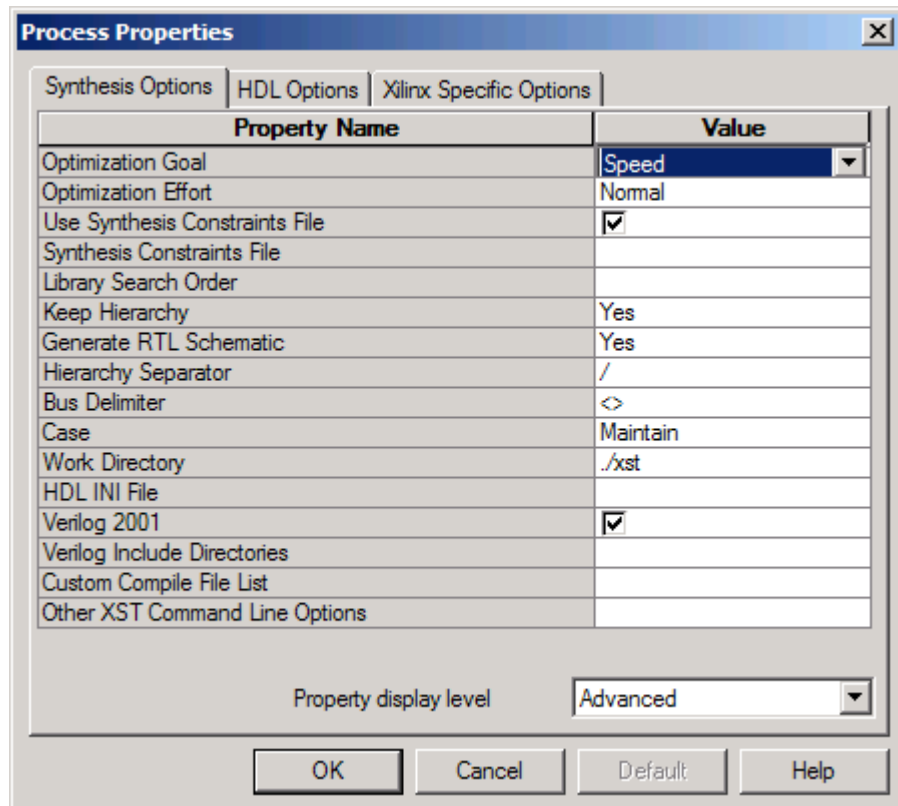
Figure 5-2: **Synthesis Options (CPLD)**

Following is a list of the synthesis options that can be selected from the dialog boxes.

- Optimization Goal
- Optimization Effort
- Enable Auto Floorplanning
- Use Synthesis Constraints File
- Synthesis Constraint File
- Library Search Order
- Keep Hierarchy*
- Global Optimization Goal
- Generate RTL Schematic
- Read Cores*
- Cores Search Directories*
- Write Timing Constraints
- Cross Clock Analysis*
- Hierarchy Separator*
- Bus Delimiter*
- Slice Utilization Ratio*
- Case*
- Work Directory*
- HDL Library Mapping File (.INI File)*
- Verilog 2001
- Verilog Include Directories (Verilog Only)*
- Custom Compile File List*
- Other XST Command Line Options*

* To view these options, go to the **Property Display Level** drop down menu at the bottom of the window, and click **Advanced**.

HDL Options

With the Process Properties dialog box displayed for the **Synthesize - XST** process, select the HDL Options tab. For FPGA device families the following dialog box displays.
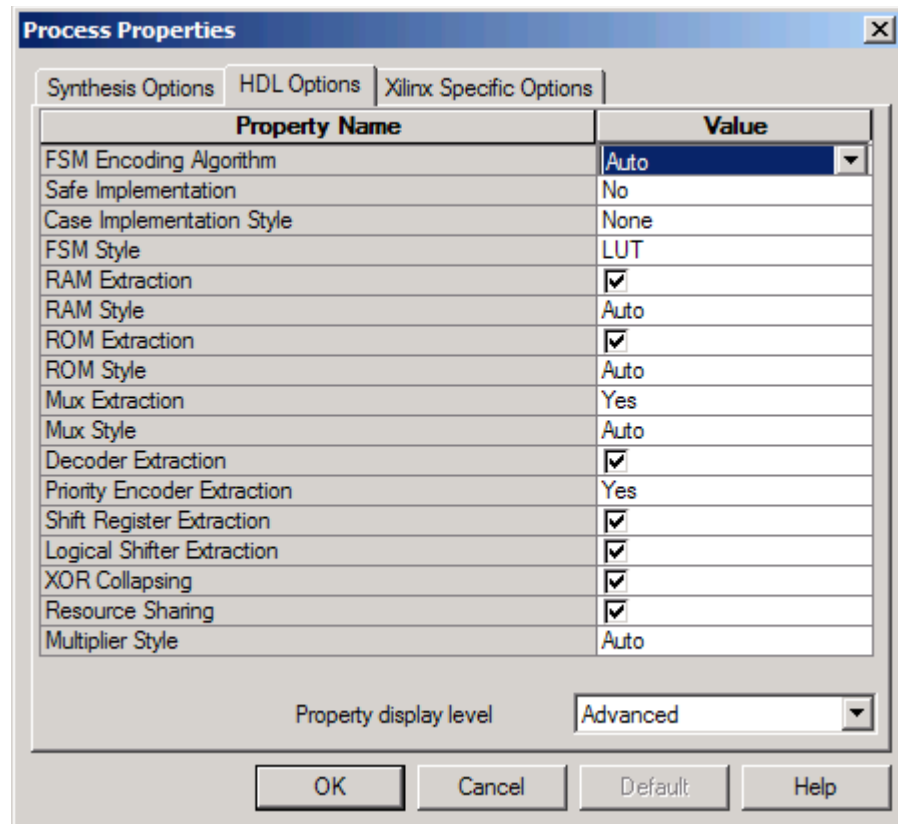
Figure 5-3: **HDL Options Tab (FPGAs)**

Following is a list of all HDL Options that can be set within the HDL Options tab of the Process Properties dialog box for FPGA devices:

- FSM Encoding Algorithm
- Safe Implementation
- Case Implementation Style
- FSM Style*
- RAM Extraction
- RAM Style
- ROM Extraction
- ROM Style
- Mux Extraction
- Mux Style
- Decoder Extraction
- Priority Encoder Extraction
- Shift Register Extraction

- *Logical Shifter Extraction*
- *XOR Collapsing*
- *Resource Sharing*
- *Multiplier Style* **
- *Use DSP48* **

* To view these options, go to the **Property Display Level** drop down menu at the bottom of the window, and click **Advanced**.

** When working with Virtex-4™ devices, Use DSP48 replaces Multiplier Style on in this dialog box.

For CPLD device families the following dialog box displays.
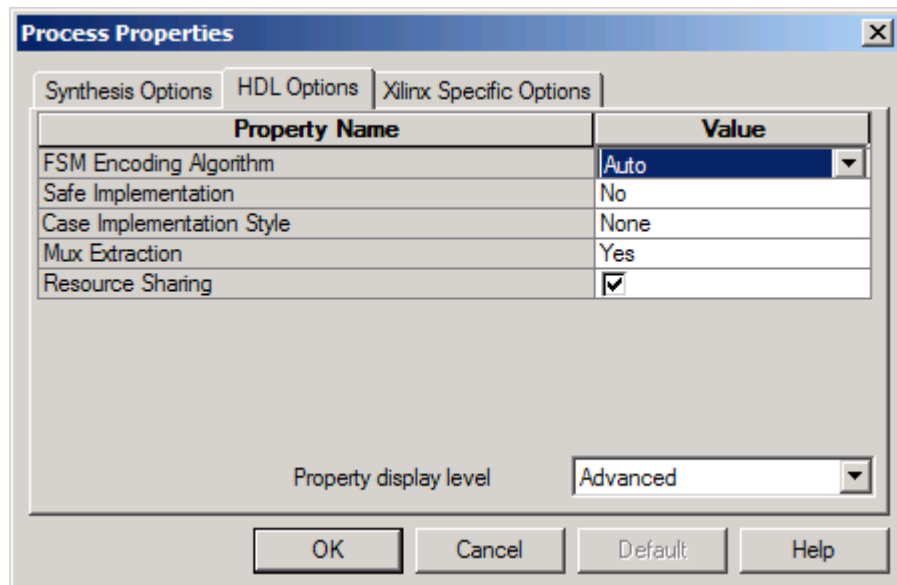
Figure 5-4: **HDL Options Tab (CPLDs)**

Following is a list of all HDL Options that can be set within the HDL Options tab of the Process Properties dialog box for CPLD devices:

- *FSM Encoding Algorithm*
- *Safe Implementation*
- *Case Implementation Style*
- *Mux Extraction*
- *Resource Sharing*

Xilinx Specific Options

From the Process Properties dialog box for the **Synthesize - XST** process, select the Xilinx Specific Options tab to display the options.

For FPGA device families, the following dialog box displays:
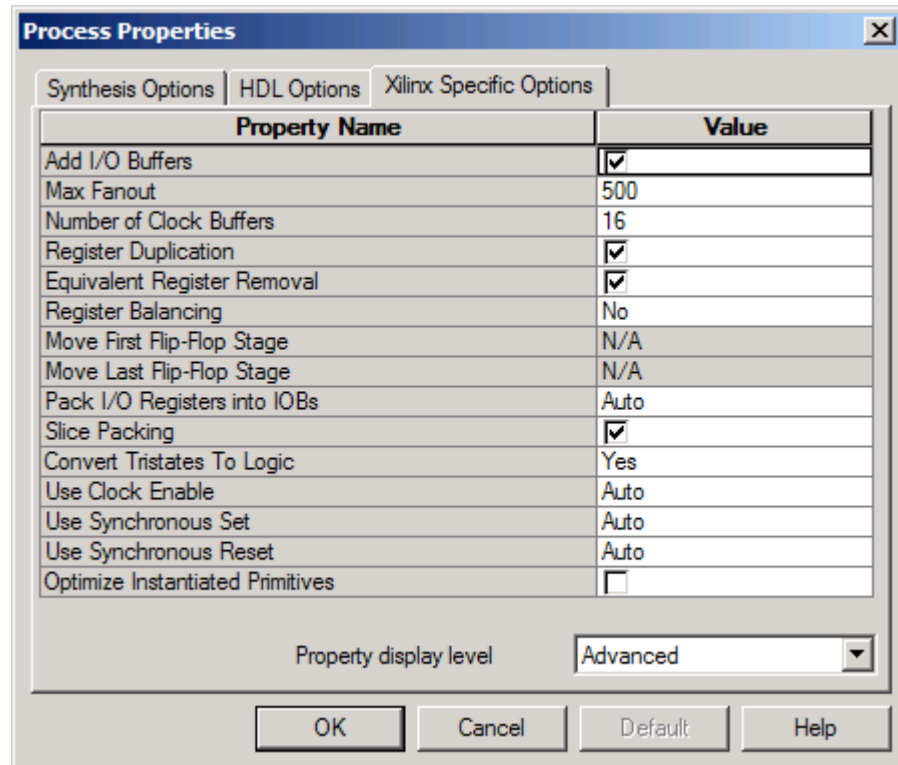
Figure 5-5: **Xilinx Specific Options (FPGAs)**

Following is the list of the Xilinx Specific Options for FPGAs:

- Add I/O Buffers
- Max Fanout
- Number of Global Clock Buffers*
- Number of Regional Clock Buffers*
- Register Duplication
- Equivalent Register Removal
- Register Balancing
- Move First Stage
- Move Last Stage
- Convert Tristates to Logic**
- Use Clock Enable
- Use Synchronous Set
- Use Synchronous Reset
- Optimize Instantiated Primitives

* To view these options, go the **Property Display Level** drop down menu at the bottom of the window, and click **Advanced**.

** Convert Tristate to Logic only appears when working with applicable devices.

For CPLD device families the following dialog box displays.
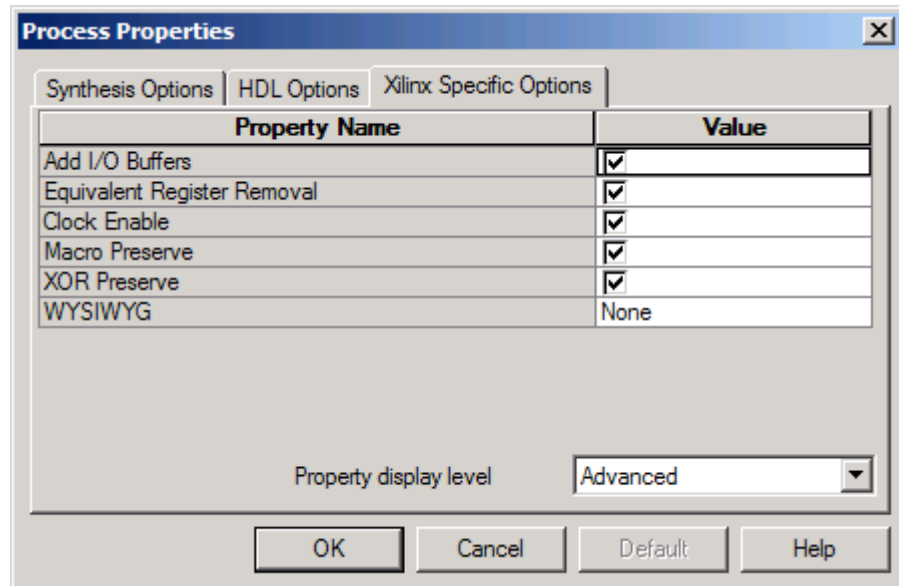
Figure 5-6: **Xilinx Specific Options (CPLDs)**

Following is a list of the Xilinx Specific Options:

- Add I/O Buffers
- Equivalent Register Removal
- Clock Enable
- Macro Preserve
- XOR Preserve
- WYSIWYG

Other XST Command Line Options

Any XST command line option can be set via the Other XST Command Line Options property in the Process Properties dialog box. This is an advanced property. Use the syntax described in Chapter 10, "Command Line Mode." Separate multiple options with a space.

While the Other XST Command Line Options property is intended for XST options not listed in the Process Properties dialog box, if an option already listed as a dialog box property is entered, precedence is given to the option entered here. Illegal or unrecognized options cause XST to stop processing and generate a message like the following one.

```
ERROR:Xst:1363 - Option "-verilog2002" is not available for command
run.
```

Custom Compile File List

By using the Custom Compile File List property, you can change the order in which source files are processed by XST. With this property, you select a user-defined compile list file that XST uses to determine the order in which it processes libraries and design files. Otherwise, XST uses an automatically generated list.

This user-defined file must list all design files and their libraries in the order in which they are to be compiled, from top to bottom. Type each file/library pair on its own line, with a semicolon separating the library from the file. The format is as follows:

```
library_name;file_name
[library_name;file_name]
...
```

Following is an example:

```
work;stopwatch.vhd
work;statmach.vhd
...
```

Note: This property is not connected to the Custom Compile File List property in the Simulation Properties dialog box, which means that a different compile list file is used for synthesis than for simulation.

VHDL Attribute Syntax

You can describe constraints with VHDL attributes in your VHDL code. Before it can be used, an attribute must be declared with the following syntax.

```
attribute AttributeName : Type ;
```

Example:

```
attribute RLOC : string  ;
```

The attribute type defines the type of the attribute value. The only allowed type for XST is **string**. An attribute can be declared in an entity or architecture. If declared in the entity, it is visible both in the entity and the architecture body. If the attribute is declared in the architecture, it cannot be used in the entity declaration. Once declared a VHDL attribute can be specified as follows:

```
attribute AttributeName of ObjectList : ObjectType is AttributeValue ;
```

Examples:

```
attribute RLOC of u123 : label is R11C1.S0 ;

attribute bufg of my_signal : signal is sr;
```

The object list is a comma separated list of identifiers. Accepted object types are entity, component, label, signal, variable and type.

Verilog Meta Comment Syntax

Constraints can be specified as follows in Verilog code:

```
// synthesis attribute AttributeName [of] ObjectName [is]
    AttributeValue
```

Example:

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
// synthesis attribute bufg of my_clock is "clk";
```

Note: The parallel_case, full_case, translate_on and translate_off directives follow a different syntax described in "Verilog Meta Comments" in Chapter 7.

Verilog-2001 Attributes

XST supports Verilog-2001 attribute statements. Attributes are comments that are used to pass specific information to software tools such as synthesis tools. Verilog-2001 attributes can be specified anywhere for operators or signals within module declarations and instantiations, and signal declarations.

Note: Other attribute declarations may be supported by the compiler, but are ignored by XST.

Attributes can be used to:

- Set constraints on individual objects (for example, module, instance, net).
- Set FULL_CASE and PARALLEL_CASE synthesis directives.

Syntax

Attributes must be bounded by the characters **(*** and ***)**, and are written using the following syntax:

```
(* attribute_name = attribute_value *)
```

Where:

- The attribute must precede the signal, module or instance declaration it refers to.
- The *attribute_value* must be a string; no integer or scalar values are allowed.
- The *attribute_value* must be between quotes.
- The default value is 1. **(* attribute_name *)** is the same as **(* attribute_name = "1" *)**.

Example 1

```
(* clock_buffer = "IBUFG" *) input CLK;
```

Example 2

```
(* INIT = "0000" *) reg [3:0] d_out;
```

Example 3

```
always@(current_state or reset)
  begin (* parallel_case *) (* full_case *)
    case (current_state)
      ...
```

Example 4

```
(* mult_style = "pipe_lut" *) MULT my_mult (a, b, c);
```

Limitations

Verilog-2001 attributes are not supported for the following.

- signal declarations
- statements
- port connections
- expression operators

XST Constraint File (XCF)

XST constraints can be specified in a file called the Xilinx Constraint File (XCF). The XCF must have an extension of .xcf. You can specify the constraint file in ISE™, by going to the **Synthesis - XST** Process Properties, clicking the Synthesis Options tab, enabling the Use Synthesis Constraints File option by clicking the check box, clicking the value field for the **Synthesis Constraints File** option, and typing the constraint file name. You can also browse for an existing file to use by clicking the box to the right of the value field. Also, to quickly enable/disable the use of a constraint file by XST, you can check or uncheck the Use Synthesis Constraint File option in this same menu. By selecting this option, you invoke the –iuc command line switch.

To specify the constraint file in command line mode, use the –uc switch with the *run* command. See Chapter 10, "Command Line Mode" for details on the *run* command and running XST from the command line.

XCF Syntax and Utilization

The XCF syntax enables you to specify a specific constraint for the entire device (globally) or for specific modules in your design. The XCF syntax is basically the same as the UCF syntax for applying constraints to nets or instances, but with an extension to the syntax to allow constraints to be applied to specific levels of hierarchy. You can use the keyword MODEL to define the entity/module that the constraint is applied to. If a constraint is applied to an entity/module, the constraint is applied to each instance of the entity/module.

In general, users should define constraints within the ISE™ process properties dialog box (or the XST run script, if running on the command line), then use the XCF file to specify exceptions to these general constraints. The constraints specified in the XCF file are applied ONLY to the module listed, and not to any submodules below it.

To apply a constraint to the entire entity/module use the following syntax:

```
MODEL entityname constraintname = constraintvalue;
```

Examples:

```
MODEL top mux_extract = false;
MODEL my_design max_fanout = 256;
```

Note: If the entity `my_design` is instantiated several times in the design, the `max_fanout=256` constraint is applied to each instance of `my_design`.

To apply constraints to specific instances or signals within an entity/module, use the INST or NET keywords:

```
BEGIN MODEL entityname
  INST instancename constraintname = constraintvalue ;
  NET signalname constraintname = constraintvalue ;
END;
```

Examples:

```
BEGIN MODEL crc32
  INST stopwatch opt_mode = area ;
  INST U2 ram_style = block ;
  NET myclock clock_buffer = true ;
  NET data_in iob = true ;
END;
```

See "Constraints Summary" for the complete list of synthesis constraints that you can apply for XST.

Native vs. Non-Native UCF Constraints Syntax

From a UCF syntax point of view, all constraints supported by XST can be divided into two groups: native UCF constraints, and non-native UCF constraints. Only Timing and Area Group constraints use native UCF syntax.

For all non-native UCF constraints, use the MODEL or BEGIN MODEL... END; constructs. This is true for pure XST constraints such as FSM_EXTRACT or RAM_STYLE, as well as for implementation non-timing constraints, such as RLOC or KEEP.

For native UCF constraints, such as PERIOD, OFFSET, TNM_NET, TIMEGRP, TIG, FROM-TO etc., use native UCF syntax, which includes the use of wildcards and hierarchical names. Do not use these constraints inside the BEGIN MODEL... END construct, otherwise XST issues an error.

IMPORTANT: If you specify timing constraints in the XCF file, Xilinx strongly suggests that you use '/' character as a hierarchy separator instead of '_'. Please refer to "Hierarchy Separator" for details on its usage.

Limitations

XCF syntax has the following limitations.

- Nested model statements are not supported in the current release.

- Instance or signal names listed between the BEGIN MODEL statement and the END statement are only the ones visible inside the entity. Hierarchical instance or signal names are not supported.

- Wildcards in instance and signal names are not supported, except in timing constraints.

- Not all native UCF constraints are supported in the current release. Refer to the *Constraints Guide* for more information.

General Constraints

This section lists various constraints that you can use with XST. These constraints apply to FPGAs, CPLDs, VHDL, and Verilog. You can set some of these options under the Synthesis Options tab of the Process Properties dialog box in Project Navigator. See "Constraints Summary" for a complete list of constraints supported by XST.

- **Add I/O Buffers**

  Add IO Buffers, (–iobuf) enables or disables IO buffer insertion. Allowed values are *yes, no.* By default, buffer insertion is enabled.

  XST automatically inserts Input/Output Buffers into the design. You can manually instantiate I/O Buffers for some or all the I/Os, and XST will insert I/O Buffers only for the remaining I/Os. If you do not want XST to insert any I/O Buffers, set this option to *no.* This option is useful to synthesize a part of a design to be instantiated later on.

  When the `yes` value is selected, IBUF and OBUF primitives are generated. IBUF/OBUF primitives are connected to I/O ports of the top-level module. When XST is called to synthesize an internal module which will be instantiated later in a larger design, you must select *no* for this option. If I/O buffers are added to a design, this design cannot be used as a submodule of another design.

  Define this option globally with the –iobuf command line option of the **run** command. Following is the basic syntax:

  ```
  –iobuf {yes|no}
  ```

  The default is *yes.*

  In Project Navigator, specify –iobuf globally with the Add IO Buffers option in the Xilinx Specific Option tab of the Process Properties dialog box.

- **Box Type**

  The Box Type (BOX_TYPE) constraint instructs XST not to synthesize the behavior of a model, and to use some predefined set of characteristics for that model's behavior. See "BOX_TYPE" in the *Constraints Guide* for details.

- **Bus Delimiter**

  The Bus Delimiter (–bus_delimiter) command line option defines the format used to write the signal vectors in the result netlist. The available possibilities are <>, [], {}, (). The default is <>.

Define this option globally with the –bus_delimiter command line option of the **run** command. Following is the basic syntax:

-bus_delimiter {**<>**|**[]**|**{}**|**()**}

The default is <>.

In Project Navigator, set –bus_delimiter globally with the Bus Delimiter option in the Synthesis Options tab of the Process Properties dialog box in the Project Navigator.

- **Case**

  The Case command line option (–case) determines if instance and net names are written in the final netlist using all lower or upper case letters or if the case is maintained from the source. Note that the case can be maintained for either Verilog or VHDL synthesis flow.

  Define this option globally with the –case command line option of the **run** command. Following is the basic syntax:

  -case {**upper**|**lower**|**maintain**}

  The default is *maintain*.

  In Project Navigator, specify –case globally with the Case option in the Synthesis Options tab of the Process Properties dialog box.

- **Case Implementation Style**

  The Case Implementation Style (–vlgcase) command line option instructs XST how to interpret Verilog Case statements. It has three possible values: *full*, *parallel* and *full-parallel*.

  ♦ If the option is not specified, then XST implements the exact behavior of the case statements.

  ♦ If *full* is used, XST assumes that the case statements are complete and avoids latch creation.

  ♦ If *parallel* is used, XST assumes that the branches cannot occur in parallel and does not use a priority encoder.

  See "Multiplexers" in Chapter 2 of this manual, as well as, "FULL_CASE" and "PARALLEL_CASE" in the *Constraints Guide* for details.

  Define this option globally with the –vlgcase command line option of the **run** command.

  -vlgcase {**full**|**parallel**|**full-parallel**}

  By default, there is no value.

  In Project Navigator specify –vlgcase globally with the Case Implementation Style option in the HDL Options tab of the Process Properties dialog box. Allowed values are Full, Parallel, and Full-Parallel, By default, the value is blank.

- **Full Case (Verilog)**

  The FULL_CASE directive is used to indicate that all possible selector values have been expressed in a case, casex or casez statement. The directive prevents XST from creating additional hardware for those conditions not expressed. See "Multiplexers" in Chapter 2 of this manual, and "FULL_CASE" in the *Constraints Guide* for details.

- **Generate RTL Schematic**

  The Generate RTL Schematic (–rtlview) command line option enables XST to generate a netlist file, representing an RTL structure of the design. This netlist can be viewed by the RTL and Technology Viewers. This option has three possible values: *yes*, *no* and *only*. When the *only* value is specified, XST stops the synthesis process just after the RTL view is generated. The file containing the RTL view has an NGR file extension.

  Generate RTL Schematic is defined globally with the –rtlview command line option of the **run** command. Following is the basic syntax:

      -rtlview {yes|no|only}

  - From the command line, the default is *no.*
  - From Project Navigator, the default is yes.

  In Project Navigator, specify –rtlview globally with the Generate RTL Schematic option in the Synthesis Options tab of the Process Properties dialog box.

- **Duplication Suffix**

  The Duplication Suffix (–duplication_suffix) command line option controls how XST names replicated flip-flops. By default, when XST replicates a flip-flop, it creates a name for the new flip-flop by taking the name of the original flip-flop and adding "_*n*" to the end of it, where *n* is an index number. For example, if the original flip-flop name is my_ff, and this flip-flop was replicated three times, XST generates flip-flops with the following names: my_ff_1, my_ff_2 and my_ff_3.

  Using the Duplication Suffix command line option, you can specify a text string to append to the end of the default name. You can use the %d escape character to specify where in the name the index number appears. For example, for the flip-flop named my_ff, if you specify _dupreg_%d with the Duplication Suffix option, XST generates the following names: my_ff_dupreg_1, my_ff_dupreg_2, and my_ff_dupreg_3. Please note that %d can be placed anywhere in the suffix definition. For example, if the Duplication Suffix value is specified as _dup_%d_reg, XST generates the following names: my_ff_dup_1_reg, my_ff_dup_2_reg and my_ff_dup_3_reg.

  Define this option globally with the –duplication_suffix command line option of the **run** command.

      -duplication_suffix string%dstring

  The default is _%d.

  Note: The –duplication_suffix option is not listed in the ISE™ Synthesis Properties dialog box. You must specify it in the Other XST Command Line Options menu item.

- **Hierarchy Separator**

  The Hierarchy Separator (–hierarchy_separator) command line option defines the hierarchy separator character that is used in name generation when the design hierarchy is flattened.

  There are two supported characters, '_' and '/'. The default is '/' for newly created projects.

  If a design contains a sub-block with instance INST1, and this sub-block contains a net, called TMP_NET, then the hierarchy is flattened and the hierarchy separator character is '_'. The name of TMP_NET becomes INST1_TMP_NET. If the hierarchy separator character is '/', then the name of the net will be 'INST1/TMP_NET'. Using '/' as a hierarchy separator is very useful in the design debugging process because this separator makes it much easier to identify a name if it is hierarchical.

Define this option globally with the –hierarchy_separator command line option of the **run** command. Following is the basic syntax:

```
-hierarchy_separator {_|/}
```

The default is '/' for newly created projects.

In Project Navigator, specify –hierarchy_separator globally with the Hierarchy Separator option in the Synthesis Options tab of the Process Properties dialog box in the Project Navigator.

- **Iostandard**

  Use the IOSTANDARD constraint to assign an I/O standard to an I/O primitive. See "IOSTANDARD" in the *Constraints Guide* for details.

- **Keep**

  The KEEP constraint is an advanced mapping constraint. When a design is mapped, some nets may be absorbed into logic blocks. When a net is absorbed into a block, it can no longer be seen in the physical design database. This may happen, for example, if the components connected to each side of a net are mapped into the same logic block. The net may then be absorbed into the block containing the components. KEEP prevents this from happening. See "KEEP" in the *Constraints Guide* for details.

- **Library Search Order**

  The Library Search Order (–lso) command line option is related to the use of mixed language (VHDL/Verilog) projects support. It allows you to specify the order in which various library files are used. It can be invoked by specifying the file containing the search order in the value field to the right of Library Search option under the Synthesis Options tab in the Process Properties dialog box in Project Navigator, or with the –lso command line option. See the "Library Search Order File" in Chapter 8 for details.

- **LOC**

  The LOC constraint defines where a design element can be placed within an FPGA/CPLD. See "LOC" in the *Constraints Guide* for details.

- **Optimization Effort**

  The Optimization Effort (OPT_LEVEL) constraint defines the synthesis optimization effort level. See "OPT_LEVEL" in the *Constraints Guide* for details.

- **Optimization Goal**

  The Optimization Goal (OPT_MODE) constraint defines the synthesis optimization strategy. Available strategies can be *speed* or *area*. See "OPT_MODE" in the *Constraints Guide* for details.

- **Parallel Case (Verilog)**

  The PARALLEL_CASE directive is used to force a case statement to be synthesized as a parallel multiplexer and prevents the case statement from being transformed into a prioritized if/elsif cascade. See "Multiplexers" in Chapter 2 of this guide. Also see "PARALLEL_CASE" in the *Constraints Guide* for details.

- **RLOC**

  The RLOC constraint is a basic mapping and placement constraint. This constraint groups logic elements into discrete sets and allows you to define the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design. See "RLOC" in the *Constraints Guide* for details.

- **Synthesis Constraint File**

  The Synthesis Constraint File (–uc) command line option specifies a synthesis constraint file for XST to use. The XCF must have an extension of .xcf. If the extension is not .xcf, XST will error out and stop processing.

  Please refer to "XST Constraint File (XCF)" for details on using the constraint file.

  Specify a file name with the –uc command line option of the **run** command. Following is the basic syntax:

      –uc *filename*

  In Project Navigator, specify a synthesis file with the Use Synthesis Constraints File option in the Synthesis Options tab of the Process Properties dialog box.

- **Translate Off/Translate On (Verilog/VHDL)**

  The Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON) directives can be used to instruct XST to ignore portions of your VHDL or Verilog code that are not relevant for synthesis; for example, simulation code. The TRANSLATE_OFF directive marks the beginning of the section to be ignored, and the TRANSLATE_ON directive instructs XST to resume synthesis from that point. See "TRANSLATE_OFF and TRANSLATE_ON" in the *Constraints Guide* for details.

- **Use Synthesis Constraints File**

  The Use Synthesis Constraints File (–iuc) command line option allows you to ignore the constraint file during synthesis.

  Define this option globally with the –iuc command line option of the **run** command. Following is the basic syntax:

      -iuc {**yes**|**no**}

  The default is *no.*

  In Project Navigator, specify –iuc globally by selecting the Use Synthesis Constraints File option under the Synthesis Options tab in the Process Properties dialog box.

- **Verilog Include Directories (Verilog Only)**

  Use the Verilog Include Directories option (–vlgincdir) to enter discrete paths to your Verilog Include Directories.

  > Define this option globally with the –vlgincdir command line option of the **run** command. Allowed values are names of directories. Please refer to "Names with Spaces" in Chapter 10 for more information.

  >   **-vlgincdir** {*directory_path* [*directory_path*]}

  There is no default.

  In Project Navigator, specify this option with the Verilog Include Directories option of the Synthesis Options tab in the Process Properties dialog box. Allowed values are names of directories. There is no default.

- **Verilog 2001**

  The Verilog 2001(–verilog2001) command line option enables or disables interpreted Verilog source code as the Verilog 2001 standard. By default Verilog source code is interpreted as the Verilog 2001 standard.

  Define this option globally with the –verilog2001 command line option of the **run** command. Following is the basic syntax:

      -verilog2001 {**yes**|**no**}

The default is *yes*.

In Project Navigator, set –verilog2001 globally with the Verilog 2001 option in the Synthesis Options tab of the Process Properties dialog box.

• **HDL Library Mapping File (.INI File)**

Use the HDL Library Mapping File command (–xsthdpini) to define the library mapping.

There is a library mapping file and two associated parameters: XSTHDPINI and XSTHDPDIR. The library mapping file contains the library name and the directory in which this library is compiled. XST maintains two library mapping files:

♦ The "pre-installed" file, which is installed during the Xilinx® software installation.

♦ The "user" file, which users may define for their own projects.

The "pre-installed" (default) INI file is named "xhdp.ini," and is located in %XILINX%\vhdl\xst. These files contain information about the locations of the standard VHDL and UNISIM libraries. These should not be modified, but the syntax can be used for user library mapping. This file appears as follows:

```
-- Default lib mapping for XST
std=$XILINX/vhdl/xst/std
ieee=$XILINX/vhdl/xst/unisim
unisim=$XILINX/vhdl/xst/unisim
aim=$XILINX/vhdl/xst/aim
pls=$XILINX/vhdl/xst/pls
```

You may use this file format to define where each of your own libraries must be placed. By default, all compiled VHDL flies will be stored in the "xst" sub-directory of the ISE™ project directory. You may place your custom INI file anywhere on a disk by:

♦ Selecting the "VHDL INI File" menu in the "Synthesis Options" tab of the Synthesis process properties in Project Navigator.

or

♦ Setting up the –xsthdpini parameter, using the following command in stand-alone mode:

```
set -xsthdpini <file_name>
```

You can give this library mapping file any name you wish, but it is best to keep the .ini classification. The format is:

```
library_name=path_to_compiled_directory
```

Note: (Use "--" for comments.)

Sample text for "my.ini":

```
work1=H:\Users\conf\my_lib\work1
work2=C:\mylib\work2
```

- **Work Directory**

  The Work Directory (–xsthdpdir) parameter defines the location in which VHDL-compiled files must be placed if the location is not defined by library mapping files. You can access this switch by:

  - Selecting the "VHDL Working Directory" menu in the "Synthesis Options" tab of the Synthesis process properties in Project Navigator.

    or

  - Using the following command in stand-alone mode:

    ```
    set –xsthdpdir <file_name>
    ```

  Example:

  Suppose three different users are working on the same project. They must share one standard, pre-compiled library, **shlib**. This library contains specific macro blocks for their project. Each user also maintains a local work library, but User 3 places it outside the project directory (i.e., in c:\temp). Users 1 and 2 will share another library ("lib12") between them, but not with User 3. The settings required for the three users are as follows:

  User 1:

  ```
  Mapping file:
  schlib=z:\sharedlibs\shlib
  lib12=z:\userlibs\lib12
  ```

  User 2:

  ```
  Mapping file:
  schlib=z:\sharedlibs\shlib
  lib12=z:\userlibs\lib12
  ```

  User 3:

  ```
  Mapping file:
  schlib=z:\sharedlibs\shlib
  ```

  User 3 will also set:

  ```
  XSTHDPDIR = c:\temp
  ```

  Define this parameter globally with the **set** –xsthdpdir command line option before running the **run** command. Following is the basic syntax:

  ```
  set -xsthdpdir directory
  ```

  The command can accept a single path only. You must specify the directory you want to use. There is no default.

In Project Navigator, specify –xsthdpdir globally with the VHDL Work Directory option of the Synthesis Options tab in the Process Properties dialog box.

HDL Constraints

This section describes encoding and extraction constraints. Most of the constraints can be set globally in the HDL Options tab of the Process Properties dialog box in Project Navigator. The only constraints that *cannot* be set in this dialog box are Enumerated Encoding, Signal Encoding, and Recovery State. The constraints described in this section apply to FPGAs, CPLDs, VHDL, and Verilog.

Note: Please note that in many cases, a particular constraint can be applied globally to an entire entity or model, or alternatively, it can be applied locally to individual signals, nets or instances. See Table 5-1 for valid constraint targets.

- **Automatic FSM Extraction**

  The Automatic FSM Extraction (FSM_EXTRACT) constraint enables or disables finite state machine extraction and specific synthesis optimizations. This option must be enabled in order to set values for the FSM Encoding Algorithm and FSM Flip-Flop Type. See "FSM_EXTRACT" in the *Constraints Guide* for details.

- **Enumerated Encoding (VHDL)**

  The Enumerated Encoding (ENUM_ENCODING) constraint can be used to apply a specific encoding to a VHDL enumerated type. See "ENUM_ENCODING" in the *Constraints Guide* for details.

  Note: Because it must preserve the external design interface, XST ignores the ENUM_ENCODING constraint when it is used on a port.

- **Equivalent Register Removal**

  The Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) constraint enables or disables removal of equivalent registers, described at the RTL Level. By default XST does not remove equivalent flip-flops if they are instantiated from a Xilinx® primitive library. To allow optimization of instantiated flip-flops and other primitives, use the OPTIMIZE_PRIMITIVES constraint. See "EQUIVALENT_REGISTER_REMOVAL" in the *Constraints Guide* for details.

- **FSM Encoding Algorithm**

  The FSM Encoding Algorithm (FSM_ENCODING) constraint selects the finite state machine coding technique to be used. The Automatic FSM Extraction option must be enabled in order to select a value for the FSM Encoding Algorithm. See "FSM_ENCODING" in the *Constraints Guide* for details.

- **Mux Extraction**

  The Mux Extract (MUX_EXTRACT) constraint enables or disables multiplexer macro inference. For each identified multiplexer description, based on some internal decision rules, XST actually creates a macro or optimizes it with the rest of the logic. See "MUX_EXTRACT" in the *Constraints Guide* for details.

- **Register Power Up**

  XST does not automatically figure out and enforce register power-up values. You must explicitly specify them if needed with the Register Power Up (REGISTER_POWERUP) constraint. See "REGISTER_POWERUP" in the *Constraints Guide* for details.

- **Resource Sharing**

  The Resource Sharing (RESOURCE_SHARING) constraint enables or disables resource sharing of arithmetic operators. See the "RESOURCE_SHARING" in the *Constraints Guide* for details.

- **Recovery State**

  The Recovery State (SAFE_RECOVERY_STATE) constraint defines a recovery state for use when a finite state machine (FSM) is implemented in Safe Implementation mode. This means that if during its work, the FSM gets into an invalid state, XST generates additional logic to force the FSM to a valid recovery state.

  See "SAFE_RECOVERY_STATE" and "SAFE_IMPLEMENTATION" in the *Constraints Guide* for details.

- **Safe Implementation**

  The Safe Implementation (SAFE_IMPLEMENTATION) constraint implements finite state machines (FSMs) in Safe Implementation mode. Safe Implementation means that XST generates additional logic that forces an FSM to a valid state (recovery state) if an FSM gets into an invalid state. By default, XST automatically selects *reset* as the recovery state. If the FSM does not have an initialization signal, XST selects *power-up* as the recovery state. The recovery state can be manually defined via the RECOVERY_STATE constraint.

  ♦ To activate Safe FSM implementation from Project Navigator, select the Safe Implementation option from the HDL Options tab of the Synthesis Process Properties dialog box in Project Navigator.

  ♦ To activate Safe FSM implementation from your HDL code, apply the SAFE_IMPLEMENTATION constraint to the hierarchical block or signal that represents the state register in the FSM.

  See "SAFE_IMPLEMENTATION" and "SAFE_RECOVERY_STATE" in the *Constraints Guide* for details.

- **Signal Encoding**

  The Signal Encoding (SIGNAL_ENCODING) constraint can be used to apply a specific encoding to signals. See "SIGNAL_ENCODING" in the *Constraints Guide* for details.

FPGA Constraints (non-timing)

This section describes FPGA HDL options. These options apply only to FPGAs—not CPLDs.

Note: Please note that in many cases, a particular constraint can be applied globally to an entire entity or model, or alternatively, it can be applied locally to individual signals, nets or instances. See Table 5-1 for valid constraint targets.

- **Buffer Type**

  Buffer Type (BUFFER_TYPE) is a new name for the CLOCK_BUFFER constraint. Since CLOCK_BUFFER will become obsolete in future releases, Xilinx strongly suggest that you use this new name. This constraint selects the type of buffer to be inserted. See "BUFFER_TYPE" in the *Constraints Guide* for details.

- **BUFGCE**

  The BUFGCE constraint implements BUFGMUX functionality by inferring a BUFGMUX primitive. This operation reduces the wiring: clock and clock enable signals are driven to *n* sequential components by a single wire. See "BUFGCE" in the *Constraints Guide* for details.

- **Clock Buffer Type**

  The Clock Buffer Type (CLOCK_BUFFER) constraint selects the type of buffer to be inserted on the net. In the current release, this constraint has been renamed BUFFER_TYPE. Since CLOCK_BUFFER will become obsolete in future releases, Xilinx strongly suggest that you use this new name. See "BUFFER_TYPE" in the *Constraints Guide* for details.

- **Cores Search Directories**

  The Cores Search Directories command line switch (–sd) tells XST to look for cores in directories other than the default one (by default XST searches for cores in the directory specified in the –ifn switch).

  Define this option globally with the –sd command line option of the **run** command. Allowed values are names of directories Please refer to "Names with Spaces" in Chapter 10 for more information.

  ```
  -sd {directory_path [directory_path]}
  ```

  There is no default.

  In Project Navigator, specify this option with the Cores Search Directories option of the Synthesis Options tab in the Process Properties dialog box. Allowed values are names of directories. There is no default.

- **Decoder Extraction**

  The Decoder Extraction (DECODER_EXTRACT) constraint enables or disables decoder macro inference. See "DECODER_EXTRACT" in the *Constraints Guide* for details.

- **Enable Auto Floorplanning**

  The Enable Floorplanning (–enable_auto_floorplanning) command line option instructs XST to propagate incremental synthesis constraints placed in the design HDL to the NGC file in the form of rangeless Area Groups where they can be used by the Floorplanner.

  Define this option globally with the –enable_auto_floorplanning command line option of the **run** command. Following is the basic syntax:

  ```
  -enable_auto_floorplanning {no|incremental_design}
  ```

  The default is *no.*

  In Project Navigator, set this option globally with the Enable Auto Floorplanning option in the Synthesis Options tab of the Process Properties dialog box.

  Refer to "Incremental Synthesis Flow" in Chapter 3 for more information.

- **FSM Style**

  The FSM Style constraint (FSM_STYLE) can be used to make large FSMs more compact and faster by implementing them in the block RAM resources provided in Virtex™ and later technologies. You can direct XST to use block RAM resources rather than LUTs (the default) to implement FSMs by using the FSM_STYLE design constraint. See "FSM_STYLE" in the *Constraints Guide* for details.

- **Resynthesize**

  The RESYNTHESIZE constraint is related to Incremental Synthesis Flow. It forces or prevents resynthesis of groups created via the INCREMENTAL_SYNTHESIS constraint. See "RESYNTHESIZE" and "INCREMENTAL_SYNTHESIS" in the *Constraints Guide* for details.

- **Incremental Synthesis**

    The Incremental Synthesis (INCREMENTAL_SYNTHESIS) constraint controls the decomposition of a design into several subgroups. This can be applied on a VHDL entity or Verilog module so that XST generates a single and separate NGC file for it and its descendents. See the "Incremental Synthesis Flow" in Chapter 3, as well as "INCREMENTAL_SYNTHESIS" in the *Constraints Guide* for details.

    Note: The INCREMENTAL_SYNTHESIS switch is not accessible via the Synthesize - XST Process Properties dialog box. This directive is only available via VHDL attributes or Verilog meta comments, or via an XST constraint file.

- **Keep Hierarchy**

    XST may automatically flatten the design to get better results by optimizing entity/module boundaries. You can use the Keep Hierarchy (KEEP_HIERARCHY) constraint to preserve the hierarchy of your design. In addition, this constraint may be propagated to the NGC file as an implementation constraint depending on its value.

    See "KEEP_HIERARCHY" in the *Constraints Guide* for details.

- **Logical Shifter Extraction**

    The Logical Shifter Extraction (SHIFT_EXTRACT) constraint enables or disables logical shifter macro inference. See "SHIFT_EXTRACT" in the *Constraints Guide* for details.

- **Map Logic on BRAM**

    The Map Logic on BRAM (BRAM_MAP) constraint is used to map an entire hierarchical block on the block RAM resources available in Virtex™ and later technologies. See "Mapping Logic onto Block RAM" in Chapter 3, and also "BRAM_MAP" in the *Constraints Guide* for details.

- **Max Fanout**

    The Max Fanout (MAX_FANOUT) constraint limits the fanout of nets or signals. See "MAX_FANOUT" in the *Constraints Guide* for details.

- **Move Last Stage**

    The Move Last Stage (MOVE_LAST_STAGE) constraint controls the retiming of registers with paths going to primary outputs. See "MOVE_LAST_STAGE" in the *Constraints Guide* for details.

- **Move First Stage**

    The Move First Stage (MOVE_FIRST_STAGE) constraint controls the retiming of registers with paths coming from primary inputs. See "MOVE_FIRST_STAGE" in the *Constraints Guide* for details.

- **Multiplier Style**

    The Multiplier Style (MULT_STYLE) constraint controls the way the macro generator implements the multiplier macros. Allowed values are *auto, block, lut, pipe_lut, kcm, pipe_block and csd.* The default is *auto,* meaning that XST looks for the best implementation for each considered macro. The *pipe_lut* option is for pipeline multipliers. The implementation style can be manually forced to use block multiplier or LUT resources available in the Spartan-3™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X and Virtex-4™ devices. Please note that *pipe_block* is supported for Virtex-4 devices only. See "MULT_STYLE" in the *Constraints Guide* for details.

- **Mux Style**

  The Mux Style (MUX_STYLE) constraint controls the way the macrogenerator implements the multiplexer macros. See "MUX_STYLE" in the *Constraints Guide* for details.

- **Number of Global Clock Buffers**

  The Number of Global Clock Buffers (–bufg) constraint controls the maximum number of BUFGs created by XST. The constraint value is an integer. The default value depends on the target family and is equal to the maximum number of available BUFGs.

  Define this option globally with the **–bufg** command line option of the **run** command. Following is the basic syntax:

      –bufg *integer*

  The constraint value is an *integer* and the default values are different for different architectures. The defaults for selected architectures are: 4 for Virtex™, Virtex-E™, Spartan-II™, Spartan-IIE™; 8 for Spartan-3™; 16 for Virtex-II™, Virtex-II Pro™, Virtex-II Pro X™; and 32 for Virtex-4™. The number of BUFGs cannot exceed the maximum number of BUFGs for the target device.

  In Project Navigator, specify –bufg globally by selecting the Number of Clock Buffers option under the Xilinx Specific Options tab in the Process Properties dialog box.

- **Number of Regional Clock Buffers**

  The Number of Regional Clock Buffers (–bufr) constraint controls the maximum number of BUFRs created by XST. The constraint value is an integer. The default value depends on the target family and is equal to the maximum number of available BUFRs.

  Define this option globally with the –bufr command line option of the run command.

  Following is the basic syntax:

      –**bufr** *integer*

  This constraint is available for Virtex-4™ devices only. The constraint value is an integer and the default value is different for each Virtex-4™ device.

  The number of BUFRs cannot exceed the maximum number of BUFRs for the target device.

  In Project Navigator, specify –bufr globally by selecting the Number of Regional Clock Buffers option under the Xilinx Specific Options tab in the Process Properties dialog box.

- **Optimize Instantiated Primitives**

  By default, XST does not optimize instantiated primitives in HDL code. The Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) constraint is used to deactivate the default. See "OPTIMIZE_PRIMITIVES" in the *Constraints Guide* for details.

- **Pack I/O Registers into IOBs**

  The Pack I/O Registers into IOBs (IOB) constraint packs flip-flops in the I/Os to improve input/output path timing. See "IOB" in the *Constraints Guide* for details.

- **Priority Encoder Extraction**

  The Priority Encoder Extraction (PRIORITY_EXTRACT) constraint enables or disables priority encoder macro inference. See "PRIORITY_EXTRACT" in the *Constraints Guide* for details.

- **RAM Extraction**

    The RAM Extraction (RAM_EXTRACT) constraint enables or disables RAM macro inference. See "RAM_EXTRACT" in the *Constraints Guide* for details.

- **RAM Style**

    The RAM Style (RAM_STYLE) constraint controls whether the macrogenerator implements the inferred RAM macros as block or distributed RAM. See "RAM_STYLE" in the *Constraints Guide* for details.

- **Register Balancing**

    The Register Balancing (REGISTER_BALANCING) constraint enables flip-flop retiming. See "REGISTER_BALANCING" in the *Constraints Guide* for details.

- **Register Duplication**

    The Register Duplication (REGISTER_DUPLICATION) constraint enables or disables register replication. See "REGISTER_DUPLICATION" in the *Constraints Guide* for details.

    Note: Register Duplication can be applied to modules/entities as well as nets/signals.

- **ROM Extraction**

    The ROM Extraction (ROM_EXTRACT) constraint enables or disables ROM macro inference. See "ROM_EXTRACT" in the *Constraints Guide* for details.

- **ROM Style**

    ROM Style (ROM_STYLE) controls the way the macrogenerator implements the inferred ROM macros. The implementation style can be manually forced to use block ROM or distributed ROM resources available in Virtex™ series, Spartan-II™ and Spartan-3™ devices. See "ROM_STYLE" in the *Constraints Guide* for details.

- **Shift Register Extraction**

    The Shift Register Extraction (SHREG_EXTRACT) constraint enables or disables shift register macro inference. See "SHREG_EXTRACT" in the *Constraints Guide* for details.

- **Slice Packing**

    The Slice Packing (–slice_packing) option enables the XST internal packer. The packer attempts to pack critical LUT-to-LUT connections within a slice or a CLB. This exploits the fast feedback connections among the LUTs in a CLB.

    Define this option globally with the –slice_packing command line option of the **run** command. Following is the basic syntax:

    ```
    -slice_packing {yes|no}
    ```

    The default is *yes.*

    In Project Navigator, set –slice_packing globally with the Slice Packing option in the Xilinx Specific Options tab in the Process Properties dialog box.

- **Uselowskewlines**

    The USELOWSKEWLINES constraint is a basic routing constraint. From a Synthesis point of view it prevents XST from using dedicated clock resources and logic replication, based on the value of the MAX_FANOUT constraint. It specifies the use of low skew routing resources for any net. See "USELOWSKEWLINES" in the *Constraints Guide* for details.

- **XOR Collapsing**

  The XOR Collapsing (XOR_COLLAPSE) constraint controls whether cascaded XORs should be collapsed into a single XOR. See "XOR_COLLAPSE" in the *Constraints Guide* for details.

- **Slice Utilization Ratio**

  The Slice Utilization Ratio (SLICE_UTILIZATION_RATIO) constraint defines the area size that XST must not exceed during timing optimization. If the constraint cannot be met, XST makes timing optimization regardless.

  This constraint can be specified by selecting the Slice Utilization Ratio option under the Synthesis Options tab in the Process Properties dialog box within Project Navigator, or with the –slice_utilization_ratio command line option. See "SLICE_UTILIZATION_RATIO" in the *Constraints Guide* for details.

- **Slice Utilization Ratio Delta**

  The Slice Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) constraint is closely related to the SLICE_UTILIZATION_RATIO constraint. It defines the tolerance margin for the SLICE_UTILIZATION_RATIO constraint. If the ratio is within the margin set, the constraint is met and timing optimization can continue. For details, see "Speed Optimization Under Area Constraint" in Chapter 3, and also see "SLICE_UTILIZATION_RATIO_MAXMARGIN" in the *Constraints Guide*.

- **Map Entity on a Single LUT**

  The Map Entity on a Single LUT (LUT_MAP) constraint forces XST to map a single block into a single LUT. If a described function on an RTL level description does not fit in a single LUT, XST issues an error message. See "LUT_MAP" in the *Constraints Guide* for details.

- **Read Cores**

  The Read Cores (–read_cores) command line option enables or disables XST to read EDIF or NGC core files for timing estimation and device utilization control. Please refer to "Cores Processing" in Chapter 3 for more information. The Read Cores command line option has three possible values:

  - *no*: disables cores processing
  - *yes*: enables cores processing, but maintains the core as a black box and does not further incorporate the core into the design
  - *optimize*: enables cores processing, and merges the core's netlist into the overall design

  Define this option globally with the –read_cores command line option of the **run** command. Following is the basic syntax:

  ```
  -read_cores {yes|no|optimize}
  ```

  The default is *yes*.

  In Project Navigator, set –read_cores globally with the Read Cores option in the Synthesis Options tab of the Process Properties dialog box. Note that the *optimize* option is not available in Project Navigator.

- **Use Carry Chain**

  XST uses carry chain resources to implement certain macros, but there are situations where you can get better results by avoiding the use of carry chain. The Use Carry Chain (USE_CARRY_CHAIN) constraint can deactivate carry chain use for macro generation. See "USE_CARRY_CHAIN" in the *Constraints Guide* for details.

- **Convert Tristates to Logic**

  Since some devices do not support internal tristates, XST automatically replaces tristates with equivalent logic. Because the logic generated from tristates can be combined and optimized with surrounding logic, the replacement of internal tristates by logic for other devices can lead to better speed, and in some cases, better area optimization.

  In some situations XST cannot make the tristate to logic replacement automatically, as it may lead to wrong design behavior or to multi-sources. This can happen when the hierarchy is preserved or XST does not have full design visibility (for example, when the design is synthesized on a block by block basis). In these cases, XST issues a warning message during the Low Level Optimization step. Depending on the design situation you can continue the design flow and the replacement can be done by MAP, or you can force the replacement by applying the TRISTATE2LOGIC constraint set to *yes* on a particular block or signal. See "TRISTATE2LOGIC" in the *Constraints Guide* for details.

- **Use Clock Enable**

  The Use Clock Enable (USE_CLOCK_ENABLE) constraint enables or disables the use of the clock enable function in flip-flops. The disabling of the clock enable function is typically used for ASIC prototyping on FPGAs. See "USE_CLOCK_ENABLE" in the *Constraints Guide* for details.

- **Use Synchronous Set**

  The Use Synchronous Set (USE_SYNC_SET) constraint enables or disables the use of synchronous set function in flip-flops. The disabling of the synchronous set function is typically used for ASIC prototyping on FPGAs. See "USE_SYNC_SET" in the *Constraints Guide* for details.

- **Use Synchronous Reset**

  The Use Synchronous Reset (USE_SYNC_RESET) constraint enables or disables the usage of synchronous reset function of flip-flops. The disabling of the Synchronous Reset function could be used for ASIC prototyping flow on FPGAs. See "USE_SYNC_RESET" in the *Constraints Guide* for details.

- **Use DSP48**

  XST enables you to use the resources of the DSP48 blocks introduced in Virtex-4™ devices. You can control the use of these resources globally for all modules in the entire design by using the –use_dsp48 option of the **run** command. See "USE_DSP48" in the *Constraints Guide* for details.

CPLD Constraints (non-timing)

This section lists options that only apply to CPLDs—not FPGAs.

Note: Please note that in many cases, a particular constraint can be applied globally to an entire entity or model, or alternatively, it can be applied locally to individual signals, nets or instances. See Table 5-1 for valid constraint targets.

- **Clock Enable**

  The Clock Enable (–pld_ce) constraint specifies how sequential logic should be implemented when it contains a clock enable, either using the specific device resources available for that or generating equivalent logic.

  This option allows you to specify the way the clock enable function will be implemented if presented in the design. Two values are available:

♦   *yes* (check box is checked): the synthesizer implements the use of the Clock Enable signal of the device

♦   *no* (check box is not checked): the Clock enable function will be implemented through equivalent logic

Keeping or not keeping the clock enable signal depends on the design logic. Sometimes, when the clock enable is the result of a Boolean expression, saying *no* with this option may improve the fitting result because the input data of the flip-flop is simplified when it is merged with the clock enable expression.

Define this constraint globally with the –pld_ce command line option of the **run** command. Following is the basic syntax:

```
-pld_ce {yes|no}
```

The default is *yes.*

In Project Navigator, specify this option globally with the Clock Enable option in the Xilinx Specific Options tab of the Process Properties dialog box.

- **Data Gate**

    The CoolRunner™-II DataGate (DATA_GATE) feature provides direct means of reducing power consumption in your design. Each I/O pin input signal passes through a latch that can block the propagation of incident transitions during periods when such transitions are not of interest to your CPLD design. Input transitions that do not affect the CPLD design function still consume power, if not latched, as they are routed among the CPLD's Function Blocks. By asserting the DataGate control I/O pin on the device, selected I/O pin inputs become latched, thereby eliminating the power dissipation associated with external transitions on those pins. See "DATA_GATE" in the *Constraints Guide* for details

- **Keep Hierarchy**

    This option is related to the hierarchical blocks (VHDL entities, Verilog modules) specified in the HDL design and does not concern the macros inferred by the HDL synthesizer. The Keep Hierarchy (KEEP_HIERARCHY) constraint enables or disables hierarchical flattening of user-defined design units, and controls whether it is passed on as an implementation constraint. See "KEEP_HIERARCHY" in the *Constraints Guide* for details.

- **Macro Preserve**

    The Macro Preserve (–pld_mp) option is useful for making the macro handling independent of design hierarchy processing. This allows you to merge all hierarchical blocks in the top module, while still keeping the macros as hierarchical modules. You can also keep the design hierarchy except for the macros, which are merged with the surrounding logic. Merging the macros sometimes gives better results for design fitting. Two values are available for this option:

    ♦   *yes* (check box is checked): macros are preserved and generated by Macro+.

    ♦   *no* (check box is not checked): macros are rejected and generated by HDL synthesizer

    Depending on the Flatten Hierarchy value, a rejected macro becomes a hierarchical block (Flatten Hierarchy=**no**) or is merged in the design logic (Flatten Hierarchy=**yes**). Very small macros (2-bit adders, 4-bit multiplexers) are always merged, independent of the Macro Preserve or Flatten Hierarchy options.

    Define this option globally with the –pld_mp command line option of the **run** command. Following is the basic syntax:

```
-pld_mp {yes|no}
```

The default is *yes*.

In Project Navigator, specify this option globally with the Macro Preserve option in the Xilinx Specific Options tab of the Process Properties dialog box.

- **No Reduce**

    The No Reduce (NOREDUCE) constraint prevents minimization of redundant logic terms that are typically included in a design to avoid logic hazards or race conditions. This constraint also identifies the output node of a combinatorial feedback loop to ensure correct mapping. See "NOREDUCE" in the *Constraints Guide* for details.

- **WYSIWYG**

    The goal of the WYSIWYG option is to have a netlist as much as possible reflect the user specification. That is, all the nodes declared in the HDL design are preserved.

    If WYSIWYG mode is enabled (**yes**), then XST preserves all the user internal signals (nodes), creates SOURCE_NODE constraints in the NGC file for all these nodes, and skips design optimization (collapse, factorization); only boolean equation minimization is performed.

    Define globally with the –**wysiwyg** command line option of the **run** command. Following is the basic syntax:

    ```
    -wysiwyg {yes|no}
    ```

    The default is *no*.

    The constraint can also be defined globally with the WYSIWYG option in the Xilinx Specific Options tab in the Process Properties dialog box within Project Navigator.

- **XOR Preserve**

    The XOR Preserve (–pld_xp) constraint enables or disables hierarchical flattening of XOR macros. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, XOR macros are preserved (check box is checked).

    The XORs inferred by HDL synthesis are also considered as macro blocks in the CPLD flow, but they are processed separately to give more flexibility for the use of device macrocells XOR gates. Therefore, you can decide to flatten its design (Flatten Hierarchy *yes*, Macro Preserve *no*) but you want to preserve the XORs. Preserving XORs has a great impact on reducing design complexity. Two values are available for this option:

    - ♦ *yes*—XOR macros are preserved
    - ♦ *no*—XOR macros are merged with surrounded logic

    Preserving the XORs, generally, gives better results; that is, the number of PTerms is lower. The *no* value is useful to obtain completely flat netlists. Sometimes, applying the global optimization on a completely flat design improves the design fitting.

    You obtain a completely flattened design when selecting the following options:

    - ♦ Flatten Hierarchy—yes
    - ♦ Macro Preserve—no
    - ♦ XOR Preserve—no

    The *no* value for this option does not guarantee the elimination of the XOR operator from the EDIF netlist. During the netlist generation, the netlist mapper tries to recognize and infer XOR gates in order to decrease the logic complexity. This process is

independent of the XOR preservation done by HDL synthesis and is guided only by the goal of complexity reduction.

Define this constraint globally with the –pld_xp command line option of the **run** command. Following is the basic syntax:

`-pld_xp {yes|no}`

The default is *yes*.

In Project Navigator, specify –pld_xp globally with the XOR Preserve option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator.

Timing Constraints

Timing constraints supported by XST can be applied either via the –glob_opt command line switch, which is the same as selecting Global Optimization Goal from the Synthesis Options tab of the Process Properties menu in Project Navigator, or via the constraints file.

- Using the –glob_opt/Global Optimization Goal method allows you to apply the five global timing constraints (ALLCLOCKNETS, OFFSET_IN_BEFORE, OFFSET_OUT_AFTER, INPAD_TO_OUTPAD and MAX_DELAY). These constraints are applied globally to the entire design. You cannot specify a value for these constraints as XST optimizes them for the best performance. Note that these constraints are overridden by constraints specified in the constraints file.

- Using the constraint file method you can specify timing constraints using native UCF syntax. XST supports constraints such as TNM_NET, TIMEGRP, PERIOD, TIG, FROM-TO etc., including wildcards and hierarchical names.

  Note: Timing constraints are only written to the NGC file when the Write Timing Constraints property is checked *yes* in the Process Properties dialog box in Project Navigator, or the *–write_timing_constraints* option is specified when using the command line. By default, they are not written to the NGC file.

Independent of the way timing constraints are specified, there are three additional options that affect timing constraint processing:

- **Cross Clock Analysis**

  The Cross Clock Analysis command (–cross_clock_analysis) allows inter-clock domain analysis during timing optimization. By default (*no*), XST does not perform this analysis.

  Define this option globally with the –cross_clock_analysis command line option of the **run** command. Following is the basic syntax:

      `–cross_clock_analysis {yes|no}`

  The default is *no*.

  In Project Navigator, specify this option globally with the Cross Clock Analysis option in the Synthesis Options tab of the Process Properties dialog box.

- **Write Timing Constraints**

  The Write Timing Constraints option (–write_timing_constraints) in one of your status reports enables or disables propagation of timing constraints to the NGC file that are specified in HDL code. These timing constraints in the NCG file will be used during place and route, as well as synthesis optimization.

  Define Write Timing Constraints globally with the –write_timing_constraints command line option of the **run** command. Following is the basic syntax.

  ```
  -write_timing_constraints {yes|no}
  ```

  The default is *no* — not to write timing constraints to the NGC file.

  In Project Navigator, specify this option globally with the Write Timing Constraints option in the Synthesis Options tab of the Process Properties dialog box.

- **Clock Signal**

  If a clock signal goes through combinatorial logic before being connected to the clock input of a flip-flop, XST cannot identify what input pin or internal signal is the real clock signal. The CLOCK_SIGNAL constraint allows you to define the clock signal. See "CLOCK_SIGNAL" in the Constraints Guide for details.

Global Timing Constraints Support

XST supports the following global timing constraints.

- **Global Optimization Goal**

  XST can optimize different regions (register to register, inpad to register, register to outpad, and inpad to outpad) of the design depending on the global optimization goal. Please refer to "Incremental Synthesis Flow" in Chapter 3 for a detailed description of supported timing constraints. The Global Optimization Goal (–glob_opt) command line option selects the global optimization goal.

  Note: You cannot specify a value for Global Optimization Goal/–glob_opt. XST optimizes the entire design for the best performance.

  The following constraints can be applied by using the Global Optimization Goal option.

  - **ALLCLOCKNETS**: optimizes the period of the entire design.
  - **OFFSET_IN_BEFORE**: optimizes the maximum delay from input pad to clock, either for a specific clock or for an entire design.
  - **OFFSET_OUT_AFTER**: optimizes the maximum delay from clock to output pad, either for a specific clock or for an entire design.
  - **INPAD_TO_OUTPAD**: optimizes the maximum delay from input pad to output pad throughout an entire design.
  - **MAX_DELAY**: incorporates all previously mentioned constraints.

  These constraints affect the entire design and only apply if no timing constraints are specified via the constraint file.

  Define this option globally with the **-glob_opt** command line option of the **run** command. Following is the basic syntax:

  ```
  -glob_opt {allclocknets|offset_in_before|offset_out_after
       |inpad_to_outpad |max_delay}
  ```

You can specify –glob_opt globally with the Global Optimization Goal option in the Synthesis Options tab of the Process Properties dialog box within the Project Navigator.

Domain Definitions

The possible domains are illustrated in the following schematic.

- **ALLCLOCKNETS** (register to register): identifies by default, all paths from register to register on the same clock for all clocks in a design. To take into account inter-clock domain delays, the command line switch –cross_clock_analysis must be set to yes.

- **OFFSET_IN_BEFORE** (inpad to register): identifies all paths from all primary input ports to either all sequential elements or the sequential elements driven by the given clock signal name.

- **OFFSET_OUT_AFTER** (register to outpad): is similar to the previous constraint, but sets the constraint from the sequential elements to all primary output ports.

- **INPAD_TO_OUTPAD** (inpad to outpad): sets a maximum combinational path constraint.

- **MAX_DELAY**: identifies all paths defined by the following timing constraints: ALLCLOCKNETS, OFFSET_IN_BEFORE, OFFSET_OUT_AFTER, INPAD_TO_OUTPAD.
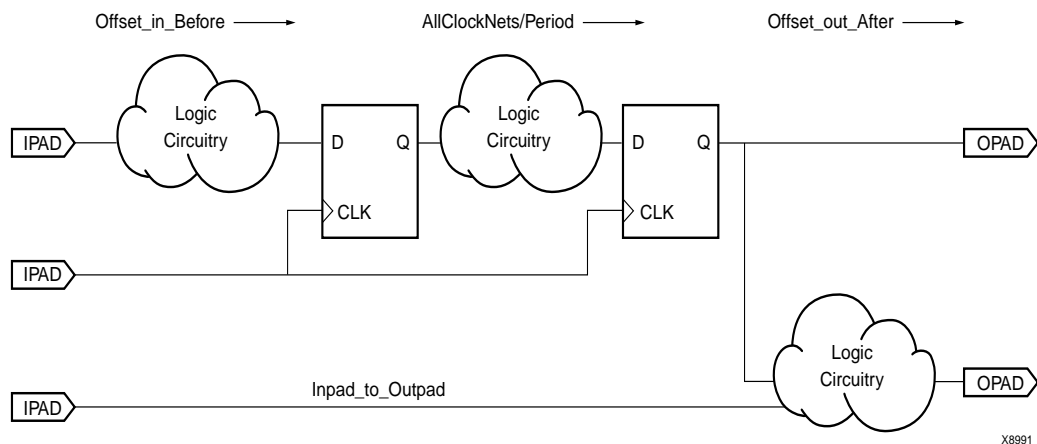
XCF Timing Constraint Support

IMPORTANT: If you specify timing constraints in the XCF file, Xilinx strongly suggests that you use '/' character as a hierarchy separator instead of '_'. Please refer to "Hierarchy Separator," page 248 for details on its usage.

IMPORTANT: If all or part of a specified timing constraint is not supported by XST, then XST generates a warning about this and ignores the unsupported timing constraint or unsupported part of it in the Timing Optimization step. If the Write Timing Constraints option is set to *yes*, XST propagates the entire constraint to the final netlist, even if it was ignored at the Timing Optimization step.

The following timing constraints are supported in the XST Constraints File (XCF).

- **Period**

  PERIOD is a basic timing constraint and synthesis constraint. A clock period specification checks timing between all synchronous elements within the clock domain as defined in the destination element group. The group may contain paths that pass between clock domains if the clocks are defined as a function of one or the other.

  See "PERIOD" in the *Constraints Guide* for details.

  XCF Syntax:

  NET *netname* **PERIOD =** *value* [{**HIGH** | **LOW**} *value*]**;**

- **Offset**

  OFFSET is a basic timing constraint. It specifies the timing relationship between an external clock and its associated data-in or data-out pin. OFFSET is used only for pad-related signals, and cannot be used to extend the arrival time specification method to the internal signals in a design.

  OFFSET allows you to:

  ♦ Calculate whether a setup time is being violated at a flip-flop whose data and clock inputs are derived from external nets.

  ♦ Specify the delay of an external output net derived from the Q output of an internal flip-flop being clocked from an external device pin.

  See "OFFSET" in the *Constraints Guide* for details.

  XCF Syntax:

  OFFSET = {**IN**|**OUT**} *offset_time* [*units*] {**BEFORE**|**AFTER**}
      clk_name [**TIMEGRP** *group_name*]**;**

- **From-To**

  FROM-TO defines a timing constraint between two groups. A group can be user-defined or predefined (FFS, PADS, RAMS). See "FROM-TO" in the *Constraints Guide* for details.

  Example:

  XCF Syntax:

  TIMESPEC TS*name* **= FROM** *group1* **TO** *group2* *value***;**

- **TNM**

  TNM is a basic grouping constraint. Use TNM (Timing Name) to identify the elements that make up a group which you can then use in a timing specification. TNM tags specific FFS, RAMs, LATCHES, PADS, BRAMS_PORTA, BRAMS_PORTB, CPUS, HSIOS, and MULTS as members of a group to simplify the application of timing specifications to the group.

  The RISING and FALLING keywords may also be used with TNMs. See "TNM" in the *Constraints Guide* for details.

  XCF Syntax:

  {**INST** | **NET** | **PIN**} *inst_net_or_pin_name* **TNM =** [*predefined_group:*]
  *identifier***;**

- **TNM Net**

  TNM_NET is essentially equivalent to TNM on a net *except* for input pad nets. (Special rules apply when using TNM_NET with the PERIOD constraint for Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X, Virtex-4™ or Spartan-3™

DLL/DCMs. See the "PERIOD Specifications on CLKDLLs and DCMs" subsection of "PERIOD" in the *Constraints Guide*.)

A TNM_NET is a property that you normally use in conjunction with an HDL design to tag a specific net. All downstream synchronous elements and pads tagged with the TNM_NET identifier are considered a group. See "TNM_NET" in the *Constraints Guide* for details.

XCF Syntax:

```
NET netname TNM_NET = [predefined_group:] identifier;
```

- **TIMEGRP**

  TIMEGRP is a basic grouping constraint. In addition to naming groups using the TNM identifier, you can also define groups in terms of other groups. You can create a group that is a combination of existing groups by defining a TIMEGRP constraint.

  You can place TIMEGRP constraints in a constraints file (XCF or NCF). You can use TIMEGRP attributes to create groups using the following methods.

  ♦ Combining multiple groups into one

  ♦ Defining flip-flop subgroups by clock sense

  See "TIMEGRP" in the *Constraints Guide* for details.

  XCF Syntax:

  ```
  TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...];
  ```

- **TIG**

  The TIG constraint causes all paths going through a specific net to be ignored for timing analyses and optimization purposes. This constraint can be applied to the name of the signal affected. See "TIG" in the *Constraints Guide* for details.

  XCF Syntax:

  ```
  NET net_name TIG;
  ```

Constraints Summary

Table 5-1 summarizes all available XST-specific non-timing related options. This table shows the allowed values for each constraint, the type of objects they can be applied to, and any usage restrictions. Default values are indicated in bold.

Note: Please note that in many cases, a particular constraint can be applied globally to an entire entity or model, or alternatively, it can be applied locally to individual signals, nets or instances.

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
XST Constraints							
box_type	primitive, black_box, user_black_box	model, inst(in model)	VHDL: component, entity Verilog: label, module	component, entity	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E,/4 XC9500™, CoolRunner™ XPLA3
buffer_type	bufgdll, ibufg, **bufgp**, ibuf, none	net (in model)	signal	signal	yes	bufgdll, ibufg, **bufgp**, ibuf, bufr, none	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4
bufgce	**yes**, no, **true**, false	net (in model)	primary clock signal	primary clock signal	no	na	Virtex™-II/II Pro/II Pro X/4
bram_map	yes, **no**, true, **false**	model	VHDL: entity Verilog: module	entity	yes	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4
clock_buffer	bufgdll, ibufg, **bufgp**, ibuf, none	net (in model)	signal	signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4
clock_signal	**yes**, no, **true**, false	clock signal, net (in model)	clock signal	clock signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4
decoder_extract	**yes**, no **true**, false	model, net (in model)	entity, signal	entity, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4
enable_auto_floorplanning	na	na	na	na	yes	**no**, incremental-_design	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4,
enum_encoding	string containing space-separated binary codes	net (in model)	type	signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
equivalent_register_ removal	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
fsm_encoding	**auto**, one-hot, compact, sequential, gray, johnson, speed1,user	model, net (in model)	entity, signal	module, signal	yes	**auto**, one-hot, compact, sequential, gray, johnson, speed1, user	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
fsm_extract	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
fsm_style	**lut**, bram	model, net (in model)	entity, signal	module, signal	yes	**lut**, bram	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
full_case	na	na	case statement	case statement	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
incremental_synthesis	**yes**, no, **true**, false	model	entity	module	no	na	Spartan™-II/IIE/3, Virtex™ II/II Pro/ II Pro X/E/4
iob	true, false, **auto**	net(in model), inst(in model)	signal, instance	signal, instance	no	true, false, **auto**	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4Pro X/E/4
iostandard	*string*: See *Constraints Guide* for details	net(in model), inst(in model)	signal, instance	signal, instance	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X, XC9500™, CoolRunner™ XPLA3/II
keep	**yes**, no **true**, false	net (in model)	signal	signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
keep_hierarchy	**yes**, no, **true**, false, soft	model	entity	module	yes	**yes**, no, soft	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X, XC9500™, CoolRunner™ XPLA3/II
loc	*string*	net(in model), inst(in model)	signal (primary IO), ,instance1	signal (primary IO), instance	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
lut_map	**yes**, no, **true**, false	model	entity. architecture	module	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
max_fanout	*integer*	model, net (in model)	entity, signal	module, signal	yes	*integer*	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
move_first_stage	**yes**, no, **true**, false	model, primary clock signal, net (in model)	entity, primary clock signal	module, primary clock signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
move_last_stage	**yes**, no, **true**, false	model, primary clock signal, net (in model)	entity, primary clock signal	module, primary clock signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
mult_style	**auto**, block, lut, pipe_lut, kcm,csd	model, net (in model)	entity, signal	module, signal	yes	**auto**, block, lut, pipe_lut	Spartan-3™, Virtex™-II/II Pro / II Pro X
mux_extract	**yes**, no, force, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no, force	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
mux_style	**auto**, muxf, muxcy	model, net (in model)	entity, signal	module, signal	yes	**auto**, muxf, muxcy	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
noreduce	**yes**, no **true**, false	net (in model)	signal	signal	no	na	XC9500™, CoolRunner™ XPLA3/II
optimize_primitives	yes, **no** true, **false**	model, instance (in model)	entity, instance	module, instance	no	yes, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
opt_level	**1**, 2	model	entity	module	yes	**1**, 2	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
opt_mode	**speed**, area	model	entity	module	yes	**speed**, area	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
parallel_case	na	na	case statement	case statement	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
priority_extract	**yes**, no, force, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no, force	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4
ram_extract	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4
ram_style	**auto**, block, distributed	model, net (in model)	entity, signal	module, signal	yes	**auto**, block, distributed	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4
register_balancing	yes, **no**, forward, backward, **true**, false	model, net(in model), inst(in model)	entity, signal, FF instance name, primary clock signal	module, signal, FF instance name, primary clock signal	yes	**yes**, no, forward, backward	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4
register_duplication	**yes**, no, **true**, false	model, net (in model)	entity, signal	module	yes	**yes**, no	Spartan™-II/IIE/3, Virtex/II/II Pro/ II Pro X/E/4
register_powerup	*string*	net (in model)	type	signal	no	na	XC9500™, CoolRunner™ XPLA3/II
resource_sharing	**yes**, no **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
resynthesize	**yes**, no, **true**, false	model	entity	module	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
rom_extract	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
rom_style	**auto**, block, distributed	model, net (in model)	entity, signal	module, signal	yes	**auto**, block, distributed	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
safe_implementation	yes, **no**, true, **false**	model, net (in model)	entity, signal	module, signal	yes	yes, **no**	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
safe_recovery_state	*string*	net (in model)	signal	signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
shift_extract	**yes**, no **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
shreg_extract	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
signal_encoding	**auto**, one-hot, user	model, net (in model)	entity, signal	module, signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
slice_utilization_ratio	*integer* (range 0-100)	model	entity	module	yes	*integer* (range 0-100)	Spartan™-II/IIE/3, Virtex™ II/II Pro/ II Pro X/E/4
slice_utilization_ratio_max margin	*integer* (range 0-100)	model	entity	module	yes	*integer* (range 0-100)	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
translate_off	na	na	local, no target	local, no target	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
translate_on	na	na	local, no target	local, no target	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
use_carry_chain	**yes**, no, **true**, false	model, net (in model)	entity, signal	module, signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
use_clock_enable	**auto, yes**, no, true, false	model, net (in model) inst(in model)	entity, signal, FF instance name	module, signal, FF instance name	yes	**auto**, yes, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
use_dsp48	**auto,** yes, no, true, false	model, net (in model)	entity, signal	module, signal	yes	**auto**, yes, no	Virtex-4™
use_sync_reset	**auto**, yes no, true, false	model, net (in model) inst(in model)	entity, signal, FF instance name	module, signal, FF instance name	yes	**auto**, yes, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
use_sync_set	**auto**, yes, no, true, false	model, net (in model) inst(in model)	entity, signal, FF instance name	module, signal, FF instance name	yes	**auto**, yes, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
uselowskewlines	**yes**, no, **true**, false	net (in model)	signal	signal	no	na	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
xor_collapse	**yes**, no **true**, false	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE,3 Virtex™/II/II Pro/ II Pro X/E/4
XST Command Line Only Options							
bufg	na	na	na	na	yes	*integer*	XC9500™, CoolRunner™ XPLA3/II
bufr	na	na	na	na	yes	*integer*	Virtex-4™
bus_delimiter	na	na	na	na	yes	< >, [ ], { }, ( )	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
case	na	na	na	na	yes	upper, lower, **maintain**	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
duplication_suffix	na	na	na	na	yes	*string*%d*string*	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
hierarchy_separator	na	na	na	na	yes	_, / default is /	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
iobuf	na	na	na	na	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
iuc	na	na	na	na	yes	yes, **no**	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
lso	na	na	na	na	yes	*file_name*	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
pld_ce	na	na	na	na	yes	**yes**, no	XC9500™, CoolRunner™ XPLA3/II
pld_mp	na	na	na	na	yes	**yes**, no	XC9500™, CoolRunner™ XPLA3/II
pld_xp	na	na	na	na	yes	**yes**, no	XC9500™, CoolRunner™ XPLA3/II
read_cores	na	na	na	na	yes	**yes**, no, optimize	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
sd	na	na	na	na	yes	*directory_path*	Spartan™-II/IIE/3, Virtex™/II/II Pro/E
tristate2logic	yes, **no**, true, **false** (internal tristates only)	model, net (in model)	entity, signal	module, signal	yes	**yes**, no	Spartan™-II/IIE, Virtex™/II/II Pro/ II Pro X/E
slice_packing	na	na	na	na	yes	**yes**, no	XC9500™, CoolRunner™ XPLA3/II
uc	na	na	na	na	yes	*file_name*.xcf	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
verilog2001	na	na	na	na	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
vlgcase	na	na	na	na	yes	full, parallel, full-parallel	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

Table 5-1: **XST-Specific Non-timing Options**

Constraint Name	Constraint Value	XCF Constraint Syntax Target	VHDL Target	Verilog Target	Cmd Line	Cmd Value	Technology
vlgincdir	na	na	na	na	yes	*dir_path*	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
wysiwyg	na	na	na	na	yes	**yes**, no	XC9500™, CoolRunner™ XPLA3/II
xsthdpdir	*directory_path*	na	na	na	yes	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
xsthdpini	*file_name*	na	na	na	yes	*file_name*	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

The following table shows the timing constraints supported by XST that you can invoke only from the command line, or the Process Properties dialog box in Project Navigator.

Table 5-2: **XST Timing Constraints Supported Only by Command Line/Process Properties Dialog Box**

Option	Process Property (ProjNav)	Values	Technology
glob_opt	Global Optimization Goal	**allclocknets** inpad_to_outpad offset_in_before offset_out_after max_delay	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
cross_clock_analysis	Cross Clock Analysis	**yes**, no	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II
write_timing_constraints	Write Timing Constraints	yes, **no**	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4, XC9500™, CoolRunner™ XPLA3/II

The following table shows the timing constraints supported by XST that you can invoke only through the Xilinx® Constraint File (XCF).

Table 5-3: **XST Timing Constraints Supported Only in XCF**

Name	Value	Target	Technology
period	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
offset	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
timespec	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
tsidentifier	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
tmn	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
tnm_net	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
timegrp	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
tig	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4
from... to...	See the *Constraints Guide* for details.	See the *Constraints Guide* for details.	Spartan™-II/IIE/3, Virtex™/II/II Pro/ II Pro X/E/4

Implementation Constraints

This section explains how XST handles implementation constraints. See the *Constraints Guide* for details on the implementation constraints supported by XST.

Handling by XST

Implementation constraints control placement and routing. They are not directly useful to XST, and are simply propagated and made available to the implementation tools. When the –write_timing_constraints switch is set to *yes,* the constraints are written in the output NGC file (Note: TIG is propagated regardless of the setting). In addition, the object that an implementation constraint is attached to is preserved.

A binary equivalent of the implementation constraints is written to the NGC file, but since it is a binary file, you cannot edit the implementation constraints there. Alternatively, you can code implementation constraints in the XCF file according to one of the following syntaxes.

To apply a constraint to an entire entity, use one of the following two XCF syntaxes:

```
MODEL EntityName PropertyName;
MODEL EntityName PropertyName=PropertyValue;
```

To apply a constraint to specific instances, nets or pins within an entity, use one of the two following syntaxes:

```
BEGIN MODEL EntityName
   {NET|INST|PIN}{NetName|InstName|SigName} PropertyName;
END;

BEGIN MODEL EntityName
   {NET|INST|PIN}{NetName|InstName|SigName} PropertyName=Propertyvalue;
END;
```

When written in VHDL code, they should be specified as follows:

```
attribute PropertyName of {NetName|InstName|PinName} : {signal|label}
is "PropertyValue";
```

In a Verilog description, they should be written as follows:

```
// synthesis attribute PropertyName of {NetName |InstName|PinName} is
"PropertyValue";
```

In Verilog-2001, where descriptions must precede the signal, module or instance they refer to, it should be written as follows:

```
(* PropertyName="PropertyValue" *)
```

Examples

Following are three examples.

Example 1

When targeting an FPGA device, use the RLOC constraint to indicate the placement of a design element on the FPGA die relative to other elements. Assuming an SRL16 instance of name srl1 to be placed at location R9C0.S0, you may specify the following in your Verilog code:

```
// synthesis attribute RLOC of srl1 : "R9C0.S0";
```

You may specify the same attribute in the XCF file with the following lines:

```
BEGIN MODEL ENTNAME
  INST sr11 RLOC=R9C0.SO;
END;
```

The binary equivalent of the following line is written to the output NGC file:

```
INST srl1 RLOC=R9C0.S0;
```

Example 2

The NOREDUCE constraint, available with CPLDs, prevents the optimization of the boolean equation generating a given signal. Assuming a local signal is assigned the arbitrary function below, and a NOREDUCE constraint attached to the signal *s*:

```
signal s : std_logic;
attribute NOREDUCE : boolean;
attribute NOREDUCE of s : signal is "true";
...
s <= a or (a and b);
```

You may specify the same attribute in the XCF file with the following lines:

```
BEGIN MODEL ENTNAME
  NET s NOREDUCE;
  NET s KEEP;
END;
```

The following statements are written to the NGC file:

```
NET s NOREDUCE;
NET s KEEP;
```

Example 3

The PWR_MODE constraint, available when targeting CPLD families, controls the power consumption characteristics of macrocells. The following VHDL statement specifies that the function generating signal *s* should be optimized for low power consumption.

```
attribute PWR_MODE : string;
attribute PWR_MODE of s : signal is "LOW";
```

You may specify the same attribute in the XCF file with the following lines:

```
MODEL ENTNAME
  NET s PWR_MODE=LOW;
  NET s KEEP;
END;
```

The following statement is written to the NGC file by XST:

```
NET s PWR_MODE=LOW;
NET s KEEP;
```

If the attribute applies to an instance (for example, IOB, DRIVE, IOSTANDARD) and if the instance is not available (not instantiated) in the HDL source, then the HDL attribute can be applied to the signal on which XST infers the instance.

Third Party Constraints

This section describes constraints of third-party synthesis vendors that are supported by XST. For each of the constraints, Table 5-4 gives the XST equivalent. For information on what these constraints actually do, please refer to the corresponding vendor documentation. Note that "NA" stands for "Not Available".

Table 5-4:   **Third Party Constraints**

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
black_box	Synplicity	box_type	na	VHDL/ Verilog
black_box_pad_pin	Synplicity	na	na	na
black_box_tri_pins	Synplicity	na	na	na
cell_list	Synopsys	na	na	na
clock_list	Synopsys	na	na	na
Enum	Synopsys	na	na	na
full_case	Synplicity/ Synopsys	full_case	na	Verilog
ispad	Synplicity	na	na	na
map_to_module	Synopsys	na	na	na
net_name	Synopsys	na	na	na
parallel_case	Synplicity Synopsys	parallel_case	na	Verilog
return_port_name	Synopsys	na	na	na
resource_sharing directives	Synopsys	resource_sharing directives	na	VHDL/ Verilog
set_dont_touch_network	Synopsys	not required	na	na
set_dont_touch	Synopsys	not required	na	na
set_dont_use_cel_name	Synopsys	not required	na	na
set_prefer	Synopsys	na	na	na
state_vector	Synopsys	na	na	na
syn_allow_retiming	Synplicity	register_balancing	na	VHDL/ Verilog

Table 5-4: **Third Party Constraints**

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
syn_black_box	Synplicity	box_type	na	VHDL/ Verilog
syn_direct_enable	Synplicity	na	na	na
syn_edif_bit_format	Synplicity	na	na	na
syn_edif_scalar_format	Synplicity	na	na	na
syn_encoding	Synplicity	fsm_encoding	yes *Note:* The value *safe* is not supported for automatic recognition. Please use safe_implementation constraint in XST to activate this mode.	VHDL/ Verilog
syn_enum_encoding	Synplicity	enum_encoding	na	VHDL
syn_hier	Synplicity	keep_hierarchy	yes *Note:* Only the values *hard* and *remove* are supported for automatic recognition	VHDL/ Verilog
syn_isclock	Synplicity	na	na	na
syn_keep	Synplicity	keep*	yes	VHDL/ Verilog
syn_maxfan	Synplicity	max_fanout	yes	VHDL/ Verilog
syn_netlist_hierarchy	Synplicity	keep_hierarchy	na	VHDL/ Verilog
syn_noarrayports	Synplicity	na	na	na
syn_noclockbuf	Synplicity	buffer_type	yes	VHDL/ Verilog
syn_noprune	Synplicity	optimize_primitives	yes	VHDL/ Verilog
syn_pipeline	Synplicity	Register Balancing	na	VHDL/ Verilog
syn_probe	Synplicity	equivalent_register-_removal	yes	VHDL/ Verilog
syn_ramstyle	Synplicity	na	na	NA
syn_reference_clock	Synplicity	na	na	NA
syn_romstyle	Synplicity	na	na	NA
syn_sharing	Synplicity	register_duplication	yes	VHDL/ Verilog

Table 5-4: **Third Party Constraints**

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
syn_state_machine	Synplicity	fsm_extract	na	VHDL/ Verilog
syn_tco <n>	Synplicity	na	na	na
syn_tpd <n>	Synplicity	na	na	na
syn_tristate	Synplicity	na	na	na
syn_tristatetomux	Synplicity	na	na	na
syn_tsu <n>	Synplicity	na	na	na
syn_useenables	Synplicity	na	na	na
syn_useioff	Synplicity	iob	na	VHDL/ Verilog
synthesis translate_off / synthesis translate_on	Synplicity/ Synopsys	synthesis translate_off / synthesis translate_on	yes	VHDL/ Verilog
xc_alias	Synplicity	na	na	na
xc_clockbuftype	Synplicity	buffer_type	na	VHDL/ Verilog
xc_fast	Synplicity	fast	na	VHDL/ Verilog
xc_fast_auto	Synplicity	fast	na	VHDL/ Verilog
xc_global_buffers	Synplicity	bufg	na	VHDL/ Verilog
xc_ioff	Synplicity	iob	na	VHDL/ Verilog
xc_isgsr	Synplicity	na	na	na
xc_loc	Synplicity	loc	yes	VHDL/ Verilog
xc_map	Synplicity	lut_map	yes **Note:** Only the value *lut* is supported for automatic recognition	VHDL/ Verilog
xc_ncf_auto_relax	Synplicity	na	na	na
xc_nodelay	Synplicity	nodelay	na	VHDL/ Verilog
xc_padtype	Synplicity	iostandard	na	VHDL/ Verilog
xc_props	Synplicity	na	na	na

Table 5-4: **Third Party Constraints**

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
xc_pullup	Synplicity	pullup	pullup	VHDL/ Verilog
xc_rloc	Synplicity	rloc	yes	VHDL/ Verilog
xc_fast	Synplicity	fast	na	VHDL/ Verilog
xc_slow	Synplicity	na	na	na
xc_uset	Synplicity	u_set	yes	VHDL/ Verilog

* You must use the KEEP constraint instead of SIGNAL_PRESERVE.

Verilog example:

```
module testkeep (in1, in2, out1);
  input in1;
  input in2;
  output out1;

  wire aux1;
  wire aux2;

// synthesis attribute keep of aux1 is "true"
// synthesis attribute keep of aux2 is "true"

  assign aux1 = in1;
  assign aux2 = in2;
  assign out1 = aux1 & aux2;

endmodule
```

The KEEP constraint can also be applied through the separate synthesis constraint file:

XCF Example Syntax:

```
BEGIN MODEL testkeep
  NET aux1 KEEP=true;
END;
```

These are the only two ways of preserving a signal/net in an HDL design and preventing optimization on the signal or net during synthesis.

Constraints Precedence

Priority depends on the file in which the constraint appears. A constraint in a file accessed later in the design flow overrides a constraint in a file accessed earlier in the design flow. Priority is as follows (first listed is the highest priority, last listed is the lowest).

1. Synthesis Constraint File
2. HDL file
3. Command Line/Process Properties dialog box in Project Navigator

Chapter 6

VHDL Language Support

This chapter explains how VHDL is supported for XST. The chapter provides details on the VHDL language, supported constructs, and synthesis options in relationship to XST. The sections in this chapter are as follows:

- *"Introduction"*
- *"File Type Support"*
- *"Data Types in VHDL"*
- *"Record Types"*
- *"Initial Values"*
- *"Objects in VHDL"*
- *"Operators"*
- *"Entity and Architecture Descriptions"*
- *"Combinatorial Circuits"*
- *"Sequential Circuits"*
- *"Functions and Procedures"*
- *"Assert Statement"*
- *"Packages"*
- *"VHDL Language Support"*
- *"VHDL Reserved Words"*

For a complete specification of the VHDL hardware description language, refer to the IEEE VHDL Language Reference Manual.

For a detailed description of supported design constraints, refer to Chapter 5, "Design Constraints." For a description of VHDL attribute syntax, see "VHDL Attribute Syntax" in Chapter 5.

Introduction

VHDL is a hardware description language that offers a broad set of constructs for describing even the most complicated logic in a compact fashion. The VHDL language is designed to fill a number of requirements throughout the design process:

- Allows the description of the structure of a system — how it is decomposed into subsystems, and how those subsystems are interconnected.
- Allows the specification of the function of a system using familiar programming language forms.

- Allows the design of a system to be simulated prior to being implemented and manufactured. This feature allows you to test for correctness without the delay and expense of hardware prototyping.

- Provides a mechanism for easily producing a detailed, device-dependent version of a design to be synthesized from a more abstract specification. This feature allows you to concentrate on more strategic design decisions, and reduce the overall time to market for the design.

File Type Support

XST supports a limited File Read capability for VHDL. You can use this file read capability, for example, to initialize RAMs from an external file. Please refer to "Initializing Block RAM" in Chapter 2 for more information.

You can use any of the following read functions, which are supported by the standard, std.textio and ieee.std_logic_textio packages, respectively:

Table 6-1: **Supported File Types**

Function	Package
file (type "text" only)	standard
access (type "line" only)	standard
file_open (file, name, open_kind)	standard
file_close (file)	standard
endfile (file)	standard
text	std.textio
line	std.textio
width	std.textio
readline (text, line)	std.textio
read (line, bit)	std.textio
read (line, bit_vector)	std.textio
read (line, boolean)	std.textio
read (line, std_ulogic)	ieee.std_logic_textio
read (line, std_ulogic_vector)	ieee.std_logic_textio
read (line, std_logic_vector)	ieee.std_logic_textio

Limitations

- XST does not support file write capability. Consequently, when a file is opened, the direction of the file is not checked and assumed to be "in".

- The only characters that may appear in the file are '0' and '1'. Other characters are not yet supported (for example, spaces, 'X', 'Z' are not supported). If any line of the file includes characters other than '0' and '1', XST rejects the design.

- Avoid using identical names for files placed in different directories.

- Avoid using conditional calls to *read* procedures, as shown in the following example. This can cause problems during simulation.

```
if SEL = '1' then
  read (MY_LINE, A(3 downto 0));
else
  read (MY_LINE, A(1 downto 0));
end if;
```

- When using the endfile function, if you use the following description style:

```
while (not endfile (MY_FILE)) loop
  readline (MY_FILE, MY_LINE);
  read (MY_LINE, MY_DATA);
end loop;
```

XST rejects the design and generates the following error message:

```
Line <MY_LINE> has not enough elements for target <MY_DATA>.
```

To fix the problem, add "*exit when endfile (MY_FILE);*" to the while loop as shown in the following example:

```
while (not endfile (MY_FILE)) loop
  readline (MY_FILE, MY_LINE);
  exit when endfile (MY_FILE);
  read (MY_LINE, MY_DATA);
end loop;
```

Data Types in VHDL

XST accepts the following VHDL basic types:

- Enumerated Types:
  - BIT ('0','1')
  - BOOLEAN (false, true)
  - REAL ($-. to $+.)
  - STD_LOGIC ('U','X','0','1','Z','W','L','H','-') where:

    'U' means uninitialized

    'X' means unknown

    '0' means low

    '1' means high

    'Z' means high impedance

    'W' means weak unknown

    'L' means weak low

    'H' means weak high

    '-' means don't care

    For XST synthesis, the '0' and 'L' values are treated identically, as are '1' and 'H'. The 'X', and '-' values are treated as don't care. The 'U' and 'W' values are not accepted by XST. The 'Z' value is treated as high impedance.

  - User defined enumerated type:

    type COLOR is (RED,GREEN,YELLOW);

- Bit Vector Types:
  - ♦ BIT_VECTOR
  - ♦ STD_LOGIC_VECTOR

    Unconstrained types (types whose length is not defined) are not accepted.
- Integer Type: INTEGER

The following types are VHDL predefined types.

- BIT
- BOOLEAN
- BIT_VECTOR
- INTEGER
- REAL

The following types are declared in the STD_LOGIC_1164 IEEE package.

- STD_LOGIC
- STD_LOGIC_VECTOR

This package is compiled in the IEEE library. In order to use one of these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

Overloaded Data Types

The following basic types can be overloaded.

- Enumerated Types:
  - ♦ STD_ULOGIC: contains the same nine values as the STD_LOGIC type, but does not contain predefined resolution functions
  - ♦ X01: subtype of STD_ULOGIC containing the 'X', '0' and '1' values
  - ♦ X01Z: subtype of STD_ULOGIC containing the 'X', '0', '1' and 'Z' values
  - ♦ UX01: subtype of STD_ULOGIC containing the 'U', 'X', '0' and '1' values
  - ♦ UX01Z: subtype of STD_ULOGIC containing the 'U', 'X', '0','1' and 'Z' values
- Bit Vector Types:
  - ♦ STD_ULOGIC_VECTOR
  - ♦ UNSIGNED
  - ♦ SIGNED

  Unconstrained types (types whose length is not defined) are not accepted.
- Integer Types:
  - ♦ NATURAL
  - ♦ POSITIVE

  Any integer type within a user-defined range. As an example, "type MSB is range 8 to 15;" means any integer greater than 7 or less than 16.

The types NATURAL and POSITIVE are VHDL predefined types.

The types STD_ULOGIC (and subtypes X01, X01Z, UX01, UX01Z), STD_LOGIC, STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR are declared in the STD_LOGIC_1164 IEEE package. This package is compiled in the library IEEE. In order to use one of these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

The types UNSIGNED and SIGNED (defined as an array of STD_LOGIC) are declared in the STD_LOGIC_ARITH IEEE package. This package is compiled in the library IEEE. In order to use these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_ARITH.all;
```

Multi-dimensional Array Types

XST supports multi-dimensional array types of up to three dimensions. Arrays can be signals, constants, or VHDL variables. You can do assignments and arithmetic operations with arrays. You can also pass multi-dimensional arrays to functions, and use them in instantiations.

The array must be fully constrained in all dimensions. An example is shown below:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

You can also declare an array as a matrix, as in the following example:

```
subtype TAB13 is array (7 downto 0,4 downto 0)
    of STD_LOGIC_VECTOR (8 downto 0);
```

The following examples demonstrate the various uses of multi-dimensional array signals and variables in assignments.

Consider the declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;

signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CST_A : TAB03 := (
  ("0000000","0000001","0000010","0000011","0000100")
  ("0010000","0010001","0010010","0100011","0010100")
  ("0100000","0100001","0100010","0100011","0100100"));
```

A multi-dimensional array signal or variable can be completely used:

```
TAB_A <= TAB_B;
TAB_C <= TAB_D;
TAB_C <= CNST_A;
```

Just an index of one array can be specified:

```
TAB_A (5) <= WORD_A;
TAB_C (1) <= TAB_A;
```

Just indexes of the maximum number of dimensions can be specified:

```
TAB_A (5) (0) <= '1';
TAB_C (2) (5) (0) <= '0'
```

Just a slice of the first array can be specified:

```
TAB_A (4 downto 1) <= TAB_B (3 downto 0);
```

Just an index of a higher level array and a slice of a lower level array can be specified:

```
TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1);
TAB_D (0) (4) (2 downto 0) <= CNST_A (5 downto 3)
```

Now add the following declaration:

```
subtype MATRIX15 is array(4 downto 0, 2 downto 0)
  of STD_LOGIC_VECTOR (7 downto 0);
```

A multi-dimensional array signal or variable can be completely used:

```
MATRIX15 <= CNST_A;
```

Just an index of one row of the array can be specified:

```
MATRIX15 (5) <= TAB_A;
```

Just indexes of the maximum number of dimensions can be specified:

```
MATRIX15 (5,0) (0) <= '1';
```

Just a slice of one row can be specified:

```
MATRIX15 (4,4 downto 1) <= TAB_B (3 downto 0);
```

Note: Indices may be variable.

Record Types

XST supports record types. An example of a record is shown below.

```
type REC1 is record
  field1: std_logic;
  field2: std_logic_vector (3 downto 0)
end record;
```

- Record types can contain other record types.
- Constants can be record types.
- Record types cannot contain attributes.
- XST supports aggregate assignments to record signals.

Initial Values

In VHDL, you can initialize registers when you declare them.

The value:

- Must be a constant.
- Cannot depend on earlier initial values.
- Cannot be a function or task call.
- Can be a parameter value propagated to a register.

When you give a register an initial value in a declaration, XST sets this value on the output of the register at global reset, or at power up. A value assigned this way is carried in the NGC file as an INIT attribute on the register, and is independent of any local reset.

Example:

```
signal arb_onebit : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

You can also assign a set/reset value to a register via your behavioral VHDL code. Do this by assigning a value to a register when the register's reset line goes to the appropriate value as in the following example.

Example:

```
process (clk, rst)
begin
  if rst='1' then
    arb_onebit <= '0';
  end if;
end process;
```

When you set the initial value of a variable in the behavioral code, it is implemented in the design as a flip-flop whose output can be controlled by a local reset; as such it is carried in the NGC file as an FDP or FDC flip-flop.

Local Reset ≠ Global Reset

Note that local reset is independent of global reset. Registers controlled by a local reset may be set to a different value from registers whose value is only reset at global reset (power up). In the following example, the register arb_onebit is set to '1' at global reset, but a pulse on the local reset (rst) can change its value to '0'.

Example:

```
entity top is
  Port (
    clk, rst : in std_logic;
    a_in : in std_logic;
    dout : out std_logic);
end top;
```

```
architecture Behavioral of top is
signal arb_onebit : std_logic := '1';

begin
  process (clk, rst)
  begin
    if rst='1' then
      arb_onebit <= '0';
    elsif (clk'event and clk='1') then
      arb_onebit <= a_in;
    end if;
  end process;

  dout <= arb_onebit;
end Behavioral;
```

This sets the initial value on the register's output to '1' at initial power up, but since this is dependent upon a local reset, the value changes to '0' whenever the local set/reset is activated.

Default Initial Values on Memory Elements

Because every memory element in a Xilinx® FPGA must come up in a known state, in certain cases, XST does not use IEEE standards for initial values. In the previous example, if signal arb_onebit were not initialized to '1', XST would assign it a default of '0' as its initial state. In this case, XST does not follow the IEEE standard, where 'U' is the default for std_logic. This process of initialization is the same for both registers and RAMs.

Where possible, XST adheres to the IEEE VHDL standard when initializing signal values. If no initial values are supplied in the VHDL code, XST uses the default values (where possible) as outlined in the XST column in Table 6-2.

Table 6-2: **Initial Values**

Type	IEEE	XST
bit	'0'	'0'
std_logic	'U'	'0'
bit_vector (3 downto 0)	"0000"	"0000"
std_logic_vector (3 downto 0)	"0000"	"0000"
integer (unconstrained)	integer'left	integer'left
integer range 7 downto 0	integer'left = 7	integer'left = 7 (coded as "111")
integer range 0 to 7	integer'left = 0	integer'left = 0 (coded as "000")
Boolean	FALSE	FALSE (coded as '0')
enum(S0,S1,S2,S3)	type'left = S0	type'left = S0 (coded as "000")

Default Initial Values on Unconnected Ports

Output ports that are left unconnected default to the values noted in the XST column of Table 6-2. If the output port has an initial condition, XST ties the unconnected output port to the explicitly defined initial condition. According to the IEEE VHDL specification, input ports cannot be left unconnected. As a result, XST always gives an error if an input port is not connected; even the **open** keyword will not suffice for an unconnected input port.

Objects in VHDL

VHDL objects include signals, variables, and constants.

Signals can be declared in an architecture declarative part and used anywhere within the architecture. Signals can also be declared in a block and used within that block. Signals can be assigned by the assignment operator "<=".

Example:

```
signal sig1 : std_logic;
sig1 <= '1';
```

Variables are declared in a process or a subprogram, and used within that process or that subprogram. Variables can be assigned by the assignment operator ":=".

Example:

```
variable var1 : std_logic_vector (7 downto 0);
var1 := "01010011";
```

Constants can be declared in any declarative region, and can be used within that region. Their value cannot be changed once declared.

Example:

```
signal sig1 : std_logic_vector (5 downto 0);
constant init0 : std_logic_vector (5 downto 0) := "010111";
sig1 <= init0;
```

Operators

Supported operators are listed in Table 6-9. This section provides an example of how to use each shift operator.

Example: sll (Shift Left Logical)

```
sig1 <= A(4 downto 0) sll 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & "00";
```

Example: srl (Shift Right Logical)

```
sig1 <= A(4 downto 0) srl 2
```

logically equivalent to:

```
sig1 <= "00" & A(4 downto 2);
```

Example: sla (Shift Left Arithmetic)

```
sig1 <= A(4 downto 0) sla 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & A(0) & A(0);
```

Example: sra (Shift Right Arithmetic)

```
sig1 <= A(4 downto 0) sra 2
```

logically equivalent to:

```
sig1 <= <= A(4) & A(4) & A(4 downto 2);
```

Example: rol (Rotate Left)

```
sig1 <= A(4 downto 0) rol 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & A(4 downto 3);
```

Example: ror (Rotate Right)

```
A(4 downto 0) ror 2
```

logically equivalent to:

```
sig1 <= A(1 downto 0) & A(4 downto 2);
```

Entity and Architecture Descriptions

A circuit description consists of two parts: the interface (defining the I/O ports) and the body. In VHDL, the entity corresponds to the interface and the architecture describes the behavior.

Entity Declaration

The I/O ports of the circuit are declared in the entity. Each port has a name, a mode (in, out, inout or buffer) and a type (ports A, B, C, D, E in the Example 6-1).

Note that types of ports must be constrained, and not more than one-dimensional array types are accepted as ports.

Architecture Declaration

Internal signals may be declared in the architecture. Each internal signal has a name and a type (signal T in Example 6-1).

Example 6-1 Entity and Architecture Declaration

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity EXAMPLE is
  port (
    A,B,C : in std_logic;
    D,E : out std_logic );
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is
  signal T : std_logic;
begin
...
end ARCHI;
```

Component Instantiation

Structural descriptions assemble several blocks and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are the component, the port and the signal. The component is the building or basic block. A port is a component I/O connector. A signal corresponds to a wire between components.

In VHDL, a component is represented by a design entity. This is actually a composite consisting of an entity declaration and an architecture body. The entity declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The architecture body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within component instantiation statements. These statements specify an instance of a component occurring inside an architecture of another component. Each component instantiation statement is labeled with an identifier. Besides naming a component declared in a local component declaration, a component instantiation statement contains an association list (the parenthesized list following the reserved word port map) that specifies which actual signals or ports are associated with which local ports of the component declaration.

Note: XST supports unconstrained vectors in component declarations.

Example 6-2 gives the structural description of a half adder composed of four nand2 components.

Example 6-2 Structural Description of a Half Adder

```
entity NAND2 is
  port (
    A,B : in BIT;
    Y : out BIT );
end NAND2;

architecture ARCHI of NAND2 is
begin
  Y <= A nand B;
end ARCHI;

entity HALFADDER is
  port (
    X,Y : in BIT;
    C,S : out BIT );
end HALFADDER;
```

```
architecture ARCHI of HALFADDER is
  component NAND2
    port (
      A,B : in BIT;
      Y : out BIT );
  end component;

  for all : NAND2 use entity work.NAND2(ARCHI);
  signal S1, S2, S3 : BIT;
  begin
    NANDA : NAND2 port map (X,Y,S3);
    NANDB : NAND2 port map (X,S3,S1);
    NANDC : NAND2 port map (S3,Y,S2);
    NANDD : NAND2 port map (S1,S2,S);
    C <= S3;
end ARCHI;
```

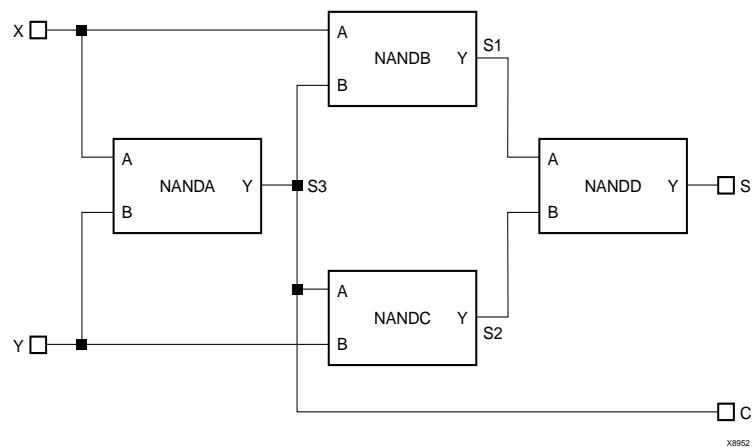The synthesized top level netlist is shown in the following figure.

Figure 6-1: **Synthesized Top Level Netlist**

Recursive Component Instantiation

XST supports recursive component instantiation (please note that direct instantiation is not supported for recursivity). Example 6-3 shows a 4-bit shift register description:

Example 6-3 4-bit shift register with Recursive Component Instantiation

```
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity single_stage is
  generic (sh_st: integer:=4);
  port (
    CLK : in std_logic;
    DI : in std_logic;
    DO : out std_logic );
end entity single_stage;
```

```
architecture recursive of single_stage is
  component single_stage
    generic (sh_st: integer);
    port (
      CLK : in std_logic;
      DI  : in std_logic;
      DO  : out std_logic );
  end component;

  signal tmp : std_logic;

  begin
    GEN_FD_LAST: if sh_st=1 generate
      inst_fd: FD port map (D=>DI, C=>CLK, Q=>DO);
    end generate;
    GEN_FD_INTERM: if sh_st /= 1 generate
      inst_fd: FD port map (D=>DI, C=>CLK, Q=>tmp);
      inst_sstage: single_stage generic map (sh_st => sh_st-1)
        port map (DI=>tmp, CLK=>CLK, DO=>DO);
    end generate;
end recursive;
```

Component Configuration

Associating an entity/architecture pair to a component instance provides the means of linking components with the appropriate model (entity/architecture pair). XST supports component configuration in the declarative part of the architecture:

```
for instantiation_list: component_name use
LibName.entity_Name(Architecture_Name);
```

Example 6-2, Structural Description of a Half Adder, shows how to use a configuration clause for component instantiation. The example contains the following "for all" statement:

```
for all : NAND2 use entity work.NAND2(ARCHI);
```

This statement indicates that all NAND2 components use the entity NAND2 and Architecture ARCHI.

Note: When the configuration clause is missing for a component instantiation, XST links the component to the entity with the same name (and same interface) and the selected architecture to the most recently compiled architecture. If no entity/architecture is found, a black box is generated during synthesis.

Generic Parameter Declaration

Generic parameters may be declared in the entity declaration part. XST supports all types for generics including integer, boolean, string, real, std_logic_vector, etc. An example of using generic parameters would be setting the width of the design. In VHDL, describing circuits with generic ports has the advantage that the same component can be repeatedly instantiated with different values of generic ports as shown in Example 6-4.

Example 6-4 Generic Instantiation of Components

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

entity addern is
  generic (width : integer := 8);
  port (
    A,B : in std_logic_vector (width-1 downto 0);
    Y   : out std_logic_vector (width-1 downto 0) );
end addern;

architecture bhv of addern is
  begin
    Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is
  port (
    X, Y, Z : in std_logic_vector (12 downto 0);
    A, B : in std_logic_vector (4 downto 0);
    S :out std_logic_vector (16 downto 0) );
end top;

architecture bhv of top is
  component addern
    generic (width : integer := 8);
    port (
      A,B : in std_logic_vector (width-1 downto 0);
      Y   : out std_logic_vector (width-1 downto 0) );
  end component;

  for all : addern use entity work.addern(bhv);
  signal C1 : std_logic_vector (12 downto 0);
  signal C2, C3 : std_logic_vector (16 downto 0);
  begin
    U1 : addern generic map (n=>13) port map (X,Y,C1);
    C2 <= C1 & A;
    C3 <= Z & B;
    U2 : addern generic map (n=>17) port map (C2,C3,S);
end bhv;
```

Generic/Attribute Conflicts

Since generics and attributes can be applied to both instances and components in your VHDL code, and attributes can also be specified in a constraints file, from time to time, conflicts will arise. To resolve these conflicts, XST uses the following rules of precedence.

1. Whatever is specified on an instance (lower level) takes precedence over what is specified on a component (higher level).

2. If a generic and an attribute are specified on either the same instance or the same component, the generic takes precedence, and XST issues a message warning of the conflict.

3. An attribute specified in the XCF file will always take precedence over attributes or generics specified in your VHDL code.

Note: When an attribute specified on an instance overrides a generic specified on a component in XST, it is possible that your simulation tool may nevertheless use the generic. This may cause the simulation results to not match the synthesis results.

Use the following matrix as a guide in determining precedence.

	Generic on an Instance	**Generic on a Component**
Attribute on an Instance	Apply Generic (XST issues warning message)	Apply Attribute (possible simulation mismatch)
Attribute on a Component	Apply Generic	Apply Generic (XST issues warning message)
Attribute in XCF	Apply Attribute (XST issues warning message)	Apply Attribute

Note: Security attributes on the block definition always have higher precedence than any other attribute or generic.

Combinatorial Circuits

The following subsections describe how XST uses various VHDL constructs for combinatorial circuits.

Concurrent Signal Assignments

Combinatorial logic may be described using concurrent signal assignments, which can be defined within the body of the architecture. VHDL offers three types of concurrent signal assignments: simple, selected and conditional. You can describe as many concurrent statements as needed; the order of concurrent signal definition in the architecture is irrelevant.

A concurrent assignment is made of two parts: left hand side, and right hand side. The assignment changes when any signal in the right part changes. In this case, the result is assigned to the signal on the left part.

Simple Signal Assignment

The following example shows a simple assignment.

T <= A and B;

Selected Signal Assignment

The following example shows a selected signal assignment.

Example 6-5 MUX Description Using Selected Signal Assignment

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

entity select_bhv is
  generic (width: integer := 8);
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0) );
end select_bhv;

architecture bhv of select_bhv is
begin
  with selector select
    T <= a when "00",
         b when "01",
         c when "10",
         d when others;
end bhv;
```

Conditional Signal Assignment

The following example shows a conditional signal assignment.

Example 6-6 MUX Description Using Conditional Signal Assignment

```
entity when_ent is
  generic (width: integer := 8);
  port (
    a, b, c, d : in std_logic_vector (width-1 downto 0);
    selector : in std_logic_vector (1 downto 0);
    T : out std_logic_vector (width-1 downto 0) );
end when_ent;

architecture bhv of when_ent is
  begin
    T <= a when selector = "00" else
         b when selector = "01" else
         c when selector = "10" else
         d;
end bhv;
```

Generate Statement

Repetitive structures are declared with the "generate" VHDL statement. For this purpose, "for I in 1 to N generate" means that the bit slice description is repeated N times. As an example, Example 6-7 gives a description of an 8-bit adder by declaring the bit slice structure.

Example 6-7 8 Bit Adder Described with a "for...generate" Statement

```
entity EXAMPLE is
  port (
    A,B  : in BIT_VECTOR (0 to 7);
    CIN  : in BIT;
    SUM  : out BIT_VECTOR (0 to 7);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  signal C : BIT_VECTOR (0 to 8);
  begin
    C(0) <= CIN;
    COUT <= C(8);
    LOOP_ADD : for I in 0 to 7 generate
    SUM(I) <= A(I) xor B(I) xor C(I);
    C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
end ARCHI;
```

The "if *condition* generate" statement is supported for static (non-dynamic) conditions. Example 6-8 shows such an example. It is a generic N-bit adder with a width ranging between 4 and 32.

Example 6-8 N Bit Adder Described with an "if...generate" and a "for… generate" Statement

```
entity EXAMPLE is
  generic (N : INTEGER := 8);
  port (
    A,B : in BIT_VECTOR (N downto 0);
    CIN : in BIT;
    SUM : out BIT_VECTOR (N downto 0);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  signal C : BIT_VECTOR (N+1 downto 0);
  begin
  L1: if (N>=4 and N<=32) generate
    C(0) <= CIN;
    COUT <= C(N+1);
    LOOP_ADD : for I in 0 to N generate
      SUM(I) <= A(I) xor B(I) xor C(I);
      C(I+1) <= (A(I) and B(I)) or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
  end generate;
end ARCHI;
```

Combinatorial Process

A process assigns values to signals differently than when using concurrent signal assignments. The value assignments are made in a sequential mode. The latest assignments may cancel previous ones. See Example 6-9. First the signal *S* is assigned to 0, but later on (for (A and B) =1), the value for *S* is changed to 1.

Example 6-9 Assignments in a Process

```
entity EXAMPLE is
  port (
    A, B : in BIT;
    S : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (A, B)
  begin
    S <= '0' ;
    if ((A and B) = '1') then
      S <= '1' ;
    end if;
  end process;
end ARCHI;
```

A process is called combinatorial when its inferred hardware does not involve any memory elements. Said differently, when all assigned signals in a process are always explicitly assigned in all paths of the Process statements, then the process in combinatorial.

A combinatorial process has a sensitivity list appearing within parentheses after the word "process". A process is activated if an event (value change) appears on one of the sensitivity list signals. For a combinatorial process, this sensitivity list must contain all signals which appear in conditions (if, case, etc.), and any signal appearing on the right hand side of an assignment.

If one or more signals are missing from the sensitivity list, XST generates a warning for the missing signals and adds them to the sensitivity list. In this case, the result of the synthesis may be different from the initial design specification.

A process may contain local variables. The variables are handled in a similar manner as signals (but are not, of course, outputs to the design).

In Example 6-10, a variable named AUX is declared in the declarative part of the process and is assigned to a value (with ":=") in the statement part of the process. Examples 6-10 and 6-11 are two examples of a VHDL design using combinatorial processes.

Example 6-10 Combinatorial Process

```
library ASYL;
use ASYL.ARITH.all;

entity ADDSUB is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    ADD_SUB : in BIT;
    S : out BIT_VECTOR (3 downto 0));
end ADDSUB;

architecture ARCHI of ADDSUB is
  begin
    process (A, B, ADD_SUB)
      variable AUX : BIT_VECTOR (3 downto 0);
```

```
      begin
        if ADD_SUB = '1' then
          AUX := A + B ;
        else
          AUX := A - B ;
        end if;
        S <= AUX;
      end process;
    end ARCHI;
```

Example 6-11 Combinatorial Process

```
entity EXAMPLE is
  port (
    A, B : in BIT;
    S : out BIT  );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (A,B)
    variable X, Y : BIT;
  begin
    X := A and B;
    Y := B and A;
    if X = Y then
      S <= '1' ;
    end if;
  end process;
end ARCHI;
```

Note: In combinatorial processes, if a signal is not explicitly assigned in all branches of "if" or "case" statements, XST generates a latch to hold the last value. To avoid latch creation, ensure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the Process statements.

Different statements can be used in a process:

- Variable and signal assignment
- If statement
- Case statement
- For...Loop statement
- Function and procedure call

The following sections provide examples of each of these statements.

If...Else Statement

If...else statements use true/false conditions to execute statements. If the expression evaluates to true, the first statement is executed. If the expression evaluates to false (or x or z), the Else statement is executed. A block of multiple statements may be executed using begin and end keywords. If... else statements may be nested. Example 6-12 shows the use of an If...else statement.

Example 6-12 MUX Description Using If...Else Statement

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

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel1, sel2 : in std_logic;
    outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
  process (a, b, c, d, sel1, sel2)
  begin
    if (sel1 = '1') then
      if (sel2 = '1') then
        outmux <= a;
      else
        outmux <= b;
      end if;
    else
      if (sel2 = '1') then
        outmux <= c;
      else
        outmux <= d;
      end if;
    end if;
  end process;
end behavior;
```

Case Statement

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The Case statement evaluates the branches in the order they are written; the first branch that evaluates to true is executed. If none of the branches match, the default branch is executed. Example 6-13 shows the use of a Case statement.

Example 6-13 MUX Description Using the Case Statement

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

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel : in std_logic_vector (1 downto 0);
    outmux : out std_logic_vector (7 downto 0));
end mux4;
```

```
architecture behavior of mux4 is
begin
  process (a, b, c, d, sel)
  begin
    case sel is
      when "00" => outmux <= a;
      when "01" => outmux <= b;
      when "10" => outmux <= c;
      when others => outmux <= d;  -- case statement must be complete
    end case;
  end process;
end behavior;
```

For...Loop Statement

The For statement is supported for:

- Constant bounds

- Stop test condition using operators <, <=, > or >=

- Next step computation falling within one of the following specifications:

  - *var* = *var* + step

  - *var* = *var* - step

  (where *var* is the loop variable and *step* is a constant value)

- Next and Exit statements are supported

Example 6-14 shows the use of a For...loop statement.

Example 6-14 For...Loop Description

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

entity countzeros is
  port (
    a : in std_logic_vector (7 downto 0);
    Count : out std_logic_vector (2 downto 0) );
end mux4;

architecture behavior of mux4 is
  signal Count_Aux: std_logic_vector (2 downto 0);
  begin
    process (a)
    begin
      Count_Aux <= "000";
      for i in a'range loop
        if (a[i] = '0') then
          Count_Aux <= Count_Aux + 1; -- operator "+" defined
                                       -- in std_logic_unsigned
        end if;
      end loop;
      Count <= Count_Aux;
  end process;
end behavior;
```

Sequential Circuits

Sequential circuits can be described using sequential processes. The following two types of descriptions are allowed by XST:

- sequential processes with a sensitivity list
- sequential processes without a sensitivity list

Sequential Process with a Sensitivity List

A process is sequential when it is not a combinatorial process. In other words, a process is sequential when some assigned signals are not explicitly assigned in all paths of the statements. In this case, the hardware generated has an internal state or memory (flip-flops or latches).

Example 6-15 provides a template for describing sequential circuits. Also refer to the chapter describing macro inference for additional details (registers, counters, etc.).

Example 6-15 Sequential Process with Asynchronous, Synchronous Parts

```
process (CLK, RST)  ...
begin
  if RST = <'0' | '1'> then
    -- an asynchronous part may appear here
    -- optional part
    .......
  elsif <CLK'EVENT | not CLK'STABLE>
    and CLK = <'0' | '1'> then
    -- synchronous part
    -- sequential statements may appear here
  end if;
end process;
```

Note: Asynchronous signals must be declared in the sensitivity list. Otherwise, XST generates a warning and adds them to the sensitivity list. In this case, the behavior of the synthesis result may be different from the initial specification.

Sequential Process without a Sensitivity List

Sequential processes without a sensitivity list must contain a Wait statement. The Wait statement must be the first statement of the process. The condition in the Wait statement must be a condition on the clock signal. Several Wait statements in the same process are accepted, but a set of specific conditions must be respected. See "Multiple Wait Statements Descriptions" for details. An asynchronous part cannot be specified within processes without a sensitivity list.

Example 6-16 shows the skeleton of such a process. The clock condition may be a falling or a rising edge.

Example 6-16 Sequential Process Without a Sensitivity List

```
process ...
begin
  wait until <CLK'EVENT | not CLK' STABLE> and CLK = <'0' | '1'>;
    ... -- a synchronous part may be specified here.
end process;
```

Note: XST does not support clock and clock enable descriptions within the same Wait statement. Instead, code these descriptions as in Example 6-17.

Note: XST does not support Wait statements for latch descriptions.

Example 6-17 Clock and Clock Enable

Not supported:

```
wait until CLOCK'event and CLOCK = '0' and ENABLE = '1' ;
```

Supported:

```
wait until CLOCK'event and CLOCK = '0' ;
  if ENABLE = '1' then ...
```

Examples of Register and Counter Descriptions

Example 6-18 describes an 8-bit register using a process with a sensitivity list. Example 6-19 describes the same example using a process without a sensitivity list containing a Wait statement.

Example 6-18 8 bit Register Description Using a Process with a Sensitivity List

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK)
  begin
    if CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

Example 6-19 8 bit Register Description Using a Process without a Sensitivity List

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    DO  : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is
begin
  process begin
    wait until CLK'EVENT and CLK = '1';
    DO <= DI;
  end process;
end ARCHI;
```

Example 6-20 describes an 8-bit register with a clock signal and an asynchronous reset signal.

Example 6-20 8 bit Register Description Using a Process with a Sensitivity List

```
entity EXAMPLE is
  port (
    DI  : in BIT_VECTOR (7 downto 0);
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK, RST)
  begin
    if RST = '1' then
      DO <= "00000000";
    elsif CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

Example 6-21 8 bit Counter Description Using a Process with a Sensitivity List

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is
  port (
    CLK : in BIT;
    RST : in BIT;
    DO  : out BIT_VECTOR (7 downto 0));
end EXAMPLE;

architecture ARCHI of EXAMPLE is
begin
  process (CLK, RST)
    variable COUNT : BIT_VECTOR (7 downto 0);
  begin
    if RST = '1' then
      COUNT := "00000000";
    elsif CLK'EVENT and CLK = '1' then
      COUNT := COUNT + "00000001";
    end if;
    DO <= COUNT;
  end process;
end ARCHI;
```

Multiple Wait Statements Descriptions

Sequential circuits can be described with multiple Wait statements in a process. When using XST, several rules must be respected to use multiple Wait statements. These rules are as follows:

- The process must only contain one Loop statement.

- The first statement in the loop must be a Wait statement.

- After each Wait statement, a Next or Exit statement must be defined.

- The condition in the Wait statements must be the same for each Wait statement.

- This condition must use only one signal — the clock signal.

- This condition must have the following form:

```
"wait [on clock_signal] until [(clock_signal'EVENT |
    not clock_signal'STABLE) and ] clock_signal = {'0' | '1'};"
```

Example 6-22 uses multiple Wait statements. This example describes a sequential circuit performing four different operations in sequence. The design cycle is delimited by two successive rising edges of the clock signal. A synchronous reset is defined providing a way to restart the sequence of operations at the beginning. The sequence of operations consists of assigning each of the four inputs: DATA1, DATA2, DATA3 and DATA4 to the output RESULT.

Example 6-22 Sequential Circuit Using Multiple Wait Statements

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
  port (
    DATA1, DATA2, DATA3, DATA4 : in STD_LOGIC_VECTOR (3 downto 0);
    RESULT : out STD_LOGIC_VECTOR (3 downto 0);
    CLK : in STD_LOGIC;
    RST : in STD_LOGIC );
end EXAMPLE;

architecture ARCH of EXAMPLE is
begin
  process begin
    SEQ_LOOP : loop
      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA1;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA2;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA3;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA4;
    end loop;
  end process;
end ARCH;
```

Functions and Procedures

The declaration of a function or a procedure provides a mechanism for handling blocks used multiple times in a design. Functions and procedures can be declared in the declarative part of an entity, in an architecture or in packages. The heading part contains the parameters: input parameters for functions and input, output and inout parameters for procedures. These parameters can be unconstrained. This means that they are not constrained to a given bound. The content is similar to the combinatorial process content.

Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package.

Example 6-23 shows a function declared within a package. The "ADD" function declared here is a single bit adder. This function is called 4 times with the proper parameters in the architecture to create a 4-bit adder. The same example described using a procedure is shown in Example 6-24.

Example 6-23 Function Declaration and Function Call

```
package PKG is
  function ADD (A,B, CIN : BIT )
  return BIT_VECTOR;
end PKG;

package body PKG is
  function ADD (A,B, CIN : BIT )
  return BIT_VECTOR is
    variable S, COUT : BIT;
    variable RESULT : BIT_VECTOR (1 downto 0);
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    RESULT := COUT & S;
    return RESULT;
  end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
 signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
 begin
  S0 <= ADD (A(0), B(0), CIN);
  S1 <= ADD (A(1), B(1), S0(1));
  S2 <= ADD (A(2), B(2), S1(1));
  S3 <= ADD (A(3), B(3), S2(1));
  S  <= S3(0) & S2(0) & S1(0) & S0(0);
  COUT <= S3(1);
end ARCHI;
```

Example 6-24 Procedure Declaration and Procedure Call

```
package PKG is
 procedure ADD (
  A,B, CIN : in BIT;
  C : out BIT_VECTOR (1 downto 0) );
end PKG;

package body PKG is
  procedure ADD (
    A,B, CIN : in BIT;
    C : out BIT_VECTOR (1 downto 0)
    ) is
    variable S, COUT : BIT;
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C := COUT & S;
  end ADD;
end PKG;

use work.PKG.all;

entity EXAMPLE is
  port (
    A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT );
end EXAMPLE;

architecture ARCHI of EXAMPLE is
  begin
  process (A,B,CIN)
    variable S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
  begin
    ADD (A(0), B(0), CIN, S0);
    ADD (A(1), B(1), S0(1), S1);
    ADD (A(2), B(2), S1(1), S2);
    ADD (A(3), B(3), S2(1), S3);
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3(1);
  end process;
end ARCHI;
```

XST supports recursive functions as well. Example 6-25 represents n! function.

Example 6-25 Recursive Function

```
function my_func(x : integer) return integer is
  begin
    if x = 1 then
      return x;
    else
      return (x*my_func(x-1));
    end if;
end function my_func;
```

Assert Statement

XST supports the use of the Assert statement. By using the Assert statement, designers can detect undesirable conditions in their VHDL designs such as bad values for generics, constants and generate conditions, or bad values for parameters in called functions. For any failed condition in an Assert statement, XST, according to the severity level, generates a warning message with the reason for the warning, or rejects the design and generates an error message and the reason for the rejection.

Note: XST supports the Assert statement only with static condition.

The following example contains a block, SINGLE_SRL, that describes a shift register. The size of the shift register depends on the SRL_WIDTH generic value. The Assert statement ensures that the implementation of a single shift register does not exceed the size of a single SRL.

Since the size of the SRL is 16 bit, and XST implements the last stage of the shift register using a flip-flop in a slice, then the maximum size of the shift register cannot exceed 17 bits. The SINGLE_SRL block is instantiated twice in the entity named TOP, the first time with SRL_WIDTH equal to 13, and the second time with SRL_WIDTH equal to 18:

```
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
  generic (SRL_WIDTH : integer := 16);
  port (
    clk : in std_logic;
    inp : in std_logic;
    outp : out std_logic);
end SINGLE_SRL;

architecture beh of SINGLE_SRL is
  signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
begin

  assert SRL_WIDTH <= 17
  report "The size of Shift Register exceeds the size of a single SRL"
  severity FAILURE;

  process (clk)
  begin
    if (clk'event and clk = '1') then
      shift_reg <= shift_reg (SRL_WIDTH-1 downto 1) & inp;
    end if;
    end process;
  outp <= shift_reg(SRL_WIDTH-1);
end beh;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
  port (
    clk : in std_logic;
    inp1, inp2 : in std_logic;
    outp1, outp2 : out std_logic);
end TOP;

architecture beh of TOP is
  component SINGLE_SRL is
  generic (SRL_WIDTH : integer := 16);
  port(
    clk : in std_logic;
    inp : in std_logic;
    outp : out std_logic);
  end component;
begin
  inst1: SINGLE_SRL generic map (SRL_WIDTH => 13)
    port map(
      clk => clk,
      inp => inp1,
      outp => outp1 );
  inst2: SINGLE_SRL generic map (SRL_WIDTH => 18)
    port map(
      clk => clk,
      inp => inp2,
      outp => outp2 );
end beh;
```

Running this example through XST results in the following error message generated by the Assert statement.

```
...
=====================================================================
*                          HDL Analysis                             *
=====================================================================
Analyzing Entity <top> (Architecture <beh>).
Entity <top> analyzed. Unit <top> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 13
Entity <single_srl> analyzed. Unit <single_srl> generated.

Analyzing generic Entity <single_srl> (Architecture <beh>).
  SRL_WIDTH = 18
ERROR:Xst - assert_1.vhd line 15: FAILURE: The size of Shift Register
exceeds the size of a single SRL
...
```

Packages

VHDL models may be defined using packages. Packages contain type and subtype declarations, constant definitions, function and procedure definitions, and component declarations.

This mechanism provides the ability to change parameters and constants of the design (for example, constant values, function definitions). Packages may contain two declarative parts: package declaration and body declaration. The body declaration includes the description of function bodies declared in the package declaration.

XST provides full support for packages. To use a given package, the following lines must be included at the beginning of the VHDL design:

```
library lib_pack;
-- lib_pack is the name of the library specified
-- where the package has been compiled (work by default)
use lib_pack.pack_name.all;
-- pack_name is the name of the defined package.
```

XST also supports predefined packages; these packages are pre-compiled and can be included in VHDL designs. These packages are intended for use during synthesis, but may also be used for simulation.

STANDARD Package

The Standard package contains basic types (bit, bit_vector, and integer). The STANDARD package is included by default.

IEEE Packages

The following IEEE packages are supported.

- std_logic_1164: defines types std_logic, std_ulogic, std_logic_vector, std_ulogic_vector, and conversion functions based on these types.

- numeric_bit: supports types unsigned, signed vectors based on type bit, and all overloaded arithmetic operators on these types. It also defines conversion and extended functions for these types.

- numeric_std: supports types unsigned, signed vectors based on type std_logic. This package is equivalent to std_logic_arith.

- math_real: supports the following.

  - ♦ Real number constants as shown in the following table:

Constant	Value	Constant	Value
math_e	e	math_log_of_2	ln2
math_1_over_e	$1/e$	math_log_of_10	ln10
math_pi	π	math_log2_of_e	$\log_2 e$
math_2_pi	2π	math_log10_of_e	$\log_{10} e$
math_1_over_pi	$1/\pi$	math_sqrt_2	$\sqrt{2}$
math_pi_over_2	$\pi/2$	math_1_oversqrt_2	$1/\sqrt{2}$

Constant	Value	Constant	Value
math_pi_over_3	$\pi/3$	math_sqrt_pi	$\sqrt{\pi}$
math_pi_over_4	$\pi/4$	math_deg_to_rad	$2\pi/360$
math_3_pi_over_2	$3\pi/2$	math_rad_to_deg	$360/2\pi$

♦ Real number functions as shown in the following table:

ceil(x)	realmax(x,y)	exp(x)	cos(x)	cosh(x)
floor(x)	realmin(x,y)	log(x)	tan(x)	tanh(x)
round(x)	sqrt(x)	log2(x)	arcsin(x)	arcsinh(x)
trunc(x)	cbrt(x)	log10(x)	arctan(x)	arccosh(x)
sign(x)	"**"(n,y)	log(x,y)	arctan(y,x)	arctanh(x)
"mod"(x,y)	"**"(x,y)	sin(x)	sinh(x)	

♦ The procedure *uniform*, which generates successive values between 0.0 and 1.0.

Note: Functions and procedures in the math_real package, as well as the real type, are for calculations only. They are not supported for synthesis in XST.

Example:

```
library ieee;
use IEEE.std_logic_signed.all;
signal a, b, c : std_logic_vector (5 downto 0);
c <= a + b;
-- this operator "+" is defined in package std_logic_signed.
-- Operands are converted to signed vectors, and function "+"
-- defined in package std_logic_arith is called with signed
-- operands.
```

Synopsys Packages

The following Synopsys packages are supported in the IEEE library.

- std_logic_arith: supports types unsigned, signed vectors, and all overloaded arithmetic operators on these types. It also defines conversion and extended functions for these types.
- std_logic_unsigned: defines arithmetic operators on std_ulogic_vector and considers them as unsigned operators.
- std_logic_signed: defines arithmetic operators on std_logic_vector and considers them as signed operators.
- std_logic_misc: defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package (and_reduce, or_reduce,...).

VHDL Language Support

The following tables indicate which VHDL constructs are supported in XST. For more information about these constructs, refer to the sections following the tables.

Table 6-3: **Design Entities and Configurations**

Entity Header	Generics	Supported (integer type only)
	Ports	Supported (no unconstrained ports)
	Entity Declarative Part	Supported
	Entity Statement Part	Unsupported
Architecture Bodies	Architecture Declarative Part	Supported
	Architecture Statement Part	Supported
Configuration Declarations	Block Configuration	Supported
	Component Configuration	Supported
Subprograms	Functions	Supported
	Procedures	Supported

Table 6-3: **Design Entities and Configurations**

Packages	STANDARD	Type TIME is not supported
	TEXTIO	Supported
	STD_LOGIC_1164	Supported
	STD_LOGIC_ARITH	Supported
	STD_LOGIC_SIGNED	Supported
	STD_LOGIC_UNSIGNED	Supported
	STD_LOGIC_MISC	Supported
	NUMERIC_BIT	Supported
	NUMERIC_EXTRA	Supported
	NUMERIC_SIGNED	Supported
	NUMERIC_UNSIGNED	Supported
	NUMERIC_STD	Supported
	MATH_REAL	Supported
	ASYL.ARITH	Supported
	ASYL.SL_ARITH	Supported
	ASYL.PKG_RTL	Supported
	ASYL.ASYL1164	Supported
Enumeration Types	BOOLEAN, BIT	Supported
	STD_ULOGIC, STD_LOGIC	Supported
	XO1, UX01, XO1Z, UX01Z	Supported
	Character	Supported
Integer Types	INTEGER	Supported
	POSITIVE	Supported
	NATURAL	Supported
Physical Types	TIME	Ignored
	REAL	Supported (only in functions for constant calculations)

Table 6-3: **Design Entities and Configurations**

Composite	BIT_VECTOR	Supported
	STD_ULOGIC_VECTOR	Supported
	STD_LOGIC_VECTOR	Supported
	UNSIGNED	Supported
	SIGNED	Supported
	Record	Supported
	Access	Supported
	File	Supported

Table 6-4: **Mode**

In, Out, Inout	Supported
Buffer	Supported
Linkage	Unsupported

Table 6-5: **Declarations**

Type	Supported for enumerated types, types with positive range having constant bounds, bit vector types, and multi-dimensional arrays
Subtype	Supported

Table 6-6: **Objects**

Constant Declaration	Supported (deferred constants are not supported)
Signal Declaration	Supported ("register" or "bus" type signals are not supported)
Variable Declaration	Supported
File Declaration	Supported
Alias Declaration	Supported
Attribute Declaration	Supported for some attributes, otherwise skipped (see Chapter 5, "Design Constraints")
Component Declaration	Supported

Table 6-7: **Specifications**

Attribute	Only supported for some predefined attributes: HIGH, LOW, LEFT, RIGHT, RANGE, REVERSE_RANGE, LENGTH, POS, ASCENDING, EVENT, LAST_VALUE. Otherwise, ignored.
Configuration	Supported only with the "all" clause for instances list. If no clause is added, XST looks for the entity/architecture compiled in the default library.
Disconnection	Unsupported

Table 6-8: **Names**

Simple Names	Supported
Selected Names	Supported
Indexed Names	Supported
Slice Names	Supported (including dynamic ranges)

Note: XST does not allow underscores as the first character of signal names (for example, _DATA_1).

Table 6-9: **Expressions**

Operators	Logical Operators: and, or, nand, nor, xor, xnor, not	Supported
	Relational Operators: =, /=, <, <=, >, >=	Supported
	& (concatenation)	Supported
	Adding Operators: +, -	Supported
	*	Supported
	/,rem	Supported if the right operand is a constant power of 2
	mod	Supported
	Shift Operators: sll, srl, sla, sra, rol, ror	Supported
	abs	Supported
	**	Only supported if the left operand is 2
	Sign: +, -	Supported

Table 6-9: **Expressions**

Operands	Abstract Literals	Only integer literals are supported
	Physical Literals	Ignored
	Enumeration Literals	Supported
	String Literals	Supported
	Bit String Literals	Supported
	Record Aggregates	Supported
	Array Aggregates	Supported
	Function Call	Supported
	Qualified Expressions	Supported for accepted predefined attributes
	Types Conversions	Supported
	Allocators	Unsupported
	Static Expressions	Supported

Table 6-10: **Supported VHDL Statements**

Wait Statement	Wait on *sensitivity_list* until *Boolean_expression.* See "Sequential Circuits" for details.	Supported with one signal in the sensitivity list and in the Boolean expression. In case of multiple Wait statements, the sensitivity list and the Boolean expression must be the same for each Wait statement. ***Note:*** XST does not support Wait statements for latch descriptions.
	Wait for t*ime_expression...* See "Sequential Circuits" for details.	Unsupported
	Assertion Statement	Supported (only for static conditions)
	Signal Assignment Statement	Supported (delay is ignored)
	Variable Assignment Statement	Supported
	Procedure Call Statement	Supported
	If Statement	Supported
	Case Statement	Supported

Table 6-10: **Supported VHDL Statements**

Loop Statement	"for... loop... end loop"	Supported for constant bounds only
	"while ... loop ... end loop"	Supported
	"loop ... end loop"	Only supported in the particular case of multiple Wait statements
	Next Statement	Supported
	Exit Statement	Supported
	Return Statement	Supported
	Null Statement	Supported
Concurrent Statement	Process Statement	Supported
	Concurrent Procedure Call	Supported
	Concurrent Assertion Statement	Ignored
	Concurrent Signal Assignment Statement	Supported (no "after" clause, no "transport" or "guarded" options, no waveforms)
	Component Instantiation Statement	Supported
	"For ... Generate"	Statement supported for constant bounds only
	"If ... Generate"	Statement supported for static condition only

VHDL Reserved Words

The following table shows the VHDL reserved words.

abs	configuration	impure	null	rem	type
access	constant	in	of	report	unaffected
after	disconnect	inertial	on	return	units
alias	downto	inout	open	rol	until
all	else	is	or	ror	use
and	elsif	label	others	select	variable
architecture	end	library	out	severity	wait
array	entity	linkage	package	signal	when
assert	exit	literal	port	shared	while
attribute	file	loop	postponed	sla	with
begin	for	map	procedure	sll	xnor
block	function	mod	process	sra	xor
body	generate	nand	pure	srl	
buffer	generic	new	range	subtype	
bus	group	next	record	then	
case	guarded	nor	register	to	
component	if	not	reject	transport	

Chapter 7

Verilog Language Support

This chapter contains the following sections.

- *"Introduction"*
- *"Behavioral Verilog Features"*
- *"Variable Part Selects"*
- *"Structural Verilog Features"*
- *"Parameters"*
- *"Parameter/Attribute Conflicts"*
- *"Verilog Limitations in XST"*
- *"Verilog Meta Comments"*
- *"Verilog Language Support Tables"*
- *"Primitives"*
- *"Verilog Reserved Keywords"*
- *"Verilog-2001 Support in XST"*

For detailed information about Verilog design constraints and options, refer to Chapter 5, "Design Constraints". For information about the Verilog attribute syntax, see "Verilog Meta Comment Syntax" in Chapter 5.

For information on setting Verilog options in the Process window of Project Navigator, refer to "General Constraints" in Chapter 5.

Introduction

Complex circuits are commonly designed using a top down methodology. Various specification levels are required at each stage of the design process. As an example, at the architectural level, a specification may correspond to a block diagram or an Algorithmic State Machine (ASM) chart. A block or ASM stage corresponds to a register transfer block (for example register, adder, counter, multiplexer, glue logic, finite state machine) where the connections are N-bit wires. Use of an HDL language like Verilog allows expressing notations such as ASM charts and circuit diagrams in a computer language. Verilog provides both behavioral and structural language structures which allow expressing design objects at high and low levels of abstraction. Designing hardware with a language like Verilog allows usage of software concepts such as parallel processing and object-oriented programming. Verilog has a syntax similar to C and Pascal, and is supported by XST as IEEE 1364.

The Verilog support in XST provides an efficient way to describe both the global circuit and each block according to the most efficient "style." Synthesis is then performed with the best synthesis flow for each block. Synthesis in this context is the compilation of high-level

behavioral and structural Verilog HDL statements into a flattened gate-level netlist, which can then be used to custom program a programmable logic device such as the Virtex™ FPGA family. Different synthesis methods are used for arithmetic blocks, glue logic, and finite state machines.

This manual assumes that you are familiar with the basic notions of Verilog. Please refer to the IEEE Verilog HDL Reference Manual for a complete specification.

Behavioral Verilog Features

This section contains descriptions of the behavioral features of Verilog.

Variable Declaration

Variables in Verilog may be declared as integers or real. These declarations are intended only for use in test code. Verilog provides data types such as reg and wire for actual hardware description.

The difference between reg and wire is whether the variable is given its value in a procedural block (reg) or in a continuous assignment (wire) Verilog code. Both reg and wire have a default width being one bit wide (scalar). To specify an N-bit width (vectors) for a declared reg or wire, the left and right bit positions are defined in square brackets separated by a colon. In Verilog-2001, both reg and wire data types can be signed or unsigned.

Example:

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

where arb_request[31] is the MSB and arb_request[0] is the LSB.

Initial Values

In Verilog-2001, you can initialize registers when you declare them.

The value:

• Must be a constant.

• Cannot depend on earlier initial values.

• Cannot be a function or task call.

• Can be a parameter value propagated to the register.

When you give a register an initial value in a declaration, XST sets this value on the output of the register at global reset, or at power up. A value assigned this way is carried in the NGC file as an INIT attribute on the register, and is independent of any local reset.

Example:

```
reg arb_onebit = 1'b0;
reg [3:0] arb_priority = 4'b1011;
```

You can also assign a set/reset (initial) value to a register via your behavioral Verilog code. Do this by assigning a value to a register when the register's reset line goes to the appropriate value as in the following example.

Example:

```
always @(posedge clk)
  begin
    if (rst)
      arb_onebit <= 1'b0;
  end
end
```

When you set the initial value of a variable in the behavioral code, it is implemented in the design as a flip-flop whose output can be controlled by a local reset; as such it is carried in the NGC file as an FDP or FDC flip-flop.

Local Reset ≠ Global Reset

Note that local reset is independent of global reset. Registers controlled by a local reset may be set to a different value than ones whose value is only reset at global reset (power up). In the following example, the register, arb_onebit, is set to '0' at global reset, but a pulse on the local reset (rst) can change its value to '1'.

Example:

```
module mult(clk, rst, A_IN, B_OUT);
  input clk,rst,A_IN;
  output B_OUT;

  reg arb_onebit = 1'b0;

  always @(posedge clk or posedge rst)
    begin
      if (rst)
        arb_onebit <= 1'b1;
      else
        arb_onebit <= A_IN;
      end
  end
B_OUT <= arb_onebit;
endmodule
```

This sets the set/reset value on the register's output at initial power up, but since this is dependent upon a local reset, the value changes whenever the local set/reset is activated.

Arrays

Verilog allows arrays of reg and wires to be defined as in the following two examples:

```
reg [3:0] mem_array [31:0];
```

The above describes an array of 32 elements each, 4 bits wide which can be assigned via behavioral Verilog code.

```
wire [7:0] mem_array [63:0];
```

The above describes an array of 64 elements each 8 bits wide which can only be assigned via structural Verilog code.

Multi-dimensional Arrays

XST supports multi-dimensional array types of up to three dimensions. Multi-dimensional arrays can be any net or any variable data type. You can code assignments and arithmetic operations with arrays, but you cannot select more than one element of an array at one

time. You cannot pass multi-dimensional arrays to system tasks or functions, or regular tasks or functions.

Examples

The following describes an array of 256 x 16 wire elements each 8 bits wide, which can only be assigned via structural Verilog code.

```
wire [7:0] array2 [0:255][0:15];
```

The following describes an array of 256 x 8 register elements, each 64 bits wide, which can be assigned via behavioral Verilog code.

```
reg [63:0] regarray2 [255:0][7:0];
```

The following is a three dimensional array. It can be described as an array of 15 arrays of 256 x 16 wire elements, each 8 bits wide, which can be assigned via structural Verilog code.

```
wire [7:0] array3 [0:15][0:255][0:15];
```

Data Types

The Verilog representation of the bit data type contains the following four values:

- 0: logic zero
- 1: logic one
- x: unknown logic value
- z: high impedance

XST includes support for the following Verilog data types:

- Net: wire, tri, triand/wand, trior/wor
- Registers: reg, integer
- Supply nets: supply0, supply1
- Constants: parameter
- Multi-Dimensional Arrays (Memories)

Net and registers can be either single bit (scalar) or multiple bit (vectors).

The following example gives some examples of Verilog data types (as found in the declaration section of a Verilog module).

Example 7-1 Basic Data Types

```
wire net1;                      // single bit net
reg r1;                         // single bit register
tri [7:0] bus1;                 // 8 bit tristate bus
reg [15:0] bus1;                // 15 bit register
reg [7:0] mem[0:127];           // 8x128 memory register
parameter state1 = 3'b001;      // 3 bit constant
parameter component = "TMS380C16";  // string
```

Legal Statements

The following are statements that are legal in behavioral Verilog.

Variable and signal assignment:

- Variable = expression
- if (condition) statement
- if (condition) statement else statement
- case (expression)

  expression: statement

   …

  default: statement

  endcase
- for (variable = expression; condition; variable = variable + expression) statement
- while (condition) statement
- forever statement
- functions and tasks

  Note: All variables are declared as integer or reg. A variable cannot be declared as a wire.

Expressions

An expression involves constants and variables with arithmetic (+, -, *,**, /,%), logical (&, &&, |, ||, ^, ~,~^, ^~, <<, >>,<<<,>>>), relational (<, ==, ===, <=, >=,!=,!==, >), and conditional (?) operators. The logical operators are further divided as bit-wise versus logical depending on whether it is applied to an expression involving several bits or a single bit. The following table lists the expressions supported by XST.

Table 7-1: **Expressions**

Concatenation	{}	Supported
Replication	{{}}	Supported
Arithmetic	+, -, *,**	Supported
	/	Supported only if second operand is a power of 2
Modulus	%	Supported only if second operand is a power of 2
Addition	+	Supported
Subtraction	-	Supported
Multiplication	*	Supported

Table 7-1: **Expressions**

Power	**	Supported • Both operands must be constants with the second operand being non-negative. • If the first operand is a 2, then the second operand may be a variable. • XST does not support the real data type. Any combination of operands that results in a real type causes an error. • The values X (unknown) and Z (high impedance) are not allowed.
Division	/	Supported XST generates incorrect logic for the division operator between signed and unsigned constants. **Example:** `-1235/3'b111`
Relational	>, <, >=, <=	Supported
Logical Negation	!	Supported
Logical AND	&&	Supported
Logical OR	||	Supported
Logical Equality	==	Supported
Logical Inequality	!=	Supported
Case Equality	===	Supported
Case Inequality	!==	Supported
Bitwise Negation	~	Supported
Bitwise AND	&	Supported
Bitwise Inclusive OR	|	Supported
Bitwise Exclusive OR	^	Supported
Bitwise Equivalence	~^, ^~	Supported
Reduction AND	&	Supported
Reduction NAND	~&	Supported
Reduction OR	|	Supported
Reduction NOR	~|	Supported
Reduction XOR	^	Supported

Table 7-1: **Expressions**

Reduction XNOR	~^, ^~	Supported
Left Shift	<<	Supported
Right Shift Signed	>>>	Supported
Left Shift Signed	<<<	Supported
Right Shift	>>	Supported
Conditional	?:	Supported
Event OR	or, ','	Supported

The following table lists the results of evaluating expressions using the more frequently used operators supported by XST.

Note: The (===) and (!==) are special comparison operators useful in simulations to check if a variable is assigned a value of (x) or (z). They are treated as (==) or (!=) in synthesis.

Table 7-2: **Results of Evaluating Expressions**

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a|b	a||b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Blocks

Block statements are used to group statements together. XST only supports sequential blocks. Within these blocks, the statements are executed in the order listed. Parallel blocks are not supported by XST. Block statements are designated by **begin** and **end** keywords, and are discussed within examples later in this chapter.

Modules

In Verilog a design component is represented by a module. The connections between components are specified within module instantiation statements. Such a statement specifies an instance of a module. Each module instantiation statement must be given a name (instance name). In addition to the name, a module instantiation statement contains an association list that specifies which actual nets or ports are associated with which local ports (formals) of the module declaration.

All procedural statements occur in blocks that are defined inside modules. There are two kinds of procedural blocks: the initial block and the always block. Within each block, Verilog uses a begin and end to enclose the statements. Since initial blocks are ignored during synthesis, only always blocks are discussed. Always blocks usually take the following format:

```
always
  begin
  statement
  …...
end
```

where each statement is a procedural assignment line terminated by a semicolon.

Module Declaration

In the module declaration, the I/O ports of the circuit are declared. Each port has a name and a mode (in, out, and inout) as shown in the example below.

```
module EXAMPLE (A, B, C, D, E);
  input A, B, C;
  output D;
  inout E;
  wire D, E;
  ...
  assign E = oe ? A : 1'bz;
  assign D = B & E;
  ...
endmodule
```

The input and output ports defined in the module declaration called EXAMPLE are the basic input and output I/O signals for the design. The inout port in Verilog is analogous to a bi-directional I/O pin on the device with the data flow for output versus input being controlled by the enable signal to the tristate buffer. The preceding example describes E as a tristate buffer with a high-true output enable signal. If oe = 1, the value of signal A is output on the pin represented by E. If oe = 0, then the buffer is in high impedance (Z) and any input value driven on the pin E (from the external logic) is brought into the device and fed to the signal represented by D.

Verilog Assignments

There are two forms of assignment statements in the Verilog language:

- Continuous Assignments
- Procedural Assignments

Continuous Assignments

Continuous assignments are used to model combinatorial logic in a concise way. Both explicit and implicit continuous assignments are supported. Explicit continuous assignments are introduced by the `assign` keyword after the net has been separately declared. Implicit continuous assignments combine declaration and assignment.

Note: Delays and strengths given to a continuous assignment are ignored by XST.

Example of an explicit continuous assignment:

```
wire par_eq_1;
…...
assign par_eq_1 = select ? b : a;
```

Example of an implicit continuous assignment:

```
wire temp_hold = a | b;
```

Note: Continuous assignments are only allowed on wire and tri data types.

Procedural Assignments

Procedural assignments are used to assign values to variables declared as regs and are introduced by always blocks, tasks, and functions. Procedural assignments are usually used to model registers and FSMs.

XST includes support for combinatorial functions, combinatorial and sequential tasks, and combinatorial and sequential always blocks.

Combinatorial Always Blocks

Combinatorial logic can be modeled efficiently using two forms of time control, the # and @ Verilog time control statements. The # time control is ignored for synthesis and hence this section describes modeling combinatorial logic with the @ statement.

A combinatorial always block has a sensitivity list appearing within parentheses after the word "always @". An always block is activated if an event (value change or edge) appears on one of the sensitivity list signals. This sensitivity list can contain any signal that appears in conditions (If, Case, for example), and any signal appearing on the right hand side of an assignment. By substituting a * without parentheses, for a list of signals, the always block is activated for an event in any of the always block's signals as described above.

Note: In combinatorial processes, if a signal is not explicitly assigned in all branches of "If" or "Case" statements, XST generates a latch to hold the last value. To avoid latch creation, be sure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the process statements.

Different statements can be used in a process:

- Variable and signal assignment
- If... else statement
- Case statement
- For and while loop statement
- Function and task call

The following sections provide examples of each of these statements.

If...Else Statement

If... else statements use true/false conditions to execute statements. If the expression evaluates to true, the first statement is executed. If the expression evaluates to false (or x or z), the else statement is executed. A block of multiple statements may be executed using begin and end keywords. If...else statements may be nested. The following example shows how a MUX can be described using an If...else statement.

Example 7-2 MUX Description Using If... Else Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

  always @(sel or a or b or c or d)
  begin
    if (sel[1])
      if (sel[0])
        outmux = d;
      else
        outmux = c;
    else
      if (sel[0])
        outmux = b;
      else
        outmux = a;
    end
endmodule
```

Case Statement

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The Case statement evaluates the branches in the order they are written. The first branch that evaluates to true is executed. If none of the branches match, the default branch is executed.

Note: Do not use unsized integers in case statements. Always size integers to a specific number of bits, or results can be unpredictable.

Casez treats all z values in any bit position of the branch alternative as a don't care.

Casex treats all x and z values in any bit position of the branch alternative as a don't care.

The question mark (?) can be used as a "don't care" in either the casez or casex case statements. The following example shows how a MUX can be described using a Case statement.

Example 7-3 MUX Description Using Case Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
  begin
    case (sel)
      2'b00: outmux = a;
      2'b01: outmux = b;
      2'b10: outmux = c;
      default: outmux = d;
    endcase
  end
endmodule
```

The preceding Case statement evaluates the values of the input sel in priority order. To avoid priority processing, it is recommended that you use a parallel-case Verilog meta comment which ensures parallel evaluation of the sel inputs as in the following.

Example:

```
case(sel)   //synthesis parallel_case
```

For and Repeat Loops

When using always blocks, repetitive or bit slice structures can also be described using the "for" statement or the "repeat" statement.

The "for" statement is supported for:

- Constant bounds
- Stop test condition using operators <, <=, > or >=
- Next step computation falling in one of the following specifications:
  - ♦ *var* = *var* + step
  - ♦ *var* = *var* - step

  (where *var* is the loop variable and *step* is a constant value).

The repeat statement is only supported for constant values.

The following example shows the use of a For Loop.

Example 7-4 For Loop Description

```
module countzeros (a, Count);
input [7:0] a;
output [2:0] Count;
reg [2:0] Count;
reg [2:0] Count_Aux;
integer i;
```

```
always @(a)
  begin
    Count_Aux = 3'b0;
  for (i = 0; i < 8; i = i+1)
    begin
      if (!a[i])
        Count_Aux = Count_Aux+1;
      end
  Count = Count_Aux;
  end

endmodule
```

While Loops

When using always blocks, use the "while" statement to execute repetitive procedures. A "while" loop executes other statements until its test expression becomes false. It is not executed if the test expression is initially false.

- The test expression is any valid Verilog expression.
- To prevent endless loops, use the "–loop_iteration_limit" switch.
- While loops can have Disable statements. The Disable statement must be use inside a labeled block, since the syntax is "disable *blockname*."

The following example shows the use of a While Loop.

Example 7-5 While Loop Description

```
parameter P = 4;
always @(ID_complete)
  begin : UNIDENTIFIED
    integer i;
    reg found;
    unidentified = 0;
    i = 0;
    found = 0;
    while (!found && (i < P))
      begin
        found = !ID_complete[i];
        unidentified[i] = !ID_complete[i];
        i = i + 1;
      end
  end
```

Sequential Always Blocks

Sequential circuit description is based on always blocks with a sensitivity list.

The sensitivity list contains a maximum of three edge-triggered events: the clock signal event (which is mandatory), possibly a reset signal event, and a set signal event. One, and only one "If...else" statement is accepted in such an always block.

An asynchronous part may appear before the synchronous part in the first and the second branch of the "If...else" statement. Signals assigned in the asynchronous part must be assigned to the constant values '0', '1', 'X' or 'Z' or any vector composed of these values.

These same signals must also be assigned in the synchronous part (that is, the last branch of the "if-else" statement). The clock signal condition is the condition of the last branch of the "if-else" statement. The following example gives the description of an 8-bit register.

Example 7-6 8 Bit Register Using an Always Block

```
module seq1 (DI, CLK, DO);
  input [7:0] DI;
  input CLK;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK)
    DO <= DI ;
endmodule
```

The following example gives the description of an 8-bit register with a clock signal and an asynchronous reset signal.

Example 7-7 8 Bit Register with Asynchronous Reset (high-true) Using an Always Block

```
module EXAMPLE (DI, CLK, RST, DO);
  input [7:0] DI;
  input CLK, RST;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK or posedge RST)
    if (RST == 1'b1)
      DO <= 8'b00000000;
    else
      DO <= DI;
endmodule
```

The following example describes an 8-bit counter.

Example 7-8 8 Bit Counter with Asynchronous Reset (low-true) Using an Always Block

```
module seq2 (CLK, RST, DO);
  input CLK, RST;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK or posedge RST)
    if (RST == 1'b1)
      DO <= 8'b00000000;
    else
      DO <= DO + 8'b00000001;
endmodule
```

Assign and Deassign Statements

Assign and deassign statements are supported within simple templates.

The following is an example of the general template for assign / deassign statements:

```verilog
module assig (RST, SELECT, STATE, CLOCK, DATA_IN);
  input RST;
  input SELECT;
  input CLOCK;
  input [0:3] DATA_IN;
  output [0:3] STATE;
  reg [0:3] STATE;

always @ (RST)
  if(RST)
    begin
      assign STATE = 4'b0;
    end
  else
    begin
      deassign STATE;
    end

always @ (posedge CLOCK)
  begin
    STATE <= DATA_IN;
  end
endmodule
```

The main limitations on support of the assign/deassign statement in XST are as follows:

- For a given signal, there must be only one assign/deassign statement. For example, XST rejects the following design:

```verilog
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
  input RST;
  input SET;
  input CLOCK;
  input DATA_IN;
  output STATE;
  reg STATE;

always @ (RST)            // block b1
  if(RST)
    assign STATE = 1'b0;
  else
    deassign STATE;

always @ (SET)            // block b1
  if(SET)
    assign STATE = 1'b1;
  else
    deassign STATE;

always @ (posedge CLOCK)  // block b2
  begin
    STATE <= DATA_IN;
  end
endmodule
```

- The assign/deassign statement must be performed in the same always block through an if/else statement. For example, XST rejects the following design:

```verilog
module dflop (RST, SET, STATE, CLOCK, DATA_IN);
  input RST;
  input SET;
  input CLOCK;
  input DATA_IN;
  output STATE;

  reg STATE;

always @ (RST or SET)       // block b1
case ({RST,SET})
  2'b00: assign STATE = 1'b0;
  2'b01: assign STATE = 1'b0;
  2'b10: assign STATE = 1'b1;
  2'b11: deassign STATE;
endcase

always @ (posedge CLOCK)    // block b2
  begin
    STATE <= DATA_IN;
  end
endmodule
```

- You cannot assign a bit/part select of a signal through an assign/deassign statement. For example, XST rejects the following design:

```verilog
module assig (RST, SELECT, STATE, CLOCK,DATA_IN);
  input RST;
  input SELECT;
  input CLOCK;
  input [0:7] DATA_IN;
  output [0:7] STATE;

  reg [0:7] STATE;

  always @ (RST)              // block b1
    if(RST)
      begin
        assign STATE[0:7] = 8'b0;
      end
    else
      begin
        deassign STATE[0:7];
      end

always @ (posedge CLOCK)    // block b2
  begin
    if (SELECT)
      STATE [0:3] <= DATA_IN[0:3];
    else
      STATE [4:7] <= DATA_IN[4:7];
end
```

Assignment Extension Past 32 Bits

If the expression on the left-hand side of an assignment is wider than the expression on the right-hand side, the left-hand side is padded to the **left** according to the following rules.

- If the right-hand expression is signed, the left-hand expression is padded with the sign bit (0 for positive, 1 for negative, z for high impedance or x for unknown).

- If the right-hand expression is unsigned, the left-hand expression is padded with `'0's`.

- For unsized x or z constants only the following rule applies. If the value of the right-hand expression's left-most bit is z (high impedance) or x (unknown), regardless of whether the right-hand expression is signed or unsigned, the left-hand expression is padded with that value (z or x, respectively).

Note: The above rules follow the Verilog-2001 standard, and are not backward compatible with Verilog-1995.

Tasks and Functions

The declaration of a function or task is intended for handling blocks used multiple times in a design. They must be declared and used in a module. The heading part contains the parameters: input parameters (only) for functions and input/output/inout parameters for tasks. The return value of a function can be declared either signed or unsigned. The content is similar to the combinatorial always block content.

Note: Recursive function and task calls are not supported.

Example 7-9 shows a function declared within a module. The ADD function declared is a single-bit adder. This function is called 4 times with the proper parameters in the architecture to create a 4-bit adder. The same example, described with a task, is shown in Example 7-10.

Example 7-9 Function Declaration and Function Call

```
module comb15 (A, B, CIN, S, COUT);
  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;
  wire [1:0] S0, S1, S2, S3;
  function signed [1:0] ADD;
    input A, B, CIN;
    reg S, COUT;
    begin
      S = A ^ B ^ CIN;
      COUT = (A&B) | (A&CIN) | (B&CIN);
      ADD = {COUT, S};
    end
  endfunction

  assign S0 = ADD (A[0], B[0], CIN),
    S1 = ADD (A[1], B[1], S0[1]),
    S2 = ADD (A[2], B[2], S1[1]),
    S3 = ADD (A[3], B[3], S2[1]),
    S = {S3[0], S2[0], S1[0], S0[0]},

    COUT = S3[1];
endmodule
```

Example 7-10 Task Declaration and Task Enable

```
module EXAMPLE (A, B, CIN, S, COUT);
  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;
  reg [3:0] S;
  reg COUT;
  reg [1:0] S0, S1, S2, S3;

  task ADD;
    input A, B, CIN;
    output [1:0] C;
    reg [1:0] C;
    reg S, COUT;

    begin
      S = A ^ B ^ CIN;
      COUT = (A&B) | (A&CIN) | (B&CIN);
      C = {COUT, S};
    end
  endtask

  always @(A or B or CIN)
  begin
    ADD (A[0], B[0], CIN, S0);
    ADD (A[1], B[1], S0[1], S1);
    ADD (A[2], B[2], S1[1], S2);
    ADD (A[3], B[3], S2[1], S3);
    S = {S3[0], S2[0], S1[0], S0[0]};
    COUT = S3[1];
  end
endmodule
```

Blocking Versus Non-Blocking Procedural Assignments

The # and @ time control statements delay execution of the statement following them until the specified event is evaluated as true. Use of blocking and non-blocking procedural assignments have time control built into their respective assignment statement.

The # delay is ignored for synthesis.

The syntax for a blocking procedural assignment is shown in the following example:

```
reg a;
a = #10 (b | c);
```

or

```
if (in1) out = 1'b0;
else out = in2;
```

As the name implies, these types of assignments block the current process from continuing to execute additional statements at the same time. These should mainly be used in simulation.

Non-blocking assignments, on the other hand, evaluate the expression when the statement executes, but allow other statements in the same process to execute as well at the same time. The variable change only occurs after the specified delay.

The syntax for a non-blocking procedural assignment is as follows:

```
variable <= @(posedge_or_negedge_bit) expression;
```

The following shows an example of how to use a non-blocking procedural assignment.

```
if (in1) out <= 1'b1;
else out <= in2;
```

Constants, Macros, Include Files and Comments

This section discusses constants, macros, include files, and comments.

Constants

By default, constants in Verilog are assumed to be decimal integers. They can be specified explicitly in binary, octal, decimal or hexadecimal by prefacing them with the appropriate syntax. For example, 4'b1010, 4'o12, 4'd10 and 4'ha all represent the same value.

Macros

Verilog provides a way to define macros as shown in the following example.

```
`define TESTEQ1 4'b1101
```

Later in the design code a reference to the defined macro is made as follows.

```
 if (request == `TESTEQ1)
```

This is shown in the following example.

```
`define myzero 0
assign mysig = `myzero;
```

Verilog provides the `ifdef and `endif constructs to determine whether a macro is defined or not. These constructs are used to define conditional compilation. If the macro called out by the `ifdef command has been defined, that code is compiled. If not, the code following the `else command is compiled. The `else is not required, but the `endif must complete the conditional statement. The `ifdef and `endif constructs are shown in the following example.

```
`ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
`else
module if_MYVAR_is_not_declared;
...
endmodule
`endif
```

Include Files

Verilog allows separating source code into more than one file. To use the code contained in another file, the current file has the following syntax:

```
`include "path/file-name-to-be-included"
```

Note: The path can be relative or absolute.

Behavioral Verilog Features

XILINX®

Multiple `include statements are allowed in a single Verilog file. This feature makes your code modular and more manageable in a team design environment where different files describe different modules of the design.

To have the file in your `include statement recognized, you must identify the directory where it resides either to ISE™ or to XST.

- By default, ISE™ searches the ISE™ project directory, so adding the file to your project directory will identify the file to ISE™.

- You can direct ISE™ to a different directory by including a path (relative or absolute) in the `include statement in your source code.

- You can point XST directly to your include file directory by using the Verilog Include Directories option. See "Verilog Include Directories (Verilog Only)" in Chapter 5.

- If the include file is required for ISE™ to construct the design hierarchy, this file must either reside in the project directory, or be referenced by a relative or absolute path. The file *need not* be added to the project.

Be aware that conflicts can occur. For example, at the top of a Verilog file you might see the following:

```
`timescale 1 ns/1 ps
`include "modules.v"
...
```

If the specified file (in this case, `modules.v`) has been added to an ISE™ project directory *and* is specified with an `include, conflicts may occur and an error message displays:

```
ERROR:Xst:1068 - fifo.v, line 2. Duplicate declarations of
module'RAMB4_S8_S8'
```

Comments

There are two forms of comments in Verilog similar to the two forms found in a language like C++.

- `//` Allows definition of a one-line comment.

- `/*` You can define a multi-line comment by enclosing it as illustrated by this sentence `*/`

Generate Statement

Generate is a construct that allows you to dynamically create Verilog code from conditional statements. This allows you to create repetitive structures or structures that are only appropriate under certain conditions. Structures that are likely to be created via a generate statement are:

- primitive or module instances
- initial or always procedural blocks
- continuous assignments
- net and variable declarations
- parameter redefinitions
- task or function definitions

XST supports the following types of generate statements:

- generate for
- generate if
- generate case

Generate For

Use a **generate for** loop to create one or more instances that can be placed inside a module. Use the generate for loop the same way you would a normal Verilog for loop with the following limitations.

- The index for a generate for loop must have a genvar variable.
- The assignments in the for loop control must refer to the genvar variable.
- The contents of the for loop must be enclosed by **begin** and **end** statements, and the **begin** statement must be named with a unique qualifier.

The following is an example of an 8-bit adder using a generate for loop.

```
generate
genvar i;

  for (i=0; i<=7; i=i+1)
    begin : for_name
      adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i],
        ci[i], sum_for[8*i+7 : 8*i], c0_or[i+1]);
    end
endgenerate
```

Generate If... else

A **generate if** statement can be used inside a generate block to conditionally control what objects get generated.

The following is an example of a generate If... else statement. The generate controls what type of multiplier is instantiated. Please note that the contents of each branch of the if... else statement must be enclosed by **begin** and **end** statements, and the **begin** statement must be named with a unique qualifier.

```
generate
  if (IF_WIDTH < 10)
    begin : if_name
      adder # (IF_WIDTH) u1 (a, b, sum_if);
    end
  else
    begin : else_name
      subtractor # (IF_WIDTH) u2 (a, b, sum_if);
    end
endgenerate
```

Generate Case

A **generate case** statement can be used inside a generate block to conditionally control what objects get generated. Use a generate case statement when there are several conditions to be tested to determine what the generated code would be. Please note that each test statement in a generate case statement must be enclosed by **begin** and **end** statements, and the **begin** statement must be named with a unique qualifier.

The following is an example of a generate case statement. The generate controls what type of adder is instantiated.

```
generate
  case (WIDTH)
    1:
      begin : case1_name
        adder #(WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
      end
    2:
      begin : case2_name
        adder #(WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
      end
    default:
      begin : d_case_name
        adder x3 (a, b, ci, sum_case, c0_case);
      end
  endcase
endgenerate
```

Variable Part Selects

Verilog 2001 adds the capability to use variables to select a group of bits from a vector. A variable part select is defined by the starting point of its range and the width of the vector, instead of being bounded by two explicit values. The starting point of the part select can vary, but the width of the part select remains constant.

- +: —Indicates that the part select increases from the starting point.
- -: —Indicates that the part select decreases from the starting point.

Example:

```
reg [3:0] data;
reg [3:0] select; // a value from 0 to 7
wire [7:0] byte = data[select +: 8];
```

Structural Verilog Features

Structural Verilog descriptions assemble several blocks of code and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are the module, the port and the signal. The component is the building or basic block. A port is a component I/O connector. A signal corresponds to a wire between components.

In Verilog, a component is represented by a design module. The module declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The module body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within component instantiation statements. These statements specify an instance of a component occurring within another component or the circuit. Each component instantiation statement is labeled with an identifier. Besides naming a component declared in a local component declaration, a component instantiation statement contains an association list (the parenthesized list) that specifies which actual signals or ports are associated with which local ports of the component declaration.

The Verilog language provides a large set of built-in logic gates which can be instantiated to build larger logic circuits. The set of logical functions described by the built-in gates includes AND, OR, XOR, NAND, NOR and NOT.

Here is an example of building a basic XOR function of two single bit inputs a and b.

```
module build_xor (a, b, c);
  input a, b;
  output c;
  wire c, a_not, b_not;
    not a_inv (a_not, a);
    not b_inv (b_not, b);
    and a1 (x, a_not, b);
    and a2 (y, b_not, a);
    or out (c, x, y);
endmodule
```

Each instance of the built-in modules has a unique instantiation name such as a_inv, b_inv, out. The wiring up of the gates describes an XOR gate in structural Verilog.

Example 7-11 gives the structural description of a half adder composed of four, 2 input nand modules.

Example 7-11 Structural Description of a Half Adder

```
module halfadd (X, Y, C, S);
  input X, Y;
  output C, S;
  wire S1, S2, S3;
  nand NANDA (S3, X, Y);
  nand NANDB (S1, X, S3);
  nand NANDC (S2, S3, Y);
  nand NANDD (S, S1, S2);
  assign C = S3;
endmodule
```
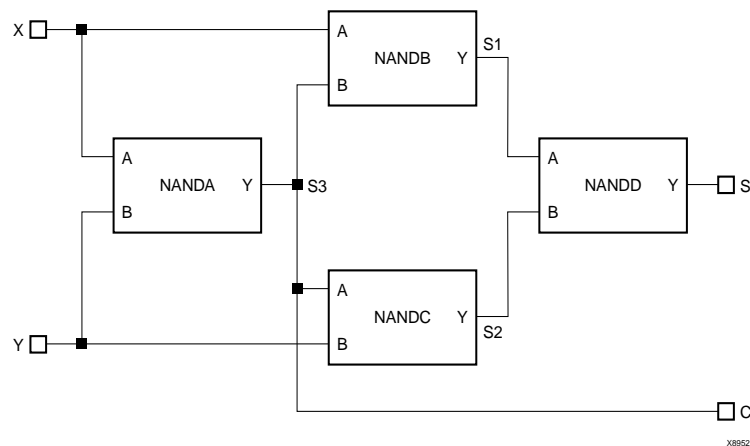

Figure 7-1: **Synthesized Top Level Netlist**

The structural features of Verilog HDL also allow you to design circuits by instantiating pre-defined primitives such as gates, registers and Xilinx® specific primitives like CLKDLL and BUFGs. These primitives are other than those included in the Verilog language. These pre-defined primitives are supplied with the XST Verilog libraries (`unisim_comp.v`).

Example 7-12 Structural Instantiation of Register and BUFG

```verilog
module foo (sysclk, in, reset, out);
  input sysclk, in, reset;
  output out;
  reg out;
  wire sysclk_out;

  FDC register (sysclk, reset, in, out);  //position based referencing
  BUFG clk (.O(sysclk_out), .I(sysclk));  //name based referencing
  ...
endmodule
```

The `unisim_comp.v` library file supplied with XST, includes the definitions for FDC and BUFG.

```verilog
module FDC ( C, CLR, D, Q);
  input C;
  input CLR;
  input D;
  output Q;
endmodule

// synthesis attribute BOX_TYPE of FDC is "BLACK_BOX"

module BUFG ( O, I);
  output O;
  input I;
endmodule

// synthesis attribute BOX_TYPE of BUFG is "BLACK_BOX"
```

Parameters

Verilog modules support defining constants known as parameters which can be passed to module instances to define circuits of arbitrary widths. Parameters form the basis of creating and using parameterized blocks in a design to achieve hierarchy and stimulate modular design techniques. The following is an example of the use of parameters. Null string parameters are not supported.

Example 7-13 Using Parameters

```verilog
module lpm_reg (out, in, en, reset, clk);
  parameter SIZE = 1;
  input in, en, reset, clk;
  output out;
  wire [SIZE-1 : 0] in;
  reg  [SIZE-1 : 0] out;
always @(posedge clk or negedge reset)
  begin
    if (!reset)
      out <= 1'b0;
    else
      if (en)
        out <= in;
      else
        out <= out;    //redundant assignment
  end
endmodule
```

```
module top ();    //portlist left blank intentionally
  ...
  wire [7:0] sys_in, sys_out;
  wire sys_en, sys_reset, sysclk;
  lpm_reg #8 buf_373 (sys_out, sys_in, sys_en, sys_reset, sysclk);
  ...
endmodule
```

Instantiation of the module lpm_reg with a instantiation width of 8 causes the instance buf_373 to be 8 bits wide.

Parameter/Attribute Conflicts

Since parameters and attributes can be applied to both instances and modules in your Verilog code, and attributes can also be specified in a constraints file, from time to time, conflicts will arise. To resolve these conflicts, XST uses the following rules of precedence.

1. Whatever is specified on an instance (lower level) takes precedence over what is specified on a module (higher level).

2. If a parameter and an attribute are specified on either the same instance or the same module, the parameter takes precedence, and XST issues a message warning of the conflict.

3. An attribute specified in the XCF file will always take precedence over attributes or parameters specified in your Verilog code.

Note: When an attribute specified on an instance overrides a parameter specified on a module in XST, it is possible that your simulation tool may nevertheless use the parameter. This may cause the simulation results to not match the synthesis results.

Use the following matrix as a guide in determining precedence.

	Parameter on an Instance	**Parameter on a Module**
Attribute on an Instance	Apply Parameter (XST issues warning message)	Apply Attribute (possible simulation mismatch)
Attribute on a Module	Apply Parameter	Apply Parameter (XST issues warning message)
Attribute in XCF	Apply Attribute (XST issues warning message)	Apply Attribute

Note: Security attributes on the module definition always have higher precedence than any other attribute or parameter.

Verilog Limitations in XST

This section describes Verilog limitations in XST support for case sensitivity, and blocking and nonblocking assignments.

Case Sensitivity

XST supports case sensitivity as follows:

- Designs can use case equivalent names for I/O ports, nets, regs and memories.

- Equivalent names are renamed using a postfix ("rnm<Index>").

- A rename construct is generated in the NGC file.

- Designs can use Verilog identifiers that differ only in case. XST renames them using a postfix as with equivalent names.

    Following is an example.

    ```
    module upperlower4 (input1, INPUT1, output1, output2);
       input input1;
       input INPUT1;
    ```

    For the above example, INPUT1 is renamed to INPUT1_rnm0.

The following restrictions apply for Verilog within XST:

- Designs using equivalent names (named blocks, tasks, and functions) are rejected.

    Example:

    ```
    ...
    always @(clk)
    begin: fir_main5
      reg [4:0] fir_main5_w1;
      reg [4:0] fir_main5_W1;
    ```

    This code generates the following error message:

    ```
    ERROR:Xst:863 - "design.v", line 6: Name conflict
    (<fir_main5/fir_main5_w1> and <fir_main5/fir_main5_W1>)
    ```

- Designs using case equivalent module names are also rejected.

    Example:

    ```
    module UPPERLOWER10 (...);
    ...
    module upperlower10 (...);
    ...
    ```

    This example generates the following error message:

    ```
    ERROR:Xst:909 - Module name conflict (UPPERLOWER10 and upperlower10).
    ```

Blocking and Nonblocking Assignments

XST rejects Verilog designs if a given signal is assigned through both blocking and nonblocking assignments as in the following example.

```
always @(in1)
begin
  if (in2)
    out1 = in1;
  else
    out1 <= in2;
end
```

If a variable is assigned in both a blocking and nonblocking assignment, the following error message is generated:

```
ERROR:Xst:880 - "design.v", line n: Cannot mix blocking and non blocking
assignments on signal <out1>.
```

There are also restrictions when mixing blocking and nonblocking assignments on bits and slices.

The following example is rejected even if there is no real mixing of blocking and non blocking assignments:

```
if (in2)
  begin
    out1[0] = 1'b0;
    out1[1] <= in1;
  end
else
  begin
    out1[0] = in2;
    out1[1] <= 1'b1;
  end
```

Errors are checked at the signal level, not at the bit level.

If there is more than a single blocking/non blocking error, only the first one is reported.

In some cases, the line number for the error might be incorrect (as there might be multiple lines where the signal has been assigned).

Integer Handling

There are several cases where XST handles integers differently from other synthesis tools, and so they must be coded in a particular way.

In **Case** statements, do not use unsized integers in case item expressions, as this causes unpredictable results. In the following example, the case item expression "4" is an unsized integer that causes unpredictable results. To avoid problems, size the "4" to 3 bits as shown below.

```
reg [2:0] condition1;

always @(condition1)
  begin
    case(condition1)
      4    : data_out = 2;    // < will generate bad logic
      3'd4 : data_out = 2;    // < will work
    endcase
  end
```

In **concatenations**, do not use unsized integers, as this causes unpredictable results. If you must use an expression that results in an unsized integer, assign the expression to a temporary signal, and use the temporary signal in the concatenation as shown below.

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog Meta Comments

XST supports meta comments in Verilog. Meta comments are comments that are understood by the Verilog parser.

Meta comments can be used as follows:

- Set constraints on individual objects (for example, module, instance, net).
- Set directives on synthesis:
    - ♦ parallel_case and full_case directives.
    - ♦ translate_on translate_off directives.
    - ♦ all tool specific directives (for example, syn_sharing), refer to Chapter 5, "Design Constraints" for details.

Meta comments can be written using the C-style (`/* ... */`) or the Verilog style (`// ...`) for comments. C-style comments can be multiple line. Verilog style comments end at the end of the line.

XST supports the following:

- Both C-style and Verilog style meta comments
- translate_on translate_off directives

```
// synthesis translate_on
// synthesis translate_off
```

- parallel_case, full_case directives

```
// synthesis parallel_case full_case
// synthesis parallel_case
// synthesis full_case
```

- Constraints on individual objects

    The general syntax is:

    // synthesis attribute *AttributeName* [**of**] *ObjectName* [**is**] *AttributeValue*

Examples:

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HUSET u1 MY_SET
// synthesis attribute fsm_extract of State2 is "yes"
// synthesis attribute fsm_encoding of State2 is "gray"
```

For a full list of constraints, refer to Chapter 5, "Design Constraints."

Verilog Language Support Tables

The following tables indicate which Verilog constructs are supported in XST. Previous sections in this chapter describe these constructs and their use within XST.

Note: XST does not allow underscores as the first character of signal names (for example, _DATA_1).

Table 7-3: **Constants**

Integer Constants	Supported
Real Constants	Supported
Strings Constants	Unsupported

Table 7-4: **Data Types**

Nets	net type	wire	Supported
		tri	Supported
		supply0, supply1	Supported
		wand, wor, triand, trior	Supported
		tri0, tri1, trireg	Unsupported
	drive strength		Ignored
Registers	reg		Supported
	integer		Supported
	real		Unsupported
	realtime		Unsupported
Vectors	net		Supported
	reg		Supported
	vectored		Supported
	scalared		Supported
Multi-Dimensional Arrays (<= 3 dimensions)			Supported
Parameters			Supported
Named Events			Unsupported

Table 7-5: **Continuous Assignments**

Drive Strength	Ignored
Delay	Ignored

Table 7-6: **Procedural Assignments**

Blocking Assignments		Supported
Non-Blocking Assignments		Supported
Continuous Procedural Assignments	assign	Supported with limitations See "Assign and Deassign Statements"
	deassign	
	force	Unsupported
	release	Unsupported
if Statement	if, if else	Supported
case Statement	case, casex, casez	Supported
forever Statement		Unsupported
repeat Statement		Supported (repeat value must be constant)
while Statement		Supported
for Statement		Supported (bounds must be static)
fork/join Statement		Unsupported
Timing Control on Procedural Assignments	delay (#)	Ignored
	event (@)	Unsupported
	wait	Unsupported
	named events	Unsupported
Sequential Blocks		Supported
Parallel Blocks		Unsupported
Specify Blocks		Ignored

Table 7-6: **Procedural Assignments**

initial Statement		Supported
always Statement		Supported
task		Supported (Recursion Unsupported)
functions		Supported (Constant Functions Unsupported, Recursion Unsupported)
disable Statement		Supported

Table 7-7: **System Tasks and Functions**

System Tasks	Ignored
System Functions	Unsupported

Table 7-8: **Design Hierarchy**

Module definition	Supported
Macromodule definition	Unsupported
Hierarchical names	Unsupported
defparam	Supported
Array of instances	Supported

Table 7-9: **Compiler Directives**

`celldefine `endcelldefine	Ignored
`default_nettype	Supported
`define	Supported
`ifdef `else `endif	Supported
`undef, `ifndef, `elsif,	Supported
`include	Supported
`resetall	Ignored
`timescale	Ignored

Table 7-9: **Compiler Directives**

`unconnected_drive` `nounconnected_drive`	Ignored
`uselib	Unsupported
`file, `line	Supported

Primitives

XST supports certain gate level primitives. The supported syntax is as follows:

```
gate_type instance_name (output, inputs, ...);
```

The following example shows Gate Level Primitive Instantiations.

```
and U1 (out, in1, in2);
bufif1 U2 (triout, data, trienable);
```

The following table shows which primitives are supported.

Table 7-10: **Primitives**

Gate Level Primitives	and nand nor or xnor xor	Supported
	buf not	Supported
	bufif0 bufif1 notif0 notif1	Supported
	pulldown pullup	Unsupported
	drive strength	Ignored
	delay	Ignored
	array of primitives	Supported
Switch Level Primitives	cmos nmos pmos rcmos rnmos rpmos	Unsupported
	rtran rtranif0 rtranif1 tran tranif0 tranif1	Unsupported
User Defined Primitives		Unsupported

Verilog Reserved Keywords

The following table shows the Verilog reserved keywords.

Table 7-11: **Verilog Reserved Keywords.**

always	end	ifnone	not	rnmos	tri
and	endcase	incdir*	notif0	rpmos	tri0
assign	endconfig*	include*	notif1	rtran	tri1
automatic	endfunction	initial	or	rtranif0	triand
begin	endgenerate	inout	output	rtranif1	trior
buf	endmodule	input	parameter	scalared	trireg
bufif0	endprimitive	instance*	pmos	show-cancelled*	use*
bufif1	endspecify	integer	posedge	signed	vectored
case	endtable	join	primitive	small	wait
casex	endtask	large	pull0	specify	wand
casez	event	liblist*	pull1	specparam	weak0
cell*	for	library*	pullup	strong0	weak1
cmos	force	localparam*	pulldown	strong1	while
config*	forever	macromodule	pulsestyle-_ondetect*	supply0	wire
deassign	fork	medium	pulsestyle-_onevent*	supply1	wor
default	function	module	rcmos	table	xnor
defparam	generate	nand	real	task	xor
design*	genvar	negedge	realtime	time	
disable	highz0	nmos	reg	tran	
edge	highz1	nor	release	tranif0	
else	if	noshow-cancelled*	repeat	tranif1	

* These keywords are reserved by Verilog, but not supported by XST.

Verilog-2001 Support in XST

XST 6.1i supports the following Verilog-2001 features. For details on Verilog -2001, see *Verilog-2001: A Guide to the New Features* by Stuart Sutherland, or *IEEE Standard Verilog Hardware Description Language* manual, (IEEE Standard 1364-2001).

- Generate statements
- Combined port/data type declarations
- ANSI-style port lists
- Module parameter port lists
- ANSI C style task/function declarations
- Comma separated sensitivity list
- Combinatorial logic sensitivity
- Default nets with continuous assigns
- Disable default net declarations
- Indexed vector part selects
- Multi-dimensional arrays
- Arrays of net and real data types
- Array bit and part selects
- Signed reg, net, and port declarations
- Signed based integer numbers
- Signed arithmetic expressions
- Arithmetic shift operators
- Automatic width extension past 32 bits
- Power operator
- N sized parameters
- Explicit in-line parameter passing
- Fixed local parameters
- Enhanced conditional compilation
- File and line compiler directives
- Variable part selects

Mixed Language Support

This chapter contains the following sections:

Introduction

XST supports mixed VHDL/Verilog projects. This chapter explains how to create mixed language projects and what the current limitations are. The following are key features of mixed language support:

- Mixing of VHDL and Verilog is restricted to design unit (cell) instantiation only. A VHDL design can instantiate a Verilog module, and a Verilog design can instantiate a VHDL entity. Any other kind of mixing between VHDL and Verilog is not supported.

- In a VHDL design, a restricted subset of VHDL types, generics and ports is allowed on the boundary to a Verilog module. Similarly, in a Verilog design, a restricted subset of Verilog types, parameters and ports is allowed on the boundary to a VHDL entity or configuration.

- XST binds VHDL design units to a Verilog module during the Elaboration step.

- Component instantiation based on default binding is used for binding Verilog modules to a VHDL design unit.

Note: Configuration specification, direct instantiation and component configurations are not supported for a Verilog module instantiation in VHDL.

In supporting mixed projects:

- VHDL and Verilog project files are unified.

- VHDL and Verilog libraries are logically unified.

- Specification of work directory for compilation (xsthdpdir), previously available only for VHDL, is also available for Verilog.

- The xhdp.ini mechanism for mapping a logical library name to a physical directory name on the host file system, previously available only for VHDL, is also available for Verilog.

- Mixed language projects accept a search order used for searching unified logical libraries in design units (cells). During elaboration, XST follows this search order for picking and binding a VHDL entity or a Verilog module to the mixed language project.

Mixed Language Project File

XST uses a dedicated mixed language project file to support mixed VHDL/Verilog designs. You can use this mixed language format not only for mixed projects, but also for purely VHDL or Verilog projects. If you use Project Navigator to run XST, Project Navigator creates the project file, and it is always a mixed language project file. If you run XST from the command line, you must create a mixed language project file for your mixed language projects.

To create a mixed language project file at the command line, use the –ifmt command line switch set to *mixed* or with its value is omitted. Please note that you can still use the VHDL and Verilog formats for existing designs. To use the VHDL format, set –ifmt to *vhdl,* and to use the Verilog format, set –ifmt to *verilog.*

The syntax for invoking a library or any external file in a mixed language project is as follows:

```
language library file_name.ext
```

The following is an example of how to invoke libraries in a mixed language project:

```
vhdl     work        my_vhdl1.vhd
verilog  work        my_vlg1.v
vhdl     my_vhdl_lib  my_vhdl2.vhd
verilog  my_vlg_lib   my_vlg2.v
```

Each line specifies a single HDL design file:

- The first column specifies whether the HDL file is VHDL or Verilog.
- The second column specifies the logic library, where the HDL is compiled. By default the logic library is "work".
- The third column specifies the name of the HDL file.

VHDL/Verilog Boundary Rules

The boundary between VHDL and Verilog is enforced at the design unit level. A VHDL design can instantiate a Verilog module. A Verilog design can instantiate a VHDL entity.

Instantiating a Verilog Module in a VHDL Design

To instantiate a Verilog module in your VHDL design, do the following.

1. Declare a VHDL component with the same name (respecting case sensitivity) as the Verilog module you want to instantiate. If the Verilog module name is not all lower case, use the Case property to preserve the case of your Verilog module. In Project Navigator, select *Maintain* for the Case option under the Synthesis Options tab in the Process Properties dialog box, or set the –case command line option to *maintain* at the command line.

2. Instantiate your Verilog component as if you were instantiating a VHDL component.

Note: Using a VHDL configuration declaration, one could attempt to bind this component to a particular design unit from a particular library. Please note that such binding is not supported. Only default Verilog module binding is supported.

The only Verilog construct that can be instantiated in a VHDL design is a Verilog module. No other Verilog constructs are visible to VHDL code.

During elaboration, all components subject to default binding are regarded as design units with the same name as the corresponding component name. In the binding process, XST treats a component name as a VHDL design unit name and searches for it in the logical library "work." If a VHDL design unit is found, then XST binds it. If XST cannot find a VHDL design unit, it treats the component name as a Verilog module name and searches for it using a case sensitive search. XST searches for the Verilog module in the user specified list of unified logical libraries in the user specified search order. See "Library Search Order File" for search order details. XST selects the first Verilog module matching the name, and binds it.

Note: Please remember that since libraries are unified, a Verilog cell by the same name as that of a VHDL design unit cannot co-exist in the same logical library. A newly compiled cell/unit overrides a previously compiled one.

Instantiating a VHDL Design Unit in a Verilog Design

To instantiate a VHDL entity, declare a module name with the same as name as the VHDL entity (optionally followed by an architecture name) that you want to instantiate, and perform a normal Verilog instantiation. The only VHDL construct that can be instantiated in a Verilog design is a VHDL entity. No other VHDL constructs are visible to Verilog code. When you do this, XST uses the entity/architecture pair as the Verilog/VHDL boundary.

XST performs the binding during elaboration. In the binding process, XST searches for a Verilog module name (it ignores any architecture name specified in the module instantiation) using the name of the instantiated module in the user specified list of unified logical libraries in the user specified order. See "Library Search Order File" for search order details. If found, XST binds the name. If XST cannot find a Verilog module, it treats the name of the instantiated module as a VHDL entity, and searches for it using a case sensitive search for a VHDL entity. XST searches for the VHDL entity in the user specified list of unified logical libraries in the user specified order, assuming that a VHDL design unit was stored with extended identifier. See "Library Search Order File" for search order details. If found, XST binds the name. XST selects the first VHDL entity matching the name, and binds it.

XST has the following limitations when instantiating a VHDL design unit from a Verilog module:

- Explicit port association must be used. That is, formal and effective port names must be specified in the port map.

- All parameters must be passed at instantiation, even if they are unchanged.

- The parameter override shall be named and not ordered. The parameter override must be done though instantiation and not through defparams.

The following is an example of the correct use of parameter override.

```
ff #(.init(2'b01)) u1 (.sel(sel), .din(din), .dout(dout));
```

The following is an *incorrect* use of the of parameter override, and is not accepted by XST.

```
ff u1 (.sel(sel), .din(din), .dout(dout));
defparam u1.init = 2'b01;
```

Port Mapping

XST uses the following rules and limitations for port mapping in mixed language projects.

- For VHDL entities instantiated in Verilog designs, XST supports the following port types.
    - in
    - out
    - inout

    Note: XST does not support VHDL buffer and linkage ports.

- For Verilog modules instantiated in VHDL designs, XST supports the following port types.
    - input
    - output
    - inout

    Note: XST does not support connection to bi-directional pass switches in Verilog.

- XST does not support unnamed Verilog ports for mixed language boundaries.

- Use an equivalent component declaration for connecting to a case sensitive port in a Verilog module. By default, XST assumes Verilog ports are in all lower case.

- XST supports the following VHDL data types for mixed language designs.
    - bit
    - bit_vector
    - std_logic
    - std_ulogic
    - std_logic_vector
    - std_ulogic_vector

XST supports the following Verilog data types for mixed language designs.

- wire
- reg

Generics Support in Mixed Language Projects

XST supports the following VHDL generic types, and their Verilog equivalents for mixed language designs.

- integer
- real
- string
- boolean

Library Search Order File

The Library Search Order (LSO) file specifies the search order that XST uses to link the libraries used in VHDL/Verilog mixed language designs. By default, XST searches the files specified in the project file in the order in which they appear in that file. XST uses the default search order when either the DEFAULT_SEARCH_ORDER keyword is used in the LSO file or the LSO file is not specified.

Project Navigator

In Project Navigator, the default name for the LSO file is `project_name.lso`. If a `project_name.lso` file does not already exist, Project Navigator automatically creates one. If Project Navigator detects an existing `project_name.lso` file, this file is preserved and used as it is. Please remember that in Project Navigator, the name of the project is the name of the top-level block. In creating a default LSO file, Project Navigator places the DEFAULT_SEARCH_ORDER keyword in the first line of the file.

Command Line

When using XST from the command line, specify the Library Search Order file by using the –lso command line switch. If the –lso switch is omitted, XST automatically uses the default library search order without using an LSO file.

Search Order Rules

XST follows the following search order rules when processing a mixed language project.

- When the LSO file contains only the DEFAULT_SEARCH_ORDER keyword, XST:
  - searches the specified library files in the order in which they appear in the project file.
  - updates the LSO file by:
    - removing the DEFAULT_SEARCH_ORDER keyword.
    - adding the list of libraries to the LSO file in the order in which they appear in the project file.

  See "Example 1".

- When the LSO file contains the DEFAULT_SEARCH_ORDER keyword, and a list of the libraries, XST:
  - searches the specified library files in the order in which they appear in the project file.
  - ignores the list of library files in the LSO file.
  - leaves the LSO file unchanged.

  See "Example 2".

- When the LSO file contains a list of the libraries without the DEFAULT_SEARCH_ORDER keyword, XST:
  - searches the library files in the order in which they appear in the LSO file.
  - leaves the LSO file unchanged.

  See "Example 3".

- When the LSO file is empty, XST:
  - generates a warning message stating that the LSO file is empty.
  - searches the files specified in the project file using the default library search order.
  - updates the LSO file by adding the list of libraries in the order that they appear in the project file.
- When the LSO file contains a library name that does not exist in the project or INI file, and the LSO file does not contain the DEFAULT_SEARCH_ORDER keyword, XST ignores the library.

  See "Example 4".

Examples

Example 1

For a project file, `my_proj.prj`, with the following contents:

```
vhdl      vhlib1  f1.vhd
verilog   rtfllib f1.v
vhdl      vhlib2  f3.vhd
LSO file Created by ProjNav
```

and an LSO file, `my_proj.lso`, created by Project Navigator, with the following contents:

```
DEFAULT_SEARCH_ORDER
```

XST uses the following search order.

```
vhlib1
rtfllib
vhlib2
```

After processing, the contents of `my_proj.lso` will be:

```
vhlib1
rtfllib
vhlib2
```

Example 2

For a project file, `my_proj.prj`, with the following contents:

```
vhdl      vhlib1  f1.vhd
verilog   rtfllib f1.v
vhdl      vhlib2  f3.vhd
```

and an LSO file, `my_proj.lso`, created with the following contents:

```
rtfllib
vhlib2
vhlib1
DEFAULT_SEARCH_ORDER
```

XST uses the following search order.

```
vhlib1
rtfllib
vhlib2
```

After processing, the contents of `my_proj.lso` will be:

```
rtfllib
vhlib2
vhlib1
DEFAULT_SEARCH_ORDER
```

Example 3

For a project file, `my_proj.prj`, with the following contents:

```
vhdl    vhlib1  f1.vhd
verilog rtfllib f1.v
vhdl    vhlib2  f3.vhd
```

and an LSO file, `my_proj.lso`, created with the following contents:

```
rtfllib
vhlib2
vhlib1
```

XST uses the following search order.

```
rtfllib
vhlib2
vhlib1
```

After processing, the contents of `my_proj.lso` will be:

```
rtfllib
vhlib2
vhlib1
```

Example 4

For a project file, `my_proj.prj`, with the following contents:

```
vhdl    vhlib1  f1.vhd
verilog rtfllib f1.v
vhdl    vhlib2  f3.vhd
```

and an LSO file, `my_proj.lso`, created with the following contents:

```
personal_lib
rtfllib
vhlib2
vhlib1
```

XST uses the following search order.

```
rtfllib
vhlib2
vhlib1
```

After processing, the contents of `my_proj.lso` will be:

```
rtfllib
vhlib2
vhlib1
```

Log File Analysis

This chapter contains the following sections:

Introduction

The XST log file related to FPGA optimization contains the following sections:

- Copyright Statement
- Table of Contents

  Use this section to quickly navigate to different LOG file sections.

  Note: These headings are not linked. Use the Find function in your text editor to navigate.

- Synthesis Options Summary
- HDL Compilation

  See "HDL Analysis" below.

- HDL Analysis

  During HDL Compilation and HDL Analysis, XST parses and analyzes VHDL/Verilog files and gives the names of the libraries into which they are compiled. During this step XST may report potential mismatches between synthesis and simulation results, potential multi-sources, and other issues.

- HDL Synthesis

  During this step, XST tries to recognize as many macros as possible to create a technology specific implementation. This is done on a block by block basis. At the end of this step XST gives an HDL Synthesis Report.

  See Chapter 2, "HDL Coding Techniques" for more details about the processing of each macro and the corresponding messages issued during the synthesis process.

- Advanced HDL Synthesis (contains HDL Synthesis Report)

  During this step XST performs advanced macro recognition and inference. In this step, XST recognizes dynamic shift registers, implements pipelined multipliers, codes state machines, etc. This report contains a summary of recognized macros in the overall design, sorted by macro type.

- Low Level Synthesis

  During this step XST reports the potential removal of equivalent flip-flops, register replication, etc.

  For more information, see "Log File Analysis" in Chapter 3.

- Final Report

  The Final report is different for FPGA and CPLD flows as follows.

  ♦ FPGA and CPLD: includes the output file name, output format, target family and cell usage.

  ♦ FPGA only: In addition to the above, the report includes the following information for FPGAs.

    - Device Utilization Summary: where XST estimates the number of slices, gives the number of flip-flops, IOBs, BRAMS, etc. This report is very close to the one produced by MAP.

    - Clock Information: gives information about the number of clocks in the design, how each clock is buffered and how many loads it has.

    - Timing report: contains Timing Summary and Detailed Timing Report. For more information, see "Log File Analysis" in Chapter 3.

    - Encrypted Modules: if a design contains encrypted modules, XST hides the information about these modules.

Reducing the Size of the LOG File

There are several ways to reduce the size of the LOG file, generated by XST. They are as follows:

- Quiet Mode
- Silent Mode
- Hiding specific messages

When running XST from within Project Navigator, you can use a Message Filtering wizard to select specific messages to filter out of the log file. See *Using the Message Filters* in the ISE™ Help for more information.

Quiet Mode

Quiet mode allows you to limit the number of messages that are printed to the computer screen (stdout).

Quiet mode can be invoked by using the –intstyle command line switch with its value set to either *ise* or *xflow* as appropriate. The *ise* option formats messages for ISE™, while the *xflow* option formats messages for XFLOW use. You can also use the old –quiet switch, but Xilinx strongly recommends that you not use this method because it will become obsolete in coming releases.

Normally, XST prints the entire log to stdout. In quiet mode, XST does *not* print the following portions of the log to stdout:

- Copyright Message
- Table Of Contents
- Synthesis Options Summary

- The following portions of the Final Report
    - ♦ Final Results header for CPLDs
    - ♦ Final Results section for FPGAs
    - ♦ The following note in the Timing Report

        NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE. FOR
        ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
        GENERATED AFTER PLACE-AND-ROUTE.

    - ♦ Timing Detail
    - ♦ CPU (XST run time)
    - ♦ Memory usage

    Note: Device Utilization Summary, Clock Information, and Timing Summary are still available
    for FPGAs.

Silent Mode

Silent mode allows you keep any messages from going to the computer screen (stdout),
while XST continues to generate the entire LOG file. Silent mode can be invoked using
–intstyle switch with value set to *silent.*

Hiding specific messages

You can hide specific messages generated by XST at the HDL or Low Level Synthesis steps
in specific situations by using the XIL_XST_HIDEMESSAGES environment variable. This
environment variable can have one of the following values.

- *none* — maximum verbosity. All messages are printed out. This is the default.
- *hdl_level* — reduce verbosity during VHDL/Verilog Analysis and HDL Basic and
  Advanced Synthesis.
- *low_level* — reduce verbosity during Low-level Synthesis
- *hdl_and_low_levels* — reduce verbosity at all stages.

The following messages are hidden when *hdl_level* and *hdl_and_low_levels* values are
specified for the XIL_XST_HIDEMESSAGES environment variable.

- WARNING:HDLCompilers:38 - *design*.v line 5 Macro *'my_macro'*
  redefined

  Note: Note: this message is issued by the Verilog compiler only.

- WARNING:Xst:916 - *design*.vhd line 5: Delay is ignored for
  synthesis.
- WARNING:Xst:766 - *design*.vhd line 5: Generating a Black Box for
  component *comp*.
- Instantiating component *comp* from Library *lib*.
- Set user-defined property "LOC = X1Y1" for instance *inst* in
  unit *block*.
- Set user-defined property "RLOC = X1Y1" for instance *inst* in
  unit *block*.
- Set user-defined property "INIT = 1" for instance *inst* in unit
  block.
- Register *reg1* equivalent to *reg2* has been removed.

The following messages are hidden when low_level and hdl_and_low_levels values are specified for the XIL_XST_HIDEMESSAGES environment variable.

- `WARNING:Xst:382 - Register reg1 is equivalent to reg2.`
- `Register reg1 equivalent to reg2 has been removed.`
- `WARNING:Xst:1710 - FF/Latch reg (without init value) is constant in block block.`
- `WARNING:Xst 1293 - FF/Latch reg is constant in block block.`
- `WARNING:Xst:1291 - FF/Latch reg is unconnected in block block.`
- `WARNING:Xst:1426 - The value init of the FF/Latch reg hinders the constant cleaning in the block block. You could achieve better results by setting this init to value.`

Timing Report

At the end of synthesis, XST reports the timing information for the design. The report shows the information for all four possible domains of a netlist: "register to register", "input to register", "register to outpad" and "inpad to outpad".

See the TIMING REPORT section of the example given in the "FPGA Log File" section for an example of the timing report sections in the XST log.

FPGA Log File

The following is an example of an XST log file for FPGA synthesis.

```
Release 7.1i - xst H.37
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

-->
TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) Advanced HDL Synthesis
     4.1) HDL Synthesis Report
  5) HDL Synthesis
  6) Low Level Synthesis
  7) Final Report
     7.1) Device utilization summary
     7.2) TIMING REPORT
=======================================================================
*                      Synthesis Options Summary                      *
=======================================================================
---- Source Parameters
Input File Name                  : stopwatch.prj
Input Format                     : mixed
Ignore Synthesis Constraint File : NO
Verilog Include Directory        :

---- Target Parameters
Output File Name                 : stopwatch
Output Format                    : NGC
Target Device                    : xc2v40-6-cs144
```

```
     ---- Source Options
     Top Module Name              : stopwatch
     Automatic FSM Extraction     : YES
     FSM Encoding Algorithm       : one-hot
     FSM Style                    : lut
     RAM Extraction               : Yes
     RAM Style                    : Auto
     ROM Extraction               : Yes
     ROM Style                    : Auto
     Mux Extraction               : YES
     Mux Style                    : Auto
     Decoder Extraction           : YES
     Priority Encoder Extraction  : YES
     Shift Register Extraction    : YES
     Logical Shifter Extraction   : YES
     XOR Collapsing               : YES
     Resource Sharing             : YES
     Multiplier Style             : auto
     Automatic Register Balancing : No

     ---- Target Options
     Add IO Buffers               : YES
     Global Maximum Fanout        : 500
     Add Generic Clock Buffer(BUFG) : 16
     Register Duplication         : YES
     Equivalent register Removal  : YES
     Slice Packing                : YES
     Pack IO Registers into IOBs  : auto

     ---- General Options
     Optimization Goal            : Speed
     Optimization Effort          : 1
     Keep Hierarchy               : YES
     Global Optimization          : AllClockNets
     RTL Output                   : Yes
     Write Timing Constraints     : NO
     Hierarchy Separator          : _
     Bus Delimiter                : <>
     Case Specifier               : maintain
     Slice Utilization Ratio      : 100
     Slice Utilization Ratio Delta : 5

     ---- Other Options
     lso                          : stopwatch.lso
     Read Cores                   : YES
     cross_clock_analysis         : NO
     verilog2001                  : YES
     Optimize Instantiated Primitives : NO
     tristate2logic               : No

     =========================================================================

     Setting FSM Encoding Algorithm to : ONE
```

```
=========================================================================
*                         HDL Compilation                               *
=========================================================================
Compiling vhdl file C:/temp/timer/smallcntr.vhd in Library work.
Architecture inside of Entity smallcntr is up to date.
Compiling vhdl file C:/temp/timer/statmach.vhd in Library work.
Architecture inside of Entity statmach is up to date.
Compiling vhdl file C:/temp/timer/decode.vhd in Library work.
Architecture behavioral of Entity decode is up to date.
Compiling vhdl file C:/temp/timer/cnt60.vhd in Library work.
Architecture inside of Entity cnt60 is up to date.
Compiling vhdl file C:/temp/timer/hex2led.vhd in Library work.
Architecture hex2led_arch of Entity hex2led is up to date.
Compiling vhdl file C:/temp/timer/stopwatch.vhd in Library work.
Architecture inside of Entity stopwatch is up to date.


=========================================================================
*                         HDL Analysis                                  *
=========================================================================
Analyzing Entity <stopwatch> (Architecture <inside>).
WARNING:Xst:766 - C:/temp/timer/stopwatch.vhd line 68: Generating a Black Box for component
<tenths>.
Entity <stopwatch> analyzed. Unit <stopwatch> generated.

Analyzing Entity <statmach> (Architecture <inside>).
Entity <statmach> analyzed. Unit <statmach> generated.

Analyzing Entity <decode> (Architecture <behavioral>).
Entity <decode> analyzed. Unit <decode> generated.

Analyzing Entity <cnt60> (Architecture <inside>).
Entity <cnt60> analyzed. Unit <cnt60> generated.

Analyzing Entity <smallcntr> (Architecture <inside>).
Entity <smallcntr> analyzed. Unit <smallcntr> generated.

Analyzing Entity <hex2led> (Architecture <hex2led_arch>).
Entity <hex2led> analyzed. Unit <hex2led> generated.



=========================================================================
*                         Advanced HDL Synthesis                        *
=========================================================================

Advanced RAM inference ...
Advanced multiplier inference ...
Optimizing FSM <FSM_0> on signal <current_state[1:6]> with one-hot encoding.
--------------------
State     | Encoding
--------------------
clear     | 000001
zero      | 000010
start     | 000100
counting  | 001000
stop      | 010000
stopped   | 100000
--------------------
Dynamic shift register inference ...
```

```
==========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                               : 3
 16x10-bit ROM                       : 1
 16x7-bit ROM                        : 2
# Counters                           : 2
 4-bit up counter                    : 2
# Registers                          : 6
 1-bit register                      : 6


==========================================================================


==========================================================================
*                          HDL Synthesis                                 *
==========================================================================


Synthesizing Unit <smallcntr>.
    Related source file is C:/temp/timer/smallcntr.vhd.
    Found 4-bit up counter for signal <qoutsig>.
    Summary:
        inferred   1 Counter(s).
Unit <smallcntr> synthesized.


Synthesizing Unit <hex2led>.
    Related source file is C:/temp/timer/hex2led.vhd.
    Found 16x7-bit ROM for signal <LED>.
    Summary:
        inferred   1 ROM(s).
Unit <hex2led> synthesized.


Synthesizing Unit <cnt60>.
    Related source file is C:/temp/timer/cnt60.vhd.
Unit <cnt60> synthesized.


Synthesizing Unit <decode>.
    Related source file is C:/temp/timer/decode.vhd.
    Found 16x10-bit ROM for signal <one_hot>.
    Summary:
        inferred   1 ROM(s).
Unit <decode> synthesized.
```

```
Synthesizing Unit <statmach>.
    Related source file is C:/temp/timer/statmach.vhd.
    Found finite state machine <FSM_0> for signal <current_state>.
      ------------------------------------------------------------
      | States          | 6                                      |
      | Transitions     | 11                                     |
      | Inputs          | 1                                      |
      | Outputs         | 2                                      |
      | Clock           | CLK (rising_edge)                      |
      | Reset           | RESET (positive)                       |
      | Reset type      | asynchronous                           |
      | Reset State     | clear                                  |
      | Power Up State  | clear                                  |
      | Encoding        | one-hot                                |
      | Implementation  | LUT                                    |
      ------------------------------------------------------------
    Summary:
        inferred   1 Finite State Machine(s).
Unit <statmach> synthesized.


Synthesizing Unit <stopwatch>.
    Related source file is C:/temp/timer/stopwatch.vhd.
WARNING:Xst:646 - Signal <strtstopinv> is assigned but never used.
Unit <stopwatch> synthesized.


=========================================================================
*                         Low Level Synthesis                           *
=========================================================================

Optimizing unit <stopwatch> ...

Optimizing unit <cnt60> ...

Optimizing unit <smallcntr> ...

Optimizing unit <statmach> ...

Optimizing unit <decode> ...

Optimizing unit <hex2led> ...
Loading device for application Rf_Device from file '2v40.nph' in environment C:Xilinx.

Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block stopwatch, actual ratio is 11.
```

```
========================================================================
*                          Final Report                               *
========================================================================
Final Results
RTL Top Level Output File Name    : stopwatch.ngr
Top Level Output File Name        : stopwatch
Output Format                     : NGC
Optimization Goal                 : Speed
Keep Hierarchy                    : YES

Design Statistics
# IOs                             : 27

Macro Statistics :
# ROMs                            : 3
#     16x10-bit ROM               : 1
#     16x7-bit ROM                : 2
# Registers                       : 2
#     4-bit register              : 2

Cell Usage :
# BELS                            : 53
#     GND                         : 1
#     LUT1                        : 12
#     LUT2                        : 3
#     LUT3                        : 4
#     LUT3_L                      : 2
#     LUT4                        : 26
#     LUT4_L                      : 5
# FlipFlops/Latches               : 14
#     FDC                         : 5
#     FDCE                        : 8
#     FDP                         : 1
# Clock Buffers                   : 1
#     BUFGP                       : 1
# IO Buffers                      : 26
#     IBUF                        : 2
#     OBUF                        : 24
# Others                          : 1
#     tenths                      : 1
=======================================================================

Device utilization summary:
---------------------------


Selected Device : 2v40cs144-6

 Number of Slices:                    30  out of    256    11%
 Number of Slice Flip Flops:          14  out of    512     2%
 Number of 4 input LUTs:              52  out of    512    10%
 Number of bonded IOBs:               26  out of     88    29%
 Number of GCLKs:                      1  out of     16     6%
```

```
========================================================================
TIMING REPORT

NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.

Clock Information:
------------------
----------------------------------+-----------------------+-------+
Clock Signal                      | Clock buffer(FF name) | Load  |
----------------------------------+-----------------------+-------+
CLK                               | BUFGP                 | 14    |
----------------------------------+-----------------------+-------+

Timing Summary:
---------------
Speed Grade: -6

   Minimum period: 3.321ns (Maximum Frequency: 301.159MHz)
   Minimum input arrival time before clock: 2.779ns
   Maximum output required time after clock: 5.835ns
   Maximum combinational path delay: 6.306ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

========================================================================
Timing constraint: Default period analysis for Clock 'CLK'
  Clock period: 3.321ns (frequency: 301.159MHz)
  Total number of paths / destination ports: 59 / 21
-----------------------------------------------------------------------
Delay:               3.321ns (Levels of Logic = 4)
  Source:            MACHINE/current_state_FFd3 (FF)
  Destination:       sixty/msbcount/qoutsig_2 (FF)
  Source Clock:      CLK rising
  Destination Clock: CLK rising

  Data Path: MACHINE/current_state_FFd3 to sixty/msbcount/qoutsig_2
                                 Gate    Net
  Cell:in->out          fanout   Delay   Delay     Logical Name (Net Name)
  ---------------------------------------------    ------------
  FDC:C->Q                  3    0.449   0.701     current_state_FFd3 (CLKEN)
  end scope: 'MACHINE'
  LUT2:I1->O                5    0.347   0.735     cnt60enable1 (cnt60enable)
  begin scope: 'sixty'
  LUT2:I1->O                4    0.347   0.553     sbce2 (msbce)
  begin scope: 'msbcount'
  FDCE:CE                        0.190             qoutsig_0
  ---------------------------------------------
  Total                          3.321ns   (1.333ns logic, 1.988ns route)
                                           (40.1% logic, 59.9% route)
```

```
===================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'CLK'
  Total number of paths / destination ports: 13 / 13
-------------------------------------------------------------------
Offset:              2.779ns (Levels of Logic = 3)
  Source:            XCOUNTER:Q_THRESH0 (PAD)
  Destination:       sixty/msbcount/qoutsig_2 (FF)
  Destination Clock: CLK rising

  Data Path: XCOUNTER:Q_THRESH0 to sixty/msbcount/qoutsig_2
                                 Gate      Net
  Cell:in->out          fanout   Delay     Delay   Logical Name (Net Name)
  -------------------------------------------     ------------
  tenths:Q_THRESH0         1     0.000     0.608   XCOUNTER (xtermcnt)
    LUT2:I0->O             5     0.347     0.735   cnt60enable1 (cnt60enable)
    begin scope: 'sixty'
    LUT2:I1->O             4     0.347     0.553   msbce2 (msbce)
    begin scope: 'msbcount'
    FDCE:CE                      0.190             qoutsig_0
  ----------------------------------------------
  Total                          2.779ns   (0.884ns logic, 1.895ns route)
                                           (31.8% logic, 68.2% route)


===================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'CLK'
  Total number of paths / destination ports: 58 / 16
-------------------------------------------------------------------
Offset:              5.835ns (Levels of Logic = 4)
  Source:            sixty/lsbcount/qoutsig_0 (FF)
  Destination:       ONESOUT<6> (PAD)
  Source Clock:      CLK rising

Data Path: sixty/lsbcount/qoutsig_0 to ONESOUT<6>
                                 Gate      Net
  Cell:in->out          fanout   Delay     Delay   Logical Name (Net Name)
  -------------------------------------------     ------------
FDCE:C->Qhttp:           12     0.449     0.914   qoutsig_0 (QOUT<0>)
end scope: 'lsbcount'
end scope: 'sixty'
begin scope: 'lsbled'
LUT4:I0->O                1     0.347     0.383   Mrom_LED_inst_lut4_101(LED<0>)
end scope: 'lsbled'
OBUF:I->O                      3.743             ONESOUT_0_OBUF(ONESOUT<0>)
-------------------------------------------------
Total                          5.835ns   (4.539ns logic, 1.296ns route)
                                          (77.8% logic, 22.2% route)
```

```
=====================================================================
Timing constraint: Default path analysis
   Total number of paths / destination ports: 41 / 11
--------------------------------------------------------------------
Delay:               6.306ns (Levels of Logic = 4)
   Source:           XCOUNTER:Q<1> (PAD)
   Destination:      TENTHSOUT<9> (PAD)

Data Path: XCOUNTER:Q<1> to TENTHSOUT<9>
                               Gate      Net
Cell:in->out          fanout   Delay     Delay     Logical Name (Net Name)
------------------------------------------------  ------------
tenths:Q<1>              10     0.000     0.000     XCOUNTER (Q<1>)
begin scope: 'decoder'
LUT4:I0->O               1      0.347     0.608     Mrom_one_hot_inst_lut4_91(one_hot<9>)
end scope: 'decoder'
LUT1:I0->O               1      0.347     0.383     TENTHSOUT<9>1(TENTHSOUT_9_OBUF)
OBUF:I->O                       3.743               TENTHSOUT_9_OBUF(TENTHSOUT<9>)
------------------------------------------------
Total                           6.306ns (5.316ns logic, 0.990ns route)
                                        (84.3% logic, 15.7% route)


=====================================================================
CPU : 5.04 / 6.55 s | Elapsed : 5.00 / 6.00 s

-->


Total memory usage is 102576 kilobytes

Number of errors   :    0 (   0 filtered)
Number of warnings :    3 (   0 filtered)
Number of infos    :    0 (   0 filtered)
```

CPLD Log File

The following is an example of an XST log file for CPLD synthesis.

```
Release 7.1i - xst H.37
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) HDL Synthesis
  5) Advanced HDL Synthesis
     5.1) HDL Synthesis Report
  6) Low Level Synthesis
  7) Final Report


=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                  : stopwatch.prj
Input Format                     : mixed
Ignore Synthesis Constraint File : NO
Verilog Include Directory        :

---- Target Parameters
Output File Name                 : stopwatch
Output Format                    : NGC
Target Device                    : xc9500xl

---- Source Options
Top Module Name                  : stopwatch
Automatic FSM Extraction         : YES
FSM Encoding Algorithm           : Auto
Mux Extraction                   : YES
Resource Sharing                 : YES

---- Target Options
Add IO Buffers                   : YES
Equivalent register Removal      : YES
MACRO Preserve                   : YES
XOR Preserve                     : YES

---- General Options
Optimization Goal                : Speed
Optimization Effort              : 1
Keep Hierarchy                   : YES
RTL Output                       : Yes
Hierarchy Separator              : _
Bus Delimiter                    : <>
Case Specifier                   : maintain

---- Other Options
lso                              : stopwatch.lso
verilog2001                      : YES
Clock Enable                     : YES
wysiwyg                          : NO
=========================================================================
```

```
=========================================================================
*                          HDL Compilation                              *
=========================================================================
Compiling vhdl file C:/temp/timer/smallcntr.vhd in Library work.
Architecture inside of Entity smallcntr is up to date.
Compiling vhdl file C:/temp/timer/statmach.vhd in Library work.
Architecture inside of Entity statmach is up to date.
Compiling vhdl file C:/temp/timer/decode.vhd in Library work.
Architecture behavioral of Entity decode is up to date.
Compiling vhdl file C:/temp/timer/cnt60.vhd in Library work.
Architecture inside of Entity cnt60 is up to date.
Compiling vhdl file C:/temp/timer/hex2led.vhd in Library work.
Architecture hex2led_arch of Entity hex2led is up to date.
Compiling vhdl file C:/temp/timer/stopwatch.vhd in Library work.
Architecture inside of Entity stopwatch is up to date.


=========================================================================
*                          HDL Analysis                                 *
=========================================================================
Analyzing Entity <stopwatch> (Architecture <inside>).
WARNING:Xst:766 - C:/temp/timer/stopwatch.vhd line 68: Generating a Black Box for component
<tenths>.
Entity <stopwatch> analyzed. Unit <stopwatch> generated.

Analyzing Entity <statmach> (Architecture <inside>).
Entity <statmach> analyzed. Unit <statmach> generated.

Analyzing Entity <decode> (Architecture <behavioral>).
Entity <decode> analyzed. Unit <decode> generated.

Analyzing Entity <cnt60> (Architecture <inside>).
Entity <cnt60> analyzed. Unit <cnt60> generated.

Analyzing Entity <smallcntr> (Architecture <inside>).
Entity <smallcntr> analyzed. Unit <smallcntr> generated.

Analyzing Entity <hex2led> (Architecture <hex2led_arch>).
Entity <hex2led> analyzed. Unit <hex2led> generated.



=========================================================================
*                          HDL Synthesis                                *
=========================================================================

Synthesizing Unit <smallcntr>.
  Related source file is C:/temp/timer/smallcntr.vhd.
    Found 4-bit up counter for signal <qoutsig>.
    Summary:
       inferred   1 Counter(s).
Unit <smallcntr> synthesized.
```

```
Synthesizing Unit <hex2led>.
    Related source file is C:/temp/timer/hex2led.vhd.
    Found 16x7-bit ROM for signal <LED>.
    Summary:
        inferred   1 ROM(s).
Unit <hex2led> synthesized.


Synthesizing Unit <cnt60>.
    Related source file is C:/temp/timer/cnt60.vhd.
Unit <cnt60> synthesized.


Synthesizing Unit <decode>.
    Related source file is C:/temp/timer/decode.vhd.
    Found 16x10-bit ROM for signal <one_hot>.
    Summary:
        inferred   1 ROM(s).
Unit <decode> synthesized.


Synthesizing Unit <statmach>.
    Related source file is C:/temp/timer/statmach.vhd.
    Found finite state machine <FSM_0> for signal <current_state>.
    +--------------------------------------------------------------------+
    | States              | 6                                            |
    | Transitions         | 11                                           |
    | Inputs              | 1                                            |
    | Outputs             | 2                                            |
    | Clock               | CLK (rising_edge)                            |
    | Reset               | RESET (positive)                             |
    | Reset type          | asynchronous                                 |
    | Reset State         | clear                                        |
    | Power Up State      | clear                                        |
    | Encoding            | automatic                                    |
    | Implementation      | automatic                                    |
    +--------------------------------------------------------------------+
    Summary:
        inferred   1 Finite State Machine(s).
Unit <statmach> synthesized.


Synthesizing Unit <stopwatch>.
    Related source file is C:/temp/timer/stopwatch.vhd.
WARNING:Xst:646 - Signal <strtstopinv> is assigned but never used.
Unit <stopwatch> synthesized.


=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================

Advanced RAM inference ...
Advanced multiplier inference ...
Analyzing FSM <FSM_0> for best encoding.
Dynamic shift register inference ...
```

```
========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                           : 3
 16x10-bit ROM                   : 1
 16x7-bit ROM                    : 2
# Counters                       : 2
 4-bit up counter                : 2
# Registers                      : 3
 1-bit register                  : 3


========================================================================

========================================================================
*                       Low Level Synthesis                            *
========================================================================
Optimizing FSM <FSM_0> on signal <current_state[1:3]> with sequential encoding.
--------------------
State     | Encoding
--------------------
clear     | 000
zero      | 001
start     | 010
counting  | 011
stop      | 100
stopped   | 101
--------------------

Optimizing unit <stopwatch> ...

Optimizing unit <cnt60> ...

Optimizing unit <smallcntr> ...

Optimizing unit <hex2led> ...

Optimizing unit <statmach> ...

Optimizing unit <decode> ...

========================================================================
*                       Final Report                                   *
========================================================================
Final Results
RTL Top Level Output File Name     : stopwatch.ngr
Top Level Output File Name         : stopwatch
Output Format                      : NGC
Optimization Goal                  : Speed
Keep Hierarchy                     : YES
Target Technology                  : xc9500xl
Macro Preserve                     : YES
XOR Preserve                       : YES
Clock Enable                       : YES
wysiwyg                            : NO

Design Statistics
# IOs                              : 27
```

```
Macro Statistics :
# Registers                         : 8
#       1-bit register             : 8
# Xors                              : 6
#       1-bit xor2                 : 6

Cell Usage :
# BELS                              : 390
#       AND2                       : 108
#       AND3                       : 8
#       AND4                       : 2
#       INV                        : 167
#       OR2                        : 97
#       OR3                        : 1
#       XOR2                       : 7
# FlipFlops/Latches                : 11
#       FDC                        : 3
#       FDCE                       : 8
# IO Buffers                       : 27
#       IBUF                       : 3
#       OBUF                       : 24
# Others                           : 1
#       tenths                     : 1
=======================================================================
CPU : 3.57 / 4.88 s | Elapsed : 3.00 / 4.00 s

-->


Total memory usage is 87156 kilobytes

Number of errors   :    0 (   0 filtered)
Number of warnings :    3 (   0 filtered)
Number of infos    :    0 (   0 filtered)
```

Chapter 10

Command Line Mode

This chapter describes how to run XST using the command line. The chapter contains the following sections.

- *"Introduction"*
- *"Launching XST"*
- *"Setting Up an XST Script"*
- *"Run Command"*
- *"Getting Help"*
- *"Set Command"*
- *"Elaborate Command"*
- *"Example 1: How to Synthesize VHDL Designs Using Command Line Mode"*
- *"Example 2: How to Synthesize Verilog Designs Using Command Line Mode"*
- *"Example 3: How to Synthesize Mixed VHDL/Verilog Designs Using Command Line Mode"*

Introduction

You can run synthesis with XST in command line mode instead of from the Process window in Project Navigator. To run synthesis from the command line, you must use the XST executable file. If you work on a workstation, the name of the executable is "xst". On a PC, the name of the executable is "xst.exe".

File Types

XST generates the following types of files:

- Design output file, NGC (.ngc)

  This file is generated in the current output directory (see the –ofn option). If run in incremental synthesis mode, XST generates multiple NGC files.

- RTL netlist for RTL and Technology Viewers (.ngr)
- Synthesis LOG file (.srp)
- Temporary files

  Temporary files are generated in the XST temp directory. By default the XST temp directory is /tmp on workstations and the directory specified by either the TEMP or TMP environment variables under Windows. The XST temp directory can be changed by using the **set –tmpdir** <*directory*> directive.

- VHDL/Verilog compilation files

  VHDL/Verilog compilation files are generated in the dump directory. The default dump directory is the "xst" subdirectory of the current directory.

Note: Xilinx strongly suggests that you *clean the XST temp directory* regularly. This directory contains the files resulting from the compilation of *all VHDL and Verilog* files during all XST sessions. Eventually, the number of files stored in the dump directory may severely impact CPU performances. This directory is not automatically cleaned by XST.

Names with Spaces

Starting in release 7.1i, XST supports file and directory names with spaces. If a file or directory name contains spaces, you must enclose this name in double quotes (**""**) as in the following example.

```
"C:\my project"
```

Due to this change, the command line syntax for switches supporting multiple directories (**–sd**,**–vlgincdir**) has changed. If multiple directories are specified for these switches, you must enclose them in braces (**{}**) as in the following example.

```
–vlgincdir {"C:\my project" C:\temp}
```

Note: In previous releases multiple directories were included in double quotes (**""**). This previous convention is still supported by XST if directory names do not contain spaces. Xilinx strongly suggests that you change existing scripts to the new syntax.

Launching XST

You can run XST in two ways.

- XST Shell — Type **xst** to enter directly into an XST shell. Enter your commands and execute them. To run synthesis, specify a complete command with all required options before running. XST does not accept a mode where you can first enter **set** *option_1*, then **set** *option_2*, and then enter **run**.

  All of the options must be set up at once. Therefore, this method is very cumbersome and Xilinx suggests that you use the script file method.

- Script File — You can store your commands in a separate script file and run all of them at once. To execute your script file, run the following workstation or PC command:

  ```
  xst –ifn in_file_name –ofn out_file_name –intstyle {silent|ise|xflow}
  ```

Note: The –ofn option is not mandatory. If you omit it, XST automatically generates a log file with the file extension .srp, and all messages display on the screen. Use the –intstyle silent option and the XIL_XST_HIDEMESSAGES environment variable to limit the number of messages printed to the screen. See the "Reducing the Size of the LOG File" in Chapter 9 for more information.

For example, assume that the text below is contained in a file foo.scr.

```
run
–ifn tt1.prj
–top tt1
–ifmt MIXED
–opt_mode SPEED
–opt_level 1
–ofn tt1.ngc
–p <parttype>
```

This script file can be executed under XST using the following command:

```
xst –ifn foo.scr
```

You can also generate a log file with the following command:

```
xst –ifn foo.scr –ofn foo.log
```

A script file can be run either using **xst –ifn** *script name*, or executed under the XST prompt, by using the **script** *script_name* command.

```
script foo.scr
```

If you make a mistake in an XST command, command option or its value, XST issues an error message and stops execution. For example, if in the previous script example VHDL is incorrectly spelled (VHDLL), XST gives the following error message:

```
--> ERROR:Xst:1361 - Syntax error in command run for option "-ifmt" :
parameter "VHDLL" is not allowed.
```

Setting Up an XST Script

An XST script is a set of commands, each command having various options. XST recognizes the following commands:

- run
- set
- elaborate

Run Command

Following is a description of the run command.

- The command begins with a keyword **run**, which is followed by a set of options and its values.

  run *option_1 value option_2 value ...*

- Each option name starts with dash (–). For instance: –ifn, –ifmt, –ofn.
- Each option has one value. There are no options without a value.
- The value for a given option can be one of the following:
  - ♦ Predefined by XST (for instance, yes or no).
  - ♦ Any string (for instance, a file name or a name of the top level entity). There are options like –vlgincdir that accept several directories as values. The directories

must be separated by spaces, and enclosed altogether by braces ({}) as in the following example.

```
–vlgincdir {c:\vlg1 c:\vlg2}
```

Please refer to for more information.

♦ An integer.

In the following tables, you can find the name of each option and its values.

- First column — the name of the options you can use in command line mode. If the option is in **bold**, it must be present in the command line.
- Second column — the option description.
- Third column — the possible values of this option. The values in **bold** are the default values.

Note: See Chapter 5, "Design Constraints" for more information about the options listed in the following tables.

.

Table 10-1: **Global Options**

Run Command Options	Description	Values
–ifn	Input/Project File Name	*file_name*
–ifmt	Input Project Format	VHDL, Verilog, **mixed**
–top	Top Level Block Name	*block_name*
–work_lib	Work Library — library where the top level block was compiled	*dir_name*, **work**
–lso	Library Search Order	*file_name*.lso
–ofn	Output File Name	*file_name*
–ofmt	Output File Format	**ngc**
–duplication_suffix	Duplication Suffix	*string*%d*string*
–case	Case	upper, lower, **maintain**
–hierarchy_separator	Hierarchy Separator	_ , /
–opt_mode	Optimization Goal	area, **speed**
–opt_level	Optimization Effort	**1**, 2
–p	Target Technology	*part-package-speed* for example: xcv50-fg456-5 : xcv50-fg456-6
–rtlview	Generate RTL Schematic	yes, **no**, only
–iuc	Ignore User Constraints	yes, **no**
–uc	Synthesis Constraints File	*file_name*.**xcf**

Table 10-1: **Global Options**

Run Command Options	Description	Values
–bus_delimiter	Bus Delimiter	**<>**, [], {}, ()
–enable_auto_floorplanning	Enable Auto Floorplanning	**no**, incremental design
–hdl_compilation_order	Respect HDL file order, specified in XST project during compilation step.	**auto**, user

Table 10-2: **VHDL Source Options**

Run Command Options	Description	Values
–ent	Entity Name	*name* **Note:** Valid only when old VHDL project format is used (–ifmt VHDL). Please use project format (–ifmt mixed) and –top switch to specify which top level block to synthesize.
–arch	Architecture	*name*

Table 10-3: **Verilog Source Options**

Run Command Options	Description	Values
–top	Top Module Name	*name* **Note:** **Not** Valid when old VHDL Project format is used (–ifmt VHDL).
–vlgcase	Case Implementation Style	full, parallel, full-parallel
–vlgincdir	Verilog Include Directories	Any valid path to directories separated by spaces, and enclosed in braces ({})
–verilog2001	Verilog-2001	**yes**, no

Table 10-4: **HDL Synthesis Options (VHDL and Verilog)**

Run Command Options	Description	Values
–fsm_extract	Automatic FSM Extraction	**yes**, no
–fsm_style	FSM Style	**lut**, bram
–fsm_encoding	Encoding Algorithm	**auto**, one-hot, compact, sequential, gray, johnson, user

Table 10-4: **HDL Synthesis Options (VHDL and Verilog)**

Run Command Options	Description	Values
–ram_extract	RAM Extract	**yes**, no
–ram_style	RAM Style	**auto**, distributed, block
–rom_extract	ROM Extract	**yes**, no
–rom_style	ROM Style	**auto**, distributed, block
–mult_style	Multiplier Style	**auto**, block, lut, pipe_lut
–mux_extract	Mux Extraction	**yes**, no, force
–mux_style	Mux Style	**auto**, MUXF, MUXCY
–decoder_extract	Decoder Extraction	**yes**, no
–priority_extract	Priority Encoder Extraction	**yes**, no, force
–safe_implementation	Safe Implementation	**yes**, no
–shreg_extract	Shift Register Extraction	**yes**, no
–shift_extract	Logical Shift Extraction	**yes**, no
–xor_collapse	XOR Collapsing	**yes**, no
–resource_sharing	Resource Sharing	**yes**, no
–use_dsp48	Use DSP48 Resources	**auto,** yes, no

Table 10-5: **Target Options (9500, 9500XL, 9500XV, XPLA3, CoolRunner-II™)**

Run Command Options	Description	Values
–iobuf	Add I/O Buffers	**yes**, no
–pld_mp	Macro Preserve	**yes**, no
–pld_xp	XOR Preserve	**yes**, no
–keep_hierarchy	Keep Hierarchy	**yes**, soft, no
–pld_ce	Clock Enable	**yes**, no
–pld_ffopt	Flip-Flop Optimization	**yes**, no
–wysiwyg	What You See Is What You Get	yes, **no**
–equivalent_- register_removal	Equivalent Register Removal	**yes**, no

Table 10-6: **Target Options (Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X, Virtex-4™, Spartan-II™, Spartan-IIE™, Spartan-3™)**

Run Command Options	Description	Values
–bufg	Maximum Number of BUFGs created by XST	*integer* — Default **4**: Virtex/E, Spartan-II/E — Default **8**: Spartan-3 — Default **16**: Virtex-II/II Pro/II Pro X — Default **32**: Virtex-4
–bufr	Maximum Number of BUFRs created by XST	*integer* — Default value depends on the target device of Virtex-4 family
–cross_clock_analysis	Enable cross clock domain optimization	yes, **no**
–equivalent_register_removal	Equivalent Register Removal	**yes**, no
–glob_opt	Global Optimization Goal	**allclocknets**, inpad_to_outpad, offset_in_before, offset_out_after, max_delay
–iob	Pack I/O Registers into IOBs	true, false, **auto**
–iobuf	Add I/O Buffers	**yes**, no
–keep_hierarchy	Keep Hierarchy	yes, soft, **no**
–max_fanout	Maximum Fanout	*integer* —Default **500** for Virtex™ -II/-II Pro/-II Pro X /-4, Spartan-3™ —Default **100** for Virtex™, Virtex -E™, Spartan-II™ and Spartan-IIE™
–optimize_primitives	Optimize Instantiated Primitives	yes, **no**
–read_cores	Read Cores	**yes**, no
–register_balancing	Register Balancing	yes, **no**, forward, backward

Table 10-6:   **Target Options (Virtex™, Virtex-E™, Virtex-II™, Virtex-II Pro™, Virtex-II Pro™ X, Virtex-4™, Spartan-II™, Spartan-IIE™, Spartan-3™)**

Run Command Options	Description	Values
–move_first_stage	Move First Flip-Flop Stage	**yes**, no
–move_last_stage	Move Last Flip-Flop Stage	**yes**, no
–register_duplication	Register Duplication	**yes**, no
–sd	Cores Search Directories	Any valid path to directories separated by spaces, and enclosed in braces ({})
–slice_packing	Slice Packing	**yes**, no
–slice_utilization_ratio	Slice Utilization Ratio	*integer* (Default **100**)
–slice_utilization_ratio_-maxmargin	Slice Utilization Ratio Delta	*integer* (Default **5**)
–tristate2logic	Replace Internal Tristates with Logic	**yes**, no
–use_clock_enable	Use Clock Enable	**auto**, yes, no
–use_sync_set	Use Synchronous Set	**auto**, yes, no
–use_sync_reset	Use Synchronous Reset	**auto**, yes, no
–write_timing_constraints	Write Timing Constraints	yes, **no**

The following options have become obsolete for the current version of XST.

Table 10-7:   **Obsolete Options**

Run Command Options	Description	Values
–complex_clken	Complex Clock Enable	**yes**, no

Getting Help

If you are working from the command line on a Unix system, XST provides an online Help function. The following information is available by typing *help* at the command line. XST's help function can give you a list of supported families, available commands, switches and their values for each supported family.

- To get a detailed explanation of an XST command, use the following syntax.

  ```
  help –arch family_name –command command_name
  ```

  where:

  ♦ *family_name* is a list of supported Xilinx® families in the current version of XST.

  ♦ *command_name* is one of the following XST commands: **run**, **set**, **elaborate**, **time**.

- To get a list of supported families, type *help* at the command line prompt with no argument. XST displays the following message.

```
--> help
```

```
ERROR:Xst:1356 – Help : Missing "–arch <family>". Please specify what
family you want to target
```

```
available families:
```

```
spartan3
```

```
spartan2
```

```
spartan2e
```

```
virtex
```

```
virtex2
```

```
virtex2p
```

```
virtex4
```

```
virtexe
```

```
xbr
```

```
xc9500
```

```
xc9500xl
```

```
xpla3
```

- To get a list of available commands for a specific family, type the following at the command line prompt with no argument.

```
help –arch family_name.
```

For example:

```
help –arch virtex
```

Example

Use the following command to get a list of available options and values for the run command for Virtex-II™.

```
--> help -arch virtex2 -command run
```

This command gives the following output.

```
-mult_style             : Multiplier Style
        block / lut / auto / pipe_lut
-bufg                   : Maximum Global Buffers
        *
-bufgce                 : BUFGCE Extraction
        YES / NO
-decoder_extract        : Decoder Extraction
        YES / NO
....

-ifn : *
-ifmt : Mixed / VHDL / Verilog
-ofn : *
-ofmt : NGC / NCD
-p : *
-ent : *
-top : *
-opt_mode : AREA / SPEED
-opt_level : 1 / 2
-keep_hierarchy : YES / NO
-vlgincdir : *
-verilog2001 : YES / NO
-vlgcase : Full / Parallel / Full-Parallel
....
```

Set Command

In addition to the run command, XST also recognizes the set command. This command accepts the options shown in the following table.

Note: See Chapter 5, "Design Constraints" for more information about the options listed in this table.

Table 10-8: **Set Command Options**

Set Command Options	Description	Values
–tmpdir	Location of all temporary files generated by XST during a session	Any valid path to a directory
–dumpdir	Location of all files resulting from VHDL compilation	Any valid path to a directory
–xsthdpdir	Work Directory — location of all files resulting from VHDL/Verilog compilation	Any valid path to a directory
–xsthdpini	HDL Library Mapping File (.INI File)	*file_name*

Elaborate Command

The goal of this command is to pre-compile VHDL/Verilog files in a specific library or to verify Verilog files without synthesizing the design. Taking into account that the compilation process is included in the "run", this command remains optional.

The elaborate command accepts the options shown in the following table.

Note: See Chapter 5, "Design Constraints" for more information about the options listed in this table.

Table 10-9: **Elaborate Command Options**

Elaborate Command Options	Description	Values
–ifn	Project File	*file_name*
–ifmt	Format	vhdl, verilog, **mixed**
–lso	Library Search Order	*file_name*.lso
–work_lib	Work Library for Compilation — library where the top level block was compiled	*name*, **work**
–verilog2001	Verilog-2001	**yes**, no
–vlgincdir	Verilog Include Directories	Any valid path to directories separated by spaces, and enclosed in braces ({})

Example 1: How to Synthesize VHDL Designs Using Command Line Mode

The goal of this example is to synthesize a hierarchical VHDL design for a Virtex™ FPGA using Command Line Mode.

The example uses a VHDL design, called watchvhd. The files for watchvhd can be found in the `ISEexamples\watchvhd` directory of the ISE™ installation directory.

This design contains 7 entities:

- stopwatch
- statmach
- tenths (a CORE Generator™ core)
- decode
- smallcntr
- cnt60
- hex2led

Example 1

1. Create a new directory, named `vhdl_m`.

2. Copy the following files from the `ISEexamples\watchvhd` directory of the ISE™ installation directory to the newly created `vhdl_m` directory.

   ♦ `stopwatch.vhd`

   ♦ `statmach.vhd`

   ♦ `decode.vhd`

   ♦ `cnt60.vhd`

   ♦ `smallcntr.vhd`

   ♦ `tenths.vhd`

   ♦ `hex2led.vhd`

To synthesize the design, which is now represented by seven VHDL files, create a project.

Please note that starting from the 6.1i release, XST supports Mixed VHDL/Verilog projects and therefore, Xilinx strongly suggests that you use the new project format whether it is a real mixed language project or not. In this example we use the new project format. To create a project file containing only VHDL files, place a list of VHDL files preceded by keyword *VHDL* in a separate file. The order of the files is not important. XST can recognize the hierarchy, and compile VHDL files in the correct order.

For the example, perform the following steps:

1. Open a new file, called `watchvhd.prj`

2. Enter the names of the VHDL files in any order into this file and save the file:

   ```
   vhdl work statmach.vhd
   vhdl work decode.vhd
   vhdl work stopwatch.vhd
   vhdl work cnt60.vhd
   vhdl work smallcntr.vhd
   vhdl work tenths.vhd
   vhdl work hex2led.vhd
   ```

3. To synthesize the design, execute the following command from XST shell or via script file:

   ```
   run –ifn watchvhd.prj –ifmt mixed –top stopwatch –ofn watchvhd.ngc –
   ofmt NGC
        –p xcv50-bg256-6 –opt_mode Speed –opt_level 1
   ```

   Note: It is mandatory to specify a top-level design block via the **–top** command line switch.

   If you want to synthesize just "hex2led" and check its performance independently of the other blocks, you can specify the top-level entity to synthesize in the command line, using the –top option (please refer to Table 10-2, page 385 for more details):

   ```
   run –ifn watchvhd.prj –ifmt mixed –ofn watchvhd.ngc -ofmt NGC
       -p xcv50-bg256-6 –opt_mode Speed -opt_level 1 –top hex2led
   ```

During VHDL compilation, XST uses the library "work" as the default. If some VHDL files must be compiled to different libraries, then you can add the name of the library just before the file name. Suppose that "hexl2led" must be compiled into the library, called `my_lib`. Then the project file must be:

```
vhdl work statmach.vhd
vhdl work decode.vhd
vhdl work stopwatch.vhd
vhdl work cnt60.vhd
vhdl work smallcntr.vhd
vhdl work vhdl tenths.vhd
vhdl my_lib work hex2led.vhd
```

Sometimes, XST is not able to recognize the order and issues the following message.

```
WARNING:XST:3204. The sort of the vhdl files failed, they will be
compiled in the order of the project file.
```

In this case you must do the following:

- Put all VHDL files in the correct order.
- Add **–hdl_compilation_order** switch with value **user** to the XST run command:

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc
   -ofmt NGC -p xcv50-bg256-6 -opt_mode Speed -opt_level 1 -top hex2led
   -hdl_compilation_order user
```

Script Mode

It can be very tedious work to enter XST commands directly in the XST shell, especially when you have to specify several options and execute the same command several times. You can run XST in a script mode as follows:

1. Open a new file named `xst.txt` in the current directory. Put the previously executed XST shell command into this file and save it.

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc
   -ofmt NGC -p xcv50-bg256-6 -opt_mode Speed -opt_level 1
```

2. From the tcsh or other shell, enter the following command to start synthesis.

```
xst -ifn stopwatch.xst
```

During this run, XST creates the following files.

♦ `watchvhd.ngc`: an NGC file ready for the implementation tools
♦ `xst.srp`: the xst log file

3. If you want to save XST messages in a different log file, for example, `watchvhd.log`, execute the following command.

```
xst -ifn stopwatch.xst -ofn watchvhd.log
```

You can improve the readability of the `xst.txt` file, especially if you use many options to run synthesis, by placing each option with its value on a separate line, respecting the following rules:

- The first line must contain only the run command without any options.
- There must be no blank lines in the middle of the command.
- Each line (except the first one) must start with a dash (–).

For the previous command example, `xst.scr` should look like the following:

```
run
-ifn watchvhd.vhd
-ifmt mixed
-top stopwatch
-ofn watchvhd.ngc
-ofmt NGC
-p xcv50-bg256-6
-opt_mode Speed
-opt_level 1
```

Example 2: How to Synthesize Verilog Designs Using Command Line Mode

The goal of this example is to synthesize a hierarchical Verilog design for a Virtex™ FPGA using Command Line Mode.

Example 2 uses a Verilog design, called watchver. These files can be found in the `ISEexamples\watchver` directory of the ISE™ installation directory.

- `stopwatch.v`
- `statmach.v`
- `decode.v`
- `cnt60.v`
- `smallcntr.v`
- `tenths.v`
- `hex2led.v`

This design contains seven modules:

- stopwatch
- statmach
- tenths (a CORE Generator™ core)
- decode
- cnt60
- smallcntr
- hex2led

Example 2

1. Create a new directory named `vlg_m`.
2. Copy the watchver design files from the `ISEexamples\watchver` directory of the ISE™ installation directory to the newly created `vlg_m` directory.

   Note: It is mandatory to specify the top-level design block via the **−top** command line switch.

To synthesize the design, which is now represented by seven Verilog files, create a project. Please note that starting from the 6.1i release XST supports Mixed VHDL/Verilog projects and therefore, Xilinx strongly suggests that you use the new project format whether it is a

real mixed language project or not. In this example, we use the new project format. To create a project file containing only Verilog files, place a list of Verilog files preceded by the keyword *verilog* in a separate file. The order of the files is not important. XST can recognize the hierarchy and compile Verilog files in the correct order. For our example:

1. Open a new file, called watch`ver.v`.

2. Enter the names of the Verilog files into this file in any order and save it:

   ```
   verilog work decode.v
   verilog work statmach.v
   verilog work stopwatch.v
   verilog work cnt60.v
   verilog work smallcntr.v
   verilog work hex2led.v
   ```

3. To synthesize the design, execute the following command from the XST shell or via a script file:

   ```
   run –ifn watchver.v –ifmt mixed -top stopwatch –ofn watchver.ngc
      –ofmt NGC –p xcv50-bg256-6 –opt_mode Speed –opt_level 1
   ```

If you want to synthesize just HEX2LED and check its performance independently of the other blocks, you can specify the top-level module to synthesize in the command line, using the –top option (please refer to Table 10-3, page 385 for more information):

   ```
   run –ifn watchver.v –ifmt Verilog –ofn watchver.ngc –ofmt NGC
      –p xcv50-bg256-6 –opt_mode Speed –opt_level 1 –top HEX2LED
   ```

Script Mode

It can be very tedious work entering XST commands directly into the XST shell, especially when you have to specify several options and execute the same command several times. You can run XST in script mode as follows.

1. Open a new file called `design.xst` in the current directory. Put the previously executed XST shell command into this file and save it.

   ```
   run -ifn watchver.prj –ifmt mixed –ofn watchver.ngc
      –ofmt NGC -p xcv50-bg256-6 -opt_mode Speed -opt_level 1
   ```

2. From the tcsh or other shell, enter the following command to start synthesis.

   ```
   xst –ifn design.xst
   ```

   During this run, XST creates the following files.

   ♦ `watchvhd.ngc`: an NGC file ready for the implementation tools

   ♦ `design.srp`: the xst script log file

3. If you want to save XST messages in a different log file, for example, `watchver.log`, execute the following command.

   ```
   xst –ifn design.xst -ofn watchver.log
   ```

You can improve the readability of the `design.xst` file, especially if you use many options to run synthesis. You can place each option with its value on a separate line, respecting the following rules:

• The first line must contain only the run command without any options.

• There must be no blank lines in the middle of the command.

• Each line (except the first one) must start with a dash (–).

For the previous command example, the stopwatch.xst file should look like the following:

```
run
-ifn watchver.prj
-ifmt mixed
-top stopwatch
-ofn watchver.ngc
-ofmt NGC
-p xcv50-bg256-6
-opt_mode Speed
-opt_level 1
```

Example 3: How to Synthesize Mixed VHDL/Verilog Designs Using Command Line Mode

The goal of this example is to synthesize a hierarchical mixed VHDL/Verilog design for a Virtex™ FPGA using Command Line Mode.

1. Create a new directory, named `vhdl_verilog`.

2. Copy the following files from the `ISEexamples\watchvhd` directory of the ISE™ installation directory to the newly created `vhdl_verilog` directory.

   ♦ stopwatch.vhd

   ♦ statmach.vhd

   ♦ decode.vhd

   ♦ cnt60.vhd

   ♦ smallcntr.vhd

   ♦ tenths.vhd

3. Copy the following file from the `ISEexamples\watchver` directory of the ISE™ installation directory to the newly created `vhdl_verilog` directory:

   ♦ hex2led.v

To synthesize the design, which is now represented by six VHDL files and one Verilog file, create a project. To create a project file, place a list of VHDL files preceded by keyword *vhdl,* and a list of Verilog files preceded by keyword *verilog* in a separate file. The order of the files is not important. XST is able to recognize the hierarchy, and compile HDL files in the correct order.

For our example:

1. Open a new file called `watchver.prj`.

2. Enter the names of the files into this file in any order and save it:

```
vhdl work decode.vhd
vhdl work statmach.vhd
vhdl work stopwatch.vhd
vhdl work cnt60.vhd
vhdl work smallcntr.vhd
vhdl work tenths.vhd
verilog work hex2led.v
```

3. To synthesize the design, execute the following command from the XST shell or via a script file:

```
run –ifn watchver.prj –ifmt mixed –top stopwatch –ofn watchver.ngc
    –ofmt NGC –p xcv50-bg256-6 –opt_mode Speed –opt_level 1
```

Note: It is mandatory to specify the top-level design block via the **–top** command line switch.

If you want to synthesize just HEX2LED and check its performance independently of the other blocks, you can specify it as the top level module to synthesize on the command line by using the –top option (please refer to for more information):

```
run –ifn watchver.prj –ifmt mixed –top hex2led –ofn watchver.ngc
    –ofmt NGC –p xcv50-bg256-6 –opt_mode Speed –opt_level 1
```

Script Mode

It can be very tedious work entering XST commands directly into the XST shell, especially when you have to specify several options and execute the same command several times. You can run XST in a script mode as follows.

1. Open a new file called `xst.txt` in the current directory. Put the previously executed XST shell command into this file and save it.

```
run –ifn watchver.prj –ifmt mixed –top stopwatch –ofn watchver.ngc
    –ofmt NGC –p xcv50-bg256-6 –opt_mode Speed –opt_level 1
```

2. From the tcsh or other shell, enter the following command to start synthesis.

```
xst –ifn stopwatch.xst
```

During this run, XST creates the following files:

♦ `watchver.ngc`: an NGC file ready for the implementation tools

♦ `xst.srp`: the xst script log file

3. If you want to save XST messages in a different log file for example, `watchver.log`, execute the following command.

```
xst –ifn stopwatch.xst –ofn watchver.log
```

You can improve the readability of the `xst.scr` file, especially if you use many options to run synthesis. You can place each option with its value on a separate line, respecting the following rules:

• The first line must contain only the run command without any options.

• There must be no blank lines in the middle of the command.

• Each line (except the first one) must start with a dash (–).

For the previous command example, the stopwatch.xst file should look like the following:

```
run
-ifn watchver.prj
-ifmt mixed
-ofn watchver.ngc
-ofmt NGC
-p xcv50-bg256-6
-opt_mode Speed
-opt_level 1
```

XST Naming Conventions

This appendix discusses net naming and instance naming conventions.The appendix contains the following sections.

- *"Net Naming Conventions"*
- *"Instance Naming Conventions"*
- *"Name Generation Control"*

Net Naming Conventions

These rules are listed in order of naming priority.

1. Maintain external pin names.
2. Keep hierarchy in signal names, using underscores as hierarchy designators.
3. Maintain output signal names of registers, including state bits. Use the hierarchical name from the level where the register was inferred.
4. Ensure that output signals of clock buffers get *_clockbuffertype* (like _BUFGP or _IBUFG) follow the clock signal name.
5. Maintain input nets to registers and tristates names.
6. Maintain names of signals connected to primitives and black boxes.
7. Name output net names of IBUFs using the form *net_name*_IBUF. For example, for an IBUF with an output net name of DIN, the output IBUF net name is DIN_IBUF.

    Name input net names to OBUFs using the form *net_name*_OBUF. For example, for an OBUF with an input net name of DOUT, the input OBUF net name is DOUT_OBUF.

Instance Naming Conventions

These rules are listed in order of naming priority.

1. Keep hierarchy in instance names, using underscores as hierarchy designators.
2. Name register instances, including state bits, for the output signal.
3. Name clock buffer instances *_clockbuffertype* (like _BUFGP or _IBUFG) after the output signal.
4. Maintain instantiation instance names of black boxes.
5. Maintain instantiation instance names of library primitives.
6. Name input and output buffers using the form _IBUF or _OBUF after the pad name.
7. Name Output instance names of IBUFs using the form *instance_name*_IBUF.

    Name input instance names to OBUFs using the form *instance_name*_OBUF.

Name Generation Control

You can control aspects of the way names are written using the following properties. You can apply these properties by either using the ISE™ Synthesis Properties dialog box or using the appropriate command line options. See Chapter 5, "Design Constraints" for more information on using these properties.

- hierarchy separator (HIERARCHY_SEPARATOR)
- bus delimiter (BUS_DELIMITER)
- case processing (CASE)
- duplication suffix (DUPLICATION_SUFFIX)

Index

A

accumulator 67
    4-bit unsigned up accumulator with asynchronous clear 68
add I/O buffers 240, 241, 246, 386, 387
adder 104
    simple signed 8-bit adder 109
    unsigned 8-bit adder 105
    unsigned 8-bit adder with carry in 105
    unsigned 8-bit adder with carry in and carry out 108
    unsigned 8-bit adder with carry out 106
adder/subtractor 104
    unsigned 8-bit adder/subtractor 110
addition 325
advanced HDL synthesis 363
alias declaration 316
allclocknets 264, 265
always statement 350
arch 385
Architecture 19
architecture 385
architecture body 314
architecture declaration 292
architecture support 19
arithmetic 325
arithmetic operation 104
array of instances 350
assert statement 310
assign statement 334
assignment extension past 32 bits 336
attribute 317
attribute declaration 316
automatic FSM extraction 253, 385

B

begin model... end 245
bit vector types 286
bitwise equivalence 326
bitwise exclusive or 326
bitwise inclusive or 326
bitwise negation 326
black box support 195
black_box 279
black_box_pad_pin 279

black_box_tri_pins 279
block multiplier 113
block RAM
    initial contents 174
block statement 328
blocking and nonblocking assignments 346
blocking assignment 349
blocking versus von-blocking 337
box type 246
box_type 195, 268
bram_map 256, 268
buffer type 254
buffer_type 254, 268
bufg 257, 387
bufgce 254, 268
bufgmux 254
bus delimiter 237, 246, 385
bus_delimiter 385, 400

C

case 237, 384, 400
case equality 326
case implementation style 87, 238, 239, 247, 385
case inequality 326
case statement 85, 87, 349
casex 330
casez 330
cell_list 279
celldefine 350
clock buffer type 255
clock enable 241, 260, 386
clock information 364
clock signal 264
clock_buffer 254, 255, 268
clock_list 279
clock_signal 264, 268
combinatorial process 299
command line 381
comparator 112
    unsigned 8-bit greater or equal comparator 112
compilation files 382
component configuration 295
component declaration 316
component instantiation 293
composite 316

concatenation 325
concurrent signal assignments 297
concurrent statement 319
conditional 327
configuration 317
configuration declaration 314
constant declaration 316
constraint file 234
constraints precedence 282
continuous procedural assignment 349
convert tristates to logic 260
CoolRunner 19
CoolRunner XPLA3 19
CoolRunner-II 19
cores search directories 237, 255, 388
counter 56
    4-bit signed up counter with asynchronous reset 64
    4-bit signed up counter with asynchronous reset and modulo maximum 66
    4-bit unsigned down counter with synchronous set 58
    4-bit unsigned up counter with asynchronous clear 57
    4-bit unsigned up counter with asynchronous clear and clock enable 62
    4-bit unsigned up counter with asynchronous load from primary input 59
    4-bit unsigned up counter with synchronous load with a constant 60
    4-bit unsigned up/down counter with asynchronous clear 63
CPLD
    constraints 260
    fitter 227, 231
    fitter multi-level optimization 231
    log file 375
    log file analysis 229
    low-level optimization 231
    synthesis options 227
    target options 386
cross clock analysis 237, 263
cross_clock_analysis 263, 387
CSD 114
custom compile file list 237, 242