XST User Guide

UG627 (v 11.1.0) April 27, 2009

XILINX®

Xilinx Trademarks and Copyright Information

Table of Contents

Chapter 1

About the XST User Guide

The *XST User Guide:*

- Describes Xilinx® Synthesis Technology (XST) support for Hardware Description Language (HDL), Xilinx devices, and design constraints for the Xilinx ISE® Design Suite software
- Discusses FPGA and CPLD optimization and coding techniques when creating designs for use with XST
- Explains how to run XST from the ISE Design Suite Process window, and from the command line

XST User Guide Contents

The *XST User Guide* includes:

- Chapter 1, About This Guide, provides an overview of the *XST User Guide*.

  Chapter 2, Introduction to Xilinx Synthesis Technology (XST) , provides general information about Xilinx Synthesis Technology (XST), and describes the changes to XST in this release.
- Chapter 3, XST Hardware Description Language (HDL) Coding Techniques, gives Hardware Description Language (HDL) coding examples for digital logic circuits.
- Chapter 4, XST FPGA Optimization, explains how constraints can be used to optimize FPGA devices; explains macro generation; and describes the FPGA device primitive support.
- Chapter 5, XST CPLD Optimization, discusses CPLD synthesis options and the implementation details for macro generation.
- Chapter 6, XST Design Constraints, provides general information about XST design constraints, as well as information about specific constraints.
- Chapter 7, XST VHDL Language Support, explains how XST supports the VHSIC Hardware Description Language (VHDL), and provides details on VHDL supported constructs and synthesis options.
- Chapter 8, XST Verilog Language Support, describes XST support for Verilog constructs and meta comments.
- Chapter 9, XST Behavioral Verilog Language Support, describes XST support for Behavioral Verilog.
- Chapter 10, XST Mixed Language Support, describes how to run an XST project that mixes Verilog and VHDL designs.
- Chapter 11, XST Log File, describes the XST log file.
- Chapter 12, XST Naming Conventions, describes XST naming conventions.
- Chapter 13, XST Command Line Mode, describes how to run XST using the command line.

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

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**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Command Line Tools User 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	See the section Additional Resources for details. Refer to Title Formats in Chapter 1 for details. See Figure 2-5 in the Virtex®-6 Handbook.

Chapter 2

Introduction to Xilinx Synthesis Technology (XST)

This chapter provides general information about Xilinx Synthesis Technology (XST), and describes the changes to XST in this release. This chapter includes:

- About XST
- What's New in Release 11.1
- Setting XST Options

About XST

Xilinx Synthesis Technology (XST) is a Xilinx® application that synthesizes Hardware Description Language (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. The NGC file takes the place of both Electronic Data Interchange Format (EDIF) and Netlist Constraints File (NCF) files.

For more information about XST, see *Xilinx Synthesis Technology (XST) - Frequently Asked Questions (FAQ)*. Search for keyword *XST FAQ*.

What's New in Release 11.1

Following are the major changes to XST for Release 11.1:

- Added new value **`soft`** for the KEEP constraint. While this value instructs XST to preserve a designated net, the KEEP constraint is not propagated to the synthesized netlist.
- Support for the Synplicity **`syn_keep`** constraint has been better aligned with the way it is treated by Synplify. A **`syn_keep`** constraint still prevents XST from removing a designated signal. However, XST no longer propagates the constraint to the synthesized netlist , allowing placement and routing to optimize away this net if needed.
- XST now supports Partitions in place of Incremental Synthesis. Incremental Synthesis is no longer supported. The incremental_synthesis and resynthesize constraints are no longer supported. For more information on Partitions, see the ISE® Design Suite Help.

Setting XST Options

Before synthesizing your design, you can set a variety of options for XST. For more information on setting XST options, see:

- ISE® Design Suite Help
- XST Design Constraints
- XST Command Line Mode

Designs are usually made up of combinatorial logic and macros such as flip-flops, adders, subtractors, counters, FSMs, and RAMs. The macros greatly improve performance of the synthesized designs. It is important to use coding techniques to model the macros so they are optimally processed by XST.

XST first tries to recognize (infer) as many macros as possible. These macros are then passed to the Low Level Optimization step. In order to obtain better optimization results, the macros are either preserved as separate blocks, or merged with surrounded logic. 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. Synthesis constraints control the processing of inferred macros. For more information, see XST Design Constraints.

Chapter 3

XST Hardware Description Language (HDL) Coding Techniques

This chapter gives Hardware Description Language (HDL) coding examples for digital logic circuits. This chapter includes:

Most sections include:

- A general description of the macro
- A sample log file
- Constraints you can use to control the macro processing in XST
- VHDL and Verilog coding examples, including a schematic diagram and pin descriptions

For more information, see XST FPGA Optimization and XST CPLD Optimization.

For information on accessing the synthesis templates from ISE® Design Suite, see the ISE Design Suite Help.

Signed and Unsigned Support in XST

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

To enable support for signed and unsigned values in Verilog, enable Verilog-2001 as follows:

- In ISE® Design Suite, select **`Verilog 2001`** as instructed in the Synthesis Options topic of ISE Design Suite Help, or
- Set the **`-verilog2001`** command line option to **`yes`**.

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

Unsigned Adder

PACKAGE	TYPE
numeric_std	unsigned
std_logic_arith	unsigned
std_logic_unsigned	std_logic_vector

To create a signed adder, use the arithmetic packages and types that operate on signed values shown in the following table.

Signed Adder

PACKAGE	TYPE
numeric_std	signed
std_logic_arith	signed
std_logic_signed	std_logic_vector

For more information about available types, see the *IEEE VHDL Manual*.

Registers Hardware Description Language (HDL) Coding Techniques

XST recognizes flip-flops with the following control signals:

- Asynchronous Set/Reset
- Synchronous Set/Reset
- Clock Enable

For more information, see Specifying INIT and RLOC.

Registers Log File

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

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

Synthesizing Unit <registers_5>.
    Related source file is "registers_5.vhd".
    Found 4-bit register for signal <Q>.
    Summary:
        inferred   4 D-type flip-flop(s).
Unit <registers_5> synthesized.

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

Macro Statistics
# Registers                                             : 1
 4-bit register                                         : 1

=========================================================================
=========================================================================
*                 Advanced HDL Synthesis                                *
=========================================================================
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Registers                                             : 4
 Flip-Flops/Latches                                     : 4

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

With the introduction of new device families such as the Virtex®-4 device family, XST may optimize different slices of the same register in different ways. For example, XST may push a part of a register into a DSP48 block, while another part may be implemented on slices, or even become a part of a shift register. XST reports the total number of FF bits in the design in the HDL Synthesis Report after the Advanced HDL Synthesis step.

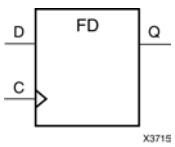
Registers Related Constraints

- Pack I/O Registers Into IOBs (IOB)
- Register Duplication (REGISTER_DUPLICATION)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- Register Balancing (REGISTER_BALANCING)

Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Flip-Flop With Positive-Edge Clock Diagram



Flip-Flop With Positive-Edge Clock Pin Descriptions

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

Flip-Flop With Positive Edge Clock VHDL Coding Example

```
--
-- Flip-Flop with Positive-Edge Clock
--
library ieee;
use ieee.std_logic_1164.all;

entity registers_1 is
    port(C, D : in std_logic;
         Q    : out std_logic);
end registers_1;

architecture archi of registers_1 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
```

Flip-Flop With Positive-Edge Clock Verilog Coding Example

```
//
// Flip-Flop with Positive-Edge Clock
//

module v_registers_1 (C, D, Q);
    input  C, D;
    output Q;
    reg    Q;

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

endmodule
```

Flip—Flop With Positive Edge Clock with INITSTATE of the Flop Set

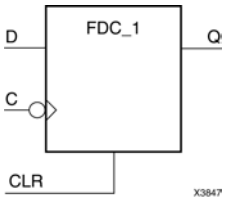
Verilog coding example:

```
module test(d, C, q);
    input d;
    input C;
    output q;

reg qtemp = 'b1 ;

always @ (posedge C)
begin
   qtemp = d;
end


 assign  q = qtemp;
endmodule
```

VHDL coding example:

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

entity registers_1 is
 port(C, D : in std_logic;
  Q : out std_logic);
end registers_1;

architecture archi of registers_1 is
signal qtemp : std_logic := '1';

begin

 process (C)

 begin
  if (C'event and C='1') then
   qtemp <= D;

  end if;
   Q <= Qtemp;
 end process;

end archi;
```

Flip-Flop With Negative-Edge Clock and Asynchronous Reset Diagram



Flip-Flop With Negative-Edge Clock and Asynchronous Reset Pin Descriptions

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

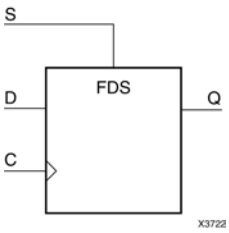
Flip-Flop With Negative-Edge Clock and Asynchronous Reset VHDL Coding Example

```
--
-- Flip-Flop with Negative-Edge Clock and Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;

entity registers_2 is
    port(C, D, CLR : in  std_logic;
         Q         : out std_logic);
end registers_2;

architecture archi of registers_2 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;
```

Flip-Flop With Negative-Edge Clock and Asynchronous Reset Verilog Coding Example

```
//
// Flip-Flop with Negative-Edge Clock and Asynchronous Reset
//

module v_registers_2 (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 Diagram



Flip-Flop With Positive-Edge Clock and Synchronous Set Pin Descriptions

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

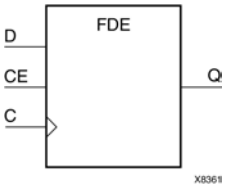
Flip-Flop With Positive-Edge Clock and Synchronous Set VHDL Coding Example

```
--
-- Flip-Flop with Positive-Edge Clock and Synchronous Set
--
library ieee;
use ieee.std_logic_1164.all;
entity registers_3 is
 port(C, D, S : in std_logic;
  Q : out std_logic);
end registers_3;

architecture archi of registers_3 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;
```

Flip-Flop With Positive-Edge Clock and Synchronous Set Verilog Coding Example

```
//
// Flip-Flop with Positive-Edge Clock and Synchronous Set
//
module v_registers_3 (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 Diagram



Flip-Flop With Positive-Edge Clock and Clock Enable Pin Descriptions

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

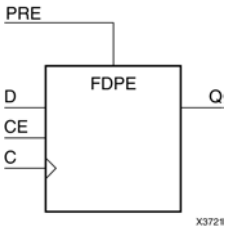
Flip-Flop With Positive-Edge Clock and Clock Enable VHDL Coding Example

```
--
-- Flip-Flop with Positive-Edge Clock and Clock Enable
--
library ieee;
use ieee.std_logic_1164.all;

entity registers_4 is
    port(C, D, CE : in std_logic;
         Q        : out std_logic);
end registers_4;

architecture archi of registers_4 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;
```

Flip-Flop With Positive-Edge Clock and Clock Enable Verilog Coding Example

```
//
// Flip-Flop with Positive-Edge Clock and Clock Enable
//

module v_registers_4 (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 Diagram



4-Bit Register With Positive-Edge Clock, Asynchronous Set, and Clock Enable Pin Descriptions

IO Pins	Description
D	Data Input
C	Positive-Edge Clock
PRE	Asynchronous Set (Active High)
CE	Clock Enable (Active High)
Q	Data Output

4-Bit Register With Positive-Edge Clock, Asynchronous Set, and Clock Enable VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity registers_5 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 registers_5;

architecture archi of registers_5 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;
```

4-Bit Register With Positive-Edge Clock, Asynchronous Set, and Clock Enable Verilog Coding Example

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

module v_registers_5 (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 Hardware Description Language (HDL) Coding Techniques

XST can recognize latches with asynchronous set/reset control signals. Latches can be described using:

- Process (VHDL)
- **`Always`** block (Verilog)
- Concurrent state assignment

XST does not support **`Wait`** statements (VHDL) for latch descriptions.

Latches 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
=========================================
...
```

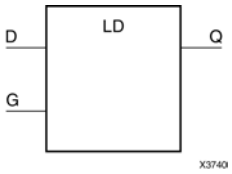
Latches Related Constraints

Pack I/O Registers Into IOBs (IOB)

Latches Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Latch With Positive Gate Diagram



Latch With Positive Gate Pin Descriptions

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

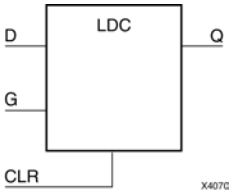
Latch With Positive Gate VHDL Coding Example

```
--
-- Latch with Positive Gate
--
library ieee;
use ieee.std_logic_1164.all;
entity latches_1 is
 port(G, D : in std_logic;
  Q : out std_logic);
end latches_1;

architecture archi of latches_1 is
begin
process (G, D)
begin
 if (G='1') then
  Q <= D;
  end if;
 end process;
end archi;
```

Latch With Positive Gate Verilog Coding Example

```
//
// Latch with Positive Gate
//
module v_latches_1 (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 Reset Diagram



Latch With Positive Gate and Asynchronous Reset Pin Descriptions

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

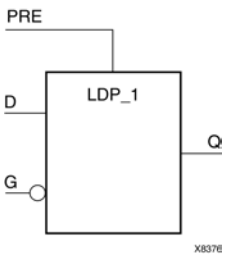
Latch With Positive Gate and Asynchronous Reset VHDL Coding Example

```
--
-- Latch with Positive Gate and Asynchronous Reset
--

library ieee;
use ieee.std_logic_1164.all;

entity latches_2 is
    port(G, D, CLR : in std_logic;
         Q : out std_logic);
end latches_2;

architecture archi of latches_2 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;
```

Latch With Positive Gate and Asynchronous Reset Verilog Coding Example

```
//
// Latch with Positive Gate and Asynchronous Reset
//

module v_latches_2 (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 Set Diagram



4-Bit Latch With Inverted Gate and Asynchronous Set Pin Descriptions

IO Pins	Description
D	Data Input
G	Inverted Gate
PRE	Asynchronous Preset (Active High)
Q	Data Output

4-Bit Latch With Inverted Gate and Asynchronous Set VHDL Coding Example

```
--
-- 4-bit Latch with Inverted Gate and Asynchronous Set
--

library ieee;
use ieee.std_logic_1164.all;

entity latches_3 is
    port(D      : in std_logic_vector(3 downto 0);
         G, PRE : in std_logic;
         Q      : out std_logic_vector(3 downto 0));
end latches_3;

architecture archi of latches_3 is
begin
    process (PRE, G, D)
    begin
        if (PRE='1') then
            Q <= "1111";
        elsif (G='0') then
            Q <= D;
        end if;
    end process;
end archi;
```

4-Bit Latch With Inverted Gate and Asynchronous Set Verilog Coding Example

```
//
// 4-bit Latch with Inverted Gate and Asynchronous Set
//

module v_latches_3 (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 Hardware Description Language (HDL) Coding Techniques

Tristate elements can be described using:

- Combinatorial process (VHDL)
- Always block (Verilog)
- Concurrent assignment

In the Tristates Coding Examples, 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.

Tristates Log File

The XST log file 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
=============================
...
```

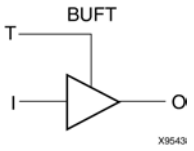
Tristates Related Constraints

Convert Tristates to Logic (TRISTATE2LOGIC)

Tristates Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Tristate Description Using Combinatorial Process and Always Block Diagram



Tristate Description Using Combinatorial Process and Always Block Pin Descriptions

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

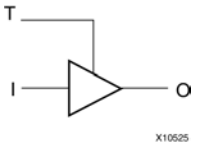
Tristate Description Using Combinatorial Process VHDL Coding Example

```
--
-- Tristate Description Using Combinatorial Process
--

library ieee;
use ieee.std_logic_1164.all;

entity three_st_1 is
    port(T : in  std_logic;
         I : in  std_logic;
         O : out std_logic);
end three_st_1;

architecture archi of three_st_1 is
begin

    process (I, T)
    begin
        if (T='0') then
            O <= I;
        else
            O <= 'Z';
        end if;
    end process;

end archi;
```

Tristate Description Using Combinatorial Always Block Verilog Coding Example

```
//
// Tristate Description Using Combinatorial Always Block
//

module v_three_st_1 (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
```

Tristate Description Using Concurrent Assignment Diagram



Tristate Description Using Concurrent Assignment Pin Descriptions

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

Tristate Description Using Concurrent Assignment VHDL Coding Example

```
--
-- Tristate Description Using Concurrent Assignment
--

library ieee;
use ieee.std_logic_1164.all;

entity three_st_2 is
    port(T : in  std_logic;
         I : in  std_logic;
         O : out std_logic);
end three_st_2;

architecture archi of three_st_2 is
begin
    O <= I when (T='0') else 'Z';
end archi;
```

Tristate Description Using Concurrent Assignment Verilog Coding Example

```
//
// Tristate Description Using Concurrent Assignment
//

module v_three_st_2 (T, I, O);
    input  T, I;
    output O;

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

endmodule
```

Counters Hardware Description Language (HDL) Coding Techniques

XST recognizes counters with the following control signals:

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

Hardware Description Language (HDL) coding styles for the following control signals are equivalent to those described in Registers Hardware Description Language (HDL) Coding Techniques.

- Clock
- Asynchronous Set/Reset
- Synchronous Set/Reset

XST supports both unsigned and signed counters.

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

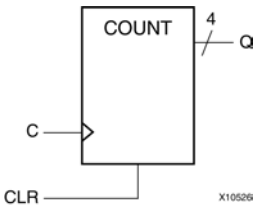
Counters Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Counters Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

4-Bit Unsigned Up Counter With Asynchronous Reset Diagram



4-Bit Unsigned Up Counter With Asynchronous Reset Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
Q	Data Output

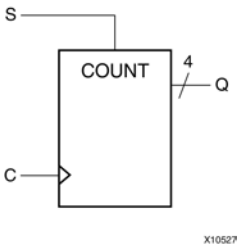
4-Bit Unsigned Up Counter With Asynchronous Reset VHDL Coding Example

```
--
-- 4-bit unsigned up counter with an asynchronous reset.
--

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

entity counters_1 is
    port(C, CLR : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_1;

architecture archi of counters_1 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;
```

4-Bit Unsigned Up Counter With Asynchronous Reset Verilog Coding Example

```
//
// 4-bit unsigned up counter with an asynchronous reset.
//

module v_counters_1 (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 Diagram



4-Bit Unsigned Down Counter With Synchronous Set Pin Descriptions

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

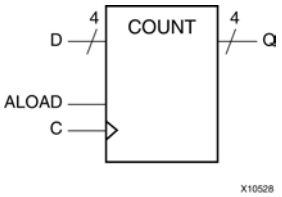
4-Bit Unsigned Down Counter With Synchronous Set VHDL Coding Example

```
--
-- 4-bit unsigned down counter with a synchronous set.
--

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

entity counters_2 is
    port(C, S : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_2;

architecture archi of counters_2 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;
```

4-Bit Unsigned Down Counter With Synchronous Set Verilog Coding Example

```
//
// 4-bit unsigned down counter with a synchronous set.
//

module v_counters_2 (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 Diagram



4-Bit Unsigned Up Counter With Asynchronous Load From Primary Input Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
ALOAD	Asynchronous Load (Active High)
D	Data Input
Q	Data Output

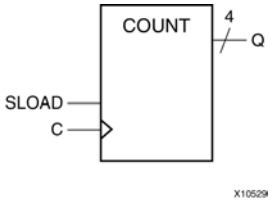
4-Bit Unsigned Up Counter With Asynchronous Load From Primary Input VHDL Coding Example

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

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

entity counters_3 is
    port(C, ALOAD : in std_logic;
         D : in std_logic_vector(3 downto 0);
         Q : out std_logic_vector(3 downto 0));
end counters_3;

architecture archi of counters_3 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;
```

4-Bit Unsigned Up Counter With Asynchronous Load From Primary Input Verilog Coding Example

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

module v_counters_3 (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 Constant Diagram



4-Bit Unsigned Up Counter With Synchronous Load With Constant Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
SLOAD	Synchronous Load (Active High)
Q	Data Output

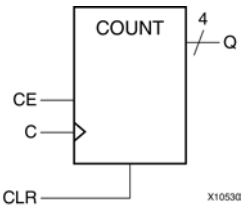
4-Bit Unsigned Up Counter With Synchronous Load With Constant VHDL Coding Example

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

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

entity counters_4 is
    port(C, SLOAD : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_4;

architecture archi of counters_4 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;
```

4-Bit Unsigned Up Counter With Synchronous Load With Constant Verilog Coding Example

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

module v_counters_4 (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 Reset and Clock Enable Diagram



4-Bit Unsigned Up Counter With Asynchronous Reset and Clock Enable Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
CE	Clock Enable
Q	Data Output

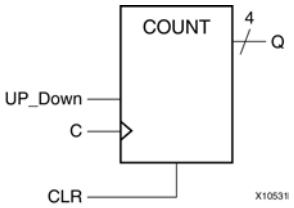
4-Bit Unsigned Up Counter With Asynchronous Reset and Clock Enable VHDL Coding Example

```
--
-- 4-bit Unsigned Up Counter with Asynchronous Reset and Clock Enable
--

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

entity counters_5 is
    port(C, CLR, CE : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_5;

architecture archi of counters_5 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;
```

4-Bit Unsigned Up Counter With Asynchronous Reset and Clock Enable Verilog Coding Example

```
//
// 4-bit Unsigned Up Counter with Asynchronous Reset and Clock Enable
//

module v_counters_5 (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 Reset Diagram



4-Bit Unsigned Up/Down Counter With Asynchronous Reset Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
UP_DOWN	Up/Down Count Mode Selector
Q	Data Output

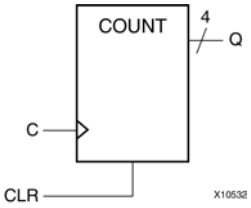
4-Bit Unsigned Up/Down Counter With Asynchronous Reset VHDL Coding Example

```
--
-- 4-bit Unsigned Up/Down counter with Asynchronous Reset
--

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

entity counters_6 is
    port(C, CLR, UP_DOWN : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_6;

architecture archi of counters_6 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;
```

4-Bit Unsigned Up/Down Counter With Asynchronous Reset Verilog Coding Example

```
//
// 4-bit Unsigned Up/Down counter with Asynchronous Reset
//

module v_counters_6 (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 Diagram



4-Bit Signed Up Counter With Asynchronous Reset Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
Q	Data Output

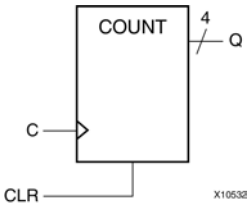
4-Bit Signed Up Counter With Asynchronous Reset VHDL Coding Example

```
--
-- 4-bit Signed Up Counter with Asynchronous Reset
--

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

entity counters_7 is
    port(C, CLR : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_7;

architecture archi of counters_7 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;
```

4-Bit Signed Up Counter With Asynchronous Reset Verilog Coding Example

```
//
// 4-bit Signed Up Counter with Asynchronous Reset
//

module v_counters_7 (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 Diagram



4-Bit Signed Up Counter With Asynchronous Reset and Modulo Maximum Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
Q	Data Output

4-Bit Signed Up Counter With Asynchronous Reset and Modulo Maximum VHDL Coding Example

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

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

entity counters_8 is
    generic (MAX : integer := 16);
    port(C, CLR : in std_logic;
         Q : out integer range 0 to MAX-1);
end counters_8;

architecture archi of counters_8 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;
```

4-Bit Signed Up Counter With Asynchronous Reset and Modulo Maximum Verilog Coding Example

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

module v_counters_8 (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
```

Accumulators Hardware Description Language (HDL) Coding Techniques

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. For more information, see Counters Hardware Description Language (HDL) Coding Techniques.

Accumulators in Virtex-4 Devices and Virtex-5 Devices

Virtex®-4 devices and Virtex-5 devices enable accumulators to be implemented on DSP48 resources. XST can push up to two 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 implements the entire macro using slice logic.

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

In **auto** mode, to control DSP48 resources for the synthesis use the DSP Utilization Ratio (DSP_UTILIZATION_RATIO) constraint. By default, XST tries to utilize all DSP48 resources. For more information, see DSP48 Block Resources.

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. To shape a macro in a specific way, use the Keep (KEEP) constraint. For example, to exclude the first register stage from the DSP48, place Keep (KEEP) constraints on the outputs of these registers.

As with other families, for Virtex-4 devices and Virtex-5 devices, XST reports the details of inferred accumulators at the HDL Synthesis step. Because accumulators are implemented within the MAC implementation mechanism, they are no longer visible in the Final Synthesis Report.

Accumulators 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
==============================
...
```

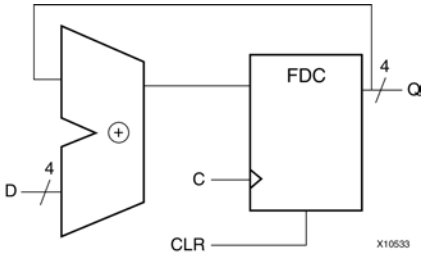
Accumulators Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Accumulators Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

4-Bit Unsigned Up Accumulator With Asynchronous Reset Diagram



4-Bit Unsigned Up Accumulator With Asynchronous Reset Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
CLR	Asynchronous Reset (Active High)
D	Data Input
Q	Data Output

4-Bit Unsigned Up Accumulator With Asynchronous Reset VHDL Coding Example

```
--
-- 4-bit Unsigned Up Accumulator with Asynchronous Reset
--

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

entity accumulators_1 is
    port(C, CLR : in std_logic;
         D : in std_logic_vector(3 downto 0);
         Q : out std_logic_vector(3 downto 0));
end accumulators_1;

architecture archi of accumulators_1 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;
```

4-Bit Unsigned Up Accumulator With Asynchronous Reset Verilog Coding Example

```
//
// 4-bit Unsigned Up Accumulator with Asynchronous Reset
//

module v_accumulators_1 (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
```

Shift Registers Hardware Description Language (HDL) Coding Techniques

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: for example, left, right

Describing Shift Registers

Ways to describe shift registers in VHDL include:

- 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)

For more information, see your VHDL and Verilog language reference manuals.

Implementing Shift Registers

Hardware Resources to Implement Shift Registers

Devices	SRL16	SRL16E	SRLC16	SRLC16E	SRLC32E
Spartan®-3 Spartan-3E Spartan-3A	Yes	Yes	Yes	Yes	No
Virtex®-4	Yes	Yes	Yes	Yes	No
Virtex-5	Yes	Yes	Yes	Yes	Yes

SRL16 and SRLC16

Both SRL16 and SRLC16 are available with or without a clock enable.

Synchronous and asynchronous control signals are not available in the SLRC16x primitives. However, XST takes advantage of dedicated SRL resources if a shift register description has only a single asynchronous or synchronous set or reset signal. Such implementation reduces area significantly.

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

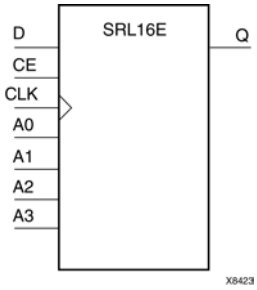
- clock
- clock enable
- serial data in
- serial data out

If your shift register *does have*, for instance, a synchronous parallel load, or multiple set or reset signals, no SRL16 is implemented. XST uses specific internal processing which enables it to produce the best final results.

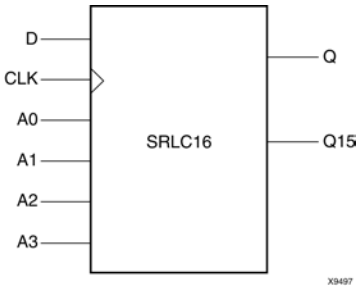
The XST log file reports recognized shift registers when they can be implemented using SRL16 or SRLC16 resources. As a result, some of the coding examples shown below (in particular those with parallel load or parallel out functionality) do not result in any specific shift register reporting.

For more information, see Specifying INIT and RLOC.

Pin Layout of SRL16E Diagram



Pin Layout of SRLC16 Diagram



Shift Registers Log File

XST recognizes shift registers in the Low Level Optimization step. The XST log file reports the size of recognized shift registers.

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

Synthesizing Unit <shift_registers_1>.
    Related source file is "shift_registers_1.vhd".
    Found 8-bit register for signal <tmp>.
    Summary:
        inferred   8 D-type flip-flop(s).
Unit <shift_registers_1> synthesized.

=========================================
*         Advanced HDL Synthesis        *
=========================================
Advanced HDL Synthesis Report
Macro Statistics
# Registers : 8
 Flip-Flops : 8
=========================================
=========================================
*         Low Level Synthesis           *
=========================================
Processing Unit <shift_registers_1> :
 Found 8-bit shift register for signal <tmp_7>.
Unit <shift_registers_1> processed.
=========================================
Final Register Report
Macro Statistics
# Shift Registers : 1
 8-bit shift register : 1
=========================================
```

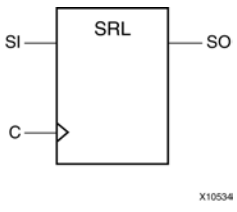
Shift Registers Related Constraints

Shift Register Extraction (SHREG_EXTRACT)

Shift Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Serial Out Diagram



8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Serial Out Pin Descriptions

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

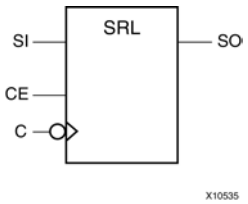
8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_1 is
    port(C, SI : in std_logic;
         SO : out std_logic);
end shift_registers_1;

architecture archi of shift_registers_1 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_1 (C, SI, SO);
    input C,SI;
    output SO;
    reg [7:0] tmp;

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

    assign SO = tmp[7];

endmodule
```

8-Bit Shift-Left Register With Negative-Edge Clock, Clock Enable, Serial In and Serial Out Diagram



8-Bit Shift-Left Register With Negative-Edge Clock, Clock Enable, Serial In and Serial Out Pin Descriptions

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

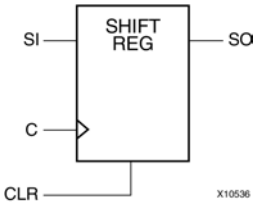
8-Bit Shift-Left Register With Negative-Edge Clock, Clock Enable, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_2 is
    port(C, SI, CE : in std_logic;
         SO : out std_logic);
end shift_registers_2;

architecture archi of shift_registers_2 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;
```

8-Bit Shift-Left Register With Negative-Edge Clock, Clock Enable, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_2 (C, CE, SI, SO);
    input C,SI, CE;
    output SO;
    reg [7:0] tmp;

    always @(negedge C)
    begin
        if (CE)
        begin
            tmp = {tmp[6:0], SI};
        end
    end

    assign SO = tmp[7];

endmodule
```

8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Reset, Serial In and Serial Out Diagram



8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Reset, Serial In and Serial Out Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
CLR	Asynchronous Reset (Active High)

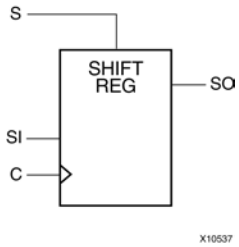
8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Reset, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_3 is
    port(C, SI, CLR : in std_logic;
         SO : out std_logic);
end shift_registers_3;

architecture archi of shift_registers_3 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Reset, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_3 (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
            tmp <= {tmp[6:0], SI};
    end

    assign SO = tmp[7];

endmodule
```

8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Set, Serial In and Serial Out Diagram



8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Set, Serial In and Serial Out Pin Descriptions

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

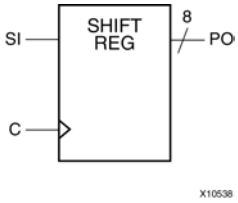
8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Set, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_4 is
    port(C, SI, S : in std_logic;
         SO : out std_logic);
end shift_registers_4;

architecture archi of shift_registers_4 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Set, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_4 (C, S, SI, SO);
    input C,SI,S;
    output SO;
    reg [7:0] tmp;

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

    assign SO = tmp[7];

endmodule
```

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



8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Parallel Out Pin Descriptions

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

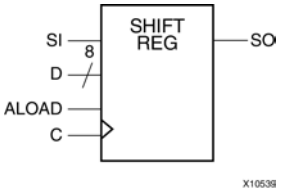
8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Parallel Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_5 is
    port(C, SI : in std_logic;
         PO : out std_logic_vector(7 downto 0));
end shift_registers_5;

architecture archi of shift_registers_5 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Serial In and Parallel Out Verilog Coding Example

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

module v_shift_registers_5 (C, SI, PO);
    input C,SI;
    output [7:0] PO;
    reg [7:0] tmp;

    always @(posedge C)

      tmp <= {tmp[6:0], SI};

    assign PO = tmp;

endmodule
```

8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out Diagram



8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
ALOAD	Asynchronous Parallel Load (Active High)
D	Data Input
SO	Serial Output

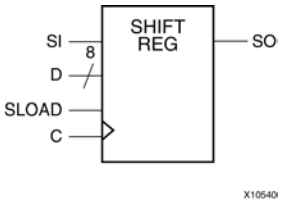
8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_6 is
    port(C, SI, ALOAD : in std_logic;
         D : in std_logic_vector(7 downto 0);
         SO : out std_logic);
end shift_registers_6;

architecture archi of shift_registers_6 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Asynchronous Parallel Load, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_6 (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 Diagram



8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
SLOAD	Synchronous Parallel Load (Active High)
D	Data Input
SO	Serial Output

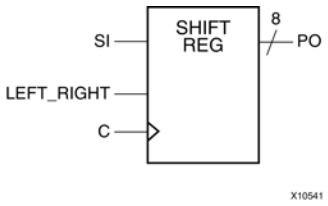
8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_7 is
    port(C, SI, SLOAD : in std_logic;
         D : in std_logic_vector(7 downto 0);
         SO : out std_logic);
end shift_registers_7;

architecture archi of shift_registers_7 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;
```

8-Bit Shift-Left Register With Positive-Edge Clock, Synchronous Parallel Load, Serial In and Serial Out Verilog Coding Example

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

module v_shift_registers_7 (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 Diagram



8-Bit Shift-Left/Shift-Right Register With Positive-Edge Clock, Serial In and Parallel Out Pin Descriptions

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

8-Bit Shift-Left/Shift-Right Register With Positive-Edge Clock, Serial In and Parallel Out VHDL Coding Example

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

library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_8 is
    port(C, SI, LEFT_RIGHT : in std_logic;
         PO : out std_logic_vector(7 downto 0));
end shift_registers_8;

architecture archi of shift_registers_8 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;
```

8-Bit Shift-Left/Shift-Right Register With Positive-Edge Clock, Serial In and Parallel Out Verilog Coding Example

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

module v_shift_registers_8 (C, SI, LEFT_RIGHT, PO);
    input C,SI,LEFT_RIGHT;
    output [7:0] 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 Registers Hardware Description Language (HDL) Coding Techniques

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 the primitives shown in the following table.

Devices	SRL16	SRL16E	SRLC16	SRLC16E	SRLC32E
Spartan®-3 Spartan-3E Spartan-3A	Yes	Yes	Yes	Yes	No
Virtex®-4	Yes	Yes	Yes	Yes	No
Virtex-5	Yes	Yes	Yes	Yes	Yes

Dynamic Shift Registers 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_shift_registers_1>.
    Related source file is "dynamic_shift_registers_1.vhd".
    Found 1-bit 16-to-1 multiplexer for signal <Q>.
    Found 16-bit register for signal <SRL_SIG>.
    Summary:
        inferred  16 D-type flip-flop(s).
        inferred   1 Multiplexer(s).
Unit <dynamic_shift_registers_1> synthesized.

=========================================
*         Advanced HDL Synthesis        *
=========================================
...
Synthesizing (advanced) Unit <dynamic_shift_registers_1>.
        Found 16-bit dynamic shift register for signal <Q>.
```

```
Unit <dynamic_shift_registers_1> synthesized (advanced).

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

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

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

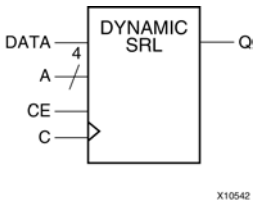
Dynamic Shift Registers Related Constraints

Shift Register Extraction (SHREG_EXTRACT)

Dynamic Shift Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

16-Bit Dynamic Shift Register With Positive-Edge Clock, Serial In and Serial Out



The following table shows pin descriptions for a dynamic register. The register can:

- Be either serial or parallel
- Be left or right
- Have a synchronous or asynchronous reset
- Have a depth up to 16 bits.

16-Bit Dynamic Shift Register With Positive-Edge Clock, Serial In and Serial Out Pin Descriptions

IO Pins	Description
C	Positive-Edge Clock
SI	Serial In
AClr	Asynchronous Reset
SClr	Synchronous Reset
SLoad	Synchronous Parallel Load
Data	Parallel Data Input Port
ClkEn	Clock Enable
LeftRight	Direction selection
SerialInRight	Serial Input Right for Bidirectional Shift Register
PSO	Serial or Parallel Output

16-Bit Dynamic Shift Register With Positive-Edge Clock, Serial In and Serial Out VHDL Coding Example

```
--
-- 16-bit dynamic shift register.
--

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

entity dynamic_shift_registers_1 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 dynamic_shift_registers_1;

architecture rtl of dynamic_shift_registers_1 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;
```

16-Bit Dynamic Shift Register With Positive-Edge Clock, Serial In and Serial Out Verilog Coding Example

```
//
// 16-bit dynamic shift register.
//

module v_dynamic_shift_registers_1 (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 Hardware Description Language (HDL) Coding Techniques

XST supports different description styles for multiplexers (MUXs), such as **If-Then-Else** or **Case**.

If you describe a MUX using a **Case** statement, and you do not specify all values of the selector, the result may be latches instead of a multiplexer. When writing MUXs, you can use **don't care** to describe selector values.

XST decides whether to infer the MUXs during the Macro Inference step. If the MUX has several inputs that are the same, XST can decide not to infer it. You can use the MUX_EXTRACT constraint to force XST to infer the MUX.

Verilog **Case** statements can be:

- **full** or **not full**
- **parallel** or **not parallel**

A Verilog **Case** statement is:

- **full** if all possible branches are specified
- **parallel** if it does not contain branches that can be executed simultaneously

Multiplexers Full and Parallel Case Statement Coding Example

```
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
```

Multiplexers Not Full But Parallel Case Statement Coding Example

```
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
```

Multiplexers Neither Full Nor Parallel Case Statement Coding Example

```
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.

Multiplexers Verilog Case Implementation Style Parameter

This characterization of the Case statements can be guided or modified by using the Case Implementation Style parameter. For more information, see XST Design Constraints. Accepted values for this parameter are **none**, **`full`**, **`parallel`**, and **`full-parallel`**.

- If **none** (default) is used, 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.

Verilog Case Statement Resources indicates the *resources* used to synthesize the Multiplexers Case Statement Examples 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.

Multiplexers Verilog Case Statement Resources

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

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

Multiplexers 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
============================
...
```

Explicit inference and reporting of multiplexers may vary depending on the targeted device families. The following coding examples are limited to 4-to-1 multiplexers. They are reported as shown above only if the target is a LUT4-based device family. For Virtex®-5 devices, multiplexers are explicitly inferred only for sizes of 8-to-1 and above.

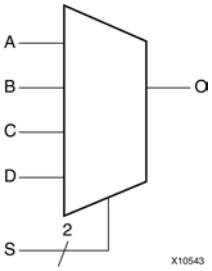
Multiplexers Related Constraints

- Mux Extraction (MUX_EXTRACT)
- Mux Style (MUX_STYLE)
- Enumerated Encoding (ENUM_ENCODING)

Multiplexers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

4-to-1 1-Bit MUX Using IF Statement Diagram



4-to-1 1-Bit MUX Using IF Statement Pin Descriptions

IO Pins	Description
a, b, c, d	Data Inputs
s	MUX Selector
o	Data Output

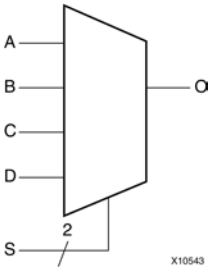
4-to-1 1-Bit MUX Using IF Statement VHDL Coding Example

```
--
-- 4-to-1 1-bit MUX using an If statement.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_1 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_1;

architecture archi of multiplexers_1 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;
```

4-to-1 1-Bit MUX Using IF Statement Verilog Coding Example

```
//
// 4-to-1 1-bit MUX using an If statement.
//

module v_multiplexers_1 (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 1-Bit MUX Using Case Statement Diagram



4-to-1 1-Bit MUX Using Case Statement Pin Descriptions

IO Pins	Description
a, b, c, d	Data Inputs
s	MUX Selector
o	Data Output

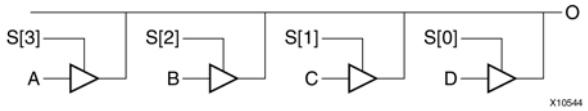
4-to-1 1-Bit MUX Using Case Statement VHDL Coding Example

```
--
-- 4-to-1 1-bit MUX using a Case statement.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_2 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_2;

architecture archi of multiplexers_2 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;
```

4-to-1 1-Bit MUX Using Case Statement Verilog Coding Example

```
//
// 4-to-1 1-bit MUX using a Case statement.
//

module v_multiplexers_2 (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 1-Bit MUX Using Tristate Buffers Diagram



4-to-1 1-Bit MUX Using Tristate Buffers Pin Descriptions

IO Pins	Description
a, b, c, d	Data Inputs
s	MUX Selector
o	Data Output

4-to-1 1-Bit MUX Using Tristate Buffers VHDL Coding Example

```
--
-- 4-to-1 1-bit MUX using tristate buffers.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_3 is
    port (a, b, c, d : in std_logic;
          s : in std_logic_vector (3 downto 0);
          o : out std_logic);
end multiplexers_3;

architecture archi of multiplexers_3 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;
```

4-to-1 1-Bit MUX Using Tristate Buffers Verilog Coding Example

```
//
// 4-to-1 1-bit MUX using tristate buffers.
//

module v_multiplexers_3 (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
```

The following coding examples illustrate how XST infers a latch when no **else** statement is described at the end of an **if/elsif** construct. Since the **else** statement is missing, XST assumes that, for the **s=11** case, **o** retains its old value, and that a memory element is needed. XST issues the following warning message.

```
WARNING:Xst:737 - Found 1-bit latch for signal <o1>.
INFO:Xst - HDL ADVISOR - Logic functions respectively driving the data
and gate enable inputs of this latch share common terms. This situation
will potentially lead to setup/hold violations and, as a result, to
simulation problems. This situation may come from an incomplete case statement
(all selector values are not covered). You should carefully review if it was
in your intentions to describe such a latch.
```

Unless you actually intended to describe such a latch, add the missing **else** statement.

Caution! Leaving out an **else** statement may result in errors during simulation.

VHDL Coding Example of a Missing Else Statement Leading to a Latch Inference

```
--
-- 3-to-1 1-bit MUX with a 1-bit latch.
--

library ieee;
use ieee.std_logic_1164.all;

entity multiplexers_4 is
    port (a, b, c: in std_logic;
          s : in std_logic_vector (1 downto 0);
          o : out std_logic);
end multiplexers_4;

architecture archi of multiplexers_4 is
begin
    process (a, b, c, 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 Coding Example of a Missing Else Statement Leading to a Latch Inference

```
//
// 3-to-1 1-bit MUX with a 1-bit latch.
//
 module v_multiplexers_4 (a, b, c, s, o);
    input a,b,c;
    input [1:0] s;
    output o;
    reg o;

    always @(a or b or c 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 Hardware Description Language (HDL) Coding Techniques

A decoder is a multiplexer whose inputs are all constant with distinct one-hot (or one-cold) coded values. For more information, see Multiplexers HDL Coding Techniques.

Decoders 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
==============================
...
```

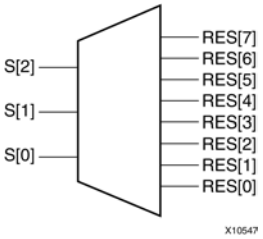
Decoders Related Constraints

Decoder Extraction (DECODER_EXTRACT)

Decoders Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

1-of-8 Decoder (One-Hot) Diagram



1-of-8 Decoders (One-Hot) Pin Descriptions

IO Pins	Description
s	Selector
res	Data Output

1-of-8 Decoder (One-Hot) VHDL Coding Example

```
--
-- 1-of-8 decoder (One-Hot)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_1 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
    end decoders_1;

architecture archi of decoders_1 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;
```

1-of-8 decoder (One-Hot) Verilog Coding Example

```
//
// 1-of-8 decoder (One-Hot)
//

module v_decoders_1 (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
```

1-of-8 Decoder (One-Cold) Pin Descriptions

IO Pins	Description
s	Selector
res	Data Output

1-of-8 decoder (One-Cold) VHDL Coding Example

```
--
-- 1-of-8 decoder (One-Cold)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_2 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_2;

architecture archi of decoders_2 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;
```

1-of-8 Decoder (One-Cold) Verilog Coding Example

```
//
// 1-of-8 decoder (One-Cold)
//

module v_decoders_2 (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
```

Decoder With Unselected Outputs Pin Descriptions

IO Pins	Description
s	Selector
res	Data Output

No Decoder Inference (Unused Decoder Output) VHDL Coding Example

```
--
-- No Decoder Inference (unused decoder output)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_3 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_3;

architecture archi of decoders_3 is
begin
    res <= "00000001" when sel = "000" else
           -- unused decoder output
           "XXXXXXXX" 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;
```

No Decoder Inference (Unused Decoder Output) Verilog Coding Example

```
//
// No Decoder Inference (unused decoder output)
//

module v_decoders_3 (sel, res);
    input [2:0] sel;
    output [7:0] res;
    reg [7:0] res;

    always @(sel)
    begin
        case (sel)
            3'b000 : res = 8'b00000001;
            // unused decoder output
            3'b001 : res = 8'bxxxxxxxx;
            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
```

No Decoder Inference (Some Selector Values Unused) VHDL Coding Example

```
--
-- No Decoder Inference (some selector values are unused)
--

library ieee;
use ieee.std_logic_1164.all;

entity decoders_4 is
    port (sel: in std_logic_vector (2 downto 0);
          res: out std_logic_vector (7 downto 0));
end decoders_4;

architecture archi of decoders_4 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;
```

No Decoder Inference (Some Selector Values Unused) Verilog Coding Example

```
//
// No Decoder Inference (some selector values are unused)
//

module v_decoders_4 (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 Hardware Description Language (HDL) Coding Techniques

XST can recognize a priority encoder, but in most cases XST does not infer it. To force priority encoder inference, use Priority Encoder Extraction (PRIORITY_EXTRACT) with the value *force*.

Xilinx® recommends that you use Priority Encoder Extraction (PRIORITY_EXTRACT) on a signal-by-signal basis. Otherwise, Priority Encoder Extraction (PRIORITY_EXTRACT) may give sub-optimal results.

Priority Encoders 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
==============================
...
```

Priority Encoders Related Constraints

Priority Encoder Extraction (PRIORITY_EXTRACT)

Priority Encoders Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

3-Bit 1-of-9 Priority Encoder Coding Examples

For this example XST may infer a priority encoder. Use Priority Encoder Extraction (PRIORITY_EXTRACT) with a value force to force its inference.

3-Bit 1-of-9 Priority Encoder Pin Descriptions

IO Pins	Description
sel	Selector
code	Encoded Output Bus

3-Bit 1-of-9 Priority Encoder VHDL Coding Example

```
--
-- 3-Bit 1-of-9 Priority Encoder
--

library ieee;
use ieee.std_logic_1164.all;

entity priority_encoder_1 is
    port ( sel : in std_logic_vector (7 downto 0);
           code :out std_logic_vector (2 downto 0));

    attribute priority_extract: string;
    attribute priority_extract of priority_encoder_1: entity is "force";
end priority_encoder_1;

architecture archi of priority_encoder_1 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;
```

3-Bit 1-of-9 Priority Encoder Verilog Coding Example

```
//
// 3-Bit 1-of-9 Priority Encoder
//

(* priority_extract="force" *)
module v_priority_encoder_1 (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 Hardware Description Language (HDL) Coding Techniques

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.

All of these I/Os are mandatory. Otherwise, XST does *not* infer a logical shifter.

When writing your Hardware Description Language (HDL) code:

- Use only logical, arithmetic, and rotate shift operators. Shift operations that fill vacated positions with values from another signal are not recognized.
- For VHDL, you can use predefined shift (for example, SLL, SRL, ROL) or concatenation operations only. For more information on predefined shift operations, see the IEEE VHDL reference manual.
- 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 positive only.
- All values of the selector must be presented.

Logical Shifters 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
=============================
...
```

Logical Shifters Related Constraints

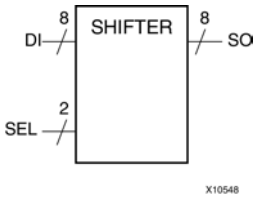
Logical Shifter Extraction (SHIFT_EXTRACT)

Logical Shifters Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

The minimal size for XST to explicitly infer logical shifter macros may vary depending on the targeted device family. The following coding examples have been validated on LUT4-based device families such as Virtex®-4 devices. For Virtex-5 devices, logical shifters are explicitly inferred only when the selector size is at least 3.

Logical Shifter One Diagram



Logical Shifter One Pin Descriptions

IO Pins	Description
DI	Data Input
SEL	Shift Distance Selector
SO	Data Output

Logical Shifter One VHDL Coding Example

```
--
-- Following is the VHDL code for a logical shifter.
--

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

entity logical_shifters_1 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_1;

architecture archi of logical_shifters_1 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;
```

Logical Shifter One Verilog Coding Example

```
//
// Following is the Verilog code for a logical shifter.
//

module v_logical_shifters_1 (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
```

Logical Shifter Two Pin Descriptions

IO Pins	Description
DI	Data Input
SEL	Shift Distance Selector
SO	Data Output

Logical Shifter Two VHDL Coding Example

XST does *not* infer a logical shifter for Logical Shifter Two, since not all selector values are presented.

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

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

entity logical_shifters_2 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_2;

architecture archi of logical_shifters_2 is
begin
    with SEL select
        SO <= DI when "00",
        DI sll 1 when "01",
        DI sll 2 when others;
end archi;
```

Logical Shifter Two Verilog Coding Example

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

module v_logical_shifters_2 (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
```

Logical Shifter Three Pin Descriptions

IO Pins	Description
DI	Data Input
SEL	Shift Distance Selector
SO	Data Output

Logical Shifter Three VHDL Coding Example

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.

```
--
-- 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.
--

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

entity logical_shifters_3 is
    port(DI : in unsigned(7 downto 0);
         SEL : in unsigned(1 downto 0);
         SO : out unsigned(7 downto 0));
end logical_shifters_3;

architecture archi of logical_shifters_3 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;
```

Logical Shifter Three Verilog Coding Example

```
//
// 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.
//

module v_logical_shifters_3 (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 Operators Hardware Description Language (HDL) Coding Techniques

XST supports the following arithmetic operators:

- Adders with:
  - Carry In
  - Carry Out
  - Carry In/Out
- Subtractors
- Adders/Subtractors
- Comparators:
  - =
  - /=
  - <
  - <=
  - >
  - >=
- Multipliers
- Dividers

XST supports the following for signed and unsigned operators:

- Adders
- Subtractors
- Comparators
- Multipliers

For more information on signed and unsigned operators support in VHDL, see Registers Hardware Description Language (HDL) Coding Techniques.

XST performs resource sharing for:

- Adders
- Subtractors
- Adders/subtractors
- Multipliers

Arithmetic Operators 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
=========================
```

Arithmetic Operators Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Adders, Subtractors, and Adders/Subtractors Hardware Description Language (HDL) Coding Techniques

The following device families allow adders and subtractors to be implemented on DSP48 resources:

- Virtex®-4
- Virtex-5
- Spartan®-3A DSP

XST supports the one 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 adder/subtractor macro does not fit in a single DSP48, XST implements the entire macro using slice logic.

Macro implementation on DSP48 blocks is controlled by DSP Utilization Ratio (DSP_UTILIZATION_RATIO) with a default value of **auto**. In **auto** mode, if an adder/subtractor is a part of a more complex macro such as a filter, XST automatically places it on the DSP block. Otherwise, XST implements adders/subtractors using LUTs. Set the value of Use DSP48 (USE_DSP48) to **yes** in order to force XST to push these macros into a DSP48. When placing an Adder/Subtractor on a DSP block, XST checks to see if it is connected to other DSP chains. If so, XST tries to take advantage of fast DSP connections, and connects this adder/subtractor to the DSP chain using these fast connections.

When implementing adders/subtractors on DSP48 blocks, XST performs automatic DSP48 resource control.

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. Use the Keep (KEEP) constraint to shape a macro in a specific way. For example, to exclude the first register stage from the DSP48, place Keep (KEEP) constraints on the outputs of these registers.

Adders, Subtractors, and Adders/Subtractors Log File

```
Synthesizing Unit <v_adders_4>.
    Related source file is "v_adders_4.v".
    Found 8-bit adder carry in/out for signal <$addsub0000>.
    Summary:
 inferred   1 Adder/Subtractor(s).
Unit <v_adders_4> synthesized.

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

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

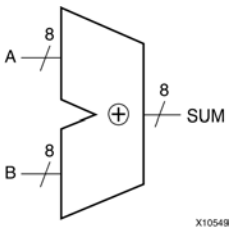
Adders, Subtractors, and Adders/Subtractors Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Adders, Subtractors, and Adders/Subtractors Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Unsigned 8-Bit Adder Diagram



Unsigned 8-Bit Adder Pin Descriptions IO Pins

IO Pins	Description
A, B	Add Operands
SUM	Add Result

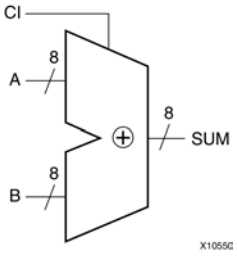
Unsigned 8-Bit Adder VHDL Coding Example

```
--
-- Unsigned 8-bit Adder
--

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

entity adders_1 is
 port(A,B : in std_logic_vector(7 downto 0);
  SUM : out std_logic_vector(7 downto 0));
end adders_1;

architecture archi of adders_1 is
begin

 SUM <= A + B;

end archi;
```

Unsigned 8-Bit Adder Verilog Coding Example

```
//
// Unsigned 8-bit Adder
//

module v_adders_1(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 Diagram



Unsigned 8-Bit Adder With Carry In Pin Descriptions IO Pins

IO Pins	Description
A, B	Add Operands
CI	Carry In
SUM	Add Result

Unsigned 8-Bit Adder With Carry In VHDL Coding Example

```
--
-- Unsigned 8-bit Adder with Carry In
--

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

entity adders_2 is
 port(A,B : in std_logic_vector(7 downto 0);
  CI : in std_logic;
  SUM : out std_logic_vector(7 downto 0));
end adders_2;

architecture archi of adders_2 is
begin

 SUM <= A + B + CI;

end archi;
```

Unsigned 8-Bit Adder With Carry In Verilog Coding Example

```
//
// Unsigned 8-bit Adder with Carry In
//

module v_adders_2(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

Before writing a **+** (plus) operation with carry out in VHDL, read the arithmetic package you plan to use. For example, **std_logic_unsigned** does not allow you to write **+** (plus) 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 **+** (plus) in this package is equal to the size of the longest argument (8 bits).

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

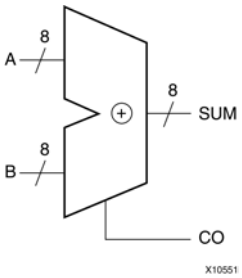
```
Res <= ("0" & A) + ("0" &   B);
```

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

Another solution is:

- Convert **A** and **B** to integers
- Convert the result back to the **std_logic** vector
- Specify the size of the vector equal to 9

Unsigned 8-Bit Adder With Carry Out Diagram



Unsigned 8-Bit Adder With Carry Out Pin Descriptions IO Pins

IO Pins	Description
A, B	Add Operands
SUM	Add Result
CO	Carry Out

Unsigned 8-Bit Adder With Carry Out VHDL Coding Example

```
--
-- 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 adders_3 is
 port(A,B : in std_logic_vector(7 downto 0);
  SUM : out std_logic_vector(7 downto 0);
  CO : out std_logic);
end adders_3;

architecture archi of adders_3 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;
```

The preceding example uses two arithmetic packages:

- **`std_logic_arith`**

  Contains the integer to **`std_logic`** conversion function (**`conv_std_logic_vector`**)

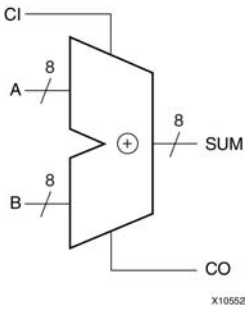
- **`std_logic_unsigned`**

  Contains the unsigned **+** (plus) operation

Unsigned 8-Bit Adder With Carry Out Verilog Coding Example

```
//
// Unsigned 8-bit Adder with Carry Out
//

module v_adders_3(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 Diagram



Unsigned 8-Bit Adder With Carry In and Carry Out Pin Descriptions IO Pins

IO Pins	Description
A, B	Add Operands
CI	Carry In
SUM	Add Result
CO	Carry Out

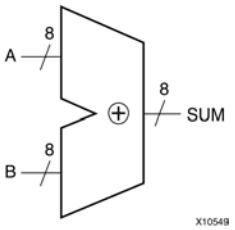
Unsigned 8-Bit Adder With Carry In and Carry Out VHDL Coding Example

```
--
-- 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 adders_4 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 adders_4;

architecture archi of adders_4 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;
```

Unsigned 8-Bit Adder With Carry In and Carry Out Verilog Coding Example

```
//
// Unsigned 8-bit Adder with Carry In and Carry Out
//

module v_adders_4(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
```

Signed 8-Bit Adder Diagram



Signed 8-Bit Adder Pin Descriptions IO Pins

IO Pins	Description
A, B	Add Operands
SUM	Add Result

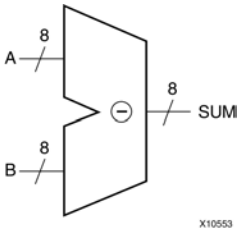
Signed 8-Bit Adder VHDL Coding Example

```
--
-- Signed 8-bit Adder
--

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

entity adders_5 is
 port(A,B : in std_logic_vector(7 downto 0);
  SUM : out std_logic_vector(7 downto 0));
end adders_5;

architecture archi of adders_5 is
begin

 SUM <= A + B;

end archi;
```

Signed 8-Bit Adder Verilog Coding Example

```
//
// Signed 8-bit Adder
//

module v_adders_5 (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 Diagram



Unsigned 8-Bit Subtractor Pin Descriptions IO Pins

IO Pins	Description
A, B	Sub Operands
RES	Sub Result

Unsigned 8-Bit Subtractor VHDL Coding Example

```
--
-- Unsigned 8-bit Subtractor
--

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

entity adders_6 is
 port(A,B : in std_logic_vector(7 downto 0);
  RES : out std_logic_vector(7 downto 0));
end adders_6;

architecture archi of adders_6 is
begin

 RES <= A - B;

end archi;
```

Unsigned 8-Bit Subtractor Verilog Coding Example

```
//
// Unsigned 8-bit Subtractor
//

module v_adders_6(A, B, RES);
 input [7:0] A;
 input [7:0] B;
 output [7:0] RES;

 assign RES = A - B;

endmodule
```

Unsigned 8-Bit Subtractor With Borrow In Pin Descriptions IO Pins

IO Pins	Description
A, B	Sub Operands
BI	Borrow In
RES	Sub Result

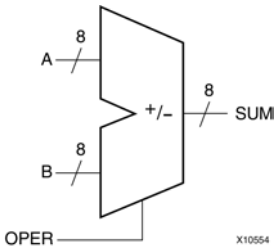
Unsigned 8-Bit Subtractor With Borrow In VHDL Coding Example

```
--
-- Unsigned 8-bit Subtractor with Borrow In
--

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


entity adders_8 is
 port(A,B : in std_logic_vector(7 downto 0);
      BI : in std_logic;
  RES : out std_logic_vector(7 downto 0));
end adders_8;

architecture archi of adders_8 is

begin

   RES   <= A - B - BI;

end archi;
```

Unsigned 8-Bit Subtractor With Borrow In Verilog Coding Example

```
//
// Unsigned 8-bit Subtractor with Borrow In
//

module v_adders_8(A, B, BI, RES);
 input [7:0] A;
 input [7:0] B;
 input BI;
 output [7:0] RES;

 assign RES = A - B - BI;

endmodule
```

Unsigned 8-Bit Adder/Subtractor Diagram



Unsigned 8-Bit Adder/Subtractor Pin Descriptions IO Pins

IO Pins	Description
A, B	Add/Sub Operands
OPER	Add/Sub Select
SUM	Add/Sub Result

Unsigned 8-Bit Adder/Subtractor VHDL Coding Example

```
--
-- Unsigned 8-bit Adder/Subtractor
--

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

entity adders_7 is
port(A,B : in std_logic_vector(7 downto 0);
 OPER: in std_logic;
 RES : out std_logic_vector(7 downto 0));
end adders_7;

architecture archi of adders_7 is
begin

 RES <= A + B when OPER='0'
  else A - B;

end archi;
```

Unsigned 8-Bit Adder/Subtractor Verilog Coding Example

```
//
// Unsigned 8-bit Adder/Subtractor
//

module v_adders_7(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
```

Comparators Hardware Description Language (HDL) Coding Techniques

This section discusses Comparators Hardware Description Language (HDL) Coding Techniques, and includes:

- Comparators Log File
- Comparators Related Constraints
- Comparators Coding Examples

Comparators 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
==============================
...
```

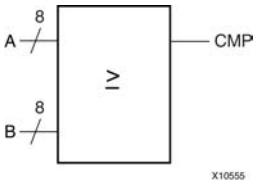
Comparators Related Constraints

None

Comparators Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Unsigned 8-Bit Greater or Equal Comparator Diagram



Unsigned 8-Bit Greater or Equal Comparator Pin Descriptions

IO Pins	Description
A, B	Comparison Operands
CMP	Comparison Result

Unsigned 8-Bit Greater or Equal Comparator VHDL Coding Example

```
--
-- Unsigned 8-bit Greater or Equal Comparator
--

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

entity comparator_1 is
    port(A,B : in  std_logic_vector(7 downto 0);
         CMP : out std_logic);
end comparator_1;

architecture archi of comparator_1 is
begin

    CMP <= '1' when A >= B else '0';

end archi;
```

Unsigned 8-Bit Greater or Equal Comparator Verilog Coding Example

```
//
// Unsigned 8-bit Greater or Equal Comparator
//

module v_comparator_1 (A, B, CMP);
    input  [7:0] A;
    input  [7:0] B;
    output CMP;

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

Multipliers Hardware Description Language (HDL) Coding Techniques

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

Registered Multipliers

In instances where a multiplier would have a registered output, XST infers a unique registered multiplier for the following devices:

- Virtex®-4 devices
- Virtex-5 devices

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 Multiplier Style (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

Multipliers

Note This section applies only to Virtex®-4 devices, Virtex-5 devices, and Spartan®-3A DSP devices.

Virtex-4 devices, Virtex-5 devices, and Spartan-3A DSP devices allow 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 obtain 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, Virtex-5 devices, and Spartan-3A DSP devices, XST can infer pipelined multipliers, not only for the LUT implementation, but for the DSP48 implementation as well. For more information, see XST Limitations.

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

In *auto* mode, use DSP Utilization Ratio (DSP_UTILIZATION_RATIO) to control DSP48 resources for the synthesis. By default, XST tries to utilize all DSP48 resources. For more information, see DSP48 Block Resources.

XST can automatically recognize the Multiplier Style (MULT_STYLE) constraint with values *lut* and *block* and then convert internally to Use DSP48 (USE_DSP48). Xilinx® recommends using the Use DSP48 (USE_DSP48) constraint for Virtex-4 device designs and Virtex-5 device designs to define FPGA resources used for multiplier implementation. Xilinx recommends using the Multiplier Style (MULT_STYLE) constraint to define the multiplier implementation method on the selected FPGA resources. If Use DSP48 (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 (USE_DSP48) is set to *no*, 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. To shape a macro in a specific way, use the Keep (KEEP) constraint. For example, to exclude the first register stage from the DSP48, place Keep (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 a constant using two methods:

- Constant Coefficient Multiplier (KCM)
- Canonical Signed Digit (CSD)

Dedicated implementations do not always provide the best results for multiplication with constants. XST can automatically choose between KCM or standard multiplier implementation. The CSD method cannot be automatically chosen. Use the Mux Style (MUX_STYLE) constraint to force CSD implementation.

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

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 Multiplier Style (MULT_STYLE) constraint.

Multipliers 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
=============================
...
```

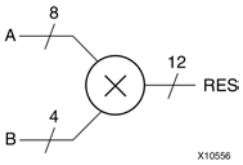
Multipliers Related Constraints

- Multiplier Style (MULT_STYLE)
- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Multipliers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Unsigned 8x4-Bit Multiplier Diagram



Unsigned 8x4-Bit Multiplier Pin Descriptions

IO Pins	Description
A, B	MULT Operands
RES	MULT Result

Unsigned 8x4-Bit Multiplier VHDL Coding Example

```
--
-- Unsigned 8x4-bit Multiplier
--

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

entity multipliers_1 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 multipliers_1;

architecture beh of multipliers_1 is
begin
    RES <= A * B;
end beh;
```

Unsigned 8x4-Bit Multiplier Verilog Coding Example

```
//
// Unsigned 8x4-bit Multiplier
//

module v_multipliers_1(A, B, RES);
    input [7:0] A;
    input [3:0] B;
    output [11:0] RES;

    assign RES = A * B;
endmodule
```

Sequential Complex Multipliers Hardware Description Language (HDL) Coding Techniques

A sequential complex multiplier is a complex multiplier that requires four cycles to make a complete multiplication by accumulating intermediate results. It requires one DSP block for implementation.

Multiplying two complex numbers A and B requires four cycles.

The first two first cycles compute:

```
Res_real = A_real * B_real - A_imag * B_imag
```

The second two cycles compute:

```
Res_imag = A_real * B_imag + A_imag * B_real
```

While several templates could be used to describe the above functionality, XST does not support using **enum** or **integer** types to describe the different DSP modes and store the **enum** values. Instead, Xilinx® recommends a very regular template to ease XST inferencing. This general accumulator template allows XST to inference a single DSP to perform the following operations:

- **Load:P <= Value**
- **Load: P <= -Value**
- **Accumulate: P <= P + Value**
- **Accumulate: P <= P - Value**

This template works with the following two control signals that perform the above four operations when combined:

- **load**
- **addsub**

Sequential Complex Multipliers Log File

None

Sequential Complex Multipliers Related Constraints

None

Sequential Complex Multipliers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Signed 18x18-bit Sequential Complex Multiplier Pin Descriptions

IO Pins	Description
CLK	Clock Signal
Oper_Load, Oper_AddSub	Control Signals controlling Load and AddSub Operations
A, B	MULT Operands
RES	MULT Result

Signed 18x18-bit Sequential Complex Multiplier VHDL Coding Example

```
--
-- Sequential Complex Multiplier
--

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

entity multipliers_8 is
    generic(A_WIDTH:        positive:=18;
            B_WIDTH:        positive:=18;
            RES_WIDTH:      positive:=48);
    port(   CLK:            in  std_logic;
            A:              in  signed(A_WIDTH-1 downto 0);
            B:              in  signed(B_WIDTH-1 downto 0);

            Oper_Load:      in  std_logic;
            Oper_AddSub:    in  std_logic;
            --  Oper_Load  Oper_AddSub  Operation
            --  0          0            R= +A*B
            --  0          1            R= -A*B
```

```
            --  1          0           R=R+A*B
            --  1          1           R=R-A*B

            RES:             out signed(RES_WIDTH-1 downto 0)
    );
end multipliers_8;

architecture beh of multipliers_8 is

    constant P_WIDTH: integer:=A_WIDTH+B_WIDTH;

    signal oper_load0: std_logic:='0';
    signal oper_addsub0: std_logic:='0';

    signal p1: signed(P_WIDTH-1 downto 0):=(others=>'0');
    signal oper_load1: std_logic:='0';
    signal oper_addsub1: std_logic:='0';

    signal res0: signed(RES_WIDTH-1 downto 0);
begin

    process (clk)
        variable acc: signed(RES_WIDTH-1 downto 0);
    begin
        if rising_edge(clk) then
            oper_load0   <= Oper_Load;
            oper_addsub0 <= Oper_AddSub;

            p1 <= A*B;
            oper_load1   <= oper_load0;
            oper_addsub1 <= oper_addsub0;

            if (oper_load1='1') then
                acc := res0;
            else
                acc := (others=>'0');
            end if;

            if (oper_addsub1='1') then
                res0 <= acc-p1;
            else
                res0 <= acc+p1;
            end if;

        end if;
    end process;

    RES <= res0;

end architecture;
```

Signed 18x18-bit Sequential Complex Multiplier Verilog Coding Example

```
module v_multipliers_8(CLK,A,B,Oper_Load,Oper_AddSub, RES);
    parameter A_WIDTH   = 18;
    parameter B_WIDTH   = 18;
    parameter RES_WIDTH = 48;
    parameter P_WIDTH   = A_WIDTH+B_WIDTH;

    input  CLK;
    input  signed [A_WIDTH-1:0] A, B;

    input  Oper_Load, Oper_AddSub;
    //  Oper_Load  Oper_AddSub  Operation
    //  0          0            R= +A*B
    //  0          1            R= -A*B
    //  1          0            R=R+A*B
    //  1          1            R=R-A*B

    output [RES_WIDTH-1:0] RES;

    reg oper_load0   = 0;
    reg oper_addsub0 = 0;

    reg signed [P_WIDTH-1:0] p1 = 0;
    reg oper_load1   = 0;
    reg oper_addsub1 = 0;

    reg signed [RES_WIDTH-1:0] res0  = 0;
    reg signed [RES_WIDTH-1:0] acc;

    always @(posedge CLK)
    begin
        oper_load0   <= Oper_Load;
        oper_addsub0 <= Oper_AddSub;

        p1 <= A*B;
        oper_load1   <= oper_load0;
        oper_addsub1 <= oper_addsub0;

        if (oper_load1==1'b1)
            acc = res0;
        else
            acc = 0;

        if (oper_addsub1==1'b1)
            res0 <= acc-p1;
        else
            res0 <= acc+p1;

    end

    assign RES = res0;

endmodule
```

Pipelined Multipliers Hardware Description Language (HDL) Coding Techniques

In order to increase the speed of designs with large multipliers, XST can infer 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.

To insert pipeline stages:

1. Describe the necessary registers in your HDL code
2. Place them after any multipliers
3. Set the Multiplier Style (MULT_STYLE) constraint to **pipe_lut**

If the target is a Virtex®-4 device or a Virtex-5 device, and implementation of a multiplier requires multiple DSP48 blocks, XST can pipeline this implementation as well. Set Multiplier Style (MULT_STYLE) for this instance to **`pipe_block`**.

In order to reach the maximum multiplier speed, XST uses the maximum number of available registers when:

- XST detects valid registers for pipelining, and
- Multiplier Style (MULT_STYLE) is set to **`pipe_lut`** or **`pipe_block`**

In order to obtain the best frequency, XST automatically calculates the maximum number of registers for each multiplier .

During the Advanced HDL Synthesis step, the XST HDL Advisor advises you to specify the optimum number of register stages if:

- You have not specified sufficient register stages, and
- Multiplier Style (MULT_STYLE) is coded directly on a signal,

XST implements the unused stages as shift registers if:

- The number of registers placed after the multiplier exceeds the maximum required, and
- Shift register extraction is activated

XST has the following 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.

Pipelined Multipliers Log File

Following is a Pipelined Multipliers log file.

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

Synthesizing Unit <multipliers_2>.
    Related source file is "multipliers_2.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>.
    Found 36-bit register for signal <pipe_2>.
    Found 36-bit register for signal <pipe_3>.
    Summary:
        inferred 180 D-type flip-flop(s).
        inferred   1 Multiplier(s).
Unit <multipliers_2> synthesized.
...
=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================

Synthesizing (advanced) Unit <multipliers_2>.
Found pipelined multiplier on signal <mult_res>:
- 4 pipeline level(s) found in a register connected to the
multiplier macro output.
Pushing register(s) into the multiplier macro.
INFO:Xst - HDL ADVISOR - You can improve the performance of the
multiplier Mmult_mult_res by adding 1 register level(s).
Unit <multipliers_2> synthesized (advanced).

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

Macro Statistics
# Multipliers                          : 1
 18x18-bit registered multiplier       : 1
=========================================================================
```

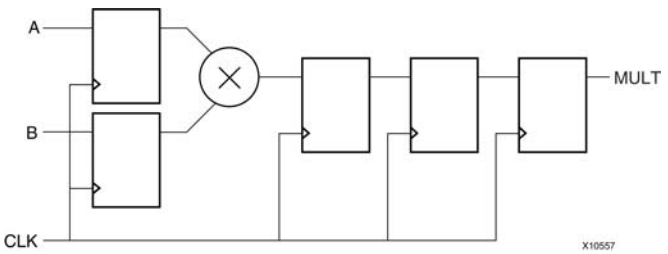
Pipelined Multipliers Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)
- Multiplier Style (MULT_STYLE)

Pipelined Multipliers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Pipelined Multiplier (Outside, Single) Diagram



Pipelined Multiplier (Outside, Single) Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
A, B	MULT Operands
MULT	MULT Result

Pipelined Multiplier (Outside, Single) VHDL Coding Example

```
--
-- Pipelined multiplier
--    The multiplication operation placed outside 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 multipliers_2 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));

    attribute mult_style: string;
    attribute mult_style of multipliers_2: entity is "pipe_lut";

end multipliers_2;

architecture beh of multipliers_2 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;
```

Pipelined Multiplier (Outside, Single) Verilog Coding Example

```
//
// Pipelined multiplier
//    The multiplication operation placed outside the
//    always block and the pipeline stages represented
//    as single registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_2(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
```

Pipelined Multiplier (Inside, Single) Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
A, B	MULT Operands
MULT	MULT Result

Pipelined Multiplier (Inside, Single) VHDL Coding Example

```
--
-- Pipelined multiplier
--    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 multipliers_3 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));

    attribute mult_style: string;
    attribute mult_style of multipliers_3: entity is "pipe_lut";

end multipliers_3;

architecture beh of multipliers_3 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;
```

Pipelined Multiplier (Inside, Single) Verilog Coding Example

```
//
// Pipelined multiplier
//    The multiplication operation placed inside the
//    process block and the pipeline stages are represented
//    as single registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_3(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
```

Pipelined Multiplier (Outside, Shift) Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
A, B	MULT Operands
MULT	MULT Result

Pipelined Multiplier (Outside, Shift) VHDL Coding Example

```
--
-- Pipelined multiplier
--    The multiplication operation placed outside the
--    process block and the pipeline stages represented
--    as shift registers.
--

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

entity multipliers_4 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));

    attribute mult_style: string;
    attribute mult_style of multipliers_4: entity is "pipe_lut";

end multipliers_4;

architecture beh of multipliers_4 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;
```

Pipelined Multiplier (Outside, Shift) Verilog Coding Example

```
//
// Pipelined multiplier
//    The multiplication operation placed outside the
//    always block and the pipeline stages represented
//    as shift registers.
//

(*mult_style="pipe_lut"*)
module v_multipliers_4(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 [2:0];
    integer i;

    assign mult_res = a_in * b_in;

    always @(posedge clk)
    begin
        a_in <= A; b_in <= B;

        pipe_regs[2] <= mult_res;
        for (i=0; i<=1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];

        MULT <= pipe_regs[0];
    end

endmodule
```

Multiply Adder/Subtractors Hardware Description Language (HDL) Coding Techniques

The Multiply Adder/Subtractor macro is a complex macro consisting of several basic macros such as:

- Multipliers
- Adder/subtractors
- Registers

The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources in the following devices:

- Virtex®-4
- Virtex-5

Multiply Adder/Subtractors in Virtex®-4 Devices and Virtex-5 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. For more information, see Multipliers Hardware Description Language (HDL) Coding Techniques and Adders, Subtractors, and Adders/Subtractors Hardware Description Language (HDL) Coding Techniques.

Macro implementation on DSP48 blocks is controlled by the Use DSP48 (USE_DSP48) constraint or command line option, with default value of *auto*. In this mode, XST implements multiply adder/subtractors taking into account DSP48 resources in the device.

In auto mode, use the DSP Utilization Ratio (DSP_UTILIZATION_RATIO) constraint to control DSP48 resources for the synthesis. By default, XST tries to utilize all available DSP48 resources. For more information, see DSP48 Block Resources.

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. To shape a macro in a specific way, use the Keep (KEEP) constraint. For example, to exclude the first register stage from the DSP48, place Keep (KEEP) constraints on the outputs of these registers.

In the log file, XST reports the details of inferred multipliers, adders, 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.

Multiply Adder/Subtractors 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 <multipliers_6>.
    Related source file is "multipliers_6.vhd".
    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 8x8-bit multiplier for signal <mult>.
    Found 16-bit addsub for signal <multaddsub>.
    Summary:
        inferred  32 D-type flip-flop(s).
        inferred   1 Adder/Subtractor(s).
        inferred   1 Multiplier(s).
Unit <multipliers_6> synthesized.
...
=========================================================================
*                    Advanced HDL Synthesis                             *
=========================================================================
...
Synthesizing (advanced) Unit <Mmult_mult>.
        Multiplier <Mmult_mult> in block <multipliers_6> and adder/subtractor
  <Maddsub_multaddsub> in block <multipliers_6> are combined into a
  MAC<Mmac_Maddsub_multaddsub>.
        The following registers are also absorbed by the MAC: <A_reg2> in block
  <multipliers_6>, <A_reg1> in block <multipliers_6>, <B_reg2> in
  block <multipliers_6>, <B_reg1> in block <multipliers_6>.
Unit <Mmult_mult> synthesized (advanced).

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

Macro Statistics
# MACs                          : 1
 8x8-to-16-bit MAC              : 1
=========================================================================
```

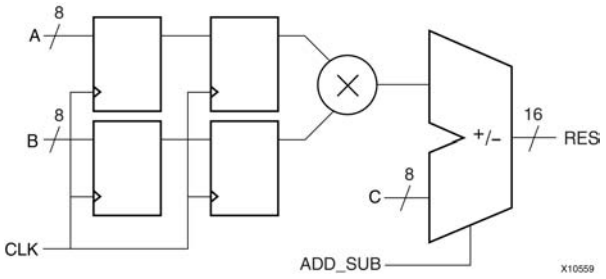
Multiply Adder/Subtractors Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Multiply Adder/Subtractors Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Multiplier Adder With 2 Register Levels on Multiplier Inputs Diagram



Multiplier Adder With 2 Register Levels on Multiplier Inputs Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
A, B, C	MULT-Add Operands
RES	MULT-Add Result

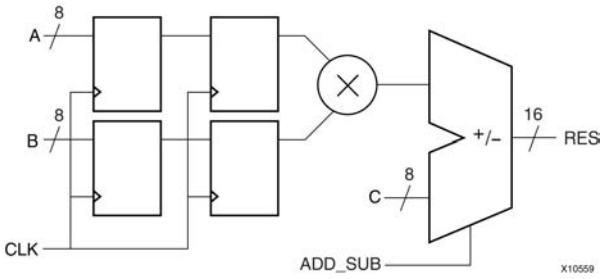
Multiplier Adder With 2 Register Levels on Multiplier Inputs VHDL Coding Example

```
--
-- Multiplier Adder with 2 Register Levels on Multiplier Inputs
--

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

entity multipliers_5 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 multipliers_5;

architecture beh of multipliers_5 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;
```

Multiplier Adder With 2 Register Levels on Multiplier Inputs Verilog Coding Example

```
//
// Multiplier Adder with 2 Register Levels on Multiplier Inputs
//

module v_multipliers_5 (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 Diagram



Multiplier Adder/Subtractor With 2 Register Levels On Multiplier Inputs Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
add_sub	AddSub Selector
A, B, C	MULT-AddSub Operands
RES	MULT-AddSub Result

Multiplier Adder/Subtractor With 2 Register Levels On Multiplier Inputs VHDL Coding Example

```
--
-- Multiplier Adder/Subtractor with
-- 2 Register Levels on Multiplier Inputs
--


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

entity multipliers_6 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 multipliers_6;

architecture beh of multipliers_6 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 <= C + mult when add_sub = '1' else C - mult;

    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;
```

Multiplier Adder/Subtractor With 2 Register Levels On Multiplier Inputs Verilog Coding Example

```
//
// Multiplier Adder/Subtractor with
// 2 Register Levels on Multiplier Inputs
//

module v_multipliers_6 (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 ? C + mult : C - mult;
    assign RES = multaddsub;

endmodule
```

Multiply Accumulate Hardware Description Language (HDL) Coding Techniques

The Multiply Accumulate macro is a complex macro consisting of several basic macros such as:

- Multipliers
- Accumulators
- Registers

The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources in the following devices:

- Virtex®-4
- Virtex-5

Multiply Accumulate in Virtex-4 Devices and Virtex-5 Devices

The Multiply Accumulate macro is a complex macro consisting of several basic macros as multipliers, accumulators, and registers. The recognition of this complex macro enables XST to implement it on dedicated DSP48 resources in Virtex®-4 devices and Virtex-5 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 two separate Multiplier and Accumulate macros, making independent decisions on each macro. For more information, see Multipliers Hardware Description Language (HDL) Coding Techniques and Accumulators Hardware Description Language (HDL) Coding Techniques.

Macro implementation on DSP48 blocks is controlled by the Use DSP48 (USE_DSP48) constraint or command line option, with default value of *auto*. In auto mode, XST implements multiply accumulate taking into account available DSP48 resources in the device.

In auto mode, use DSP Utilization Ratio (DSP_UTILIZATION_RATIO) to control DSP48 resources. XST tries to utilize as many DSP48 resources as possible. For more information, see DSP48 Block Resources.

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. To shape a macro in a specific way, use the Keep (KEEP) constraint. For example, to exclude the first register stage from the DSP48, place Keep (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.

Multiply Accumulate 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 <multipliers_7a>.
    Related source file is "multipliers_7a.vhd".
    Found 8x8-bit multiplier for signal <$n0002> created at line 28.
    Found 16-bit up accumulator for signal <accum>.
    Found 16-bit register for signal <mult>.
    Summary:
        inferred   1 Accumulator(s).
        inferred  16 D-type flip-flop(s).
        inferred   1 Multiplier(s).
Unit <multipliers_7a> synthesized....
=========================================================================
*                    Advanced HDL Synthesis                             *
=========================================================================
...
Synthesizing (advanced) Unit <Mmult__n0002>.
        Multiplier <Mmult__n0002> in block <multipliers_7a> and accumulator
  <accum> in block <multipliers_7a> are combined into a MAC<Mmac_accum>.
        The following registers are also absorbed by the MAC: <mult> in block
   <multipliers_7a>. Unit <Mmult__n0002> synthesized (advanced).

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

Macro Statistics
# MACs                              : 1
 8x8-to-16-bit MAC                  : 1

=========================================================================
```

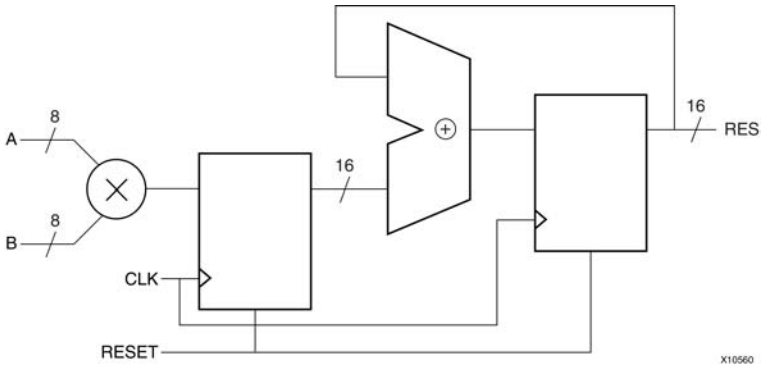
Multiply Accumulate Related Constraints

- Use DSP48 (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)

Multiply Accumulate Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Multiplier Up Accumulate With Register After Multiplication Diagram



Multiplier Up Accumulate With Register After Multiplication Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
reset	Synchronous Reset
A, B	MAC Operands
RES	MAC Result

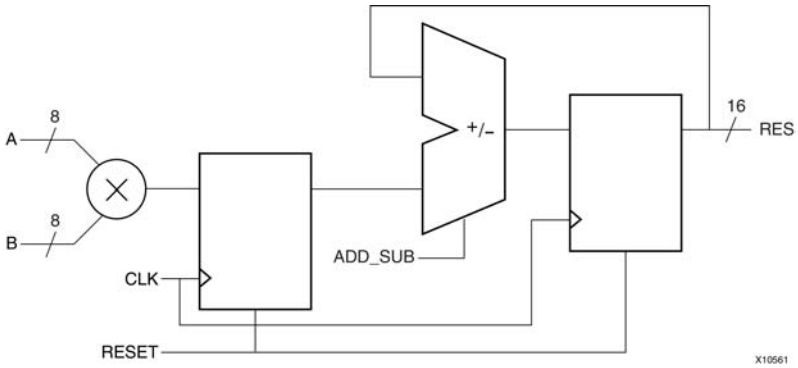
Multiplier Up Accumulate With Register After Multiplication VHDL Coding Example

```
--
-- Multiplier Up Accumulate with Register After Multiplication
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity multipliers_7a 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 multipliers_7a;

architecture beh of multipliers_7a 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;
```

Multiplier Up Accumulate With Register After Multiplication Verilog Coding Example

```
//
// Multiplier Up Accumulate with Register After Multiplication
//
module v_multipliers_7a (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 Diagram



Multiplier Up/Down Accumulate With Register After Multiplication Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
reset	Synchronous Reset
add_sub	AddSub Selector
A, B	MAC Operands
RES	MAC Result

Multiplier Up/Down Accumulate With Register After Multiplication VHDL Coding Example

```
--
-- Multiplier Up/Down Accumulate with Register After Multiplication.
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity multipliers_7b 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 multipliers_7b;

architecture beh of multipliers_7b 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;
```

Multiplier Up/Down Accumulate With Register After Multiplication Verilog Coding Example

```
//
// Multiplier Up/Down Accumulate with Register After Multiplication.
//

module v_multipliers_7b (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 Hardware Description Language (HDL) Coding Techniques

Dividers are supported only 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.

Dividers 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. If the divider does not correspond to the case supported by XST, then XST issues the following error message:

```
...
ERROR:Xst:719 - file1.vhd (Line 172).
Operator is not supported yet : 'DIVIDE'
...
```

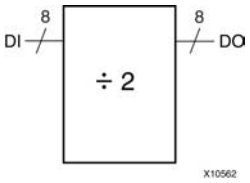
Dividers Related Constraints

None

Dividers Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Division by Constant 2 Divider Diagram



Division by Constant 2 Divider Pin Descriptions

IO Pins	Description
DI	Division Operands
DO	Division Result

Division by Constant 2 Divider VHDL Coding Example

```
--
-- Division By Constant 2
--

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

entity divider_1 is
    port(DI : in  unsigned(7 downto 0);
         DO : out unsigned(7 downto 0));
end divider_1;

architecture archi of divider_1 is
begin

    DO <= DI / 2;

end archi;
```

Division by Constant 2 Divider Verilog Coding Example

```
//
// Division By Constant 2
//

module v_divider_1 (DI, DO);
    input  [7:0] DI;
    output [7:0] DO;

    assign DO = DI / 2;

endmodule
```

Resource Sharing Hardware Description Language (HDL) Coding Techniques

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.

XST supports resource sharing for:

- Adders
- Subtractors
- Adders/subtractors
- Multipliers

If the optimization goal is speed, disabling resource sharing may give better results. To improve clock frequency, XST recommends deactivating resource sharing at the Advanced HDL Synthesis step.

Resource Sharing 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.
...
```

Resource Sharing Related Constraints

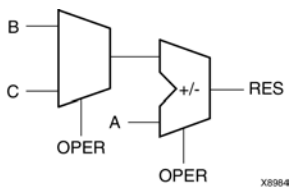
Resource Sharing (RESOURCE_SHARING)

Resource Sharing Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

For the following VHDL and Verilog examples, XST gives the following solution.

Resource Sharing Diagram



Resource Sharing Pin Descriptions

IO Pins	Description
A, B, C	Operands
OPER	Operation Selector
RES	Data Output

Resource Sharing VHDL Coding Example

```
--
-- Resource Sharing
--

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

entity resource_sharing_1 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 resource_sharing_1;

architecture archi of resource_sharing_1 is
begin

    RES <= A + B when OPER='0' else A - C;

end archi;
```

Resource Sharing Verilog Coding Example

```
//
// Resource Sharing
//


module v_resource_sharing_1 (A, B, C, OPER, RES);
    input  [7:0] A, B, C;
    input  OPER;
    output [7:0] RES;
    wire   [7:0] RES;

    assign RES = !OPER ? A + B : A - C;

endmodule
```

RAMs and ROMs Hardware Description Language (HDL) Coding Techniques

If you do not want to instantiate RAM primitives to keep your Hardware Description Language (HDL) code architecture independent, use XST automatic RAM recognition. 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 and dual-port write
- Parity bits
- Block Ram with Byte-Wide Write Enable
- Simple dual-port BRAM

XST does not support RAMs and 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.

If a given template can be implemented using Block and Distributed RAM, XST implements BLOCK ones. Use the RAM Style (RAM_STYLE) constraint to control RAM implementation and select a desirable RAM type. For more information, see XST Design Constraints.

The following block RAM features are *not* supported:

- Parity bits
- Different aspect ratios on each port
- Simple dual-port distributed RAMs
- Quad-port distributed RAMs

XST uses speed-oriented implementation to implement RAMs on BRAM resources. This gives good results for speed, but may require more BRAM resources than area-oriented implementation. XST does not support area-oriented BRAM implementation. Xilinx® recommends the CORE Generator™ software for area-oriented implementation.

For more information on RAM implementation, see XST FPGA Optimization.

XST can:

- Implement Finite State Machine (FSM) components For more information, see Finite State Machine (FSM) Hardware Description Language (HDL) Coding Techniques.
- Map general logic onto block RAMs. For more information, see Mapping Logic Onto Block RAM.

XST automatically controls BRAM resources on the target device. BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) allows you to specify the number of BRAM blocks that XST must not exceed during synthesis.

To achieve better design speed, XST implements small RAMs and ROMs using distributed resources. RAMs and ROMs are considered *small* if their sizes follow the rules shown in the following table.

Rules for Small RAMs and ROMs

Devices	Size (bits) * Width (bits)
Virtex®-4	<= 512
Virtex-5	<= 512

Use RAM Style (RAM_STYLE) and ROM Style (ROM_STYLE) to force implementation of small RAMs and ROMs on BRAM resources.

XST calculates the available BRAM resources for inference using the following formula:

`Total_Number_of_Available_BRAMs - Number_of_Reserved_BRAMs`

In this formula

`Total_Number_of_Available_BRAMs`

is the number of BRAMs specified by the BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) constraint. By default it is 100%.

The **`Number of Reserved_BRAMs`** encapsulates:

- The number of instantiated BRAMs in the Hardware Description Language (HDL) code from the UNISIM library
- The number of RAM which were forced to be implemented as BRAMs by the RAM Style (RAM_STYLE) and ROM Style (ROM_STYLE) constraints
- The number of BRAMs generated using BRAM mapping optimizations (BRAM_MAP).

Where there are available BRAM resources, XST implements the largest inferred RAMs and ROMs using BRAM, and the smallest on distributed resources.

If the **`Number_of_Reserved_BRAMs`** exceeds available resources, XST implements them as block RAMs, and all inferred RAMs are implemented on distributed memory.

As soon as this process is completed, XST can automatically pack two small single-port BRAMs in a single BRAM primitive. This optimization is controlled by the Automatic BRAM Packing (AUTO_BRAM_PACKING) constraint. It is disabled by default.

For more information, see BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) and Automatic BRAM Packing (AUTO_BRAM_PACKING).

RAMs and ROMs Log File

The XST log file reports the type and size of recognized RAM as well as complete information on its I/O ports. RAM recognition consists of two steps.

- During the HDL Synthesis step, XST recognizes the presence of the memory structure in the Hardware Description Language (HDL) code.
- During the Advanced HDL Synthesis step, XST decides how to implement a specific memory (that is, whether to use Block or Distributed memory resources).

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

Synthesizing Unit <rams_16>.
 Related source file is "rams_16.vhd".
 Found 64x16-bit dual-port RAM <Mram_RAM> for signal <RAM>.
 Found 16-bit register for signal <doa>.
 Found 16-bit register for signal <dob>.
 Summary:
 inferred 1 RAM(s).
inferred32 D-type flip-flop(s).
Unit <rams_16> synthesized.
```

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

Macro Statistics
# RAMs : 1
 64x16-bit dual-port RAM : 1
# Registers: 2
 16-bit register : 2

=========================================================================

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

Synthesizing (advanced) Unit <rams_16>.
INFO:Xst - The RAM <Mram_RAM> will be implemented as a BLOCK RAM, absorbing
 the following register(s): <doa> <dob>

 ----------------------------------------------------------------------
    | ram_type           | Block                                 |          |
    -----------------------------------------------------------------------
    | Port A                                                               |
    |     aspect ratio   | 64-word x 16-bit                      |          |
    |     mode           | write-first                           |          |
    |     clkA           | connected to signal <clka>            | rise     |
    |     enA            | connected to signal <ena>             | high     |
    |     weA            | connected to internal <wea>           | high     |
    |     addrA          | connected to signal <addra>           |          |
    |     diA            | connected to internal <dia>           |          |
    |     doA            | connected to signal <doa>             |          |
    -----------------------------------------------------------------------
    | optimization       | speed                                 |          |
=========================================================================

-----------------------------------------------------------------------
    | ram_type           | Block                                 |          |
    -----------------------------------------------------------------------
    | Port B                                                               |
    |     aspect ratio   | 64-word x 16-bit                      |          |
    |     mode           | write-first                           |          |
    |     clkB           | connected to signal <clkb>            | rise     |
    |     enB            | connected to signal <enb>             | high     |
    |     weB            | connected to internal <web>           | high     |
    |     addrB          | connected to signal <addrb>           |          |
    |     diB            | connected to internal <dib>           |          |
    |     doB            | connected to signal <dob>             |          |
    -----------------------------------------------------------------------
    | optimization       | speed                                 |          |
=========================================================================
-----------------------------------------------------------------------
Unit <rams_16> synthesized (advanced).

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

Macro Statistics
# RAMs : 1
 64x16-bit dual-port block RAM : 1

=========================================================================
```

RAMs and ROMs Related Constraints

- BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)
- Automatic BRAM Packing (AUTO_BRAM_PACKING)
- RAM Extraction (RAM_EXTRACT)
- RAM Style (RAM_STYLE)
- ROM Extraction (ROM_EXTRACT)
- ROM Style (ROM_STYLE)

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.

RAMs and ROMs Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Block RAM resources in the following devices offer different read/write synchronization modes:

- Virtex®-4
- Virtex-5
- Spartan®-3
- Spartan-3E
- Spartan-3A

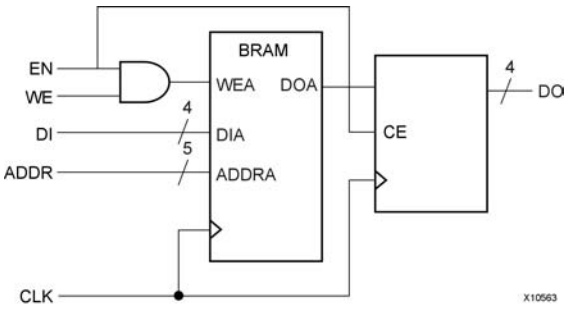
The following coding examples describe a single-port block RAM. You can deduce descriptions of dual-port block RAMs from these examples. Dual-port block RAMs can be configured with a different read/write mode on each port. Inference supports this capability.

Support For Read/Write Modes summarizes support for read/write modes according to the targeted devices and how XST handles it.

Support For Read/Write Modes

Devices	Inferred Modes	Behavior
Spartan-3 Spartan-3E Spartan-3A Virtex-4 Virtex-5	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
CPLD	none	RAM inference completely disabled

Single-Port RAM in Read-First Mode Diagram



Single-Port RAM in Read-First Mode Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
en	Clock Enable
addr	Read/Write Address
di	Data Input
do	Data Output

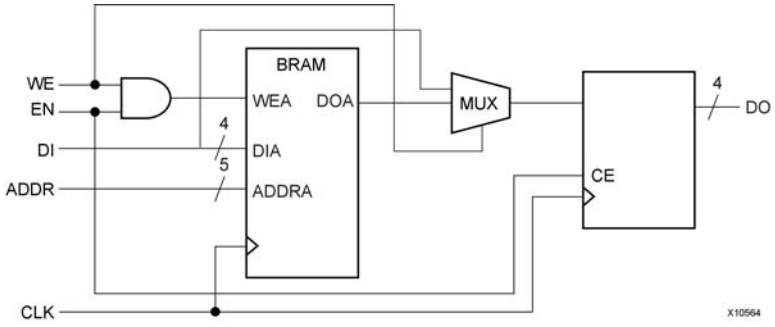
Single-Port RAM in Read-First Mode VHDL Coding Example One

```
--
-- Read-First Mode
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity rams_01 is
    port (clk  : in std_logic;
          we   : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Single-Port RAM in Read-First Mode Verilog Coding Example One

```
//
// Read-First Mode
//
module v_rams_01 (clk, en, we, addr, di, do);
    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
              RAM[addr]<=di;
            do <= RAM[addr];
        end
    end

endmodule
```

Single-Port RAM in Write-First Mode Diagram



Single-Port RAM in Write-First Mode Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
en	Clock Enable
addr	Read/Write Address
di	Data Input
do	Data Output

Single-Port RAM in Write-First Mode VHDL Coding Example One

```
--
-- Write-First Mode (template 1)
--

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

entity rams_02a is
    port (clk  : in std_logic;
          we   : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_02a;

architecture syn of rams_02a is
    type ram_type is array (63 downto 0)
         of std_logic_vector (15 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;
```

Single-Port RAM in Write-First Mode VHDL Coding Example Two

```
--
-- Write-First Mode (template 2)
--

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

entity rams_02b is
port (clk  : in std_logic;
      we   : in std_logic;
      en   : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      di   : in std_logic_vector(15 downto 0);
      do   : out std_logic_vector(15 downto 0));
end rams_02b;

architecture syn of rams_02b is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addr: std_logic_vector(5 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;
                end if;
                read_addr <= addr;
            end if;
        end if;
    end process;

    do <= ram(conv_integer(read_addr));

end syn;
```

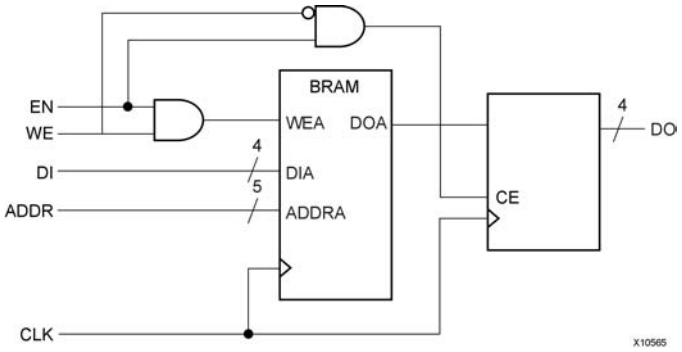
Single-Port RAM In Write-First Mode Verilog Coding Example One

```
//
// Write-First Mode (template 1)
//


module v_rams_02a (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
            begin
                RAM[addr] <= di;
                do <= di;
            end
            else
                do <= RAM[addr];
        end
    end
endmodule
```

Single-Port RAM In Write-First Mode Verilog Coding Example Two

```
//
// Write-First Mode (template 2)
//

module v_rams_02b (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [5:0] 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
```

Single-Port RAM In No-Change Mode Diagram



Single-Port RAM In No-Change Mode Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
en	Clock Enable
addr	Read/Write Address
di	Data Input
do	Data Output

Single-Port RAM In No-Change Mode VHDL Coding Example Two

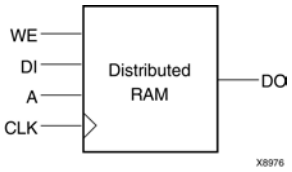
```
--
-- No-Change Mode (template 1)
--

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

entity rams_03 is
    port (clk  : in std_logic;
          we   : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_03;

architecture syn of rams_03 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Single-Port RAM In No-Change Mode Verilog Coding Example Two

```
//
// No-Change Mode (template 1)
//

module v_rams_03 (clk, we, en, addr, di, do);

    input  clk;
    input  we;
    input  en;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
              RAM[addr] <= di;
            else
              do <= RAM[addr];
        end
    end

endmodule
```

The following descriptions are directly mappable onto *distributed RAM only*.

Single-Port RAM With Asynchronous Read Diagram



Single-Port RAM With Asynchronous Read Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

Single-Port RAM With Asynchronous Read VHDL Coding Example

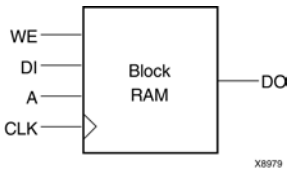
```
--
-- Single-Port RAM with Asynchronous Read
--

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

entity rams_04 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_04;

architecture syn of rams_04 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Single-Port RAM With Asynchronous Read Verilog Coding Example

```
//
// Single-Port RAM with Asynchronous Read
//

module v_rams_04 (clk, we, a, di, do);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end

    assign do = ram[a];

endmodule
```

The following description implements a true synchronous read. A true synchronous read is the synchronization mechanism in Virtex device block RAMs, where the read address is registered on the RAM clock edge. Such descriptions are *directly mappable onto block RAM*, as shown in the diagram below. The same descriptions can also be mapped onto *Distributed RAM*.

Single-Port RAM With Synchronous Read (Read Through) Diagram



Single-Port RAM With Synchronous Read (Read Through) Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
a	Read/Write Address
di	Data Input
do	Data Output

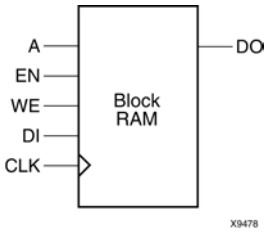
Single-Port RAM With Synchronous Read (Read Through) VHDL Coding Example

```
--
-- Single-Port RAM with Synchronous Read (Read Through)
--

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

entity rams_07 is
    port (clk : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_07;

architecture syn of rams_07 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 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;
```

Single-Port RAM With Synchronous Read (Read Through) Verilog Coding Example

```
//
// Single-Port RAM with Synchronous Read (Read Through)
//

module v_rams_07 (clk, we, a, di, do);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [5:0] 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 Diagram



Single-Port RAM With Enable Pin Descriptions

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

Single-Port RAM With Enable VHDL Coding Example

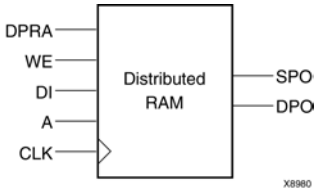
```
--
-- Single-Port RAM with Enable
--

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

entity rams_08 is
    port (clk : in std_logic;
          en  : in std_logic;
          we  : in std_logic;
          a   : in std_logic_vector(5 downto 0);
          di  : in std_logic_vector(15 downto 0);
          do  : out std_logic_vector(15 downto 0));
end rams_08;

architecture syn of rams_08 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 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;
```

Single-Port RAM With Enable Verilog Coding Example

```
//
// Single-Port RAM with Enable
//

module v_rams_08 (clk, en, we, a, di, do);

    input  clk;
    input  en;
    input  we;
    input  [5:0] a;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [5:0] 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
```

The following diagram shows where the two output ports are used. It is directly mappable onto *Distributed RAM only*.

Dual-Port RAM With Asynchronous Read Diagram



Dual-Port RAM With Asynchronous Read Pin Descriptions

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

Dual-Port RAM With Asynchronous Read VHDL Coding Example

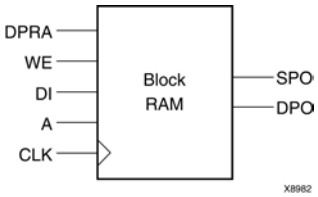
```
--
-- Dual-Port RAM with Asynchronous Read
--

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

entity rams_09 is
    port (clk  : in std_logic;
          we   : in std_logic;
          a    : in std_logic_vector(5 downto 0);
          dpra : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          spo  : out std_logic_vector(15 downto 0);
          dpo  : out std_logic_vector(15 downto 0));
end rams_09;

architecture syn of rams_09 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Dual-Port RAM With Asynchronous Read Verilog Coding Example

```
//
// Dual-Port RAM with Asynchronous Read
//

module v_rams_09 (clk, we, a, dpra, di, spo, dpo);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];

    always @(posedge clk) begin
        if (we)
            ram[a] <= di;
    end

    assign spo = ram[a];
    assign dpo = ram[dpra];

endmodule
```

The following descriptions are directly mappable onto block RAM, as shown in the diagram below. They may also be implemented with Distributed RAM.

Dual-Port RAM With Synchronous Read (Read Through) Diagram



Dual-Port RAM With Synchronous Read (Read Through) Pin Descriptions

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

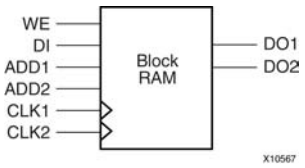
Dual-Port RAM With Synchronous Read (Read Through) VHDL Coding Example

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
--

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

entity rams_11 is
    port (clk  : in std_logic;
          we   : in std_logic;
          a    : in std_logic_vector(5 downto 0);
          dpra : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          spo  : out std_logic_vector(15 downto 0);
          dpo  : out std_logic_vector(15 downto 0));
end rams_11;

architecture syn of rams_11 is
    type ram_type is array (63 downto 0)
         of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_a : std_logic_vector(5 downto 0);
    signal read_dpra : std_logic_vector(5 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;
```

Dual-Port RAM With Synchronous Read (Read Through) Verilog Coding Example

```
//
// Dual-Port RAM with Synchronous Read (Read Through)
//

module v_rams_11 (clk, we, a, dpra, di, spo, dpo);

    input  clk;
    input  we;
    input  [5:0] a;
    input  [5:0] dpra;
    input  [15:0] di;
    output [15:0] spo;
    output [15:0] dpo;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_a;
    reg    [5:0] 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
```

Dual-Port RAM With Synchronous Read (Read Through) and Two Clocks Diagram



Dual-Port RAM With Synchronous Read (Read Through) and Two Clocks Pin Descriptions

IO Pins	Description
clk1	Positive-Edge Write/Primary Read Clock
clk2	Positive-Edge Dual Read Clock
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

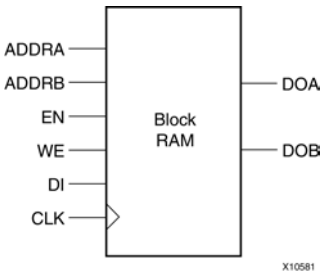
Dual-Port RAM With Synchronous Read (Read Through) and Two Clocks VHDL Coding Example

```
--
-- Dual-Port RAM with Synchronous Read (Read Through)
-- using More than One Clock
--

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

entity rams_12 is
    port (clk1 : in std_logic;
          clk2 : in std_logic;
          we   : in std_logic;
          add1 : in std_logic_vector(5 downto 0);
          add2 : in std_logic_vector(5 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 rams_12;

architecture syn of rams_12 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_add1 : std_logic_vector(5 downto 0);
    signal read_add2 : std_logic_vector(5 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;
```

Dual-Port RAM With Synchronous Read (Read Through) and Two Clocks Verilog Coding Example

```
//
// Dual-Port RAM with Synchronous Read (Read Through)
// using More than One Clock
//

module v_rams_12 (clk1, clk2, we, add1, add2, di, do1, do2);

    input  clk1;
    input  clk2;
    input  we;
    input  [5:0] add1;
    input  [5:0] add2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_add1;
    reg    [5:0] read_add2;

    always @(posedge clk1) begin
        if (we)
            ram[add1] <= di;
        read_add1 <= add1;
    end

    assign do1 = ram[read_add1];

    always @(posedge clk2) begin
        read_add2 <= add2;
    end

    assign do2 = ram[read_add2];

endmodule
```

Dual-Port RAM With One Enable Controlling Both Ports Diagram



Dual-Port RAM With One Enable Controlling Both Ports Pin Descriptions

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

Dual-Port RAM With One Enable Controlling Both Ports VHDL Coding Example

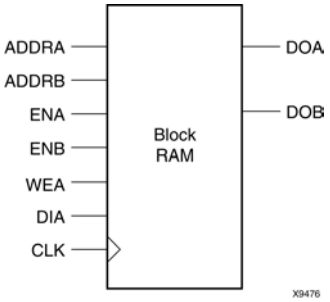
```
--
-- Dual-Port RAM with One Enable Controlling Both Ports
--

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

entity rams_13 is
    port (clk   : in std_logic;
          en    : in std_logic;
          we    : in std_logic;
          addra : in std_logic_vector(5 downto 0);
          addrb : in std_logic_vector(5 downto 0);
          di    : in std_logic_vector(15 downto 0);
          doa   : out std_logic_vector(15 downto 0);
          dob   : out std_logic_vector(15 downto 0));
end rams_13;

architecture syn of rams_13 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 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;
```

Dual-Port RAM With One Enable Controlling Both Ports Verilog Coding Example

```
//
// Dual-Port RAM with One Enable Controlling Both Ports
//

module v_rams_13 (clk, en, we, addra, addrb, di, doa, dob);

    input  clk;
    input  en;
    input  we;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] di;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

    always @(posedge clk) begin
        if (en)
        begin
            if (we)
                ram[addra] <= di;
            read_addra <= addra;
            read_addrb <= addrb;
        end
    end

    assign doa = ram[read_addra];
    assign dob = ram[read_addrb];

endmodule
```

The following descriptions are directly mappable onto block RAM, as shown in the diagram.

Dual Port RAM With Enable on Each Port Diagram



Dual Port RAM With Enable on Each Port Pin Descriptions

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

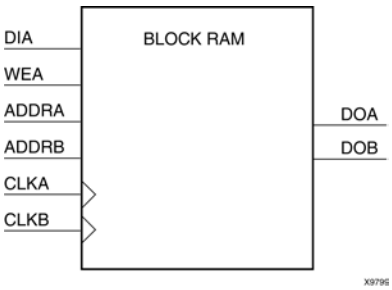
Dual Port RAM With Enable on Each Port VHDL Coding Example

```
--
-- Dual-Port RAM with Enable on Each Port
--

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

entity rams_14 is
    port (clk   : in std_logic;
          ena   : in std_logic;
          enb   : in std_logic;
          wea   : in std_logic;
          addra : in std_logic_vector(5 downto 0);
          addrb : in std_logic_vector(5 downto 0);
          dia   : in std_logic_vector(15 downto 0);
          doa   : out std_logic_vector(15 downto 0);
          dob   : out std_logic_vector(15 downto 0));
end rams_14;

architecture syn of rams_14 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal RAM : ram_type;
    signal read_addra : std_logic_vector(5 downto 0);
    signal read_addrb : std_logic_vector(5 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;

            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;
```

Dual Port RAM With Enable on Each Port Verilog Coding Example

```
//
// Dual-Port RAM with Enable on Each Port
//

module v_rams_14 (clk,ena,enb,wea,addra,addrb,dia,doa,dob);

    input  clk;
    input  ena;
    input  enb;
    input  wea;
    input  [5:0] addra;
    input  [5:0] addrb;
    input  [15:0] dia;
    output [15:0] doa;
    output [15:0] dob;
    reg    [15:0] ram [63:0];
    reg    [5:0] read_addra;
    reg    [5:0] read_addrb;

    always @(posedge clk) begin
        if (ena)
        begin
            if (wea)
                ram[addra] <= dia;
            read_addra <= addra;
        end

        if (enb)
            read_addrb <= addrb;
    end

    assign doa = ram[read_addra];
    assign dob = ram[read_addrb];

endmodule
```

Dual-Port Block RAM With Different Clocks Diagram

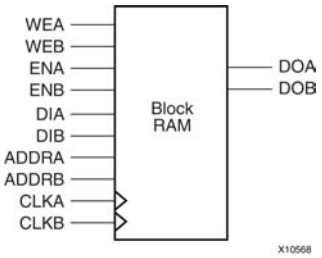


Dual-Port Block RAM With Different Clock Pin Descriptions

IO Pins	Description
clka	Positive-Edge Clock
clkb	Positive-Edge Clock
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

XST supports dual-port block RAMs with two write ports for VHDL and Verilog. The concept of dual-write ports implies not only distinct data ports, but the possibility of distinct write clocks and write enables as well. Distinct write clocks also mean 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. In VHDL, the description of this type of block RAM is based on the usage of shared variables. The XST VHDL analyzer accepts shared variables, but errors out in the HDL Synthesis step if a shared variable does not describe a valid RAM macro.

Dual-Port Block RAM With Two Write Ports Diagram



Dual-Port Block RAM With Two Write Ports Pin Descriptions

IO Pins	Description
clka, clkb	Positive-Edge Clock
ena	Primary Global Enable (Active High)
enb	Dual Global Enable (Active High)
wea, web	Primary Synchronous Write Enable (Active High)
addra	Write Address/Primary Read Address
addrb	Dual Read Address
dia	Primary Data Input
dib	Dual Data Input
doa	Primary Output Port
dob	Dual Output Port

Dual-Port Block RAM With Two Write Ports VHDL Coding Example

This is the most general example. It has different clocks, enables, and write enables.

```
--
-- Dual-Port Block RAM with Two Write Ports
--

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

entity rams_16 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(5 downto 0);
         addrb : in std_logic_vector(5 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 rams_16;

architecture syn of rams_16 is
    type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
    shared variable RAM : ram_type;
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 coding examples recommended for single-write RAMs. The order of appearance of the different lines of code is significant.

Dual-Port Block RAM With Two Write Ports Verilog Coding Example

This is the most general example. It has different clocks, enables, and write enables.

```
//
// Dual-Port Block RAM with Two Write Ports
//

module v_rams_16 (clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

    input  clka,clkb,ena,enb,wea,web;
    input  [5:0]  addra,addrb;
    input  [15:0] dia,dib;
    output [15:0] doa,dob;
    reg    [15:0] ram [63:0];
    reg    [15:0] doa,dob;


    always @(posedge clka) begin
        if (ena)
        begin
            if (wea)
                ram[addra] <= dia;
            doa <= ram[addra];
        end
    end

    always @(posedge clkb) begin
        if (enb)
        begin
            if (web)
                ram[addrb] <= dib;
            dob <= ram[addrb];
        end
    end

endmodule
```

Write-First Synchronization Coding Example One

```
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;
```

Write-First Synchronization Coding Example Two

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 Synchronization Coding Example

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 Synchronization Coding Example

```
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;
```

XST supports single and dual-port block RAM with Byte-wide Write Enable for VHDL and Verilog. The RAM can be seen as a collection of equal size columns. During a write cycle, you separately control writing into each of these columns.

In the Multiple Write Statement method, there is one separate write access statement, including the description of the related write enable, for each column.

The Single Write Statement method allows you to describe only one write access statement. The write enables are described separately outside the main sequential process.

The two methods for describing column-based RAM writes are shown in the following coding examples.

XST currently supports the second solution only (Single Write Statement).

Multiple Write Statement VHDL Coding Example

```
type ram_type is array (SIZE-1 downto 0)
              of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;

(...)

process(clk)
begin
  if posedge(clk) then
      if we(1) = '1' then
        RAM(conv_integer(addr))(2*WIDTH-1 downto WIDTH) <= di(2*WIDTH-1 downto WIDTH);
      end if;
      if we(0) = '1' then
        RAM(conv_integer(addr))(WIDTH-1 downto 0) <= di(WIDTH-1 downto 0);
      end if;

      do <= RAM(conv_integer(addr));
  end if;
end process;
```

Multiple Write Statement Verilog Coding Example

```
reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];

always @(posedge clk)
begin
    if (we[1]) then
        RAM[addr][2*WIDTH-1:WIDTH] <= di[2*WIDTH-1:WIDTH];
    end if;
    if (we[0]) then
        RAM[addr][WIDTH-1:0] <= di[WIDTH-1:0;
    end if;

    do <= RAM[addr];
end
```

Single Write Statement VHDL Coding Example

```
type ram_type is array (SIZE-1 downto 0)
              of std_logic_vector (2*WIDTH-1 downto 0);
signal RAM : ram_type;
signal di0, di1 : std_logic_vector (WIDTH-1 downto 0);

(...)

-- Write enables described outside main sequential process
process(we, di, addr)
begin

  if we(1) = '1' then
    di1 <= di(2*WIDTH-1 downto WIDTH);
  else
    di1 <= RAM(conv_integer(addr))(2*WIDTH-1 downto WIDTH);
  end if;

  if we(0) = '1' then
    di0 <= di(WIDTH-1 downto 0);
  else
    di0 <= RAM(conv_integer(addr))(WIDTH-1 downto 0);
  end if;

end process;

process(clk)
begin
  if posedge(clk) then
    if en = '1' then
      RAM(conv_integer(addr)) <= di1 & di0; -- single write access statement
      do <= RAM(conv_integer(addr));
    end if;
  end if;
end process;
```

Single Write Statement Verilog Coding Example

```
reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
reg    [DI_WIDTH-1:0]   di0, di1;

always @(we or di or addr)
begin
    if (we[1])
        di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
    else
        di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

    if (we[0])
        di0 = di[DI_WIDTH-1:0];
    else
       di0 = RAM[addr][DI_WIDTH-1:0];
end

always @(posedge clk)
begin
   RAM[addr]<={di1,di0};
   do <= RAM[addr];
end
```

To simplify the understanding of byte-wide write enable templates, the following coding examples use single-port block RAMs. XST supports dual-port Block RAM, as well as byte-wide write enable.

Read-First Mode: Single-Port BRAM with Byte-wide Write Enable (2 Bytes) Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Write Enable
addr	Write/Read Address
di	Data Input
do	RAM Output Port

Read-First Mode: Single-Port BRAM With Byte-Wide Write Enable (2 Bytes) VHDL Coding Example

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
--

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

entity rams_24 is
    generic (SIZE       : integer := 512;
             ADDR_WIDTH : integer := 9;
             DI_WIDTH   : integer := 8);

    port (clk  : in  std_logic;
          we   : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_24;

architecture syn of rams_24 is

    type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
    signal RAM : ram_type;

    signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= RAM(conv_integer(addr));
        end if;
    end process;

end syn;
```

Read-First Mode: Single-Port BRAM With Byte-wide Write Enable (2 Bytes) Verilog Coding Example

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
//

module v_rams_24 (clk, we, addr, di, do);

    parameter SIZE       = 512;
    parameter ADDR_WIDTH = 9;
    parameter DI_WIDTH   = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0]   di0, di1;

    always @(we or di)
    begin
        if (we[1])
            di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        else
            di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

        if (we[0])
            di0 = di[DI_WIDTH-1:0];
        else
            di0 = RAM[addr][DI_WIDTH-1:0];

    end

    always @(posedge clk)
    begin
        RAM[addr]<={di1,di0};
        do <= RAM[addr];
    end

endmodule
```

Write-First Mode: Single-Port BRAM with Byte-wide Write Enable (2 Bytes) Pin Descriptions

IO Pins	Description
Clk	Positive-Edge Clock
We	Write Enable
Addr	Write/Read Address
Di	Data Input
Do	RAM Output Port

Write-First Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes) VHDL Coding Example

```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Write-First Mode
--

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

entity rams_25 is
    generic (SIZE       : integer := 512;
             ADDR_WIDTH : integer := 9;
             DI_WIDTH   : integer := 8);

    port (clk  : in  std_logic;
          we   : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_25;

architecture syn of rams_25 is
   type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
   signal RAM : ram_type;

   signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
   signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
            do0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
            do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= do1 & do0;
        end if;
    end process;

end syn;
```

Write-First Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes) Verilog Coding Example

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Write-First Mode
//

module v_rams_25 (clk, we, addr, di, do);

    parameter SIZE       = 512;
    parameter ADDR_WIDTH = 9;
    parameter DI_WIDTH   = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0]   di0, di1;
    reg    [DI_WIDTH-1:0]   do0, do1;

    always @(we or di)
    begin
        if (we[1])
            begin
                di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
                do1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
            end
        else
            begin
                di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
                do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            end

        if (we[0])
            begin
                di0 <= di[DI_WIDTH-1:0];
                do0 <= di[DI_WIDTH-1:0];
            end
        else
            begin
                di0 <= RAM[addr][DI_WIDTH-1:0];
                do0 <= RAM[addr][DI_WIDTH-1:0];
            end

    end

    always @(posedge clk)
    begin
        RAM[addr]<={di1,di0};
        do <= {do1,do0};
    end

endmodule
```

No-Change Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes) Pin Descriptions

IO Pins	Description
Clk	Positive-Edge Clock
We	Write Enable
Addr	Write/Read Address
Di	Data Input
Do	RAM Output Port

XST infers latches for **do1** and **do0** signals during the basic HDL Synthesis. These latches are absorbed by BRAM during the Advanced HDL Synthesis step.

No-Change Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes) VHDL Coding Example

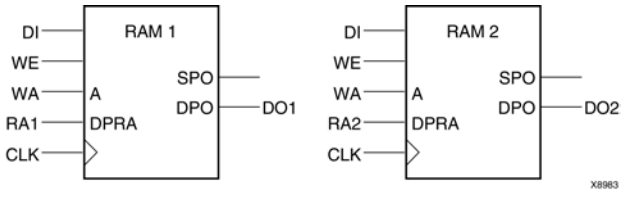
```
--
-- Single-Port BRAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
--

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

entity rams_26 is
    generic (SIZE       : integer := 512;
             ADDR_WIDTH : integer := 9;
             DI_WIDTH   : integer := 8);

    port (clk  : in std_logic;
          we   : in  std_logic_vector(1 downto 0);
          addr : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
          di   : in  std_logic_vector(2*DI_WIDTH-1 downto 0);
          do   : out std_logic_vector(2*DI_WIDTH-1 downto 0));
end rams_26;

architecture syn of rams_26 is
   type ram_type is array (SIZE-1 downto 0) of std_logic_vector (2*DI_WIDTH-1 downto 0);
   signal RAM : ram_type;

   signal di0, di1 : std_logic_vector (DI_WIDTH-1 downto 0);
   signal do0, do1 : std_logic_vector (DI_WIDTH-1 downto 0);
begin

    process(we, di)
    begin
        if we(1) = '1' then
            di1 <= di(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        else
            di1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
            do1 <= RAM(conv_integer(addr))(2*DI_WIDTH-1 downto 1*DI_WIDTH);
        end if;

        if we(0) = '1' then
            di0 <= di(DI_WIDTH-1 downto 0);
        else
            di0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
            do0 <= RAM(conv_integer(addr))(DI_WIDTH-1 downto 0);
        end if;
    end process;

    process(clk)
    begin
        if (clk'event and clk = '1') then
            RAM(conv_integer(addr)) <= di1 & di0;
            do <= do1 & do0;
        end if;
    end process;

end syn;
```

No-Change Mode: Single-Port BRAM with Byte-Wide Write Enable (2 Bytes) in Verilog Coding Example

```
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in No-Change Mode
//

module v_rams_26 (clk, we, addr, di, do);

    parameter SIZE       = 512;
    parameter ADDR_WIDTH = 9;
    parameter DI_WIDTH   = 8;

    input  clk;
    input  [1:0] we;
    input  [ADDR_WIDTH-1:0] addr;
    input  [2*DI_WIDTH-1:0] di;
    output [2*DI_WIDTH-1:0] do;
    reg    [2*DI_WIDTH-1:0] RAM [SIZE-1:0];
    reg    [2*DI_WIDTH-1:0] do;

    reg    [DI_WIDTH-1:0]   di0, di1;
    reg    [DI_WIDTH-1:0]   do0, do1;

    always @(we or di)
    begin
        if (we[1])
            di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
        else
            begin
                di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
                do1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];
            end

        if (we[0])
            di0 <= di[DI_WIDTH-1:0];
        else
            begin
                di0 <= RAM[addr][DI_WIDTH-1:0];
                do0 <= RAM[addr][DI_WIDTH-1:0];
            end

    end

    always @(posedge clk)
    begin
        RAM[addr]<={di1,di0};
        do <= {do1,do0};
    end

endmodule
```

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 in the following figure.

Multiple-Port RAM Descriptions Diagram



Multiple-Port RAM Descriptions Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
wa	Write Address
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

Multiple-Port RAM Descriptions VHDL Coding Example

```
--
-- Multiple-Port RAM Descriptions
--

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

entity rams_17 is
    port (clk : in std_logic;
          we  : in std_logic;
          wa  : in std_logic_vector(5 downto 0);
          ra1 : in std_logic_vector(5 downto 0);
          ra2 : in std_logic_vector(5 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 rams_17;

architecture syn of rams_17 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Multiple-Port RAM Descriptions Verilog Coding Example

```
//
// Multiple-Port RAM Descriptions
//

module v_rams_17 (clk, we, wa, ra1, ra2, di, do1, do2);

    input  clk;
    input  we;
    input  [5:0] wa;
    input  [5:0] ra1;
    input  [5:0] ra2;
    input  [15:0] di;
    output [15:0] do1;
    output [15:0] do2;
    reg    [15:0] ram [63:0];

    always @(posedge clk)
    begin
        if (we)
            ram[wa] <= di;
    end

    assign do1 = ram[ra1];
    assign do2 = ram[ra2];

endmodule
```

XST supports block RAM with reset on the data outputs, as offered with Virtex-4 device, Virtex-5 device, 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 re-settable 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

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 allowed for the primary data output only. The dual output may be used in Read-First Mode only.

Block RAM With Reset Pin Descriptions

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

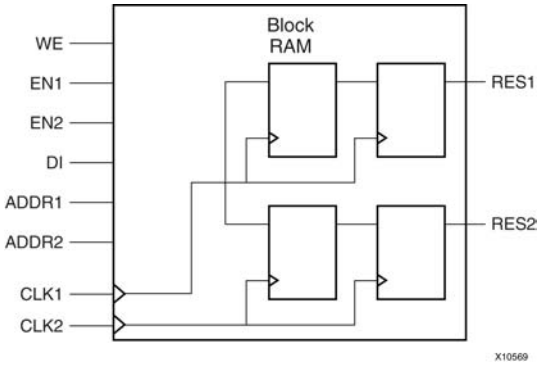
Block RAM With Reset VHDL Coding Example

```
--
-- Block RAM with Reset
--

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

entity rams_18 is
    port (clk  : in std_logic;
          en   : in std_logic;
          we   : in std_logic;
          rst  : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di   : in std_logic_vector(15 downto 0);
          do   : out std_logic_vector(15 downto 0));
end rams_18;

architecture syn of rams_18 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 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;
```

Block RAM With Reset Verilog Coding Example

```
//
// Block RAM with Reset
//

module v_rams_18 (clk, en, we, rst, addr, di, do);

    input  clk;
    input  en;
    input  we;
    input  rst;
    input  [5:0] addr;
    input  [15:0] di;
    output [15:0] do;
    reg    [15:0] ram [63:0];
    reg    [15:0] do;

    always @(posedge clk)
    begin
        if (en) // optional enable
        begin
            if (we) // write enable
                ram[addr] <= di;

            if (rst) // optional reset
                do <= 16'h0000;
            else
                do <= ram[addr];
        end
    end

endmodule
```

Block RAM With Optional Output Registers Diagram



Block RAM With Optional Output Registers Pin Descriptions

IO Pins	Description
clk1, clk2	Positive-Edge Clock
we	Write Enable
en1, en2	Clock Enable (Active High)
addr1	Primary Read Address
addr2	Dual Read Address
di	Data Input
res1	Primary Output Port
res2	Dual Output Port

Block RAM With Optional Output Registers VHDL Coding Example

```
--
-- Block RAM with Optional Output Registers
--

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

entity rams_19 is
    port (clk1, clk2   : in std_logic;
          we, en1, en2 : in std_logic;
          addr1        : in std_logic_vector(5 downto 0);
          addr2        : in std_logic_vector(5 downto 0);
          di           : in std_logic_vector(15 downto 0);
          res1         : out std_logic_vector(15 downto 0);
          res2         : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is
    type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
    signal ram : ram_type;
    signal do1 : std_logic_vector(15 downto 0);
    signal do2 : std_logic_vector(15 downto 0);
begin

    process (clk1)
    begin
        if rising_edge(clk1) then
            if we = '1' then
                ram(conv_integer(addr1)) <= di;
            end if;
            do1 <= ram(conv_integer(addr1));
        end if;
    end process;

    process (clk2)
    begin
        if rising_edge(clk2) then
            do2 <= ram(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;
```

Block RAM With Optional Output Registers Verilog Coding Example

```
//
// Block RAM with Optional Output Registers
//

module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);

    input  clk1;
    input  clk2;
    input  we, en1, en2;
    input  [5:0] addr1;
    input  [5:0] addr2;
    input  [15:0] di;
    output [15:0] res1;
    output [15:0] res2;
    reg    [15:0] res1;
    reg    [15:0] res2;
    reg    [15:0] RAM [63:0];
    reg    [15:0] do1;
    reg    [15:0] 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
```

Initializing RAM Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Block and distributed RAM initial contents can be specified by initialization of the signal describing the memory array in your HDL code. Do this directly in your HDL code, or specify a file containing the initialization data. See:

- Initializing RAM Directly in Hardware Description Language (HDL) Code
- Initializing RAM From an External File

XST supports RAM initialization in both VHDL and Verilog.

Initializing RAM Directly in Hardware Description Language (HDL) Code

The following coding examples show how to initialize RAM directly in Hardware Description Language (HDL) code. For further information, see Initializing RAM from an External File below.

RAM Initial Contents VHDL Coding Example (Hexadecimal)

To specify RAM initial contents, initialize the signal describing the memory array in the VHDL code as shown in the following coding 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));
```

Initializing Block RAM Verilog Coding Example (Hexadecimal)

To specify RAM initial contents, initialize the signal describing the memory array in your Verilog code using initial statements as shown in the following coding example:

```
...
reg [19:0] ram [63:0];
initial begin
  ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
  ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
  ...
 ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
end
...
always @(posedge clk)
begin
 if (we)
  ram[addr] <= di;
 do <= ram[addr];
end
```

RAM Initial Contents VHDL Coding Example (Binary)

RAM initial contents can be specified in hexadecimal, as shown in *RAM Initial Contents VHDL Coding Example (Hexadecimal)*, or in binary as shown in the following coding 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");
```

Initializing Block RAM Verilog Coding Example (Binary)

RAM initial contents can be specified in hexadecimal, as shown in *Initializing Block RAM Verilog Coding Example (Hexadecimal)*, or in binary as shown in the following coding example:

```
...
reg [15:0] ram [63:0];
initial begin
    ram[63] = 16'b0111100100000101;
    ram[62] = 16'b0000010110111101;
    ram[61] = 16'b1100001101010000;
    ...
    ram[0] = 16'b0000100101110011;
end
...
```

Single-Port BRAM Initial Contents VHDL Coding Example

```
--
-- Initializing Block RAM (Single-Port BRAM)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20a is
    port (clk : in std_logic;
          we : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          di : in std_logic_vector(19 downto 0);
          do : out std_logic_vector(19 downto 0));
end rams_20a;

architecture syn of rams_20a is

   type ram_type is array (63 downto 0) 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");

begin

  process (clk)
  begin
    if rising_edge(clk) then
     if we = '1' then
      RAM(conv_integer(addr)) <= di;
     end if;
    do <= RAM(conv_integer(addr));
    end if;
  end process;

end syn;
```

Single-Port BRAM Initial Contents Verilog Coding Example

```
//
// Initializing Block RAM (Single-Port BRAM)
//
module v_rams_20a (clk, we, addr, di, do);
 input clk;
 input we;
 input [5:0] addr;
 input [19:0] di;
 output [19:0] do;
 reg [19:0] ram [63:0];
 reg [19:0] do;

 initial begin
  ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
  ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
  ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
  ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
  ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
  ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
  ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
  ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
  ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
  ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h02021;
  ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;

  ram[30] = 20'h04001; ram[29] = 20'h00342; ram[28] = 20'h0232B;
  ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
  ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
  ram[21] = 20'h02023; ram[20] = 20'h00303; ram[19] = 20'h02433;
  ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
  ram[15] = 20'h00102; ram[14] = 20'h02137; ram[13] = 20'h02036;
  ram[12] = 20'h00301; ram[11] = 20'h00102; ram[10] = 20'h02237;
  ram[9] = 20'h04004; ram[8] = 20'h00304; ram[7] = 20'h04040;
  ram[6] = 20'h02500; ram[5] = 20'h02500; ram[4] = 20'h02500;
  ram[3] = 20'h0030D; ram[2] = 20'h02341; ram[1] = 20'h08201;
  ram[0] = 20'h0400D;
 end

 always @(posedge clk)
 begin
  if (we)
   ram[addr] <= di;
  do <= ram[addr];
 end


endmodule
```

Dual-Port RAM Initial Contents VHDL Coding Example

```
--
-- Initializing Block RAM (Dual-Port BRAM)
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_20b 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 rams_20b;

architecture syn of rams_20b is

 type ram_type is array (255 downto 0) of std_logic_vector (15 downto 0);
 signal RAM : ram_type:= (255 downto 100 => X"B8B8", 99 downto 0 => X"8282");

begin

 process (clk1)
 begin
  if rising_edge(clk1) then
   if we = '1' then
    RAM(conv_integer(addr1)) <= di;
   end if;
   do1 <= RAM(conv_integer(addr1));
  end if;
 end process;

 process (clk2)
 begin
  if rising_edge(clk2) then
   do2 <= RAM(conv_integer(addr2));
  end if;
 end process;

end syn;
```

Dual-Port RAM Initial Contents Verilog Coding Example

```
//
// Initializing Block RAM (Dual-Port BRAM)
//

module v_rams_20b (clk1, clk2, we, addr1, addr2, di, do1, do2);
 input clk1, clk2;
 input we;
 input [7:0] addr1, addr2;
 input [15:0] di;
 output [15:0] do1, do2;

 reg [15:0] ram [255:0];
 reg [15:0] do1, do2;
 integer index;

 initial begin
  for (index = 0 ; index <= 99 ; index = index + 1) begin
   ram[index] = 16'h8282;
  end

  for (index= 100 ; index <= 255 ; index = index + 1) begin
    ram[index] = 16'hB8B8;
  end
 end

 always @(posedge clk1)
 begin
  if (we)
   ram[addr1] <= di;
  do1 <= ram[addr1];
 end

 always @(posedge clk2)
 begin
  do2 <= ram[addr2];
 end

endmodule
```

Initializing RAM From an External File

The following coding examples show how to initialize RAM from an external file. For further information, see Initializing RAM Directly in Hardware Description Language (HDL) Code above.

To initialize RAM from values contained in an external file, use a read function in the VHDL code. For more information, see XST VHDL File Type Support. 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 RAM array.
- Following is an example of the contents of a file initializing an 8 x 32-bit RAM with binary values:

```
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

Initializing Block RAM (External Data File)

RAM initial values may be stored in an external data file that is accessed from within the HDL code. The data file must be pure binary or hexadecimal content with no comments or other information. Following is an example of the contents of a file initializing an 8 x 32-bit RAM with binary values. For both examples, the data file referenced is called rams_20c.data.

```
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
00001111000011110000111100001111
01001010001000001100000010000100
00000000001111100000000001000001
11111101010000011100010000100100
```

Initializing Block RAM (External Data File) VHDL Coding Example

In the following coding example, the loop that generates the initial value is controlled by testing that we are in the RAM address range. The following coding examples show initializing Block RAM from an external data file.

```
--
-- Initializing Block RAM from external data file
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;

entity rams_20c is
 port(clk : in std_logic;
  we : in std_logic;
  addr : in std_logic_vector(5 downto 0);
  din : in std_logic_vector(31 downto 0);
  dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is

  type RamType is array(0 to 63) of bit_vector(31 downto 0);

 impure 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("rams_20c.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 syn;
```

If there are not enough lines in the external data file, XST issues the following message.
`ERROR:Xst - raminitfile1.vhd line 40:  Line <RamFileLine has not enough elements for target <RAM<63>>.`

Initializing Block RAM (External Data File) Verilog Coding Example

To initialize RAM from values contained in an external file, use a **$readmemb** or **$readmemh** system task in your Verilog code. For more information, see XST Behavioral Verilog Language Support. Set up the initialization file as follows.

- Arrange 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.
- Use **$readmemb** for binary and **$readmemh** for hexadecimal representation. To avoid the possible difference between XST and simulator behavior, Xilinx® recommends that you use index parameters in these system tasks. See the following coding example:

  ```
  $readmemb("rams_20c.data",ram, 0, 7);
  ```

Create as many lines in the file as there are rows in the RAM array.

```
//
// Initializing Block RAM from external data file
//

module v_rams_20c (clk, we, addr, din, dout);
 input clk;
 input we;
 input [5:0] addr;
 input [31:0] din;
 output [31:0] dout;

 reg [31:0] ram [0:63];
 reg [31:0] dout;

 initial
 begin
  $readmemb("rams_20c.data",ram, 0, 63);
 end

 always @(posedge clk)
 begin
  if (we)
   ram[addr] <= din;
  dout <= ram[addr];
 end

endmodule
```

ROMs Using Block RAM Resources Hardware Description Language (HDL) Coding Techniques

XST can use block RAM resources to implement ROMs with synchronous outputs or address inputs. These ROMs are implemented as single-port or dual-port block RAMs depending on the HDL description.

XST can infer block ROM across hierarchies if Keep Hierarchy (KEEP_HIERARCHY) is set to **no**. In this case, ROM and the data output or address register can be described in separate hierarchy blocks. This inference is performed during Advanced HDL Synthesis.

Using block RAM resources to implement ROMs is controlled by the ROM Style (ROM_STYLE) constraint. For more information about ROM Style (ROM_STYLE), see XST Design Constraints. For more information about ROM implementation, see XST FPGA Optimization.

ROMs Using Block RAM Resources Log File

```
=========================================================================
*                        HDL Synthesis                                  *
=========================================================================
Synthesizing Unit <rams_21a>.
    Related source file is "rams_21a.vhd".
    Found 64x20-bit ROM for signal <$varindex0000> created at line 38.
    Found 20-bit register for signal <data>.
    Summary:
       inferred   1 ROM(s).
       inferred  20 D-type flip-flop(s).
Unit <rams_21a> synthesized.
=========================================================================
HDL Synthesis Report
Macro Statistics
# ROMs                                                 : 1
 64x20-bit ROM                                         : 1
# Registers                                            : 1
 20-bit register                                       : 1
=========================================================================
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
INFO:Xst - Unit <rams_21a> : The ROM <Mrom__varindex0000> will be implemented
 as a read-only BLOCK RAM, absorbing the register: <data>.
    -----------------------------------------------------------------------
    | ram_type           | Block                             |          |
    -----------------------------------------------------------------------
    | Port A                                                 |          |
    |     aspect ratio   | 64-word x 20-bit (6.9%)           |          |
    |     mode           | write-first                       |          |
    |     clkA           | connected to signal <clk>         | rise     |
    |     enA            | connected to signal <en>          | high     |
    |     weA            | connected to internal node        | high     |
    |     addrA          | connected to signal <addr>        |          |
    |     diA            | connected to internal node        |          |
    |     doA            | connected to signal <data>        |          |
    -----------------------------------------------------------------------
=========================================================================
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                                 : 1
 64x20-bit single-port block RAM                       : 1
=========================================================================
```

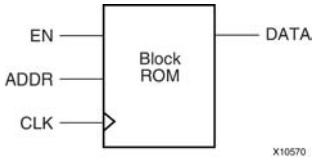
ROMs Using Block RAM Resources Related Constraints

ROM Style (ROM_STYLE)

ROMs Using Block RAM Resources Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

ROM With Registered Output Diagram



ROM With Registered Output Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
en	Synchronous Enable (Active High)
addr	Read Address
data	Data Output

ROM With Registered Output VHDL Coding Example One

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 1)
--

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

entity rams_21a is
    port (clk  : in std_logic;
          en   : in std_logic;
          addr : in std_logic_vector(5 downto 0);
          data : out std_logic_vector(19 downto 0));
end rams_21a;

architecture syn of rams_21a is

    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_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");

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;
```

ROM With Registered Output VHDL Coding Example Two

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered output (template 2)
--

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

entity rams_21b is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_21b;

architecture syn of rams_21b is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_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");

    signal rdata : std_logic_vector(19 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;
```

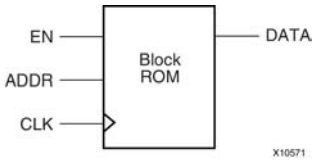
ROM With Registered Output Verilog Coding Example One

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 1)
//


module v_rams_21a (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;

    always @(posedge clk) begin
        if (en)
            case(addr)
                6'b000000: data <= 20'h0200A;   6'b100000: data <= 20'h02222;
                6'b000001: data <= 20'h00300;   6'b100001: data <= 20'h04001;
                6'b000010: data <= 20'h08101;   6'b100010: data <= 20'h00342;
                6'b000011: data <= 20'h04000;   6'b100011: data <= 20'h0232B;
                6'b000100: data <= 20'h08601;   6'b100100: data <= 20'h00900;
                6'b000101: data <= 20'h0233A;   6'b100101: data <= 20'h00302;
                6'b000110: data <= 20'h00300;   6'b100110: data <= 20'h00102;
                6'b000111: data <= 20'h08602;   6'b100111: data <= 20'h04002;
                6'b001000: data <= 20'h02310;   6'b101000: data <= 20'h00900;
                6'b001001: data <= 20'h0203B;   6'b101001: data <= 20'h08201;
                6'b001010: data <= 20'h08300;   6'b101010: data <= 20'h02023;
                6'b001011: data <= 20'h04002;   6'b101011: data <= 20'h00303;
                6'b001100: data <= 20'h08201;   6'b101100: data <= 20'h02433;
                6'b001101: data <= 20'h00500;   6'b101101: data <= 20'h00301;
                6'b001110: data <= 20'h04001;   6'b101110: data <= 20'h04004;
                6'b001111: data <= 20'h02500;   6'b101111: data <= 20'h00301;
                6'b010000: data <= 20'h00340;   6'b110000: data <= 20'h00102;
                6'b010001: data <= 20'h00241;   6'b110001: data <= 20'h02137;
                6'b010010: data <= 20'h04002;   6'b110010: data <= 20'h02036;
                6'b010011: data <= 20'h08300;   6'b110011: data <= 20'h00301;
                6'b010100: data <= 20'h08201;   6'b110100: data <= 20'h00102;
                6'b010101: data <= 20'h00500;   6'b110101: data <= 20'h02237;
                6'b010110: data <= 20'h08101;   6'b110110: data <= 20'h04004;
                6'b010111: data <= 20'h00602;   6'b110111: data <= 20'h00304;
                6'b011000: data <= 20'h04003;   6'b111000: data <= 20'h04040;
                6'b011001: data <= 20'h0241E;   6'b111001: data <= 20'h02500;
                6'b011010: data <= 20'h00301;   6'b111010: data <= 20'h02500;
                6'b011011: data <= 20'h00102;   6'b111011: data <= 20'h02500;
                6'b011100: data <= 20'h02122;   6'b111100: data <= 20'h0030D;
                6'b011101: data <= 20'h02021;   6'b111101: data <= 20'h02341;
                6'b011110: data <= 20'h00301;   6'b111110: data <= 20'h08201;
                6'b011111: data <= 20'h00102;   6'b111111: data <= 20'h0400D;
            endcase
    end

endmodule
```

ROM With Registered Output Verilog Coding Example Two

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 2)
//

module v_rams_21b (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
    reg        [19:0] rdata;


    always @(addr) begin
        case(addr)
                6'b000000: rdata <= 20'h0200A;   6'b100000: rdata <= 20'h02222;
                6'b000001: rdata <= 20'h00300;   6'b100001: rdata <= 20'h04001;
                6'b000010: rdata <= 20'h08101;   6'b100010: rdata <= 20'h00342;
                6'b000011: rdata <= 20'h04000;   6'b100011: rdata <= 20'h0232B;
                6'b000100: rdata <= 20'h08601;   6'b100100: rdata <= 20'h00900;
                6'b000101: rdata <= 20'h0233A;   6'b100101: rdata <= 20'h00302;
                6'b000110: rdata <= 20'h00300;   6'b100110: rdata <= 20'h00102;
                6'b000111: rdata <= 20'h08602;   6'b100111: rdata <= 20'h04002;
                6'b001000: rdata <= 20'h02310;   6'b101000: rdata <= 20'h00900;
                6'b001001: rdata <= 20'h0203B;   6'b101001: rdata <= 20'h08201;
                6'b001010: rdata <= 20'h08300;   6'b101010: rdata <= 20'h02023;
                6'b001011: rdata <= 20'h04002;   6'b101011: rdata <= 20'h00303;
                6'b001100: rdata <= 20'h08201;   6'b101100: rdata <= 20'h02433;
                6'b001101: rdata <= 20'h00500;   6'b101101: rdata <= 20'h00301;
                6'b001110: rdata <= 20'h04001;   6'b101110: rdata <= 20'h04004;
                6'b001111: rdata <= 20'h02500;   6'b101111: rdata <= 20'h00301;
                6'b010000: rdata <= 20'h00340;   6'b110000: rdata <= 20'h00102;
                6'b010001: rdata <= 20'h00241;   6'b110001: rdata <= 20'h02137;
                6'b010010: rdata <= 20'h04002;   6'b110010: rdata <= 20'h02036;
                6'b010011: rdata <= 20'h08300;   6'b110011: rdata <= 20'h00301;
                6'b010100: rdata <= 20'h08201;   6'b110100: rdata <= 20'h00102;
                6'b010101: rdata <= 20'h00500;   6'b110101: rdata <= 20'h02237;
                6'b010110: rdata <= 20'h08101;   6'b110110: rdata <= 20'h04004;
                6'b010111: rdata <= 20'h00602;   6'b110111: rdata <= 20'h00304;
                6'b011000: rdata <= 20'h04003;   6'b111000: rdata <= 20'h04040;
                6'b011001: rdata <= 20'h0241E;   6'b111001: rdata <= 20'h02500;
                6'b011010: rdata <= 20'h00301;   6'b111010: rdata <= 20'h02500;
                6'b011011: rdata <= 20'h00102;   6'b111011: rdata <= 20'h02500;
                6'b011100: rdata <= 20'h02122;   6'b111100: rdata <= 20'h0030D;
                6'b011101: rdata <= 20'h02021;   6'b111101: rdata <= 20'h02341;
                6'b011110: rdata <= 20'h00301;   6'b111110: rdata <= 20'h08201;
                6'b011111: rdata <= 20'h00102;   6'b111111: rdata <= 20'h0400D;
        endcase
    end

    always @(posedge clk) begin
        if (en)
            data <= rdata;
    end
endmodule
```

ROM With Registered Address Diagram



ROM With Registered Address Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
en	Synchronous Enable (Active High)
addr	Read Address
data	Data Output
clk	Positive-Edge Clock

ROM With Registered Address VHDL Coding Example

```
--
-- ROMs Using Block RAM Resources.
-- VHDL code for a ROM with registered address
--

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

entity rams_21c is
port (clk : in std_logic;
      en : in std_logic;
      addr : in std_logic_vector(5 downto 0);
      data : out std_logic_vector(19 downto 0));
end rams_21c;

architecture syn of rams_21c is
    type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
    signal ROM : rom_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");

    signal raddr : std_logic_vector(5 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;
```

ROM With Registered Address Verilog Coding Example

```
//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered address
//

module v_rams_21c (clk, en, addr, data);

    input      clk;
    input      en;
    input      [5:0] addr;
    output reg [19:0] data;
    reg        [5:0] raddr;

    always @(posedge clk) begin
        if (en)
            raddr <= addr;
    end

    always @(raddr) begin
       case(raddr)
                6'b000000: data <= 20'h0200A;   6'b100000: data <= 20'h02222;
                6'b000001: data <= 20'h00300;   6'b100001: data <= 20'h04001;
                6'b000010: data <= 20'h08101;   6'b100010: data <= 20'h00342;
                6'b000011: data <= 20'h04000;   6'b100011: data <= 20'h0232B;
                6'b000100: data <= 20'h08601;   6'b100100: data <= 20'h00900;
                6'b000101: data <= 20'h0233A;   6'b100101: data <= 20'h00302;
                6'b000110: data <= 20'h00300;   6'b100110: data <= 20'h00102;
                6'b000111: data <= 20'h08602;   6'b100111: data <= 20'h04002;
                6'b001000: data <= 20'h02310;   6'b101000: data <= 20'h00900;
                6'b001001: data <= 20'h0203B;   6'b101001: data <= 20'h08201;
                6'b001010: data <= 20'h08300;   6'b101010: data <= 20'h02023;
                6'b001011: data <= 20'h04002;   6'b101011: data <= 20'h00303;
                6'b001100: data <= 20'h08201;   6'b101100: data <= 20'h02433;
                6'b001101: data <= 20'h00500;   6'b101101: data <= 20'h00301;
                6'b001110: data <= 20'h04001;   6'b101110: data <= 20'h04004;
                6'b001111: data <= 20'h02500;   6'b101111: data <= 20'h00301;
                6'b010000: data <= 20'h00340;   6'b110000: data <= 20'h00102;
                6'b010001: data <= 20'h00241;   6'b110001: data <= 20'h02137;
                6'b010010: data <= 20'h04002;   6'b110010: data <= 20'h02036;
                6'b010011: data <= 20'h08300;   6'b110011: data <= 20'h00301;
                6'b010100: data <= 20'h08201;   6'b110100: data <= 20'h00102;
                6'b010101: data <= 20'h00500;   6'b110101: data <= 20'h02237;
                6'b010110: data <= 20'h08101;   6'b110110: data <= 20'h04004;
                6'b010111: data <= 20'h00602;   6'b110111: data <= 20'h00304;
                6'b011000: data <= 20'h04003;   6'b111000: data <= 20'h04040;
                6'b011001: data <= 20'h0241E;   6'b111001: data <= 20'h02500;
                6'b011010: data <= 20'h00301;   6'b111010: data <= 20'h02500;
                6'b011011: data <= 20'h00102;   6'b111011: data <= 20'h02500;
                6'b011100: data <= 20'h02122;   6'b111100: data <= 20'h0030D;
                6'b011101: data <= 20'h02021;   6'b111101: data <= 20'h02341;
                6'b011110: data <= 20'h00301;   6'b111110: data <= 20'h08201;
                6'b011111: data <= 20'h00102;   6'b111111: data <= 20'h0400D;
        endcase
    end

endmodule
```

Pipelined Distributed RAM Hardware Description Language (HDL) Coding Techniques

In order to increase the speed of designs, XST can infer pipelined distributed RAM. By interspersing registers between the stages of distributed RAM, pipelining can significantly increase the overall frequency of your design. The effect of pipelining is similar to Flip-Flop Retiming.

To insert pipeline stages:

1. Describe the necessary registers in your Hardware Description Language (HDL) code
2. Place them after any distributed RAM
3. Set the RAM Style (RAM_STYLE) constraint to **`pipe_distributed`**

In order to reach the maximum distributed RAM speed, XST uses the maximum number of available registers when:

- It detects valid registers for pipelining, and
- RAM _STYLE is set to **`pipe_distributed`**

In order to obtain the best frequency, XST automatically calculates the maximum number of registers for each RAM.

During the Advanced HDL Synthesis step, the XST HDL Advisor advises you to specify the optimum number of register stages if:

- You have not specified sufficient register stages, and
- RAM _STYLE is coded directly on a signal

XST implements the unused stages as shift registers if:

- The number of registers placed after the multiplier exceeds the maximum required, and
- Shift register extraction is activated

XST cannot pipeline RAM if registers contain asynchronous set/reset signals. XST can pipeline RAM if registers contain synchronous reset signals.

Pipelined Distributed RAM Log File

Following is the log file for Pipelined Distributed RAM.

```
=========================================================================
*                           HDL Synthesis                               *
=========================================================================
Synthesizing Unit <rams_22>.
    Related source file is "rams_22.vhd".
    Found 64x4-bit single-port RAM for signal <RAM>.
    Found 4-bit register for signal <do>.
    Summary:
 inferred   1 RAM(s).
 inferred   4 D-type flip-flop(s).
Unit <rams_22> synthesized.

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

Macro Statistics
# RAMs                                                 : 1
 64x4-bit single-port RAM                              : 1
# Registers                                            : 1
 4-bit register                                        : 1

=========================================================================
=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================
INFO:Xst - Unit <rams_22> : The RAM <Mram_RAM> will be implemented as a
distributed RAM, absorbing the following register(s): <do>.

    -----------------------------------------------------------
    | aspect ratio       | 64-word x 4-bit              |       |
    | clock              | connected to signal <clk>    | rise  |
    | write enable       | connected to signal <we>     | high  |
    | address            | connected to signal <addr>   |       |
    | data in            | connected to signal <di>     |       |
    | data out           | connected to internal node   |       |
    | ram_style          | distributed                  |       |
    -----------------------------------------------------------

Synthesizing (advanced) Unit <rams_22>.
 Found pipelined ram on signal <_varindex0000>:
  - 1 pipeline level(s) found in a register on signal <_varindex0000>.
  Pushing register(s) into the ram macro.

INFO:Xst:2390 - HDL ADVISOR - You can improve the performance of
the ram Mram_RAM by adding 1 register level(s) on output signal _varindex0000.
Unit <rams_22> synthesized (advanced).
=========================================================================
Advanced HDL Synthesis Report
Macro Statistics
# RAMs                                                 : 1
 64x4-bit registered single-port distributed RAM       : 1

=========================================================================
```

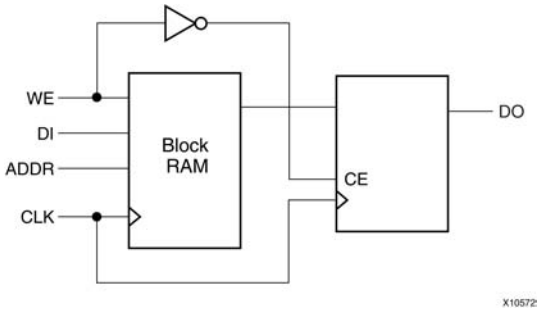
Pipelined Distributed RAM Related Constraints

- RAM Extraction (RAM_EXTRACT)
- RAM Style (RAM_STYLE)
- ROM Extraction (ROM_EXTRACT)
- ROM Style (ROM_STYLE)
- BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)
- Automatic BRAM Packing (AUTO_BRAM_PACKING)

Pipelined Distributed RAM Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Pipelined Distributed RAM Diagram



Pipelined Distributed RAM Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (Active High)
addr	Read/Write Address
di	Data Input
do	Data Output

Pipelined Distributed RAM VHDL Coding Example

```
--
-- Pipeline distributed RAMs
--

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

entity rams_22 is
    port (clk  : in std_logic;
          we   : in std_logic;
          addr : in std_logic_vector(8 downto 0);
          di   : in std_logic_vector(3 downto 0);
          do   : out std_logic_vector(3 downto 0));
end rams_22;

architecture syn of rams_22 is
    type ram_type is array (511 downto 0) of std_logic_vector (3 downto 0);
    signal RAM : ram_type;

    signal pipe_reg: std_logic_vector(3 downto 0);

    attribute ram_style: string;
    attribute ram_style of RAM: signal is "pipe_distributed";
begin

    process (clk)
    begin
        if clk'event and clk = '1' then
                if we = '1' then
                    RAM(conv_integer(addr)) <= di;
                else
                    pipe_reg <= RAM( conv_integer(addr));
                end if;
        do <= pipe_reg;
        end if;
    end process;

end syn;
```

Pipelined Distributed RAM Verilog Coding Example

```
//
// Pipeline distributed RAMs
//


module v_rams_22 (clk, we, addr, di, do);

    input  clk;
    input  we;
    input  [8:0] addr;
    input  [3:0] di;
    output [3:0] do;
    (*ram_style="pipe_distributed"*)
  reg    [3:0] RAM [511:0];
    reg    [3:0] do;
  reg    [3:0] pipe_reg;

    always @(posedge clk)
    begin
        if (we)
          RAM[addr] <= di;
        else
          pipe_reg <= RAM[addr];
    do <= pipe_reg;
    end

endmodule
```

Finite State Machine (FSM) Hardware Description Language (HDL) Coding Techniques

Xilinx Synthesis Technology (XST):

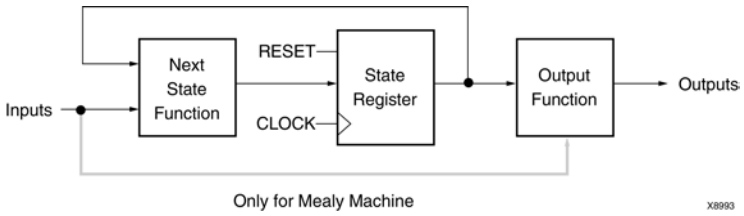
- Includes a large set of templates to describe Finite State Machine (FSM) components
- Can apply several state encoding techniques to obtain better performance or less area
- Can re-encode your initial encoding
- Can handle only synchronous state machines

To disable FSM extraction, use Automatic FSM Extraction (FSM_EXTRACT).

Describing a Finite State Machine (FSM) Component

There are many ways to describe a Finite State Machine (FSM) component. A traditional FSM representation incorporates Mealy and Moore machines, as shown in the following diagram. XST supports both models.

Finite State Machine (FSM) Representation Incorporating Mealy and Moore Machines Diagram



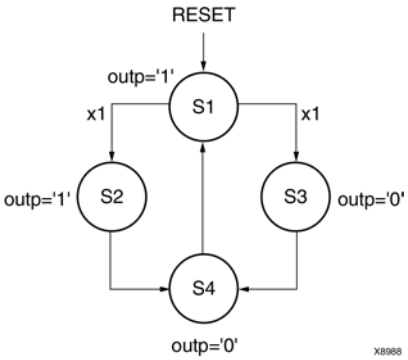
For HDL, **`process`** (VHDL) and **`always`** blocks (Verilog) are the best ways to describe FSM components. Xilinx® uses **`process`** to refer to both VHDL processes and Verilog **`always`** blocks.

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.

Bubble Diagram



State Registers

State registers must be initialized with an asynchronous or synchronous signal, or have the power-up value defined by Register Power Up (REGISTER_POWERUP) Otherwise, XST does not recognize FSM. See Registers HDL Coding Techniques for coding examples on how to write Asynchronous and Synchronous initialization signals.

In VHDL, the type of a state register can be a different type, such as:

- **`integer`**
- **`bit_vector`**
- **`std_logic_vector`**

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 as follows:

```
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 coding example 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.

Finite State Machine (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.

Finite State Machine (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 State Encoding
- One-Hot State Encoding
- Gray State Encoding
- Compact State Encoding
- Johnson State Encoding
- Sequential State Encoding
- Speed1 State Encoding
- User State Encoding

Auto State Encoding

In Auto State Encoding, XST tries to select the best suited encoding algorithm for each FSM.

One-Hot State Encoding

One-Hot State 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 State Encoding is 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 State Encoding

Gray State 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 State Encoding

Compact State Encoding consists of minimizing the number of bits in the state variables and flip-flops. This technique is based on hypercube immersion. Compact State Encoding is appropriate when trying to optimize area.

Johnson State Encoding

Like Gray State Encoding, Johnson State Encoding shows benefits with state machines containing long paths with no branching.

Sequential State Encoding

Sequential State Encoding consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.

Speed1 State Encoding

Speed1 State 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 State Encoding

In User State Encoding, XST uses the original encoding specified in the HDL file. For example, if you use enumerated types for a state register, use the Enumerated Encoding (ENUM_ENCODING) constraint to assign a specific binary value to each state. For more information, see XST Design Constraints.

RAM-Based Finite State Machine (FSM) Synthesis

Large Finite State Machine (FSM) components can be made more compact and faster by implementing them in the block RAM resources provided in Virtex® devices and later technologies. FSM Style (FSM_STYLE) directs XST to use block RAM resources for FSMs.

Values for FSM Style (FSM_STYLE) are:

- **`lut`** (default)

  XST maps the FSM using LUTs.

- **`bram`**

  XST maps the FSM onto block RAM.

Invoke FSM Style (FSM_STYLE) as follows:

- ISE® Design Suite

  Select **LUT** or **Block RAM** as instructed in the *HDL Options* topics of the ISE Design Suite Help.

- Command line

  Use the **-fsm_style** command line option.

- Hardware Description Language (HDL) code

  Use FSM Style (FSM_STYLE)

If it cannot implement a state machine on block RAM, XST:

- Issues a warning in the Advanced HDL Synthesis step of the log file.
- Automatically implements the state machine using LUTs.

For example, if FSM has an asynchronous reset, it cannot be implemented using block RAM. In this case XST informs you:

```
...
=========================================================================
*                     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 Finite State Machine (FSM) Implementation

XST can add logic to your Finite State Machine (FSM) implementation that will let your state machine recover from an invalid state. If during its execution, a state machine enters 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:

- In ISE® Design Suite, select Safe Implementation as instructed in the *HDL Options* topic of ISE Design Suite Help, or
- Apply the Safe Implementation (SAFE_IMPLEMENTATION) constraint to the hierarchical block or signal that represents the state register.

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. To manually define the recovery state, apply the Safe Recovery State (SAFE_RECOVERY_STATE) constraint.

Finite State Machine (FSM) Log File

The XST log file reports the full information of recognized Finite State Machine (FSM) components 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. If the target is an FPGA device, XST reports this encoding at the HDL Synthesis step. If the target is a CPLD device, then XST reports this encoding at the Low Level Optimization step.

```
...
Synthesizing Unit <fsm_1>.
    Related source file is "/state_machines_1.vhd".
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------
    | States             | 4                            |
    | Transitions        | 5                            |
    | Inputs             | 1                            |
    | Outputs            | 4                            |
    | Clock              | clk (rising_edge)            |
    | Reset              | reset (positive)             |
    | Reset type         | asynchronous                 |
    | Reset State        | s1                           |
    | Power Up State     | s1                           |
    | Encoding           | automatic                    |
    | Implementation     | LUT                          |
    -----------------------------------------------------
    Found 1-bit register for signal <outp>.
    Summary:
        inferred   1 Finite State Machine(s).
        inferred   1 D-type flip-flop(s).
Unit <fsm_1> synthesized.


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

Macro Statistics
# Registers                        : 1
 1-bit register                    : 1

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

Advanced Registered AddSub inference ...
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state/FSM_0> on signal <state[1:2]> with gray encoding.
-------------------
 State | Encoding
-------------------
 s1    | 00
 s2    | 01
 s3    | 11
 s4    | 10
-------------------
=========================================================
HDL Synthesis Report

Macro Statistics
# FSMs                             : 1
=========================================================
```

Finite State Machine (FSM) Related Constraints

- Automatic Finite State Machine (FSM) Extraction (FSM_EXTRACT)
- Finite State Machine (FSM) Style (FSM_STYLE)
- Finite State Machine (FSM) Encoding Algorithm (FSM_ENCODING)
- Enumerated Encoding (ENUM_ENCODING)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Safe Recovery State (SAFE_RECOVERY_STATE)

Finite State Machine (FSM) Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Finite State Machine (FSM) With One Process Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
reset	Asynchronous Reset (Active High)
x1	FSM Input
outp	FSM Output

Finite State Machine (FSM) With One Process VHDL Coding Example

```
--
-- State Machine with a single process.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_1 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_1 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;
```

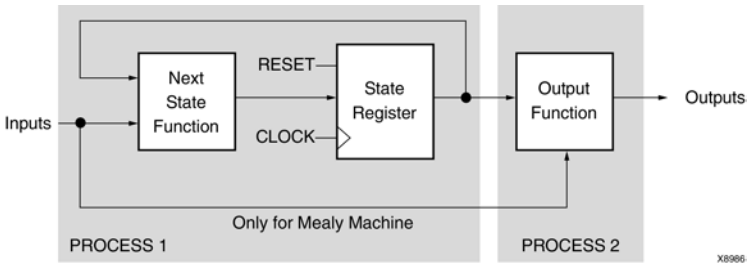
Finite State Machine (FSM) With Single Always Block Verilog Coding Example

```
//
// State Machine with a single always block.
//

module v_fsm_1 (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;

    initial begin
        state = 2'b00;
    end

    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'b0;
                                end
                        end
                    s2: begin
                            state <= s4; outp <= 1'b1;
                        end
                    s3: begin
                            state <= s4; outp <= 1'b0;
                        end
                    s4: begin
                            state <= s1; outp <= 1'b0;
                        end
                endcase
            end
    end

endmodule
```

Finite State Machine (FSM) With Two Processes

To eliminate a register from the **`outputs`**, remove all assignments **`outp <=…`** from the Clock synchronization section. This can be done by introducing two processes as shown below.

Finite State Machine (FSM) With Two Processes Diagram



Finite State Machine (FSM) With Two Processes Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
reset	Asynchronous Reset (Active High)
x1	FSM Input
outp	FSM Output

Finite State Machine (FSM) With Two Processes VHDL Coding Example

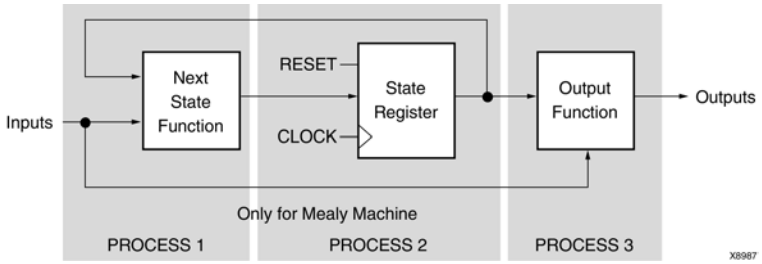
```
--
-- State Machine with two processes.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_2 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_2 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;
```

Finite State Machine (FSM) With Two Always Blocks Verilog Coding Example

```
//
// State Machine with two always blocks.
//

module v_fsm_2 (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;

    initial begin
        state = 2'b00;
    end

    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
```

You can also separate the NEXT State function from the state register.

Finite State Machine (FSM) With Three Processes Diagram



Finite State Machine (FSM) With Three Processes Pin Descriptions

IO Pins	Description
clk	Positive-Edge Clock
reset	Asynchronous Reset (Active High)
x1	FSM Input
outp	FSM Output

Finite State Machine (FSM) With Three Processes VHDL Coding Example

```
--
-- State Machine with three processes.
--

library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_3 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;
```

Finite State Machine (FSM) With Three Always Blocks Verilog Coding Example

```
//
// State Machine with three always blocks.
//

module v_fsm_3 (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;

    initial begin
        state = 2'b00;
    end

    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
                    next_state = s3;
            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
```

Black Boxes Hardware Description Language (HDL) Coding Techniques

Your design may contain Electronic Data Interchange Format (EDIF) or NGC files generated by

- Synthesis tools
- Schematic text editors
- Any other design entry mechanism

These modules must be instantiated in your code in order to be connected to the rest of your design. To do so in XST, use Black Box instantiation in the VHDL or 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 Register Transfer Level (RTL) model, as well as your own implementation of this block in the form of an EDIF netlist. The RTL model is valid for simulation purposes only. Use the BoxType (BOX_TYPE) constraint to 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. For more information, see XST General Constraints and the *Constraints Guide*.

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.

Black Box Log File

Since XST recognizes Black Boxes before macro inference, the Black Box log file differs from the log files 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
....
```

Black Box Related Constraints

BoxType (BOX_TYPE)

BoxType was introduced for device primitive instantiation in XST. See Device Primitive Support before using BoxType.

Black Box Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Black Box VHDL Coding Example

```
--
-- Black Box
--

library ieee;
use ieee.std_logic_1164.all;

entity black_box_1 is
    port(DI_1, DI_2 : in std_logic;
         DOUT : out std_logic);
end black_box_1;

architecture archi of black_box_1 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;
```

Black Box Verilog Coding Example

```
//
// Black Box
//

module v_my_block (in1, in2, dout);
    input in1, in2;
    output dout;
endmodule

module v_black_box_1 (DI_1, DI_2, DOUT);
    input DI_1, DI_2;
    output DOUT;

    v_my_block inst (
        .in1(DI_1),
        .in2(DI_2),
        .dout(DOUT));

endmodule
```

For more information on component instantiation, see your VHDL and Verilog language reference manuals.

Chapter 4

XST FPGA Optimization

This chapter:

- Explains how constraints can be used to optimize FPGA devices
- Explains macro generation
- Describes the FPGA primitive support

This chapter includes:

XST performs the following steps during FPGA synthesis and optimization:

- Mapping and optimization on an entity by entity or module by module basis
- Global optimization on the complete design

The output is an NGC file.

This section describes:

- The constraints that can be applied to fine-tune synthesis and optimization
- Macro generation
- The log file
- The timing models used during synthesis and optimization
- The constraints available for timing-driven synthesis
- The generated NGC file
- Support for primitives

FPGA Specific Synthesis Options

XST supports options to fine-tune synthesis in accordance with user constraints. For information about each option, see XST FPGA Constraints (Non-Timing).

The following options relate to the FPGA specific optimization of synthesis:

- Extract BUFGCE (BUFGCE)
- Cores Search Directories (-sd)
- Decoder Extraction (DECODER_EXTRACT)
- FSM Style (FSM_STYLE)
- Global Optimization Goal (-glob_opt)
- Keep Hierarchy (KEEP_HIERARCHY)
- Logical Shifter Extraction (SHIFT_EXTRACT)
- Map Logic on BRAM (BRAM_MAP)
- Max Fanout (MAX_FANOUT)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)
- Multiplier Style (MULT_STYLE)
- Mux Style (MUX_STYLE)
- Number of Global Clock Buffers (-bufg)
- Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)
- Pack I/O Registers Into IOBs (IOB)
- Priority Encoder Extraction (PRIORITY_EXTRACT)
- RAM Style (RAM_STYLE)
- Register Balancing (REGISTER_BALANCING)
- Register Duplication (REGISTER_DUPLICATION)
- Signal Encoding (SIGNAL_ENCODING)
- Slice Packing (-slice_packing)
- Use Carry Chain (USE_CARRY_CHAIN)
- Write Timing Constraints (-write_timing_constraints)
- XOR Collapsing (XOR_COLLAPSE)

Macro Generation

The FPGA Device Macro Generator module provides the XST HDL Flow with a catalog of functions. These functions are identified by the inference engine from the Hardware Description Language (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 the selected Virtex® architecture or Spartan® architecture, 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 discussion 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 XST 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 (USE_CARRY_CHAIN) constraint to deactivate this feature.

Arithmetic Functions in Macro Generation

For Arithmetic Functions, XST provides the following elements:

- Adders, Subtractors and Adder/Subtractors
- 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 in Macro Generation

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 dedicated carry-ANDs to improve performance.

Multiplexers in Macro Generation

For multiplexers, the Macro Generator provides the following two architectures:

- MUXFx based multiplexers
- Dedicated Carry-MUXs based multiplexers

For 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.

You still may have MUXFx elements in the final netlist even if multiplexer inference is disabled using the MUX_EXTRACT constraint. These elements come from the general mapping procedure of Boolean equations.

Priority Encoders in Macro Generation

The **`if/elsif`** structure described in Priority Encoders Hardware Description Language (HDL) Coding Techniques 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.

Use the Priority Encoder Extraction (PRIORITY_EXTRACT) constraint to enable or disable priority encoder inference.

XST does not generally, infer, and so does not generate, a large number of priority encoders. To enable priority encoders, use the Priority Encoder Extraction (PRIORITY_EXTRACT) constraint with the *force* option.

Decoders in Macro Generation

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.

Use the Decoder Extraction (DECODER_EXTRACT) constraint to enable or disable decoder inference.

Shift Registers in Macro Generation

XST builds two types of shift registers:

- 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.

Use the Shift Register Extraction (SHREG_EXTRACT) constraint to enable and disable shift register inference.

RAMs in Macro Generation

Two types of RAM are available during inference and generation:

- Distributed RAM

  If the RAM is asynchronous READ, Distributed RAM is inferred and generated.
- Block RAM

  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**.

Primitives Used by XST

This section applies to the following devices:

- Virtex®-4
- Spartan®-3

For these devices, XST uses the primitives shown in the following table.

Primitives Used by XST

RAM	Clock Edge	Primitives
Single-Port Synchronous Distributed RAM	Distributed Single-Port RAM with positive clock edge	RAM16X1S, RAM16X2S, RAM16X4S, RAM16X8S, RAM32X1S, RAM32X2S, RAM32X4S, RAM32X8S, RAM64X1S, RAM64X2S, RAM128X1S
Single-Port Synchronous Distributed RAM	Distributed Single-Port RAM with negative clock edge	RAM16X1S_1, RAM32X1S_1, RAM64X1S_1, RAM128X1S_1
Dual-Port Synchronous Distributed RAM	Distributed Dual-Port RAM with positive clock edge	RAM16X1D, RAM32X1D, RAM64X1D
Dual-Port Synchronous Distributed RAM	Distributed Dual-Port RAM with negative clock edge	RAM16X1D_1, RAM32X1D_1, RAM64X1D_1
Single-Port Synchronous Block RAM	N/A	RAMB4_Sn
Dual-Port Synchronous Block RAM	N/A	RAMB4_Sm_Sn

Controlling Implementation of Inferred RAM

To better control 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 (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*

Apply the RAM Style (RAM_STYLE) attribute to:

- A signal that defines the RAM, or
- The instance name of the RAM

The RAM Style (RAM_STYLE) attribute can also be global.

If the RAM resources are limited, XST can generate additional RAMs using registers. To generate additional RAMs using registers, use RAM Extraction (RAM_EXTRACT) with the value set to no.

ROMs in Macro Generation

A ROM can be inferred when all assigned contexts in a **`Case`** or **`If...else`** statement are constants. Macro inference considers only ROMs of at least 16 words with no width restriction. For example, the following Hardware Description Language (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 shown in the following coding 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));
```

ROM Extraction (ROM_EXTRACT) can be used to disable the inference of ROMs. Set the value to *yes* to enable ROM inference. Set the value to *no* to disable ROM inference.

The default is **`yes`**.

Two types of ROM are available during inference and generation:

- Distributed 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 ROM

  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.

ROM Style (ROM_STYLE) specifies which type of synchronous ROM XST infers:

- 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 RAM Style (RAM_STYLE) as a VHDL attribute or a Verilog meta comment to an individual signal, or to the entity or module of the ROM. RAM Style (RAM_STYLE) can also be applied globally from ISE® Design Suite in **Process > Properties**, or from the command line.

DSP48 Block Resources

XST can automatically implement the following macros on a DSP48 block:

- Adders/subtractors
- Accumulators
- Multipliers
- Multiply adder/subtractors
- Multiply accumulate (MAC)

XST also supports the registered versions of these macros.

Macro implementation on DSP48 blocks is controlled by the Use DSP48 (USE_DSP48) constraint or 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 (USE_DSP48) constraint or command line option value to **`yes`**.

XST performs automatic resource control in auto mode for all macros. Use the DSP Utilization Ratio (DSP_UTILIZATION_RATIO) constraint in this mode to control available DSP48 resources for the synthesis. By default, XST tries to utilize all available DSP48 resources as much as possible.

If the number of user-specified DSP slices exceeds the number of available DSP resources on the target FPGA device, XST issues a warning, and uses only available DSP resources on the chip for synthesis. Disable automatic DSP resource management to see the number of DSPs that XST can potentially infer for a specific design. To disable automatic DSP resource management, set *value* = **`-1`**.

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. Use Keep (KEEP) to shape a macro in a specific way. For example, if your design has a multiplier with two register levels on each input, place Keep (KEEP) constraints on the outputs of these registers to exclude the first register stage from the DSP48.

DSP48 blocks do not support registers with Asynchronous Set/Reset signals. Since such registers cannot be absorbed by DSP48, this may lead to sub-optimal performance. The Asynchronous to Synchronous (ASYNC_TO_SYNC) constraint allows you to replace Asynchronous Set/Reset signals with Synchronous signals throughout the entire design. This allows absorption of registers by DSP48, thereby improving quality of results.

Replacing Asynchronous Set/Reset signals by Synchronous signals makes the generated NGC netlist NOT equivalent to the initial RTL description. You must ensure that the synthesized design satisfies the initial specification. For more information, see Asynchronous to Synchronous (ASYNC_TO_SYNC).

For more information on individual macro processing, see XST Hardware Description Language (HDL) Coding Techniques.

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.

XST can build complex DSP macros and DSP48 chains across the hierarchy when the Keep Hierarchy (KEEP_HIERARCHY) command line option is set to **no**. This is the default in ISE® Design Suite.

Mapping Logic Onto Block RAM

If your design does not fit into the target device, you can place some of the design logic into unused block RAM:

1. Put the part of the RTL description to be placed into block RAM in a separate hierarchical block.
2. Attach a BRAM_MAP (Map Logic on BRAM) constraint to the separate hierarchical block, either directly in Hardware Description Language (HDL) code, or in the XST Constraint File (XCF).

XST cannot automatically decide which logic can be placed in block RAM.

Logic placed into a separate block must satisfy the following criteria:

- All outputs are registered.
- The block contains only one level of registers, which are output registers.
- All output registers have the same control signals.
- The output registers have a Synchronous Reset signal.
- The block does not contain multisources or tristate busses.
- The Keep (KEEP) constraint 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 issues a warning. If the logic cannot be placed in a single block RAM primitive, XST spreads it over several block RAMs.

Mapping Logic Onto Block RAM Log Files

This section contains examples of Mapping Logic Onto Block RAM Log Files:

- Mapping Logic Onto Block RAM Log File Example One
- Mapping Logic Onto Block RAM Log File Example Two

Mapping Logic Onto Block RAM Log File Example One

```
...
=========================================================================
*                        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 29.
    Summary:
        inferred   4 D-type flip-flop(s).
        inferred   1 Adder/Subtractor(s).
Unit <logic_bram_1> synthesized.

=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
...
Entity <logic_bram_1> mapped on BRAM.
...
=========================================================================
HDL Synthesis Report

Macro Statistics
# Block RAMs                       : 1
 256x4-bit single-port block RAM   : 1

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

Mapping Logic Onto Block RAM Log File Example Two

```
...
=========================================================================
*                      Advanced HDL Synthesis                           *
=========================================================================
...
INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.
                Output FF <RES> must have a synchronous reset.
```

Mapping Logic Onto Block RAM Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

8-Bit Adders With Constant in a Single Block Ram Primitive VHDL Coding Example

```
--
-- The following example places 8-bit adders with
-- constant in a single block RAM primitive
--

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;
```

8-Bit Adders With Constant in a Single Block Ram Primitive Verilog Coding Example

```
//
// The following example places 8-bit adders with
// constant in a single block RAM primitive
//

(* bram_map="yes" *)
module v_logic_bram_1 (clk, rst, A, B, RES);

    input  clk, rst;
    input  [3:0] A, B;
    output [3:0] RES;
    reg    [3:0] RES;

    always @(posedge clk)
    begin
        if (rst)
            RES <= 4'b0000;
        else
            RES <= A + B + 8'b0001;
    end

endmodule
```

Asynchronous Reset VHDL Coding Example

```
--
-- In the following example, an asynchronous reset is used and
-- so, the logic is not mapped onto block RAM
--

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

entity logic_bram_2 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_2 : entity is "yes";

end logic_bram_2;

architecture beh of logic_bram_2 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;
```

Asynchronous Reset Verilog Coding Example

```
//
// In the following example, an asynchronous reset is used and
// so, the logic is not mapped onto block RAM
//

(* bram_map="yes" *)
module v_logic_bram_2 (clk, rst, A, B, RES);

    input  clk, rst;
    input  [3:0] A, B;
    output [3:0] RES;
    reg    [3:0] RES;

    always @(posedge clk or posedge rst)
    begin
        if (rst)
            RES <= 4'b0000;
        else
            RES <= A + B + 8'b0001;
    end

endmodule
```

Flip-Flop Retiming

Flip-flop retiming 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
- 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. It respects the same constraints as all other optimization techniques. Since retiming is iterative, 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 occurs 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
- Whether it is a forward or backward retiming

Limitations of Flip-Flop Retiming

Flip-flop retiming has 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 (KEEP) property.
- Flip-flops are not moved backward if the input signal has the Keep (KEEP) property.
- Instantiated flip-flops are moved only if the Optimize Instantiated Primitives constraint or command line option is set to **yes**.
- Flip-Flops are moved across instantiated primitives only if the Optimize Instantiated Primitives command line option or constraint is set to **yes**.
- Flip-flops with both a set and a reset are not moved.

Controlling Flip-Flop Retiming

Use the following constraints to control flip-flop retiming:

- Register Balancing (REGISTER_BALANCING)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)

Partitions

XST now supports Partitions in place of Incremental Synthesis. Incremental Synthesis is no longer supported. The incremental_synthesis and resynthesize constraints are no longer supported. For more information on Partitions, see the ISE® Design Suite Help.

Speed Optimization Under Area Constraint

XST performs timing optimization under the area constraint. This option is named:

- LUT-FF Pairs Utilization Ratio (Virtex®-5 devices)
- Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)" (all other FPGA devices)

Define in ISE® Design Suite with:

Process > Properties > XST Synthesis Options.

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 inference.

If this constraint is specified, XST makes an area estimation. If the specified constraint is met, XST continues timing optimization trying not to exceed the constraint. If the design is larger than requested, XST tries to reduce the area first. If the area constraint is met, XST begins timing optimization.

Speed Optimization Under Area Constraint Example One (100%)

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 begins 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.

=========================================================================
```

Speed Optimization Under Area Constraint Example Two (70%)

If the area constraint cannot be met, XST ignores it during timing optimization and runs low level synthesis to achieve the best frequency. In the following example, the target area constraint is set to 70%. Since XST was unable to satisfy the target area constraint, XST issues the following warning:

```
...
=========================================================================
*
*                     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. 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.

Speed Optimization Under Area Constraint Example Three (55%)

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

In some situations, it is important to disable automatic resource management. To do so, specify **`-1`** as the value for SLICE_UTILIZATION_RATIO.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) can be attached to a specific block of a design. You can specify an absolute number of slices (or FF-LUT pairs) as a percentage of the total number.

FPGA Optimization Report Section

During design optimization, XST reports:

- 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 (REGISTER_DUPLICATION) constraint.

Design Optimization Report Example

```
Starting low level synthesis ...
Optimizing unit <down4cnt> ...
Optimizing unit <doc_readwrite> ...
 ...
Optimizing unit <doc> ...
Building and optimizing final netlist ...
The FF/Latch <doc_readwrite/state_D2> in Unit <doc> is equivalent to the following 2 FFs/Latches,
which will be removed : <doc_readwrite/state_P2> <doc_readwrite/state_M2>Register
doc_reset_I_reset_out has been replicated 2 time(s)
Register wr_l has been replicated 2 time(s)
```

Cell Usage Report

The Cell Usage section of the Final Report gives the count of all the primitives used in the design. The primitives are classified in the following groups:

- BELS Cell Usage
- Flip-Flops and Latches Cell Usage
- RAMS Cell Usage
- SHIFTERS Cell Usage
- Tristates Cell Usage
- Clock Buffers Cell Usage
- IO Buffers Cell Usage
- LOGICAL Cell Usage
- OTHER Cell Usage

BELS Cell Usage

The BELS group in the Cell Usage section of the Final Report contains all the logical cells that are basic elements of the targeted FPGA device family, for example:

- LUTs
- MUXCY
- MUXF5
- MUXF6
- MUXF7
- MUXF8

Flip-Flops and Latches Cell Usage

The Flip-Flops and Latches group in the Cell Usage section of the Final Report contains all the flip-flops and latches that are primitives of the targeted FPGA device family, for example:

- FDR
- FDRE
- LD

RAMS Cell Usage

The RAMS group in the Cell Usage section of the Final Report contains all the RAMs.

SHIFTERS Cell Usage

The SHIFTERS group in the Cell Usage section of the Final Report contains all the shift registers that use the Virtex® device primitive:

- TSRL16
- SRL16_1
- SRL16E
- SRL16E_1
- SRLC

Tristates Cell Usage

The Tristates group in the Cell Usage section of the Final Report contains all the tristate primitives:

BUFT

Clock Buffers Cell Usage

The Clock Buffers group in the Cell Usage section of the Final Report contains all the clock buffers:

- BUFG
- BUFGP
- BUFGDLL

IO Buffers Cell Usage

The IO Buffers group in the Cell Usage section of the Final Report contains all the standard I/O buffers (except the clock buffer):

- IBUF
- OBUF
- IOBUF
- OBUFT
- IBUF_GTL ...

LOGICAL Cell Usage

The LOGICAL group in the Cell Usage section of the Final Report contains all the logical cells primitives that are not basic elements:

- AND2
- OR2 ...

OTHER Cell Usage

The OTHER group in the Cell Usage section of the Final Report contains all the cells that have not been classified in the previous groups.

Cell Usage Report Example

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

Where XST estimates the number of slices and gives, for example, the number of flip-flops, IOBs, and BRAMS. This report is very close to the one produced by MAP.

A short table gives information about the number of clocks in the design, how each clock is buffered, and how many loads it has.

A short table gives information about the number of asynchronous set/reset signals in the design, how each signal is buffered, and how many loads it has.

Timing Report

At the end of synthesis, XST reports the timing information for the design. The Timing Report shows the information for all four possible domains of a netlist:

- register to register
- input to register
- register to outpad
- inpad to outpad

Timing Report Example

These timing numbers are only a synthesis estimate. For accurate timing information, see the TRACE report generated after place-and-route.

```
Clock Information:
------------------
-----------------------------------+------------------------+-------+
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+
CLK                                | BUFGP                  | 11    |
-----------------------------------+------------------------+-------+

Asynchronous Control Signals Information:
----------------------------------------
-----------------------------------+--------------------------------+-------+
Control Signal                     | Buffer(FF name)                | Load  |
-----------------------------------+--------------------------------+-------+
rstint(MACHINE/current_state_Out01:O)| NONE(sixty/lsbcount/qoutsig_3)| 4     |
RESET                              | IBUF                           | 3     |
sixty/msbclr(sixty/msbclr:O)       | NONE(sixty/msbcount/qoutsig_3)| 4     |
-----------------------------------+--------------------------------+-------+

Timing Summary:
---------------
Speed Grade: -12

   Minimum period: 2.644ns (Maximum Frequency: 378.165MHz)
   Minimum input arrival time before clock: 2.148ns
   Maximum output required time after clock: 4.803ns
   Maximum combinational path delay: 4.473ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'CLK'
  Clock period: 2.644ns (frequency: 378.165MHz)
  Total number of paths / destination ports: 77 / 11
-------------------------------------------------------------------------
Delay:               2.644ns (Levels of Logic = 3)
  Source:            MACHINE/current_state_FFd3 (FF)
  Destination:       sixty/msbcount/qoutsig_3 (FF)
  Source Clock:      CLK rising
  Destination Clock: CLK rising

  Data Path: MACHINE/current_state_FFd3 to sixty/msbcount/qoutsig_3
                              Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDC:C->Q             8   0.272   0.642  MACHINE/current_state_FFd3 (MACHINE/current_state_FFd3)
     LUT3:I0->O           3   0.147   0.541  Ker81 (clkenable)
     LUT4_D:I1->O         1   0.147   0.451  sixty/msbce (sixty/msbce)
     LUT3:I2->O           1   0.147   0.000  sixty/msbcount/qoutsig_3_rstpot (N43)
     FDC:D                    0.297          sixty/msbcount/qoutsig_3
    ----------------------------------------
    Total                     2.644ns (1.010ns logic, 1.634ns route)
                                      (38.2% logic, 61.8% route)
```

Timing Report Timing Summary Section

The Timing Summary section of the Timing Report summarizes the timing paths for all four 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, *No path found* is printed instead of the value.

Timing Report Timing Detail Section

The Timing Detail section of the Timing Report describes the most critical path in detail for each region:

- Start point of the path
- End point of the path
- Maximum delay of the path
- 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)
```

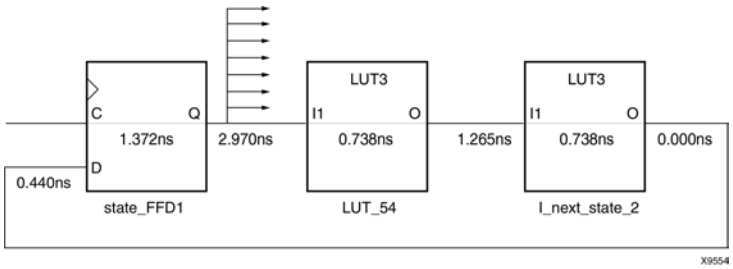
The detailed path shows:

- Cell type
- Input and output of this gate
- Fanout at the output
- Gate delay
- Net delay estimate
- Name of the instance.

When entering a hierarchical block, **`begin scope`** is printed. When exiting a hierarchical block, **`end scope`** is printed.

Timing Report Schematic

The preceding report corresponds to the following schematic:



Timing Report Paths and Ports

The Timing Report section shows the number of analyzed paths and ports. If XST is run with timing constraints, it also shows the number of failed paths and ports. The number of analyzed and failed paths shows how many timing problems there are in the design. The number of analyzed and failed ports may show 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 that the number of failed destination ports is only two flip-flops. In that case, it is sufficient to analyze the design description for these two flip-flops only in order to detect the changes necessary to meet timing.

Implementation Constraints

XST writes all implementation constraints generated from Hardware Description Language (HDL) or constraint file attributes (LOC, ...) into the output NGC file.

Keep (KEEP) properties are generated during buffer insertion for maximum fanout control or for optimization.

FPGA Device Primitive Support

XST enables you to instantiate device primitives directly in your VHDL or Verilog code. Primitives such as the following can be manually inserted in your HDL design through instantiation:

- MUXCY_L
- LUT4_L
- CLKDLL
- RAMB4_S1_S16
- IBUFG_PCI33_5
- NAND3b2

These primitives:

- Are compiled in the UNISIM library
- Are not optimized by XST by default
- 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.

In order to simplify instantiation of complex primitives as RAMs, XST supports an additional library called UniMacro. For more information, see the *Libraries Guides*.

Generating Primitives Through Attributes

Some primitives can be generated through attributes:

- Buffer Type (BUFFER_TYPE) 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.
- I/O Standard (IOSTANDARD) can be used to assign an I/O standard to an I/O primitive. For example, the following assigns PCI33_5 I/O standard to the I/O port:

```
// synthesis attribute IOSTANDARD of in1 is PCI33_5
```

Primitives and Black Boxes

The primitive support is based on the concept of the black box. For information on the basics of black box support, see Safe FSM Implementation.

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 Xilinx® device primitive. To avoid confusion about how XST interprets this module, attach BoxType (BOX_TYPE) to the component declaration of MUXF5.

If BoxType (BOX_TYPE) is attached to the MUXF5 with a value of:

- **`primitive`**, or **`black_box`**

  XST tries to interpret this module as a Xilinx device 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 Xilinx device primitive, XST renames it to a unique name and issues a warning. For example, MUX5 could be renamed to MUX51 as shown in the following log file example:

```
...
=========================================================================
*                     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 BoxType (BOX_TYPE) is not attached to the MUXF5, XST processes this block as a user hierarchical block. If the name of the user black box is the same as that of a Xilinx device primitive, XST renames it to a unique name and issues a warning.

VHDL and Verilog Xilinx Device Primitives Libraries

XST provides dedicated libraries, both in VHDL and Verilog, simplifying instantiation of Xilinx device primitives in your HDL source code These libraries contain the complete set of Xilinx device primitives declarations with a BoxType (BOX_TYPE) constraint attached to each component.

VHDL Xilinx Device Primitives Device Libraries

In VHDL, 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\unisims_vcomp.vhd` file of the XST installation.

Verilog Xilinx Device Primitives Device Libraries

In Verilog, the UNISIM library is precompiled. XST automatically links it with your design.

Primitive Instantiation Guidelines

Use UPPERCASE for generic (VHDL) and parameter (Verilog) values when instantiating primitives. For example the ODDR element has the following component declaration in the UNISIM library:

```
component ODDR
generic
  (DDR_CLK_EDGE : string := "OPPOSITE_EDGE";
   INIT : bit := '0';
   SRTYPE : string := "SYNC");

port(Q : out std_ulogic;
     C : in std_ulogic;
     CE : in std_ulogic;
     D1 : in std_ulogic;
     D2 : in std_ulogic;
     R : in std_ulogic;
     S : in std_ulogic);
end component;
```

When you instantiate this primitive in your code, the values of DDR_CLK_EDGE and SRTYPE generics must be in uppercase. If not, XST issues a warning stating that unknown values are used.

Some primitives, such as LUT1, enable you to use an INIT during instantiation. The two ways to pass an INIT to the final netlist are:

- Attach an INIT attribute to the instantiated primitive.
- Pass the INIT with the generics mechanism (VHDL), or the parameters mechanism (Verilog). Xilinx recommends this method, since it allows you to use the same code for synthesis and simulation.

Reporting of Instantiated Device Primitives

XST does not issue any message concerning instantiation of instantiated device primitives during HDL synthesis because the BoxType (BOX_TYPE) attribute with its value, *primitive*, is attached to each primitive in the UNISIM library.

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 BoxType (BOX_TYPE) constraint to it with a value of *user_black_box*, XST issues a warning as shown in the following log file example:

```
...
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.
...
```

Primitives Related Constraints

- BoxType (BOX_TYPE)
- The PAR constraints that can be passed from HDL to NGC without processing

Primitives Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Passing an INIT Value Via the INIT Constraint VHDL Coding Example

```
--
-- Passing an INIT value via the INIT constraint.
--

library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_1 is
    port(I0,I1 : in std_logic;
         O     : out std_logic);
end primitive_1;

architecture beh of primitive_1 is

    attribute INIT: string;
    attribute INIT of inst: label is "1";

begin

    inst: LUT2 port map (I0=>I0,I1=>I1,O=>O);

end beh;
```

Passing an INIT Value Via the INIT Constraint Verilog Coding Example

```
//
// Passing an INIT value via the INIT constraint.
//

module v_primitive_1 (I0,I1,O);
    input I0,I1;
    output O;

    (* INIT="1" *)
  LUT2 inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

Passing an INIT Value Via the Generics Mechanism VHDL Coding Example

```
--
-- Passing an INIT value via the generics mechanism.
--

library ieee;
use ieee.std_logic_1164.all;

library unisim;
use unisim.vcomponents.all;

entity primitive_2 is
    port(I0,I1 : in std_logic;
         O    : out std_logic);
end primitive_2;

architecture beh of primitive_2 is
begin

    inst: LUT2 generic map (INIT=>"1")
          port map (I0=>I0,I1=>I1,O=>O);

end beh;
```

Passing an INIT Value Via the Parameters Mechanism Verilog Coding Example

```
//
// Passing an INIT value via the parameters mechanism.
//

module v_primitive_2 (I0,I1,O);
    input I0,I1;
    output O;

    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));

endmodule
```

Passing an INIT Value Via the Defparam Mechanism Verilog Coding Example

```
//
// Passing an INIT value via the defparam mechanism.
//

module v_primitive_3 (I0,I1,O);
    input I0,I1;
    output O;

    LUT2 inst (.I0(I0), .I1(I1), .O(O));
    defparam inst.INIT = 4'h1;

endmodule
```

Using the UniMacro Library

In order to simplify instantiation of such complex primitives as RAMs, XST supports an additional library called UniMacro. UniMacro libraries are supported for Virtex®-4 devices, Virtex-5 devices, and newer devices. For more information, see the *Libraries Guides*.

In VHDL, declare library **`unimacro`** with its package **`vcomponents`** in your source code:

```
library unimacro;
use unimacro.vcomponents.all;
```

The source code of this package can be found in the `vhdl\src\ unisims\unisims_vcomp.vhd` file in the XST installation.

In Verilog, the UniMacro library is precompiled. XST automatically links it with your design.

Cores Processing

If a design contains cores represented by an Electronic Data Interchange Format (EDIF) or an NGC file, XST can automatically read them for timing estimation and area utilization control. Use ISE® Design Suite **Process > Properties > Synthesis Options > Read Cores** 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. XST reads cores by default.

Cores Processing VHDL Coding Example

In the following VHDL coding example, the block **my_add** is an adder, which is represented as a black box in the design whose netlist was generated by the CORE Generator™ software.

```
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;
```

Read Cores Enabled or Disabled

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, XST issues 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.

By default, XST reads Electronic Data Interchange Format (EDIF) and NGC cores from the current (project) directory. If the cores are not in the project directory, specify the directory in which the cores are located with Cores Search Directories (-sd).

Specifying INIT and RLOC

Use the UNISIM library to directly instantiate LUT components in your Hardware Description Language (HDL) code. To specify a function that a particular LUT must execute, apply an INIT constraint to the instance of the LUT. To place an instantiated LUT or register in a particular slice of the chip, 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 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. XST automatically recognizes the XC_MAP constraint supported by Synplicity.

Passing an INIT Value Via the LUT_MAP Constraint Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

The following coding examples show how to pass an INIT value using the LUT_MAP constraint:

- Passing an INIT Value Via the LUT_MAP Constraint VHDL Coding Example
- Passing an INIT Value Via the LUT_MAP Constraint Verilog Coding Example

In these examples, 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. For more information, see Map Entity on a Single LUT (LUT_MAP)

Passing an INIT Value Via the LUT_MAP Constraint VHDL Coding Example

```
--
-- Mapping on LUTs via LUT_MAP constraint
--

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 inits_rlocs_1 is
    port(A,B,C : in std_logic;
         REZ : out std_logic);
end inits_rlocs_1;

architecture beh of inits_rlocs_1 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;
```

Passing an INIT Value Via the LUT_MAP Constraint Verilog Coding Example

```
//
// Mapping on LUTs via LUT_MAP constraint
//

(* LUT_MAP="yes" *)
module v_and_one (A, B, REZ);
    input A, B;
    output REZ;


    and and_inst(REZ, A, B);

endmodule

// ------------------------------------------------

(* LUT_MAP="yes" *)
module v_and_two (A, B, REZ);
    input A, B;
    output REZ;

    or or_inst(REZ, A, B);

endmodule

// ------------------------------------------------

module v_inits_rlocs_1 (A, B, C, REZ);
    input A, B, C;
    output REZ;

    wire tmp;

    v_and_one inst_and_one (A, B, tmp);
    v_and_two inst_and_two (tmp, C, REZ);

endmodule
```

Specifying INIT Value for a Flip-Flop Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

If a function cannot be mapped on a single LUT, XST issues an error message and interrupts synthesis. To define an INIT value for a flip-flop or a shift register, described at RTL level, 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 or shift register.

In the following coding examples, 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.

Specifying INIT Value for a Flip-Flop VHDL Coding Example

```
--
-- Specification on an INIT value for a flip-flop, described at RTL level
--

library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_2 is
    port (CLK : in std_logic;
          DI  : in std_logic_vector(3 downto 0);
          DO  : out std_logic_vector(3 downto 0));
end inits_rlocs_2;

architecture beh of inits_rlocs_2 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;
```

Specifying INIT Value for a Flip-Flop Verilog Coding Example

```
//
// Specification on an INIT value for a flip-flop,
// described at RTL level
//

module v_inits_rlocs_2 (clk, di, do);
    input  clk;
    input  [3:0] di;
    output [3:0] do;
    reg    [3:0] tmp;

    initial begin
        tmp = 4'b1011;
    end

    always @(posedge clk)
    begin
        tmp <= di;
    end

    assign do = tmp;

endmodule
```

Specifying INIT and RLOC Values for a Flip-Flop Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

To infer a register and place it in a specific location of a chip, attach an RLOC constraint to the **tmp** signal as shown in the following coding examples.

XST propagates it to the final netlist. This feature is supported for registers, and also for inferred block RAM if it can be implemented on a single block RAM primitive.

Specifying INIT and RLOC Values for a Flip-Flop VHDL Coding Example

```
--
-- Specification on an INIT and RLOC values for a flip-flop, described at RTL level
--

library ieee;
use ieee.std_logic_1164.all;

entity inits_rlocs_3 is
    port (CLK : in std_logic;
          DI : in std_logic_vector(3 downto 0);
          DO : out std_logic_vector(3 downto 0));
end inits_rlocs_3;

architecture beh of inits_rlocs_3 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;
```

Specifying INIT and RLOC Values for a Flip-Flop Verilog Coding Example

```
//
// Specification on an INIT and RLOC values for a flip-flop,
// described at RTL level
//

module v_inits_rlocs_3 (clk, di, do);
    input  clk;
    input  [3:0] di;
    output [3:0] do;
    (* RLOC="X3Y0 X2Y0 X1Y0 X0Y0" *)
  reg    [3:0] tmp;

    initial begin
        tmp = 4'b1011;
    end

    always @(posedge clk)
    begin
        tmp <= di;
    end

    assign do = tmp;

endmodule
```

Using PCI Flow With XST

To satisfy placement constraints and meet timing requirements when using PCI™ flow with XST:

- For VHDL, ensure that the names in the generated netlist are all in *UPPER* case.

  The default case is *lower*. Specify the case in ISE® Design Suite in **Process > Properties > Synthesis Options > Case**.

- For Verilog, ensure that Case is set to *maintain*.

  The default case is *maintain*. Specify the case in ISE Design Suite in **Process > Properties > Synthesis Options > Case**.

- Preserve the hierarchy of the design.

  Specify the Keep Hierarchy (KEEP_HIERARCHY) setting in ISE Design Suite in **Process > Properties > Synthesis Options > Keep Hierarchy**.

- Preserve equivalent flip-flops.

  XST removes equivalent flip-flops by default. Specify the Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) setting in ISE Design Suite in **Process > Properties > Xilinx® Specific Options > Equivalent Register Removal**

Preventing Logic and Flip-Flop Replication

To prevent logic and flip-flop replication caused by a high fanout flip-flop set/reset signal:

- Set a high maximum fanout value for the entire design in ISE® Design Suite in **Process > Properties > Synthesis Options > Max Fanout**, or
- Use Max Fanout (MAX_FANOUT) to set a high maximum fanout value for the initialization signal connected to the RST port of PCI™ core (for example, **`max_fanout=2048`**).

Disabling Read Cores

Disabling Read Cores prevents XST from automatically reading PCI™ cores for timing and area estimation. In reading PCI cores, XST may perform logic optimization that does not allow the design to meet timing requirements, or which might lead to errors during MAP. To disable Read Cores, uncheck it in ISE® Design Suite in**Process > Properties > Synthesis Options > Read Cores**.

By default, XST reads cores for timing and area estimation.

Chapter 5

XST CPLD Optimization

This chapter discusses CPLD synthesis options and the implementation details for macro generation. This chapter includes:

- CPLD Synthesis Options
- Implementation Details for Macro Generation
- CPLD Synthesis Log File Analysis
- CPLD Synthesis Constraints
- Improving Results in CPLD Synthesis

CPLD Synthesis Options

XST generates an NGC file ready for the CPLD fitter.

The general flow of XST for CPLD synthesis is:

1. Hardware Description Language (HDL) synthesis of VHDL or Verilog designs
2. Macro inference
3. Module optimization
4. NGC file generation

This section describes supported CPLD families. It lists the XST options related *only* to CPLD synthesis that can be set ISE® Design Suite in **Process > Properties**.

CPLD Synthesis Supported Devices

XST supports CPLD synthesis for the following devices:

- CoolRunner™ XPLA3
- CoolRunner-II
- XC9500
- XC9500XL

The synthesis for CoolRunner XPLA3 device families and XC9500XL device families includes clock enable processing. You can allow or invalidate the clock enable signal. When invalidated, it is replaced by equivalent logic. The selection of the macros that use the clock enable (counters, for instance) depends on the device type. A counter with clock enable is accepted for the CoolRunner XPLA3 device families and XC9500XL device families, but rejected (replaced by equivalent logic) for XC9500 devices.

Setting CPLD Synthesis Options

Set the following CPLD synthesis options in ISE® Design Suite in **Process > Properties > Synthesis Options.** For more information, see XST CPLD Constraints (Non-Timing).

- Keep Hierarchy (KEEP_HIERARCHY)
- Macro Preserve (-pld_mp)
- XOR Preserve (-pld_xp)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- Clock Enable (-pld_ce)
- WYSIWYG (-wysiwyg)
- No Reduce (NOREDUCE)

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 command line option, which can take two values:

- **yes**

  macro generation is allowed.

- **no**

  macro generation is inhibited.

The general macro generation flow is:

1. Hardware Description Language (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 optimization gives better results for larger components.

CPLD Synthesis Log File Analysis

XST messages related to CPLD synthesis are located after the following message:

```
Low Level Synthesis
```

The XST log file 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
```

CPLD Synthesis Constraints

The constraints (attributes) specified in the Hardware Description Language (HDL) design or in the constraint files are written by XST into the NGC file as signal properties.

Improving Results in CPLD Synthesis

XST produces optimized netlists for the CPLD fitter, which:

- Fits them in specified devices
- Creates the download programmable files

The CPLD low-level optimization of XST consists of:

- Logic minimization
- Subfunction collapsing
- Logic factorization
- Logic decomposition

Optimization results in an NGC netlist corresponding to Boolean equations. The CPLD fitter reassembles these equations to fit the best of the macrocell capacities. A special XST optimization process, known as equation shaping, is applied for XC9500 and XC9500XL 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. This algorithm tries to reduce the number of levels of critical paths.

Xilinx® recommends CPLD fitter multi-level optimization because of the special optimizations done by the fitter:

- D to T flip-flop conversion
- De Morgan Boolean expression selection

Obtaining Better Frequency

The frequency depends on the number of logic levels (logic depth). To reduce the number of levels, Xilinx® recommends the following options:

- Optimization Effort

  Set Optimization Effort to **`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

  Set Optimization Goal to **Speed**. The priority is the reduction of number of levels.

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, beginning with 20 and finishing with 50 with a step of 5. Statistically the value 30 gives the best results for frequency.

The following tries, in this order, may give successively better results for frequency:

- Obtaining Better Frequency Try 1
- Obtaining Better Frequency Try 2
- Obtaining Better Frequency Try 3
- Obtaining Better Frequency Try 4

The CPU time increases from Try 1 to Try 4.

Obtaining Better 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

Obtaining Better Frequency Try 2

Flatten the user hierarchy. In this case optimization 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

Obtaining Better Frequency 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

Obtaining Better Frequency Try 4

Apply the equation shaping algorithm. Options to be selected:

- Optimization effort

  2 or **High**
- Macro Preserve

  no
- Keep Hierarchy

  no

Fitting 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 is **`-wysiwyg`** *yes*. This option may be useful when the design cannot be simplified by optimization and the complexity (in number of P-Terms) is near the device capacity. It may be that optimization, 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.

Chapter 6

XST Design Constraints

This chapter provides general information about XST design constraints, as well as information about specific constraints.

For general information about XST design constraints, see:

- List of XST Design Constraints
- Setting Global Constraints and Options
- VHDL Attribute Syntax
- Verilog-2001 Attributes
- XST Constraint File (XCF)
- Constraints Priority
- XST Specific Non-Timing Options
- XST Command Line Only Options

For information about specific XST design constraints, see:

- XST General Constraints
- XST HDL Constraints
- XST FPGA Constraints (Non-Timing)
- XST CPLD Constraints (Non-Timing)
- XST Timing Constraints
- XST Implementation Constraints
- XST-Supported Third Party Constraints

Constraints help you meet your design goals and obtain the best implementation of your circuit. Constraints control various aspects of synthesis, as well as placement and routing. Synthesis algorithms and heuristics automatically provide optimal results in most situations. If synthesis fails to initially achieve optimal results, use available constraints to try other synthesis alternatives.

The following mechanisms are available to specify constraints:

- Options provide global control on most synthesis aspects. They can be set either in ISE® Design Suite in **Process > Properties > Synthesis Options,** or by the **`run`** command from the command line.
- VHDL attributes can be directly inserted into the VHDL code and attached to individual elements of the design to control both synthesis, and placement and routing.
- Constraints can be added as Verilog attributes (preferred) or Verilog meta comments.
- Constraints can be specified in a separate constraint file.

Global synthesis settings are typically defined in ISE Design Suite in **Process > Properties > Synthesis Options**, or from the command line. VHDL and Verilog attributes and Verilog meta comments can be inserted in your source code to specify different choices for individual parts of the design.

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).

Follow these general rules:

- Several constraints can be applied on signals. In this case, the constraint must be placed in the block where the signal is declared and used.
- If a constraint can be applied on an entity (VHDL), then it can also be applied on the component declaration. The ability to apply constraints on components is not explicitly stated for each individual constraint, since it is a general XST rule.
- Some third party synthesis tools allow you to apply constraints on architectures. XST allows constraints on architectures only for those third party constraints automatically supported by XST.

List of XST Design Constraints

Following is a list of XST Design Constraints, organized by type:

XST General Constraints

The following constraints are found in XST General Constraints.

- Add I/O Buffers (-iobuf)
- BoxType (BOX_TYPE)
- Bus Delimiter (-bus_delimiter)
- Case (-case)
- Case Implementation Style (-vlgcase)
- Verilog Macros (-define)
- Duplication Suffix (-duplication_suffix)
- Full Case (FULL_CASE)
- Generate RTL Schematic (-rtlview)
- Generics (-generics)
- Hierarchy Separator (-hierarchy_separator)
- I/O Standard (IOSTANDARD)
- Keep (KEEP)
- Keep Hierarchy (KEEP_HIERARCHY)
- Library Search Order (-lso)
- LOC
- Netlist Hierarchy (-netlist_hierarchy)
- Optimization Effort (OPT_LEVEL)
- Optimization Goal (OPT_MODE)
- Parallel Case (PARALLEL_CASE)
- RLOC
- Save (S / SAVE)
- Synthesis Constraint File (-uc)
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)
- Use Synthesis Constraints File (-iuc)
- Verilog Include Directories (-vlgincdir)
- Verilog 2001 (-verilog2001)
- HDL Library Mapping File (-xsthdpini)
- Work Directory (-xsthdpdir)

XST Hardware Description Language (HDL) Constraints

The following constraints are found in XST HDL Constraints.

- Automatic FSM Extraction (FSM_EXTRACT)
- Enumerated Encoding (ENUM_ENCODING)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- FSM Encoding Algorithm (FSM_ENCODING)
- Mux Extraction (MUX_EXTRACT)
- Register Power Up (REGISTER_POWERUP)
- Resource Sharing (RESOURCE_SHARING)
- Safe Recovery State (SAFE_RECOVERY_STATE)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Signal Encoding (SIGNAL_ENCODING)

XST FPGA Constraints (Non-Timing)

The following are XST FPGA Constraints (Non-Timing).

- Asynchronous to Synchronous (ASYNC_TO_SYNC)
- Automatic BRAM Packing (AUTO_BRAM_PACKING)
- BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)
- Buffer Type (BUFFER_TYPE)
- Extract BUFGCE (BUFGCE)
- Cores Search Directories (-sd)
- Decoder Extraction (DECODER_EXTRACT)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- FSM Style (FSM_STYLE)
- Power Reduction (POWER)
- Read Cores (READ_CORES)
- Logical Shifter Extraction (SHIFT_EXTRACT)
- LUT Combining (LC)
- Map Logic on BRAM (BRAM_MAP)
- Max Fanout (MAX_FANOUT)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)
- Multiplier Style (MULT_STYLE)
- Mux Style (MUX_STYLE)
- Number of Global Clock Buffers (-bufg)
- Number of Regional Clock Buffers (-bufr)
- Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)
- Pack I/O Registers Into IOBs (IOB)
- Priority Encoder Extraction (PRIORITY_EXTRACT)
- RAM Extraction (RAM_EXTRACT)
- RAM Style (RAM_STYLE)
- Reduce Control Sets (REDUCE_CONTROL_SETS)
- Register Balancing (REGISTER_BALANCING)

- Register Duplication (REGISTER_DUPLICATION)
- ROM Extraction (ROM_EXTRACT)
- ROM Style (ROM_STYLE)
- Shift Register Extraction (SHREG_EXTRACT)
- Slice Packing (-slice_packing)
- XOR Collapsing (XOR_COLLAPSE)
- Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)
- Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN)
- Map Entity on a Single LUT (LUT_MAP)
- Use Carry Chain (USE_CARRY_CHAIN)
- Convert Tristates to Logic (TRISTATE2LOGIC)
- Use Clock Enable (USE_CLOCK_ENABLE)
- Use Synchronous Set (USE_SYNC_SET)
- Use Synchronous Reset (USE_SYNC_RESET)
- Use DSP48 (USE_DSP48)

XST CPLD Constraints (Non-Timing)

The following constraints are found in XST CPLD Constraints (Non-Timing:

- Clock Enable (-pld_ce)
- Data Gate (DATA_GATE)
- Macro Preserve (-pld_mp)
- No Reduce (NOREDUCE)
- WYSIWYG (-wysiwyg)
- XOR Preserve (-pld_xp)

XST Timing Constraints

The following constraints are found in XST Timing Constraints.

- Cross Clock Analysis (-cross_clock_analysis)
- Write Timing Constraints (-write_timing_constraints)
- Clock Signal (CLOCK_SIGNAL)
- Global Optimization Goal (-glob_opt)
- XCF Timing Constraint Support
- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)

XST Implementation Constraints

The following constraints are found in XST Implementation Constraints.

- RLOC
- NOREDUCE
- PWR_MODE

Third Party Constraints

For a discussion of Third Party Constraints and their XST equivalents, see XST-Supported Third Party Constraints.

Setting Global Constraints and Options

This section explains how to set global constraints and options in ISE® Design Suite in **Process > Properties**.

For a description of each constraint that applies generally (that is, to FPGA devices, CPLD devices, VHDL, and Verilog) see the *Constraints Guide*.

Except for **Value** fields with check boxes, there is a pulldown arrow or browse button in each **Value** field. The arrow is not visible until you click in the **Value** field.

Setting Synthesis Options

To set Hardware Description Language (HDL) synthesis options from ISE® Design Suite:

1. Select a source file from the **Source File** window.
2. Right-click **Synthesize - XST** in the **Process** window.
3. Select **Properties**.
4. Select **Synthesis Options**.
5. Depending on the device type you have selected (FPGA or CPLD devices), one of two dialog boxes opens.
6. Select any of the following synthesis options:

- Optimization Goal (OPT_MODE)
- Optimization Effort (OPT_LEVEL)
- Use Synthesis Constraints File (-iuc)
- Synthesis Constraint File (-uc)
- Library Search Order (-lso)
- Global Optimization Goal (-glob_opt)
- Generate RTL Schematic (-rtlview)
- Write Timing Constraints (-write_timing_constraints)
- Verilog 2001 (-verilog2001)

To view the following options, select **Edit > Preferences > Processes > Property Display Level > Advanced**:

- Keep Hierarchy (KEEP_HIERARCHY)
- Cores Search Directories (-sd)
- Cross Clock Analysis (-cross_clock_analysis)
- Hierarchy Separator (-hierarchy_separator)
- Bus Delimiter (-bus_delimiter)
- Case (-case)
- Work Directory (-xsthdpdir)
- HDL Library Mapping File (-xsthdpini)
- Verilog Include Directories (-vlgincdir)
- Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)

Setting Hardware Description Language (HDL) Options

You can set Hardware Description Language (HDL) options for FPGA devices and CPLD devices.

Setting Hardware Description Language (HDL) Options for FPGA Devices

To set Hardware Description Language (HDL) options for FPGA devices, in ISE® Design Suite select **Process > Properties > Synthesize - XST > HDL Options**.

The following HDL Options can be set for FPGA devices:

- FSM Encoding Algorithm (FSM_ENCODING)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Case Implementation Style (-vlgcase)
- FSM Style (FSM_STYLE)

  To view FSM Style, select **Edit > Preferences > Processes > Property Display Level > Advanced**.
- RAM Extraction (RAM_EXTRACT)
- RAM Style (RAM_STYLE)
- ROM Extraction (ROM_EXTRACT)
- ROM Style (ROM_STYLE)
- Mux Extraction (MUX_EXTRACT)
- Mux Style (MUX_STYLE)
- Decoder Extraction (DECODER_EXTRACT)
- Priority Encoder Extraction (PRIORITY_EXTRACT)
- Shift Register Extraction (SHREG_EXTRACT)
- Logical Shifter Extraction (SHIFT_EXTRACT)
- XOR Collapsing (XOR_COLLAPSE)
- Resource Sharing (RESOURCE_SHARING)
- Multiplier Style (MULT_STYLE)

  For later devices, Multiplier Style is renamed as follows:
  - Use DSP48 (Virtex®-4 devices)
  - Use DSP Block (Virtex-5 devices and Spartan®-3A DSP devices)
- Use DSP48 (USE_DSP48)

Setting Hardware Description Language (HDL) Options for CPLD Devices

To set Hardware Description Language (HDL) options for CPLD devices, in ISE Design Suite select **Process > Properties > Synthesize - XST > Options**.

The following HDL Options can be set for CPLD devices:

- FSM Encoding Algorithm (FSM_ENCODING)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Case Implementation Style (-vlgcase)
- Mux Extraction (MUX_EXTRACT)
- Resource Sharing (RESOURCE_SHARING)

Setting Xilinx Specific Options

You can set Xilinx® specific options for:

- FPGA devices
- CPLD devices

.

Setting Xilinx Specific Options for FPGA Devices

To set Xilinx specific options for FPGA devices, in ISE® Design Suite select **Process > Properties > Synthesis Options > Xilinx Specific Options**.

The following Xilinx specific options can be set for FPGA devices:

- Add I/O Buffers (-iobuf)
- LUT Combining (LC)
- Max Fanout (MAX_FANOUT)
- Register Duplication (REGISTER_DUPLICATION)
- Reduce Control Sets (REDUCE_CONTROL_SETS)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- Register Balancing (REGISTER_BALANCING)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)
- Convert Tristates to Logic (TRISTATE2LOGIC)

  Convert Tristate to Logic appears only when working with devices with internal tristate resources.
- Use Clock Enable (USE_CLOCK_ENABLE)
- Use Synchronous Set (USE_SYNC_SET)
- Use Synchronous Reset (USE_SYNC_RESET)

To display the following options, select **Edit > Preferences > Processes > Property Display Level > Advanced**:

- Number of Global Clock Buffers (-bufg)
- Number of Regional Clock Buffers (-bufr)

Setting Xilinx Specific Options for CPLD Devices

To set Xilinx specific options for CPLD devices, in ISE Design Suite select **Process > Properties > Synthesis Options > Xilinx Specific Options**.

The following Xilinx specific options can be set for CPLD devices:

- Add I/O Buffers (-iobuf)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- Clock Enable (-pld_ce)
- Macro Preserve (-pld_mp)
- XOR Preserve (-pld_xp)
- WYSIWYG (-wysiwyg)

Setting Other XST Command Line Options

Set other XST command line options in ISE® Design Suite in **Process > Properties > Other XST Command Line Options**. This is an advanced property. Use the syntax described in XST Command Line Mode. Separate multiple options with a space.

While **Other XST Command Line Options** is intended for XST options not listed in **Process > Properties**, if an option already listed is entered, precedence is given to that option. Illegal or unrecognized options cause XST to stop processing and generate a message such as:

```
ERROR:Xst:1363 - Option "-verilog2002" is not available for command run.
```

Custom Compile File List

Use the Custom Compile File List property to 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.

List all design files and their libraries in the order in which they are to be compiled, from top to bottom. Type each file and library pair on its own line, with a semicolon separating the library from the file as follows:

library_name;file_name [*library_name;file_name*] ...

Following is an example:

```
work;stopwatch.vhd
work;statmach.vhd
...
```

Since this property is not connected to **Simulation Properties > Custom Compile File List**, a different compile list file is used for synthesis than for simulation.

VHDL Attribute Syntax

You can describe constraints with VHDL attributes in the VHDL code. Before it can be used, an attribute must be declared with the following syntax:

```
attribute AttributeName : Type ;
```

VHDL Attribute Syntax Example One

```
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 ;
```

VHDL Attribute Syntax Example Two

```
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.

Follow these general rules:

- If a constraint can be applied on an entity (VHDL), then it can also be applied on the component declaration. The ability to apply constraints on components is not explicitly stated for each individual constraint, since it is a general XST rule.
- Some third party synthesis tools allow you to apply constraints on architectures. XST allows constraints on architectures only for those third party constraints automatically supported by XST.

Verilog-2001 Attributes

XST supports Verilog-2001 attribute statements. Attributes are comments that 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. Other attribute declarations may be supported by the compiler, but are ignored by XST.

Use attributes to:

- Set constraints on individual objects, for example:
  - module
  - instance
  - net
- Set the following synthesis constraints
  - Full Case (FULL_CASE)
  - Parallel Case (PARALLEL_CASE)

Verilog-2001 Attributes Syntax

Verilog-2001 attributes are bounded by the asterisk character (*), and use the following syntax:

(* *attribute_name* = *attribute_value* *)

where

- The *attribute* precedes the signal, module, or instance declaration to which it refers.
- The *attribute_value* is a string. No integer or scalar values are allowed.
- The *attribute_value* is between quotes.
- The default is **1**.
- (*** attribute_name ***) is the same as (*** attribute_name = "1" ***).

Verilog-2001 Attributes Syntax Example One

```
(* clock_buffer = "IBUFG" *) input CLK;
```

Verilog-2001 Attributes Syntax Example Two

```
(* INIT = "0000" *) reg [3:0] d_out;
```

Verilog-2001 Attributes Syntax Example Three

```
always@(current_state or reset) begin (* parallel_case *) (* full_case *) case
(current_state) ...
```

Verilog-2001 Attributes Syntax Example Four

```
(* mult_style = "pipe_lut" *) MULT my_mult (a, b, c);
```

Verilog-2001 Limitations

Verilog-2001 attributes are not supported for:

- Signal declarations
- Statements
- Port connections
- Expression operators

Verilog-2001 Meta Comments

Constraints can also be specified in Verilog code using meta comments. The Verilog-2001 format is the preferred syntax, but the meta comment style is still supported. Use the following syntax:

```
// synthesis attribute AttributeName [of] ObjectName [is] AttributeValue
```

Verilog-2001 Meta Comments Examples

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
// synthesis attribute bufg of my_clock is "clk"
```

The following constraints use a different syntax:

- Parallel Case (PARALLEL_CASE)
- Full Case (FULL_CASE)
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)

For more information, see Verilog Attributes and Meta Comments.

XST Constraint File (XCF)

XST constraints can be specified in the XST Constraint File (XCF). The XCF has an extension of `.xcf`. For information on specifying the XCF in ISE® Design Suite, see the ISE Design Suite Help.

To specify the XCF in command line mode, use Synthesis Constraint File (-uc) with the **run** command. For more information about the **run** command and running XST from the command line, see XST Command Line Mode.

XST Constraint File (XCF) Syntax and Utilization

The XST Constraint File (XCF) syntax enables you to specify a specific constraint for:

- The entire device (globally), or
- Specific modules

The XCF syntax is basically the same as the User Constraints File (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. Use the keyword **`MODEL`** to define the entity or module to which the constraint is applied. If a constraint is applied to an entity or module, the constraint is applied to each instance of the entity or module.

Define constraints in ISE® Design Suite in **Process > Properties**, or the XST run script, if running on the command line. Specify exceptions in the XCF file. 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 or module use the following syntax:

```
MODEL entityname constraintname = constraintvalue ;
```

XST Constraint File (XCF) Example One

```
MODEL top mux_extract = false;
MODEL my_design max_fanout = 256;
```

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 or module, use the **INST** or **NET** keywords. XST does not support constraints that are applied to VHDL variables.

```
BEGIN MODEL entityname   INST instancename constraintname = constraintvalue ;   NET
signalname constraintname = constraintvalue ; END;
```

XST Constraint File (XCF) Example Two

```
BEGIN MODEL crc32
  INST stopwatch opt_mode = area ;
  INST U2 ram_style = block ;
  NET myclock clock_buffer = true ;
  NET data_in iob = true ;
END;
```

For a complete list of XST synthesis constraints, see XST Specific Non-Timing Options.

Native and Non-Native User Constraint File (UCF) Constraints Syntax

All constraints supported by XST can be divided into two groups:

- Native User Constraints File (UCF) Constraints
- Non-Native User Constraints File (UCF) Constraints

Native User Constraints File (UCF) Constraints

Only Timing and Area Group constraints use native User Constraints File (UCF) syntax.

Use native UCF syntax, including wildcards and hierarchical names, for native UCF constraints such as:

- Period (PERIOD)
- Offset (OFFSET)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)
- From-To (FROM-TO)

Do not use these constraints inside the **BEGIN MODEL... END** construct. If you do, XST issues an error.

Non-Native User Constraints File (UCF) Constraints

For all non-native User Constraints File (UCF) constraints, use the **MODEL** or **BEGIN MODEL... END;** constructs. This includes:

- Pure XST constraints such as:
  - Automatic FSM Extraction (FSM_EXTRACT)
  - RAM Style (RAM_STYLE)
- Implementation non-timing constraints such as:
  - RLOC
  - Keep (KEEP)

If you specify timing constraints in the XST Constraint File (XCF), Xilinx® recommends that you use a forward slash (/) as a hierarchy separator instead of an underscore (_). For more information, see Hierarchy Separator (-hierarchy_separator).

XST Constraint File (XCF) Syntax Limitations

XST Constraint File (XCF) syntax has the following limitations:

- Nested model statements are not supported.
- 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 User Constraints File (UCF) constraints are supported. For more information, see the *Constraints Guide*.

Constraints Priority

Constraints 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, from highest to lowest:

1. Synthesis Constraint File
2. Hardware Description Language (HDL) file
3. ISE® Design Suite **Process > Properties**, or the command line

XST Specific Non-Timing Options

The following table shows:

- Allowed values for each constraint
- Type of objects to which they can be applied
- Usage restrictions

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.

XST Specific Non-Timing Options

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
BoxType)	primitive black_box user_black_box	entity inst	module inst	model inst (in model)	N/A	N/A
Map Logic on BRAM	yes no	entity	module	model	N/A	N/A
Buffer Type	bufgdll ibufg bufg bufgp ibuf bufr none	signal	signal	net (in model)	N/A	N/A

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
Clock Signal	yes no	primary clock signal	primary clock signal	net (in model)	-bufgce	yes no default: no
Clock Signal	yes no	clock signal	clock signal	clock signal net (in model)	N/A	N/A
Decoder Extraction	yes no	entity signal	entity signal	model net (in model)	-decoder _extract	yes no default: yes
Enumerated Encoding	string containing space-separated binary codes	type	signal	net (in model)	N/A	N/A
Equivalent Register Removal	yes no	entity signal	module signal	model net (in model)	-equivalent _register _removal	yes no default: yes
FSM Encoding Algorithm	auto one-hot compact sequential gray johnson speed1 user	entity signal	module signal	model net (in model)	-fsm _encoding	auto one-hot compact sequential gray johnson speed1 user default: auto
Automatic FSM Extraction	yes no	entity signal	module signal	model net (in model)	-fsm _extract	yes no default: yes
FSM Style	lut bram	entity signal	module signal	model net (in model)	-fsm _style	lut bram default: lut
Full Case	N/A	N/A	case statement	N/A	N/A	N/A
Pack I/O Registers Into IOBs	true false auto	signal instance	signal instance	net (in model) inst (in model)	-iob	true false auto default: auto

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
I/O Standard	string For more information, see the *Constraints Guide*	signal instance	signal instance	net (in model) inst (in model)	N/A	N/A
Keep	true false soft	signal	signal	net (in model)	N/A	N/A
Keep Hierarchy	yes no soft	entity	module	model	-keep _hierarchy	yes no soft default (FPGA): no default (CPLD): yes
LOC	string	signal (primary IO) instance	signal (primary IO) instance	net (in model) inst (in model)	N/A	N/A
Map Entity on a Single LUT	yes no	entity architecture	module	model	N/A	N/A
Max Fanout	integer	entity signal	module signal	model net (in model)	-max _fanout	integer default: see detailed description
Move First Stage	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (in model)	-move _first _stage	yes no default: yes
Move Last Stage	yes no	entity primary clock signal	module primary clock signal	model primary clock signal net (in model	-move _last _stage	yes no default: yes

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
Multiplier Style	auto block pipe_block kcm csd lut pipe_lut	entity signal	module signal	model net (in model)	-mult _style	auto block pipe_block kcm csd lut pipe_lut default: auto
Mux Extraction	yes no force	entity signal	module signal	model net (in model)	-mux _extract	yes no force default: yes
Mux Style	auto muxf muxcy	entity signal	module signal	model net (in model)	-mux _style	auto muxf muxcy default: auto
No Reduce	yes no	signal	signal	net (in model)	N/A	N/A
Optimization Effort	1 2	entity	module	model	-opt _level	1 2 default: 1
Optimization Goal	speed area	entity	module	model	-opt _mode	speed area default: speed
Optimize Instantiated Primitives	yes no	entity instance	module instance	model instance (in model)	-optimize _primitives	yes no default: no
Parallel Case	N/A	N/A	case statement	N/A	N/A	N/A
Power Reduction	yes no	entity	module	model	-power	yes no default: no
Priority Encoder Extraction	yes no force	entity signal	module signal	model net (in model)	-priority _extract	yes no force default: yes

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
RAM Extraction	yes no	entity signal	module signal	model net (in model)	-ram _extract	yes no default: yes
RAM Style	auto block distributed pipe_distributed block_power1 block_power2	entity signal	module signal	model net (in model)	-ram _style	auto block distributed default: auto
Read Cores	yes no optimize	entity component	module label	model inst (in model)	-read _cores	yes no optimize default: yes
Register Balancing	yes no forward backward	entity signal FF instance name	module signal FF instance name primary clock signal	modelnet (in model)inst (in model)	-register _balancing	yes no forward backward default: no
Register Duplication	yes no	entity signal	module	model net (in model)	-register _duplication	yes no default: yes
Register Power Up	string	type	signal	net (in model)	N/A	N/A
Resource Sharing	yes no	entity signal	module signal	model net (in model)	-resource _sharing	yes no default: yes
ROM Extraction	yes no	entity signal	module signal	model net (in model)	-rom _extract	yes no default: yes
ROM Style	auto block distributed	entity signal	module signal	model net (in model)	-rom _style	auto block distributed default: auto
Save	yes no	signal inst of primitive	signal inst of primitive	net (in model) inst of primitive (in model)	N/A	N/A

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
Safe Implementation	yes no	entity signal	module signal	model net (in model)	-safe _implementation	yes no default: no
Safe Recovery State	string	signal	signal	net (in model)	N/A	N/A
Logical Shifter Extraction	yes no	entity signal	module signal	model net (in model)	-shift _extract	yes no default: yes
Shift Register Extraction	yes no	entity signal	module signal	model net (in model)	-shreg extract	yes no default: yes
Signal Encoding	auto one-hot user	entity signal	module signal	model net (in model)	-signal _encoding	auto one-hot user default: auto
Slice Utilization Ratio	integer (range -1 to 100) integer% (range -1 to 100) integer#	entity	module	model	-slice _utilization _ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# default: 100
Slice Utilization Ratio Delta	integer (range 0 to 100) integer% (range 0 to 100) integer#	entity	module	model	-slice _utilization _ratio _maxmargin	integer (range 0 to 100) integer% (range 0 to 100) integer# default: 0
Translate Off Translate On	N/A	local no target	local no target	N/A	N/A	N/A
Convert Tristates to Logic	yes no	entity signal	modulesignal	model net (in model)	-tristate2logic	yes no default: yes
Use Carry Chain	yes no	entity signal	module signal	model net (in model)	-use _carry _chain	yes no default: yes

Constraint Name	Constraint Value	VHDL Target	Verilog Target	XCF Target	Command Line	Command Value
Use Clock Enable	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _clock _enable	auto yes no default: auto
Use DSP48	auto yes no	entity signal	module signal	model net (in model)	-use _dsp48	auto yes no default: auto
Use Synchronous Reset	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _sync _reset	auto yes no default: auto
Use Synchronous Set	auto yes no	entity signal FF instance name	module signal FF instance name	model net (in model) inst (in model)	-use _sync _set	auto yes no default: auto
XOR Collapsing	yes no	entity signal	module signal	model net (in model)	-xor _collapse	yes no default: yes

XST Command Line Only Options

XST Specific Non-Timing Options: XST Command Line Only

Constraint Name	Command Line	Command Value
VHDL Top Level Architecture	-arch	architecture_name default: N/A
Asynchronous to Synchronous	-async_to_sync	yes no default: no
Automatic BRAM Packing	-auto_bram_packing	yes no default: no

Constraint Name	Command Line	Command Value
BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)	-bram_utilization_ ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# default: 100
Maximum Global Clock Buffers	-bufg	Integer default: max number of buffers in target device
Maximum Regional Clock Buffers	-bufr	Integer default: max number of buffers in target device
Bus Delimiter	-bus_delimiter	< > [ ] { } () default: <>
Case	-case	upper lower maintain default: maintain
Verilog Macros	-define	{name = value} default: N/A
DSP Utilization Ratio (DSP_UTILIZATION_RATIO)	-dsp_utilization_ratio	integer (range -1 to 100) integer% (range -1 to 100) integer# default: 100
Duplication suffix	-duplication_suffix	string%dstring default: _%d
VHDL Top-Level block (Valid only when old VHDL project format is used (-ifmt VHDL). Use project format (-ifmt mixed) and - top option to specify which top level block to synthesize.)	-ent	entity_name default: N/A
Generics	-generics	{name = value} default: N/A
HDL File Compilation Order	-hdl_compilation_order	auto user default: auto

Constraint Name	Command Line	Command Value
Hierarchy Separator	-hierarchy_separator	_ / default: /
Input Format	-ifmt	mixed vhdl verilog default: mixed
Input/Project File Name	-ifn	file_name default: N/A
Add I/O Buffers	-iobuf	yes no default: yes
Ignore User Constraints	-iuc	yes no default: no
Library Search Order	-lso	file_name.lso default: N/A
LUT Combining	-lc	auto area off default: off
Netlist Hierarchy	-netlist_hierarchy	as_optimized rebuilt default: as_optimized
Output File Format	-ofmt	ngc default: ngc
Output File Name	-ofn	file_name default: N/A
Target Device	-p	part-package-speed (For example: xc5vfx30t-ff324-2) default: N/A
Clock Enable	-pld_ce	yes no default: yes
Macro Preserve	-pld_mp	yes no default: yes

Constraint Name	Command Line	Command Value
XOR Preserve	-pld_xp	yes no default: yes
Reduce Control Sets	-reduce_control_sets	auto no default: no
Generate RTL Schematic	-rtlview	yes no only default: no
Cores Search Directories	-sd	directories default: N/A
Slice Packing	-slice_packing	yes no default: yes
Top Level Block	-top	block_name default: N/A
Synthesis Constraints File	-uc	file_name.xcf default: N/A
Verilog 2001	-verilog2001	yes no default: yes
Case Implementation Style	-vlgcase	full parallel full-parallel default: N/A
Verilog Include Directories	-vlgincdir	directories default: N/A
Work Library	-work_lib	directory default: work
wysiwyg	-wysiwyg	yes no default: no
Work Directory	-xsthdpdir	Directory default: ./xst
HDL Library Mapping File	-xsthdpini	file_name.ini default: N/A

XST Timing Options

You can invoke XST timing options from:

- ISE® Design Suite **Process > Properties**
- The command line
- The XST Constraint File (XCF)

XST Timing Options: Process > Properties or Command Line

The following table shows the XST timing constraints that you can invoke only from ISE® Design Suite **Process > Properties**, or from the command line.

XST Timing Constraints Supported Only in Process > Properties, or Command Line

Option	Process Property (ISE Design Suite)	Values
glob_opt	Global Optimization Goal	allclocknetsinpad _to_outpadoffset _in_beforeoffset _out_aftermax _delay default: allclocknets
cross_clock_analysis	Cross Clock Analysis	yes no default: no
write_timing_constraints	Write Timing Constraints	yes no default: no

XST Timing Options: XST Constraint File (XCF)

The following XST timing constraints can be applied for synthesis only through the XST Constraint File (XCF):

- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)
- Timing Specifications (TIMESPEC)
- Timing Specification Identifier (TSidentifier)

These timing constraints influence synthesis optimization, and can be passed on to place and route by selecting the Write Timing Constraints command line option.

For more information as to the Value and Target of each constraint, see the *Constraints Guide*.

XST General Constraints

The following general constraints apply to FPGA devices, CPLD devices, VHDL, and Verilog. You can set some of these options in ISE® Design Suite in **Process > Properties > Synthesis Options**.

- Add I/O Buffers (-iobuf)
- BoxType (BOX_TYPE)
- Bus Delimiter (-bus_delimiter)
- Case (-case)
- Case Implementation Style (-vlgcase)
- Verilog Macros (-define)
- Duplication Suffix (-duplication_suffix)
- Full Case (FULL_CASE)
- Generate RTL Schematic (-rtlview)
- Generics (-generics)
- Hierarchy Separator (-hierarchy_separator)
- I/O Standard (IOSTANDARD)
- Keep (KEEP)
- Keep Hierarchy (KEEP_HIERARCHY)
- Library Search Order (-lso)
- LOC
- Netlist Hierarchy (-netlist_hierarchy)
- Optimization Effort (OPT_LEVEL)
- Optimization Goal (OPT_MODE)
- Parallel Case (PARALLEL_CASE)
- RLOC
- Save (S / SAVE)
- Synthesis Constraint File (-uc)
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)
- Use Synthesis Constraints File (-iuc)
- Verilog Include Directories (-vlgincdir)
- Verilog 2001 (-verilog2001)
- HDL Library Mapping File (-xsthdpini)
- Work Directory (-xsthdpdir)

-iobuf (Add I/O Buffers)

`-iobuf` (Add I/O Buffers) enables or disables I/O buffer insertion. XST automatically inserts Input/Output Buffers into the design. If you manually instantiate I/O Buffers for some or all the I/Os, XST inserts I/O Buffers only for the remaining I/Os. If you do not want XST to insert I/O Buffers, set **`-iobuf`** to **`no`**. Add I/O Buffers is useful to synthesize a part of a design to be instantiated later on.

You can set this value in ISE® Design Suite by setting **Process > Properties > Xilinx®-Specific Options > Add I/O Buffers**.

-iobuf (Add I/O Buffers) Architecture Support

Architecture independent.

-iobuf (Add I/O Buffers) Applicable Elements

Applies to the entire design.

-iobuf (Add I/O Buffers) Propagation Rules

Applies to design primary IOs.

-iobuf (Add I/O Buffers) Syntax

`-iobuf` `{yes|no|true|false|soft}`

Allowed values are:

- **`yes`** (the default) tells XST to generate IBUF and OBUF primitives and connected them to I/O ports of the top-level module.
- **`no`** tells XST not to generate IBUF and OBUF primitives, and must be used when XST is called to synthesize an internal module that is instantiated later in a larger design. If I/O buffers are added to a design, this design cannot be used as a submodule of another design.
- **`true`**
- **`false`**
- **`soft`**

-iobuf (Add I/O Buffers) Syntax Example

```
xst run -iobuf yes
```

This command line example adds I/O buffers to the top level module of the design.

BOX_TYPE (BoxType)

BOX_TYPE (BoxType) is a synthesis constraint.

BOX_TYPE values are:

- **`primitive`**
- **`black_box`**
- **`user_black_box`**

These values instruct XST not to synthesize the behavior of a module.

The **`black_box`** value is equivalent to **`primitive.`** It will eventually become obsolete.

If **`user_black_box`** is specified, XST reports inference of a black box in the log file. It does not do so if **`primitive`** is specified.

If BOX_TYPE is applied to at least a single instance of a block of a design, BOX_TYPE is propagated to all other instances of the entire design. This feature was implemented for Verilog and XST Constraint File (XCF) in order to have a VHDL-like support, where BOX_TYPE can be applied to a component.

BOX_TYPE (BoxType) Architecture Support

Architecture independent.

BOX_TYPE (BoxType) Applicable Elements

Applies to the following design elements:

- VHDL
  component, entity
- Verilog
  module, instance
- XST Constraint File (XCF)
  model, instance

BOX_TYPE (BoxType) Propagation Rules

Applies to the design element to which it is attached.

BOX_TYPE (BoxType) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

BOX_TYPE (BoxType) VHDL Syntax Example

Declare as follows:

```
attribute box_type: string;
```

Specify as follows:

attribute box_type of {*component_name* | *entity_name* }: **{component | entity} is** "{*primitive* | *black_box* | *user_black_box*}";

BOX_TYPE (BoxType) Verilog Syntax Example

Place immediately before the instantiation:

(* **box_type =** "*{primitive | black_box | user_black_box}*" *)

BOX_TYPE (BoxType) XST Constraint File (XCF) Syntax Example One

MODEL "*entity_name*" **box_type**="{*primitive* |*black_box* | *user_black_box*}";

BOX_TYPE (BoxType) XST Constraint File (XCF) Syntax Example Two

BEGIN MODEL "*entity_name*"

INST "*instance_name*"

box_type="{*primitive* | *black_box* | *user_black_box*}";

END;

-bus_delimiter (Bus Delimiter)

`-bus_delimiter` (Bus Delimiter) vectors in the result netlist.

You can also set this value in ISE® Design Suite by setting **Process > Properties > Synthesis Options > Bus Delimiter**.

-bus_delimiter (Bus Delimiter) Architecture Support

Architecture independent.

-bus_delimiter (Bus Delimiter) Applicable Elements

Applies to syntax.

-bus_delimiter (Bus Delimiter) Propagation Rules

Not applicable.

-bus_delimiter (Bus Delimiter) Syntax

`-bus_delimiter {<> |[] |{} |() }`

The default delimiter is **<>**.

-bus_delimiter (Bus Delimiter) Syntax Example

`xst run -bus_delimiter []`

This example defines bus delimiters globally as square braces ([ ])

-case (Case)

`-case` (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. The case can be maintained for either Verilog or VHDL synthesis flow.

You can also set this value in ISE® Design Suite with **Process > Properties > Synthesis Options > Case**.

-case (Case) Architecture Support

Architecture independent.

-case (Case) Applicable Elements

Applies to syntax.

-case (Case) Propagation Rules

Not applicable.

-case (Case) Syntax

`-case {upper | lower |maintain }`

The default value is **`maintain`**

-case (Case) Syntax Example

`xst run -case upper`

Defines case globally to upper case.

-vlgcase (Case Implementation Style)

`-vlgcase` (Case Implementation Style) is valid for Verilog designs only.

`-vlgcase` instructs XST how to interpret Verilog Case statements. It has three possible values.

- **`full`**
  XST assumes that the case statements are complete, and avoids latch creation.
- **`parallel`**
  XST assumes that the branches cannot occur in parallel, and does not use a priority encoder.
- **`full-parallel`**
  XST assumes that the case statements are complete, and that the branches cannot occur in parallel, therefore saving latches and priority encoders.
- If the option is not specified, XST implements the exact behavior of the case statements.

-vlgcase (Case Implementation Style) Architecture Support

Architecture independent.

-vlgcase (Case Implementation Style) Applicable Elements

Applies to the entire design.

-vlgcase (Case Implementation Style) Propagation Rules

Not applicable.

For more information, see:

- Multiplexers Hardware Description Language (HDL) Coding Techniques
- FULL_CASE (Full Case)
- PARALLEL_CASE (Parallel Case).

You can also set this value in ISE® Design Suite with **Process > Properties > HDL Options > Case Implementation Style**.

-vlgcase (Case Implementation Style) Syntax

```
-vlgcase {full |parallel |full-parallel }
```

By default, there is no value.

-vlgcase (Case Implementation Style) Syntax Example

```
xst run -vlgcase full
```

Defines Case Implementation Style globally to parallel.

-define (Verilog Macros)

-define (Verilog Macros allows you to define (or redefine) Verilog macros. This allows you to easily modify the design configuration without any Hardware Description Language (HDL) source modifications, such as for IP core generation and testing flows. If the defined macro is not used in the design, no message is given.

-define is valid for Verilog designs only.

To define Verilog macros in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the **Verilog Macros** property.

Do not use curly braces (**{...}**) when specifying macros.

-define (Verilog Macros) Architecture Support

Architecture independent.

-define (Verilog Macros) Applicable Elements

Applies to the entire design.

-define (Verilog Macros) Propagation Rules

Not applicable.

-define (Verilog Macros) Syntax

```
-define {name[=value] name[=value]}
```

- *name* is a macro name
- *value* is the macro text

The default is an empty definition.

```
-define {}
```

Note

- Values for macros are not mandatory.
- Place the values inside curly braces (**{...}**).
- Separate the values with spaces.
- Macro text can be specified between quotation marks (**"..."**), or without them. If the macro text contains spaces, you must use quotation marks (**"..."**).

```
-define {macro1=Xilinx macro2="Xilinx Virtex4"}
```

-define (Verilog Macros) Syntax Example

```
xst run -define macro1=Xilinx macro2="Xilinx Virtex4"
```

Defines two macros named **macro1** and **macro2**.

-duplication_suffix (Duplication Suffix)

-duplication_suffix (Duplication Suffix) 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 instance, 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**
- **my_ff_3**

-duplication_suffix lets you change the string that is added to the original name.

To set this value in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the **Other XST Command Line Options** property.

-duplication_suffix (Duplication Suffix) Architecture Support

Architecture independent.

-duplication_suffix (Duplication Suffix) Architecture Applicable Elements

Applies to files.

-duplication_suffix (Duplication Suffix) Architecture Propagation Rules

Not applicable.

-duplication_suffix (Duplication Suffix) Architecture Syntax

-duplication_suffix *string*%d*string*

The default is **%d**.

-duplication_suffix (Duplication Suffix) Architecture Syntax Example One

```
xst run –duplication_suffix _dupreg_%d
```

If the flip-flop named **my_ff** is duplicated three times, this command tells XST to generate the following names:

- **my_ff_dupreg_1**
- **my_ff_dupreg_2**
- **my_ff_dupreg_3**

-duplication_suffix (Duplication Suffix) Architecture Syntax Example Two

```
xst run –duplication_suffix _dup_%d_reg
```

The **%d** escape character can be placed anywhere in the suffix definition. If the flip-flop named **my_ff** is duplicated three times, this command tells XST to generate the following names:

- **my_ff_dup_1_reg**
- **my_ff_dup_2_reg**
- **my_ff_dup_3_reg**

FULL_CASE (Full Case)

FULL_CASE (Full Case) is valid for Verilog designs only. FULL_CASE indicates that all possible selector values have been expressed in a **case**, **casex** or **casez** statement. The FULL_CASE directive prevents XST from creating additional hardware for those conditions not expressed. For more information, see Multiplexers HDL Coding Techniques.

FULL_CASE (Full Case) Architecture Support

Architecture independent.

FULL_CASE (Full Case) Applicable Elements

Applies to **case** statements in Verilog meta comments.

FULL_CASE (Full Case) Propagation Rules

Not applicable.

FULL_CASE (Full Case) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

FULL_CASE (Full Case) Verilog Syntax Example

The syntax is:

```
(* full_case *)
```

Since FULL_CASE does not contain a target reference, the attribute immediately precedes the selector:

```
(* full_case *)
casex select
4’b1xxx: res = data1;
4’bx1xx: res = data2;
4’bxx1x: res = data3;
4’bxxx1: res = data4;
endcase
```

FULL_CASE is also available as a meta comment in the Verilog code. The syntax differs from the standard meta comment syntax as shown in the following:

```
// synthesis full_case
```

Since FULL_CASE does not contain a target reference, the meta comment immediately follows the selector:

```
casex select // synthesis full_case
4’b1xxx: res = data1;
4’bx1xx: res = data2;
4’bxx1x: res = data3;
4’bxxx1: res = data4;
endcase
```

FULL_CASE (Full Case) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-vlgcase [full|parallel|full-parallel]
```

FULL_CASE (Full Case) ISE Design Suite Syntax Example

Note For Verilog files only.

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Full Case.

For Case Implementation Style, select **Full** as a Value.

-rtlview (Generate RTL Schematic)

-rtlview (Generate RTL Schematic) tells XST to generate a netlist file representing a Register Transfer Level (RTL) structure of the design. This netlist can be viewed by the RTL and Technology Viewers. **-rtlview** has three possible values:

- *yes* tells XST to generate an RTL view.
- *no* tells XST not to generate the RTL view.
- *only* tells XST to stop the synthesis once the RTL view is generated.

The file containing the RTL view has an NGR file extension.

You can also set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Generate RTL Schematic**.

-rtlview (Generate RTL Schematic) Architecture Support

Architecture independent.

-rtlview (Generate RTL Schematic) Applicable Elements

Applies to files.

-rtlview (Generate RTL Schematic) Propagation Rules

Not applicable.

-rtlview (Generate RTL Schematic) Syntax

```
-rtlview {yes|no|only}
```

The default is **no**.

-rtlview (Generate RTL Schematic) Syntax Example

```
-rtlview yes
```

Tells XST to generate a netlist file representing the RTL structure of the design.

-generics (Generics)

-generics (Generics) allows you to redefine generics (VHDL) or parameters (Verilog) values defined in the top-level design block. This allows you to easily modify the design configuration without any Hardware Description Language (HDL) source modifications, such as for IP core generation and testing flows. If the defined value does not correspond to the data type defined in the VHDL or Verilog code, then XST tries to detect the situation and issues a warning, ignoring the command line definition.

In some situations, XST may fail to detect a type mismatch. In that case, XST attempts to apply this value by adopting it to the type defined in the VHDL or Verilog file without any warning. Be sure that the value you specified corresponds to the type defined in the VHDL or Verilog code. If a defined generic or parameter name does not exist in the design, no message is given, and the definition is ignored.

You can also set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Generics, Parameters**.

-generics (Generics) Architecture Support

Architecture independent.

generics (Generics) Applicable Elements

Applies to the entire design.

generics (Generics) Propagation Rules

Not applicable.

-generics (Generics) Syntax

`xst run -generics {`*`name=value name=value ...`***`}`**

name is the name of a generic or parameter of the top level design block

value is the value of a generic or parameter of the top level design block

The default is an empty definition.

`-generics {}`

Follow these rules:

- Place the values inside curly braces (**`{...}`**).
- Separate the values with spaces.
- XST can accept as values only constants of scalar types. Composite data types (arrays or records) are supported only in the following situations:
  - **`string`**
  - **`std_logic_vector`**
  - **`std_ulogic_vector`**
  - **`signed, unsigned`**
  - **`bit_vector`**
- There are no spaces between the prefix and the corresponding value:

-generics (Generics) syntax Example

```
-generics {company="Xilinx" width=5 init_vector=b100101}
```

This command sets *company* to *Xilinx®*, *width* to *5*, and *init_vector* to *b100101*.

-hierarchy_separator (Hierarchy Separator)

`-hierarchy_separator` (Hierarchy Separator) defines the hierarchy separator character that is used in name generation when the design hierarchy is flattened.

The two supported characters are:

- _ (underscore)
- **`/`** (forward slash)

The default is **`/`** (forward slash) 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 **`/`** (forward slash). The name TMP_NET becomes INST1_TMP_NET. If the hierarchy separator character is **`/`** (forward slash), the net name is INST1/TMP_NET.

Using **`/`** (forward slash) as a hierarchy separator is useful in design debugging because the **`/`** (forward slash) separator makes it much easier to identify a name if it is hierarchical.

To specify the hierarchy separator in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the Hierarchy Separator property.

-hierarchy_separator (Hierarchy Separator) Architecture Support

Architecture independent.

-hierarchy_separator (Hierarchy Separator) Applicable Elements

Applies to files.

-hierarchy_separator (Hierarchy Separator) Propagation Rules

Not applicable.

-hierarchy_separator (Hierarchy Separator) Syntax

```
-hierarchy_separator {/|_}
```

The default is **/** (forward slash) for newly created projects.

-hierarchy_separator (Hierarchy Separator) Syntax Example

```
xst run -hierarchy_separator _
```

Sets the hierarchy separator to "_" (underscore)

IOSTANDARD (I/O Standard)

Use I/O Standard (IOSTANDARD) to assign an I/O standard to an I/O primitive. For more information, see IOSTANDARD in the *Constraints Guide*.

KEEP (Keep)

KEEP (Keep) 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.

In addition to **true** and **false** values supported by the implementation flow, XST supports a **soft** value. If this value is specified XST preserves the designated net as in the case of the **true** value, but does not attach the KEEP constraint in the final netlist to this net.

KEEP preserves the existence of the signal in the final netlist, but not its structure. For example, if your design has a 2-bit multiplexer selector and you attach KEEP to it, this signal is preserved in the final netlist. But the multiplexer could be automatically re-encoded by XST using one-hot encoding. As a consequence, this signal in the final netlist is four bits wide instead of the original two. To preserve the structure of the signal, in addition to KEEP, you must also use Enumerated Encoding (ENUM_ENCODING)'

For more information, see KEEP in the *Constraints Guide*.

KEEP_HIERARCHY (Keep Hierarchy)

KEEP_HIERARCHY (Keep Hierarchy) is a synthesis and implementation constraint. If hierarchy is maintained during synthesis, the implementation tools use Keep Hierarchy to preserve the hierarchy throughout implementation, and allow a simulation netlist to be created with the desired hierarchy.

XST can flatten the design to obtain better results by optimizing entity or module boundaries. You can set Keep Hierarchy to **true** so that the generated netlist is hierarchical and respects the hierarchy and interface of any entity or module in your design.

Keep Hierarchy is related to the hierarchical blocks (VHDL entities, Verilog modules) specified in the Hardware Description Language (HDL) design, and does not concern the macros inferred by the HDL synthesizer.

Keep Hierarchy values are:

- **`true`**

  Allows the preservation of the design hierarchy, as described in the HDL project. If this value is applied to synthesis, it is also propagated to implementation.

  The default is **`true`** for CPLD devices.

- **`false`**

  Hierarchical blocks are merged in the top level module.

  The default is **`false`** for FPGA devices.

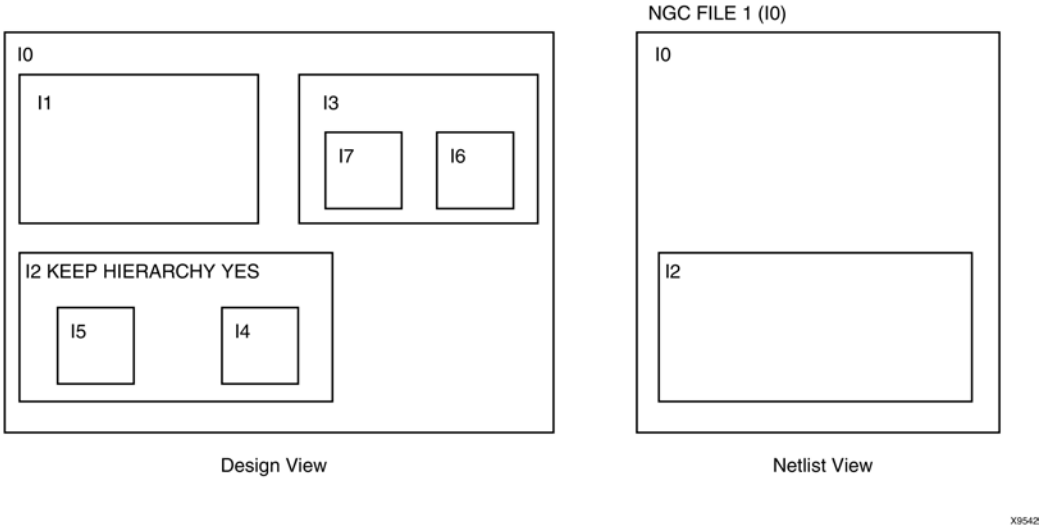
- **`soft`**

  Allows the preservation of the design hierarchy in synthesis, but KEEP_HIERARCHY is not propagated to implementation.

In general, a Hardware Description Language (HDL) design is a collection of hierarchical blocks. Preserving the hierarchy gives the advantage of fast processing because the optimization is done on separate pieces of reduced complexity. Nevertheless, very often, merging the hierarchy blocks improves the fitting results (fewer PTerms and device macrocells, better frequency) because the optimization processes (collapsing, factorization) are applied globally on the entire logic.

In the following figure, if Keep Hierarchy is set to the entity or module I2, the hierarchy of I2 is in the final netlist, but its contents I4, I5 are flattened inside I2. I1, I3, I6, and I7 are also flattened.

KEEP_HIERARCHY (Keep Hierarchy) Diagram



KEEP_HIERARCHY (Keep Hierarchy) Architecture Support

Architecture independent.

KEEP_HIERARCHY (Keep Hierarchy) Applicable Elements

Applies to logical blocks, including blocks of hierarchy or symbols.

KEEP_HIERARCHY (Keep Hierarchy) Propagation Rules

Applies to the entity or module to which it is attached.

KEEP_HIERARCHY (Keep Hierarchy) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

KEEP_HIERARCHY (Keep Hierarchy) Schematic Syntax Example

- Attach to the entity or module symbol.
- Attribute Name: KEEP_HIERARCHY
- Attribute Values: YES, NO

KEEP_HIERARCHY (Keep Hierarchy) VHDL Syntax Example

Declare as follows:

```
attribute keep_hierarchy : string;
```

Specify as follows:

```
attribute keep_hierarchy of architecture_name : architecture is "{yes|no|true|false|soft}";
```

The default is **no** for FPGA devices.

The default is **yes** for CPLD devices.

KEEP_HIERARCHY (Keep Hierarchy) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* keep_hierarchy = "{yes|no|true|false|soft}" *)
```

KEEP_HIERARCHY (Keep Hierarchy) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name" keep_hierarchy={yes|no|true|false|soft};
```

KEEP_HIERARCHY (Keep Hierarchy) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-keep_hierarchy {yes|no|soft}
```

The default is **no** for FPGA devices.

The default is **yes** for CPLD devices.

For more information, see XST Command Line Mode.

KEEP_HIERARCHY (Keep Hierarchy) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Keep Hierarchy.

-lso (Library Search Order)

-lso (Library Search Order) specifies the location of the library search order file.

For more information about the LSO file, see Library Search Order (LSO) Files in Mixed Language Projects.

To specify the library search order file in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the **Library Search Order** property.

-lso (Library Search Order) Architecture Support

Architecture independent.

-lso (Library Search Order) Applicable Elements

Applies to files.

-lso (Library Search Order) Propagation Rules

Not applicable.

-lso (Library Search Order) Syntax

`-lso` *`file_name`*`.lso`

There is no default file name. If not specified, XST uses the default search order.

-lso (Library Search Order) Syntax Example

```
xst elaborate -lso c:/data/my_libraries/my.lso
```

Specifies `c:/data/my_libraries/my.lso` as the file that sets your library search order.

LOC

The LOC constraint defines where a design element can be placed within an FPGA or CPLD device. For more information, see LOC in the in the

-netlist_hierarchy (Netlist Hierarchy)

Use **`-netlist_hierarchy`** (Netlist Hierarchy) to control the form in which the final NGC netlist is generated. Netlist Hierarchy allows you to write the hierarchical netlist even if the optimization was done on a partially or fully flattened design.

If the value of Netlist Hierarchy is:

- **`as_optimized`**

  XST takes into account the Keep Hierarchy (KEEP_HIERARCHY) constraint, and generates the NGC netlist in the form in which it was optimized. In this mode, some hierarchical blocks can be flattened, and some can maintain hierarchy boundaries.

- **`rebuilt`**

  XST writes a hierarchical NGC netlist, regardless of the “Keep Hierarchy (KEEP_HIERARCHY) constraint.

To set this option in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the Netlist Hierarchy property.

-netlist_hierarchy (Netlist Hierarchy) Syntax

`-netlist_hierarchy` `{as_optimized|rebuilt}`

The default is **`as_optimized`**.

-netlist_hierarchy (Netlist Hierarchy) Syntax Example

```
-netlist_hierarchy rebuilt
```

XST writes a hierarchical NGC netlist regardless of the KEEP_HIERARCHY constraint.

OPT_LEVEL (Optimization Effort)

OPT_LEVEL (Optimization Effort) defines the synthesis optimization effort level.

Allowed OPT_LEVEL values are:

- **1** (normal optimization)

  Use **1** (normal optimization) for very fast processing, especially for hierarchical designs. In speed optimization mode, Xilinx® recommends using **1** (normal optimization) for the majority of designs. 1 (normal optimization) is the default.

- **2** (higher optimization)

  While **2** (higher optimization) is more time consuming, it sometimes gives better results in the number of slices/macrocells or maximum frequency. Selecting **2** (higher optimization) usually results in increased synthesis run times, and does not always bring optimization gain.

OPT_LEVEL (Optimization Effort) Architecture Support

Architecture independent.

OPT_LEVEL (Optimization Effort) Applicable Elements

Applies to the entire design, or to an entity or module.

OPT_LEVEL (Optimization Effort) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

OPT_LEVEL (Optimization Effort) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

OPT_LEVEL (Optimization Effort) VHDL Syntax Example

Declare as follows:

```
attribute opt_level: string;
```

Specify as follows:

```
attribute opt_level of entity_name: entity   is "{1 | 2}";
```

OPT_LEVEL (Optimization Effort) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* opt_level = "{1 | 2}" *)
```

OPT_LEVEL (Optimization Effort) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name" opt_level={1 | 2};
```

OPT_LEVEL (Optimization Effort) XST Command Line Syntax Example

Define globally with the **-opt_level** command line option:

```
-opt_level {1 | 2}
```

The default is **1**.

OPT_LEVEL (Optimization Effort) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Optimization Effort.

OPT_MODE (Optimization Goal)

OPT_MODE (Optimization Goal) defines the synthesis optimization strategy.

Available Optimization Goal values are:

- **speed**

  The priority of **speed** is to reduce the number of logic levels and therefore to increase frequency. **speed** is the default.

- **area**

  The priority of **area** is to reduce the total amount of logic used for design implementation and therefore improve design fitting.

OPT_MODE (Optimization Goal) Architecture Support

Architecture independent.

OPT_MODE (Optimization Goal) Applicable Elements

Applies to the entire design, or to an entity or module.

OPT_MODE (Optimization Goal) Propagation Rules

Applies to the entity or module to which it is attached.

OPT_MODE (Optimization Goal) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

OPT_MODE (Optimization Goal) VHDL Syntax Example

Declare as follows:

```
attribute opt_mode: string;
```

Specify as follows:

```
attribute opt_mode of entity_name: entity is "{speed | area}";
```

OPT_MODE (Optimization Goal) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* opt_mode = "{speed | area}" *)
```

OPT_MODE (Optimization Goal) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name"opt_mode={speed | area};
```

OPT_MODE (Optimization Goal) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run-opt_mode {area | speed}
```

The default is **speed**.

OPT_MODE (Optimization Goal) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Optimization Goal.

PARALLEL_CASE (Parallel Case)

PARALLEL_CASE (Parallel Case) is valid for Verilog designs only. PARALLEL_CASE forces a case statement to be synthesized as a parallel multiplexer and prevents the case statement from being transformed into a prioritized **`if...elsif`** cascade. For more information, see Multiplexers Hardware Description Language (HDL) Coding Techniques.

PARALLEL_CASE (Parallel Case) Architecture Support

Architecture independent.

PARALLEL_CASE (Parallel Case) Applicable Elements

Applies to **`case`** statements in Verilog meta comments only.

PARALLEL_CASE (Parallel Case) Propagation Rules

Not applicable.

PARALLEL_CASE (Parallel Case) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

PARALLEL_CASE (Parallel Case) Verilog Syntax Examples

The syntax is:

`(* parallel_case *)`

Since PARALLEL_CASE does not contain a target reference, the attribute immediately precedes the selector.

```
(* parallel_case *)
casex select
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

PARALLEL_CASE is also available as a meta comment in the Verilog code. The syntax differs from the standard meta comment syntax as shown in the following:

`// synthesis parallel_case`

Since PARALLEL_CASE does not contain a target reference, the meta comment immediately follows the selector:

```
casex select // synthesis parallel_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

PARALLEL_CASE (Parallel Case) XST Command Line Syntax Example

Define in the XST command line as follows:

`xst run -vlgcase {full | parallel | full-parallel}`

RLOC (RLOC)

RLOC (RLOC) is a basic mapping and placement constraint. RLOC 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. For more information, see RLOC in the *Constraints Guide*.

S (Save)

S (Save) is an advanced mapping constraint. When the design is mapped, some nets may be absorbed into logic blocks, and some elements such as LUTs can be optimized away. When a net is absorbed into a block, or a block is optimized away, it can no longer be seen in the physical design database. S (SAVE) prevents this from happening. Several optimization techniques such as nets or blocks replication and register balancing are also disabled by the S (SAVE) constraint.

If S (SAVE) is applied to a net, XST preserves the net with all elements directly connected to it in the final netlist. This includes nets connected to these elements.

If S (SAVE) is applied to a block such as a LUT, XST preserves the LUT with all signals connected to it.

For more information, see the *Constraints Guide*.

-uc (Synthesis Constraint File)

`-uc` (Synthesis Constraint File) specifies a synthesis constraint file for XST to use. The XST Constraint File (XCF) has an extension of `.xcf`. If the extension is not `.xcf`, XST errors out and stops processing. For more information, see XST Constraint File (XCF).

You can also set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Synthesis Constraint File**.

-uc (Synthesis Constraint File) Architecture Support

Architecture independent.

-uc (Synthesis Constraint File) Applicable Elements

Applies to files.

-uc (Synthesis Constraint File) Propagation Rules

Not applicable.

-uc (Synthesis Constraint File) Syntax

The command line syntax is:

```
xst run -uc filename
```

-uc (Synthesis Constraint File) Syntax Example

```
-uc my_constraints.xcf
```

Specifies `my_constraints.xcf` as the constraint file for this project.

TRANSLATE_OFF (Translate Off)

TRANSLATE_OFF (Translate Off) and TRANSLATE_ON (Translate On) instruct XST to ignore portions of VHDL or Verilog code that are not relevant for synthesis, such as simulation code.

- TRANSLATE_OFF marks the beginning of the section to be ignored.
- TRANSLATE_ON instructs XST to resume synthesis from that point.

TRANSLATE_OFF and TRANSLATE_ON are also Synplicity and Synopsys directives that XST supports in Verilog. Automatic conversion is also available in VHDL and Verilog.

TRANSLATE_OFF and TRANSLATE_ON can be used with the following words:

- synthesis
- Synopsys
- pragma

TRANSLATE_OFF (Translate Off) Architecture Support

Architecture independent.

TRANSLATE_OFF (Translate Off) Applicable Elements

Applies locally.

TRANSLATE_OFF (Translate Off) Propagation Rules

Instructs the synthesis tool to enable or disable portions of code

TRANSLATE_OFF (Translate Off) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

TRANSLATE_OFF (Translate Off) VHDL Syntax Example

In VHDL, write TRANSLATE_OFF (Translate Off) as follows:

```
-- synthesis translate_off
...code not synthesized...
-- synthesis translate_on
```

TRANSLATE_OFF (Translate Off) Verilog Syntax Example

TRANSLATE_OFF (Translate Off) is available as VHDL or Verilog meta comments. The Verilog syntax differs from the standard meta comment syntax presented earlier, as shown in the following coding example:

```
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```

-iuc (Use Synthesis Constraints File)

-iuc (Use Synthesis Constraints File) allows you to ignore the constraints file during synthesis.

-iuc (Use Synthesis Constraints File) Architecture Support

Architecture independent.

-iuc (Use Synthesis Constraints File) Applicable Elements

Applies to files.

-iuc (Use Synthesis Constraints File) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite with **Process > Properties > Synthesis Options > Use Synthesis Constraints File**.

-iuc (Use Synthesis Constraints File) Syntax

xst run -iuc {*yes* | *no*}

The default is **no**.

-iuc (Use Synthesis Constraints File) Syntax Example

```
xst run -iuc yes
```

Tells XST to use the constraints file during synthesis.

-vlgincdir (Verilog Include Directories)

`-vlgincdir` (Verilog Include Directories) is used to help the parser find files referenced by **`` `include ``** statements. When an **`` `include ``** statement references a file, XST looks in different areas in this order:

- Relative to the current directory.
- Relative to the inc directories.
- Relative to the current file.

Note **`-vlgincdir`** should be used with **`` `include ``**

-vlgincdir (Verilog Include Directories) Architecture Support

Architecture independent.

-vlgincdir (Verilog Include Directories) Applicable Elements

Applies to directories.

-vlgincdir (Verilog Include Directories) Propagation Rules

Not applicable.

To specify the Verilog include directories in ISE® Design Suite:

1. Select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the **Verilog Include Directories** property.

-vlgincdir (Verilog Include Directories) Syntax

`-vlgincdir` {*directory_path* [*directory_path*]}

directory_path is the name of a directory. For more information, see Names With Spaces in Command Line Mode.

-vlgincdir (Verilog Include Directories) Syntax Example

`xst elaborate -vlgincdir` *c:/my_verilog*

Adds `c:/my_verilog` to the list of directories in which XST looks for a file.

-verilog2001 (Verilog 2001)

`-verilog2001` (Verilog 2001) enables or disables interpreted Verilog source code as the Verilog 2001 standard. By default Verilog source code is interpreted as the Verilog 2001 standard.

You can also set this value in ISE® Design Suite with **Process > Properties > Synthesis Options > Verilog 2001**

-verilog2001 (Verilog 2001) Architecture Support

Architecture independent.

-verilog2001 (Verilog 2001) Applicable Elements

Applies to syntax.

-verilog2001 (Verilog 2001) Propagation Rules

Not applicable.

-verilog2001 (Verilog 2001) Syntax

`xst run -verilog2001` {**`yes`**|**`no`**}

The default is **`yes`**.

-verilog2001 (Verilog 2001) Syntax Example

`xst elaborate -verilog2001 no`

XST will not interpret Verilog code according to the Verilog 2001 standard.

-xsthdpini (HDL Library Mapping File)

Use **`-xsthdpini`** (HDL Library Mapping File) to define the library mapping.

XST maintains two library mapping files:

- The pre-installed file, which is installed during the Xilinx® software installation
- The user file, which you may define for your own projects

The pre-installed (default) `INI` file is named `xhdp.ini`, and is located in `%XILINX%\vhdl\xst`.This file contains information about the locations of the standard VHDL and UNISIM libraries. This file should not be modified, but you can copy the syntax for your own library mapping file.

To set the library mapping file location in ISE® Design Suite:

1. In ISE Design Suite, select **Process > Properties > Synthesis Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the HDL **`INI` File** property.

A library mapping file looks like the following:

```
-- 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
```

Use this file format to define where each of your own libraries must be placed. By default, all compiled VHDL flies are stored in the `xst` sub-directory of the ISE Design Suite project directory.

The library mapping file contains a list of libraries, one per line with the following information:

- The library name
- The directory in which the library is compiled

You can give this library mapping file any name you wish, but it is best to keep the `.ini` classification.

The format for each line is:

library_name=path_to_compiled_directory

Use double dash (--) to start a comment line.

-xsthdpini (HDL Library Mapping File) Architecture Support

Architecture independent.

-xsthdpini (HDL Library Mapping File) Applicable Elements

Applies to files.

-xsthdpini (HDL Library Mapping File) Propagation Rules

Not applicable.

-xsthdpini (HDL Library Mapping File) Syntax

`-xsthdpini` *`file_name`*

You can specify only one library mapping file.

-xsthdpini (HDL Library Mapping File) Syntax Example

```
xst set -xsthdpini c:/data/my_libraries/my.ini file_name
```

Specifies `c:/data/my_libraries/my.ini` as the file that will point to all of your libraries.

You must run this **`set`** command before any **`run`** commands.

MY.INI Example Text

```
work1=H:\Users\conf\my_lib\work1
work2=C:\mylib\work2
```

-xsthdpdir (Work Directory)

`-xsthdpdir` (Work Directory) defines the location in which VHDL-compiled files must be placed if the location is not defined by library mapping files. To access Work Directory:

- In ISE® Design Suite, select **Process > Properties > Synthesis Options > VHDL Working Directory**, or
- Use the following command in stand-alone mode:

  `set -xsthdpdir` *directory*

-xsthdpdir (Work Directory) Example

Assume the following for purposes of this example:

- Three different users are working on the same project.
- They share one standard, pre-compiled library, **`shlib`**.
- This library contains specific macro blocks for their project.
- Each user also maintains a local work library.
- User 3 places her local work library outside the project directory (for example, in c:\temp).
- Users 1 and 2 share another library (lib12) between them, but not with User 3.

The settings required for the three users are as follows:

-xsthdpdir (Work Directory) Example User One

```
Mapping file:
schlib=z:\sharedlibs\shlib
lib12=z:\userlibs\lib12
```

-xsthdpdir (Work Directory) Example User Two

```
Mapping file:
schlib=z:\sharedlibs\shlib
lib12=z:\userlibs\lib12
```

-xsthdpdir (Work Directory) Example User Three

```
Mapping file:
schlib=z:\sharedlibs\shlib
```

User Three will also set:

```
XSTHDPDIR = c:\temp
```

-xsthdpdir (Work Directory) Architecture Support

Architecture independent.

-xsthdpdir (Work Directory) Applicable Elements

Applies to directories.

-xsthdpdir (Work Directory) Propagation Rules

Not applicable.

-xsthdpdir (Work Directory) Syntax Examples

-xsthdpdir (Work Directory) XST Command Line Syntax Example

Define Work Directory globally with the **`set -xsthdpdir`** command line option before running the **`run`** command:

```
set -xsthdpdir directory
```

Work Directory can accept a single path only. You must specify the directory. There is no default.

-xsthdpdir (Work Directory) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > VHDL Work Directory.

To view Work Directory, select:

Edit > Preferences > Processes > Property Display Level > Advanced.

XST Hardware Description Language (HDL) Constraints

This section describes Hardware Description Language (HDL) design constraints that can be used with XST:

- Automatic FSM Extraction (FSM_EXTRACT)
- Enumerated Encoding (ENUM_ENCODING)
- Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL)
- FSM Encoding Algorithm (FSM_ENCODING)
- Mux Extraction (MUX_EXTRACT)
- Register Power Up (REGISTER_POWERUP)
- Resource Sharing (RESOURCE_SHARING)
- Safe Recovery State (SAFE_RECOVERY_STATE)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Signal Encoding (SIGNAL_ENCODING)

The constraints described in this chapter apply to:

- FPGA devices
- CPLD devices
- VHDL code
- Verilog code

Most of the constraints can be set globally in ISE® Design Suite in **Process > Properties > HDL Options**. The only constraints that *cannot* be set in **Process > Properties** are:

- Enumerated Encoding (ENUM_ENCODING)
- Safe Recovery State (SAFE_RECOVERY_STATE)
- Signal Encoding (SIGNAL_ENCODING)

FSM_EXTRACT (Automatic FSM Extraction)

FSM_EXTRACT (Automatic FSM Extraction) enables or disables finite state machine extraction and specific synthesis optimizations. In order to set values for the FSM Encoding Algorithm and FSM Flip-Flop Type, Automatic FSM Extraction must be enabled.

FSM_EXTRACT (Automatic FSM Extraction) Architecture Support

Architecture independent.

FSM_EXTRACT (Automatic FSM Extraction) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

FSM_EXTRACT (Automatic FSM Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

FSM_EXTRACT (Automatic FSM Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

FSM_EXTRACT (Automatic FSM Extraction) VHDL Syntax Example

Declare as follows:

```
attribute fsm_extract: string;
```

Specify as follows:

```
attribute fsm_extract of {entity_name | signal_name }: {entity |  signal}   is "{yes | no}";
```

FSM_EXTRACT (Automatic FSM Extraction) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* fsm_extract = "{yes | no}" *)
```

FSM_EXTRACT (Automatic FSM Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name " fsm_extract={yes | no | true | false };
```

FSM_EXTRACT (Automatic FSM Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"   NET "signal_name " fsm_extract={yes  | no | true | false};END;
```

FSM_EXTRACT (Automatic FSM Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-fsm_extract {yes | no}
```

The default is **yes**.

FSM_EXTRACT (Automatic FSM Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > FSM Encoding Algorithm. These options are:

- If FSM Encoding Algorithm is set to **None** , and **-fsm_extract** is set to **no, -fsm_encoding** does not influence synthesis.
- In all other cases, **-fsm_extract** is set to **yes**, and **-fsm_encoding** is set to the selected value. For more information about **-fsm_encoding**, see FSM Encoding Algorithm (FSM_ENCODING).

ENUM_ENCODING (Enumerated Encoding)

ENUM_ENCODING (Enumerated Encoding) applies a specific encoding to a VHDL enumerated type. The value is a string containing space-separated binary codes. You can specify ENUM_ENCODING only as a VHDL constraint on the considered enumerated type.

When describing a Finite State Machine (FSM) using an enumerated type for the state register, you may specify a particular encoding scheme with ENUM_ENCODING. In order for this encoding to be used by XST, set FSM Encoding Algorithm (FSM_ENCODING) to **user** for the considered state register.

ENUM_ENCODING (Enumerated Encoding) Architecture Support

Architecture independent.

ENUM_ENCODING (Enumerated Encoding) Applicable Elements

Applies to signals or types.

Because ENUM_ENCODING must preserve the external design interface, XST ignores ENUM_ENCODING when it is used on a port.

ENUM_ENCODING (Enumerated Encoding) Propagation Rules

Applies to the signal or type to which it is attached.

ENUM_ENCODING (Enumerated Encoding) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

ENUM_ENCODING (Enumerated Encoding) VHDL Syntax Example

Specify as a VHDL constraint on the considered enumerated type:

```
...
architecture behavior of example is
type statetype is (ST0, ST1, ST2, ST3);
attribute enum_encoding of statetype : type is "001   010 100 111";
signal state1 : statetype;
signal state2 : statetype;
begin
...
```

ENUM_ENCODING (Enumerated Encoding) XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name" NET "signal_name" enum_encoding="string";END;
```

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal)

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) 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. Flip-flop optimization includes removing:

- Equivalent flip-flops for FPGA and CPLD devices
- Flip-flops with constant inputs for CPLD devices

This processing increases the fitting success as a result of the logic simplification implied by the flip-flops elimination.

Equivalent Register Removal values are:

- **yes** (default)

  Flip-flop optimization is allowed.
- **no**

  Flip-flop optimization is inhibited. The flip-flop optimization algorithm is time consuming. For fast processing, use **no**.
- **true** (XCF only)
- **false** (XCF only)

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) Architecture Support

Architecture independent.

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) Propagation Rules

Removes equivalent flip-flops and flip-flops with constant inputs.

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) VHDL Syntax Example

Declare as follows:

```
attribute equivalent_register_removal: string;
```

Specify as follows:

```
attribute equivalent_register_removal of  {entity_name | signal_name} : {signal
| entity}   is "{yes  | no}";
```

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* equivalent_register_removal  = "{yes | no}" *)
```

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) XST Constraint File (XCF) Syntax Example One

```
MODEL " entity_name" equivalent_register_removal= {yes | no | true | false};
```

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL " entity_name" NET "signal_name " equivalent_register_removal=
{yes | no | true | false}; END;
```

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-equivalent_register_removal  {yes | no}
```

The default is **yes**.

EQUIVALENT_REGISTER_REMOVAL (Equivalent Register Removal) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx®Specific Options > Equivalent Register Removal .

FSM_ENCODING (FSM Encoding Algorithm)

FSM_ENCODING (FSM Encoding Algorithm) selects the Finite State Machine (FSM) coding technique. In order to select a value for the FSM Encoding Algorithm, Automatic FSM Extraction must be enabled.

FSM Encoding Algorithm values are:

- Auto
- One-Hot
- Compact
- Sequential
- Gray
- Johnson
- Speed1
- User

FSM Encoding Algorithm defaults to **auto** The best coding technique is automatically selected for each individual state machine.

FSM_ENCODING (FSM Encoding Algorithm) Architecture Support

Architecture independent.

FSM_ENCODING (FSM Encoding Algorithm) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

FSM_ENCODING (FSM Encoding Algorithm) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

FSM_ENCODING (FSM Encoding Algorithm) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

FSM_ENCODING (FSM Encoding Algorithm) Algorithm VHDL Syntax Example

Declare as follows:

```
attribute fsm_encoding:   string;
```

Specify as follows:

```
attribute fsm_encoding of {entity_name | signal_name}:  {entity | signal}   is "{auto | one-hot
| compact | sequential | gray | johnson | speed1 | user}";
```

The default is **auto**.

FSM_ENCODING (FSM Encoding Algorithm) Verilog Syntax Example

Place FSM Encoding Algorithm immediately before the module or signal declaration:

Place immediately before the module or signal declaration:

```
(* fsm_encoding  = "{auto | one-hot | compact | sequential | gray | johnson | speed1 | user}" *)
```

The default is **auto**.

FSM_ENCODING (FSM Encoding Algorithm) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name " fsm_encoding={auto | one-hot | compact | sequential |
gray | johnson | speed1 | user} ;
```

FSM_ENCODING (FSM Encoding Algorithm) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL " entity_name"NET "signal_name " fsm_encoding={auto  | one-hot | compact
| sequential | gray | johnson | speed1 | user };END;
```

FSM_ENCODING (FSM Encoding Algorithm) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-fsm_encoding {auto | one-hot | compact | sequential | gray | johnson | speed1 | user}
```

The default is **auto**.

FSM_ENCODING (FSM Encoding Algorithm) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > FSM Encoding Algorithm.

These options are:

- If the FSM Encoding Algorithm menu is set to **None** , and **-fsm_extract** is set to **no** , **-fsm_encoding** has no influence on the synthesis.
- In all other cases, **-fsm_extract** is set to **yes** and **-fsm_encoding** is set to the value selected in the menu. For more information, see Automatic FSM Extraction (FSM_EXTRACT).

MUX_EXTRACT (Mux Extraction)

MUX_EXTRACT (Mux Extraction) enables or disables multiplexer macro inference.

MUX_EXTRACT values are:

- **yes**
- **no**
- **force**
- **true** (XCF only)
- **false** (XCF only)

By default, multiplexer inference is enabled (**yes**). 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. The **force** value overrides those decision rules, and forces XST to create the MUX macro.

MUX_EXTRACT (Mux Extraction) Architecture Support

Architecture independent.

MUX_EXTRACT (Mux Extraction) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

MUX_EXTRACT (Mux Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

MUX_EXTRACT (Mux Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MUX_EXTRACT (Mux Extraction) VHDL Syntax Example

Declare as follows:

```
attribute mux_extract:  string;
```

Specify as follows:

```
attribute mux_extract of {signal_name | entity_name}: {entity | signal} is "{yes | no | force}";
```

The default is **yes**.

MUX_EXTRACT (Mux Extraction) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* mux_extract = "{yes | no | force}" *)
```

The default is **yes**.

MUX_EXTRACT (Mux Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" mux_extract={yes | no | true | false | force};
```

MUX_EXTRACT (Mux Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name" NET "signal_name" mux_extract={yes | no | true | false | force};END;
```

MUX_EXTRACT (Mux Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-mux_extract {yes | no | force}
```

The default is **yes**.

MUX_EXTRACT (Mux Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options.

REGISTER_POWERUP (Register Power Up)

XST does not automatically calculate and enforce register power-up values. You must explicitly specify them if needed using REGISTER_POWERUP (Register Power Up). This XST synthesis constraint can be assigned to a VHDL enumerated type, or it may be directly attached to a VHDL signal or a Verilog register node through a VHDL attribute or Verilog meta comment. The value may be a binary string or a symbolic code value.

REGISTER_POWERUP (Register Power Up) Architecture Support

Applies to the following devices only. Does not apply to any other devices.

- All CPLD devices
- Spartan®-3A devices

REGISTER_POWERUP (Register Power Up) Applicable Elements

Applies to signals or types.

REGISTER_POWERUP (Register Power Up) Propagation Rules

Applies to the signal or type to which it is attached.

REGISTER_POWERUP (Register Power Up) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

REGISTER_POWERUP (Register Power Up) VHDL Syntax Example One

The register is defined with a predefined VHDL type such as **`std_logic_vector`**. The REGISTER_POWERUP value is necessarily a binary code.

```
signal myreg : std_logic_vector (3 downto 0); attribute register_powerup of myreg : signal is "0001";
```

REGISTER_POWERUP (Register Power Up) VHDL Syntax Example Two

The register is defined with an enumerated type (symbolic state machine). REGISTER_POWERUP (Register Power Up) is attached to the signal and its value is one of the symbolic states defined. Actual power-up code differs depending on how the state machine is encoded.

```
type state_type is (s1, s2, s3, s4, s5); signal state1 : state_type;
```

REGISTER_POWERUP (Register Power Up) VHDL Syntax Example Three

REGISTER_POWERUP (Register Power Up) is attached to an enumerated type. All registers defined with that type inherit the constraint.

```
type state_type is (s1, s2, s3, s4, s5);
attribute register_powerup of state_type : type is "s1";
signal state1, state2 : state_type;
```

REGISTER_POWERUP (Register Power Up) VHDL Syntax Example Four

For enumerated type objects, the power-up value may also be defined as a binary code. However, if automatic encoding is enabled and leads to a different encoding scheme (in particular a different code width), the power-up value is ignored.

```
type state_type is (s1, s2, s3, s4, s5);
attribute enum_encoding of state_type : type is "001   011 010 100 111";
attribute register_powerup of state_type : type is "100";
signal state1 : state_type;
```

REGISTER_POWERUP (Register Power Up) Verilog Syntax Example

Place REGISTER_POWERUP (Register Power Up) immediately before the signal declaration:

```
(* register_powerup = "<value>"   *)
```

REGISTER_POWERUP (Register Power Up) XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name"

NET "signal_name" register_powerup="string";

END;
```

RESOURCE_SHARING (Resource Sharing)

RESOURCE_SHARING (Resource Sharing) enables or disables resource sharing of arithmetic operators.

RESOURCE_SHARING values are:

- **`yes`** (default)
- **`no`**
- **`force`**
- **`true`** (XCF only)
- **`false`** (XCF only)

RESOURCE_SHARING (Resource Sharing) Architecture Support

Architecture independent.

RESOURCE_SHARING (Resource Sharing) Applicable Elements

Applies to the entire design, or to design elements.

RESOURCE_SHARING (Resource Sharing) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

RESOURCE_SHARING (Resource Sharing) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

RESOURCE_SHARING (Resource Sharing) VHDL Syntax Example

Declare as follows:

```
attribute resource_sharing: string;
```

Specify as follows:

```
attribute resource_sharing of   entity_name: entity   is "{yes | no}";
```

RESOURCE_SHARING (Resource Sharing) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* resource_sharing = "{yes | no}" *)
```

RESOURCE_SHARING (Resource Sharing) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" resource_sharing={yes | no | true | false};
```

RESOURCE_SHARING (Resource Sharing) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" resource_sharing={yes | no | true | false};
END;
```

RESOURCE_SHARING (Resource Sharing) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-resource_sharing {yes | no}
```

The default is **yes**.

RESOURCE_SHARING (Resource Sharing) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

HDL Options > Resource Sharing.

SAFE_RECOVERY_STATE (Safe Recovery State)

SAFE_RECOVERY_STATE (Safe Recovery State) defines a recovery state for use when a Finite State Machine (FSM) is implemented in Safe Implementation mode. If the FSM enters an invalid state, XST uses additional logic to force the FSM to a valid recovery state. By implementing FSM in safe mode, XST collects all code not participating in the normal FSM behavior and treats it as illegal.

XST uses logic that returns the FSM synchronously to the:

- Known state
- Reset state
- Power up state
- State you specified using SAFE_RECOVERY_STATE

For more information, see Safe Implementation (SAFE_IMPLEMENTATION).

SAFE_RECOVERY_STATE (Safe Recovery State) Architecture Support

Architecture independent.

SAFE_RECOVERY_STATE (Safe Recovery State) Applicable Elements

Applies to a signal representing a state register.

SAFE_RECOVERY_STATE (Safe Recovery State) Propagation Rules

Applies to the signal to which it is attached.

SAFE_RECOVERY_STATE (Safe Recovery State) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

SAFE_RECOVERY_STATE (Safe Recovery State) VHDL Syntax Example

Declare as follows:

```
attribute safe_recovery_state: string;
```

Specify as follows:

```
attribute safe_recovery_state of {signal_name}:signal is "<value>";
```

SAFE_RECOVERY_STATE (Safe Recovery State) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* safe_recovery_state = "<value>" *)
```

SAFE_RECOVERY_STATE (Safe Recovery State) XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name"

NET "signal_name" safe_recovery_state="<value>";

END;
```

SAFE_IMPLEMENTATION (Safe Implementation)

SAFE_IMPLEMENTATION (Safe Implementation) implements Finite State Machine (FSM) components in Safe Implementation mode. In Safe Implementation mode, XST generates additional logic that forces an FSM to a valid state (recovery state) if the FSM enters 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.

Define the recovery state manually with Safe Recovery State (SAFE_RECOVERY_STATE).

To activate SAFE_IMPLEMENTATION in:

- ISE® Design Suite

  Select **Process > Properties > HDL Options > Safe Implementation**.
- Hardware Description Language (HDL)

  Apply SAFE_IMPLEMENTATION to the hierarchical block or signal that represents the state register in the FSM.

SAFE_IMPLEMENTATION (Safe Implementation) Architecture Support

Architecture independent.

SAFE_IMPLEMENTATION (Safe Implementation) Applicable Elements

Applies to an entire design through the XST command line, to a particular block (entity, architecture, component), or to a signal.

SAFE_IMPLEMENTATION (Safe Implementation) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

SAFE_IMPLEMENTATION (Safe Implementation) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

SAFE_IMPLEMENTATION (Safe Implementation) VHDL Syntax Example

Declare as follows:

```
attribute safe_implementation: string;
```

Specify as follows:

```
attribute safe_implementation of {entity_name | component_name | signal_name}: {entity
| component | signal} is "{yes | no}";
```

SAFE_IMPLEMENTATION (Safe Implementation) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* safe_implementation = "{yes | no}" *)
```

SAFE_IMPLEMENTATION (Safe Implementation) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" safe_implementation={yes | no | true | false};
```

SAFE_IMPLEMENTATION (Safe Implementation) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" safe_implementation={yes | no | true | false};
END;
```

SAFE_IMPLEMENTATION (Safe Implementation) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-safe_implementation {yes | no}
```

The default is **no**.

SAFE_IMPLEMENTATION (Safe Implementation) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

HDL Options > Safe Implementation.

SIGNAL_ENCODING (Signal Encoding)

SIGNAL_ENCODING (Signal Encoding) selects the coding technique to use for internal signals.

SIGNAL_ENCODING values are:

- **`auto`**

  The default. The best coding technique is automatically selected for each individual signal.
- **`one-hot`**

  Forces the encoding to a one-hot encoding
- **`user`**

  Forces XST to keep your encoding

SIGNAL_ENCODING (Signal Encoding) Architecture Support

Architecture independent.

SIGNAL_ENCODING (Signal Encoding) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

SIGNAL_ENCODING (Signal Encoding) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

SIGNAL_ENCODING (Signal Encoding) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

SIGNAL_ENCODING (Signal Encoding) VHDL Syntax Example

Declare as follows:

```
attribute signal_encoding:  string;
```

Specify as follows:

```
attribute signal_encoding of   {component_name | signal_name | entity_name | label_name}:
{component | signal | entity | label} is   "{auto | one-hot | user}";
```

The default is **auto**.

SIGNAL_ENCODING (Signal Encoding) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* signal_encoding = "{auto | one-hot | user}" *)
```

The default is **auto**.

SIGNAL_ENCODING (Signal Encoding) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" signal_encoding = {auto | one-hot | user};
```

SIGNAL_ENCODING (Signal Encoding) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name" signal_encoding = {auto | one-hot | user};

END;
```

SIGNAL_ENCODING (Signal Encoding) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-signal_encoding {auto | one-hot | user}
```

The default is **auto**.

XST FPGA Constraints (Non-Timing)

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 XST Specific Non-Timing Options, and XST Specific Non-Timing Options: XST Command Line Only for valid constraint targets.

The following XST FPGA constraints (non-timing) apply *only* to FPGA devices. These constraints do *not* apply to CPLD devices.

- Asynchronous to Synchronous (ASYNC_TO_SYNC)

- Automatic BRAM Packing (AUTO_BRAM_PACKING)
- BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)
- Buffer Type (BUFFER_TYPE)
- Extract BUFGCE (BUFGCE)
- Cores Search Directories (-sd)
- Decoder Extraction (DECODER_EXTRACT)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- FSM Style (FSM_STYLE)
- Power Reduction (POWER)
- Read Cores (READ_CORES)
- Logical Shifter Extraction (SHIFT_EXTRACT)
- LUT Combining (LC)
- Map Logic on BRAM (BRAM_MAP)
- Max Fanout (MAX_FANOUT)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)
- Multiplier Style (MULT_STYLE)
- Mux Style (MUX_STYLE)
- Number of Global Clock Buffers (-bufg)
- Number of Regional Clock Buffers (-bufr)
- Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)
- Pack I/O Registers Into IOBs (IOB)
- Priority Encoder Extraction (PRIORITY_EXTRACT)
- RAM Extraction (RAM_EXTRACT)
- RAM Style (RAM_STYLE)
- Reduce Control Sets (REDUCE_CONTROL_SETS)
- Register Balancing (REGISTER_BALANCING)
- Register Duplication (REGISTER_DUPLICATION)
- ROM Extraction (ROM_EXTRACT)
- ROM Style (ROM_STYLE)
- Shift Register Extraction (SHREG_EXTRACT)
- Slice Packing (-slice_packing)
- XOR Collapsing (XOR_COLLAPSE)
- Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)
- Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN)
- Map Entity on a Single LUT (LUT_MAP)
- Use Carry Chain (USE_CARRY_CHAIN)
- Convert Tristates to Logic (TRISTATE2LOGIC)
- Use Clock Enable (USE_CLOCK_ENABLE)
- Use Synchronous Set (USE_SYNC_SET)
- Use Synchronous Reset (USE_SYNC_RESET)
- Use DSP48 (USE_DSP48)

ASYNC_TO_SYNC (Asynchronous to Synchronous)

ASYNC_TO_SYNC (Asynchronous to Synchronous) allows you to replace Asynchronous Set/Reset signals with Synchronous signals throughout the entire design. This allows absorption of registers by DSP48 and BRAMs, thereby improving quality of results. In addition, this feature may have a positive impact on power optimization.

Although XST can place FSMs on BRAMs, in most cases an FSM has an Asynchronous Set/Reset signal, which does not allow FSM implementation on BRAMs. ASYNC_TO_SYNC allows you to more easily place FSMs on BRAMs, by eliminating the need to manually change the design.

Replacing Asynchronous Set/Reset signals by Synchronous signals makes the generated NGC netlist NOT equivalent to the initial RTL description. You must ensure that the synthesized design satisfies the initial specification. XST issues the following warning:

```
WARNING: You have requested that asynchronous control signals of sequential elements be treated as if
they were synchronous.  If you haven't done so yet, please carefully review the related documentation
material.  If you have opted to asynchronously control flip-flop initialization, this feature allows
you to better explore the possibilities offered by the Xilinx solution without having to go through
a painful rewriting effort.  However, be well aware that the synthesis result, while providing you
with a good way to assess final device usage and design performance, is not functionally equivalent
to your HDL description.  As a result, you will not be able to validate your design by comparison of
pre-synthesis and post-synthesis simulation results.  Please also note that in general we strongly
recommend synchronous flip-flop initialization.
```

ASYNC_TO_SYNC (Asynchronous to Synchronous) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

ASYNC_TO_SYNC (Asynchronous to Synchronous) Applicable Elements

Applies to the entire design.

ASYNC_TO_SYNC (Asynchronous to Synchronous) Propagation Rules

Not applicable.

ASYNC_TO_SYNC (Asynchronous to Synchronous) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

ASYNC_TO_SYNC (Asynchronous to Synchronous) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -async_to_sync{yes | no}
```

The default is **no**.

ASYNC_TO_SYNC (Asynchronous to Synchronous) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Asynchronous to Synchronous.

AUTO_BRAM_PACKING (Automatic BRAM Packing)

Automatic BRAM Packing (AUTO_BRAM_PACKING) allows you to pack two small BRAMs in a single BRAM primitive as dual-port BRAM. XST packs BRAMs together only if they are situated in the same hierarchical level.

AUTO_BRAM_PACKING Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

AUTO_BRAM_PACKING Applicable Elements

Applies to the entire design.

AUTO_BRAM_PACKING Propagation Rules

Not applicable.

AUTO_BRAM_PACKING Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Automatic BRAM Packing XST Command Line Syntax Example

Define in the XST command line as follows:

```
-auto_bram_packing {yes | no}
```

The default is **no**.

Automatic BRAM Packing ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Automatic BRAM Packing.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio)

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) defines the number of BRAM blocks that XST must not exceed during synthesis. BRAMs in the design may come not only from BRAM inference processes, but from instantiation and BRAM mapping optimizations. You may isolate an RTL description of logic in a separate block, and then ask XST to map this logic to BRAM. For more information, see Mapping Logic Onto Block RAM.

Instantiated BRAMs are the primary candidates for available BRAM resources. The inferred RAMs are placed on the remaining BRAM resources. However, if the number of instantiated BRAMs exceeds the number of available resources, XST does not modify the instantiations and implement them as block RAMs. The same behavior occurs if you force specific RAMs to be implemented as BRAMs. If there are no resources, XST respects user constraints, even if the number of BRAM resources is exceeded.

If the number of user-specified BRAMs exceeds the number of available BRAM resources on the target FPGA device, XST issues a warning, and uses only available BRAM resources on the chip for synthesis. However, you may disable automatic BRAM resource management by using value **-1**. This can be used to see the number of BRAMs XST can potentially infer for a specific design.

You may experience significant synthesis time if the number of BRAMs in the design significantly exceeds the number of available BRAMs on the target FPGA device (hundreds of BRAMs). This may happen due to a significant increase in design complexity when all non-fittable BRAMs are converted to distributed RAMs.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) Applicable Elements

Applies to the entire design.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) Propagation Rules

Not applicable.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) XST Command Line Syntax Examples

Define in the XST command line as follows:

```
-bram_utilization_ratio <integer>[%][#]
```

where

<*integer*> range is [-1 to 100] when % is used or both % and # are omitted

The default is **`100`**.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) XST Command Line Syntax Example One

```
-bram_utilization_ratio 50
```

means 50% of BRAMs blocks in the target device

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) XST Command Line Syntax Example Two

```
-bram_utilization_ratio 50%
```

means 50% of BRAMs blocks in the target device

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) XST Command Line Syntax Example Three

```
-bram_utilization_ratio 50#
```

means 50 BRAMs blocks

There must be no space between the integer value and the percent (**%**) or pound (**#**) characters.

In some situations, you can disable automatic BRAM resource management (for example, to see how many BRAMs XST can potentially infer for a specific design). To disable automatic resource management, specify **`-1`** (or any negative value) as a constraint value.

BRAM_UTILIZATION_RATIO (BRAM Utilization Ratio) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > BRAM Utilization Ratio.

In ISE Design Suite, you can define the value of BRAM Utilization Ratio only as a percentage. The definition of the value in the form of absolute number of BlockRAMs is not supported.

BUFFER_TYPE (Buffer Type)

Buffer Type (BUFFER_TYPE) is a new name for CLOCK_BUFFER. Since CLOCK_BUFFER will become obsolete in future releases, Xilinx® recommends that you use this new name. BUFFER_TYPE selects the type of buffer to be inserted on the input port or internal net. The **`bufr`** value is supported for Virtex®-4 devices and Virtex-5 devices only.

BUFFER_TYPE (Buffer Type) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

BUFFER_TYPE (Buffer Type) Applicable Elements

Applies to signals.

BUFFER_TYPE (Buffer Type) Propagation Rules

Applies to the signal to which it is attached.

BUFFER_TYPE (Buffer Type) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

BUFFER_TYPE (Buffer Type) VHDL Syntax Example

Declare as follows:

```
attribute buffer_type: string;
```

Specify as follows:

```
attribute buffer_type of  signal_name: signal is " {bufgdll  | ibufg |  bufgp | ibuf  | bufr |  none}";
```

BUFFER_TYPE (Buffer Type) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* buffer_type = "{bufgdll | ibufg | bufgp | ibuf | bufr | none}" *)
```

BUFFER_TYPE (Buffer Type) XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name " NET " signal_name"   buffer_type={bufgdll  | ibufg
|  bufgp | ibuf  | bufr |  none}; END;
```

BUFGCE (Extract BUFGCE)

BUFGCE (Extract BUFGCE) 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.

BUFGCE must be attached to the primary clock signal.

BUFGCE values are:

- **yes**
- **no**

BUFGCE is accessible through Hardware Description Language (HDL) code. If **bufgce=yes**, XST implements BUFGMUX functionality if possible. All flip-flops must have the same clock enable signal.

BUFGCE (Extract BUFGCE) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

BUFGCE (Extract BUFGCE) Applicable Elements

Applies to clock signals.

BUFGCE (Extract BUFGCE) Propagation Rules

Applies to the signal to which it is attached.

BUFGCE (Extract BUFGCE) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

BUFGCE (Extract BUFGCE) VHDL Syntax Example

Declare as follows:

```
attribute bufgce : string;
```

Specify as follows:

```
attribute bufgce of signal_name : signal is "{yes | no}";
```

BUFGCE (Extract BUFGCE) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* bufgce = "{yes | no}" *)
```

BUFGCE (Extract BUFGCE) XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name" NET "primary_clock_signal" bufgce={yes | no | true | false};END;
```

-sd (Cores Search Directories)

-sd (Cores Search Directories) tells XST to look for cores in directories other than the default. By default XST searches for cores in the directory specified in the **-ifn** option.

-sd (Cores Search Directories) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

-sd (Cores Search Directories) Applicable Elements

Applies to the entire design.

-sd (Cores Search Directories) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Cores Search Directory**
.

-sd (Cores Search Directories) Syntax

-sd {*directory_path* [*directory_path*]]

There is no default.

-sd (Cores Search Directories) Syntax Example

```
xst run -sd c:/data/cores c:/ise/cores
```

Tells XST to search for cores in `c:/data/cores` and `c:/ise/cores` in addition to the default directory. For more information, see Names With Spaces in Command Line Mode.

Decoder Extraction (DECODER_EXTRACT)

Decoder Extraction (DECODER_EXTRACT) enables or disables decoder macro inference.

Decoder Extraction (DECODER_EXTRACT) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

Decoder Extraction (DECODER_EXTRACT) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

Decoder Extraction (DECODER_EXTRACT) Propagation Rules

When attached to a net or signal, Decoder Extraction applies to the attached signal.

When attached to an entity or module, Decoder Extraction is propagated to all applicable elements in the hierarchy within the entity or module.

Decoder Extraction (DECODER_EXTRACT) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Decoder Extraction (DECODER_EXTRACT) VHDL Syntax Example

Declare as follows:

```
attribute decoder_extract: string;
```

Specify as follows:

```
attribute decoder_extract of {entity_name | signal_name}: {entity | signal} is "{yes | no}";
```

Decoder Extraction (DECODER_EXTRACT) Verilog Syntax Example

Place Decoder Extraction immediately before the module or signal declaration:

```
(* decoder_extract "{yes | no}" *)
```

Decoder Extraction (DECODER_EXTRACT) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" decoder_extract={yes | no | true | false};
```

Decoder Extraction (DECODER_EXTRACT) XST Constraint File (XCF) Syntax Example Two

```
BEGIN

MODEL "entity_name" NET "signal_name" decoder_extract={yes | no | true | false};

END;
```

Decoder Extraction (DECODER_EXTRACT) XST Command Line Syntax ISE Design Suite Example

Define in the XST command line as follows:

```
xst run -decoder_extract {yes | no}
```

The default is **yes**.

Decoder Extraction (DECODER_EXTRACT) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Decoder Extraction.

Decoder Extraction values are:

- **yes** (default)
- **no** (check box in not checked)

DSP_UTILIZATION_RATIO (DSP Utilization Ratio)

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) defines the number of DSP slices (in absolute number or percent of slices) that XST must not exceed during synthesis optimization.

The default is 100% of the target device.

DSP slices in the design may come not only from DSP inference processes, but also from instantiation. Instantiated DSP slices are the primary candidates for available DSP resources. The inferred DSPs are placed on the remaining DSP resources. If the number of instantiated DSPs exceeds the number of available resources, XST does not modify the instantiations and implement them as block DSP slices. The same behavior occurs if you force specific macro implementation to be implemented as DSP slices by using the Use DSP48 (USE_DSP48) constraint. If there are no resources, XST respects user constraints even if the number of DSP slices is exceeded.

If the number of user-specified DSP slices exceeds the number of available DSP resources on the target FPGA device, XST issues a warning, and uses only available DSP resources on the chip for synthesis.

You can disable automatic DSP resource management (for example, to see how many DSPs XST can potentially infer for a specific design) by specifying **-1** (or any negative value) as a constraint value.

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) Architecture Support

Applies to the following FPGA devices only. Does not apply to any other FPGA devices. Does not apply to CPLD devices.

- Spartan®-3A DSP
- Virtex®-4
- Virtex-5

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) Applicable Elements

Applies to the entire design.

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) Propagation Rules

Not applicable.

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-dsp_utilization_ratio  integer[%|#]
```

where

<*integer*> is [-1 to 100] when **%** is used or both **%** and **#** are omitted.

To specify a percent of total slices use **%**. To specify an absolute number of slices use **#**

The default is **%**.

For example:

- To specify 50% of DSP blocks of the target device enter the following:
  -dsp_utilization_ratio 50
- To specify 50% of DSP blocks of the target device enter the following:
  -dsp_utilization_ratio 50%
- To specify 50 DSP blocks enter the following:
  -dsp_utilization_ratio 50#

Note There must be no space between the integer value and the percent (**%**) or pound (**#**) characters.

DSP_UTILIZATION_RATIO (DSP Utilization Ratio) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > DSP Utilization Ratio.

In ISE Design Suite, you can define the value of DSP Utilization Ratio only as a percentage. You can not define the value as an absolute number of slices.

FSM_STYLE (FSM Style)

FSM_STYLE (FSM Style) can make large Finite State Machine (FSM) components more compact and faster by implementing them in the block RAM resources provided in Virtex® devices and later technologies. Use FSM_STYLE to direct XST to use block RAM resources rather than LUTs (default) to implement FSMs. FSM_STYLE is both a global and a local constraint.

FSM_STYLE (FSM Style) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

FSM_STYLE (FSM Style) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

FSM_STYLE (FSM Style) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

FSM_STYLE (FSM Style) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

FSM_STYLE (FSM Style) VHDL Syntax Example

Declare as follows:

```
attribute fsm_style: string;
```

Declare as follows:

```
attribute fsm_style of  {entity_name | signal_name}: {entity | signal}   is "{lut | bram}";
```

The default is **lut**.

FSM_STYLE (FSM Style) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* fsm_style = "{lut | bram}" *)
```

FSM_STYLE (FSM Style) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" fsm_style = {lut | bram};
```

FSM_STYLE (FSM Style) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name NET "signal_name" fsm_style = {luxt | bram};END;"
```

FSM_STYLE (FSM Style) XST Constraint File (XCF) Syntax Example Three

```
BEGIN MODEL "entity_name" INST "instance_name" fsm_style = {lut | bram};END;
```

FSM_STYLE (FSM Style) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > FSM Style.

POWER (Power Reduction)

POWER (Power Reduction) instructs XST to optimize the design to consume as little power as possible. Macro processing decisions are made to implement functions in a manner than uses minimal power. Although POWER is allowed in both AREA and SPEED modes, it may negatively impact the final overall area and speed of the design.

In the current release, power optimization done by XST is dedicated to DSP48 and BRAM blocks.

XST supports two BRAM optimization methods:

- Method One does not significantly impact area and speed. Method One is used by default when power optimization is enabled.
- Method Two saves more power, but may significantly impact area and speed.

Both methods can be controlled by using the RAM Style (RAM_STYLE) constraint with **`block_power1`** for Method One and **`block_power2`** for Method Two.

In some situations, XST may issue an HDL Advisor message giving you tips on how to improve your design. For example, if XST detects that Read First mode is used for BRAM, XST recommends that you use Write First or No Change modes.

POWER (Power Reduction) Architecture Support

Applies to Virtex®-4 devices and Virtex-5 devices only. Does not apply to any other FPGA devices. Does not apply to CPLD devices.

POWER (Power Reduction) Applicable Elements

Applies to:

- A component or entity (VHDL)
- A model or label (instance) (Verilog)
- A model or INST (in model) (XCF)
- The entire design (XST command line)

POWER (Power Reduction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

POWER (Power Reduction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

POWER (Power Reduction) VHDL Syntax Example

Declare as follows:

```
attribute power: string;
```

Specify as follows:

```
attribute power of {component name | entity_name}: {component | entity} is "{yes | no}";
```

The default is **no**.

POWER (Power Reduction) Verilog Syntax Example

Place this constraint immediately before the module declaration or instantiation:

```
(* power = "{yes | no}" *)
```

The default is **no**.

POWER (Power Reduction) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name" power = {yes | no | true | false};
```

The default is **false**.

POWER (Power Reduction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -power {yes | no}
```

The default is **no**.

POWER (Power Reduction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Power Reduction.

READ_CORES (Read Cores)

Use READ_CORES (Read Cores) to enable or disable the ability of XST to read Electronic Data Interchange Format (EDIF) or NGC core files for timing estimation and device utilization control. By reading a specific core, XST is better able to optimize logic around the core, since it sees how the logic is connected. However, in some cases the READ_CORES operation must be disabled in XST in order to obtain the desired results. For example, the PCI™ core must not be visible to XST, since the logic directly connected to the PCI core must be optimized differently as compared to other cores. READ_CORES allows you to enable or disable read operations on a core by core basis.

For more information, see Cores Processing.

READ_CORES has three possible values:

- **no (false)**

  Disables cores processing
- **yes (true)**

  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 cores netlist into the overall design. This value is available through the XST command line mode only.

READ_CORES (Read Cores) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

READ_CORES (Read Cores) Applicable Elements

Since this constraint can be used with BoxType (BOX_TYPE) the set of objects on which the both constraints can be applied must be the same.

Apply READ_CORES to:

- A component or entity (VHDL)
- A model or label (instance) (Verilog)
- A model or INST (in model) (XCF)
- The entire design (XST command line)

If READ_CORES is applied to at least a single instance of a block, then READ_CORES is applied to all other instances of this block for the entire design.

READ_CORES (Read Cores) Propagation Rules

Not applicable.

READ_CORES (Read Cores) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

READ_CORES (Read Cores) VHDL Syntax Example

Declare as follows:

```
attribute read_cores: string;
```

Specify as follows:

```
attribute read_cores of {component_name | entity_name} : {component | entity}
is "{yes | no | optimize}";
```

The default is **yes**.

READ_CORES (Read Cores) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* read_cores = "{yes | no | optimize}" *)
```

The default is **yes**.

READ_CORES (Read Cores) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" read_cores = {yes | no | true | false | optimize};
```

READ_CORES (Read Cores) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"  END;

INST "instance_name "read_cores =  {yes | no | true | false | optimize};

END;
```

The default is **yes**.

READ_CORES (Read Cores) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-read_cores {yes | no | optimize}
```

READ_CORES (Read Cores) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Read Cores.

The **optimize** option is not available in ISE Design Suite.

SHIFT_EXTRACT (Logical Shifter Extraction)

SHIFT_EXTRACT (Logical Shifter Extraction) enables or disables logical shifter macro inference.

SHIFT_EXTRACT values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

SHIFT_EXTRACT (Logical Shifter Extraction) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

SHIFT_EXTRACT (Logical Shifter Extraction) Applicable Elements

Applies to the entire design, or to design elements and nets.

SHIFT_EXTRACT (Logical Shifter Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

SHIFT_EXTRACT (Logical Shifter Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

SHIFT_EXTRACT (Logical Shifter Extraction) VHDL Syntax Example

Declare as follows:

```
attribute shift_extract:  string;
```

Specify as follows:

```
attribute shift_extract of {entity_name | signal_name}: {signal | entity}   is "{yes | no}";
```

SHIFT_EXTRACT (Logical Shifter Extraction) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* shift_extract = "{yes | no}" *)
```

SHIFT_EXTRACT (Logical Shifter Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" shift_extract={yes | no | true | false};
```

SHIFT_EXTRACT (Logical Shifter Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" shift_extract={yes | no | true | false};
END;
```

SHIFT_EXTRACT (Logical Shifter Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-shift_extract {yes | no}
```

The default is **yes**.

SHIFT_EXTRACT (Logical Shifter Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Logical Shifter Extraction.

LC (LUT Combining)

LC (LUT Combining) enables the merging of LUT pairs with common inputs into single dual-output LUT6s in order to improve design area. This optimization process may reduce design speed.

LC supports three values:

- **auto**
  XST tries to make a trade-off between area and speed.
- **area**
  XST performs maximum LUT combining to provide as small an implementation as possible.
- **off**
  Disables LC.

LC (LUT Combining) Architecture Support

Applies to Virtex®-5 devices only. Does not apply to any other FPGA devices. Does not apply to CPLD devices.

LC (LUT Combining) Applicable Elements

Applies to the entire design.

LC (LUT Combining) Propagation Rules

Not applicable.

LC (LUT Combining) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

LC (LUT Combining) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -lc {auto | area | off}
```

The default is **off**.

LC (LUT Combining) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > LUT Combining.

BRAM_MAP (Map Logic on BRAM)

BRAM_MAP (Map Logic on BRAM) is used to map an entire hierarchical block on the block RAM resources available in Virtex® devices and later technologies.

BRAM_MAP values are:

- **yes**
- **no** (default)

BRAM_MAP is both a global and a local constraint. For more information, see Mapping Logic Onto Block RAM.

BRAM_MAP (Map Logic on BRAM) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

BRAM_MAP (Map Logic on BRAM) Applicable Elements

Applies to BRAMs.

BRAM_MAP (Map Logic on BRAM) Propagation Rules

Isolate the logic (including output register) to be mapped on RAM in a separate hierarchical level. Logic that does not fit on a single block RAM is not mapped. Ensure that the whole entity fits, not just part of it.

The attribute BRAM_MAP is set on the instance or entity. If no block RAM can be inferred, the logic is passed to Global Optimization, where it is optimized. The macros *are not* inferred. Be sure that XST has mapped the logic.

BRAM_MAP (Map Logic on BRAM) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

BRAM_MAP (Map Logic on BRAM) VHDL Syntax Example

Declare as follows:

```
attribute bram_map: string;
```

Specify as follows:

```
attribute bram_map of component_name : component is "{yes | no}";
```

BRAM_MAP (Map Logic on BRAM) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* bram_map = "{yes | no}" *)
```

BRAM_MAP (Map Logic on BRAM) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" bram_map = {yes | no | true | false};
```

BRAM_MAP (Map Logic on BRAM) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
 INST "instance_name" bram_map = {yes | no | true | false};
END;
```

MAX_FANOUT (Max Fanout)

MAX_FANOUT (Max Fanout) limits the fanout of nets or signals. The value is an integer. The default value varies depending on the targeted device family as shown in the following table. MAX_FANOUT is both a global and a local constraint.

MAX_FANOUT (Max Fanout) Default Value

Devices	Default Value
Spartan®-3 Spartan-3E Spartan-3A Spartan-3A DSP	500
Virtex®-4	500
Virtex-5	100000 (One Hundred Thousand)

Large fanouts can cause routability problems. XST tries to limit fanout by duplicating gates or by inserting buffers. This limit is not a technology limit but a guide to XST. It may happen that this limit is not exactly respected, especially when this limit is small (less than 30).

In most cases, fanout control is performed by duplicating the gate driving the net with a large fanout. If the duplication cannot be performed, buffers are inserted. These buffers are protected against logic trimming at the implementation level by defining a Keep (KEEP) attribute in the NGC file.

If the register replication option is set to **no**, only buffers are used to control fanout of flip-flops and latches.

MAX_FANOUT is global for the design, but you can control maximum fanout independently for each entity or module or for given individual signals by using constraints.

If the actual net fanout is less than the MAX_FANOUT value, XST behavior depends on how MAX_FANOUT is specified.

- If the value of MAX_FANOUT is set in ISE® Design Suite in the command line, or is attached to a specific hierarchical block, XST interprets its value as a guidance.
- If MAX_FANOUT is attached to a specific net, XST does not perform logic replication. Putting MAX_FANOUT on the net may prevent XST from having better timing optimization.

For example, suppose that the critical path goes through the net, which actual fanout is 80 and set Max Fanout value to 100. If MAX_FANOUT is specified in ISE Design Suite, XST may replicate it, trying to improve timing. If MAX_FANOUT is attached to the net itself, XST does not perform logic replication.

MAX_FANOUT (Max Fanout) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

MAX_FANOUT (Max Fanout) Applicable Elements

Applies to the entire design.

MAX_FANOUT (Max Fanout) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

MAX_FANOUT (Max Fanout) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MAX_FANOUT (Max Fanout) VHDL Syntax Example

Declare as follows:

```
attribute max_fanout: string;
```

Specify as follows:

```
attribute max_fanout of {signal_name | entity_name}: {signal | entity} is "integer";
```

MAX_FANOUT (Max Fanout) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* max_fanout = "integer" *)
```

MAX_FANOUT (Max Fanout) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" max_fanout=integer;
```

MAX_FANOUT (Max Fanout) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
 NET "signal_name" max_fanout=integer;
END;
```

MAX_FANOUT (Max Fanout) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-max_fanout integer
```

MAX_FANOUT (Max Fanout) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Max Fanout.

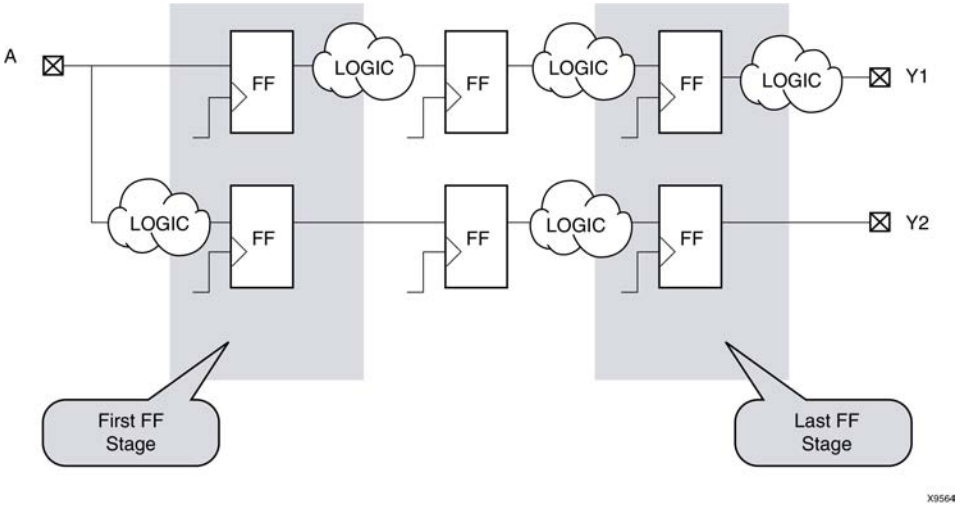
MOVE_FIRST_STAGE (Move First Stage)

MOVE_FIRST_STAGE (Move First Stage) controls the retiming of registers with paths coming from primary inputs. Both MOVE_FIRST_STAGE and MOVE_LAST_STAGE (Move Last Stage) relate to Register Balancing.

Note

- A flip-flop (FF in the diagram) belongs to the First Stage if it is on the paths coming from primary inputs.
- A flip-flop belongs to the Last Stage if it is on the paths going to primary outputs.

MOVE_FIRST_STAGE (Move First Stage) Diagram



During register balancing:

- First Stage flip-flops are moved forward
- Last Stage flip-flops are moved backward.

This process can dramatically increase input-to-clock and clock-to-output timing, which is not desirable. To prevent this, you may use OFFSET_IN_BEFORE and OFFSET_IN_AFTER constraints.

If:

- The design does not have a strong requirements, or
- You want to see the first results without touching the first and last flip-flop stages,

You can use two additional constraints:

- MOVE_FIRST_STAGE
- MOVE_LAST_STAGE

Both constraints may have either of two values: **yes** or **no**.

- **MOVE_FIRST_STAGE=no**

  Prevents the first flip-flop stage from moving
- **MOVE_LAST_STAGE=no**

  Prevents the last flip-flop stage from moving

Several constraints influence register balancing. For more information, see Register Balancing (REGISTER_BALANCING).

MOVE_FIRST_STAGE (Move First Stage) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

MOVE_FIRST_STAGE (Move First Stage) Applicable Elements

MOVE_FIRST_STAGE (Move First Stage) applies to the following only:

- Entire design
- Single modules or entities
- Primary clock signal

MOVE_FIRST_STAGE (Move First Stage) Propagation Rules

For Move First Stage propagation rules, see the figure above.

MOVE_FIRST_STAGE (Move First Stage) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MOVE_FIRST_STAGE (Move First Stage) VHDL Syntax Example

Declare as follows:

```
attribute move_first_stage : string;
```

Specify as follows:

```
attribute move_first_stage of   {entity_name | signal_name}: {signal | entity} is "{yes | no}";
```

MOVE_FIRST_STAGE (Move First Stage) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* move_first_stage = "{yes | no}" *)
```

MOVE_FIRST_STAGE (Move First Stage) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" move_first_stage={yes | no | true | false};
```

MOVE_FIRST_STAGE (Move First Stage) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "primary_clock_signal" move_first_stage={yes | no | true | false};
END;
```

MOVE_FIRST_STAGE (Move First Stage) XST Command Line Syntax

Define in the XST command line as follows:

```
xst run -move_first_stage {yes | no}
```

The default is **yes**.

MOVE_FIRST_STAGE (Move First Stage) ISE Design Suite Syntax

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Move First Flip-Flop Stage.

MOVE_LAST_STAGE (Move Last Stage)

MOVE_LAST_STAGE (Move Last Stage) controls the retiming of registers with paths going to primary outputs. Both Move Last Stage and Move First Stage (MOVE_FIRST_STAGE) relate to Register Balancing.

MOVE_LAST_STAGE (Move Last Stage) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

MOVE_LAST_STAGE (Move Last Stage) Applicable Elements

Applies to the following:

- Entire design
- Single modules or entities
- Primary clock signal

MOVE_LAST_STAGE (Move Last Stage) Propagation Rules

MOVE_LAST_STAGE (Move Last Stage) propagation rules, see Move First Stage (MOVE_FIRST_STAGE).

MOVE_LAST_STAGE (Move Last Stage) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MOVE_LAST_STAGE (Move Last Stage) Syntax Example

Declare as follows:

```
attribute move_last_stage : string;
```

Specify as follows:

```
attribute move_last_stage of  {entity_name  |  signal_name}: {signal  |  entity}  is "{yes  |  no}";
```

Move Last Stage Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* move_last_stage = "{yes  |  no}" *)
```

Move Last Stage XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" move_last_stage={yes  |  no  |  true  |  false};
```

Move Last Stage XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "primary_clock_signal"  move_last_stage={yes  |  no  |  true  |  false};
END;
```

Move Last Stage XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -move_last_stage {yes  |  no}
```

The default is **yes**.

Move Last Stage ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Move Last Stage.

MULT_STYLE (Multiplier Style)

MULT_STYLE (Multiplier Style) controls the way the macrogenerator implements the multiplier macros.

MULT_STYLE values are:

- **`auto`**

  The default is **`auto`**.

  The default instructs XST to look for the best implementation for each considered macro.
- **`block`**
- **`pipe_block`**

  The **`pipe_block`** option is used to pipeline DSP48 based multipliers. It is available for Virtex®-4 devices, Virtex-5 devices, and Spartan®-3A DSP devices only
- **`kcm`**
- **`csd`**
- **`lut`**
- **`pipe_lut`**

  The **`pipe_lut`** option is for pipeline slice-based multipliers.

MULT_STYLE (Multiplier Style) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

MULT_STYLE (Multiplier Style) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

MULT_STYLE (Multiplier Style) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

MULT_STYLE (Multiplier Style) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MULT_STYLE (Multiplier Style) VHDL Syntax Example

Declare as follows:

```
attribute mult_style: string;
```

Specify as follows:

```
attribute mult_style of {signal_name | entity_name}: {signal | entity}   is "{auto
| block | pipe_block | kcm | csd | lut | pipe_lut}";
```

MULT_STYLE (Multiplier Style) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* mult_style = "{auto | block | pipe_block | kcm | csd | lut | pipe_lut}" *)
```

MULT_STYLE (Multiplier Style) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" mult_style={auto | block | pipe_block | kcm | csd | lut | pipe_lut};
```

MULT_STYLE (Multiplier Style) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name" NET "signal_name" mult_style={auto | block | pipe_block
| kcm | csd | lut | pipe_lut}
;END;
```

MULT_STYLE (Multiplier Style) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -mult_style {auto | block | pipe_block | kcm | csd | lut | pipe_lut}
```

The **-mult_style** command line option is not supported for Virtex-4 devices, Virtex-5 devices, or Spartan-3A DSP devices. For those devices, use **-use_dsp48**.

MULT_STYLE (Multiplier Style) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Multiplier Style.

MUX_STYLE (Mux Style)

MUX_STYLE (Mux Style) controls the way the macrogenerator implements the multiplexer macros.

MUX_STYLE values are:

- **auto** (default)
- **muxf**
- **muxcy**

DOCTYPE command **auto**. XST looks for the best implementation for each considered macro.

Available MUX_STYLE (Mux Style) Implementation Styles Devices

Devices	Resources
Spartan®-3	MUXF
Spartan-3E	MUXF6
Spartan-3A	MUXCY
Spartan-3A DSP	MUXF7
Virtex®-4	MUXF8
Virtex-5	

MUX_STYLE (Mux Style) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

MUX_STYLE (Mux Style) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

MUX_STYLE (Mux Style) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

MUX_STYLE (Mux Style) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

MUX_STYLE (Mux Style) VHDL Syntax Example

Declare as follows:

```
attribute mux_style: string;
```

Specify as follows:

```
attribute mux_style of {signal_name | entity_name}: {signal | entity}   is "{auto | muxf | muxcy}";
```

The default is **auto**.

MUX_STYLE (Mux Style) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* mux_style = "{auto | muxf | muxcy}" *)
```

The default is **auto**.

MUX_STYLE (Mux Style) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" mux_style={auto | muxf | muxcy};
```

MUX_STYLE (Mux Style) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name" NET "signal_name" mux_style={auto | muxf | muxcy};END;
```

MUX_STYLE (Mux Style) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -mux_style {auto | muxf | muxcy}
```

DOCTYPE command **auto**.

MUX_STYLE (Mux Style) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Mux Style.

-bufg (Number of Global Clock Buffers)

-bufg (Number of Global Clock Buffers) controls the maximum number of BUFGs created by XST. The value is an integer. The default value depends on the target device, and is equal to the maximum number of available BUFGs.

-bufg (Number of Global Clock Buffers) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

-bufg (Number of Global Clock Buffers) Applicable Elements

Applies to the entire design.

-bufg (Number of Global Clock Buffers) Propagation Rules

Not applicable.

To set the number of global clock buffers in ISE® Design Suite:

1. Select **Process > Properties > Xilinx®-Specific Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the Number of Clock Buffers property.

-bufg (Number of Global Clock Buffers) Syntax

The value is an integer, and cannot exceed the maximum number of BUFGs available in the target device.

The default values are different for different architectures. Defaults for selected architectures are shown below.

Devices	Default Value
Virtex®-4 Virtex-5	32
Spartan®-3	8
Spartan-3E Spartan-3A Spartan-3A DSP	24

-bufg (Number of Global Clock Buffers) Syntax Example

```
xst run -bufg 8
```

Sets the number of global clock buffers to 8.

-bufr (Number of Regional Clock Buffers)

-bufr (Number of Regional Clock Buffers) controls the maximum number of BUFRs created by XST. The value is an integer. The default value depends on the target device, and is equal to the maximum number of available BUFRs.

-bufr (Number of Regional Clock Buffers) Architecture Support

- May be used with Virtex®-4 devices only.
- May NOT be used with Virtex-5 devices.
- Does not apply to CPLD devices.

-bufr (Number of Regional Clock Buffers) Applicable Elements

Applies to the entire design.

-bufr (Number of Regional Clock Buffers) Propagation Rules

Not applicable.

To set the number of regional clock buffers in ISE® Design Suite:

1. Select **Process > Properties > Xilinx®-Specific Options**.
2. From the **Property display level** list, select **Advanced**.
3. Set the **Number of Regional Clock Buffers** property.

-bufr (Number of Regional Clock Buffers) Syntax

```
xst run -bufr integer
```

The value is an integer, and cannot exceed the maximum number of BUFRs for the target device.

-bufr (Number of Regional Clock Buffers) Syntax Example

```
xst run -bufr 6
```

Sets the number or regional clock buffers to 6.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives)

By default, XST does not optimize instantiated primitives in Hardware Description Languages (HDLs). Use OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) to deactivate the default. OPTIMIZE_PRIMITIVES allows XST to optimize Xilinx® library primitives that have been instantiated in an HDL.

Optimization of instantiated primitives is limited by the following factors:

- If an instantiated primitive has specific constraints such as RLOC attached, XST preserves it as is.
- Not all primitives are considered by XST for optimization. Such hardware elements as MULT18x18, BRAMs, and DSP48 are not optimized (modified) even if optimization of instantiated primitives is enabled.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) Applicable Elements

Applies to hierarchical blocks, components, and instances.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) Propagation Rules

Applies to the component or instance to which it is attached.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) Schematic Syntax Examples

- Attach to a valid instance
- Attribute Name

  OPTIMIZE_PRIMITIVES
- Attribute Values
  - **yes**
  - **no** (default)

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) VHDL Syntax Example

Declare as follows:

```
attribute optimize_primitives:  string;
```

Specify as follows:

```
attribute optimize_primitives of {component_name | entity_name | label_name}:
{component | entity | label} is "{yes | no}";
```

Optimize Instantiated Primitives Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* optimize_primitives   = "{yes | no}" *)
```

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name" optimize_primitives = {yes | no | true | false};
```

OPTIMIZE_PRIMITIVES (Optimize Instantiated Primitives) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Optimize Instantiated Primitives.

IOB (Pack I/O Registers Into IOBs)

IOB (Pack I/O Registers Into IOBs) packs flip-flops in the I/Os to improve input/output path timing.

When IOB is set to **auto**, the action XST takes depends on the Optimization setting:

- If Optimization is set to **area**, XST packs registers as tightly as possible to the IOBs in order to reduce the number of slices occupied by the design.
- If Optimization is set to **speed**, XST packs registers to the IOBs provided they are not covered by timing constraints (in other words, they are not taken into account by timing optimization). For example, if you specify a period constraint, XST packs a register to the IOB if it is not covered by the period constraint. If a register is covered by timing optimization, but you do want to pack it to an IOB, you must apply the IOB constraint locally to the register.

For more information, see IOB in the *Constraints Guide*.

PRIORITY_EXTRACT (Priority Encoder Extraction)

PRIORITY_EXTRACT (Priority Encoder Extraction) enables or disables priority encoder macro inference.

PRIORITY_EXTRACT (values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **force** (XCF only)

For each identified priority encoder description, based on some internal decision rules, XST actually creates a macro or optimize it with the rest of the logic. The **force** value allows you to override those decision rules, and force XST to extract the macro.

PRIORITY_EXTRACT (Priority Encoder Extraction) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

PRIORITY_EXTRACT (Priority Encoder Extraction) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

PRIORITY_EXTRACT (Priority Encoder Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

PRIORITY_EXTRACT (Priority Encoder Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

PRIORITY_EXTRACT (Priority Encoder Extraction) VHDL Syntax Example

Declare as follows:

```
attribute priority_extract:  string;
```

Specify as follows:

```
attribute priority_extract of   {signal_name | entity_name}: {signal |
entity}   is "{yes | no | force}";
```

The default is **yes**.

PRIORITY_EXTRACT (Priority Encoder Extraction) Verilog Syntax Example

Place immediately before the module or signal declaration:

PRIORITY_EXTRACT (Priority Encoder Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" priority_extract={yes | no | true | false | force};
```

PRIORITY_EXTRACT (Priority Encoder Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name" NET "signal_name"  priority_extract={yes | no | true | false | force};END;
```

PRIORITY_EXTRACT (Priority Encoder Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -priority_extract {yes | no | force}
```

The default is **yes**.

PRIORITY_EXTRACT (Priority Encoder Extraction) ISE Design Suite Syntax Example

Define this constraint globally in ISE® Design Suite in:

Process > Properties > HDL Options > Priority Encoder Extraction.

RAM_EXTRACT (RAM Extraction)

RAM_EXTRACT (RAM Extraction) enables or disables RAM macro inference.

RAM_EXTRACT) values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

RAM_EXTRACT (RAM Extraction) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

RAM_EXTRACT (RAM Extraction) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

RAM_EXTRACT Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

RAM_EXTRACT (RAM Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

RAM_EXTRACT (RAM Extraction) VHDL Syntax Example

Declare as follows:

```
attribute ram_extract: string;
```

Specify as follows:

```
attribute ram_extract of {signal_name | entity_name}: {signal | entity} is "{yes | no}";
```

RAM_EXTRACT (RAM Extraction) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* ram_extract = "{yes | no}" *)
```

RAM_EXTRACT (RAM Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" ram_extract={yes | no | true | false};
```

RAM_EXTRACT (RAM Extraction) XST Constraint File (XCF) Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" ram_extract={yes | no | true | false};
END;
```

RAM_EXTRACT (RAM Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -ram_extract {yes | no}
```

The default is **yes**.

RAM_EXTRACT (RAM Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > RAM Extraction.

RAM_STYLE (RAM Style)

RAM_STYLE (RAM Style) controls the way the macrogenerator implements the inferred RAM macros.

RAM_STYLE) values are:

- **auto** (default)
- **block**
- **distributed**
- **pipe_distributed**
- **block_power1**
- **block_power2**

The default is **auto**.

XST looks for the best implementation for each inferred RAM.

You must use **block_power1** and **block_power2** in order to achieve power-oriented BRAM optimization. For more information, see Power Reduction (POWER).

The implementation style can be manually forced to use block RAM or distributed RAM resources.

You can specify **pipe_distributed**, **block_power1**, and **block_power2** only through VHDL or Verilog or XCF constraints.

RAM_STYLE (RAM Style) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

Block_power1 and **block_power2** are supported for Virtex®-4 devices and Virtex-5 devices only.

RAM_STYLE (RAM Style) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

RAM_STYLE (RAM Style) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

RAM_STYLE (RAM Style) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

RAM_STYLE (RAM Style) VHDL Syntax Example

Declare as follows:

```
attribute ram_style: string;
```

Specify as follows:

```
attribute ram_style of {signal_name | entity_name}: {signal | entity}   is "{auto | block |
distributed | pipe_distributed | block_power1 | block_power2}";
```

The default is **auto**.

RAM_STYLE (RAM Style) Verilog Syntax Example

Place this constraint immediately before the module or signal declaration:

```
(* ram_style = "{auto | block | distributed | pipe_distributed | block_power1 | block_power2}" *)
```

The default is **auto**.

RAM_STYLE (RAM Style) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" ram_style={auto | block | distributed | pipe_distributed
| block_power1 | block_power2};
```

RAM_STYLE (RAM Style) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name"  ram_style={auto | block | distributed | pipe_distributed
| block_power1 | block_power2};

END;
```

RAM_STYLE (RAM Style) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -ram_style {auto | block | distributed}
```

The default is **auto**.

The **pipe_distributed** value is not accessible through the command line.

RAM_STYLE (RAM Style) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > RAM Style.

REDUCE_CONTROL_SETS (Reduce Control Sets)

REDUCE_CONTROL_SETS (Reduce Control Sets) allows you to reduce the number of control sets and, as a consequence, reduce the design area. Reducing the control set number should improve the packing process in **map**, and therefore reduce the number of used slices even if the number of LUTs is increased.

REDUCE_CONTROL_SETS supports two values:

- **`auto`**

  XST optimizes automatically, and reduces the existing control sets in the design.

- **`no`**

  XST performs no control set optimization.

REDUCE_CONTROL_SETS (Reduce Control Sets) Architecture Support

Applies to Virtex®-5 devices only. Does not apply to any other FPGA devices. Does not apply to CPLD devices.

REDUCE_CONTROL_SETS (Reduce Control Sets) Applicable Elements

Applies to the entire design.

REDUCE_CONTROL_SETS (Reduce Control Sets) Propagation Rules

Not applicable.

REDUCE_CONTROL_SETS (Reduce Control Sets) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

REDUCE_CONTROL_SETS (Reduce Control Sets) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -reduce_control_sets {auto | no}
```

The default is **no**.

REDUCE_CONTROL_SETS (Reduce Control Sets) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx Specific Options > Reduce Control Sets

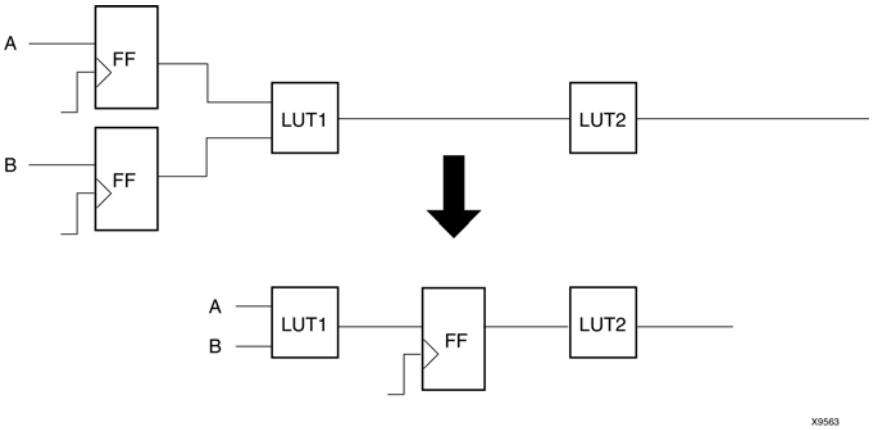
REGISTER_BALANCING (Register Balancing)

REGISTER_BALANCING (Register Balancing) enables flip-flop retiming. The main goal of register balancing is to move flip-flops and latches across logic to increase clock frequency.

The two categories of REGISTER_BALANCING (are:

- Forward Register Balancing
- Backward Register Balancing

Forward Register Balancing

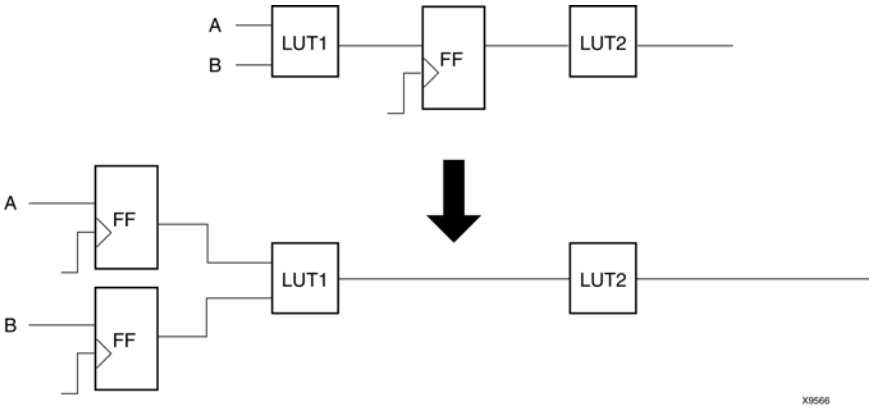


Forward Register Balancing moves a set of flip-flops at the inputs of a LUT to a single flip-flop at its output.

When replacing several flip-flops with one, select the name based on the name of the LUT across which the flip-flops are moving as shown in the following:

*LutName*_FRB*Id*

Backward Register Balancing



Backward Register Balancing moves a flip-flop at the output of a LUT to a set of flip-flops at its inputs.

As a consequence the number of flip-flops in the design can be increased or decreased.

The new flip-flop has the same name as the original flip-flop with an indexed suffix as shown in the following:

*OriginalFFName*_BRBId

Register Balancing values are:

- **`yes`**
  Both forward and backward retiming are allowed.
- **`no (default)`**
  Neither forward nor backward retiming is allowed.
- **`forward`**
  Only forward retiming is allowed
- **`backward`**
  Only backward retiming is allowed.
- **`true`** (XCF only)
- **`false`** (XCF only)

Two additional constraints control register balancing:

- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)

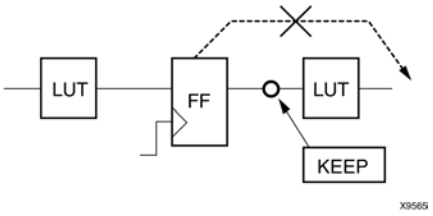
Several other constraints also influence register balancing:

- Keep Hierarchy (KEEP_HIERARCHY)
  - If the hierarchy is preserved, flip-flops are moved only inside the block boundaries.
  - If the hierarchy is flattened, flip-flops may leave the block boundaries.
- Pack I/O Registers Into IOBs (IOB)
  If IOB=TRUE, register balancing is not applied to the flip-flops having this property.
- Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)
  Instantiated flip-flops are moved only if OPTIMIZE_PRIMITIVES=YES.
- Flip-flops are moved across instantiated primitives only if OPTIMIZE_PRIMITIVES=YES.

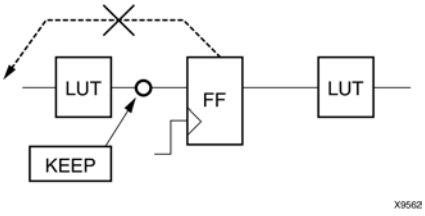
Keep (KEEP)

If applied to the output flip-flop signal, the flip-flop is not moved forward.

Applied to the Input Flip-Flop Signal



If applied to the input flip-flop signal, the flip-flop is not moved backward.



If applied to both the input and output of the flip-flop, it is equivalent to **`REGISTER_BALANCING=no`**

REGISTER_BALANCING (Register Balancing) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

REGISTER_BALANCING (Register Balancing) Applicable Elements

Applies to:

- The entire design using the command line or ISE® Design Suite
- An entity or module
- A signal corresponding to the flip-flop description (RTL)
- A flip-flop instance
- The Primary Clock Signal

  In this case the register balancing is performed only for flip-flops synchronized by this clock.

REGISTER_BALANCING (Register Balancing) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

REGISTER_BALANCING (Register Balancing) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

REGISTER_BALANCING (Register Balancing) VHDL Syntax Example

Declare as follows:

```
attribute register_balancing:   string;
```

Specify as follows:

```
attribute register_balancing of {signal_name | entity_name}:  {signal | entity} is "{yes | no | forward
| backward}";
```

REGISTER_BALANCING (Register Balancing) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* register_balancing =  "{yes | no | forward | backward}" *)
```

The default is **no**.

REGISTER_BALANCING (Register Balancing) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" register_balancing={yes | no | true | false | forward | backward};
```

REGISTER_BALANCING (Register Balancing) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"NET "primary_clock_signal" register_balancing={yes | no |
true | false | forward | backward};"END;
```

REGISTER_BALANCING (Register Balancing) XST Constraint File (XCF) Example Three

```
BEGIN MODEL "entity_name"INST " instance_name" register_balancing={yes | no |
true | false | forward | backward};END;
```

REGISTER_BALANCING (Register Balancing) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -register_balancing {yes | no | forward | backward}
```

The default is **no**.

REGISTER_BALANCING (Register Balancing) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx Specific Options > Register Balancing.

REGISTER_DUPLICATION (Register Duplication)

REGISTER_DUPLICATION (Register Duplication) enables or disables register replication.

REGISTER_DUPLICATION values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

The default is **yes**.

Register replication is enabled, and is performed during timing optimization and fanout control.

REGISTER_DUPLICATION (Register Duplication) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

REGISTER_DUPLICATION (Register Duplication) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

REGISTER_DUPLICATION (Register Duplication) Propagation Rules

Applies to the entity or module to which it is attached.

REGISTER_DUPLICATION (Register Duplication) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

REGISTER_DUPLICATION (Register Duplication) VHDL Syntax Example

Declare as follows:

```
attribute register_duplication: string;
```

Specify as follows:

```
attribute register_duplication of entity_name: entity is "{yes | no}";
```

REGISTER_DUPLICATION (Register Duplication) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* register_duplication = "{yes | no}" *)
```

REGISTER_DUPLICATION (Register Duplication) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" register_duplication={yes | no | true | false};
```

REGISTER_DUPLICATION (Register Duplication) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" register_duplication={yes | no | true | false};
END;
```

REGISTER_DUPLICATION (Register Duplication) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx Specific Options > Register Duplication.

ROM_EXTRACT (ROM Extraction)

ROM_EXTRACT (ROM Extraction) enables or disables ROM macro inference.

ROM_EXTRACT values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

The default is **yes**.

Typically, a ROM can be inferred from a **case** statement where all assigned contexts are constant values.

ROM_EXTRACT (ROM Extraction) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

ROM_EXTRACT (ROM Extraction) Applicable Elements

Applies to the entire design, or to a design element or signal.

ROM_EXTRACT (ROM Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

ROM_EXTRACT (ROM Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

ROM_EXTRACT (ROM Extraction) VHDL Syntax Example

Declare as follows:

```
attribute rom_extract: string;
```

Specify as follows:

```
attribute rom_extract of {signal_name | entity_name}: {signal | entity}  is "{yes | no}";
```

ROM_EXTRACT (ROM Extraction) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* rom_extract = "{yes | no}" *)
```

ROM_EXTRACT (ROM Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" rom_extract={yes | no | true | false};
```

ROM_EXTRACT (ROM Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" rom_extract={yes | no | true | false};
END;
```

ROM_EXTRACT (ROM Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -rom_extract {yes | no}
```

The default is **yes**.

ROM_EXTRACT (ROM Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > ROM Extraction.

ROM_STYLE (ROM Style)

ROM_STYLE (ROM Style) controls the way the macrogenerator implements the inferred ROM macros. ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE.

ROM_STYLE values are:

- **auto** (default)
- **block**

The default is **auto**.

XST looks for the best implementation for each inferred ROM. The implementation style can be manually forced to use block ROM or distributed ROM resources.

ROM_STYLE (ROM Style) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

ROM_STYLE (ROM Style) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

ROM_STYLE (ROM Style) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

ROM_STYLE (ROM Style) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

ROM_STYLE (ROM Style) VHDL Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE.

Declare as follows:

```
attribute rom_style: string;
```

Specify as follows:

```
attribute rom_style of {signal_name | entity_name}: {signal | entity}   is
"{auto | block | distributed}";
```

The default is **auto**.

ROM_STYLE (ROM Style) Verilog Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE.

Declare as follows:

```
(* rom_style = "{auto | block | distributed}" *)
```

The default is **auto**.

ROM_STYLE (ROM Style) XST Constraint File (XCF) Syntax Example One

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE.

```
MODEL "entity_name" rom_style={auto | block | distributed};
```

ROM_STYLE (ROM Style) XST Constraint File (XCF) Syntax Example Two

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE (ROM Style).

```
BEGIN MODEL "entity_name"
NET "signal_name"  rom_style={auto | block | distributed};
END;
```

ROM_STYLE (ROM Style) XST Command Line Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE (ROM Style).

Define in the XST command line as follows:

```
xst run -rom_style {auto | block | distributed}
```

The default is **auto**.

ROM_STYLE (ROM Style) ISE Design Suite Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to **yes** in order to use ROM_STYLE (ROM Style).

Define in ISE® Design Suite with:

Process > Properties > HDL Options > ROM Style.

SHREG_EXTRACT (Shift Register Extraction)

SHREG_EXTRACT (Shift Register Extraction) enables or disables shift register macro inference.

SHREG_EXTRACT values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

Enabling SHREG_EXTRACT for FPGA devices results in the usage of dedicated hardware resources such as SRL16 and SRLC16. For more information, see Shift Registers Hardware Description Language (HDL) Coding Techniques.

SHREG_EXTRACT (Shift Register Extraction) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

SHREG_EXTRACT (Shift Register Extraction) Applicable Elements

Applies to the entire design, or to a design element or signal.

SHREG_EXTRACT (Shift Register Extraction) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

SHREG_EXTRACT (Shift Register Extraction) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

SHREG_EXTRACT (Shift Register Extraction) VHDL Syntax Example

Declare as follows:

```
attribute shreg_extract : string;
```

Specify as follows:

```
attribute shreg_extract of {signal_name | entity_name}: {signal | entity} is "{yes | no}";
```

SHREG_EXTRACT (Shift Register Extraction) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* shreg_extract = "{yes | no}" *)
```

SHREG_EXTRACT (Shift Register Extraction) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" shreg_extract={yes | no | true | false};
```

SHREG_EXTRACT (Shift Register Extraction) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" shreg_extract={yes | no | true | false};
END;
```

SHREG_EXTRACT (Shift Register Extraction) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -shreg_extract {yes | no}
```

The default is **yes**.

SHREG_EXTRACT (Shift Register Extraction) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Shift Register Extraction.

-slice_packing (Slice Packing)

-slice_packing (Slice Packing) 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.

-slice_packing (Slice Packing) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

-slice_packing (Slice Packing) Applicable Elements

Applies to the entire design.

-slice_packing (Slice Packing) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite in **Process > Properties > Xilinx® Specific Options > Slice Packing**.

-slice_packing (Slice Packing) Syntax

```
-slice_packing {yes|no}
```

-slice_packing (Slice Packing) Syntax Example

```
xst run -slice_packing no
```

Disables the XST internal slice packer.

USELOWSKEWLINES (Use Low Skew Lines)

Use Low Skew Lines (USELOWSKEWLINES) is a basic routing constraint. During synthesis, Use Low Skew Lines prevents XST from using dedicated clock resources and logic replication, based on the value of the Max Fanout (MAX_FANOUT) constraint. Use Low Skew Lines specifies the use of low skew routing resources for any net. For more information, see USELOWSKEWLINES in the *Constraints Guide*.

XOR_COLLAPSE (XOR Collapsing)

XOR_COLLAPSE (XOR Collapsing) controls whether cascaded XORs should be collapsed into a single XOR.

XOR_COLLAPSE values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

XOR_COLLAPSE (XOR Collapsing) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

XOR_COLLAPSE (XOR Collapsing) Applicable Elements

Applies to cascaded XORs.

XOR_COLLAPSE (XOR Collapsing) Propagation Rules

Not applicable.

XOR_COLLAPSE (XOR Collapsing) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XOR_COLLAPSE (XOR Collapsing) VHDL Syntax Example

Declare as follows:

```
attribute xor_collapse:   string;
```

Specify as follows:

```
attribute xor_collapse {signal_name | entity_name}: {signal | entity} is "{yes | no}";
```

The default is **yes**.

XOR_COLLAPSE (XOR Collapsing) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* xor_collapse = "{yes | no}" *)
```

The default is **yes**.

XOR_COLLAPSE (XOR Collapsing) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" xor_collapse={yes | no | true | false};
```

XOR_COLLAPSE (XOR Collapsing) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name" xor_collapse={yes | no | true | false};

XOR Collapsing END;
```

XOR_COLLAPSE (XOR Collapsing) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -xor_collapse {yes | no}
```

The default is **yes**.

XOR_COLLAPSE (XOR Collapsing) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > XOR Collapsing.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) defines the area size in absolute numbers or percent of total numbers of

- LUT-FF pairs (Virtex®-5 devices)
- slices (all other devices)

that XST must not exceed during timing optimization.

If the area constraint cannot be satisfied, XST will make timing optimization regardless of the area constraint. To disable automatic resource management, specify **-1** as a constraint value. For more information, see Speed Optimization Under Area Constraint.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) Propagation Rules

Applies to the entity or module to which it is attached.

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Slice (LUT-FF Pairs) Utilization Ratio VHDL Syntax Examples

Declare as follows:

```
attribute slice_utilization_ratio: string;
```

Specify as follows:

```
attribute slice_utilization_ratio of   entity_name : entity is "integer";
attribute slice_utilization_ratio of entity_name : entity is "integer%";
attribute slice_utilization_ratio of entity_name : entity is "integer#";
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **-1** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

Slice (LUT-FF Pairs) Utilization Ratio Verilog Syntax Examples

Place immediately before the module declaration or instantiation:

```
(* slice_utilization_ratio = "integer" *)
(* slice_utilization_ratio = "integer%" *)
(* slice_utilization_ratio = "integer#" *)
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **-1** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

Slice (LUT-FF Pairs) Utilization Ratio XST Constraint File (XCF) Syntax Examples

```
MODEL "entity_name" slice_utilization_ratio=integer;
MODEL "entity_name" slice_utilization_ratio=integer%;
MODEL "entity_name" slice_utilization_ratio=integer#;
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **-1** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

There must be no space between the integer value and the percent (**%**) or pound (**#**) characters.

You must surround the integer value and the percent (**%**) and pound (**#**) characters with double quotes (**"..."**) because the percent (**%**) and pound (**#**) characters are special characters in the XST Constraint File (XCF).

Slice (LUT-FF Pairs) Utilization Ratio XST Command Line Syntax Examples

Define in the XST command line as follows:

```
xst run -slice_utilization_ratio integer
xst run -slice_utilization_ratio integer%
xst run -slice_utilization_ratio integer#
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **-1** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

Slice (LUT-FF Pairs) Utilization Ratio ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Slice Utilization Ratio or **Process > Properties > Synthesis Options > LUT-FF Pairs Utilization Ratio**.

In ISE Design Suite, you can define the value of Slice (LUT-FF Pairs) Utilization Ratio only as a percentage. You can not define the value as an absolute number of slices.

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN)

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) is closely related to Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO). Slice (LUT-FF Pairs) Utilization Ratio Delta defines the tolerance margin for Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO). The value of the parameter can be defined in the form of percentage as well as an absolute number of slices or LUT-FF pairs.

If the ratio is within the margin set, the constraint is met and timing optimization can continue. For more information, see Speed Optimization Under Area Constraint.

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) Applicable Elements

Applies to the entire design, or to an entity, component, module, or signal.

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) Propagation Rules

Applies to the entity or module to which it is attached.

Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Slice (LUT-FF Pairs) Utilization Ratio Delta VHDL Syntax Examples

Declare as follows:

```
attribute slice_utilization_ratio_maxmargin: string;
```

Specify as follows:

```
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer";
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer%";
attribute slice_utilization_ratio_maxmargin of entity_name : entity is "integer#";
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **0** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

Slice (LUT-FF Pairs) Utilization Ratio Delta Verilog Syntax Examples

Place immediately before the module declaration or instantiation:

```
(* slice_utilization_ratio_maxmargin = "integer" *)
(* slice_utilization_ratio_maxmargin = "integer%" *)
(* slice_utilization_ratio_maxmargin = "integer#" *)
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **0** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

Slice (LUT-FF Pairs) Utilization Ratio Delta XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" slice_utilization_ratio_maxmargin=integer;
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer%";
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer#";
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **0** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

There must be no space between the integer value and the percent (**%**) or pound (**#**) characters.

You must surround the integer value and the percent (**%**) and pound (**#**) characters with double quotes (**"..."**) because the percent (**%**) and pound (**#**) characters are special characters in the XST Constraint File (XCF).

Slice (LUT-FF Pairs) Utilization Ratio Delta XST Command Line Syntax Examples

Define in the XST command line as follows:

```
xst run -slice_utilization_ratio_maxmargin integer
xst run -slice_utilization_ratio_maxmargin integer%
xst run -slice_utilization_ratio_maxmargin integer#
```

XST interprets the integer values in the first two examples above as a percentage and in the last example as an absolute number of slices or FF-LUT pairs.

The integer value range is **0** to **100** when percent (**%**) is used or both percent (**%**) and pound (**#**) are omitted.

LUT_MAP (Map Entity on a Single LUT)

LUT_MAP (Map Entity on a Single LUT) 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.

Use the UNISIM library to directly instantiate LUT components in your Hardware Description Language (HDL) code. To specify a function that a particular LUT must execute, apply an INIT constraint to the instance of the LUT. To place an instantiated LUT or register in a particular slice, attach an RLOC constraint to the same instance.

It is not always convenient to calculate INIT functions and different methods 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 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. For more information, see Specifying INIT and RLOC.

XST automatically recognizes the XC_MAP constraint supported by Synplicity.

LUT_MAP (Map Entity on a Single LUT) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

LUT_MAP (Map Entity on a Single LUT) Applicable Elements

Applies to a VHDL entity or Verilog module.

LUT_MAP (Map Entity on a Single LUT) Propagation Rules

Applies to the entity or module to which it is attached.

LUT_MAP (Map Entity on a Single LUT) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

LUT_MAP (Map Entity on a Single LUT) VHDL Syntax Example

Declare as follows:

```
attribute lut_map: string;
```

Specify as follows:

```
attribute lut_map of entity_name   : entity is "{yes | no}";
```

LUT_MAP (Map Entity on a Single LUT) Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```
(* lut_map = "{yes | no}" *)
```

LUT_MAP (Map Entity on a Single LUT) XST Constraint File (XCF) Syntax Example

```
MODEL "entity_name" lut_map={yes | no | true | false};
```

USE_CARRY_CHAIN (Use Carry Chain)

XST uses carry chain resources to implement certain macros, but there are situations where you can obtain better results by not using carry chain. USE_CARRY_CHAIN (Use Carry Chain) can deactivate carry chain use for macro generation. USE_CARRY_CHAIN is both a global and a local constraint.

USE_CARRY_CHAIN) values are:

- **yes** (default)
- **no**

USE_CARRY_CHAIN (Use Carry Chain) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

USE_CARRY_CHAIN (Use Carry Chain) Applicable Elements

Applies to the entire design, or to signals.

USE_CARRY_CHAIN (Use Carry Chain) Propagation Rules

Applies to the signal to which it is attached.

USE_CARRY_CHAIN (Use Carry Chain) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

USE_CARRY_CHAIN (Use Carry Chain) Schematic Syntax Example

- Attach to a valid instance
- Attribute Name

  USE_CARRY_CHAIN
- Attribute Values
  - **yes**
  - **no**

USE_CARRY_CHAIN (Use Carry Chain) VHDL Syntax Example

Declare as follows:

```
attribute use_carry_chain: string;
```

Specify as follows:

```
attribute use_carry_chain of signal_name: signal is "{yes | no}";
```

USE_CARRY_CHAIN (Use Carry Chain) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* use_carry_chain = "{yes | no}" *)
```

USE_CARRY_CHAIN (Use Carry Chain) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" use_carry_chain={yes | no | true | false}X;
```

USE_CARRY_CHAIN (Use Carry Chain) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name" use_carry_chain={yes | no | true | false};

END;
```

USE_CARRY_CHAIN (Use Carry Chain) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -use_carry_chain {yes | no}
```

The default is **yes**.

TRISTATE2LOGIC (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. But in general tristate to logic replacement may lead to area increase. If the optimization goal is Area, you should apply TRISTATE2LOGIC (Convert Tristates to Logic) set to **no**.

TRISTATE2LOGIC values are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

There are some limitations to the TRISTATE2LOGIC constraint

- Only internal tristates are replaced by logic. The tristates of the top module connected to output pads are preserved.
- TRISTATE2LOGIC does not apply to technologies that do not have internal tristates, such as Spartan®-3 devices or Virtex®-4 devices. In this case, the conversion of tristates to logic is performed automatically. In some situations XST is unable to make the replacement automatically, due to the fact that this may lead to wrong design behavior or multi-source. This may happen when the hierarchy is preserved or XST does not have full design visibility (for example, design is synthesized on a block-by-block basis). In these cases, XST issues a warning at the low level optimization step. Depending on the particular design situation, you may continue the design flow and the replacement could be done by MAP, or you can force the replacement by applying Convert Tristates to Logic set to **yes** on a particular block or signal.
- The situations in which XST is unable to replace a tristate by logic are:
  - The tristate is connected to a black box.
  - The tristate is connected to the output of a block, and the hierarchy of the block is preserved.
  - The tristate is connected to a top-level output.
  - Convert Tristates to Logic is set to **no** on the block where tristates are placed, or on the signals to which tristates are connected.

TRISTATE2LOGIC (Convert Tristates to Logic) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

TRISTATE2LOGIC (Convert Tristates to Logic) Applicable Elements

Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal

TRISTATE2LOGIC (Convert Tristates to Logic) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

TRISTATE2LOGIC (Convert Tristates to Logic) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

TRISTATE2LOGIC (Convert Tristates to Logic) VHDL Syntax Example

Declare as follows:

```
attribute tristate2logic: string;
```

Specify as follows:

```
attribute tristate2logic of {entity_name|component_name|signal_name}:
{entity|component|signal} is "{yes|no}";
```

TRISTATE2LOGIC (Convert Tristates to Logic) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* tristate2logic = "{yes|no}" *)
```

TRISTATE2LOGIC (Convert Tristates to Logic) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" tristate2logic={yes|no|true|false};
```

TRISTATE2LOGIC (Convert Tristates to Logic) XST Constraint File (XCF) Example Two

```
BEGIN MODEL "entity_name"
 NET "signal_name" tristate2logic={yes|no|true|false};
END;
```

TRISTATE2LOGIC (Convert Tristates to Logic) XST Command Line Syntax Example

Define in the XST command line as follows:

```
run -tristate2logic {yes|no}
```

The default is **yes**.

TRISTATE2LOGIC (Convert Tristates to Logic) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Convert Tristates to Logic.

USE_CLOCK_ENABLE (Use Clock Enable)

USE_CLOCK_ENABLE (Use Clock Enable) enables or disables the clock enable function in flip-flops. The disabling of the clock enable function is typically used for ASIC prototyping on FPGA devices.

By detecting USE_CLOCK_ENABLE with a value of **no** or **false**, XST avoids using CE resources in the final implementation. Moreover, for some designs, putting the Clock Enable function on the data input of the flip-flop allows better logic optimization and therefore better QOR. In **auto** mode, XST tries to estimate a trade off between using a dedicated clock enable input of a flip-flop input and putting clock enable logic on the D input of a flip-flop. In a case where a flip-flop is instantiated by you, XST removes the clock enable only if the Optimize Instantiated Primitives option is set to **yes**.

Use USE_CLOCK_ENABLE values are:

- **auto** (default)
- **yes**
- **no**
- **true** (XCF only)
- **false** (XCF only)

USE_CLOCK_ENABLE (Use Clock Enable) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

USE_CLOCK_ENABLE (Use Clock Enable) Applicable Elements

Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

USE_CLOCK_ENABLE (Use Clock Enable) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

USE_CLOCK_ENABLE (Use Clock Enable) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

USE_CLOCK_ENABLE (Use Clock Enable) VHDL Syntax Example

Declare as follows:

```
attribute use_clock_enable: string;
```

Specify as follows:

```
attribute use_clock_enable of  {entity_name | component_name | signal_name | instance_name}:
{entity | component | signal | label} is "{auto | yes | no}";
```

USE_CLOCK_ENABLE (Use Clock Enable) Verilog Syntax Example

Place Use Clock Enable immediately before the instance, module or signal declaration:

```
(* use_clock_enable = "{auto | yes | no}" *)
```

USE_CLOCK_ENABLE (Use Clock Enable) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" use_clock_enable={auto | yes | no | true | false};
```

USE_CLOCK_ENABLE (Use Clock Enable) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" use_clock_enable={auto | yes | no | true | false};
END
```

USE_CLOCK_ENABLE (Use Clock Enable) XST Constraint File (XCF) Syntax Example Three

```
BEGIN MODEL
"entity_name ;"
INST "instance_name" use_clock_enable={auto | yes | no | true | false};
END
```

USE_CLOCK_ENABLE (Use Clock Enable) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -use_clock_enable {auto | yes | no}
```

The default is **auto**.

USE_CLOCK_ENABLE (Use Clock Enable) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx Specific Options > Use Clock Enable.

USE_SYNC_SET (Use Synchronous Set)

USE_SYNC_SET (Use Synchronous Set) enables or disables the synchronous set function in flip-flops. The disabling of the synchronous set function is typically used for ASIC prototyping on FPGA devices. Detecting Use Synchronous Set with a value of **no** or **false**, XST avoids using synchronous reset resources in the final implementation. Moreover, for some designs, putting synchronous reset function on data input of the flip-flop allows better logic optimization and therefore better QOR.

In **auto** mode, XST tries to estimate a trade off between using dedicated Synchronous Set input of a flip-flop input and putting Synchronous Set logic on the D input of a flip-flop. In a case where a flip-flop is instantiated by you, XST removes the synchronous reset only if the Optimize Instantiated Primitives option is set to **yes**.

USE_SYNC_SET (Use Synchronous Set) values are:

- **auto** (default)
- **yes**
- **no**
- **true** (XCF only)
- **false** (XCF only)

USE_SYNC_SET (Use Synchronous Set) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

USE_SYNC_SET (Use Synchronous Set) Applicable Elements

Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

USE_SYNC_SET (Use Synchronous Set) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

USE_SYNC_SET (Use Synchronous Set) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

USE_SYNC_SET (Use Synchronous Set) VHDL Syntax Example

Declare as follows:

```
attribute use_sync_set: string;
```

Specify as follows:

```
attribute use_sync_set of {entity_name | component_name | signal_name | instance_name}:
{entity | component | signal | label} is "{auto | yes | no}";
```

USE_SYNC_SET (Use Synchronous Set) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* use_sync_set = "{auto | yes | no}" *)
```

USE_SYNC_SET (Use Synchronous Set) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" use_sync_set={auto | yes | no | true | false};
```

USE_SYNC_SET (Use Synchronous Set) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name" use_sync_set={auto | yes | no | true | false};

END;
```

USE_SYNC_SET (Use Synchronous Set) XST Constraint File (XCF) Syntax Example Three

```
BEGIN MODEL "entity_name"

INST "instance_name" use_sync_set={auto | yes | no | true | false};

END;
```

USE_SYNC_SET (Use Synchronous Set) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -use_sync_set {auto | yes | no}
```

The default is **auto**.

USE_SYNC_SET (Use Synchronous Set) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > Use Synchronous Set.

USE_SYNC_RESET (Use Synchronous Reset)

USE_SYNC_RESET (Use Synchronous Reset) 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 FPGA devices.

Detecting USE_SYNC_RESET with a value of **no** or **false**, XST avoids using synchronous reset resources in the final implementation. Moreover, for some designs, putting synchronous reset function on data input of the flip-flop allows better logic optimization and therefore better QOR.

In **auto** mode, XST tries to estimate a trade off between using a dedicated Synchronous Reset input on a flip-flop input and putting Synchronous Reset logic on the D input of a flip-flop. In a case where a flip-flop is instantiated by you, XST removes the synchronous reset only if the Optimize Instantiated Primitives option is set to **yes**.

USE_SYNC_RESET (Use Synchronous Reset) values are:

- **auto** (default)
- **yes**
- **no**
- **true** (XCF only)
- **false** (XCF only)

USE_SYNC_RESET (Use Synchronous Reset) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

USE_SYNC_RESET (Use Synchronous Reset) Applicable Elements

Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

USE_SYNC_RESET (Use Synchronous Reset) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

USE_SYNC_RESET (Use Synchronous Reset) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

USE_SYNC_RESET (Use Synchronous Reset) VHDL Syntax Example

Declare as follows:

```
attribute use_sync_reset: string;
```

Specify as follows:

```
attribute use_sync_reset of {entity_name | component_name | signal_name | instance_name}:{entity
| component | signal | label} is "{auto | yes | no}";
```

USE_SYNC_RESET (Use Synchronous Reset) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* use_sync_reset = "{auto | yes | no}" *)
```

USE_SYNC_RESET (Use Synchronous Reset) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" use_sync_reset={auto | yes | no | true | false};
```

USE_SYNC_RESET (Use Synchronous Reset) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name"

NET "signal_name" use_sync_reset={auto | yes | no | true | false};

END;
```

USE_SYNC_RESET (Use Synchronous Reset) XST Constraint File (XCF) Syntax Example Three

```
BEGIN MODEL "entity_name"

INST "instance_name" use_sync_reset={auto | yes | no | true | false};

END;
```

USE_SYNC_RESET (Use Synchronous Reset) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -use_sync_reset {auto | yes | no}
```

The default is **auto**.

USE_SYNC_RESET (Use Synchronous Reset) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx Specific Options > Use Synchronous Reset.

USE_DSP48 (Use DSP48)

This constraint is called:

- Use DSP48 (Virtex®-4 devices)
- Use DSP Block (Virtex-5 devices and Spartan®-3A DSP devices)

XST enables you to use the resources of the DSP48 blocks introduced in Virtex-4 devices.

The default is **auto**.

In **auto** mode, XST automatically implements such macros as MAC and accumulates on DSP48, but some of them as adders are implemented on slices. You have to force their implementation on DSP48 using a value of **yes** or **true**. For more information on supported macros and their implementation control, see XST Hardware Description Language (HDL) Coding Techniques.

Several macros (for example, MAC) that can be placed on DSP48 are in fact a composition of simpler macros such as multipliers, accumulators, and registers. To achieve the best performance, XST by default tries to infer and implement the maximum macro configuration. To shape a macro in a specific way, use the Keep (KEEP) constraint. For example, DSP48 allows you to implement a multiple with two input registers. To leave the first register stage outside of the DSP48, place the Keep (KEEP) constraint in their outputs.

Use DSP48 values are:

- **auto** (default)
- **yes**
- **no**
- **true** (XCF only)
- **false** (XCF only)

In **auto** mode you can control the number of available DSP48 resources for synthesis using DSP Utilization Ratio (DSP_UTILIZATION_RATIO). By default, XST tries to utilize, as much as possible, all available DSP48 resources. For more information, see DSP48 Block Resources.

USE_DSP48 (Use DSP48) Architecture Support

Applies to the following FPGA devices only. Does not apply to any other FPGA devices. Does not apply to CPLD devices.

- Spartan-3A DSP
- Virtex-4
- Virtex-5

USE_DSP48 (Use DSP48) Applicable Elements

Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a macro described at the RTL level

USE_DSP48 (Use DSP48) Propagation Rules

Applies to the entity, component, module, or signal to which it is attached.

USE_DSP48 (Use DSP48) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

USE_DSP48 (Use DSP48) VHDL Syntax Example

Declare as follows:

```
attribute use_dsp48: string;
```

Specify as follows:

```
attribute use_dsp48 of "entity_name | component_name | signal_name}: {entity |
component | signal}is "{auto | yes | no}";
```

USE_DSP48 (Use DSP48) Verilog Syntax Example

Place immediately before the module or signal declaration:

```
(* use_dsp48 = "{auto | yes | no}" *)
```

USE_DSP48 (Use DSP48) XST Constraint File (XCF) Syntax Example One

```
MODEL "entity_name" use_dsp48={auto | yes | no | true | false};
```

USE_DSP48 (Use DSP48) XST Constraint File (XCF) Syntax Example Two

```
BEGIN MODEL "entity_name   NET "signal_name" use_dsp48={auto | yes | no | true | false};END;"
```

USE_DSP48 (Use DSP48) XST Command Line Syntax Example

Define in the XST command line as follows:

```
-use_dsp48 {auto|yes|no}
```

The default is **auto**.

USE_DSP48 (Use DSP48) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > HDL Options > Use DSP48.

XST CPLD Constraints (Non-Timing)

The following XST CPLD constraints (non-timing) apply to CPLD devices only. They do not apply to FPGA devices.

-pld_ce (Clock Enable)

`-pld_ce` (Clock Enable) 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.

Clock Enable values are:

- **`yes`**

  The synthesizer implements the clock enable signal of the device.

- **`no`**

  The clock enable function is 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, setting Clock Enable to **`no`** may improve the fitting result. The input data of the flip-flop is simplified when it is merged with the clock enable expression.

-pld_ce (Clock Enable) Architecture Support

Applies to all CPLD devices. Does not apply to FPGA devices.

-pld_ce (Clock Enable) Applicable Elements

Applies to an entire design through the XST command line.

-pld_ce (Clock Enable) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite in **Process > Properties > Xilinx-Specific Options > Clock Enable**.

-pld_ce (Clock Enable) Syntax

```
xst run -pld_ce {yes|no}
```

The default is **`yes`**.

-pld_ce (Clock Enable) Syntax Example

```
xst run -pld_ce yes
```

Defines Clock Enable globally to **`yes`** so that the clock enable function is implemented through equivalent logic.

DATA_GATE (Data Gate)

Data Gate (DATA_GATE) 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 Data Gate 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.

For more information, see DATA_GATE in the *Constraints Guide*.

DATA_GATE Architecture Support

Data Gate applies to CoolRunner™-II devices only.

-pld_mp (Macro Preserve)

-pld_mp (Macro Preserve) makes 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.

Macro Preserve values are:

- **yes**

  Macros are preserved and generated by Macro+.

- **no**

  Macros are rejected and generated by HDL synthesizer

Depending on the Flatten Hierarchy value, a rejected macro is either merged in the design logic, or becomes a hierarchical block as shown below.

Flatten Hierarchy Value	Disposition
yes	Merged in the design logic
no	Becomes a hierarchical block

Very small macros such as 2-bit adders and 4-bit multiplexers are always merged, independent of the Macro Preserve or Flatten Hierarchy options.

-pld_mp (Macro Preserve) Architecture Support

Applies to all CPLD devices. Does not apply to FPGA devices.

-pld_mp (Macro Preserve) Applicable Elements

Applies to the entire design.

-pld_mp (Macro Preserve) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite in **Process > Properties > Xilinx®-Specific Options > Macro Preserve**.

-pld_mp (Macro Preserve) Syntax

```
xst run -pld_mp {yes|no}
```

The default is **yes**.

-pld_mp (Macro Preserve) Syntax Example

```
xst run -pld_mp no
```

Macros are rejected and generated by HDL synthesizer.

NOREDUCE (No Reduce)

NOREDUCE (No Reduce):

- Prevents minimization of redundant logic terms that are typically included in a design to avoid logic hazards or race conditions
- Identifies the output node of a combinatorial feedback loop to ensure correct mapping

For more information, see NOREDUCE in the *Constraints Guide*.

-wysiwyg (WYSIWYG)

-wysiwyg (WYSIWYG) makes a netlist reflect the user specification as closely as possible. That is, all the nodes declared in the Hardware Description Language (HDL) design are preserved.

If WYSIWYG mode is enabled (**yes**), XST:

- Preserves all user internal signals (nodes)
- Creates SOURCE_NODE constraints in the NGC file for all these nodes
- Skips design optimization (collapse, factorization)

Only boolean equation minimization is performed.

-wysiwyg (WYSIWYG) Architecture Support

Applies to all CPLD devices. Does not apply to FPGA devices.

-wysiwyg (WYSIWYG) Applicable Elements

Applies to an entire design through the XST command line.

-wysiwyg (WYSIWYG) Propagation Rules

Not applicable.

-wysiwyg (WYSIWYG) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

-wysiwyg (WYSIWYG) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -wysiwyg {yes | no}
```

The default is **no**.

-wysiwyg (WYSIWYG) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Xilinx® Specific Options > WYSIWYG.

-pld_xp (XOR Preserve)

-pld_xp (XOR Preserve) enables or disables hierarchical flattening of XOR macros.

The XORs inferred by Hardware Description Language (HDL) synthesis are also considered as macro blocks in the CPLD flow. They are processed separately to give more flexibility for using 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.

XOR Preserve values are:

- **yes** (default)

  XOR macros are preserved
- **no**

  XOR macros are merged with surrounded logic

Preserving XORs generally gives better results. That is, the number of PTerms is lower. Use the **no** value to obtain completely flat netlists. Applying global optimization on a completely flat design sometimes improves design fitting.

To obtain a completely flattened design, select the following options:

- Flatten Hierarchy

  yes
- Macro Preserve

  no
- XOR Preserve

  no

The **no** value does not guarantee the elimination of the XOR operator from the Electronic Data Interchange Format (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 Hardware Description Language (HDL) synthesis, and is guided only by the goal of complexity reduction.

-pld_xp (XOR Preserve) Architecture Support

Applies to all CPLD devices. Does not apply to FPGA devices.

-pld_xp (XOR Preserve) Applicable Elements

Applies to the entire design.

-pld_xp (XOR Preserve) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite with **Process > Properties > Xilinx-Specific Options > XOR Preserve**.

-pld_xp (XOR Preserve) Syntax

```
xst run -pld_xp {yes|no}
```

The default is **yes**.

-pld_xp (XOR Preserve) Syntax Example

```
xst run -pld_xp no
```

XOR macros are merged with surrounded logic.

XST Timing Constraints

This section discusses how to apply XST timing constraints, and gives information on specific constraints.

Applying Timing Constraints

Apply XST-supported timing constraints with:

- Global Optimization Goal (**-glob_opt**)
- ISE® Design Suite**Process > Properties > Synthesis Options > Global Optimization Goal**
- User Constraints File (UCF)

Applying Timing Constraints Using Global Optimization Goal

Global Optimization Goal (**-glob_opt**) allows you to apply the five global timing constraints:

- ALLCLOCKNETS
- OFFSET_IN_BEFORE
- OFFSET_OUT_AFTER
- INPAD_TO_OUTPAD
- MAX_DELAY

These constraints are applied globally to the entire design. You cannot specify a value for these constraints, since XST optimizes them for the best performance. These constraints are overridden by constraints specified in the User Constraints File (UCF).

Applying Timing Constraints Using the User Constraints File (UCF)

The User Constraints File (UCF) allows you to specify timing constraints using native UCF syntax. XST supports constraints such as:

- Timing Name (TNM)
- Timegroup (TIMEGRP)
- Period (PERIOD)
- Timing Ignore (TIG)
- From-To (FROM-TO)

XST supports wildcards and hierarchical names with these constraints.

Writing Constraints to the NGC File

Timing constraints are not written to the NGC file by default. Timing constraints are written to the NGC file only when:

- **Write Timing Constraints** is checked **yes** in ISE Design Suite in **Process > Properties**, or
- The **-write_timing_constraints** option is specified when using the command line.

Additional Options Affecting Timing Constraint Processing

Three additional options affect timing constraint processing, regardless of how the timing constraints are specified:

- Cross Clock Analysis (-cross_clock_analysis)
- Write Timing Constraints (-write_timing_constraints)
- Clock Signal (CLOCK_SIGNAL)

-cross_clock_analysis (Cross Clock Analysis)

-cross_clock_analysis (Cross Clock Analysis) tells XST to perform inter-clock domain analysis during timing optimization. By default (**no**), XST does not perform this analysis.

-cross_clock_analysis (Cross Clock Analysis) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

-cross_clock_analysis (Cross Clock Analysis) Applicable Elements

Applies to the entire design.

-cross_clock_analysis (Cross Clock Analysis) Propagation Rules

Not applicable.

Set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Cross Clock Analysis**

.

-cross_clock_analysis (Cross Clock Analysis) Syntax

```
xst run -cross_clock_analysis {yes|no}
```

-cross_clock_analysis (Cross Clock Analysis) Syntax Example

```
xst run -cross_clock_analysis yes
```

Tells XST to perform inter-clock domain analysis during timing optimization.

-write_timing_constraints (Write Timing Constraints)

Timing constraints are written to the NGC file only when:

- **Write Timing Constraints** is checked **yes** in ISE® Design Suite in**Process > Properties**, or
- The **-write_timing_constraints** option is specified when using the command line.

Timing constraints are not written to the NGC file by default.

-write_timing_constraints (Write Timing Constraints) Architecture Support

Architecture independent.

-write_timing_constraints (Write Timing Constraints) Applicable Elements

Applies to an entire design through the XST command line.

-write_timing_constraints (Write Timing Constraints) Propagation Rules

Not applicable.

-write_timing_constraints (Write Timing Constraints) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

-write_timing_constraints (Write Timing Constraints) XST Command Line Syntax Example

Define in the XST command line as follows:

```
xst run -write_timing_constraints {yes | no}
```

The default is **yes**.

-write_timing_constraints (Write Timing Constraints) ISE Design Suite Syntax Example

Define in ISE® Design Suite with:

Process > Properties > Synthesis Options > Write Timing Constraints.

CLOCK_SIGNAL (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. CLOCK_SIGNAL (Clock Signal) allows you to define the clock signal.

CLOCK_SIGNAL (Clock Signal) Architecture Support

Applies to all FPGA devices. Does not apply to CPLD devices.

CLOCK_SIGNAL (Clock Signal) Applicable Elements

Applies to signals.

CLOCK_SIGNAL (Clock Signal) Propagation Rules

Applies to clock signals.

CLOCK_SIGNAL (Clock Signal) Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

CLOCK_SIGNAL (Clock Signal) VHDL Syntax Example

Declare as follows:

```
attribute clock_signal : string;
```

Specify as follows:

```
attribute  clock_signal of signal_name : signal is {yes | no};
```

CLOCK_SIGNAL (Clock Signal) Verilog Syntax Example

Place immediately before the signal declaration:

```
(* clock_signal = "{yes | no}" *)
```

CLOCK_SIGNAL (Clock Signal)l XST Constraint File (XCF) Syntax Example

```
BEGIN MODEL "entity_name" NET "primary_clock_signal" clock_signal={yes | no | true | false};END;
```

-glob_opt (Global Optimization Goal)

Depending on the Global Optimization Goal, XST can optimize the following design regions:

- Register to register
- Inpad to register
- Register to outpad
- Inpad to outpad)

-glob_opt (Global Optimization Goal) selects the global optimization goal. For a detailed description of supported timing constraints, see Partitions.

You cannot specify a value for Global Optimization Goal. XST optimizes the entire design for the best performance.

Apply the following constraints with Global Optimization Goal:

- **ALLCLOCKNETS** (register to register) optimizes the period of the entire design

  XST will identify, by default, all paths from register to register on the same clock for all clocks in a design. To take inter-clock domain delays into account, set Cross Clock Analysis (-cross_clock_analysis) to **yes**.

- **OFFSET_IN_BEFORE** (inpad to register) optimizes the maximum delay from input pad to clock, either for a specific clock or for an entire design.

  XST will identify all paths from either all sequential elements or the sequential elements driven by the given clock signal name to all primary output ports.

- **OFFSET_OUT_AFTER** (register to outpad) optimizes the maximum delay from clock to output pad, either for a specific clock or for an entire design.

  XST will identify all paths from all primary input ports to either all sequential elements or the sequential elements driven by the given clock signal name.

- **INPAD_TO_OUTPAD** (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. They apply only if no timing constraints are specified in the constraint file.

Set this value in ISE® Design Suite in **Process > Properties > Synthesis Options > Global Optimization Goal**.

-glob_opt (Global Optimization Goal) Syntax

```
-glob_opt
{allclocknets|offset_in_before|offset_out_after|inpad_to_outpad|max_delay}
```

-glob_opt (Global Optimization Goal) Syntax Example

xst run -glob_opt OFFSET_OUT_AFTER

Optimizes the maximum delay from clock to output pad for the entire design

Global Optimization Goal Domain Definitions

The possible domains are shown in the following schematic.

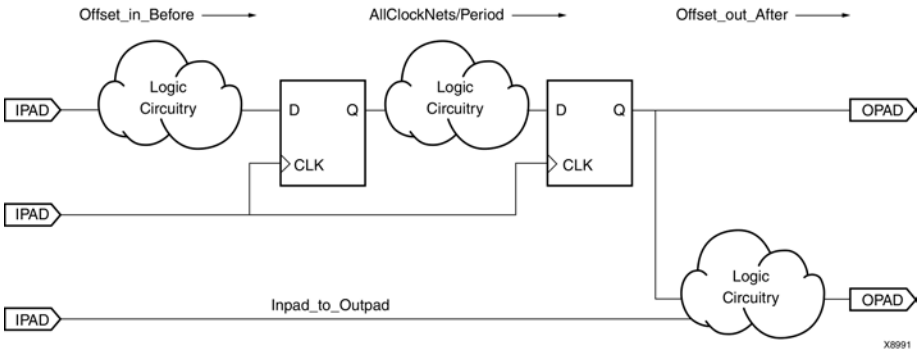
- ALLCLOCKNETS (register to register) Identifies, by default, all paths from register to register on the same clock for all clocks in a design. To take inter-clock domain delays into account, set Cross Clock Analysis (–cross_clock_analysis) to yes.
- OFFSET_IN_BEFORE (inpad to register) Identifies all 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) 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 Identifies all paths defined by the following timing constraints:

  ALLCLOCKNETS

  OFFSET_IN_BEFORE

  OFFSET_OUT_AFTER

  INPAD_TO_OUTPAD



XST Constraint File (XCF) Timing Constraint Support

If you specify timing constraints in the XST Constraint File (XCF), Xilinx® recommends that you use a forward slash (/) as a hierarchy separator instead of an underscore (_). For more information, see Hierarchy Separator (-hierarchy_separator).

If all or part of a specified timing constraint is not supported by XST, then XST issues a warning, 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 XCF:

- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)

PERIOD (Period)

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. For more information, see PERIOD in the *Constraints Guide*.

Period XST Constraint File (XCF) Syntax Example

```
NET netname   PERIOD = value [{HIGH | LOW} value];
```

OFFSET (Offset)

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

For more information, see OFFSET in the *Constraints Guide*.

OFFSET (Offset) Syntax Examples

```
OFFSET = {IN | OUT} offset_time [units] {BEFORE | AFTER}  clk_name[ TIMEGRP group_name];
```

FROM-TO (From-To)

From-To (FROM-TO) defines a timing constraint between two groups. A group can be user-defined or predefined (FFS, PADS, RAMS). For more information, see FROM-TO in the *Constraints Guide*.

From-To XST Constraint File (XCF) Syntax Example

```
TIMESPEC  TSname = FROM  group1 TO group2  value;
```

TNM (Timing Name)

TNM (Timing Name) is a basic grouping constraint. Use TNM 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 TNM constraints. For more information, see TNM in the *Constraints Guide*.

TNM (Timing Name) Syntax Examples

```
{INST | NET | PIN} inst_net_or_pin_name  TNM = [predefined_group :] identifier;
```

TNM_NET (Timing Name on a Net)

TNM_NET (Timing Name on a 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 DLL/DCMs. For more information, see PERIOD Specifications on CLKDLLs and DCMs in the *Constraints Guide*.

A TNM_NET is a property that you normally use in conjunction with a Hardware Description Language (HDL) design to tag a specific net. All downstream synchronous elements and pads tagged with the TNM_NET identifier are considered a group. For more information, see TNM_NET in the *Constraints Guide*.

TNM_NET (Timing Name on a Net) Syntax Examples

```
NET netname TNM_NET = [predefined_group :] identifier;
```

TIMEGRP (Timegroup)

TIMEGRP (Timegroup) 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.

Place TIMEGRP constraints in an XST Constraint File (XCF) or a Netlist Constraints File (NCF). Use TIMEGRP attributes to create groups using the following methods.

- Combining multiple groups into one
- Defining flip-flop subgroups by clock sense

For more information, see TIMEGRP in the *Constraints Guide*

TIMEGRP (Timegroup) Syntax Examples

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...];
```

TIG (Timing Ignore)

TIG (Timing Ignore) causes all paths going through a specific net to be ignored for timing analyses and optimization purposes. TIG can be applied to the name of the signal affected. For more information, see TIG in the *Constraints Guide*

TIG (Timing Ignore) XST Constraint File (XCF) Syntax Example

```
NET net_name TIG;
```

XST Implementation Constraints

Implementation constraints control placement and routing. They are not directly used by XST, but are propagated and made available to the implementation tools. The object to which an implementation constraint is attached is preserved.

A binary equivalent of the implementation constraint is written to the NGC file. Since the file is binary, you cannot edit an implementation constraint in the NGC file.

You can code an implementation constraint in the XST Constraint File (XCF) as illustrated in Implementation Constraints Syntax Examples.

For more information, see the *Constraints Guide*.

Implementation Constraints Syntax Examples

This section gives the following Implementation Constraints syntax examples:

- Implementation Constraints XST Constraint File (XCF) Syntax Examples
- Implementation Constraints VHDL Syntax Examples
- Implementation Constraints Verilog Syntax Examples

Implementation Constraints XST Constraint File (XCF) Syntax Examples

To apply an implementation constraint to an entire entity, use either of the following XST Constraint File (XCF) syntaxes:

```
MODEL EntityName PropertyName;
```

```
MODEL EntityName PropertyName=PropertyValue;
```

To apply an implementation constraint to specific instances, nets, or pins within an entity, use either of the 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;
```

Implementation Constraints VHDL Syntax Examples

Specify implementation constraints in VHDL as follows:

```
attribute PropertyName of {NetName | InstName | PinName} : {signal | label} is "PropertyValue";
```

Implementation Constraints Verilog Syntax Examples

Specify implementation constraints in Verilog as follows:

```
// synthesis attribute PropertyName of {NetName | InstName | PinName} is "PropertyValue";
```

In Verilog-2001, where descriptions precede the signal, module, or instance to which they refer, specify implementation constraints as follows:

```
(* PropertyName="PropertyValue" *)
```

RLOC

Applies to all FPGA devices. Does not apply to CPLD devices.

Use RLOC 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 the Verilog code:

```
// synthesis attribute RLOC of srl1 : "R9C0.S0";
```

You may specify the same attribute in the XST Constraint File (XCF) as follows:

```
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;
```

For more information, see RLOC in the *Constraints Guide*.

NOREDUCE

Applies to all CPLD devices. Does not apply to FPGA devices.

NOREDUCE prevents the optimization of the boolean equation generating a given signal. Assuming a local signal is assigned the arbitrary function below, and NOREDUCE 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);
```

Specify NOREDUCE in the XST Constraint File (XCF) as follows:

```
BEGIN MODEL ENTNAME
  NET s NOREDUCE;
  NET s KEEP;
END;
```

XST writes the following statements to the NGC file:

```
NET s NOREDUCE;
NET s KEEP;
```

For more information, see NOREDUCE in the or more information, see NOREDUCE in the

PWR_MODE (Power Mode)

PWR_MODE (Power Mode) 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";
```

Specify PWR_MODE in the XST Constraint File (XCF) as follows:

```
MODEL ENTNAME
  NET s PWR_MODE=LOW;
  NET s KEEP;
END;
```

XST writes the following statement to the NGC file:

```
NET s PWR_MODE=LOW;
NET s KEEP;
```

The Hardware Description Language (HDL) attribute can be applied to the signal on which XST infers the instance if:

- The attribute applies to an instance (for example, Pack I/O Registers Into IOBs (IOB), DRIVE, IOSTANDARD), and
- The instance is not available (not instantiated) in the HDL source

PWR_MODE (Power Mode) Architecture Support

Applies to all CPLD devices. Does not apply to FPGA devices.

XST-Supported Third Party Constraints

This section describes constraints of third-party synthesis vendors that are supported by XST.

XST Equivalents to Third Party Constraints

This section shows the XST equivalent for each of the third party constraints. For specific information on these constraints, see the vendor documentation.

Several third party constraints are automatically supported by XST, as shown in the table below. Constraints marked **yes** are fully supported. If a constraint is only partially supported, the support conditions are shown in the Automatic Recognition column.

The following rules apply:

- VHDL uses standard attribute syntax. No changes are needed to the Hardware Description Language (HDL) code.
- For Verilog with third party metacomment syntax, the metacomment syntax must be changed to conform to XST conventions. The constraint name and its value can be used as shown in the third party tool.
- For Verilog 2001 attributes, no changes are needed to the HDL code. The constraint is automatically translated as in the case of VHDL attribute syntax

XST Equivalents to Third Party Constraints

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
black_box	Synplicity	BoxType	N/A	VHDL, Verilog
black_box_pad_pin	Synplicity	N/A	N/A	N/A
black_box_tri_pins	Synplicity	N/A	N/A	N/A
cell_list	Synopsys	N/A	N/A	N/A
clock_list	Synopsys	N/A	N/A	N/A
Enum	Synopsys	N/A	N/A	N/A
full_case	Synplicity Synopsys	Full Case	N/A	Verilog
ispad	Synplicity	N/A	N/A	N/A
map_to_module	Synopsys	N/A	N/A	N/A
net_*name*	Synopsys	N/A	N/A	N/A
parallel_case	Synplicity Synopsys	Parallel Case	N/A	Verilog
return_port_*name*	Synopsys	N/A	N/A	N/A
resource_sharing directives	Synopsys	Resource Sharing	N/A	VHDL, Verilog
set_dont_touch_network	Synopsys	not required	N/A	N/A
set_dont_touch	Synopsys	not required	N/A	N/A
set_dont_use_cel_*name*	Synopsys	not required	N/A	N/A
set_prefer	Synopsys	N/A	N/A	N/A
state_vector	Synopsys	N/A	N/A	N/A
syn_allow_retiming	Synplicity	Register Balancing	N/A	VHDL, Verilog
syn_black_box	Synplicity	BoxType	Yes	VHDL, Verilog
syn_direct_enable	Synplicity	N/A	N/A	N/A
syn_edif_bit_format	Synplicity	N/A	N/A	N/A

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
syn_edif_scalar_format	Synplicity	N/A	N/A	N/A
syn_encoding	Synplicity	FSM Encoding Algorithm	Yes (The value *safe* is not supported for automatic recognition. Use Safe Implementation in XST to activate this mode.)	VHDL, Verilog
syn_enum_encoding	Synplicity	Enumerated Encoding	N/A	VHDL
syn_hier	Synplicity	Keep Hierarchy	Yes syn_hier = hardrecognized askeep_hierarchy = soft syn_hier = removerecognized askeep_hierarchy = no (XST supports only the values *hard* and *remove* for syn_hier in automatic recognition.)	VHDL, Verilog
syn_isclock	Synplicity	N/A	N/A	N/A
syn_keep	Synplicity	Keep	Yes (XST preserves the designated net in the final netlist, but does not attach any KEEP constraint to it.)	VHDL, Verilog
syn_maxfan	Synplicity	Max Fanout	Yes	VHDL, Verilog
syn_netlist_hierarchy	Synplicity	Netlist Hierarchy	N/A	VHDL, Verilog
syn_noarrayports	Synplicity	N/A	N/A	N/A
syn_noclockbuf	Synplicity	Buffer Type	Yes	VHDL, Verilog
syn_noprune	Synplicity	Optimize Instantiated Primitives	Yes	VHDL, Verilog
syn_pipeline	Synplicity	Register Balancing	N/A	VHDL, Verilog
syn_preserve	Synplicity	Equivalent Register Removal	Yes	VHDL, Verilog
syn_ramstyle	Synplicity	ram_extract and ram_style	Yes XST implements RAMs in **no_rw_check** mode regardless if **no_rw_check** is specified or not the **area** value is ignored	VHDL, Verilog
syn_reference_clock	Synplicity	N/A	N/A	N/A
syn_replicate	Synplicity	Register Duplication	Yes	VHDL, Verilog

Name	Vendor	XST Equivalent	Automatic Recognition	Available For
syn_romstyle	Synplicity	rom_extract and rom_style	Yes	VHDL, Verilog
syn_sharing	Synplicity	N/A	N/A	VHDL, Verilog
syn_state_machine	Synplicity	Automatic FSM Extraction	Yes	VHDL, Verilog
syn_tco <n>	Synplicity	N/A	N/A	N/A
syn_tpd <n>	Synplicity	N/A	N/A	N/A
syn_tristate	Synplicity	N/A	N/A	N/A
syn_tristatetomux	Synplicity	N/A	N/A	N/A
syn_tsu <n>	Synplicity	N/A	N/A	N/A
syn_useenables	Synplicity	N/A	N/A	N/A
syn_useioff	Synplicity	Pack I/O Registers Into IOBs	N/A	VHDL, Verilog
synthesis translate_off synthesis translate_on	Synplicity Synopsys	Translate Off Translate On	Yes	VHDL, Verilog
xc_alias	Synplicity	N/A	N/A	N/A
xc_clockbuftype	Synplicity	Buffer Type	N/A	VHDL, Verilog
xc_fast	Synplicity	FAST	N/A	VHDL, Verilog
xc_fast_auto	Synplicity	FAST	N/A	VHDL, Verilog
xc_global_buffers	Synplicity	BUFG (XST)	N/A	VHDL, Verilog
xc_ioff	Synplicity	Pack I/O Registers Into IOBs	N/A	VHDL, Verilog
xc_isgsr	Synplicity	N/A	N/A	N/A
xc_loc	Synplicity	LOC	Yes	VHDL, Verilog
xc_map	Synplicity	LUT_MAP	Yes (XST supports only the value lut for automatic recognition.)	VHDL, Verilog
xc_ncf_auto_relax	Synplicity	N/A	N/A	N/A
xc_nodelay	Synplicity	NODELAY	N/A	VHDL, Verilog
xc_padtype	Synplicity	I/O Standard	N/A	VHDL, Verilog
xc_props	Synplicity	N/A	N/A	N/A
xc_pullup	Synplicity	PULLUP	N/A	VHDL, Verilog
xc_rloc	Synplicity	RLOC	Yes	VHDL, Verilog
xc_fast	Synplicity	FAST	N/A	VHDL, Verilog
xc_slow	Synplicity	N/A	N/A	N/A
xc_uset	Synplicity	U_SET	Yes	VHDL, Verilog

Third Party Constraints Syntax Examples

This section contains the following third party constraints syntax examples:

- Third Party Constraints Verilog Syntax Example
- Third Party Constraints XST Constraint File (XCF) Syntax Example

Third Party Constraints Verilog Syntax Example

```
module testkeep (in1, in2, out1);
 input in1;
 input in2;
 output out1;
 (* keep = "yes" *) wire aux1;
 (* keep = "yes" *) wire aux2;
 assign aux1 = in1;
 assign aux2 = in2;
 assign out1 = aux1 & aux2;
endmodule
```

Third Party Constraints XST Constraint File (XCF) Syntax Example

The Keep (KEEP) constraint can also be applied through the separate synthesis constraint file:

```
BEGIN MODEL testkeep
 NET aux1 KEEP=true;
END;
```

These are the only two ways of preserving a signal/net in a Hardware Description Language (HDL) design and preventing optimization on the signal or net during synthesis.

Chapter 7

XST VHDL Language Support

This chapter explains how XST supports the VHSIC Hardware Description Language (VHDL), and provides details on VHDL supported constructs and synthesis options. This chapter includes:

- VHDL IEEE Support
- XST VHDL File Type Support
- Debugging Using Write Operation in VHDL
- VHDL Data Types
- VHDL Record Types
- VHDL Initial Values
- VHDL Objects
- VHDL Operators
- Entity and Architecture Descriptions in VHDL
- VHDL Combinatorial Circuits
- VHDL Sequential Circuits
- VHDL Functions and Procedures
- VHDL Assert Statements
- Using Packages to Define VHDL Models
- VHDL Constructs Supported in XST
- VHDL Reserved Words

For more information, see:

- IEEE *VHDL Language Reference Manual*
- XST Design Constraints
- VHDL Attribute Syntax

VHDL offers a broad set of constructs for compactly describing complicated logic:

- VHDL allows the description of the structure of a system — how it is decomposed into subsystems, and how those subsystems are interconnected.
- VHDL allows the specification of the function of a system using familiar programming language forms.
- VHDL allows the design of a system to be simulated before being implemented and manufactured. This feature allows you to test for correctness without the delay and expense of hardware prototyping.
- VHDL 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.

VHDL IEEE Support

XST supports:

- VHDL IEEE std 1076-1987
- VHDL IEEE std 1076-1993
- VHDL IEEE std 1076-2006 (partially implemented)

XST allows instantiation for VHDL IEEE std 1076-2006 when:

- The formal port is a buffer and the associated actual is an out
- The formal port is an out and the associated actual is a buffer

VHDL IEEE Conflicts

VHDL IEEE std 1076-1987 constructs are accepted if they do not conflict with VHDL IEEE std 1076-1993. In case of a conflict, Std 1076-1993 behavior overrides std 1076-1987.

In cases where:

- Std 1076-1993 requires a construct to be an erroneous case, but
- Std 1076-1987 accepts it,

XST issues a warning instead of an error. An error would stop analysis.

VHDL IEEE Conflict Example

Following is an example of a VHDL IEEE conflict:

- Std 1076-1993 requires an impure function to use the **`impure`** keyword while declaring a function.
- Std 1076-1987 has no such requirement.

In this case, XST:

- Accepts the VHDL code written for Std 1076-1987
- Issues a warning stating Std 1076-1993 behavior

Non-LRM Compliant Constructs in VHDL

XST supports some non-LRM compliant constructs. XST supports a specific non-LRM compliant construct when:

- The construct is supported by majority of synthesis or simulation third-party tools, and
- It is a real language limitation for design coding, and has no impact on quality of results or problem detection in the design.

For example, the LRM does not allow instantiation when the formal port is a **`buffer`** and the effective one is an **`out`** (and vice-versa).

XST VHDL File Type Support

XST supports a limited File Read and File Write capability for VHDL:

- Use File Read capability, for example, to initialize RAMs from an external file.
- Use File Write capability for debugging processes, or to write a specific constant or generic value to an external file.

For more information, see Initializing RAM Coding Examples.

Use any of the read functions shown in the following table. These read functions are supported by the following packages:

- **`standard`**
- **`std.textio`**
- **`ieee.std_logic_textio`**

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
readline (line, bit, boolean)	std.textio
read (line, bit)	std.textio
readline (line, bit_vector, boolean)	std.textio
read (line, bit_vector)	std.textio
read (line, boolean, boolean)	std.textio
read (line, boolean)	std.textio
read (line, character, boolean)	std.textio
read (line, character)	std.textio
read (line, string, boolean)	std.textio
read (line, string)	std.textio
write (file, line)	std.textio
write (line, bit, boolean)	std.textio
write (line, bit)	std.textio
write (line, bit_vector, boolean)	std.textio
write (line, bit_vector)	std.textio
write (line, boolean, boolean)	std.textio
write (line, boolean)	std.textio
write (line, character, boolean)	std.textio
write (line, character)	std.textio
write (line, integer, boolean)	std.textio
write (line, integer)	std.textio
write (line, string, boolean)	std.textio
write (line, string)	std.textio
read (line, std_ulogic, boolean)	ieee.std_logic_textio
read (line, std_ulogic)	ieee.std_logic_textio

Function	Package
read (line, std_ulogic_vector), boolean	ieee.std_logic_textio
read (line, std_ulogic_vector)	ieee.std_logic_textio
read (line, std_logic_vector, boolean)	ieee.std_logic_textio
read (line, std_logic_vector)	ieee.std_logic_textio
write (line, std_ulogic, boolean)	ieee.std_logic_textio
write (line, std_ulogic)	ieee.std_logic_textio
write (line, std_ulogic_vector, boolean)	ieee.std_logic_textio
write (line, std_ulogic_vector)	ieee.std_logic_textio
write (line, std_logic_vector, boolean)	ieee.std_logic_textio
write (line, std_logic_vector)	ieee.std_logic_textio
hread	ieee.std_logic_textio

Debugging Using Write Operation in VHDL Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Debugging Using Write Operation in VHDL Coding Example

```
--
-- Print 2 constants to the output file
--

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use STD.TEXTIO.all;
use IEEE.STD_LOGIC_TEXTIO.all;

entity file_support_1 is
    generic (data_width: integer:= 4);
    port( clk, sel: in std_logic;
            din: in std_logic_vector (data_width - 1 downto 0);
            dout: out std_logic_vector (data_width - 1 downto 0));
end file_support_1;

architecture Behavioral of file_support_1 is
    file results : text is out "test.dat";
    constant base_const: std_logic_vector(data_width - 1 downto 0):= conv_std_logic_vector(3,data_width);
    constant new_const:  std_logic_vector(data_width - 1 downto 0):= base_const + "1000";
begin

    process(clk)
        variable txtline : LINE;
    begin
        write(txtline,string'("--------------------"));
        writeline(results, txtline);
        write(txtline,string'("Base Const: "));
        write(txtline,base_const);
        writeline(results, txtline);


        write(txtline,string'("New  Const: "));
        write(txtline,new_const);
        writeline(results, txtline);
        write(txtline,string'("--------------------"));
        writeline(results, txtline);

        if (clk'event and clk='1') then
            if (sel = '1') then
                dout <= new_const;
            else
                dout <= din;
            end if;
        end if;
    end process;

end Behavioral;
```

Rules for Debugging Using Write Operation in VHDL

Follow these rules for rules for debugging using write operation in VHDL:

- During a **std_logic** read operation, the only allowed characters are **0** and **1**. Other values such as **X** and **Z** are not allowed. XST rejects the design if the file includes characters other than **0** and **1**, except that XST ignores a blank space character.
- Do not use identical names for files placed in different directories.
- Do not use conditional calls to read procedures, as shown in the following coding example:

```
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 issues 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 coding example:

```
while (not endfile (MY_FILE)) loop
  readline (MY_FILE, MY_LINE);
  exit when endfile (MY_FILE);
  read (MY_LINE, MY_DATA);
end loop;
```

VHDL Data Types

This section discusses VHDL Data Types, including:

- Accepted VHDL Data Types
- VHDL Overloaded Data Types
- VHDL Multi-Dimensional Array Types

Accepted VHDL Data Types

XST accepts the following VHDL data types:

- VHDL Enumerated Types
- VHDL User-Defined Enumerated Types
- VHDL Bit Vector Types
- VHDL Integer Types
- VHDL Predefined Types
- VHDL STD_LOGIC_1164 IEEE Types

VHDL Enumerated Types

Type	Values	Meaning	Comment
BIT	0, 1	--	--
BOOLEAN	false, true	--	--
REAL	$-. to $+.	--	--
STD_LOGIC	U	unitialized	Not accepted by XST
	X	unknown	Treated as don't care
	0	low	Treated identically to L
	1	high	Treated identically to H
	Z	high impedance	Treated as high impedance
	W	weak unknown	Not accepted by XST
	L	weak low	Treated identically to 0
	H	weak high	Treated identically to 1
	-	don't care	Treated as don't care

VHDL User-Defined Enumerated Types

type COLOR is (RED, GREEN, YELLOW) ;

VHDL Bit Vector Types

- BIT_VECTOR
- STD_LOGIC_VECTOR

Unconstrained types (types whose length is not defined) are not accepted.

VHDL Integer Types

INTEGER

VHDL Predefined Types

- BIT
- BOOLEAN
- BIT_VECTOR
- INTEGER
- REAL

VHDL STD_LOGIC_1164 IEEE Types

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. To use one of these types, add the following two lines to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

VHDL Overloaded Data Types

The following data types can be overloaded:

- VHDL Overloaded Enumerated Types
- VHDL Overloaded Bit Vector Types
- VHDL Overloaded Integer Types
- VHDL Overloaded STD_LOGIC_1164 IEEE Types
- VHDL Overloaded STD_LOGIC_ARITH IEEE Types

VHDL 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*, and *Z* values

VHDL Overloaded Bit Vector Types

- STD_ULOGIC_VECTOR
- UNSIGNED
- SIGNED

Unconstrained types (types whose length is not defined) are not accepted.

VHDL Overloaded Integer Types

- NATURAL
- POSITIVE

Any integer type within a user-defined range. For 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.

VHDL Overloaded STD_LOGIC_1164 IEEE Types

The following types are declared in the STD_LOGIC_1164 IEEE package:

- STD_ULOGIC (and subtypes X01, X01Z, UX01, UX01Z)
- STD_LOGIC
- STD_ULOGIC_VECTOR
- STD_LOGIC_VECTOR

This package is compiled in the library IEEE. To use one of these types, add the following two lines to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

VHDL Overloaded STD_LOGIC_ARITH IEEE Types

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. To use these types, add the following two lines to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_ARITH.all;
```

VHDL Multi-Dimensional Array Types

XST supports multi-dimensional array types of up to three dimensions. BRAMs are not inferred. Arrays can be:

- Signals
- Constants
- 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, as shown in the following coding example:

```
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 shown in the following coding example:

```
subtype TAB13 is array (7 downto 0,4 downto 0) of STD_LOGIC_VECTOR (8 downto 0);
```

The following coding examples demonstrate the uses of multi-dimensional array signals and variables in assignments.

Multi-Dimensional Array VHDL Coding Example One

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 CNST_A : TAB03 := (
("00000000","01000001","01000010","10000011","00001100"),

("00100000","00100001","00101010","10100011","00101100"),

("01000010","01000010","01000100","01000111","01000100"));
```

The following can now be specified:

- A multi-dimensional array signal or variable: **`TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;`**
- An index of one array: **`TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;`**
- Indexes of the maximum number of dimensions: **`TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'`**
- A slice of the first array:

  ```
  TAB_A (4 downto 1) <= TAB_B (3 downto 0);
  ```

- An index of a higher level array and a slice of a lower level array: **`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)`**

Multi-Dimensional Array VHDL Coding Example Two

Add the following declaration:
`subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR (7 downto 0);signal MATRIX_A : MATRIX15;`

The following can now be specified:

- A multi-dimensional array signal or variable: **`MATRIXA <= CNST_A;`**
- An index of one row of the array: **`MATRIXA (5) <= TAB_A;`**
- Indexes of the maximum number of dimensions: **`MATRIXA (5,0) (0) <= '1';`**

Indices may be variable.

VHDL Record Types

XST supports record types, as shown in the following coding example:

```
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.

VHDL Initial Values

In VHDL, you can initialize registers when you declare them.

The value:

- Is 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. The assigned value is carried in the NGC file as an INIT attribute on the register, and is independent of any local reset.

```
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 in behavioral VHDL code. Assign a value to a register when the register reset line goes to the appropriate value. See the following coding 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.

VHDL Local Reset/Global Reset

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 Local Reset/Global Reset VHDL Coding 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.

Local Reset/Global Reset VHDL Coding Example

The following coding example sets the initial value on the register output to **1** (one) at initial power up, but since this is dependent upon a local reset, the value changes to **0** (zero) whenever the local set/reset is activated.

```
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;
```

Default Initial Values on Memory Elements in VHDL

Because every memory element in a Xilinx® FPGA device must come up in a known state, in certain cases, XST does not use IEEE standards for initial values. In the Local Reset/Global Reset VHDL Coding Example, if signal **arb_onebit** were not initialized to **1** (one), XST would assign it a default of **0** (zero) 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 shown in the XST column in the following table.

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**)

Unconnected output ports default to the values shown in the XST column of VHDL Initial Values. 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 issues an error message if an input port is not connected. Even the **open** keyword is not sufficient for an unconnected input port.

VHDL Objects

VHDL objects include:

- Signals
- Variables
- Constants

Signals in VHDL

Signals in VHDL 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 **<=**.

```
signal sig1 : std_logic;
sig1 <= '1';
```

Variables in VHDL

Variables in VHDL are declared in a process or a subprogram, and used within that process or that subprogram. Variables can be assigned by the assignment operator **:=**.

```
variable var1 :  std_logic_vector (7 downto 0); var1 := "01010011";
```

Constants in VHDL

Constants in VHDL can be declared in any declarative region, and can be used within that region. Their values cannot be changed once declared.

```
signal sig1 : std_logic_vector (5 downto 0);
constant init0 : std_logic_vector (5 downto 0) := "010111";
sig1 <= init0;
```

VHDL Operators

Supported operators are listed in VHDL Operators. This section provides examples of how to use each shift operator.

Operators VHDL Coding Example One

```
sll (Shift Left Logical)
sig1 <= A(4 downto 0) sll 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & "00";
```

Operators VHDL Coding Example Two

```
srl (Shift Right Logical)
sig1 <= A(4 downto 0) srl 2
```

logically equivalent to:

```
sig1 <= "00" & A(4 downto 2);
```

Operators VHDL Coding Example Three

```
sla (Shift Left Arithmetic)
sig1 <= A(4 downto 0) sla 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & A(0) & A(0);
```

Operators VHDL Coding Example Four

```
sra (Shift Right Arithmetic)
sig1 <= A(4 downto 0) sra 2
```

logically equivalent to:

```
sig1 <= <= A(4) & A(4) & A(4 downto 2);
```

Operators VHDL Coding Example Five

```
rol (Rotate Left)
sig1 <= A(4 downto 0) rol 2
```

logically equivalent to:

```
sig1 <= A(2 downto 0) & A(4 downto 3);
```

Operators VHDL Coding Example Six

```
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 in VHDL

Entity and architecture descriptions in VHDL include:

- Circuit Descriptions
- Entity Declarations
- Architecture Declarations
- Component Instantiation
- Recursive Component Instantiation
- Component Configuration
- Generic Parameter Declarations
- Generic and Attribute Conflict

VHDL Circuit Descriptions

A circuit description in VHDL consists of two parts:

- The interface (defining the I/O ports)
- The body

In VHDL:

- The entity corresponds to the interface
- The architecture describes the behavior

VHDL Entity Declarations

The I/O ports of the circuit are declared in the entity. Each port has:

- A name
- A mode (in, out, inout, or buffer)
- A type (ports A, B, C, D, E in the Entity and Architecture Declaration VHDL Coding Example)

Types of ports must be constrained. Not more than one-dimensional array types are accepted as ports.

VHDL Architecture Declarations

Internal signals may be declared in the architecture. Each internal signal has:

- A name
- A type (signal T in the *Entity and Architecture Declaration VHDL Coding Example*)

Entity and Architecture Declaration VHDL Coding Example

```
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;
```

VHDL Component Instantiation

Structural descriptions assemble several blocks, and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are:

- Component

  The component is the building or basic block.
- Port

  A port is a component I/O connector.
- Signal

  A signal corresponds to a wire between components.

In VHDL, a component is represented by a design entity. The design entity is a composite consisting of:

- Entity declaration

  The entity declaration provides the external view of the component. It describes what can be seen from the outside, including the component ports.
- Architecture body

  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. The association list specifies which actual signals or ports are associated with which local ports of the component declaration.

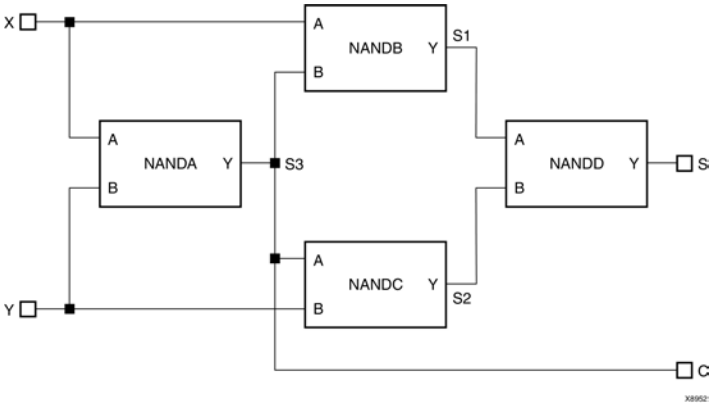
XST supports unconstrained vectors in component declarations.

Structural Description of Half Adder VHDL Coding Example

The following coding example shows the structural description of a half adder composed of four nand2 components:

```
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;
```

Synthesized Top Level Netlist Diagram



VHDL Recursive Component Instantiation

XST supports recursive component instantiation. Direct instantiation is not supported for recursion. To prevent endless recursive calls, the number of recursions is limited by default to 64. Use the **`-recursion_iteration_limit`** option to control the number of allowed recursive calls.

4-Bit Shift Register With Recursive Component Instantiation VHDL Coding Example

```
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;
```

VHDL Component Configuration

Associating an entity and architecture pair to a component instance provides the means of linking components with the appropriate model (entity and architecture pair). XST supports component configuration in the declarative part of the architecture:

```
for instantiation_list : component_name use LibName.entity_Name(Architecture_Name) ;
```

The Structural Description of Half Adder VHDL Coding Example 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.

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 or architecture is found, a black box is generated during synthesis.

VHDL Generic Parameter Declarations

Generic parameters may be declared in the entity declaration part. XST supports all types for generics including, for example:

- Integer
- Boolean
- String
- Real
- Std_logic_vector

An example of using generic parameters is 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 the following coding example.

Describing Circuits With Generic Ports VHDL Coding Example

```
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;
```

The GENERICS command line option allows you to redefine generics (VHDL) values defined in the top-level design block. This allows you to easily modify the design configuration without any Hardware Description Language (HDL) source modifications, such as for IP core generation and testing flows. For more information, see Generics (-generics)
.

VHDL Generic and Attribute Conflicts

Since generics and attributes can be applied to both instances and components in the VHDL code, and attributes can also be specified in a constraints file, from time to time, conflicts may 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 XST Constraint File (XCF) always takes precedence over attributes or generics specified in the VHDL code.

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 as a guide in determining precedence in VHDL.

Precedence in VHDL

	Generic on an Instance	Generic on a Component
Attribute on an Instance	Apply Generic (XST issues warning)	Apply Attribute (possible simulation mismatch)
Attribute on a Component	Apply Generic	Apply Generic (XST issues warning)
Attribute in XCF	Apply Attribute XST issues warning)	Apply Attribute

Security attributes on the block definition always have higher precedence than any other attribute or generic.

VHDL Combinatorial Circuits

XST supports the following VHDL combinatorial circuits:

- Concurrent Signal Assignments
- Generate Statements
- Combinatorial Processes
- If...Else Statements
- Case Statements
- For...Loop Statements

VHDL Concurrent Signal Assignments

Combinatorial logic in VHDL may be described using concurrent signal assignments. These can be defined within the body of the architecture. VHDL offers three types of concurrent signal assignments:

- Simple
- Selected
- Conditional

You can describe as many concurrent statements as needed. The order of concurrent signal definition in the architecture is irrelevant.

A concurrent assignment consists of two sides:

- Left hand
- Right hand

The assignment changes when any signal in the right side changes. In this case, the result is assigned to the signal on the left side.

Simple Signal Assignment VHDL Coding Example

```
T <= A and B;
```

MUX Description Using Selected Signal Assignment VHDL Coding Example

```
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;
```

MUX Description Using Conditional Signal Assignment VHDL Coding Example

```
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;
```

VHDL Generate Statements

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.

8-Bit Adder Described With For...Generate Statement VHDL Coding Example

The following coding example describes an 8-bit adder by declaring the bit slice structure (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. The *N-Bit Adder Described With If...Generate and For… Generate Statement VHDL Coding Example* shows such an example. It is a generic N-bit adder with a width ranging between 4 and 32 (N-bit adder described with an **`if...generate`** and a **`for… generate`** statement).

N-Bit Adder Described With If...Generate and For… Generate Statement VHDL Coding Example

```
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;
```

VHDL Combinatorial Processes

A process assigns values to signals differently than when using concurrent signal assignments. The value assignments are made in a sequential mode. Later assignments may cancel previous ones. See Assignments in a Process VHDL Coding Example. First the signal *S* is assigned to **`0`**, but later on *(for (A and B) =1)*, the value for *S* is changed to **`1`**.

A process is 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, the process is 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 in conditions (for example, **`if`** and **`case`**)
- All signals on the right hand side of an assignment

If one or more signals are missing from the sensitivity list, XST issues a warning message 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 Combinatorial Process VHDL Coding Example One, 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.

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

Following are examples of each of these statements.

Assignments in a Process VHDL Coding Example

```
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;
```

Combinatorial Process VHDL Coding Example One

```
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;
```

Combinatorial Process VHDL Coding Example Two

```
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;
```

VHDL If...Else Statements

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.

If...Else Statement VHDL Coding Example

```
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;
```

VHDL Case Statements

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.

Case Statement VHDL Coding Example

```
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;
```

VHDL For...Loop Statements

The **for** statement is supported for:

- Constant bounds
- Stop test condition using any of the following operators:
  - <
  - <=
  - >
  - >=
- 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

For...Loop Statement VHDL Coding Example

```
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;
```

VHDL Sequential Circuits

Sequential circuits can be described using sequential processes. XST allows:

- VHDL Sequential Process With a Sensitivity List
- VHDL Sequential Process Without a Sensitivity List

VHDL 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).

The Sequential Process With Asynchronous, Synchronous Parts VHDL Coding Example provides a template for describing sequential circuits. For more information, see XST Hardware Description Language (HDL) Coding Techniques, describing macro inference (for example, registers and counters).

Sequential Process With Asynchronous, Synchronous Parts VHDL Coding Example

Declare asynchronous signals in the sensitivity list. Otherwise, XST issues 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.

```
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;
```

VHDL 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. For more information, see VHDL Multiple Wait Statements Descriptions. An asynchronous part cannot be specified within processes without a sensitivity list.

Sequential Process Without a Sensitivity List VHDL Coding Example

The following VHDL coding example shows the skeleton of the process described in this section. The clock condition may be a falling or a rising edge.

```
process ...
begin
  wait until <CLK'EVENT | not CLK' STABLE> and CLK = <'0' | '1'>;
    ... -- a synchronous part may be specified here.
end process;
```

XST does not support clock and clock enable descriptions within the same Wait statement. Instead, code these descriptions as shown in Clock and Clock Enable (Not Supported) VHDL Coding Example.

XST does not support Wait statements for latch descriptions.

Clock and Clock Enable (Not Supported) VHDL Coding Example

```
wait until CLOCK'event and CLOCK = '0' and ENABLE = '1' ;
```

Clock and Clock Enable (Supported) VHDL Coding Example

```
"8 Bit Counter Description Using a Process with a Sensitivity List" if ENABLE = '1' then ...
```

Register and Counter Descriptions VHDL Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

8-Bit Register Description Using a Process With a Sensitivity List Example VHDL Coding Example

```
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;
```

8 Bit Register Description Using a Process Without a Sensitivity List Containing a Wait Statement VHDL Coding Example

```
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;
```

8-Bit Register With Clock Signal and Asynchronous Reset Signal VHDL Coding Example

```
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;
```

8-Bit Counter Description Using a Process With a Sensitivity List VHDL Coding Example

```
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;
```

VHDL Multiple Wait Statements Descriptions

Sequential circuits can be described in VHDL with multiple **Wait** statements in a process. Follow these rules when using multiple **Wait** statements:

- The process contains only one **Loop** statement.
- The first statement in the loop is a **Wait** statement.
- After each **Wait** statement, a **Next** or **Exit** statement is defined.
- The condition in the **Wait** statements is the same for each **Wait** statement.
- This condition use only one signal — the clock signal.
- This condition has the following form:

```
"wait [on clock_signal] until [(clock_signal'EVENT |     not clock_signal'STABLE)
and ] clock_signal = {'0' | '1'};"
```

Sequential Circuit Using Multiple Wait Statements VHDL Coding Example

The following VHDL coding example 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 following four inputs to the output **RESULT**:

- **DATA1**
- **DATA2**
- **DATA3**
- **DATA4**

```
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;
```

VHDL Functions and Procedures

The declaration of a function or a procedure in VHDL 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:

- Input parameters for functions and input
- Output and inout parameters for procedures.

These parameters can be unconstrained. 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.

Function Declaration and Function Call VHDL Coding Example

The following VHDL coding example shows a function declared within a package. The ADD function declared here is a single bit adder. This function is called four times with the proper parameters in the architecture to create a 4-bit adder. The same example using a procedure is shown in Procedure Declaration and Procedure Call VHDL Coding Example.

```
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;
```

Procedure Declaration and Procedure Call VHDL Coding Example

```
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;
```

Recursive Function VHDL Coding Example

XST supports recursive functions. The following coding example represents **n!** 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;
```

VHDL Assert Statements

XST supports Assert statements. Assert statements enable you to detect undesirable conditions in 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, issues a warning message, or rejects the design and issues an error message. XST supports the Assert statement only with static condition.

The following coding example contains a block, **SINGLE_SRL**, which 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 Shift Register LUT (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_SR**L 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.

SINGLE_SRL Describing a Shift Register VHDL Coding Example

```
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;
```

SINGLE_SRL Describing a Shift Register VHDL Error Message

If you run the SINGLE_SRL Describing a Shift Register VHDL Coding Example, XST issues the following error message:

```
...
=========================================================================
*                            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
...
```

Using Packages to Define VHDL Models

VHDL models may be defined using packages. Packages contain:

- Type and subtype declarations
- Constant definitions
- Function and procedure definitions
- Component declarations

Using packages to define VHDL models provides the ability to change parameters and constants of the design, such as constant values and function definitions.

Packages may contain two declarative parts:

- Body declaration
- Package 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, include the following lines 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*.

Using Standard Packages to Define VHDL Models

The Standard package contains basic types:

- **bit**
- **bit_vector**
- **integer**

The Standard package is included by default.

Using IEEE Packages to Define VHDL Models

XST supports the following IEEE packages:

- **`std_logic_1164`**

  Supports the following types:

  – **`std_logic`**
  – **`std_ulogic`**
  – **`std_logic_vector`**
  – **`std_ulogic_vector`**

  It also supports conversion functions based on these types.

- **`numeric_bit`**

  Supports the following types based on type bit:

  – Unsigned vectors
  – Signed vectors

  It also supports:

  – All overloaded arithmetic operators on these types
  – Conversion and extended functions for these types

- **`numeric_std`**

  Supports the following types based on type **`std_logic`**:

  – Unsigned vectors
  – Signed vectors

  This package is equivalent to **`std_logic_arith`**.

- **`math_real`**

  Supports the following:

  – Real number constants as shown in VHDL Real Number Constants
  – Real number functions as shown in VHDL Real Number Constants
  – The procedure **`uniform`**, which generates successive values between 0.0 and 1.0

VHDL Real Number Constants

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		math_log2_of_e	log2e
math_2_pi		math_log10_of_e	log10e
math_1_over_pi		math_sqrt_2	
math_pi_over_2		math_1_oversqrt_2	
math_pi_over_3		math_sqrt_pi	
math_pi_over_4		math_deg_to_rad	
math_3_pi_over_2		math_rad_to_deg	

VHDL Real Number Functions

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)	

Functions and procedures in the **math_real** packages, as well as the **real** type, are for calculations only. They are not supported for synthesis in XST.

Following is an 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.
```

Using Synopsys Packages to Define VHDL Models

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, such as:

  - **and_reduce**
  - **or_reduce**

VHDL Constructs Supported in XST

XST supports the following VHDL Constructs:

- Design Entities and Configurations
- Expressions
- Statements

VHDL Design Entities and Configurations

XST supports VHDL design entities and configurations except as noted below.

- VHDL Entity Headers
  - Generics

    Supported, but integer type only
  - Ports

    Supported, but unconstrained ports are not allowed
  - Entity Statement Part

    Unsupported
- VHDL Packages

  STANDARD

  Type TIME is not supported
- VHDL Physical Types
  - TIME

    Ignored
  - REAL

    Supported, but only in functions for constant calculations
- VHDL Modes

  Linkage

  Unsupported
- VHDL Declarations

  Type

  Supported for enumerated types, types with positive range having constant bounds, bit vector types, and multi-dimensional arrays
- VHDL Objects
  - Constant Declaration

    Supported except for deferred constant
  - Signal Declaration

    Supported except for register and bus type signals
  - Attribute Declaration

    Supported for some attributes, otherwise skipped (see XST Design Constraints)
- VHDL Specifications
  - Attribute

    Supported for some predefined attributes only: **`HIGHLOW, LEFT, RIGHT, RANGE, REVERSE_RANGE, LENGTH, POS, ASCENDING, EVENT, LAST_VALUE`**
  - Configuration

    Supported only with the **`all`** clause for instances list. If no clause is added, XST looks for the entity or architecture compiled in the default library
  - Disconnection

    Unsupported

XST does not allow underscores as the first character of signal names (for example, **`_DATA_1`**).

VHDL Expressions

XST supports the following expressions:

- VHDL Operators
- VHDL Operands

VHDL Operators

Operator	Supported/Unsupported
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 if the right operand is a constant power of 2
Shift Operators: sll, srl, sla, sra, rol, ror	Supported
abs	Supported
**	Only supported if the left operand is 2
Sign: +, -	Supported

VHDL Operands

Operand	Supported/Unsupported
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

VHDL Statements

XST supports all VHDL statements except as noted below:

- VHDL Wait Statements
- VHDL Loop Statements
- VHDL Concurrent Statements

VHDL Wait Statements

Wait Statement	Supported/Unsupported
Wait on *sensitivity_list* until *Boolean_expression*. For more information, see VHDL Sequential Circuits.	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 *time_expression*... For more information, see VHDL Sequential Circuits.	Unsupported
Assertion Statement	Supported (only for static conditions)
Signal Assignment Statement	Supported (delay is ignored)

VHDL Loop Statements

Loop Statement	Supported/Unsupported
for... loop... end loop	Supported for constant bounds only. Disable statements are not supported.
loop ... end loop	Only supported in the particular case of multiple Wait statements

VHDL Concurrent Statements

Concurrent Statement	Supported/Unsupported
Concurrent Signal Assignment Statement	Supported (no after clause, no transport or **guarded** options, no waveforms) **UNAFFECTED** is supported.
For ... Generate	Statement supported for constant bounds only
If ... Generate	Statement supported for static condition only

VHDL Reserved Words

abs	access	after	alias
all	and	architecture	array
assert	attribute	begin	block
body	buffer	bus	case
component	configuration	constant	disconnect
downto	else	elsif	end
entity	exit	file	for
function	generate	generic	group
guarded	if	impure	in
inertial	inout	is	label
library	linkage	literal	loop
map	mod	nand	new
next	nor	not	null
of	on	open	or
others	out	package	port
postponed	procedure	process	pure
range	record	register	reject
rem	report	return	rol
ror	select	severity	signal
shared	sla	sll	sra
srl	subtype	then	to
transport	type	unaffected	units
until	use	variable	wait
when	while	with	xnor
xor			

Chapter 8

XST Verilog Language Support

This chapter describes XST support for Verilog constructs and meta comments, and includes:

- Behavioral Verilog
- Variable Part Selects
- Structural Verilog Features
- Verilog Parameters
- Verilog Parameter and Attribute Conflicts
- Verilog Limitations in XST
- Verilog Attributes and Meta Comments
- Verilog Constructs Supported in XST
- Verilog System Tasks and Functions Supported in XST
- Verilog Primitives
- Verilog Reserved Keywords
- Verilog-2001 Support in XST

For more information, see:

- Verilog design constraints and options
  XST Design Constraints
- Verilog attribute syntax
  Verilog-2001 Attributes
- Setting Verilog options in the Process window of ISE® Design Suite
  XST General Constraints

Complex circuits are commonly designed using a top down methodology. Various specification levels are required at each stage of the design process. For 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 where the connections are N-bit wires, such as:

- Register
- Adder
- Counter
- Multiplexer
- Glue logic
- Finite State Machine (FSM)

A Hardware Description Language (HDL) such as Verilog allows the expression of notations such as ASM charts and circuit diagrams in a computer language.

Verilog provides both behavioral and structural language structures. These structures allow expressing design objects at high and low levels of abstraction. Designing hardware with a language such as Verilog allows using 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 Hardware Description Language (HDL) statements into a flattened gate-level netlist, which can then be used to custom program a programmable logic device such as a Virtex® device. Different synthesis methods are used for arithmetic blocks, glue logic, and Finite State Machine (FSM) components.

The *XST User Guide* assumes that you are familiar with basic Verilog concepts. For more information, see the *IEEE Verilog HDL Reference Manual*.

Behavioral Verilog

For information about Behavioral Verilog, see XST Behavioral Verilog Language Support.

Variable Part Selects

Verilog 2001 adds the capability of using 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.

Variable Part Select Symbols

Symbol	Meaning
+ (plus)	The part select increases from the starting point
- (minus)	The part select decreases from the starting point

Variable Part Select Verilog Coding 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:

- Component

  The building or basic block
- Port

  A component I/O connector
- 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.

Verilog 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
- NOT

Building a Basic XOR Function Structural Verilog Coding Example

Following 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:

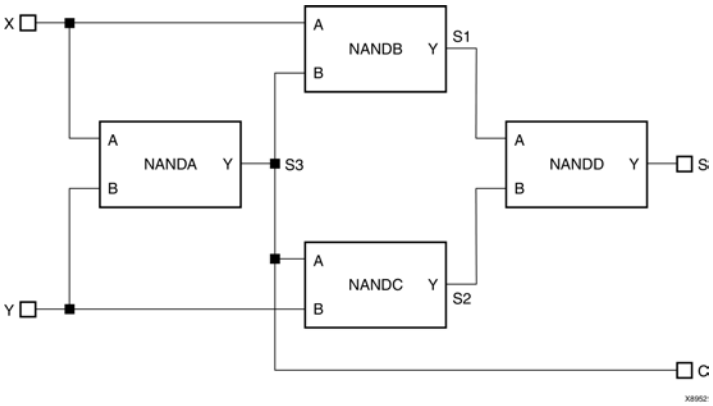
- a_inv
- b_inv
- out

Structural Description of a Half Adder Structural Verilog Coding Example

The following coding example shows the structural description of a half adder composed of four, 2 input nand modules:

```
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
```

Synthesized Top Level Netlist



The structural features of Verilog also allow you to design circuits by instantiating pre-defined primitives such as gates, registers and Xilinx® specific primitives such as CLKDLL and BUFGs. These primitives are other than those included in Verilog. These pre-defined primitives are supplied with the XST Verilog libraries (`unisim_comp.v`).

Structural Instantiation of REGISTER and BUFG Structural Verilog Coding Example

```
module foo (sysclk, in, reset, out);
input sysclk, in, reset;
output out;
reg out;
wire sysclk_out;
FDC register (out, sysclk_out, reset, in); //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.

```
(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module FDC (Q, C, CLR, D);
parameter INIT = 1'b0;
output Q;
input C;
input CLR;
input D;
endmodule

(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module BUFG ( O, I);
 output O;
 input I;
endmodule
```

Verilog Parameters

Verilog modules allow you to define constants known as parameters. Parameters 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.

Verilog Parameters Coding Example

The following Verilog coding example shows the use of parameters. Null string parameters are not supported:

```
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.

The Generics (-generics) command line option allows you to redefine parameters (Verilog) values defined in the top-level design block. This allows you to easily modify the design configuration without any Hardware Description Language (HDL) source modifications, such as for IP core generation and testing flows.

Verilog Parameter and 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, conflicts will occasionally arise.

To resolve these conflicts, XST uses the following rules of precedence:

1. Specifications on an instance (lower level) takes precedence over specifications 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. XST issues a warning message.
3. An attribute specified in the XST Constraint File (XCF) takes precedence over attributes or parameters specified in the Verilog code.

When an attribute specified on an instance overrides a parameter specified on a module in XST, the simulation tool may use the parameter anyway. If that happens, the *simulation* results may not match the *synthesis* results.

Verilog Parameter and Attribute Conflicts Precedence

Use the following table as a guide in determining precedence.

Verilog Parameter and Attribute Conflicts Precedence

	Parameter on an Instance	**Parameter on a Module**
Attribute on an Instance	Apply Parameter (XST issues warning)	Apply Attribute (possible simulation mismatch)
Attribute on a Module	Apply Parameter	Apply Parameter (XST issues warning)
Attribute in XCF	Apply Attribute (XST issues warning)	Apply Attribute

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.

Verilog Case Sensitivity

Since Verilog is case sensitive, module and instance names can be made unique by changing capitalization. However, for compatibility with file names, mixed language support, and other tools, Xilinx® recommends that you do not rely on capitalization only to make instance names unique.

XST does not allow module names to differ by capitalization only. It renames instances and signal names to ensure that lack of case sensitivity support in other tools in your flow does not adversely impact your design.

XST Support for Verilog Case Sensitivity

XST supports Verilog 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 in case only. XST renames them using a postfix as with equivalent names.

For instance:

```
module upperlower4 (input1, INPUT1, output1, output2);
  input input1;
  input INPUT1;
```

For this example, **`INPUT1`** is renamed to **`INPUT1_rnm0`**.

Verilog Restrictions Within XST

Code using equivalent names (named blocks, tasks, and functions) such as the following is rejected:

```
...
always @(clk)
begin: fir_main5
  reg [4:0] fir_main5_w1;
  reg [4:0] fir_main5_W1;
```

XST issues the following error message:

```
ERROR:Xst:863 - "design.v", line 6:  Name conflict (<fir_main5/fir_main5_w1> and
<fir_main5/fir_main5_W1>)
```

Code using case equivalent module names such as the following is rejected:

```
module UPPERLOWER10 (...);
...
module upperlower10 (...);
...
```

XST issues the following error message:

```
ERROR:Xst:909 - Module name conflict (UPPERLOWER10 and upperlower10)
```

Verilog Blocking and Nonblocking Assignments

XST rejects Verilog designs if a given signal is assigned through both blocking and nonblocking assignments as shown in the following coding example:

```
always @(in1)
begin
  if (in2)
    out1 = in1;
  else
    out1 <= in2;
end
```

If a variable is assigned in both a blocking and nonblocking assignment, XST issues the following error message:

`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 coding 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 one blocking or non-blocking error, only the first 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).

Verilog Integer Handling

XST handles integers differently from other synthesis tools in several instances. They must be coded in a particular way.

Integer Handling in Verilog Case Statements

Unsized integers in case item expressions may cause unpredictable results. In the following coding example, the case item expression **4** is an unsized integer that causes unpredictable results. To avoid problems, size the **4** to **3** bits as follows:

```
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
```

Integer Handling in Verilog Concatenations

Unsized integers in concatenations may cause unpredictable results. If you 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 follows:

```
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```

Verilog Attributes and Meta Comments

XST supports both Verilog-2001 style attributes and meta comments in Verilog. Xilinx® recommends Verilog-2001 attributes since they are more generally accepted. Meta comments are comments that are understood by the Verilog parser.

Verilog-2001 Attributes

XST supports Verilog-2001 attribute statements. Attributes are comments that 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. Other attribute declarations may be supported by the compiler, but are ignored by XST.

Use attributes to:

- Set constraints on individual objects, for example:
  - module
  - instance
  - net
- Set the following synthesis constraints
  - Full Case (FULL_CASE)
  - Parallel Case (PARALLEL_CASE)

Verilog Meta Comments

Use Verilog meta comments to:

- Set constraints on individual objects such as:
  - module
  - instance
  - net
- Set directives on synthesis:
  - **`parallel_case`** and **`full_case`** directives
  - **`translate_on`** and **`translate_off`** directives
  - all tool specific directives (for example, **`syn_sharing`**)

For more information, see XST Design Constraints.

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:

- Both C-style and Verilog style meta comments
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON) constraints

```
// synthesis translate_on
// synthesis translate_off
```

- Parallel Case (PARALLEL_CASE) constraints

```
// synthesis parallel_case full_case // synthesis parallel_case
// synthesis full_case
```

- Constraints on individual objects

The general syntax is:

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

Verilog Meta Comments Coding Example

```
// 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"
```

Verilog Constructs Supported in XST

This section discusses Verilog Constructs Supported in XST, including:

- Constants
- Data Types
- Continuous Assignments
- Procedural Assignments
- Design Hierarchies
- Compiler Directives

Note XST does not allow underscores as the first character of signal names (for example, **`_DATA_1`**)

Verilog Constants Supported in XST

Constant	Supported/Unsupported
Integer Constants	Supported
Real Constants	Supported
Strings Constants	Unsupported

Verilog Data Types Supported in XST

All Verilog data types are supported in XST except as noted below.

- Net types

  `tri0`, **`tri1`**, and **`trireg`** are unsupported.
- Drive strengths

  All drive strengths are ignored.
- Registers

  Real and realtime registers are unsupported
- Named events

  All named events are unsupported.

Verilog Continuous Assignments Supported in XST

Continuous Assignment	Supported/Unsupported
Drive Strength	Ignored
Delay	Ignored

Verilog Procedural Assignments Supported in XST

Verilog Procedural Assignments are supported in XST except as noted below:

- **`assign`**
  Supported with limitations. See Behavioral Verilog Assign and Deassign Statements.
- **`deassign`**
  Supported with limitations. See Behavioral Verilog Assign and Deassign Statements.
- **`force`**
  Unsupported
- **`release`**
  Unsupported
- **`forever`** statements
  Unsupported
- **`repeat`** statements
  Supported, but repeat value must be constant
- **`for`** statements
  Supported, but bounds must be static
- **`delay (#)`**
  Ignored
- **`event (@)`**
  Unsupported
- **`wait`**
  Unsupported
- **`Named Events`**
  Unsupported
- **`Parallel Blocks`**
  Unsupported
- **`Specify Blocks`**
  Ignored
  `Disable`
  Supported except in For and Repeat Loop statements.

Verilog Design Hierarchies Supported in XST

Design Hierarchy	Supported/Unsupported
module definition	Supported
macromodule definition	Unsupported
hierarchical names	Unsupported
defparam	Supported
array of instances	Supported

Verilog Compiler Directives Supported in XST

Compiler Directive	Supported/Unsupported
`celldefine `endcelldefine	Ignored
`default_nettype	Supported
`define	Supported
`ifdef `else `endif	Supported
`undef, `ifndef, `elsif,	Supported
`include	Supported
`resetall	Ignored
`timescale	Ignored
`unconnected_drive `nounconnected_drive	Ignored
`uselib	Unsupported
`file, `line	Supported

Verilog System Tasks and Functions Supported in XST

System Task or Function	Supported/Unsupported	Comment
$display	Supported	Escape sequences are limited to %d, %b, %h, %o, %c and %s
$fclose	Supported	
$fdisplay	Supported	
$fgets	Supported	
$finish	Supported	$finish is supported for statically never active conditional branches only
$fopen	Supported	
$fscanf	Supported	Escape sequences are limited to %b and %d
$fwrite	Supported	
$monitor	Ignored	
$random	Ignored	
$readmemb	Supported	
$readmemh	Supported	
$signed	Supported	
$stop	Ignored	
$strobe	Ignored	
$time	Ignored	
$unsigned	Supported	
$write	Supported	Escape sequences are limited to %d, %b, %h, %o, %c and %s
all others	Ignored	

The XST Verilog compiler ignores unsupported system tasks.

The **$signed** and **$unsigned** system tasks can be called on any expression using the following syntax:

`$signed(expr)` or `$unsigned(expr)`

The return value from these calls is the same size as the input value. Its sign is forced regardless of any previous sign.

The **$readmemb** and **$readmemh** system tasks can be used to initialize block memories. For more information, see Initializing RAM From an External File.

Use **$readmemb** for binary and **$readmemh** for hexadecimal representation. To avoid the possible difference between XST and simulator behavior, Xilinx® recommends that you use index parameters in these system tasks. See the following coding example:

```
$readmemb("rams_20c.data",ram, 0, 7);
```

The remainder of the system tasks can be used to display information to your computer screen and log file during processing, or to open and use a file during synthesis. You must call these tasks from within initial blocks. XST supports a subset of escape sequences, specifically:

- **%h**
- **%d**
- **%o**
- **%b**
- **%c**
- **%s**

Verilog $display Syntax Example

The following example shows the syntax for **$display** that reports the value of a binary constant in decimal format:

```
parameter c = 8'b00101010;
initial
  begin
    $display ("The value of c is %d", c);
  end
```

The following information is written to the log file during the HDL Analysis phase:

```
Analyzing top module <example>.
c = 8'b00101010
"foo.v" line 9: $display : The value of c is 42
```

Verilog Primitives

XST supports certain gate-level primitives. The supported syntax is:

gate_type instance_name (*output*, *inputs*,...);

Following is a gate-level primitive instantiations coding example:

```
and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
```

XST supports the following Verilog Gate Level primitives except for the following:

- Pulldown and pullup

  Unsupported
- Drive strength and delay

  Ignored
- Arrays of primitives

  Unsupported

XST does not support Verilog Switch-Level primitives, such as:

- cmos, nmos, pmos, rcmos, rnmos, rpmos
- rtran, rtranif0, rtranif1, tran, tranif0, tranif1

XST does not support Verilog user-defined primitives.

Verilog Reserved Keywords

Keywords marked with an asterisk (*) are reserved by Verilog, but are not supported by XST.

always	and	assign	automatic
begin	buf	bufif0	bufif1
case	casex	casez	cell*
cmos	config*	deassign	default
defparam	design*	disable	edge
else	end	endcase	endconfig*
endfunction	endgenerate	endmodule	endprimitive
endspecify	endtable	endtask	event
for	force	forever	fork
function	generate	genvar	highz0
highz1	if	ifnone	incdir*
include*	initial	inout	input
instance*	integer	join	large
liblist*	library*	localparam*	macromodule
medium	module	nand	negedge
nmos	nor	noshow-cancelled*	not
notif0	notif1	or	output
parameter	pmos	posedge	primitive
pull0	pull1	pullup	pulldown
pulsestyle- _ondetect*	pulsestyle- _onevent*	rcmos	real
realtime	reg	release	repeat
rnmos	rpmos	rtran	rtranif0
rtranif1	scalared	show-cancelled*	signed
small	specify	specparam	strong0
strong1	supply0	supply1	table
task	time	tran	tranif0
tranif1	tri	tri0	tri1
triand	trior	trireg	use*
vectored	wait	wand	weak0
weak1	while	wire	wor
xnor	xor		

Verilog-2001 Support in XST

XST supports the following Verilog-2001 features. For more information, 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
- Recursive Tasks and Functions
- Constant Functions

Chapter 9

XST Behavioral Verilog Language Support

This chapter describes XST support for Behavioral Verilog, and includes:

Behavioral Verilog Variable Declarations

Variables in Verilog may be declared as integers or real. These declarations are intended for use in test code only. 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.

Behavioral Verilog Variable Declarations Coding Example

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

where

- **arb_request[31]** is the MSB
- **arb_request[0]** is the LSB

Behavioral Verilog Initial Values

In Verilog-2001, you can initialize registers when you declare them.

The value:

- Is a constant
- Cannot depend on earlier initial values
- Cannot be a function or task call
- Can be a parameter value propagated to the register
- Specifies all bits of a vector

Behavioral Verilog Initial Values Coding Example

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.

```
reg arb_onebit = 1'b0;
reg [3:0] arb_priority = 4'b1011;
```

You can also assign a set/reset (initial) value to a register in your behavioral Verilog code. Assign value to a register when the register reset line goes to the appropriate value as shown in the following coding 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.

Behavioral Verilog Local Reset

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 Behavioral Verilog Local Reset Coding 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**.

Behavioral Verilog Local Reset Coding 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 output at initial power up, but since this is dependent upon a local reset, the value changes whenever the local set/reset is activated.

Behavioral Verilog Arrays Coding Examples

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Verilog allows arrays of **reg** and **wires** to be defined as shown in the following coding examples.

Behavioral Verilog Arrays Coding Example

The following coding example describes an array of 32 elements each, 4 bits wide which can be assigned in *behavioral* Verilog code:

```
reg [3:0] mem_array [31:0];
```

Structural Verilog Arrays Coding Example

The following coding example describes an array of 64 elements each 8 bits wide which can be assigned only in *structural* Verilog code:

```
wire [7:0] mem_array [63:0];
```

Behavioral Verilog Multi-Dimensional Arrays

XST supports multi-dimensional array types of up to two 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 to regular tasks or functions.

Behavioral Verilog Multi-Dimensional Arrays Coding Example One

The following Verilog coding example describes an array of 256 x 16 wire elements each 8 bits wide, which can be assigned only in *structural* Verilog code:

```
wire [7:0] array2 [0:255][0:15];
```

Behavioral Verilog Multi-Dimensional Arrays Coding Example Two

The following Verilog coding example describes an array of 256 x 8 register elements, each 64 bits wide, which can be assigned in *behavioral* Verilog code:

```
reg [63:0] regarray2 [255:0][7:0];
```

Behavioral Verilog Data Types

The Verilog representation of the bit data type contains the following values:

- **0**

  logic zero
- **1**

  logic one
- **x**

  unknown logic value
- **z**

  high impedance

XST supports 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)
- Multiple bit (vectors)

Behavioral Verilog Data Types Coding Example

The following Behavioral Verilog coding example shows sample Verilog data types found in the declaration section of a Verilog module:

```
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
```

Behavioral Verilog Legal Statements

The following statements 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

All variables are declared as **integer** or **reg**. A variable cannot be declared as a **wire**.

Behavioral Verilog Expressions

An expression involves constants and variables with arithmetic, logical, relational, and conditional operators as shown in Operators Supported in Behavioral Verilog. 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.

Operators Supported in Behavioral Verilog

Arithmetic	Logical	Relational	Conditional
+	&	<	?
-	&&	==	
*	|	===	
**	||	<=	
/	^	>=	
%	~	>=	
	~^	!=	
	^~	!==	
	<<	>	
	>>		
	<<<		
	>>>		

Expressions Supported in Behavioral Verilog

Expression	Symbol	Supported/Unsupported
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

Expression	Symbol	Supported/Unsupported
Power	**	Supported • Both operands are 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
Reduction XNOR	~^, ^~	Supported
Left Shift	<<	Supported
Right Shift Signed	>>>	Supported
Left Shift Signed	<<<	Supported
Right Shift	>>	Supported
Conditional	?:	Supported
Event OR	or, ','	Supported

Results of Evaluating Expressions in Behavioral Verilog

The following table lists the results of evaluating expressions using the more frequently used operators supported by XST.

The **(===)** and **(!==)** operators 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.

Results of Evaluating Expressions in Behavioral Verilog

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

Behavioral Verilog Blocks

Block statements are used to group statements together. XST supports sequential blocks only. Within these blocks, the statements are executed in the order listed. Block statements are designated by **begin** and **end** keywords.

XST does not support parallel blocks.

Behavioral Verilog 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 has 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. The two kinds of procedural blocks are:

- **Initial** block
- **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
```

Each statement is a procedural assignment line terminated by a semicolon.

Behavioral Verilog Module Declarations

The I/O ports of the circuit are declared in the module declaration. Each port has:

- A name
- A mode
  - **in**
  - **out**
  - **inout**

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 in-out 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 Behavioral Verilog Module Declaration Coding 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**, 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.

Behavioral Verilog Module Declaration Coding Example

```
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
```

Behavioral Verilog 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. Delays and strengths given to a continuous assignment are ignored by XST.

Continuous assignments are allowed on wire and tri data types only.

Coding examples are accurate as of the date of publication. Download updates from ftp://ftp.xilinx.com/pub/documentation/misc/examples_v9.zip.

Behavioral Verilog Explicit Continuous Assignment Coding Example

```
wire par_eq_1;
....
assign par_eq_1 = select ? b : a;
```

Behavioral Verilog Implicit Continuous Assignment Coding Example

```
wire temp_hold = a | b;
```

Continuous assignments are allowed on **wire** and **tri** data types only.

Behavioral Verilog Procedural Assignments

Procedural assignments are:

- Used to assign values to variables declared as regs
- Introduced by **always** blocks, tasks, and functions
- Usually used to model registers and Finite State Machine (FSM) components

XST supports:

- Combinatorial functions
- Combinatorial and sequential tasks
- Combinatorial and sequential always blocks

Behavioral Verilog Combinatorial Always Blocks

Combinatorial logic can be modeled efficiently using two forms of Verilog time control statements:

- # (pound)
- @ (asterisk)

Since the # (pound) time control statement is ignored for synthesis, this discussion describes modeling combinatorial logic with the @ (asterisk) time control 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** or **Case**, for example), and any signal appearing on the right hand side of an assignment. By substituting an @ (asterisk) 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.

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.

Various 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

Behavioral Verilog 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.

Behavioral Verilog If...Else Statement Coding Example

The following coding example shows how a MUX can be described using an **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
```

Behavioral Verilog Case Statements

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.

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.

Behavioral Verilog Case Statement Coding Example

The following coding example shows how a MUX can be described using a **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, Xilinx® recommends that you use a parallel-case Verilog attribute to ensure parallel evaluation of the **sel** inputs as shown in the following:

```
(* parallel_case *) case(sel)
```

Behavioral Verilog 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 supported for constant values only.

Disable statements are not supported.

Behavioral Verilog For Loop Coding Example

```
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
```

Behavioral Verilog 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** option.
- While loops can have **disable** statements. The **disable** statement is used inside a labeled block, since the syntax is **disable <blockname>**.

Behavioral Verilog While Loop Coding Example

```
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
```

Behavioral Verilog 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:

- A clock signal event (mandatory)
- A reset signal event (possibly)
- 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 are assigned to the following constant values:

- **`0`**
- **`1`**
- **`X`**
- **`Z`**
- Any vector composed of these values

These same signals are also 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.

8 Bit Register Using an Always Block Behavioral Verilog Coding Example

```
module seq1 (DI, CLK, DO);
  input [7:0] DI;
  input CLK;
  output [7:0] DO;
  reg [7:0] DO;

  always @(posedge CLK)
    DO <= DI ;
```

8 Bit Register with Asynchronous Reset (High-True) Using an Always Block Behavioral Verilog Coding Example

```
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
```

8 Bit Counter with Asynchronous Reset (Low-True) Using an Always Block Behavioral Verilog Coding Example

```
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
```

Behavioral Verilog Assign and Deassign Statements

Assign and deassign statements are supported within simple templates.

Behavioral Verilog Assign and Deassign Statements General Template Coding Example

```
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:

- For a given signal, there is only one assign/deassign statement.
- The assign/deassign statement is performed in the same always block through an if/else statement.
- You cannot assign a bit/part select of a signal through an assign/deassign statement.

Behavioral Verilog Assign/Deassign Statement Coding Example

For a given signal, there is only one assign/deassign statement. For example, XST rejects the following design:

```
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
```

Behavioral Verilog Assign/Deassign Statement Performed in Same Always Block

The assign/deassign statement is performed in the same always block through an **`if...else`** statement. For example, XST rejects the following design:

```
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
```

Cannot Assign Bit/Part Select of Signal Through Assign/Deassign Statement

You cannot assign a bit/part select of a signal through an assign/deassign statement. For example, XST rejects the following design:

```
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
```

Behavioral Verilog 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
  - **x** for unknown
- If the right-hand expression is unsigned, the left-hand expression is padded with **0**s (zeroes).
- 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).

The above rules follow the Verilog-2001 standard. They are not backward compatible with Verilog-1995.

Behavioral Verilog 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.

The Behavioral Verilog Function Declared Within a Module Coding Example shows a function declared within a module. The ADD function declared is a single-bit adder. This function is called four times with the proper parameters in the architecture to create a 4-bit adder.

The Behavioral Verilog Function Declared Within a Module Coding Example described with a task, is shown in the following coding example.

Behavioral Verilog Function Declared Within a Module Coding Example

```
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
```

Behavioral Verilog Task Declaration and Task Enable Coding Example

The following coding example shows the Behavioral Verilog Function Declared Within a Module Coding Example described with a task:

```
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
```

Behavioral Verilog Recursive Tasks and Functions

Verilog-2001 adds support for recursive tasks and functions. You can use recursion only with the **`automatic`** keyword. To prevent endless recursive calls, the number of recursions is limited by default to 64. Use the **`-recursion_iteration_limit`** option to control the number of allowed recursive calls.

Behavioral Verilog Syntax Using Recursion Coding Example

```
function  automatic [31:0] fac;
input [15:0] n;
if (n == 1)
  fac = 1;
else
  fac = n * fac(n-1);  //recursive function call
endfunction
```

Behavioral Verilog Constant Functions

Verilog-2001 adds support for constant functions. XST supports function calls to calculate constant values.

Evaluation of a Constant Function Behavioral Verilog Coding Example

```
module rams_cf (clk, we, a, di, do);
parameter DEPTH=1024;
input clk;
input we;
input [4:0] a;
input [3:0] di;
output [3:0] do;

reg [3:0] ram [size(DEPTH):0];

always @(posedge clk) begin
if (we)
ram[a] <= di;
end
assign do = ram[a];

function integer size;
input depth;
integer i;
begin
    size=1;
  for (i=0; 2**i<depth; i=i+1)
    size=i+1;
end
endfunction

endmodule
```

Behavioral Verilog Blocking Versus Non-Blocking Procedural Assignments

The pound (#) and asterisk (@) time control statements delay execution of the statement following them until the specified event is evaluated as true. Blocking and non-blocking procedural assignments have time control built into their respective assignment statement. The pound (#) delay is ignored for synthesis.

Behavioral Verilog Blocking Procedural Assignment Syntax Example

The syntax for a blocking procedural assignment is shown in the following coding 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 occurs only after the specified delay.

Behavioral Verilog Non-Blocking Procedural Assignment Syntax Example

The syntax for a non-blocking procedural assignment is shown in the following coding example:

```
variable <= @(posedge_or_negedge_bit ) expression;
```

Behavioral Verilog Non-Blocking Procedural Assignment Example

The following shows an example of how to use a non-blocking procedural assignment:

```
if (in1) out <= 1'b1;
else out <= in2;
```

Behavioral Verilog 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, the following all represent the same value:

- 4'b1010
- 4'o12
- 4'd10
- 4'ha

Behavioral Verilog Macros

Verilog provides a way to define macros as shown in the following coding 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 coding example:

```
`define myzero 0
assign mysig = `myzero;
```

The Verilog **`ifdef** and **`endif** constructs determine whether or not a macro is defined. 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 **`endif** must complete the conditional statement.

The **`ifdef** and **`endif** constructs are shown in the following coding example:

```
`ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
`else
module if_MYVAR_is_not_declared;
...
endmodule
`endif
```

The Verilog Macros (-define) command line option allows you to define (or redefine) Verilog macros. This allows you to easily modify the design configuration without any Hardware Description Language (HDL) source modifications, such as for IP core generation and testing flows.

Behavioral Verilog Include Files

Verilog allows you to separate source code into more than one file. To reference the code contained in another file, use the following syntax in the current file:

```
 `include "path/file-to-be-included"
```

The path can be relative or absolute.

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 enable the file in your **`include** statement to be recognized, identify the directory where it resides, either to ISE® Design Suite or to XST.

- Since ISE Design Suite searches the ISE Design Suite project directory by default, adding the file to your project directory identifies the file to ISE Design Suite
- To direct ISE Design Suite to a different directory, include a path (relative or absolute) in the **`include** statement in your source code.
- To point XST directly to your include file directory, use Verilog Include Directories (-vlgincdir)
- If the include file is required for ISE Design Suite 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 (`modules.v`) has been added to an ISE Design Suite project directory *and* is specified with **`include**, conflicts may occur. In that case, XST issues an error message:

```
ERROR:Xst:1068 - fifo.v, line 2. Duplicate declarations of module'RAMB4_S8_S8'
```

Behavioral Verilog Comments

Behavioral Verilog supports two forms of comments as shown in the following table. Behavioral Verilog comments are similar to the comments used in a language such as C++.

Behavioral Verilog Comment Types

Symbol	Description	Used for	Example
//	Double forward slash	One-line comments	**//** Define a one-line comment as illustrated by this sentence
/*	Slash asterisk	Multi-line comments	**/*** Define a multi-line comment by enclosing it as illustrated by this sentence ***/**

Behavioral Verilog Generate Statements

A **generate** statement allows you to dynamically create Verilog code from conditional statements. This allows you to create repetitive structures or structures that are appropriate only under certain conditions.

Structures likely to be created using 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 **generate** statements:

- Behavioral Verilog Generate For Statements
- Behavioral Verilog Generate If... else Statements
- Behavioral Verilog Generate Case Statements

Behavioral Verilog Generate For Statements

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 has a *genvar* variable.
- The assignments in the **for** loop control refers to the *genvar* variable.
- The contents of the **for** loop are enclosed by **begin** and **end** statements. The **begin** statement is named with a unique qualifier.

8-Bit Adder Using a Generate For Loop Behavioral Verilog Coding Example

```
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
```

Behavioral Verilog Generate If... else Statements

Use a **generate if... else** statement inside a **generate** block to conditionally control which objects are generated.

Generate If... else Statement Behavioral Verilog Coding Example

In the following coding example of a **generate if... else** statement, **generate** controls the type of multiplier that is instantiated.

- The contents of each branch of the **if... else** statement are enclosed by **begin** and **end** statements.
- The **begin** statement is 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
```

Behavioral Verilog Generate Case Statements

Use a **generate case** statement inside a generate block to conditionally control which objects are generated. Use a **generate case** statement when there are several conditions to be tested to determine what the generated code would be.

- Each test statement in a **generate case** is enclosed by **begin** and **end** statements.
- The **begin** statement is named with a unique qualifier.

Behavioral Verilog Generate Case Statement Coding Example

In the following coding example of a **`generate case`** statement, **`generate`** controls the type of adder that 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
```

Chapter 10

XST Mixed Language Support

This chapter describes how to run an XST project that mixes Verilog and VHDL designs, and includes:

- Mixed Language Project Files
- VHDL and Verilog Boundary Rules in Mixed Language Projects
- Port Mapping in Mixed Language Projects
- Generics Support in Mixed Language Projects
- Library Search Order (LSO) Files in Mixed Language Projects

XST supports mixed VHDL and Verilog projects.

- Mixing VHDL and Verilog is restricted to design unit (cell) instantiation only.
  - A VHDL design can instantiate a Verilog module.
  - A Verilog design can instantiate a VHDL entity.
  - No other 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.
- 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 Elaboration.
- Component instantiation based on default binding is used for binding Verilog modules to a VHDL design unit.
- Configuration specification, direct instantiation and component configurations are not supported for a Verilog module instantiation in VHDL.
- VHDL and Verilog project files are unified.
- VHDL and Verilog libraries are logically unified.
- Specification of the work directory for compilation (`xsthdpdir`), previously available only for VHDL, is now 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 now 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 Files

XST uses dedicated mixed language project files to support mixed VHDL and Verilog designs. You can use this mixed language format not only for mixed projects, but also for purely VHDL or Verilog projects.

- If you run XST from ISE® Design Suite, it creates the project file. It is always a mixed language project file.
- If you run XST from the command line, you must create the mixed language project file yourself.

To create a mixed language project file at the command line, use the **-ifmt** command line option set to *mixed* or with its value is omitted. 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:

```
language library file_name.ext
```

The following example shows 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 Hardware Description Language (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. The default logic library is **work**.
- The third column specifies the name of the HDL file.

VHDL and Verilog Boundary Rules in Mixed Language Projects

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:

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.
   a. In ISE® Design Suite, select **Process > Properties > Synthesis Options > Case > Maintain**, or
   b. 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.

Using a VHDL configuration declaration, you could attempt to bind this component to a particular design unit from a particular library. 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. During binding, XST treats a component name as a VHDL design unit name and searches for it in the logical library **work**. If XST finds a VHDL design unit, 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. For more information, see Library Search Order (LSO) Files in Mixed Language Projects. XST selects the first Verilog module matching the name, and binds it.

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:

1. Declare a module name with the same as name as the VHDL entity (optionally followed by an architecture name) that you want to instantiate.
2. 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. During binding, XST searches for a Verilog module name using the name of the instantiated module in the user-specified list of unified logical libraries in the user-specified order. XST ignores any architecture name specified in the module instantiation. For more information, see Library Search Order (LSO) Files in Mixed Language Projects.

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. For more information, see Library Search Order (LSO) Files in Mixed Language Projects. 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:

- Use explicit port association. Specify formal and effective port names in the port map.
- All parameters are passed at instantiation, even if they are unchanged.
- The parameter override is named and not ordered. The parameter override occurs through instantiation, and not through defparams.

Correct Use of Parameter Override Coding Example

```
ff #(.init(2'b01)) u1 (.sel(sel), .din(din), .dout(dout));
```

Correct Use of Parameter Override Coding Example

The following example is not accepted by XST.

```
ff u1 (.sel(sel), .din(din), .dout(dout));
defparam u1.init = 2'b01;
```

Port Mapping in Mixed Language Projects

Port Mapping in mixed language projects includes:

- VHDL in Verilog Port Mapping
- Verilog in VHDL Port Mapping
- VHDL in Mixed Language Port Mapping
- Verilog in Mixed Language Port Mapping

VHDL in Verilog Port Mapping

For VHDL entities instantiated in Verilog designs, XST supports the following port types:

- **`in`**
- **`out`**
- **`inout`**

XST does not support VHDL buffer and linkage ports.

Verilog in VHDL Port Mapping

For Verilog modules instantiated in VHDL designs, XST supports the following port types:

- input
- output
- inout

XST does not support connection to bi-directional pass options 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.

VHDL in Mixed Language Port Mapping

XST supports the following VHDL data types for mixed language designs:

- **bit**
- **bit_vector**
- **std_logic**
- **std_ulogic**
- **std_logic_vector**
- **std_ulogic_vector**

Verilog in Mixed Language Port Mapping

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 (LSO) Files in Mixed Language Projects

The Library Search Order (LSO) file specifies the search order that XST uses to link the libraries used in VHDL and 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:

- The **DEFAULT_SEARCH_ORDER** keyword is used in the LSO file, or
- The LSO file is not specified

Specifying the Library Search Order (LSO) File in ISE Design Suite

In ISE® Design Suite, the default name for the Library Search Order (LSO) file is `project_name.lso`. If a `project_name.lso` file does not already exist, ISE Design Suite automatically creates one.

If ISE Design Suite detects an existing `project_name.lso` file, this file is preserved and used as is. In ISE Design Suite, the name of the project is the name of the top-level block. In creating a default LSO file, ISE Design Suite places the **DEFAULT_SEARCH_ORDER** keyword in the first line of the file.

Specifying the Library Search Order (LSO) File in the Command Line

Library Search Order (LSO) **(-lso)** specifies the Library Search Order (LSO) file when using XST from the command line. If the **-lso** option is omitted, XST automatically uses the default library search order without using an LSO file.

Library Search Order (LSO) Rules

When processing a mixed language project, XST obeys the following search order rules, depending on the contents of the Library Search Order (LSO) file:

- Library Search Order (LSO) Empty
- DEFAULT_SEARCH_ORDER Keyword Only
- DEFAULT_SEARCH_ORDER Keyword and List of Libraries
- List of Libraries Only
- DEFAULT_SEARCH_ORDER Keyword and Non-Existent Library Name”

Library Search Order (LSO) Empty

When the Library Search Order (LSO) file is empty, XST:

- Issues a warning 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.

DEFAULT_SEARCH_ORDER Keyword Only

When the Library Search Order (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

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 ISE® Design Suite, with the following contents:

```
DEFAULT_SEARCH_ORDER
```

XST uses the following search order.

```
vhlib1
rtfllib
vhlib2
```

After processing, the contents of `my_proj.lso` is:

```
vhlib1
rtfllib
vhlib2
```

DEFAULT_SEARCH_ORDER Keyword and List of Libraries

When the Library Search Order (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

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` is:

```
rtfllib
vhlib2
vhlib1
DEFAULT_SEARCH_ORDER
```

List of Libraries Only

When the Library Search Order (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

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` is:

```
rtfllib
vhlib2
vhlib1
```

DEFAULT_SEARCH_ORDER Keyword and Non-Existent Library Name

When the Library Search Order (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.

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` is:

```
rtfllib
vhlib2
vhlib1
```

Chapter 11

XST Log File

This chapter describes the XST log file, and includes:

XST FPGA Log File Contents

The XST FPGA log file contains the following:

- Copyright Statement
- Table of Contents
- Synthesis Options Summary
- HDL Compilation
- Design Hierarchy Analyzer
- HDL Analysis
- HDL Synthesis Report
- Advanced HDL Synthesis Report
- Low Level Synthesis
- Partition Report
- Final Report

XST FPGA Log File Copyright Statement

The XST FPGA log file copyright statement contains:

- ISE® Design Suite release number
- Xilinx® notice of copyright.

XST FPGA Log File Table of Contents

The XST FPGA log file table of contents lists the major sections in the log file. Use the table of contents to navigate to different log file sections. These headings are not linked. Use the Find function in your text editor.

XST FPGA Log File Synthesis Options Summary

The XST FPGA log file Synthesis Options Summary contains information relating to:

- Source Parameters
- Target Parameters
- Source Options
- Target Options
- General Options
- Other Options

XST FPGA Log File Hardware Description Language (HDL) Compilation

For information on Hardware Description Language (HDL) Compilation, see XST FPGA Log File Hardware Description Language (HDL) Analysis.

XST FPGA Log File Design Hierarchy Analyzer

For information on Design Hierarchy Analyzer, see XST FPGA Log File HDL Analysis.

XST FPGA Log File Hardware Description Language (HDL) Analysis

During Hardware Description Language (HDL) Compilation, Design Hierarchy Analyzer, and HDL Analysis, XST:

- Parses and analyzes VHDL and Verilog files
- Recognizes the design hierarchy
- 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.

XST FPGA Log File Hardware Description Language (HDL) Synthesis Report

During Hardware Description Language (HDL) Synthesis, XST tries to recognize as many basic 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 issues the HDL Synthesis Report. For more information about the processing of each macro and the corresponding messages issued during synthesis, see XST Hardware Description Language (HDL) Coding Techniques.

XST FPGA Log File Advanced HDL Synthesis Report

XST performs advanced macro recognition and inference. In this step, XST:

- Recognizes, for example, dynamic shift registers
- Implements pipelined multipliers
- Codes state machines

The Advanced HDL Synthesis Report contains a summary of recognized macros in the overall design, sorted by macro type.

XST FPGA Log File Low Level Synthesis

XST reports the potential removal of, for example, equivalent flip-flops and register replication. For more information, see FPGA Optimization Report Section.

XST FPGA Log File Partition Report

If the design is partitioned, the XST FPGA log file Partition Report contains information detailing the design partitions.

XST FPGA Log File Final Report

The XST FPGA log file Final Report includes:

- Final Results, including
  - RTL Top Level Output File Name (for example, `stopwatch.ngr`)
  - Top Level Output File Name (for example, `stopwatch`)
  - Output Format (for example, NGC)
  - Optimization Goal (for example, Speed)
  - Whether the Keep Hierarchy constraint is used (for example, No)
- Cell usage

  Cell usage reports on, for example, the number and type of BELS, Clock Buffers, and IO Buffers.
- Device Utilization Summary

  The Device Utilization Summary estimates the number of slices, and gives, for example, the number of flip-flops, IOBs, and BRAMS. The Device Utilization Summary closely approximates the report produced by MAP.
- Partition Resource Summary

  The Partition Resource Summary estimates the number of slices, and gives, for example, the number of flip-flops, IOBs, and BRAMS for each partition. The Partition Resource Summary closely resembles the report produced by MAP.
- Timing Report

  At the end of synthesis, XST reports the timing information for the design. The Timing Report shows the information for all four possible domains of a netlist:
  - register to register
  - input to register
  - register to outpad
  - inpad to outpad

    For an example, see the Timing Report section in XST FPGA Log File Example. For more information, see FPGA Optimization Report Section.
- Encrypted Modules

  If a design contains encrypted modules, XST hides the information about these modules.

Reducing the Size of the XST Log File

To reduce the size of the XST log file:

- Use Message Filtering
- Use Quiet Mode
- Use Silent Mode
- Hide Specific Messages

Use Message Filtering

When running XST from ISE® Design Suite, use the Message Filtering wizard to select specific messages to filter out of the log file. For more information, see *Using the Message Filters* in the ISE Design Suite Help.

Use Quiet Mode

Quiet Mode limits the number of messages printed to the computer screen (**stdout**). To invoke Quiet Mode, set the **-intstyle** command line option to either of the following:

- **ise**

  Formats messages for ISE® Design Suite

- **xflow**

  Formats messages for XFLOW

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 CPLD devices
  - Final Results section for FPGA devices
  - A note in the Timing Report stating that the timing numbers are only a synthesis estimate.
  - Timing Detail
  - CPU (XST run time)
  - Memory usage

The following sections are still available for FPGA devices:

- Device Utilization Summary
- Clock Information
- Timing Summary

Use Silent Mode

Silent Mode prevents any messages from being sent to the computer screen (**stdout**), although XST continues to generate the entire log file. To invoke Silent Mode, set the **-intstyle** command line option to *silent*.

Hide Specific Messages

To hide specific messages at the HDL or Low Level Synthesis steps, set the *XIL_XST_HIDEMESSAGES* environment variable to one of the values shown in the following table.

XIL_XST_HIDEMESSAGES Environment Variable Values

Value	Meaning
none (default)	Maximum verbosity. All messages are printed out.
hdl_level	Reduce verbosity during VHDL or 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.

Messages Hidden When Value is Set to hdl_level and hdl_and_low_levels

The following messages are hidden when the value of the *XIL_XST_HIDEMESSAGES* environment variable is set to **hdl_level** and **hdl_and_low_levels**:

- `WARNING:HDLCompilers:38 - design.v line 5 Macro 'my_macro' redefined`

  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.`

Messages Hidden When Value is Set to low_level or hdl_and_low_levels

The following messages are hidden when the value of the *XIL_XST_HIDEMESSAGES* environment variable is set to **low_level** or **hdl_and_low_levels**:

- `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.`

Macros in XST Log Files

XST log files contain detailed information about the set of macros and associated signals inferred by XST from the VHDL or Verilog source on a block by block basis.

Macro inference is done in two steps:

1. HDL Synthesis

   XST recognizes as many simple macro blocks as possible, such as adders, subtractors, and registers.
2. Advanced HDL Synthesis

   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 Macro Recognition report at the Advanced HDL Synthesis step is formatted the same as the corresponding report at the HDL Synthesis step.

XST gives overall statistics of recognized macros twice:

- After the HDL Synthesis step
- After the Advanced HDL Synthesis step

XST no longer lists statistics of preserved macros in the final report.

XST Log File Examples

This section gives the following XST log file examples:

- Recognized Macros XST Log File Example
- Additional Macro Processing XST Log File Example
- XST FPGA Log File Example
- XST CPLD Log File Example

Recognized Macros XST Log File Example

The following log file example shows the set of recognized macros on a block by block basis, as well as the overall macro statistics after this step.

```
=========================================================
*                    HDL Synthesis                      *
=========================================================
...
Synthesizing Unit <decode>.
    Related source file is "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 "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     | LUT                          |
    -----------------------------------------------------
    Summary:
        inferred   1 Finite State Machine(s).
Unit <statmach> synthesized.
...
=========================================================================
HDL Synthesis Report

Macro Statistics
# ROMs                                                 : 3
 16x10-bit ROM                                         : 1
 16x7-bit ROM                                          : 2
# Counters                                             : 2
 4-bit up counter                                      : 2

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

Additional Macro Processing XST Log File Example

The following XST FPGA log file example shows the additional macro processing done during the Advanced HDL Synthesis step and the overall macro statistics after this step.

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

Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <MACHINE/current_state/FSM_0> on signal <current_state[1:3]> with gray encoding.
----------------------
 State    | Encoding
----------------------
 clear    | 000
 zero     | 001
 start    | 011
 counting | 010
 stop     | 110
 stopped  | 111
----------------------

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

Macro Statistics
# FSMs                                                 : 1
# ROMs                                                 : 3
 16x10-bit ROM                                         : 1
 16x7-bit ROM                                          : 2
# Counters                                             : 2
 4-bit up counter                                      : 2
# Registers                                            : 3
 Flip-Flops/Latches                                    : 3

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

XST FPGA Log File Example

The following is an example of an XST log file for FPGA synthesis. Release 10.1 - xst K.31 (nt64)

TABLE OF CONTENTS

```
=========================================================================
* Synthesis Options Summary *
=========================================================================
---- Source Parameters
Input File Name : "stopwatch.prj"
Input Format : mixed
Ignore Synthesis Constraint File : NO
---- Target Parameters
Output File Name : "stopwatch"
Output Format : NGC
Target Device : xc4vlx15-12-sf363
---- Source Options
Top Module Name : stopwatch
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FSM Style : lut
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Mux Style : Auto
Decoder Extraction : YES
Priority Encoder Extraction : YES
Shift Register Extraction : YES
Logical Shifter Extraction : YES
XOR Collapsing : YES
ROM Style : Auto
Mux Extraction : YES
Resource Sharing : YES
Asynchronous To Synchronous : NO
Use DSP Block : auto
Automatic Register Balancing : No
---- Target Options
Add IO Buffers : YES
Global Maximum Fanout : 500
Add Generic Clock Buffer(BUFG) : 32
Number of Regional Clock Buffers : 16
Register Duplication : YES
```

Slice Packing : YES
Optimize Instantiated Primitives : NO
Use Clock Enable : Auto
Use Synchronous Set : Auto
Use Synchronous Reset : Auto
Pack IO Registers into IOBs : auto
Equivalent register Removal : YES
---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Power Reduction : NO
Library Search Order : stopwatch.lso
Keep Hierarchy : NO
Netlist Hierarchy : as_optimized
RTL Output : Yes
Global Optimization : AllClockNets
Read Cores : YES
Write Timing Constraints : NO
Cross Clock Analysis : NO
Hierarchy Separator : /
Bus Delimiter : <>
Case Specifier : maintain
Slice Utilization Ratio : 100
BRAM Utilization Ratio : 100
DSP48 Utilization Ratio : 100
Verilog 2001 : YES
Auto BRAM Packing : NO
Slice Utilization Ratio Delta : 5

==========

==========

* HDL Compilation *

==========

Compiling verilog file "smallcntr.v" in library work
Compiling verilog file "statmach.v" in library work
Module <smallcntr> compiled
Compiling verilog file "hex2led.v" in library work
Module <statmach> compiled
Compiling verilog file "decode.v" in library work

Module <hex2led> compiled
Compiling verilog file "cnt60.v" in library work
Module <decode> compiled
Compiling verilog file "stopwatch.v" in library work
Module <cnt60> compiled
Module <stopwatch> compiled
No errors in compilation
Analysis of file <"stopwatch.prj"> succeeded.
Compiling vhdl file "C:/xst/watchver/tenths.vhd" in Library work.
Entity <tenths> compiled.
Entity <tenths> (Architecture <tenths_a>) compiled.
Compiling vhdl file "C:/xst/watchver/dcm1.vhd" in Library work.
Entity <dcm1> compiled.
Entity <dcm1> (Architecture <BEHAVIORAL>) compiled.
===
* Design Hierarchy Analysis *
===
Analyzing hierarchy for module <stopwatch> in library <work>.
Analyzing hierarchy for entity <dcm1> in library <work> (architecture <BEHAVIORAL>).
Analyzing hierarchy for module <statmach> in library <work> with parameters.
clear = "000001"
counting = "001000"
start = "000100"
stop = "010000"
stopped = "100000"
zero = "000010"
Analyzing hierarchy for module <decode> in library <work>.
Analyzing hierarchy for module <cnt60> in library <work>.
Analyzing hierarchy for module <hex2led> in library <work>.
Analyzing hierarchy for module <smallcntr> in library <work>.
===
* HDL Analysis *
===
Analyzing top module <stopwatch>.
Module <stopwatch> is correct for synthesis.
Analyzing Entity <dcm1> in library <work> (Architecture <BEHAVIORAL>).
Set user-defined property "CAPACITANCE = DONT_CARE" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.
Set user-defined property "IBUF_DELAY_VALUE = 0" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.

Set user-defined property "IOSTANDARD = DEFAULT" for instance <CLKIN_IBUFG_INST> in unit <dcm1>.

Set user-defined property "CLKDV_DIVIDE = 2.0000000000000000" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKFX_DIVIDE = 1" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKFX_MULTIPLY = 4" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKIN_DIVIDE_BY_2 = FALSE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKIN_PERIOD = 20.0000000000000000" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLKOUT_PHASE_SHIFT = NONE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "CLK_FEEDBACK = 1X" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DESKEW_ADJUST = SYSTEM_SYNCHRONOUS" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DFS_FREQUENCY_MODE = LOW" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DLL_FREQUENCY_MODE = LOW" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DSS_MODE = NONE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "DUTY_CYCLE_CORRECTION = TRUE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "FACTORY_JF = C080" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "PHASE_SHIFT = 0" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "SIM_MODE = SAFE" for instance <DCM_INST> in unit <dcm1>.

Set user-defined property "STARTUP_WAIT = TRUE" for instance <DCM_INST> in unit <dcm1>.

Entity <dcm1> analyzed. Unit <dcm1> generated.

Analyzing module <statmach> in library <work>.

clear = 6'b000001

counting = 6'b001000

start = 6'b000100

stop = 6'b010000

stopped = 6'b100000

zero = 6'b000010

Module <statmach> is correct for synthesis.

Analyzing module <decode> in library <work>.

Module <decode> is correct for synthesis.

Analyzing module <cnt60> in library <work>.

Module <cnt60> is correct for synthesis.

Analyzing module <smallcntr> in library <work>.

Module <smallcntr> is correct for synthesis.

Analyzing module <hex2led> in library <work>.

Module <hex2led> is correct for synthesis.

==========

* HDL Synthesis *

==========

Performing bidirectional port resolution...

Synthesizing Unit <statmach>.
Related source file is "statmach.v".
Found finite state machine <FSM_0> for signal <current_state>.

States	6
Transitions	15
Inputs	2
Outputs	2
Clock	CLK (rising_edge)
Reset	RESET (positive)
Reset type	asynchronous
Reset State	000001
Encoding	automatic
Implementation	LUT

Found 1-bit register for signal <CLKEN>.
Found 1-bit register for signal <RST>.
Summary:
inferred 1 Finite State Machine(s).
inferred 2 D-type flip-flop(s).
Unit <statmach> synthesized.
Synthesizing Unit <decode>.
Related source file is "decode.v".
Found 16x10-bit ROM for signal <ONE_HOT>.
Summary:
inferred 1 ROM(s).
Unit <decode> synthesized.
Synthesizing Unit <hex2led>.
Related source file is "hex2led.v".
Found 16x7-bit ROM for signal <LED>.
Summary:
inferred 1 ROM(s).
Unit <hex2led> synthesized.
Synthesizing Unit <smallcntr>.
Related source file is "smallcntr.v".
Found 4-bit up counter for signal <QOUT>.
Summary:
inferred 1 Counter(s).

Unit <smallcntr> synthesized.
Synthesizing Unit <dcm1>.
Related source file is "C:/xst/watchver/dcm1.vhd".
Unit <dcm1> synthesized.
Synthesizing Unit <cnt60>.
Related source file is "cnt60.v".
Unit <cnt60> synthesized.
Synthesizing Unit <stopwatch>.
Related source file is "stopwatch.v".
Unit <stopwatch> synthesized.
===
HDL Synthesis Report
Macro Statistics
ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
Counters : 2
4-bit up counter : 2
Registers : 2
1-bit register : 2
===
===
* Advanced HDL Synthesis *
===
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <MACHINE/current_state/FSM> on signal <current_state[1:3]> with sequential encoding.

State | Encoding

000001 | 000
000010 | 001
000100 | 010
001000 | 011
010000 | 100
100000 | 101

Loading device for application Rf_Device from file '4vlx15.nph' in environment C:\xilinx.
Executing edif2ngd -noa "tenths.edn" "tenths.ngo"

Release 10.1 - edif2ngd K.31 (nt64)
Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.
INFO:NgdBuild - Release 10.1 edif2ngd K.31 (nt64)
INFO:NgdBuild - Copyright (c) 1995-2008 Xilinx, Inc. All rights reserved.
Writing module to "tenths.ngo"...
Reading core <tenths_c_counter_binary_v8_0_xst_1.ngc>.
Loading core <tenths_c_counter_binary_v8_0_xst_1> for timing and area information for instance <BU2>.
Loading core <tenths> for timing and area information for instance <xcounter>.
===
Advanced HDL Synthesis Report
Macro Statistics
ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
Counters : 2
4-bit up counter : 2
Registers : 5
Flip-Flops : 5
===
===
* Low Level Synthesis *
===
Optimizing unit <stopwatch> ...
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block stopwatch, actual ratio is 0.
Number of LUT replicated for flop-pair packing : 0
Final Macro Processing ...
===
Final Register Report
Macro Statistics
Registers : 13
Flip-Flops : 13
===
===
* Partition Report *
===
Partition Implementation Status

```
---------------------------
No Partitions were found in this design.
---------------------------
================================================================================
* 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 : NO
Design Statistics
# IOs : 27
Cell Usage :
# BELS : 70
# GND : 2
# INV : 1
# LUT1 : 3
# LUT2 : 1
# LUT2_L : 1
# LUT3 : 8
# LUT3_D : 1
# LUT3_L : 1
# LUT4 : 37
# LUT4_D : 1
# LUT4_L : 4
# MUXCY : 3
# MUXF5 : 2
# VCC : 1
# XORCY : 4
# FlipFlops/Latches : 17
# FDC : 13
# FDCE : 4
# Clock Buffers : 1
# BUFG : 1
# IO Buffers : 27
# IBUF : 2
```

```
# IBUFG : 1
# OBUF : 24
# DCM_ADVs : 1
# DCM_ADV : 1
=========================================================================
Device utilization summary:
---------------------------
Selected Device : 4vlx15sf363-12
Number of Slices: 32 out of 6144 0%
Number of Slice Flip Flops: 17 out of 12288 0%
Number of 4 input LUTs: 58 out of 12288 0%
Number of IOs: 27
Number of bonded IOBs: 27 out of 240 11%
Number of GCLKs: 1 out of 32 3%
Number of DCM_ADVs: 1 out of 4 25%
---------------------------
Partition Resource Summary:
---------------------------
No Partitions were found in this design.
---------------------------
=========================================================================
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 | Inst_dcm1/DCM_INST:CLK0| 17 |
-----------------------------------+------------------------+-------+
Asynchronous Control Signals Information:
----------------------------------------
-----------------------------------+--------------------------+-------+
Control Signal | Buffer (FF name) | Load |
-----------------------------------+--------------------------+-------+
MACHINE/RST(MACHINE/RST:Q) | NONE(sixty/lsbcount/QOUT_1)| 8 |
```

RESET | IBUF | 5 |
sixty/msbclr(sixty/msbclr_f5:O) | NONE(sixty/msbcount/QOUT_0)| 4 |
---+---+---+
Timing Summary:

Speed Grade: -12
Minimum period: 2.282ns (Maximum Frequency: 438.212MHz)
Minimum input arrival time before clock: 1.655ns
Maximum output required time after clock: 4.617ns
Maximum combinational path delay: No path found
Timing Detail:

All values displayed in nanoseconds (ns)
=
Timing constraint: Default period analysis for Clock 'CLK'
Clock period: 2.282ns (frequency: 438.212MHz)
Total number of paths / destination ports: 134 / 21

Delay: 2.282ns (Levels of Logic = 4)
Source: xcounter/BU2/U0/q_i_1 (FF)
Destination: sixty/msbcount/QOUT_1 (FF)
Source Clock: CLK rising
Destination Clock: CLK rising
Data Path: xcounter/BU2/U0/q_i_1 to sixty/msbcount/QOUT_1
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)
--- ---
FDCE:C->Q 12 0.272 0.672 U0/q_i_1 (q(1))
LUT4:I0->O 11 0.147 0.492 U0/thresh0_i_cmp_eq00001 (thresh0)
end scope: 'BU2'
end scope: 'xcounter'
LUT4_D:I3->O 1 0.147 0.388 sixty/msbce (sixty/msbce)
LUT3:I2->O 1 0.147 0.000 sixty/msbcount/QOUT_1_rstpot (sixty/msbcount/QOUT_1_rstpot)
FDC:D 0.017 sixty/msbcount/QOUT_1

Total 2.282ns (0.730ns logic, 1.552ns route)
(32.0% logic, 68.0% route)
=

Timing constraint: Default OFFSET IN BEFORE for Clock 'CLK'
Total number of paths / destination ports: 4 / 3

Offset: 1.655ns (Levels of Logic = 3)
Source: STRTSTOP (PAD)
Destination: MACHINE/current_state_FSM_FFd3 (FF)
Destination Clock: CLK rising
Data Path: STRTSTOP to MACHINE/current_state_FSM_FFd3
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)

IBUF:I->O 4 0.754 0.446 STRTSTOP_IBUF (STRTSTOP_IBUF)
LUT4:I2->O 1 0.147 0.000 MACHINE/current_state_FSM_FFd3-In_F (N48)
MUXF5:I0->O 1 0.291 0.000 MACHINE/current_state_FSM_FFd3-In (MACHINE/current_state_FSM_FFd3-In)
FDC:D 0.017 MACHINE/current_state_FSM_FFd3

Total 1.655ns (1.209ns logic, 0.446ns route)
(73.0% logic, 27.0% route)
==========
Timing constraint: Default OFFSET OUT AFTER for Clock 'CLK'
Total number of paths / destination ports: 96 / 24

Offset: 4.617ns (Levels of Logic = 2)
Source: sixty/lsbcount/QOUT_1 (FF)
Destination: ONESOUT<6> (PAD)
Source Clock: CLK rising
Data Path: sixty/lsbcount/QOUT_1 to ONESOUT<6>
Gate Net
Cell:in->out fanout Delay Delay Logical Name (Net Name)

FDC:C->Q 13 0.272 0.677 sixty/lsbcount/QOUT_1 (sixty/lsbcount/QOUT_1)
LUT4:I0->O 1 0.147 0.266 lsbled/Mrom_LED21 (lsbled/Mrom_LED2)
OBUF:I->O 3.255 ONESOUT_2_OBUF (ONESOUT<2>)

Total 4.617ns (3.674ns logic, 0.943ns route)
(79.6% logic, 20.4% route)
==========
Total REAL time to Xst completion: 20.00 secs

Total CPU time to Xst completion: 19.53 secs

-->

Total memory usage is 333688 kilobytes

Number of errors : 0 (0 filtered)

Number of warnings : 0 (0 filtered)

Number of infos : 1 (0 filtered)

XST CPLD Log File Example

The following is an example of an XST log file for CPLD synthesis.

Release 10.1 - xst K.31 (nt64)

TABLE OF CONTENTS

1) Synthesis Options Summary

2) HDL Compilation

3) Design Hierarchy Analysis

4) HDL Analysis

5) HDL Synthesis

5.1) HDL Synthesis Report

6) Advanced HDL Synthesis

6.1) Advanced HDL Synthesis Report

7) Low Level Synthesis

8) Partition Report

9) Final Report

===

* Synthesis Options Summary *

===

---- Source Parameters

Input File Name : "stopwatch.prj"

Input Format : mixed

Ignore Synthesis Constraint File : NO

---- Target Parameters

Output File Name : "stopwatch"

Output Format : NGC

Target Device : CoolRunner2 CPLDs

---- Source Options

Top Module Name : stopwatch

Automatic FSM Extraction : YES

FSM Encoding Algorithm : Auto

Safe Implementation : No

Mux Extraction : YES
Resource Sharing : YES
---- Target Options
Add IO Buffers : YES
MACRO Preserve : YES
XOR Preserve : YES
Equivalent register Removal : YES
---- General Options
Optimization Goal : Speed
Optimization Effort : 1
Library Search Order : stopwatch.lso
Keep Hierarchy : YES
Netlist Hierarchy : as_optimized
RTL Output : Yes
Hierarchy Separator : /
Bus Delimiter : <>
Case Specifier : maintain
Verilog 2001 : YES
---- Other Options
Clock Enable : YES
wysiwyg : NO
===

===
* HDL Compilation *
===
Compiling verilog file "smallcntr.v" in library work
Compiling verilog file "tenths.v" in library work
Module <smallcntr> compiled
Compiling verilog file "statmach.v" in library work
Module <tenths> compiled
Compiling verilog file "hex2led.v" in library work
Module <statmach> compiled
Compiling verilog file "decode.v" in library work
Module <hex2led> compiled
Compiling verilog file "cnt60.v" in library work
Module <decode> compiled
Compiling verilog file "stopwatch.v" in library work
Module <cnt60> compiled

Module <stopwatch> compiled
No errors in compilation
Analysis of file <"stopwatch.prj"> succeeded.
===
* Design Hierarchy Analysis *
===
Analyzing hierarchy for module <stopwatch> in library <work>.
Analyzing hierarchy for module <statmach> in library <work> with parameters.
clear = "000001"
counting = "001000"
start = "000100"
stop = "010000"
stopped = "100000"
zero = "000010"
Analyzing hierarchy for module <tenths> in library <work>.
Analyzing hierarchy for module <decode> in library <work>.
Analyzing hierarchy for module <cnt60> in library <work>.
Analyzing hierarchy for module <hex2led> in library <work>.
Analyzing hierarchy for module <smallcntr> in library <work>.
===
* HDL Analysis *
===
Analyzing top module <stopwatch>.
Module <stopwatch> is correct for synthesis.
Analyzing module <statmach> in library <work>.
clear = 6'b000001
counting = 6'b001000
start = 6'b000100
stop = 6'b010000
stopped = 6'b100000
zero = 6'b000010
Module <statmach> is correct for synthesis.
Analyzing module <tenths> in library <work>.
Module <tenths> is correct for synthesis.
Analyzing module <decode> in library <work>.
Module <decode> is correct for synthesis.
Analyzing module <cnt60> in library <work>.
Module <cnt60> is correct for synthesis.

Analyzing module <smallcntr> in library <work>.
Module <smallcntr> is correct for synthesis.
Analyzing module <hex2led> in library <work>.
Module <hex2led> is correct for synthesis.

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

Performing bidirectional port resolution...
Synthesizing Unit <statmach>.
Related source file is "statmach.v".
Found finite state machine <FSM_0> for signal <current_state>.

```
-----------------------------------------------------------------------
| States             | 6                                              |
| Transitions        | 15                                             |
| Inputs             | 2                                              |
| Outputs            | 2                                              |
| Clock              | CLK (rising_edge)                              |
| Reset              | RESET (positive)                               |
| Reset type         | asynchronous                                   |
| Reset State        | 000001                                         |
| Encoding           | automatic                                      |
| Implementation     | automatic                                      |
-----------------------------------------------------------------------
```

Found 1-bit register for signal <CLKEN>.
Found 1-bit register for signal <RST>.
Summary:
inferred 1 Finite State Machine(s).
inferred 2 D-type flip-flop(s).
Unit <statmach> synthesized.
Synthesizing Unit <tenths>.
Related source file is "tenths.v".
Found 4-bit up counter for signal <Q>.
Summary:
inferred 1 Counter(s).
Unit <tenths> synthesized.
Synthesizing Unit <decode>.
Related source file is "decode.v".
Found 16x10-bit ROM for signal <ONE_HOT>.

Summary:
inferred 1 ROM(s).
Unit <decode> synthesized.
Synthesizing Unit <hex2led>.
Related source file is "hex2led.v".
Found 16x7-bit ROM for signal <LED>.
Summary:
inferred 1 ROM(s).
Unit <hex2led> synthesized.
Synthesizing Unit <smallcntr>.
Related source file is "smallcntr.v".
Found 4-bit up counter for signal <QOUT>.
Summary:
inferred 1 Counter(s).
Unit <smallcntr> synthesized.
Synthesizing Unit <cnt60>.
Related source file is "cnt60.v".
Unit <cnt60> synthesized.
Synthesizing Unit <stopwatch>.
Related source file is "stopwatch.v".
Found 1-bit register for signal <strtstopinv>.
Summary:
inferred 1 D-type flip-flop(s).
Unit <stopwatch> synthesized.
===
HDL Synthesis Report
Macro Statistics
ROMs : 3
16x10-bit ROM : 1
16x7-bit ROM : 2
Counters : 3
4-bit up counter : 3
Registers : 3
1-bit register : 3
===
===
* Advanced HDL Synthesis *
===

Analyzing FSM <FSM_0> for best encoding.

Optimizing FSM <MACHINE/current_state/FSM> on signal <current_state[1:3]> with sequential encoding.

State | Encoding

000001 | 000

000010 | 001

000100 | 010

001000 | 011

010000 | 100

100000 | 101

==========

Advanced HDL Synthesis Report

Macro Statistics

ROMs : 3

16x10-bit ROM : 1

16x7-bit ROM : 2

Counters : 3

4-bit up counter : 3

Registers : 6

Flip-Flops : 6

==========

==========

* Low Level Synthesis *

==========

Optimizing unit <stopwatch> ...

Optimizing unit <statmach> ...

Optimizing unit <decode> ...

Optimizing unit <hex2led> ...

Optimizing unit <tenths> ...

Optimizing unit <smallcntr> ...

Optimizing unit <cnt60> ...

==========

* Partition Report *

==========

Partition Implementation Status

No Partitions were found in this design.

===

* 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 : CoolRunner2 CPLDs

Macro Preserve : YES

XOR Preserve : YES

Clock Enable : YES

wysiwyg : NO

Design Statistics

IOs : 28

Cell Usage :

BELS : 413

AND2 : 120

AND3 : 10

AND4 : 6

INV : 174

OR2 : 93

OR3 : 1

XOR2 : 9

FlipFlops/Latches : 18

FD : 1

FDC : 5

FDCE : 12

IO Buffers : 28

IBUF : 4

OBUF : 24

===

Total REAL time to Xst completion: 7.00 secs

Total CPU time to Xst completion: 6.83 secs

-->

Total memory usage is 196636 kilobytes

Number of errors : 0 (0 filtered)

Number of warnings : 0 (0 filtered)

Number of infos : 0 (0 filtered)

Chapter 12

XST Naming Conventions

This chapter describes XST Naming Conventions, and includes:

XST Net Naming Conventions

The following XST net naming conventions are listed in order of naming priority:

1. Maintain external pin names.
2. Keep hierarchy in signal names, using forward slashes (/) or 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`** (such as **`_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`.
8. 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_OBU`**F.
9. Base names for internal (combinatorial) nets on user HDL signal names where possible.

XST Instance Naming Conventions

Xilinx® highly recommends that you use the following instance naming conventions. To use instance naming conventions from previous releases of ISE® Design Suite, insert the following command line option in the XST command line:

`-old_instance_names` *1*

The following rules are listed in order of naming priority:

1. Keep hierarchy in instance names, using forward slashes (/) or underscores (_) as hierarchy designators.

   When instance names are generated from VHDL or Verilog generate statements, labels from generate statements are used in composition of instance names.

   For example, for the following VHDL generate statement:

   ```
   i1_loop: for i in 1 to 10 generate
   inst_lut:LUT2 generic map (INIT => "00")
   ```

   XST generates the following instance names for LUT2:

   ```
   i1_loop[1].inst_lut
   i1_loop[2].inst_lut
   i1_loop[9].inst_lut
   ...
   i1_loop[10].inst_lut
   ```

2. Name register instances, including state bits, for the output signal.
3. Name clock buffer instances **_clockbuffertype** (such as **_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**.
8. Name input instance names to OBUFs using the form **instance_name_OBUF**.

XST Name Generation Control

Use the following properties to control aspects of the manner in which names are written. Apply these properties in ISE® Design Suite with **Synthesis Properties**, or the appropriate command line options. For more information, see XST Design Constraints.

- Hierarchy Separator (-hierarchy_separator)
- Bus Delimiter (-bus_delimiter)
- Case (-case)
- Duplication Suffix (-duplication_suffix)

Chapter 13

XST Command Line Mode

This chapter describes how to run XST using the command line, including the XST run and set commands and their options. This chapter includes:

Running XST in Command Line Mode

To run XST in command line mode:

- On a workstation, run **`xst`**
- On a PC, run **`xst.exe`**

XST File Types in Command Line Mode

XST generates the following files types in command line mode:

- Design output file, NGC (**`.ngc`**)

  This file is generated in the current output directory (see the **`-ofn`** option).
- Register Transfer Level (RTL) netlist for RTL and Technology Viewers (**`.ngr`**)
- Synthesis log file (**`.srp`**)
- Temporary files

Temporary Files in Command Line Mode

Temporary files are generated in the XST temp directory in command line mode. By default, the XST temp directory is:

- Workstations

  `/tmp`
- Windows

  The directory specified by either the *TEMP* or *TMP* environment variable

Use **`set -tmpdir <`***directory***`>`** to change the XST temp directory.

VHDL or Verilog compilation files are generated in the temp directory. The default temp directory is the `xst` subdirectory of the current directory.

Xilinx recommends that you clean the XST temp directory regularly. The temp 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 temp directory may severely impact CPU performance. XST does not automatically clean the temp directory.

Names With Spaces in Command Line Mode

XST supports file and directory names with spaces in command line mode. Enclose file or directory names containing spaces in double quotes: `C:\my project`

The command line syntax for options supporting multiple directories (**`-sd`**, **`-vlgincdir`**) has changed. Enclose multiple directories in braces: **`-vlgincdir {"C:\my project" C:\temp}`**

In previous releases, multiple directories were included in double quotes. XST still supports this convention, provided directory names do not contain spaces. Xilinx® recommends that you change existing scripts to the new syntax.

Launching XST in Command Line Mode

You can launch XST in command line mode using:

- The XST shell
- A script file

Launching XST in Command Line Mode Using the 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. XST does not accept a mode where you can first enter **`set`** *option_1*, then **`set`** *option_2*, and then enter **`run`**.

Since all options are set at the same time, Xilinx® recommends that you use a script file.

Launching XST in Command Line Mode Using a Script File

Store your commands in a separate script file and run them all 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}
```

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 following to limit the number of messages printed to the screen:

- The **`-intstyle`** silent option
- The XIL_XST_HIDEMESSAGES environment variable
- The message filter feature in ISE® Design Suite

For more information, see Reducing the Size of the XST Log File.

For example, assume that the following text 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 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.
```

If you created your project using ISE Design Suite, and have run XST at least once from ISE Design Suite, you can switch to XST command line mode and use the script and project files that were created by ISE Design Suite. To run XST from the command line, run the following command from project directory:

```
xst -ifn <top_level_block>.xst -ofn <top_level_block>.syr
```

Setting Up an XST Script

An XST script is a set of commands, each command having various options. You can set up an XST script using any of the following commands

- Run
- Set
- Elaborate

Setting Up an XST Script Using the Run Command

The **`run`** command is the main synthesis command. It allows you to run synthesis in its entirety, beginning with the parsing of the Hardware Description Language (HDL) files, and ending with the generation of the final netlist. The run keyword can be used only once per script file.

The **`run`** command begins with a keyword **`run`**, which is followed by a set of options and its values:**`run`** *option_1 value option_2 value ...*

To improve the readability of your script file, place each option-value pair on a separate line:

Use the pound (#) character to comment out options, or place additional comments in the script file:

```
run
option_1 value
# option_2 value
option_3 value
```

Observe the following rules:

- The first line contains only the run command without any options.
- There are no blank lines in the middle of the command.
- Each option name begins with dash (-). For example: *-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). Options such as **-vlgincdir** accept several directories as values. Separate the directories by spaces, and enclose them in braces ({}):

    ```
    -vlgincdir {c:\vlg1 c:\vlg2}
    ```

    For more information, see Names With Spaces in Command Line Mode.
  - An integer

XST Specific Non-Timing Options and XST Specific Non-Timing Options: XST Command Line Only summarize XST specific non-timing related options, including **run** command options and their values.

XST provides online Help from the UNIX command line. The following information is available by typing *help* at the command line. The XST help function provides a list of supported families, available commands, options and their values for each supported device family.

To see 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 see a list of supported families, type **help** at the command line prompt with no argument. XST issues the following message.

```
--> help
ERROR:Xst:1356 - Help : Missing "-arch <family>". Please specify what family you want to target
available families:
  acr2
  aspartan3
  aspartan3a
  aspartan3adsp
  aspartan3e
  avirtex4
  fpgacore
  qrvirtex4
  qvirtex4
  spartan3
  spartan3a
  spartan3adsp
  spartan3e
  virtex4
  virtex5
  xa9500xl
  xbr
  xc9500
  xc9500xl
  xpla3
```

To see a list of commands for a specific device, type the following at the command line prompt with no argument.

help -arch *family_name*.

For example:

help -arch virtex

Use the following command to see a list of options and values for the **run** command for Virtex®-5 devices.

```
--> help -arch virtex5 -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
....
```

Setting Up an XST Script Using the Set Command

XST recognizes the Set command. The Set command accepts the options shown in the following table. For more information, see XST Design Constraints.

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
-xsthdpdir	Work Directory — location of all files resulting from VHDL or Verilog compilation	Any valid path to a directory
-xsthdpini	HDL Library Mapping File (.INI File)	file_name

Setting Up an XST Script Using the Elaborate Command

Use the Elaborate command to pre-compile VHDL and Verilog files in a specific library, or to verify Verilog files without synthesizing the design. Since compilation is included in the run, the Elaborate command is optional.

The Elaborate command accepts the options shown in the following table. For more information about these options, see XST Design Constraints.

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*
-vlgpath	Verilog Search Paths	Any valid path to directories separated by spaces, and enclosed in double quotes (**"..."**)
-vlgincdir	Verilog Include Directories	Any valid path to directories separated by spaces, and enclosed in braces (**{...}**)

Synthesizing VHDL Designs Using Command Line Mode

The following coding example shows how to synthesize a hierarchical VHDL design for a Virtex® device 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® Design Suite installation directory.

This design contains seven entities:

- stopwatch
- statmach
- tenths (a CORE Generator™ software core)
- decode
- smallcntr
- cnt60
- hex2led

Following is an example of how to synthesize a VHDL design using command line mode.

1. Create a new directory, named `vhdl_m`.
2. Copy the following files from the `ISEexamples\watchvhd` directory of the ISE Design Suite 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.

XST supports mixed VHDL and Verilog projects. Xilinx® recommends that you use the new project format, whether or not it is a real mixed language project. 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 the XST shell or the script file:

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

You must specify a top-level design block with the **`-top`** command line option.

To synthesize just **`hex2led`** and check its performance independently of the other blocks, you can specify the top-level entity to synthesize on the command line, using the **`-top`** option. For more information, see XST Specific Non-Timing Options.

```
run -ifn watchvhd.prj -ifmt mixed -ofn watchvhd.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode
Speed -opt_level 1 -top hex2led
```

During VHDL compilation, XST uses the library **`work`** as the default. If some VHDL files are to be compiled to different libraries, add the library name before the file name. For example, to compile **`hexl2led`** into the library **`my_lib`**, write the project file as follows:

```
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 my_lib hex2led.vhd
```

If XST does not recognize the order, it issues the following warning:

```
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:

- Put all VHDL files in the correct order.
- Add the **`-hdl_compilation_order`** option with value **`user`** to the XST **`run`** command:

```
run -ifn watchvhd.prj -ifmt mixed -top stopwatch -ofn watchvhd.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1 -top hex2led -hdl_compilation_order user
```

Running XST in Script Mode (VHDL)

It can be tedious to enter XST commands directly in the XST shell, especially when you have to specify several options and execute the same command several times. To run XST in script mode:

1. Open a new file named `stopwatch.xst` 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 xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
   ```

2. From the **`tcsh`** or other shell, enter the following command to begin 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

To save XST messages in a different log file (for example, `watchvhd.log`), run the following command:

```
xst -ifn stopwatch.xst -ofn watchvhd.log
```

To improve the readability of the `stopwatch.xst` file, especially if you use many options to run synthesis, place each option with its value on a separate line. Observe these rules:

- The first line contains only the **`run`** command without any options.
- There are no blank lines in the middle of the command.
- Each line (except the first one) begins with a dash (-).

An error occurs if a leading space is inadvertently entered in the value field. From 8.1i Service Pack 1 forward, ISE® Design Suite automatically strips leading spaces from a process value. Accordingly, the `.xst` file written by ISE Design Suite is not affected by leading spaces. If you hand-edit the `.xst` file and run XST from the command line, manually delete any leading spaces.

For the previous command example, `stopwatch.xst` should look like the following:

```
run
-ifn watchvhd.prj
-ifmt mixed
-top stopwatch
-ofn watchvhd.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

Synthesizing Verilog Designs Using Command Line Mode

The following coding example shows the synthesis of a hierarchical Verilog design for a Virtex® device using command line mode.

The example uses a Verilog design called *watchver*. These files are found in the `ISEexamples\watchver` directory of the ISE® Design Suite 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™ software core)
- decode
- cnt60
- smallcntr
- hex2led

For the example:

1. Create a new directory named `vlg_m`.
2. Copy the `watchver` design files from the `ISEexamples\watchver` directory of the ISE Design Suite installation directory to the newly created `vlg_m` directory.

Specify the top-level design block with the **`-top`** command line option.

To synthesize the design, which is now represented by seven Verilog files, create a project. XST now supports mixed VHDL and Verilog projects. Therefore, Xilinx® recommends 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 `watchver.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 a script file:

```
run -ifn watchver.v -ifmt mixed -top stopwatch -ofn watchver.ngc -ofmt NGC -p
xc5vfx30t-2-ff324 -opt_mode Speed -opt_level 1
```

To synthesize just HEX2LED and check its performance independently of the other blocks, specify the top-level module to synthesize in the command line, using the **`-top`** option. For more information, see XST Specific Non-Timing Options.

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.ngc -ofmt NGC -p xc5vfx30t-2-ff324 -opt_mode
Speed -opt_level 1 -top HEX2LED
```

Running XST in Script Mode (Verilog)

It can be tedious to enter XST commands directly into the XST shell, especially when you have to specify several options and execute the same command several times.

To run XST in script mode:

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 xc5vfx30t-2-ff324
   -opt_mode Speed -opt_level 1
   ```

2. From the **`tcsh`** or other shell, enter the following command to begin 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

To save XST messages in a different log file (for example, `watchver.log`), run:

```
xst -ifn design.xst -ofn watchver.log
```

To improve the readability of the `design.xst` file, especially if you use many options to run synthesis, place each option with its value on a separate line. Observe the following rules:

- The first line contains only the **`run`** command without any options.
- There are no blank lines in the middle of the command.
- Each line (except the first one) begins with a dash (-).

For the previous command example, the `design.xst` file should look like the following:

```
run
-ifn watchver.prj
-ifmt mixed
-top stopwatch
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```

Synthesizing Mixed Designs Using Command Line Mode

This example shows the synthesis of a hierarchical mixed VHDL and Verilog design for a Virtex® device 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® Design Suite installation directory to the newly-created `vhdl_verilog` directory.
   - `stopwatch.vhd`
   - `statmach.vhd`
   - `decode.vhd`
   - `cnt60.vhd`
   - `smallcntr.vhd`
   - `tenths.vhd`
3. Copy the `hex2led.v` file from the `ISEexamples\watchver` directory of the ISE Design Suite installation directory to the newly created `vhdl_verilog` directory.

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 recognizes the hierarchy and compiles Hardware Description Language (HDL) files in the correct order.

Running XST in Script Mode (Mixed Language)

It can be tedious to enter XST commands directly into the XST shell, especially when you have to specify several options and execute the same command several times. To run XST in script mode:

1. Open a new file called `stopwatch.xst` 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 xc5vfx30t-2-ff324
   -opt_mode Speed -opt_level 1
   ```

2. From the tcsh or other shell, enter the following command to begin 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

To save XST messages to a different log file (for example, `watchver.log`) run:

```
xst -ifn stopwatch.xst -ofn watchver.log
```

To improve the readability of the `stopwatch.xst` file, especially if you use many options to run synthesis, place each option with its value on a separate line. Observe the following rules:

- The first line contains only the **`run`** command without any options.
- There are no blank lines in the middle of the command.
- Each line (except the first one) begins with a dash (-).

For the previous command example, the `stopwatch.xst` file should look like:

```
run
-ifn watchver.prj
-ifmt mixed
-ofn watchver.ngc
-ofmt NGC
-p xc5vfx30t-2-ff324
-opt_mode Speed
-opt_level 1
```