Xilinx Synthesis Technology (XST) User Guide

About This Manual

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

Manual Contents

This manual contains the following chapters and appendixes.

- Chapter 1, “Introduction,” provides a basic description of XST and lists supported architectures.
- Chapter 2, “HDL Coding Techniques,” describes a variety of VHDL and Verilog coding techniques that can be used for various digital logic circuits, such as registers, latches, tristates, RAMs, counters, accumulators, multiplexers, decoders, and arithmetic operations. The chapter also provides coding techniques for state machines and black boxes.
- Chapter 3, “FPGA Optimization,” explains how constraints can be used to optimize FPGAs and explains macro generation. The chapter also describes Virtex primitives that are supported.
- Chapter 4, “CPLD Optimization,” discusses CPLD synthesis options and the implementation details for macro generation.

- Chapter 5, “Design Constraints,” describes constraints supported for use with XST. The chapter explains which attributes and properties can be used with FPGAs, CPLDs, VHDL and Verilog. The chapter also explains how to set options from the Process Properties dialog box within Project Navigator.
- Chapter 6, “VHDL Language Support,” explains how VHDL is supported for XST. The chapter provides details on the VHDL language, supported constructs, and synthesis options in relationship to XST.
- Chapter 7, “Verilog Language Support,” describes XST support for Verilog constructs and meta comments.
- Chapter 8, “Command Line Mode,” describes how to run XST using the command line. The chapter describes the xst, run, and set commands and their options.
- Appendix A, “XST Naming Conventions” discusses net naming and instance naming conventions.

Additional Resources

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

Resource	Description/URL
Tutorials	Tutorials covering Xilinx design flows, from design entry to verification and debugging `http://support.xilinx.com/support/techsup/tutorials/index.htm`
Answers Database	Current listing of solution records for the Xilinx software tools Search this database using the search function at `http://support.xilinx.com/support/searchtd.htm`
Application Notes	Descriptions of device-specific design techniques and approaches `http://support.xilinx.com/apps/appsweb.htm`

Resource	Description/URL
Data Book	Pages from *The Programmable Logic Data Book*, which contain device-specific information on Xilinx device characteristics, including readback, boundary scan, configuration, length count, and debugging `http://support.xilinx.com/partinfo/databook.htm`
Xcell Journals	Quarterly journals for Xilinx programmable logic users `http://support.xilinx.com/xcell/xcell.htm`
Technical Tips	Latest news, design tips, and patch information for the Xilinx design environment `http://support.xilinx.com/support/techsup/journals/index.htm`

Conventions

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

Typographical

The following conventions are used for all documents.

- `Courier font` indicates messages, prompts, and program files that the system displays.

  ```
  speed grade: - 100
  ```

- **`Courier bold`** indicates literal commands that you enter in a syntactical statement. However, braces "{ }" in Courier bold are not literal and square brackets "[ ]" in Courier bold are literal only in the case of bus specifications, such as bus [7:0].

  `rpt_del_net=`

  `Courier bold` also indicates commands that you select from a menu.

  `File` → **`Open`**

- *Italic font* denotes the following items.

  - Variables in a syntax statement for which you must supply values

    `edif2ngd` *`design_name`*

  - References to other manuals

See the *Development System Reference Guide* for more information.

- Emphasis in text

  If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.

- Square brackets "[ ]" indicate an optional entry or parameter. However, in bus specifications, such as bus [7:0], they are required.

  edif2ngd *[option_name] design_name*

- Braces "{ }" enclose a list of items from which you must choose one or more.

  lowpwr ={**on**|**off**}

- A vertical bar " | " separates items in a list of choices.

  lowpwr ={**on**|**off**}

- A vertical ellipsis indicates repetitive material that has been omitted.

  ```
  IOB #1: Name = QOUT'
  IOB #2: Name = CLKIN'
  .
  .
  .
  ```

- A horizontal ellipsis "...." indicates that an item can be repeated one or more times.

  `allow block` *block_name loc1 loc2locn;*

Online Document

The following conventions are used for online documents.

- Red-underlined text indicates an interbook link, which is a cross-reference to another book. Click the red-underlined text to open the specified cross-reference.

- Blue-underlined text indicates an intrabook link, which is a cross-reference within a book. Click the blue-underlined text to open the specified cross-reference.

Contents

Chapter 3 FPGA Optimization

Chapter 5 Design Constraints

Chapter 1

Introduction

This chapter contains the following sections.

- "Architecture Support"
- "XST Flow"

Architecture Support

The XST software supports only the following Xilinx architectures in this release.

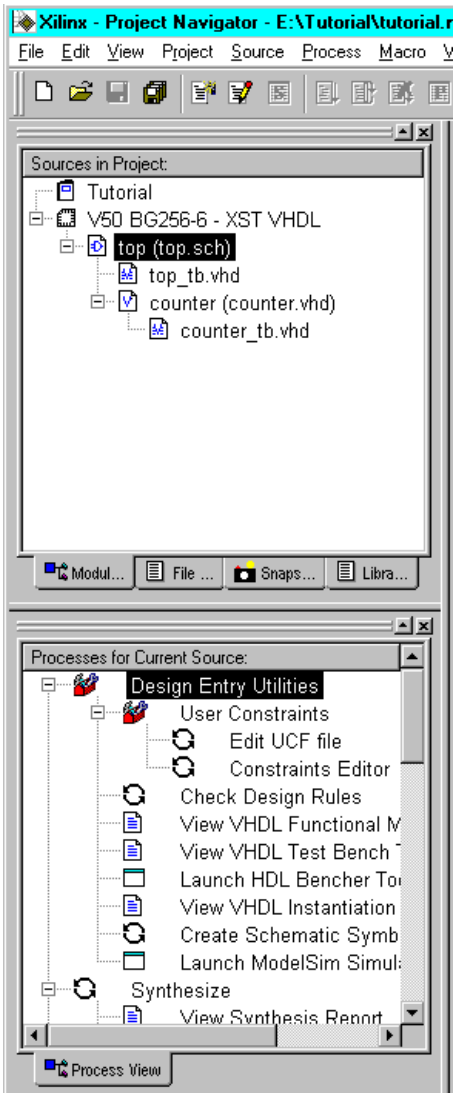
- Spartan™ / -II
- Virtex™ / -E / -II
- XC9500™ / XL / XV

XST Flow

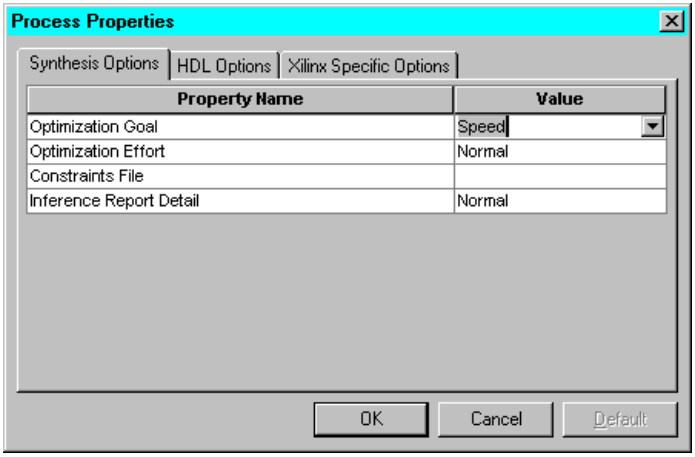
XST is a Xilinx tool that synthesizes HDL designs to create EDIF netlists. This manual describes XST support for Xilinx devices, HDL languages, and design constraints. The manual explains how to use various design optimization and coding techniques when creating designs for use with XST.

Before you synthesize your design, you can set a variety of options for XST.

1. Select your top-level design in the Source window.



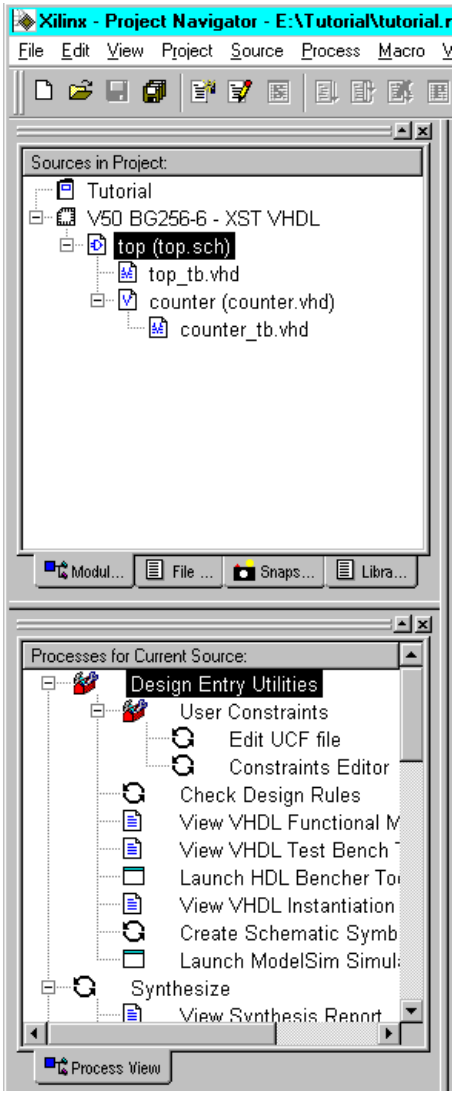
2. To set the options, right click Synthesize in the Process window.
3. Select Properties to display the Process Properties dialog box.



4. Set the desired Synthesis, HDL, and Xilinx Specific Options.

   For a complete description of these options, refer to the “Setting Constraints and Options” section in the “Design Constraints” chapter.

5. When a design is ready to synthesize, you can invoke XST within the Project Navigator. With the top-level source file selected, double-click Synthesize in the Process window.



Note To run XST from the command line, refer to the "Command Line Mode" chapter for details.

6. When synthesis is complete, view the results by double-clicking View Synthesis Report. Following is a portion of a sample report.

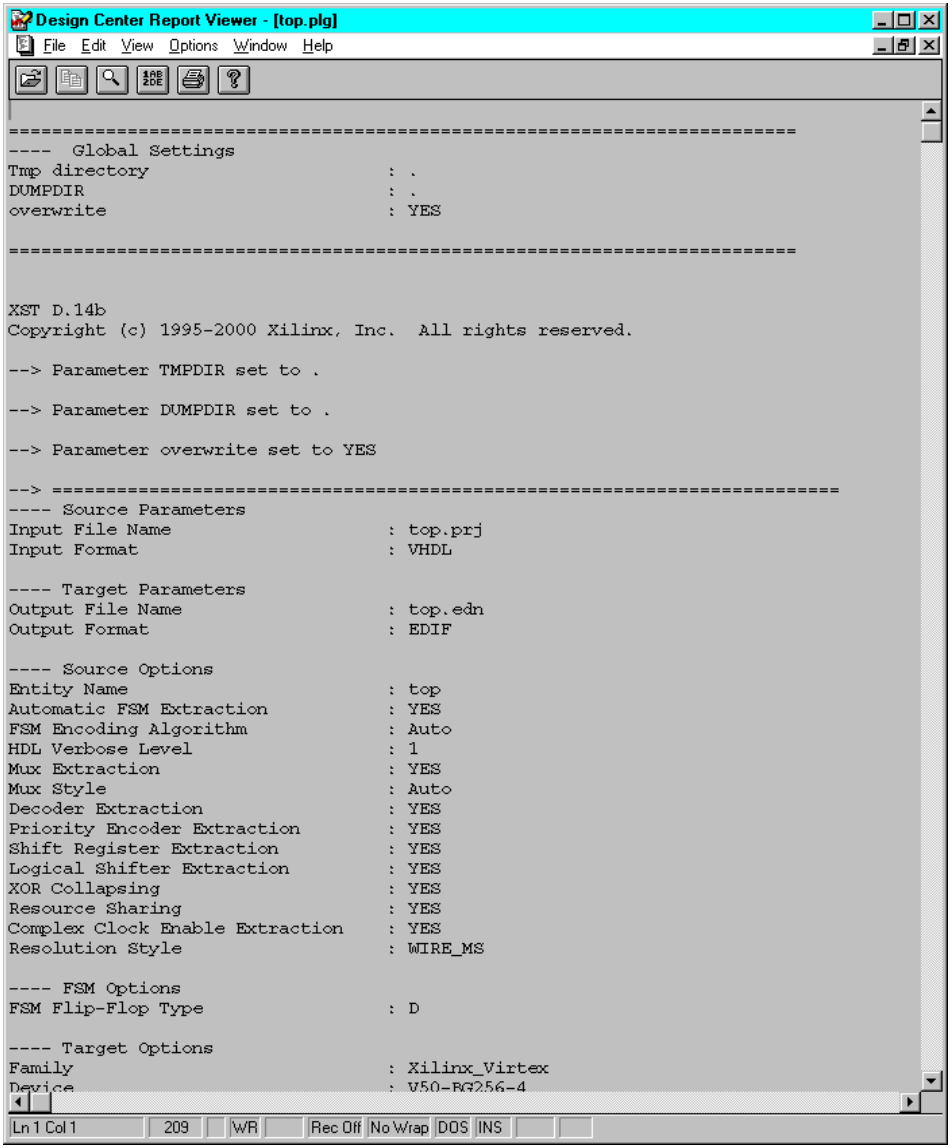


Figure 1-1 View Synthesis Report

Chapter 2

HDL Coding Techniques

This chapter contains the following sections:

Introduction

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

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

Note Please refer to the “Design Constraints” chapter for more details on constraints and their utilization.

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

- The set of macros and associated signals, inferred by XST from the VHDL / Verilog source on a block by block basis
- The overall statistics of recognized macros
- The number and type of macros preserved by low level optimization.

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

```
Synthesizing Unit <timer>.
  Extracting finite state machine <FSM_0> for signal <state>.
  ...
  Extracting 4-bit register for signal <min1>.
  ...
  Extracting 4-bit addsub for internal node.
  Summary:
    inferred   1 Finite State Machine(s).
    inferred  17 D-type flip-flop(s).
    inferred   4 Adder/Subtractor(s).
Unit <timer> synthesized.

Synthesizing Unit <decod>.
  Extracting 1-bit xor2 for signal <colon>.
  Extracting 1-bit xor2 for internal node.
  ...
  Extracting 1-bit xor2 for internal node.
  Summary:
    inferred  29 Xor(s).
Unit <decod> synthesized.
...
```

The following example displays the overall statistics of recognized macros.

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

 Macro Statistics
 # FSMs                :1
 # Registers           :5
    4-bit register     :4
    1-bit register     :1
 # Adders/Subtractors  :4
    4-bit addsub       :3
    4-bit subtractor   :1
 # Xors                :29
     1-bit xor2        :29

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

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

```
...
=========================================
Final Results
...
Macro Statistics
# FSMs                : 1
# Adders/Subtractors  : 4
4-bit addsub          : 3
4-bit subtractor      : 1
...
=========================================
```

This chapter discusses the following Macro Blocks:

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

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

Note For macro implementation details please refer to the "FPGA Optimization" chapter and the "CPLD Optimization" chapter.

The following table provides a list of all the examples in this chapter as well as a list of VHDL and Verilog synthesis templates available from the Language Templates in the Project Navigator.

1. To access the synthesis templates from the Project Navigator, select **`Edit`** → **`Language Templates...`**
2. Click the + sign for either VHDL or Verilog.
3. Click the + sign next to Synthesis Templates.

Table 2-1 VHDL and Verilog Examples and Templates

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

Table 2-1 VHDL and Verilog Examples and Templates

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

Table 2-1 VHDL and Verilog Examples and Templates

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

Table 2-1 VHDL and Verilog Examples and Templates

Macro Blocks	Chapter Examples	Language Templates
Multiplexers	4-to-1 1-bit MUX using IF Statement 4-to-1 MUX Using CASE Statement 4-to-1 MUX Using Tristate Buffers No 4-to-1 MUX	 4-to-1 MUX Design with CASE Statement 4-to-1 MUX Design with Tristate Construct
Decoders	VHDL (One-Hot) Verilog (One-Hot) VHDL (One-Cold) Verilog (One-Cold)	3-to-8 Decoder, Synchronous with Reset
Priority Encoders	3-Bit 1-of-9 Priority Encoder	8-to-3 encoder, Synchronous with Reset
Logical Shifters	Example 1 Example 2 Example 3	None

Table 2-1 VHDL and Verilog Examples and Templates

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

Table 2-1 VHDL and Verilog Examples and Templates

Macro Blocks	Chapter Examples	Language Templates
RAMs	Single Port RAM with Asynchronous Read Single Port RAM with "false" Synchronous Read Single-Port RAM with Synchronous Read (Read Through) Dual-port RAM with Asynchronous Read Dual-port RAM with False Synchronous Read Dual-port RAM with Synchronous Read (Read Through) Multiple-Port RAM Descriptions	Single Port Block RAM Single Port Distributed RAM Dual Port Block RAM Dual Port Distributed RAM
State Machines	FSM: 1 Process FSM: 2 Processes FSM: 3 Processes	Binary State Machine One-Hot State Machine
Black Boxes	VHDL Verilog	

Signed/Unsigned Support

In XST, some macros such as adders or counters can be implemented for signed and unsigned values. If you use VHDL, then depending on the operation and type of the operands, you have to include additional packages in your code. For example, in order to create an Unsigned Adder you can use the following arithmetic packages and types operating on unsigned values:

PACKAGE	TYPE
numeric_std	unsigned
numeric_unsigned	std_logic_vector
std_logic_arith	unsigned
std_logic_unsigned	std_logic_vector

In order to create a Signed Adder you can use arithmetic packages and types operating on signed values.

PACKAGE	TYPE
numeric_std	signed
numeric_signed	std_logic_vector
std_logic_arith	signed
std_logic_signed	std_logic_vector

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

Registers

XST is able to recognize flip-flops with the following control signals:

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

Log File

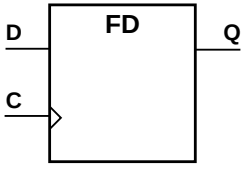
The XST log file reports the type and size of recognized flip-flops during the macro recognition step:

```
...
Synthesizing Unit <flop>.
  Extracting 1-bit register for signal <q>.
  Summary:
    inferred   1 D-type flip-flop(s).

Unit <flop> synthesized.
...

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

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

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

Related Constraints

A related constraint is IOB.

DFF with Positive-Edge Clock



X3715

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

VHDL Code

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

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

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

Verilog Code

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

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

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

```
    end
endmodule
```

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

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

you can also use

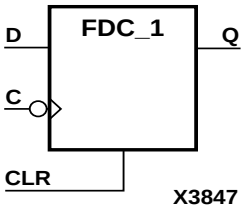
```
if (rising_edge(C)) then
```

and for negative-edge one you can use the

```
if (falling_edge(C)) then
```

construct.

DFF with Negative-Edge Clock and Asynchronous Clear



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

VHDL Code

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

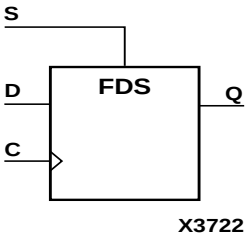
```
library ieee;
use ieee.std_logic_1164.all;

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

Verilog Code

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

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

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

DFF with Positive-Edge Clock and Synchronous Set



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

VHDL Code

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

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

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

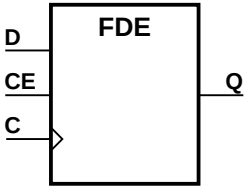
```
          end if;
        end if;
    end process;
end archi;
```

Verilog Code

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

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

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

DFF with Positive-Edge Clock and Clock Enable



X8361

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

VHDL Code

Following is the equivalent VHDL code for the DFF with a positive-edge clock and clock Enable.

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

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

Verilog Code

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

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

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

Latches

XST is able to recognize latches with the Asynchronous Set/Clear control signals.

Latches can be described using:

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

Log File

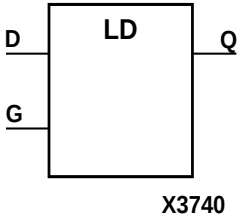
The XST log file reports the type and size of recognized latches during the macro recognition step:

```
...
Synthesizing Unit <latch>.
    Extracting 1-bit latch for signal <q>.
    Summary:
      inferred 1 D-type latch(s).
Unit <latch> synthesized.
...
=============================
HDL Synthesis Report

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

Related Constraints

A related constraint is IOB.

Latch with Positive Gate



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

VHDL Code

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

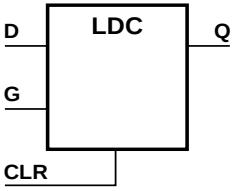
```
library ieee;
use ieee.std_logic_1164.all;

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

Verilog Code

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

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

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

Latch with Positive Gate and Asynchronous Clear



X4070

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

VHDL Code

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

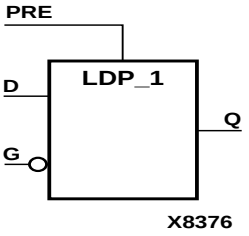
```
library ieee;
use ieee.std_logic_1164.all;

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

Verilog Code

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

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

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

4-bit Latch with Inverted Gate and Asynchronous Preset



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

VHDL Code

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

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

entity latch is
  port(D : in std_logic_vector(3 downto 0);
       G, PRE : in  std_logic;
       Q : out std_logic_vector(3 downto 0));
end latch;
architecture archi of latch is
  begin
    process (PRE, G)
      begin
        if (PRE='1') then
          Q <= "1111";
        elsif (G='0') then
```

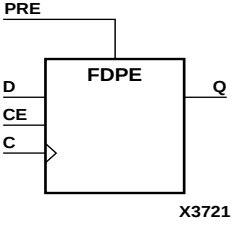
```
          Q <= D;
        end if;
    end process;
end archi;
```

Verilog Code

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

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

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

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



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

VHDL Code

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

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

entity flop is
  port(C, CE, PRE : in std_logic;
       D : in  std_logic_vector (3 downto 0);
       Q : out std_logic_vector (3 downto 0));
end flop;
architecture archi of flop is
  begin
    process (C, PRE)
      begin
        if (PRE='1') then
          Q <= "1111";
        elsif (C'event and C='1')then
          if (CE='1') then
            Q <= D;
          end if;
        end if;
    end process;
end archi;
```

Verilog Code

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

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

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

Tristates

Tristate elements can be described using the following:

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

Log File

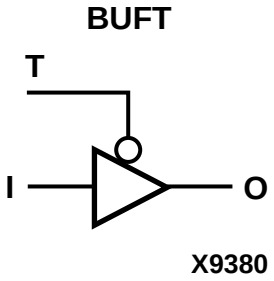
The XST log reports the type and size of recognized tristates during the macro recognition step:

```
...
Synthesizing Unit <three_st>.
  Extracting tristate(s) for signal <q>.
  Summary:
      inferred   1 Tristate(s).
Unit <three_st> synthesized.
...
=============================
HDL Synthesis Report

Found no macro
=============================
...
```

Related Constraints

There are no related constraints available.

Description Using Combinatorial Process and Always Block



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

VHDL Code

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

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

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

Verilog Code

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

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

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

Description Using Concurrent Assignment

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

VHDL Code

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

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

entity three_st is
  port(T  : in  std_logic;
       I : in  std_logic;
       O : out std_logic);
end three_st;
architecture archi of three_st is
  begin
      O <= I when (T='0')
             else 'Z';
end archi;
```

Verilog Code

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

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

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

Counters

XST is able to recognize counters with the following controls signals:

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

HDL coding styles for the following control signals are equivalent to the ones described in the “Registers” section of this chapter:

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

Moreover, XST supports unsigned as well as signed counters.

Log File

The XST log file reports the type and size of recognized counters during the macro recognition step:

```
...
Synthesizing Unit <counter>.
  Extracting 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
============================
...
```

4-bit Unsigned Up Counter with Asynchronous Clear

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

VHDL Code

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

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

entity counter is
  port(C, CLR : in  std_logic;
```

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

Verilog Code

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

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

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

4-bit Unsigned Down Counter with Synchronous Set

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

VHDL Code

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

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

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

Verilog Code

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

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

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

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

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

VHDL Code

Following is the VHDL code for a 4-bit unsigned Up Counter with asynchronous load from primary input.

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

entity counter is
```

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

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

Verilog Code

Following is the Verilog code for a 4-bit unsigned Up Counter with asynchronous load from primary input.

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

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

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

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

VHDL Code

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

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

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

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

Verilog Code

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

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

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

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

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

VHDL Code

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

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

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

Verilog Code

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

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

  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
```

```
      else
        if (CE)
          tmp = tmp + 1'b1;
    end
  assign Q = tmp;
endmodule
```

4-bit Unsigned Up/Down counter with Asynchronous Clear

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

VHDL Code

Following is the VHDL code for a 4-bit unsigned Up / Down counter with asynchronous clear.

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

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

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

```
        elsif (C'event and C='1') then
          if (up_down='1') then
            tmp <= tmp + 1;
          else
            tmp <= tmp - 1;
          end if;
        end if;
    end process;
    Q <= tmp;
end archi;
```

Verilog Code

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

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

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

4-bit Signed Up Counter with Asynchronous Reset

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

VHDL Code

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

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

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

Verilog Code

There is no equivalent Verilog code.

No constraints are available.

Accumulators

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

- In a counter, the destination and first operand is a signal or variable and the other operand is a constant equal to 1: A <= A + 1.
- In an accumulator, the destination and first operand is a signal or variable, and the second operand is either:
  - a signal or variable: A <= A + B.
  - a constant not equal to 1: A <= A + Constant.

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

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

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

Log File

The XST log file reports the type and size of recognized accumulators during the macro recognition step:

```
...
Synthesizing Unit <counter>.
  Extracting 4-bit up accumulator for signal <tmp>.
  Summary:
    inferred 1 Accumulator(s).
Unit <counter> synthesized.
...
============================
HDL Synthesis Report

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

4-bit Unsigned Up Accumulator with Asynchronous Clear

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

VHDL Code

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

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

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

Verilog Code

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

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

  always @(posedge C or posedge CLR)
    begin
      if (CLR)
        tmp = 4'b0000;
```

```
      else
        tmp = tmp + D;
    end
  assign Q = tmp;
endmodule
```

No constraints are available.

Shift Registers

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

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

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

- concatenation operator

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

- "for loop" construct

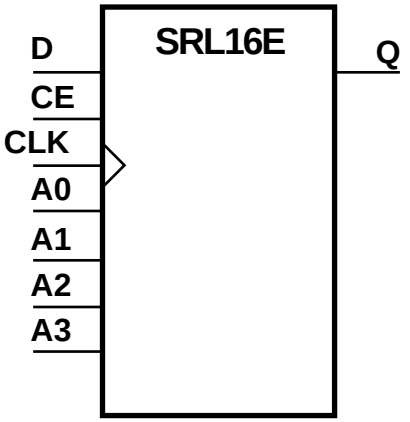
  ```
  for i in 0 to 6 loop
  shreg(i+1) <= shreg(i);
  end loop;
  shreg(0) <= SI;
  ```

- predefined shift operators as sll, srl, for example.

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

FPGAs:

Before writing Shift Register behavior it is important to recall that Virtex has a specific hardware resources to implement them: SRL16.



However, SRL16 supports only LEFT shift operation for a limited number of IO signals:

- Clock
- Clock enable
- Serial data in
- Serial data out

It means, that if your shift register *does have*, for instance, a synchronous parallel load, no SRL16 will be implemented. XST will not try to infer SR4x, SR8x or SR16x macros. It will use specific internal processing which allows it to bring the best final results.

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

Note In VHDL you have to use the "for loop" construct to infer SRL16. In Verilog, you have to use the "<<" construct to infer SRL16. XST currently does not infer SRL16 when using the concatenation operator.

Log File

The XST log file reports the type and size of recognized shift registers during the macro recognition step:

```
...
Synthesizing Unit <shift>.
  Extracting  8-bit  shift  register  for  signal
<tmp<7>>.
  Summary:
    inferred 1 Shift Register(s).
Unit <shift> synthesized.
...
=============================
HDL Synthesis Report
Macro Statistics
# Shift Registers: 1
  8-bit shift register: 1
=============================
...
```

Related Constraints

A related constraint is shreg_extract.

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

Note For this example, XST will infer SRL16.

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

VHDL Code

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

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

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

Verilog Code

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

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

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

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

Note For this example, XST will infer SRL16E_1.

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

VHDL Code

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

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

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

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

Verilog Code

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

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

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

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

Note For this example XST will not infer SRL16, therefore the concatenation operator can be used.

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

VHDL Code

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

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

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

Verilog Code

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

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

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

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

Note For this example XST will not infer SRL16; therefore, the concatenation operator can be used.

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

VHDL Code

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

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

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

Verilog Code

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

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

  always @(posedge C)
  begin
    if (S)
      tmp = 8'b11111111;
    else
```

```
      begin
        tmp = {tmp[6:0], SI};
      end
  end
  assign SO  = tmp[7];
endmodule
```

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

Note For this example XST will not infer SRL16; therefore, the concatenation operator can be used.

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

VHDL Code

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

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

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

Verilog Code

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

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

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

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

Note For this example XST will not infer SRL16; therefore, the concatenation operator can be used.

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

VHDL Code

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

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

entity shift is
  port(C, SI, ALOAD : in std_logic;
```

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

Verilog Code

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

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

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

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

Note For this example XST will not infer SRL16; therefore, the concatenation operator can be used.

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

VHDL Code

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

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

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

```
    SO <= tmp(7);
end archi;
```

Verilog Code

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

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

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

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

Note For this example XST will not infer SRL16; therefore, the concatenation operator can be used.

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

VHDL Code

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

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

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

Verilog Code

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

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

  always @(posedge C)
  begin
    if (left_right==1'b0)
      begin
        tmp = {tmp[6:0], SI};
```

```
      end
    else
      begin
        tmp = {SI, tmp[6:0]};
      end
  end
  assign PO  = tmp;
endmodule
```

Multiplexers

XST supports different description styles for multiplexers: if-then-else, case, for example. When writing MUXs, you must pay particular attention in order to avoid traps. For example, if you describe a MUX using a Case statement and you do not specify all values of the selector you may get latches instead of a muliplexer. Writing MUXs you can also use “don't cares” to describe selector values.

During the macro inference step, XST makes a decision to infer or not infer the MUXs. For example, if the MUX has several inputs which are the same, then XST can decide not to infer it. In the case that you do want to infer the MUX, you can force XST using the design constraint called "mux_extract".

If you use Verilog, then you have to be aware that Verilog case statements can be either full / not full or parallel / not parallel. A case statement is:

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

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

Full and Parallel Case

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

  reg [1:0] o1;

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

not Full but Parallel

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

  reg [1:0] o1;

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

neither Full nor Parallel

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

  reg [1:0] o1;

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

XST automatically determines the characteristics of the case statements and generates logic using Multiplexers, Priority Encoders and Latches that best implement the exact behavior of the case statement.

This characterization of the case statements can be guided or modified by using the Case Implementation Style parameter (Please refer to the "Design Constraints" chapter for more details). Accepted values for this parameter are **`default`**, **`full`**, **`parallel`** and **`full-parallel`**:

- If the default is used, XST will implement the exact behavior of the case statements.
- If full is used, XST will consider that case statements are complete and will avoid latch creation.
- If parallel is used, XST will consider that the branches cannot occur in parallel and will not use a priority encoder.
- If full-parallel is used, XST will consider that case statements are complete and that the branches cannot occur in parallel, therefore saving latches and priority encoders.

The following table indicates the *resources* used to synthesize the three examples above using the four Case Implementation Styles. The term "resources" means the functionality. For example, if using "notfull_notparallel" with the Case Implementation Style "default", from the functionality point of view XST will implement Priority Encoder + Latch. But, it does not inevitably mean that XST will *infer* the priority encoder during the macro recognition step.

Case Implementation	Full	notfull	notfull_notparallel
default	MUX	Latch	Priority Encoder + Latch
parallel		Latch	Latch
full		MUX	Priority Encoder
full-parallel		MUX	MUX

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

Log File

The XST log file reports the type and size of recognized MUXs during the macro recognition step:

```
...
Synthesizing Unit <mux>.
    Extracting 1-bit 4-to-1 multiplexer for signal
<o>.
    Summary:
  inferred   1 Multiplexer(s).
Unit <mux> synthesized.
...
============================
Macro Statistics
# Multiplexers : 1
  1-bit 4-to-1 multiplexer : 1
============================
...
```

Related Constraints

Related constraints are mux_extract and mux_style.

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

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

VHDL Code

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

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

entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);
end mux;
architecture archi of mux is
  begin
    process (a, b, c, d, s)
    begin
      if    (s = "00") then o <= a;
      elsif (s = "01") then o <= b;
      elsif (s = "10") then o <= c;
      else o <= d;
      end if;
  end process;
end archi;
```

Verilog Code

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

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

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

4-to-1 MUX Using CASE Statement

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

VHDL Code

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

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

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

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

Verilog Code

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

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

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

4-to-1 MUX Using Tristate Buffers

This section shows VHDL and Verilog examples for a 4-to-1 Mux using tristate buffers.

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

VHDL Code

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

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

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

architecture archi of mux is
begin

o <= a when (s(0)='0') else 'Z';
o <= b when (s(1)='0') else 'Z';
o <= c when (s(2)='0') else 'Z';
o <= d when (s(3)='0') else 'Z';

end archi;
```

Verilog Code

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

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

```
output o;

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

endmodule
```

No 4-to-1 MUX

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

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

VHDL Code

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

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

entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);
end mux;
architecture archi of mux is
  begin
    process (a, b, c, d, s)
    begin
      if    (s = "00") then o <= a;
      elsif (s = "01") then o <= b;
      elsif (s = "10") then o <= c;
```

```
      end if;
    end process;
end archi;
```

Verilog Code

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

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

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

Decoders

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

Log File

The XST log file reports the type and size of recognized decoders during the macro recognition step:

```
...
Synthesizing Unit <dec>.
    Extracting 1-of-8 decoder for signal <res>.
    Summary:
  inferred   1 Decoder(s).
Unit <dec> synthesized.
...
=============================
Macro Statistics
# Decoders : 1
  1-of-8 decoder: 1
=============================
...
```

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

Related Constraints

A related constraint is decoder_extract.

VHDL (One-Hot)

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

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

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

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

Verilog (One-Hot)

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

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

  always @(sel or res)
  begin
    case (sel)
      8'b000   : res = "00000001";
      8'b001   : res = "00000010";
      8'b010   : res = "00000100";
      8'b011   : res = "00001000";
      8'b100   : res = "00010000";
      8'b101   : res = "00100000";
      8'b110   : res = "01000000";
      default  : res = "10000000";
    endcase
  end
endmodule
```

VHDL (One-Cold)

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

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

entity dec is
```

```
  port (sel: in std_logic_vector (2 downto 0);
        res: out std_logic_vector (7 downto 0));
end dec;
architecture archi of dec is
  begin
    res <=  "11111110" when sel = "000" else
            "11111101" when sel = "001" else
            "11111011" when sel = "010" else
            "11110111" when sel = "011" else
            "11101111" when sel = "100" else
            "11011111" when sel = "101" else
            "10111111" when sel = "110" else
            "01111111";
end archi;
```

Verilog (One-Cold)

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

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

  always @(sel)
  begin
    case (sel)
      8'b000   : res = "11111110";
      8'b001   : res = "11111101";
      8'b010   : res = "11111011";
      8'b011   : res = "11110111";
      8'b100   : res = "11101111";
      8'b101   : res = "11011111";
      8'b110   : res = "10111111";
      default  : res = "01111111";
    endcase
  end
endmodule
```

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

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

VHDL

Following is the VHDL code.

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

entity dec is
  port (sel: in std_logic_vector (2 downto 0);
        res: out std_logic_vector (7 downto 0));
end dec;
architecture archi of dec is
  begin
    res <=  "00000001" when sel = "000" else
    -- 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;
```

Verilog

Following is the Verilog code.

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

  always @(sel)
  begin
    case (sel)
      8'b000   : res = "00000001";
      // unused decoder output
      8'b001   : res = "xxxxxxxx";
      8'b010   : res = "00000100";
      8'b011   : res = "00001000";
      8'b100   : res = "00010000";
      8'b101   : res = "00100000";
      8'b110   : res = "01000000";
      default  : res = "10000000";
    endcase
  end
endmodule
```

On the contrary, the following description leads to the inference of a 1-of-8 decoder.

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

VHDL

Following is the VHDL code.

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

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

Verilog

Following is the Verilog code.

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

  always @(sel or res)
  begin
    case (sel)
      8'b000   : res = "00000001";
      8'b001   : res = "00000010";
      8'b010   : res = "00000100";
      8'b011   : res = "00001000";
      8'b100   : res = "00010000";
      8'b101   : res = "00100000";
  // 110 and 111 selector values are unused
      default  : res = "xxxxxxxx";
    endcase
  end
endmodule
```

Priority Encoders

XST is able to recognize a priority encoder. But in most cases XST will not infer it. You have to use the "priority_extract" constraint with the value "Force" to force priority encoder inference. Xilinx strongly suggests that you use this constraint on the signal-by-signal basis; otherwise, the constraint may guide you towards sub-optimal results.

Log File

The XST log file reports the type and size of recognized priority encoders during the macro recognition step:

```
Synthesizing Unit <pencod>.
    Extracting 3-bit 1-of-9 priority encoder for
signal <code>.
    Summary:
  inferred 3 Priority encoder(s).
Unit <pencod> synthesized.
...
============================
HDL Synthesis Report


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

3-Bit 1-of-9 Priority Encoder

Note For this example XST may infer a priority encoder. You must use the "priority_extract" constraint with a value "Force" to force its inference.

Related Constraint

A related constraint is priority_extract.

VHDL

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

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

entity priority is
  port ( sel     : in std_logic_vector (7 downto 0);
         code    : out std_logic_vector (2 downto 0));
end priority;
architecture archi of priority is
  begin
    code <= "000" when sel(0) = '1' else
            "001" when sel(1) = '1' else
            "010" when sel(2) = '1' else
            "011" when sel(3) = '1' else
            "100" when sel(4) = '1' else
            "101" when sel(5) = '1' else
            "110" when sel(5) = '1' else
            "111" when sel(5) = '1' else
            "xxx";
end archi;
```

Verilog

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

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

  always @(sel)
  begin
                if (sel[0]) code <= "000";
           else if (sel[1]) code <= "001";
           else if (sel[2]) code <= "010";
           else if (sel[3]) code <= "011";
           else if (sel[4]) code <= "100";
           else if (sel[5]) code <= "101";
           else if (sel[6]) code <= "110";
           else if (sel[7]) code <= "111";
```

```
            else              code <= 3'bxxx;
  end
endmodule
```

Logical Shifters

Xilinx defines a Logical Shifter as a combinatorial circuit with 2 inputs and 1 output:

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

Note *All* these IOs are mandatory; otherwise, XST will *not* infer a Logical Shifter.

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

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

Log File

The XST log file reports the type and size of a recognized logical shifter during the macro recognition step:

```
...
Synthesizing Unit <shift>.
    Extracting  combinational  logic  shifter  for
signal <so>.
    Summary:
  inferred   1 Combinational logic shifter(s).
Unit <shift> synthesized.
...
============================
Macro Statistics
# Logic shifters                  : 1
  8-bit shifter logical left      : 1
============================
...
```

Related Constraints

A related constraint is shift_extract.

Example 1

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

VHDL

Following is the VHDL code for a logical shifter.

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

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

Verilog

Following is the Verilog code for a logical shifter.

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

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

Example 2

XST will *not* infer a Logical Shifter for this example.

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

VHDL

Following is the VHDL code.

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

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

Verilog

Following is the Verilog code.

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

  always @(DI or sel)
  begin
```

```
    case (sel)
      2'b00   : SO <= DI;
      2'b01   : SO <= DI << 1;
      default : SO <= DI << 2;
      endcase
  end
endmodule
```

Example 3

XST will *not* infer a Logical Shifter for this example.

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

VHDL

Following is the VHDL code.

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

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

Verilog

Following is the Verilog code.

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

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

Arithmetic Operations

XST supports the following arithmetic operations:

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

Adders, Subtractors, Comparators and Multipliers are supported for signed and unsigned operations.

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

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

Adders, Subtractors, Adders/Subtractors

This section provides HDL examples of adders and subtractors

Log File

The XST log file reports the type and size of recognized adder, subtractor and adder / subtractor during the macro recognition step:

```
..
Synthesizing Unit <adder>.
    Extracting 8-bit adder carry in for signal
<sum>.
    Summary:
  inferred   1 Adder/Subtractor(s).
Unit <adder> synthesized.
...
=============================
HDL Synthesis Report


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

Unsigned 8-bit Adder

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

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

VHDL

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

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

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

Verilog

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

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

  assign SUM = A + B;
endmodule
```

Unsigned 8-bit Adder with Carry In

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

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

VHDL

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

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

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

Verilog

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

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

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

Unsigned 8-bit Adder with Carry Out

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

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

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

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

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

  In this case, XST recognizes that this 9-bit adder can be implemented as 8-bit ones with Carry Out.

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

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

VHDL

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

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

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

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

In the preceding example, two arithmetic packages are used:

- std_logic_arith. It contains the integer to std_logic conversion function, that is, conv_std_logic_vector.
- std_logic_unsigned. It contains the unsigned "+" operation.

Verilog

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

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

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

Unsigned 8-bit Adder with Carry In and Carry Out

This section contains VHDL and Verilog code for an unsigned 8-bit adder with Carry In and Carry Out.

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

VHDL

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

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

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

Verilog

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

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

Simple Signed 8-bit Adder

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

VHDL

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

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

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

Verilog

There is no equivalent Verilog code.

Unsigned 8-bit Subtractor

The following table describes the IO pins for an unsigned 8-bit subtractor.

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

VHDL

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

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

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

Verilog

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

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

  assign RES = A - B;
endmodule
```

Unsigned 8-bit Adder/Subtractor

The following table describes the IO pins for an unsigned 8-bit adder/subtractor.

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

VHDL

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

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

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

Verilog

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

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

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

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

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

Log File

The XST log file reports the type and size of recognized comparators during the macro recognition step:

```
...
Synthesizing Unit <compar>.
    Extracting 8-bit comparator greatequal for internal node.
    Summary:
  inferred   1 Comparator(s).
Unit <compar> synthesized.
...
============================
HDL Synthesis Report


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

Unsigned 8-bit Greater or Equal Comparator

The following table describes the IO pins for the comparator.

IO pins	Description
A[7:0], B[7:0]	Add/Sub Operands
cmp	Comparison Result

VHDL

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

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

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

Verilog

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

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

  assign cmp = A >= B ? 1'b1 : 1'b0;
endmodule
```

Multipliers

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

Log File

The XST log file reports the type and size of recognized multipliers during the macro recognition step:

```
...
Synthesizing Unit <mult>.
    Extracting 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
============================
...
```

Unsigned 8x4-bit Multiplier

The following table describes the IO pins.

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

VHDL

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

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

entity mult is
  port(A : in std_logic_vector(7 downto 0);
       B : in std_logic_vector(3 downto 0);
```

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

Verilog

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

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

  assign RES = A * B;
endmodule
```

Dividers

Divisions are not supported, except when the divisor is a constant and is a power of 2. In that case, the operator is implemented as a shifter; otherwise, an error message will be issued by XST.

Log File

When you implement a division with a constant with the power of 2, XST does not issue any message during the macro recognition step. In case your division does not correspond to the case supported by XST, the following error message displays:

```
...
ERROR :   (VHDL_0045). des.vhd (Line 11).
Operator is not supported yet : 'INVALID OPERATOR'
...
```

Division By Constant 2

The following table describes the IO pins.

IO pins	Description
DI[7:0], B[7:0]	DIV Operands
DO[7:0]	DIV Result

VHDL

Following is the VHDL code for a Division By Constant 2 divider.

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

entity divider is
  port(DI  : in std_logic_vector(7 downto 0);
       DO  : out std_logic_vector(7 downto 0));
end divider;
architecture archi of divider is
  begin
    DO <= DI / 2;
end archi;
```

Verilog

Following is the Verilog code for a Division By Constant 2 divider.

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

  assign DO = DI / 2;
endmodule
```

Resource Sharing

The goal of resource sharing (also known as folding) is to minimize the number of operators and the subsequent logic in the synthesized design. This optimization is based on the principle that two similar arithmetic resources may be implemented as one single arithmetic operator if they are never used at the same time. XST performs both resource sharing and, if required, reduces of the number of multiplexers that are created in the process.

XST supports resource sharing for adders, subtractors, adders / subtractors and multipliers.

Log File

The XST log file reports the type and size of recognized arithmetic blocks and multiplexers during the macro recognition step:

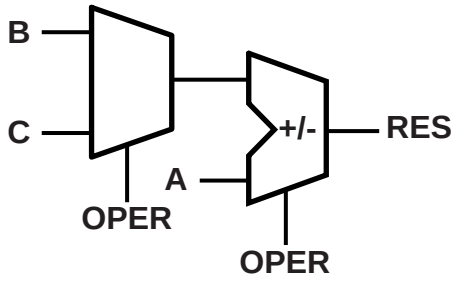
```
...
Synthesizing Unit <addsub>.
    Extracting 8-bit 2-to-1 multiplexer for internal node.
    Extracting 8-bit addsub for signal <res>.
    Summary:
  inferred   1 Adder/Subtractor(s).
  inferred   8 Multiplexer(s).
Unit <addsub> synthesized.
...
=============================
HDL Synthesis Report

Macro Statistics
# Multiplexers                      : 1
  8-bit 2-to-1 multiplexer          : 1
# Adders/Subtractors                : 1
  8-bit addsub                      : 1
=============================
...
```

Related Constraint

The related constraint is resource_sharing.

Example

For the following VHDL / Verilog example, XST will give the following solution:



X8984

IO pins	Description
A[7:0], B[7:0], B[7:0]	DIV Operands
oper	Operation Selector
RES[7:0]	Data Output

VHDL

Following is the VHDL example for resource sharing.

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

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

```
                    else A - C;
end archi;
```

Verilog

Following is the Verilog code for resource sharing.

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

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

RAMs

If you do not want to instantiate RAM primitives in order to keep your HDL code technology independent, XST offers an automatic RAM recognition capability. XST can infer distributed as well as Block RAM. It covers the following characteristics, offered by these RAM types:

- Synchronous write
- Write enable
- Asynchronous or synchronous read
- Reset of the data output latches
- Single, dual or multiple-port read
- Single port write

The type of the 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.

Here is the list of VHDL / Verilog templates which will be described below:

- Single port RAM with asynchronous read
- Single port RAM with "false" synchronous read
- Single-port RAM with synchronous read (Read Through)
- Dual-port RAM with asynchronous read
- Dual-port RAM with false synchronous read
- Dual-port RAM with synchronous read (Read Through)
- Multiple-port RAM descriptions

If a given template can be implemented using Block and Distributed RAM, XST will implement BLOCK ones. You can use the "ram_style" attribute to control RAM implementation and select a desirable RAM type. Please refer to the “Design Constraints” chapter for more details.

Please note that the following features specifically available with Block RAM are *not* yet supported:

- Dual write port
- Data output reset
- RAM enable
- Different aspect ratios on each port

Please refer to the “FPGA Optimization” chapter for more details on RAM implementation.

Log File

The XST log file reports the type and size of recognized RAM as well as complete information on its I/O ports during the macro recognition step:

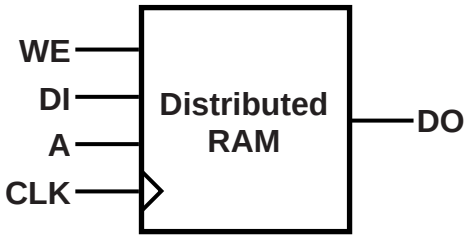
```
...
Synthesizing Unit <raminfr>.
  Extracting 128-bit single-port RAM for signal <ram>.
-----------------------------------------------------------------
-
| ram type         | distributed                        |         |
| implementation   | auto                               |         |
| aspect ratio     | 32-word x 4-bit                    |         |
| clock            | connected to signal <clk>          | rise    |
| write enable     | connected to signal <we>           | high    |
| address          | connected to signal <a>            |         |
| data in          | connected to signal <di>           |         |
| data out         | connected to signal <do>           |         |
-----------------------------------------------------------------
-
    Summary:
  inferred   1 RAM(s).
Unit <raminfr> synthesized.
...
=============================
HDL Synthesis Report

Macro Statistics
# RAMs                          : 1
  128-bit single-port RAM       : 1
=============================
...
```

Related Constraints

Related constraints are ram_extract and ram_style.

Single Port RAM with Asynchronous Read

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



X8976

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

VHDL

Following is the VHDL code for a single port RAM with asynchronous read.

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

entity raminfr is
  port (clk : in std_logic;
        we  : in std_logic;
        a   : in std_logic_vector(4 downto 0);
        di  : in std_logic_vector(3 downto 0);
```

```
        do  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
    end if;
  end process;
  do <= RAM(conv_integer(a));
end syn;
```

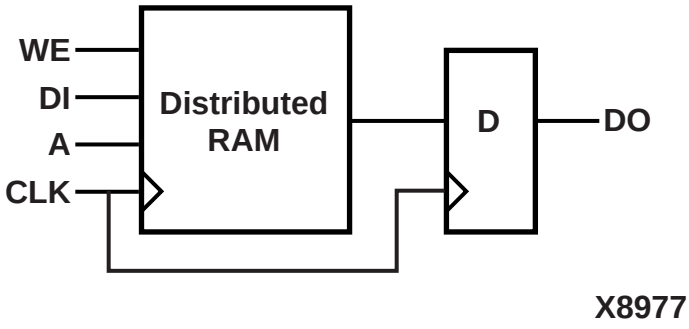
Verilog

Following is the Verilog code for a single port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);

input clk;
input we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];

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

Single Port RAM with "false" Synchronous Read

The following descriptions do not implement true synchronous read access as defined by the Virtex block RAM specification, where the read address is registered. They are *only mappable onto Distributed RAM* with an additional buffer on the data output, as shown below:



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

VHDL

Following is the VHDL code for a single port RAM with “false” synchronous read.

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

entity raminfr is
  port (clk : in std_logic;
        we  : in std_logic;
```

```
        a   : in std_logic_vector(4 downto 0);
        di  : in std_logic_vector(3 downto 0);
        do  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0)
    of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      do <= RAM(conv_integer(a));
    end if;
  end process;
end syn
```

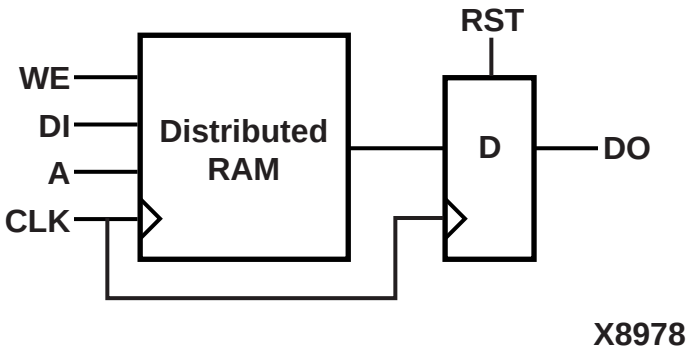
Verilog

Following is the Verilog code for a single port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);

input        clk;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [3:0] do;

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

```
endmodule
```

The following descriptions, featuring an additional reset of the RAM output, are also *only mappable onto Distributed RAM* with an additional resetable buffer on the data output as shown in the following figure:



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

VHDL

Following is the VHDL code.

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

entity raminfr is
  port (clk : in std_logic;
        we  : in std_logic;
```

```
        rst : in std_logic;
        a   : in std_logic_vector(4 downto 0);
        di  : in std_logic_vector(3 downto 0);
        do  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      if (rst = '1') then
        do <= (others => '0');
      else
        do <= RAM(conv_integer(a));
      end if;
    end if;
  end process;
end syn;
```

Verilog

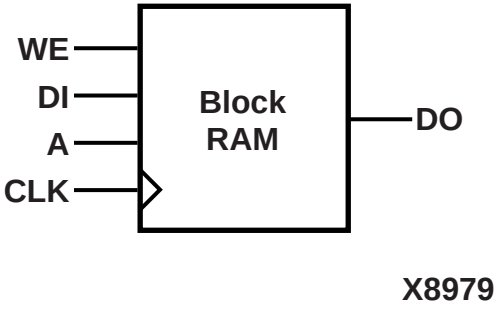
Following the Verilog code.

```
module raminfr (clk, we, rst, a, di, do);

input clk;
input we;
input rst;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [3:0] do;

  always @(posedge clk) begin
```

```
    if (we)
      ram[a] <= di;
    if (rst)
      do <= 4'b0;
    else
      do <= ram[a];
  end
endmodule
```

Single-Port RAM with Synchronous Read (Read Through)

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



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

VHDL

Following is the VHDL code for a single-port RAM with synchronous read (read through).

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

entity raminfr is
  port (clk : in std_logic;
        we  : in std_logic;
        a   : in std_logic_vector(4 downto 0);
        di  : in std_logic_vector(3 downto 0);
        do  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_a : std_logic_vector(4 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      read_a <= a;
    end if;
  end process;
  do <= RAM(conv_integer(read_a));
end syn;
```

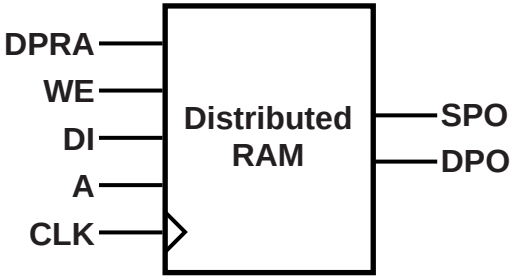
Verilog

Following is the Verilog code for a single-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, di, do);

input clk;
input we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
  end
  assign do = ram[read_a];
endmodule
```

Dual-port RAM with Asynchronous Read

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



X8980

IO pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
a	Write Address / Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL

Following is the VHDL code for a dual-port RAM with asynchronous read.

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

entity raminfr is
  port (clk  : in std_logic;
```

```
        we   : in std_logic;
        a    : in std_logic_vector(4 downto 0);
        dpra : in std_logic_vector(4 downto 0);
        di   : in std_logic_vector(3 downto 0);
        spo  : out std_logic_vector(3 downto 0);
        dpo  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
    of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
    end if;
  end process;
  spo <= RAM(conv_integer(a));
  dpo <= RAM(conv_integer(dpra));
 end syn;
```

Verilog

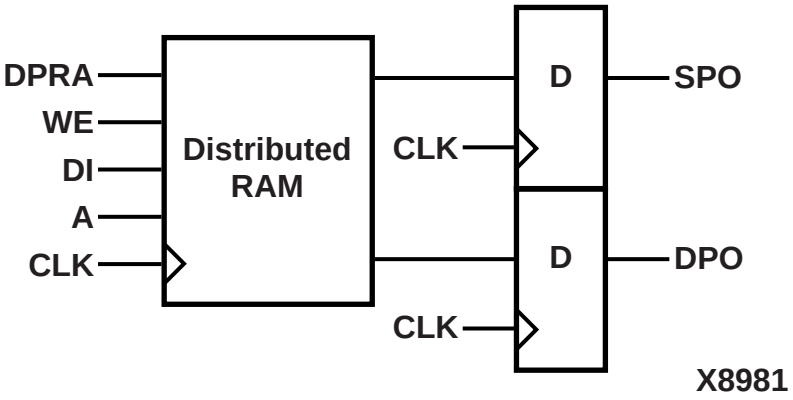
Following is the Verilog code for a dual-port RAM with asynchronous read.

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

input clk;
input we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];

  always @(posedge clk) begin
```

```
    if (we)
      ram[a] <= di;
  end
  assign spo = ram[a];
  assign dpo = ram[dpra];
endmodule
```

Dual-port RAM with False Synchronous Read

The following descriptions will be mapped onto Distributed RAM with additional registers on the data outputs. Please note that this template *does not* describe dual-port block RAM.



IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
a	Write Address / Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL

Following is the VHDL code for a dual-port RAM with false synchronous read.

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

entity raminfr is
  port (clk  : in std_logic;
        we   : in std_logic;
        a    : in std_logic_vector(4 downto 0);
        pra : in std_logic_vector(4 downto 0);
        di   : in std_logic_vector(3 downto 0);
        spo  : out std_logic_vector(3 downto 0);
        dpo  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      spo <= RAM(conv_integer(a));
      dpo <= RAM(conv_integer(dpra));
    end if;
  end process;
end syn;
```

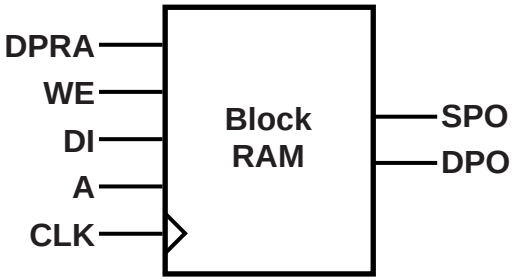
Verilog

Following is the Verilog code for a dual-port RAM with false synchronous read.

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

input clk;
input we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
reg    [3:0] spo;
reg    [3:0] dpo;

  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    spo = ram[a];
    dpo = ram[dpra];
  end
endmodule
```

Dual-port RAM with Synchronous Read (Read Through)

The following descriptions are directly mappable onto *Block RAM*, as shown in the following figure. (They may also be implemented with *Distributed RAM*.).



IO Pins	Description
clk	Positive-Edge Clock
we	Synchronous Write Enable (active High)
a	Write Address / Primary Read Address
dpra	Dual Read Address
di	Data Input
spo	Primary Output Port
dpo	Dual Output Port

VHDL

Following is the VHDL code for a dual-port RAM with synchronous read (read through).

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

entity raminfr is
  port (clk  : in std_logic;
        we   : in std_logic;
        a    : in std_logic_vector(4 downto 0);
        dpra : in std_logic_vector(4 downto 0);
        di   : in std_logic_vector(3 downto 0);
```

```
        spo  : out std_logic_vector(3 downto 0);
        dpo  : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_a : std_logic_vector(4 downto 0);
  signal read_dpra : std_logic_vector(4 downto 0);
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
      read_a <= a;
      read_dpra <= dpra;
    end if;
  end process;
  spo <= RAM(conv_integer(read_a));
  dpo <= RAM(conv_integer(read_dpra));
end syn;
```

Verilog

Following is the Verilog code for a dual-port RAM with synchronous read (read through).

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

input clk;
input we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
reg    [4:0] read_dpra;
```

```
  always @(posedge clk) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
    read_dpra <= dpra;
  end
  assign spo = ram[read_a];
  assign dpo = ram[read_dpra];
endmodule
```

Note The two RAM ports may be synchronized on distinct clocks, as shown in the following description. In this case, only a Block RAM implementation will be applicable.

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

VHDL

Following is the VHDL code.

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

entity raminfr is
  port (clk1 : in std_logic;
        clk2 : in std_logic;
        we   : in std_logic;
        add1 : in std_logic_vector(4 downto 0);
        add2 : in std_logic_vector(4 downto 0);
        di   : in std_logic_vector(3 downto 0);
```

```
        do1  : out std_logic_vector(3 downto 0);
        do2  : out std_logic_vector(3 downto 0));
end raminfr;

architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
  signal read_add1 : std_logic_vector(4 downto 0);
  signal read_add2 : std_logic_vector(4 downto 0);
begin
  process (clk1)
  begin
    if (clk1'event and clk1 = '1') then
      if (we = '1') then
        RAM(conv_integer(add1)) <= di;
      end if;
      read_add1 <= add1;
    end if;
  end process;
  do1 <= RAM(conv_integer(read_add1));

  process (clk2)
  begin
    if (clk2'event and clk2 = '1') then
      read_add2 <= add2;
    end if;
  end process;
  do2 <= RAM(conv_integer(read_add2));
end syn;
```

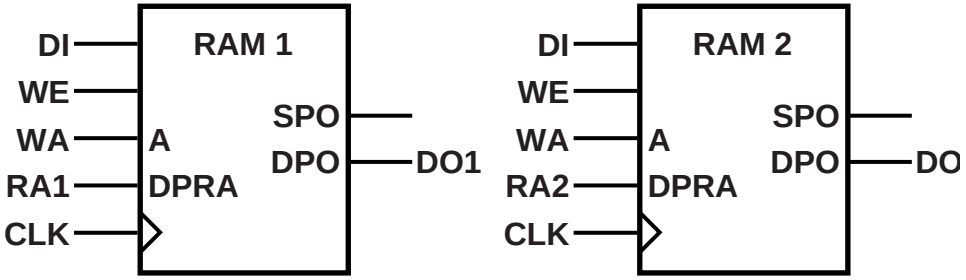
Verilog

Following is the Verilog code.

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

input clk1;
input clk2;
input we;
input  [4:0] add1;
input  [4:0] add2;
input  [3:0] di;
output [3:0] do1;
output [3:0] do2;
reg    [3:0] ram [31:0];
reg    [4:0] read_add1;
reg    [4:0] 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
```

Multiple-Port RAM Descriptions

XST can identify RAM descriptions with two or more read ports accessing the RAM contents at different addresses than the write address. However, there can only be one write port. The following descriptions will be implemented by replicating the RAM contents for each output port, as shown:



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

VHDL

Following is the VHDL code for a multiple-port RAM.

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

entity raminfr is
  port (clk : in std_logic;
        we  : in std_logic;
        wa  : in std_logic_vector(4 downto 0);
        ra1 : in std_logic_vector(4 downto 0);
        ra2 : in std_logic_vector(4 downto 0);
```

```
        di  : in std_logic_vector(3 downto 0);
        do1 : out std_logic_vector(3 downto 0);
        do2 : out std_logic_vector(3 downto 0));
end raminfr;
architecture syn of raminfr is
  type ram_type is array (31 downto 0)
        of std_logic_vector (3 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(wa)) <= di;
      end if;
    end if;
  end process;
  do1 <= RAM(conv_integer(ra1));
  do2 <= RAM(conv_integer(ra2));
end syn;
```

Verilog

Following is the Verilog code for a multiple-port RAM.

```
module raminfr
      (clk, we, wa, ra1, ra2, di, do1, do2);

input clk;
input we;
input  [4:0] wa;
input  [4:0] ra1;
input  [4:0] ra2;
input  [3:0] di;
output [3:0] do1;
output [3:0] do2;
reg    [3:0] ram [31:0];

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

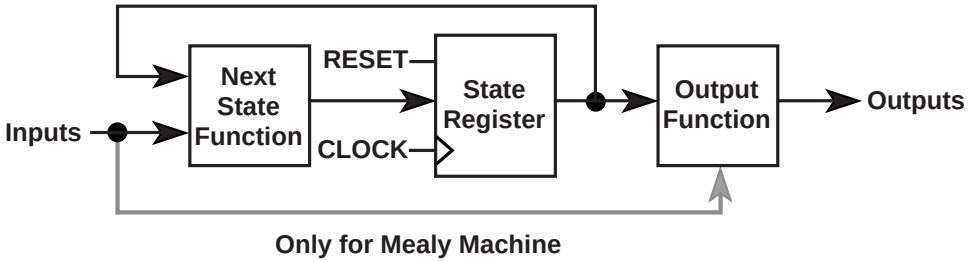
```
  assign do1 = ram[ra1];
  assign do2 = ram[ra2];
endmodule
```

State Machines

XST proposes a large set of templates to describe Finite State Machines (FSMs). By default, XST tries to recognize FSMs from VHDL / Verilog code and apply several state encoding techniques (it can re-encode the user's initial encoding) to get better performance or less area. However, you can disable FSM extraction using a "fsm_extract" design constraint.

Please note that XST can handle only synchronous state machines.

There are many ways to describe FSMs. A traditional FSM representations incorporates Mealy and Moore machines:



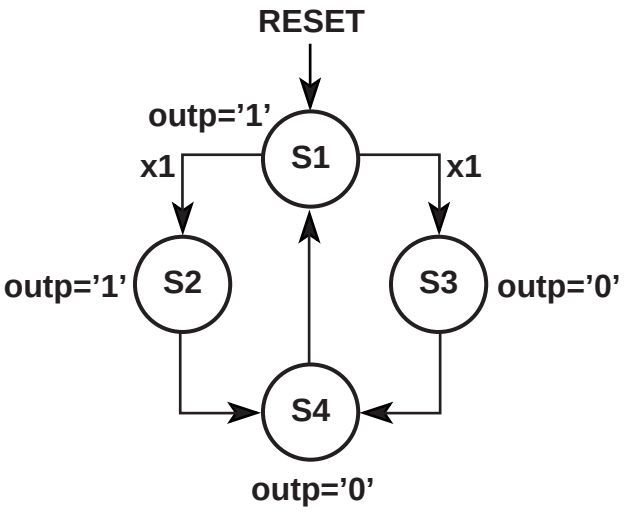
X8993

For HDL, process (VHDL) and always blocks (Verilog) are the most suitable ways for describing FSMs. (For description convenience Xilinx uses "process" to refer to both: VHDL processes and Verilog always blocks).

You may have several processes (1, 2 or 3) in your description, depending upon how you consider and decompose the different parts of the preceding model. Following is an example of the Moore Machine with Asynchronous Reset "reset".

- 4 states: s1, s2, s3, s4
- 5 transitions
- 1 input: "x1"
- 1 output: "outp"

This model is represented by the following bubble diagram:



Related Constraints

Related constraints are:

- fsm_extract
- fsm_encoding
- fsm_fftype
- enum_encoding

FSM: 1 Process

Please note, in this example output signal "outp" is a *register*.

VHDL

Following is the VHDL code for an FSM with a single process.

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

entity fsm is
  port ( clk, reset, x1 : IN std_logic;
                  outp : OUT std_logic);
end entity;
architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type ;
begin
  process (clk,reset)
  begin
    if (reset ='1') then
      state <=s1; outp<='1';
    elsif (clk='1' and clk'Event) then
      case state is
        when s1 => if x1='1' then state <= s2;
                    else          state <= s3;
                    end if;
                    outp <= '1';
        when s2 => state <= s4; outp <= '1';
        when s3 => state <= s4; outp <= '0';
        when s4 => state <= s1; outp <= '0';
      end case;
    end if;
  end process;
end beh1;
```

Verilog

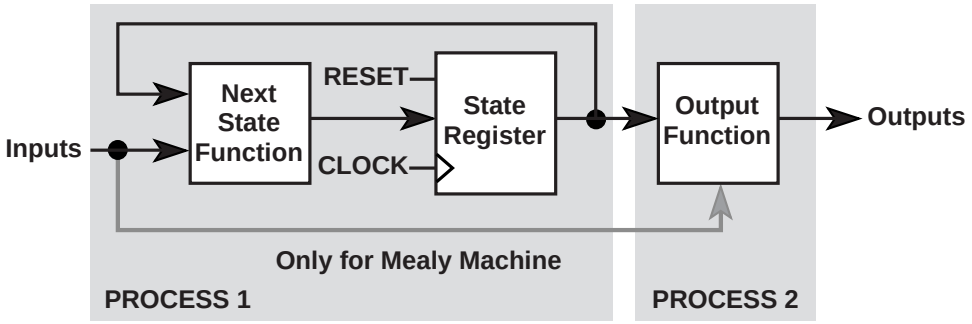
Following is the Verilog code for an FSM with a single process.

```
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;

reg [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;

  always@(posedge clk or posedge reset)
  begin
    if (reset)
      begin
        state = s1; outp = 1'b1;
      end
    else
      begin
        case (state)
          s1: begin
                if (x1==1'b1) state = s2;
                else          state = s3;
                outp = 1'b1;
              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
```

FSM: 2 Processes

To eliminate a register from the "outputs", you can remove all assignments “outp <=…” from the Clock synchronization section.

This can be done by introducing two processes as shown in the following figure.



VHDL

Following is VHDL code for an FSM with two processes.

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

entity fsm is
  port ( clk, reset, x1 : IN std_logic;
                  outp : OUT std_logic);
end entity;
architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state: state_type ;
begin
  process1: process (clk,reset)
  begin
    if (reset =’1’) then state <=s1;
    elsif (clk=’1’ and clk’Event) then
      case state is
        when s1 => if x1=’1’ then state <= s2;
                   else           state <= s3;
```

```
                    end if;
          when s2 => state <= s4;
          when s3 => state <= s4;
          when s4 => state <= s1;
        end case;
      end if;
    end process process1;

process2 : process (state)
    begin
      case state is
        when s1 => outp <= '1';
        when s2 => outp <= '1';
        when s3 => outp <= '0';
        when s4 => outp <= '0';
      end case;
end process process2;
end beh1;
```

Verilog

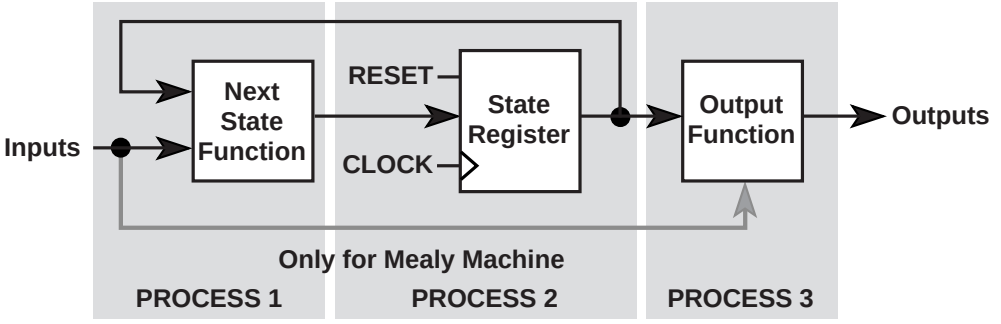
Following is the Verilog code for an FSM with two processes.

```
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;

reg [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;

  always @(posedge clk or posedge reset)
  begin
    if (reset)
      state = s1;
    else
      begin
        case (state)
          s1: if (x1==1'b1) state = s2;
              else          state = s3;
          s2: state = s4;
```

```
            s3: state = s4;
            s4: state = s1;
          endcase
        end
    end

    always @(state)
    begin
          case (state)
            s1: outp = 1'b1;
            s2: outp = 1'b1;
            s3: outp = 1'b0;
            s4: outp = 1'b0;
          endcase
    end
endmodule
```

FSM: 3 Processes

You can also separate the NEXT State function from the State Register:



Separating the NEXT State function from the State Register provides the following description:

VHDL

Following is the VHDL code for an FSM with three processes.

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;

entity fsm is
  port ( clk, reset, x1 : IN std_logic;
                    outp : OUT std_logic);
end entity;
architecture beh1 of fsm is
  type state_type is (s1,s2,s3,s4);
  signal state, next_state: state_type ;
begin
  process1: process (clk,reset)
  begin
    if (reset ='1') then
      state <=s1;
    elsif (clk='1' and clk'Event) then
      state <= next_state;
    end if;
  end process process1;

  process2 : process (state, x1)
  begin
    case state is
          when s1 => if x1='1' then
                       next_state <= s2;
                     else
                       next_state <= s3;
                     end if;
          when s2 => next_state <= s4;
          when s3 => next_state <= s4;
          when s4 => next_state <= s1;
    end case;
end process process2;

process3 : process (state)
  begin
      case state is
          when s1 => outp <= '1';
          when s2 => outp <= '1';
          when s3 => outp <= '0';
          when s4 => outp <= '0';
      end case;
```

```
end process process3;
end beh1;
```

Verilog

Following is the Verilog code for an FSM with three processes.

```
module fsm (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;

reg [1:0] state;
reg [1:0] next_state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;

  always @(posedge clk or posedge reset)
  begin
    if (reset)  state = s1;
    else        state = next_state;
  end

  always @(state or x1)
  begin
        case (state)
          s1: if (x1==1'b1) next_state = s2;
              else          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
```

State Registers

State Registers must to be initialized with an asynchronous or synchronous signal. XST does not support FSM without initialization signals. Please refer to the “Registers” section of this chapter for templates on how to write Asynchronous and Synchronous initialization signals.

In VHDL the type of a state register can be a different type: integer, bit_vector, std_logic_vector, for example. But it is common and convenient to define an enumerated type containing all possible state values and to declare your state register with that type.

In Verilog, the type of state register can be an integer or a set of defined parameters. In the following Verilog examples the state assignments could have been made like this:

```
parameter [3:0]
  s1 = 4'b0001,
  s2 = 4'b0010,
  s3 = 4'b0100,
  s4 = 4'b1000;
reg [3:0] state;
```

These parameters can be modified to represent different state encoding schemes.

Next State Equations

Next state equations can be described directly in the sequential process or in a distinct combinational process. The simplest template is based on a case statement. If using a separate combinational process, its sensitivity list should contain the state signal and all FSM inputs.

FSM Outputs

Non-registered outputs are described either in the combinational process or concurrent assignments. Registered outputs must be assigned within the sequential process.

FSM Inputs

Registered inputs are described using internal signals, which are assigned in the sequential process.

State Encoding Techniques

XST supports the following state encoding techniques.

- Auto
- One-Hot
- Gray
- Compact
- Johnson
- Sequential
- User

Auto

In this mode XST tries to select the best suited encoding algorithm for each FSM.

One-Hot

One-hot encoding is the default encoding scheme. Its principle is to associate one code bit and also one flip-flop to each state. At a given clock cycle during operation, one and only state variable is asserted. Only two state variables toggle during a transition between two states. One-hot encoding is very appropriate with most FPGA targets where a large number of flip-flops are available. It is also a good alternative when trying to optimize speed or to reduce power dissipation.

Gray

Gray encoding guarantees that only one state variable switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching. In addition, this coding technique minimizes hazards and glitches. Very good results can be obtained when implementing the state register with T flip-flops.

Compact

Compact encoding, consists of minimizing the number of state variables and flip-flops. This technique is based on hypercube immersion. Compact encoding is appropriate when trying to optimize area.

Johnson

Like Gray, Johnson encoding shows benefits with state machines containing long paths with no branching.

Sequential

Sequential encoding consists of identifying long paths and applying successive radix two codes to the states on these paths. Next state equations are minimized.

User

In this mode XST uses original encoding, specified in the HDL file. For example if you use enumerated types for a state register, then in addition you can use the "enum_encoding" constraint to assign a specific binary value to each state. Please refer to the “Design Constraints” chapter for more details.

Log File

The XST log file reports the full information of recognized FSM during the macro recognition step. Moreover, if you allow XST to choose the best encoding algorithm for your FSMs, it will report the one it chose for each FSM.

```
...
Synthesizing Unit <fsm>.
    Extracting finite state machine <FSM_0> for signal <state>.
-----------------------------------------------------------------
-
| States            | 4                                         |
| Transitions       | 5                                         |
| Inputs            | 1                                         |
| Outputs           | 1                                         |
| Reset type        | asynchronous                              |
| Encoding          | automatic                                 |
| State register    | D  flip-flops                             |
-----------------------------------------------------------------
-
    Summary:
  inferred   1 Finite State Machine(s).
Unit <fsm> synthesized.
...
 ============================
HDL Synthesis Report

Macro Statistics
# FSMs                          : 1
==============================
...
Optimizing FSM <FSM_0> with One-Hot encoding and D flip-flops.
...
```

Black Box Support

Your design may contain EDIF netlists generated by synthesis tools, schematic editors, or any other design entry mechanism. These modules must be instantiated in your code to be connected to the rest of your design. This can be achieved in XST using black box instantiation in the VHDL / Verilog code. The netlist will be propagated to the final top-level netlist without being processed by XST. Moreover, XST allows you to attach specific constraints to these black box instantiations, which will be passed to the EDIF netlist or NCF file.

Log File

From the flow point of view, the recognition of black boxes in XST is done before macro inference process. Therefore the LOG file differs from the one generated for other macros.

```
...
Analyzing Entity <shift>(Architecture <archi>).
WARNING    :    (VHDL_0103).c:\jm\des.vhd    (Line
24).Generating   a   Black   Box   for   component
<my_block>.
Entity <shift> analyzed.
Unit <shift> generated.
...
```

Related Constraints

XST has a "BOX_TYPE" constraint which can be applied to black boxes. However, it was introduced essentially for the Virtex Primitive instantiation in XST. Please read the “Virtex Primitive Support” section in the “Design Constraints” chapter before using this constraint.

VHDL

Following is the VHDL code for a black box.

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

entity black_b is
  port(DI_1, DI_2 : in  std_logic;
       DOUT       : out std_logic);
end black_b;

architecture archi of black_b is
  component my_block
    port (
      I1 : in std_logic;
      I2 : in std_logic;
      O : out std_logic);
  end component;

begin
  inst: my_block port map (I1=>DI_1, I2=>DI_2, O=>DOUT);
end archi;
```

Verilog

Following is the Verilog code for a black box.

```
module my_block (in1, in2, dout);
  input in1, in2;
  output dout;
endmodule

module black_b (DI_1, DI_2, DOUT);
  input DI_1, DI_2;
  output DOUT;
  my_block inst (.in1(DI_1), .in2(DI_2), .dout(DOUT));
endmodule
```

Note Please refer to the VHDL / Verilog language reference manuals for more information on component instantiation.

Chapter 3

FPGA Optimization

This chapter contains the following sections:

- "Introduction"
- "Virtex Specific Options"
- "Timing Constraints"
- "Macro Generation"
- "Log File Analysis"
- "NCF Generation"
- "Virtex Primitive Support"

Introduction

XST performs the following steps during FPGA synthesis and optimization:

- Mapping and optimization on a entity / module by entity / module basis
- Global optimization on the complete design

The output of this process is one or several EDIF files and a single .ncf file.

This chapter describes the following:

- Constraints that can be applied to tune this synthesis and optimization process
- Macro generation step
- Information in the log file

- Timing model used during the synthesis and optimization process
- Constraints available for timing driven synthesis
- Information on the generated .ncf file
- Information on the support for primitives

Virtex Specific Options

XST support a set of options that allows the tuning of the synthesis process according to the user constraints. This section lists the options that relate to the FPGA-specific optimization of the synthesis process. For details about each option, see the “FPGA Options” section of the “Design Constraints” chapter

Following is a list of FPGA options.

- “Mux Style”
- “RAM Style”
- “Speed Grade for Timing Analysis”
- “Max Fanout”
- “Add Generic Clock Buffer”
- “Maximum Number of Clock Buffers Created by XST”
- “Sig_isclock”
- “Add IO Buffers”
- “Register Duplication”
- “Keep Hierarchy”
- “Incremental Synthesis”
- “Resynthesis”
- “Global Optimization Goal”

Timing Constraints

XST supports both global and specific timing constraints. Basically, global timing constraints are applied to a domain (inpad_to_outpad or offset_in_before for instance), specific timing constraints are either applied to a domain or to a given specific signal and are associated with a value (expressed in ns or MHz).

Definitions

The four possible domains are illustrated in the following schematic.

ALLCLOCKNETS (register to register) identifies all paths from any clock signal to any clock signal.

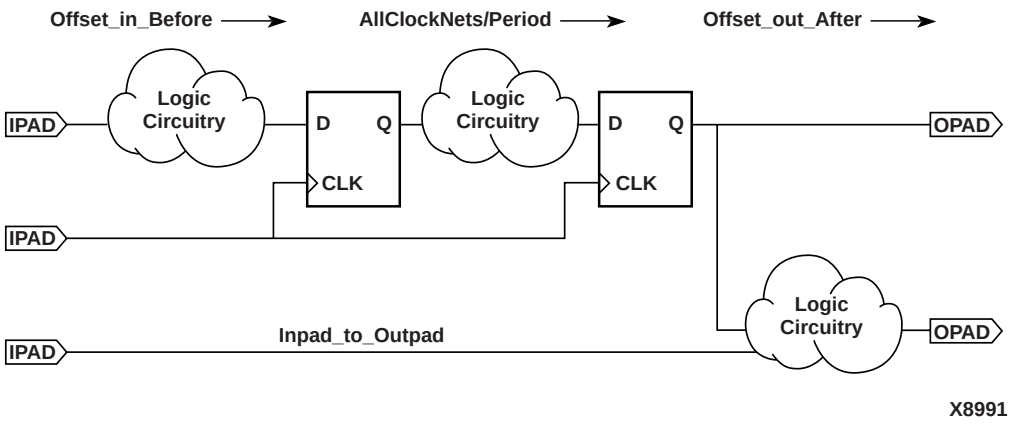
PERIOD identifies all paths between all sequential element controlled by the given signal name.

OFFSET_IN_BEFORE (inpad to register) identifies all paths from all primary input ports to either all sequential elements or the sequential elements driven by the given clock signal name.

OFFSET_OUT_AFTER (register to outpad) is similar to the previous constraint but sets the constraint from the sequential elements to all primary output ports.

INPAD_TO_OUTPAD (inpad to outpad) sets a maximum combinational path constraint.

MAX_DELAY identifies the longest path of the complete design.



The following table summarizes global and specific constraints.

Constraints	Applies to	Global/Specific
ALLCLOCKNETS	top entity / module	Global or Specific
PERIOD	specific clock signal	Specific
OFFSET_IN_BEFORE	top entity / module or specific clock signal	Global or Specific
OFFSET_OUT_AFTER	top entity / module or specific clock signal	Global or Specific
INPAD_TO_OUTPAD	top entity / module	Global or Specific
MAX_DELAY	top entity / module	Global

The main differences between a global and a specific constraint are as follows.

- Specific timing constraints specify both a domain *and* a value.
- For specific constraints, XST will stop optimizing the design, when the value specified in the specific constraint is met. For global constraints, XST will continue optimization as long as there is improvement.
- Specific timing constraints can be set only using the constraint file or by setting attributes directly in your HDL file.
- Global constraints can be set only from the Process Properties dialog box in the Project Navigator or from the on line options.

Examples

These examples illustrate how to set a constraint using the constraint file.

```
attribute ALLCLOCKNETS of top : entity is "10ns";
attribute PERIOD of clk : signal is "100MHz" ;
attribute OFFSET_IN_BEFORE of clk1 : signal is
"10.4ns";
```

or

```
attribute OFFSET_IN_BEFORE of top : entity is "14ns";
attribute OFFSET_OUT_AFTER of TOP : entity is "5ns";
```

or

```
attribute OFFSET_OUT_AFTER of clk2 : signal is
"2.5ns";

attribute INPAD_TO_OUTPAD of top : entity is "15ns";
```

Note See the “Design Constraints” chapter for more details on how to set constraints in your HDL description.

Timing Model

The timing model used by XST for timing analysis takes into account both logical delays and net delays. These delays are highly dependent on the speed grade that can be specified to XST. These delays are also dependent on the selected technology (Virtex, VirtexE, ...). Logical delays data are identical to the delays reported by Trce (Timing analyzer after Place and Route). The Net delay model is estimated based on the fanout load.

Priority

Constraints are processed in the following order:

- Specific constraints on signals (will contain a value)
- Specific constraints on top module (will contain a value)
- Global constraints on top module (no value)

For example:

- PERIOD will have priority over ALLCLOCKNETS
- Constraints on two different domains or two different signals have the same priority (that is, INPAD_TO_OUTPAD can be applied with PERIOD; PERIOD clk1 can be applied with PERIOD clk2).

Limitations

If multiple specific constraints are set, XST will stop optimization either when all constraints are satisfied or when optimization does not succeed in satisfying the current most critical constraint.

Macro Generation

The Virtex Macro Generator module provides the XST HDL Flow with a catalog of functions. These functions are identified by the inference engine from the HDL description; their characteristics are handed to the Macro Generator for optimal implementation. The set of inferred functions ranges in complexity from simple arithmetic operators such as adders, accumulators, counters, and multiplexers to more complex building blocks such as multipliers, shift registers and memories.

Inferred functions are optimized to deliver the highest levels of performance and efficiency for Virtex architectures and then integrated into the rest of the design. In addition, the generated functions are optimized through their borders depending on the design context.

This section categorizes, by function, all available macros and briefly describes technology resources used in the building and optimization phase.

Macro Generation can be controlled through attributes. These attributes are listed in each subsection. For general information on attributes see the “Design Constraints” chapter.

Arithmetic Functions

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 are implemented using LUTs and the dedicated carry-XORs (XORCY). In addition, XST benefits from a dedicated carry-ANDs (MULTAND) resource for high-speed multiplier implementation.

Loadable Functions

For Loadable functions XST provides the following elements:

- Loadable Up, Down and Up / Down Binary Counters
- Loadable Up, Down and Up / Down Accumulators

XST is able to provide synchronously loadable, cascadable binary counters and accumulators inferred in the HDL flow. Fast carry logic is used to cascade the different stages of the macros. Synchronous loading and count functions are packed in the same LUT primitive for optimal implementation.

For Up / Down counters and accumulators, XST uses the dedicated carry-ANDs to improve the performance.

Multiplexers

For multiplexers the Macro Generator provides the following two architectures:

- MUXF5 / MUXF6 based multiplexers
- Dedicated Carry-MUXs based multiplexers

MUXF5 / MUXF6 based multiplexers are generated by using the optimal tree structure of MUXF5 and MUXF6 primitives which allows compact implementation of large inferred multiplexers. For example, XST can implement an 8:1 multiplexer in a single CLB. In some cases dedicated carry-MUXs are generated; these can provide more efficient implementations especially for very large multiplexers.

In order to have a better control on 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 will be implemented on a MUXF5 / MUXF6 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" / "force" can be used to disable / force the inference of the multiplexer.

Priority Encoder

The described if/elsif structure described in the “Priority Encoders” section of the “HDL Coding Techniques” chapter will be implemented with a 1-of-n priority encoder.

XST uses the MUXCY primitive to chain the conditions of the priority encoder which results in its high-speed implementation.

You can enable/disable priority encoder inference using the priority_extract property.

Generally, XST does not infer and therefore generate a large number of priority encoders. Therefore, Xilinx recommends that you use the "priority-extract = force" constraint.

Decoder

A decoder is a multiplexer whose inputs are all constant with distinct one-hot (or one-cold) coded values. An n-bit or 1-of-m decoder is mainly characterized by an m-bit data output and an n-bit selection input, such that n**(2-1) < m <= n**2.

Once XST has inferred the decoder, the implementation uses the MUXF5 or MUXCY primitive depending on the size of the decoder.

You can enable/disable decoder inference using the decoder_extract property.

Shift Register

Two types of shift register are built by XST:

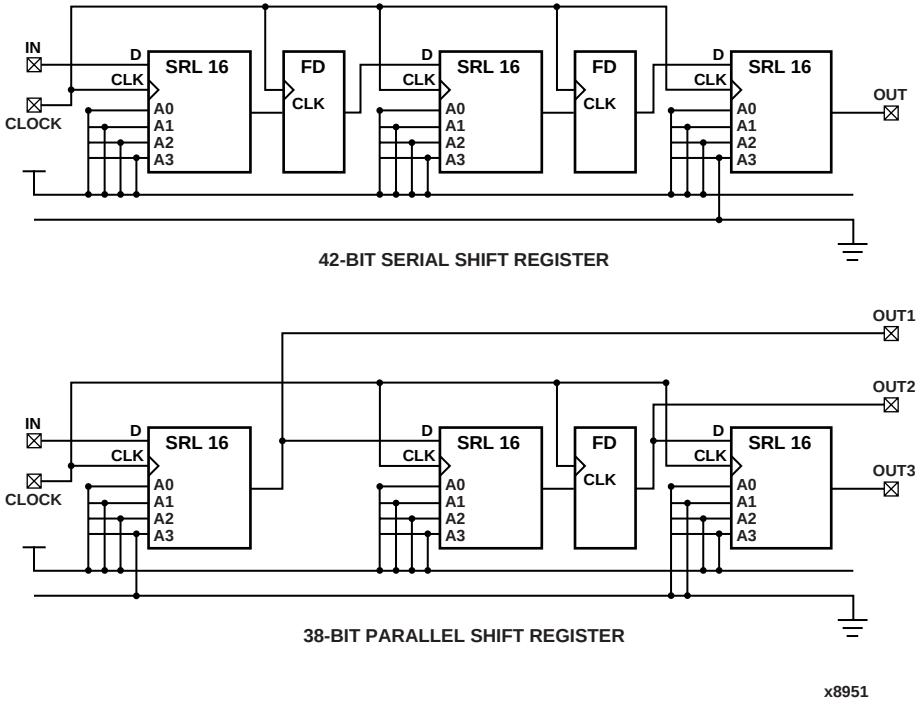
- Serial shift register with single output
- Parallel shift register with multiple outputs

The length of the shift register can vary from 1 bit to 16 bits as determined from the following formula:

Width = (8*A3)+(4*A2)+(2*A1)+A0+1

If A3, A2, A1 and A0 are all zeros (0000), the shift register is one-bit long. If they are all ones (1111), it is 16-bits long.

For serial shift register SRL16, flip flops are chained to the appropriate width. For example, the serial shift register (Width=42) will be implemented according to the following scheme.



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. For example the parallel 38-bit shift register with 3 outputs in the positions 7, 24 and 37 will be implemented according to the scheme shown in the preceding figure.

You can enable / disable shift register inference using the shreg_extract property.

RAMs

Two types of RAM are available in the inference and generation stages: Distributed and Block RAMs.

- If the RAM is asynchronous READ, Distributed RAM is inferred and generated
- If the RAM is synchronous READ, Block RAM is inferred. In this case, XST can implement Block RAM or Distributed RAM. The default is Block RAM

In the case of Distributed RAM, XST uses:

- RAM16X1S and RAM32X1S primitives for Single-Port Synchronous Distributed RAM
- RAM16X1D primitives for Dual-Port Synchronous Distributed RAM

In the case of Block RAM XST uses:

- RAMB4_Sn primitives for Single-Port Synchronous Block RAM
- RAMB4_Sn_Sn primitives for Dual-Port Synchronous Block RAM

In order to have a better control on 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 attribute ram_style specifies that an inferred RAM be generated using

- Block RAM if the value is "block"
- Distributed RAM if the value is "distributed"

You can apply the ram_style attribute either to a signal that defines the RAM or the instance name of the RAM. This attribute can also be global.

If the RAM resources are limited, XST can generate additional RAMs using registers using the attribute ram_extract with the value NO.

Log File Analysis

The XST log file related to FPGA optimization contains the following sections:

- Design optimization
- Resource usage report
- Timing report

Design Optimization

During design optimization, XST reports the following:

- Potential removal of equivalent flip-flops.

  Two flip-flops (latches) are equivalent when they have the same data and control pins

- Register replication

  Register replication is performed either for timing performance improvement or for satisfying maximum fanout constraints. Register replication can be turned off using the register_duplication constraint.

Following is a portion of the log file.

```
Starting low level synthesis...
Optimizing unit <down4cnt> ...
Optimizing unit <doc_readwrite> ...
 ...
Optimizing unit <doc> ...
Building and optimizing final netlist ...
Register doc_readwrite_state_D2 equivalent to
   doc_readwrite_cnt_ld has been removed
Register I_cci_i2c_wr_l equivalent to wr_l has been
   removed
Register doc_reset_I_reset_out has been replicated
   2 time(s)
Register wr_l has been replicated 2 time(s)
```

Resource Usage

In the Final Report, the Cell Usage section reports the count of all the primitives used in the design. These primitives are classified in 8 groups:

- BELS

  This group contains all the logical cells that are basic elements of the Virtex technology, for example, LUTs, MUXCY, MUXF5, MUXF6.

- Flip-flops and Latches

  This group contains all the flip-flops and latches that are primitives of the Virtex technology, for example, FDR, FDRE, LD.

- RAMS

  This group contains all the RAMs.

- SHIFTERS

  This group contains all the shift registers that use the Virtex primitives. Namely SRL16, SRL16_1, SRL16E, SRL16E_1.

- Tri-States

  This group contains all the tri-state primitives, namely the BUFT.

- Clock Buffers

  This group contains all the clock buffers, namely BUFG, BUFGP, BUFGDLL.

- IO Buffers

  This group contains all the standard I/O buffers, except the clock buffer, namely IBUF, OBUF, IOBUF, OBUFT, IBUF_GTL ...

- LOGICAL

  This group contains all the logical cells primitives that are not basic elements, namely AND2, OR2, ...

- OTHER

  This group contains all the cells that have not been classified in the previous groups.

The following section is an example of an XST report for cell usage:

```
=========================================================
...
Cell Usage :
# IO Buffers : 24
# OBUF : 8
# IBUF : 16
# BELS : 70
 # LUT4 : 33
# LUT3 : 3
 # LUT2 : 34
=========================================================
```

Timing Report

At the end of the synthesis, XST reports the timing information for the design. The report shows the information for all four possible domains of a netlist: "register to register", "input to register", "register to outpad" and "inpad to outpad".

The following is an example of a timing report section in the XST log:

```
TIMING REPORT
NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
 FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
GENERATED AFTER PLACE-and-ROUTE

Timing Summary:
----------
Speed Grade: -6
Minimum period: 7.523ns (Maximum Frequency: 132.926MHz)
Minimum input arrival time before clock: 8.945ns
Maximum output required time before clock: 14.220ns
Maximum combinational path delay: 10.899ns

Timing Detail:
All values displayed in nanoseconds (ns)
Path from Clock ’sysclk’ rising to Clock ’sysclk’ rising : 7.523ns
(Slack: -7.523ns)
```

```
                                  Gate     Net
Cell:in->out          fanout  Delay   Delay    Logical Name
----------------------------------------     ------------
FDC:C->Q               15   1.372   2.970    I_state_2
 begin scope: 'block1'
LUT3:I1->O              1   0.738   1.265    LUT_54
 end scope: 'block1'
LUT3:I0->O              1   0.738   0.000    I_next_state_2
FDC:D                       0.440            I_state_2
----------------------------------------
Total                                 7.523ns
```

Timing Summary

The Timing Summary section gives a summary of the timing paths for all 4 domains:

The path from any clock to any clock in the design:

```
Minimum period: 7.523ns (Maximum Frequency:
132.926MHz)
```

The maximum path from all primary inputs to the sequential elements:

```
Minimum input arrival time before clock: 8.945ns
```

The maximum path from the sequential elements to all primary outputs:

```
Maximum output required time before clock: 14.220ns
```

The maximum path from inputs to outputs:

```
Maximum combinational path delay: 10.899ns
```

If there is no path in the domain concerned "No path found" is then printed instead of the value.

Timing Detail

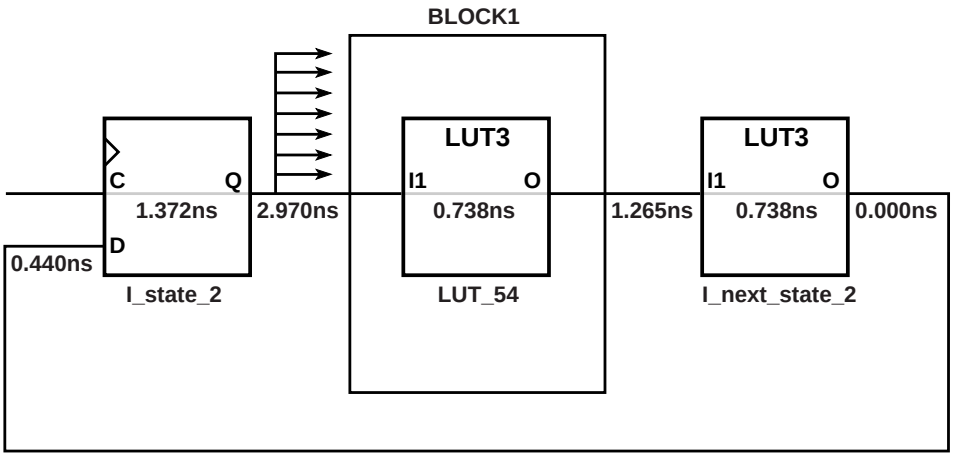
The Timing Detail section describes the most critical path in detail for each region:

The start point and end point of the path, the maximum delay of this path, and the slack. The start and end points can be: **Clock** (with the phase: rising / falling) or **Port**:

```
Path from Clock 'sysclk' rising to Clock 'sysclk'
rising : 7.523ns (Slack: -7.523ns)
```

The detailed path shows the cell type, the input and output of this gate, the fanout at the output, the gate delay, the net delay estimated and the name of the instance. When entering a hierarchical block, **`begin scope`** is printed, and similarly **`end scope`** is also printed when exiting a block.

The preceding report corresponds to the following schematic:



NCF Generation

Besides EDIF files, XST generates a .ncf file that contains all implementation constraints generated from HDL attributes (LOC, ...).

For example:

If the Verilog source file contains the following constraints:

```
// synthesis attribute HU_SET of u10 is MY_SET
// synthesis attribute LOC of in4 is L8
```

then the .ncf file will contain:

```
INST u10 HU_SET=MY_SET;
```

```
NET in4 LOC=L8;
```

KEEP properties generated by the buffer insertion process (for maximum fanout control or for optimization purpose, these optimization can be turned off using the maximum_fanout option), for example:

```
NET IBUF_dclk_1 keep;
```

Virtex Primitive Support

XST allows you to instantiate Virtex primitives directly in your VHDL/Verilog code. Virtex primitives and macros such as MUXCY_L, LUT4_L, CLKDLL, RAMB4_S1_S16, IBUFG_PCI33_5, and NAND3b2 can be manually inserted in your HDL design through instantiation. These primitives are not optimized by XST and will be available in the final EDIF file(s). Timing information is available for most of the primitives, allowing XST to perform efficient timing driven optimization.

Some of these primitives can be generated through attributes:

- clock_buffer will force the use of BUFGDLL, IBUFG or BUFGP
- iostandard can be used to assign an I/O standard to an I/O primitive, for example:

```
// synthesis attribute IOSTANDARD of in1 is
PCI33_5
```

will force generation of a _PCI33_5 IO (IBUF_PCI33_5, ...)

The primitive support is based on the notion of the Black Box. Refer to the “Black Box Support” section of the “HDL Coding Techniques” chapter for the basics of the black box support.

However, 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 Virtex Primitive. So, in order to avoid the confusion about how XST will interpret this module you have to use or not use a special constraint, called "BOX_TYPE". The only possible value for BOX_TYPE is “black_box”. This attribute must be attached to the component declaration of MUXF5.

In the case this attribute

- is attached to the MUXF5. XST will try interpret this module as a Virtex Primitive. In the case it is
  - *true*, XST will use its parameters, for instance in critical path estimation
  - *false*, XST will process it as a regular black box
- is not attached to the MUXF5. Then XST will process this block as a Black Box.

In order to simplify the instantiation process, XST comes with VHDL and Verilog Virtex libraries. These libraries contain the complete set of Virtex Primitives declarations with an attached "BOX_TYPE" constraint to each component. If you use

- VHDL, then you have to declare library "unisim" with its package "vcomponents" in your source code.

  ```
  library unisim;

  use unisim.vcomponents.all;
  ```

  The source code of this package can be found in the "vhdl\src\unisims_vcomp.vhd" file of the XST installation.
- Verilog, then you have to include a library file "unisim_comp.v" in your source code. This file can be found in the "verilog\src\iSE" directory of the XST installation.

  ```
  ‘include "c:\xst\verilog\src\iSE\unisim_comp.v"
  ```

Some primitives, like LUT1, allow you to use INIT during instantiation. In the VHDL case it is implemented via generic: code;

VHDL

Following is the VHDL code.

```
----- Component LUT1 -----
component LUT1
-- synopsys translate_off
   generic(
      TimingChecksOn: Boolean := DefaultTimingChecksOn;
      InstancePath: STRING := "*";
      Xon: Boolean := DefaultXon;
      MsgOn: Boolean := DefaultMsgOn;
      tpd_I0_O        :  VitalDelayType01 := (0.100 ns, 0.100 ns);
      tipd_I0         :  VitalDelayType01 := (0.000 ns, 0.000 ns);
      INIT            :  bit_vector);
--    INIT            :  bit_vector := X"0");

-- synopsys translate_on
   port(
      O               :  out   STD_ULOGIC;
      I0              :  in    STD_ULOGIC);
end component;
attribute BOX_TYPE of LUT1 : component is "BLACK_BOX";
```

Verilog

Following is the Verilog code.

```
module LUT1 (0, IO);
  input I0;
  output O;
endmodule
// synthesis attribute BOX_TYPE of LUT1 is "BLACK_BOX"
```

Log File

XST does not issue any message concerning instantiation of the Virtex primitives during HDL synthesis. Please note, that in the case you instantiate your own black box and you attach the "BOX_TYPE" attribute to the component, then XST will not issue a message like this:

```
...
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.
...
```

Instantiation of MUXF5

In this example, the component is directly declared in the HDL design file.

VHDL

Following is the VHDL code for instantiation of MUXF5.

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

entity black_b is
  port(DI_1, DI_2, SI : in  std_logic;
       DOUT           : out std_logic);
end black_b;

architecture archi of black_b is
  component MUXF5
    port (
      O  : out STD_ULOGIC;
      IO : in  STD_ULOGIC;
      I1 : in  STD_ULOGIC;
      S  : in  STD_ULOGIC);
  end component;
  attribute BOX_TYPE: string;
```

```
  attribute BOX_TYPE of MUXF5: component is "BLACK_BOX";
begin
  inst: MUXF5 port map (I0=>DI_1, I1=>DI_2, S=>SI, O=>DOUT);
end archi;
```

Verilog

Following is the Verilog code for instantiation of a MUXF5.

```
module MUXF5 (O, I0, I1, S);
  output 0;
  input IO, I1, S;
endmodule
// synthesis attribute BOX_TYPE of MUXF5 is "BLACK_BOX"

module black_b (DI_1, DI_2, SI, DOUT);
  input DI_1, DI_2, SI;
  output DOUT;
  MUXF5 inst (.I0(DI_1), .I1(DI_2), .S(SI), .O(DOUT));
endmodule
```

Instantiation of MUXF5 with XST Virtex Libraries

Following are VHDL and Verilog examples of an instantiation of a MUXF5 with XST Virtex Libraries.

VHDL

Following is the VHDL code.

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

entity black_b is
  port(DI_1, DI_2, SI : in  std_logic;
       DOUT       : out std_logic);
end black_b;

architecture archi of black_b is
  begin
  inst: MUXF5 port map (I0=>DI_1, I1=>DI_2, S=>SI, O=>DOUT);
```

```
end archi;
```

Verilog

Following is the Verilog code.

```
‘include "c:\xst\verilog\src\iSE\unisim_comp.v"

module black_b (DI_1, DI_2, SI, DOUT);
  input DI_1, DI_2, SI;
  output DOUT;

  MUXF5 inst (.I0(DI_1), .I1(DI_2), .S(SI), .O(DOUT));
endmodule
```

Related Constraints

Related constraints are BOX_TYPE and different PAR constraints that can be passed from HDL to EDIF / NCF without processing.

Chapter 4

CPLD Optimization

This chapter contains the following sections.

- "CPLD Synthesis Options"
- "Implementation Details for Macro Generation"
- "Log File Analysis"
- "NCF File"
- "Improving Results"

CPLD Synthesis Options

This section describes the CPLD supported families and the specific options.

Introduction

XST performs device specific synthesis for XC9500/XL/XV families and generates EDIF netlists ready for the CPLD fitter.

The general flow of XST for CPLD synthesis is the following:

1. HDL synthesis of VHDL/Verilog designs
2. Macro inference
3. Module optimization
4. EDIF netlist generation

The output of XST for CPLD synthesis consists of the following files:

- Hierarchical EDIF netlist
- NCF file

Global CPLD Synthesis Options

This section describes supported CPLD families and lists the XST options related *only* to CPLD synthesis that can only be set from the Process Properties dialog box within the Project Navigator.

Families

Three families are supported by XST for CPLD synthesis:

- XC9500
- XC9500xl
- XC9500xv

The synthesis for the XC9500xl and XC9500xv families include the clock enable processing; you have the possibility to allow or to invalidate the clock enable signal (when invalidating, it will be replaced by equivalent logic). Also, the selection of the macros which use the clock enable (counters, for instance) depends on the family type. A counter with clock enable will be accepted for XC9500xl/xv families, but rejected (replaced by equivalent logic) for XC9500 devices.

List of Options

Following is a list of CPLD synthesis options that can only be set from the Process Properties dialog box within the Project Navigator. For details about each option, refer to the “CPLD Options” section of the “Design Constraints” chapter.

- “Macro Generator”
- “Flatten Hierarchy”
- “Macro Preserve”
- “XOR Preserve”
- “FF Optimization”

Implementation Details for Macro Generation

XST processes the following macros:

- adders
- subtractors
- add/sub
- multipliers
- comparators
- multiplexers
- counters
- logical shifters
- registers (flip-flops and latches)
- XORs

The macro generation is decided by the Macro Preserve option, which can take two values: *yes* - macro generation is allowed or *no* - macro generation is inhibited. The general macro generation flow is the following:

- HDL infers macros and submits them to the low-level synthesizer;
- Low-level synthesizer accepts or rejects the macros depending on the resources required for the macro implementations.

An accepted macro becomes a hierarchical block (Macro+) or a LogiBlOX black box (depending on the Macro Generator option). For a rejected macro two cases are possible:

- If the hierarchy is kept (Flatten Hierarchy *NO*), the macro becomes a hierarchical block;
- If the hierarchy is not kept (Flatten Hierarchy *YES*), the macro is merged with the surrounded logic.

The rejected macro is replaced by equivalent logic generated by the HDL synthesizer.

It may be that a rejected macro will be decomposed by the HDL synthesizer in component blocks so that one component may be a new macro requiring less resources that the initial one, and the smaller macro may be accepted by XST. For instance, a DFF macro with clock enable (CE) cannot be accepted when mapping onto the XC9500 family. In this case the HDL synthesizer will submit two new macros:

- a DFF macro without Clock Enable signal
- a MUX macro implementing the Clock Enable function

Very small macros (2-bit adders, 4-bit Multiplexers, shifters with shift distance less than 2) are always merged with the surrounded logic, independently of the Preserve Macro or Flatten Hierarchy options, because the optimization process gives better results for larger components.

When selecting *Macro Generator=LogiBlox*, the CPLD low level synthesizer rejects all the macros which are not supported by LogiBlox (macros with bit sizes greater than 64 bits, multipliers, logical shifters, multiplexers with more than 8 bus entries). Also, counters and registers with the following signals: asynchronous load, synchronous load, out enable, count enable are rejected.

When selecting Macro Generator=Macro+, the CPLD low level synthesizer rejects the Adders/Subtractors, multiplexers, signed multipliers, XORs with 2 1-bit entries, and registers.

Latch macros are rejected and generated by the HDL synthesizer. XST implements the latches by separate EDIF netlists:

- *plslat.edn*: latch with active high enable
- *plslatc.edn*: latch with asynchronous reset and active high enable
- *plslatp.edn*: latch with asynchronous set and active high enable
- *plslatpc.edn*: latch with asynchronous set, asynchronous reset and active high enable
- *plslatnpc.edn*: latch with asynchronous set, asynchronous reset and active low enable

Log File Analysis

XST messages related to CPLD synthesis are located after the following message:

```
Starting low level synthesis ...
```

The log file printed by XST contains:

- Tracing of progressively unit optimizations:

  `Optimizing unit` *`unit_name`* `...`

- Information, warnings or fatal messages related to unit optimization:
  - When equation shaping is applied:

    `Collapsing ...`

    `Critical path optimization ...`

  - Removing equivalent flip-flops:

    `ff1` `and` *`ff2`* `are equivalent:` *`ff2`* `is removed`

  - User constraints fulfilled by XST:

    `ncf constraint :` *`constraint_name`*`[=`*`value`*`] :` *`signal_name`*

  - Removing properties of unused signals

    `node not used, ncf property` *`constraint_name`* `removed :` *`signal_name`*

  - Removing properties for signal which cannot have a NET in the EDIF netlist (tied to VCC/GND, equivalent with other signals):

    `no NET, ncf property removed :` *`signal_name`*

  - Multi-sources

    `Multi-source on signal` *`signal_name`* `replaced by logic`

    (if ResolutionStyle = {wire_or | wire_and})

    `Multi-source on signal` *`signal_name`* `not replaced by logic`

(if ResolutionStyle = wire_ms) (fatal)

- The final netlist merging (for hierarchical designs): `Merging netlists ...`
- Final results statistics:

```
Final Results
 Output file name : file_name
 Output format : edif
 Optimization criterion : {area | speed}
 Target Technology : {9500 | 9500xl | 9500xv}
 Flatten Hierarchy : {yes | no}
 Macro Preserve : {yes | no}
 Macro Generation : {Macro+ | LogiBlox | Auto}
 XOR Preserve : {yes | no}

Macro Statistics
 # FSMs  : nb_of_FSMs
 # 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
 # Counters : nb_of_counters
   n-bit {up | down | updown } counter:
   nb_of_n_bit_counters
```

```
Design Statistics
 Edif Instances : nb_of_instances
 I/Os : nb_of_io_ports

Other data
 .NCF file name : ncf_file_name
 HDL output file name : hdl_file_name
 HDL output format : {vhdl | verilog}
```

NCF File

The constraints (attributes) specified in the HDL design or in the constraint file are written by XST into a constraint file with the same name as the output netlist and the extension ncf. The path of the .ncf file is the same as output netlist. The .ncf file is read and processed by the CPLD fitter.

The .ncf file is always created by XST, even if no constraints are specified. A previous .ncf file is not saved; it is always overwritten.

The NCF file generated by XST contains constraints on NETs.

When the equation shaping processing is applied, the .ncf file contains Keep constraints for internal signals of the design and Collapse constraints for all other AND/OR/INV primitives. So, the CPLD fitter is forced to respect the equation tailoring done by XST. An indication for the value of -pterms option of the CPLD fitter is also written (as a comment) in .ncf file:

```
# Minimum value for -pterms option is n
```

You must specify this value for -pterms option when running the CPLD fitter.

Improving Results

XST produces optimized netlists for the CPLD fitter which fits them in specified devices and creates the download programmable files. The CPLD low-level optimization of XST consists of logic minimization, subfunction collapsing, logic factorization, and logic decomposition. The result of the optimization process is an EDIF netlist corresponding to Boolean equation which will be reassembled by the CPLD fitter to fit the best the macrocell capacities. A special XST optimization process, known as equation shaping, is applied when the following options are selected:

- Flatten Hierarchy *yes*
- Optimization Effort *2*
- Optimization Criteria *speed*
- Macro Generator *auto*

In this case, XST optimizes and reduces the Boolean equations to sizes accepted by device macrocells and forces the CPLD fitter to respect these operations through Keep / Collapse constraints written in the .ncf file. The .ncf file also contains an indication on the number of PTerms that you must specify when calling the CPLD fitter:

```
# Minimum value for -pterms option is n.
```

The equation shaping processing includes also a critical path optimization algorithm, which tries to reduce the number of levels of critical paths.

The CPLD fitter multi-level optimization is still recommended because the special optimizations done by the fitter (D to T flip-flop conversion, De Morgan Boolean expression selection).

How to Obtain Better Frequency?

The frequency depends on the number of logic levels (logic depth). In order to reduce the number of levels, the following options are recommended:

- Optimization Effort 2: this value implies the calling of the collapsing algorithm, which tries to reduces the number of levels without increasing the complexity beyond certain limits;
- Optimization Criteria speed: the priority is the reduction of number of levels.

The following tries, in this order, may give successively better results for frequency:

Try 1: only optimization effort 2 and speed optimization are selected. The other options have default values:

- Optimization effort *2*
- Optimization Criteria *speed*

Try 2: the user hierarchy is flattened. In this case the optimization process has a global view of the design and the depth reduction may be better:

- Optimization effort *2*
- Optimization Criteria *area*
- Flatten Hierarchy *yes*

Try 3: the macros are merged with surrounded logic, the design flattening is increased:

- Optimization effort *2*
- Optimization Criteria *area*
- Flatten Hierarchy *yes*
- Macro Preserve *no*

Try 4: applying the equation shaping algorithm. The value of the -pterms option which must be used for the CPLD fitter is written as a warning message in the .ncf file. Options to be selected:

- Optimization effort *2*
- Optimization Criteria *speed*

- Flatten Hierarchy *yes*
- Macro Generator *Auto*

The CPU time is increasing from try 1 to try 4.

Obtaining the best frequency depends also on the CPLD fitter optimization. Xilinx recommends running the multi-level optimization of the CPLD fitter with different values for the -pterms options, starting with 20 and finishing with 50 (except for equation shaping), with a step of 5. Statistically the value 30 gives the best results for frequency.

How to Fit a Large Design?

If a design does not fit in the selected device, over passing the number of device macrocells or device PTerm capacity, an area optimization must be selected for XST. Statistically, the best area results are obtained for the following options:

- Optimization effort *2*
- Optimization Criteria *area*
- Default values for other *options*

Other options that may be tried for better fitting:

- Macro Generator LogiBlox. Some LogiBlox macros, especially adders / subtractors / addsubs, give better results in the number of macrocells and the results are improved when these macros are not merged with the surrounded logic;
- Optimization effort 1: for this effort, the collapsing algorithm is not called; sometimes the collapsing optimization increases the design complexity (number of PTerms) and the fitting may fail.

Chapter 5

Design Constraints

This chapter describes constraints, options, and attributes supported for use with XST.

This chapter contains the following sections.

- “Introduction”
- “Setting Constraints and Options”
- “XST Constraints”
- “HDL Inference and Optimization”
- “FPGA Options”
- “CPLD Options”
- “Summary”
- “Implementation Constraints”
- “Third Party Constraints”
- “Constraints Precedence”

Introduction

Constraints are essential to help you meet your design goals or obtain the best implementation of your circuit. Constraints are available in XST to control various aspects of the synthesis process itself, as well as placement and routing. Synthesis algorithms and heuristics have been tuned to automatically provide optimal results in most situations. In some cases, however, synthesis may fail to initially achieve optimal results; some of the available constraints allow you to explore different synthesis alternatives.

Several mechanisms are available to specify constraints:

- Options provide global control on most synthesis aspects. They can be set either from within the Process Properties dialog box in the Project Navigator or from the command line.
- VHDL attributes can be directly inserted in your VHDL code and attached to individual elements of the design to control both synthesis and placement and routing.
- Similarly, constraints can be Verilog meta comments in your Verilog code.
- If needed, constraints can also be specified in a separate constraints file.

Typically, global synthesis settings are defined within the Process Properties dialog box in the Project Navigator or with command line arguments, while VHDL attributes or Verilog meta comments can be inserted in your source code to specify different choices for individual parts of the design. Note that the local specification of a constraint overrides its global setting. Similarly, if a constraint is set both on a node (or an instance) and on the enclosing design unit, the former takes precedence for the considered node (or instance).

Setting Constraints and Options

This section explains how to set global constraints and options from the Process Properties dialog box within the Project Navigator.

For a brief description of each constraint that applies generally, that is, to FPGAs, CPLDs, VHDL, and Verilog, refer to the “XST Constraints” section in this chapter.

You can find descriptions of the FPGA constraints in the “FPGA Options” section.

You can find descriptions of the CPLD constraints in the “CPLD Options” section.

Note Except for the Value fields with check boxes, there is a pulldown arrow or browse button in each Value field. However, you cannot see the arrow until you click in the Value field.

Synthesis Options

In order to specify the VHDL synthesis options from the Project Navigator:

1. Select a source file from the Source file window.
2. Right click on Synthesize in the Process window.
3. Select **Properties**.
4. When the Process Properties dialog box displays, click the Synthesis Options tab.

   Depending on the HDL language (VHDL or Verilog) and the device family you have selected (FPGA or CPLD), one of three dialog boxes displays:

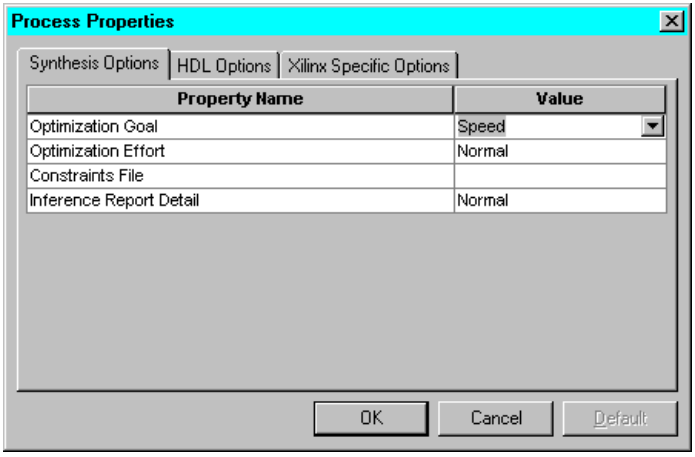


Figure 5-1 Synthesis Options (VHDL and FPGA)

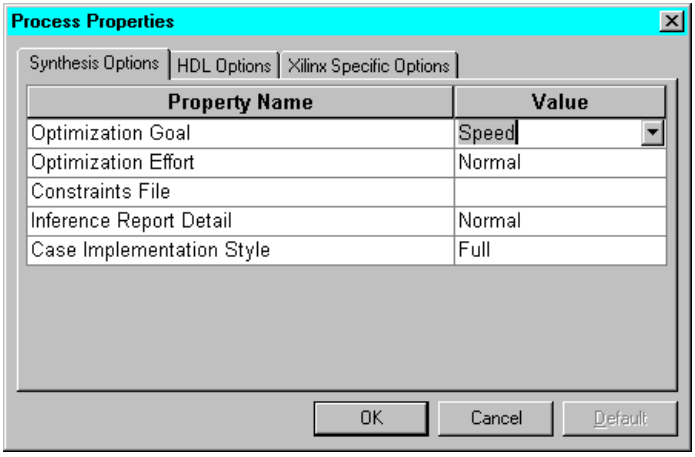


Figure 5-2 Synthesis Options (Verilog and FPGA)

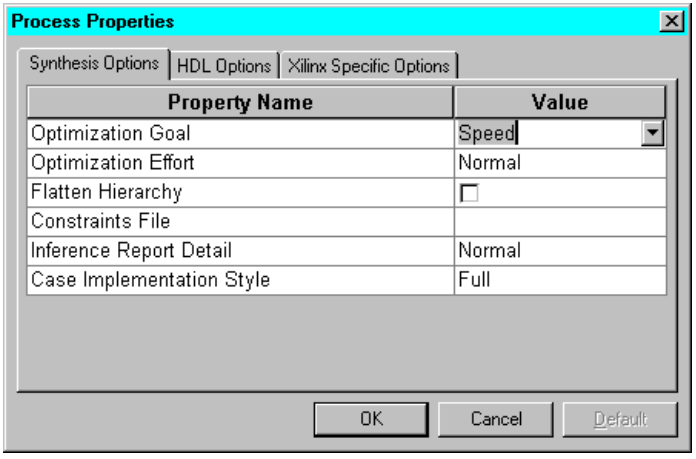


Figure 5-3 Synthesis Options (VHDL or Verilog and CPLD)

Following is a list of the Synthesis Options that can be selected from the dialog boxes:

- "Optimization Goal"
- "Optimization Effort"
- "Flatten Hierarchy" (CPLDs only)
- "Constraints File"
- "Inference Report Detail"
- Case Implementation Style

  Refer to the "Full Case (Verilog)" section and the "Parallel Case (Verilog)" section.

Constraints File

You can set the name of a constraints file used by XST by entering the name of the constraints file in the Value for Constraints File. *Do not use directory or file names that contain spaces.*

Inference Report Detail

This option is not supported for 3.1i.

HDL Options

With the Process Properties dialog box displayed for the Synthesize process, select the HDL Option tab. The following dialog box displays.

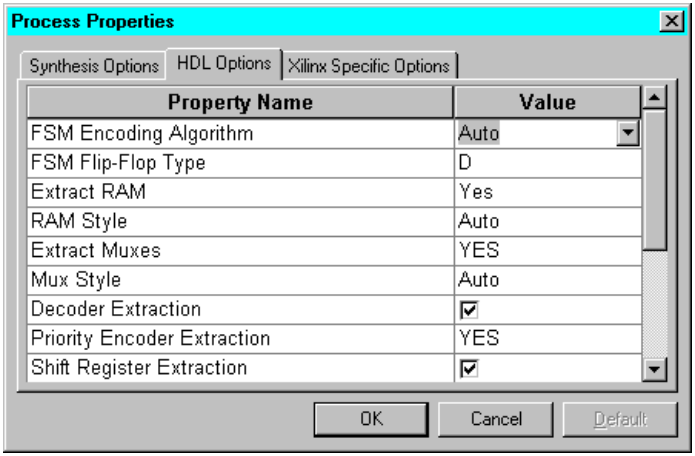


Figure 5-4 HDL Options Tab

Some of the HDL Options cannot be set within the Process Properties dialog box. Following is a list of all HDL Options—including those that cannot be set within the HDL Options tab of the Process Properties dialog box:

- "Automatic FSM Extraction"
- "FSM Encoding Algorithm"
- "FSM Flip-Flop Type"
- "Enumeration Encoding"
- "Extract RAM"
- "RAM Style"
- "Extract Muxes"
- "Mux Style"
- "Decoder Extraction"
- "Priority Encoder Extraction"
- "Shift Register Extraction"
- "Logical Shifter Extraction"
- "XOR Collapsing"
- "Resource Sharing"

- “Complex Clock Enable Extraction”
- “Resolution Style”

Xilinx Specific Options

From the Process Properties dialog box for the Synthesize process, select the Xilinx Specific Options tab to display the options.

Depending on the device family, one of the following dialog boxes displays:

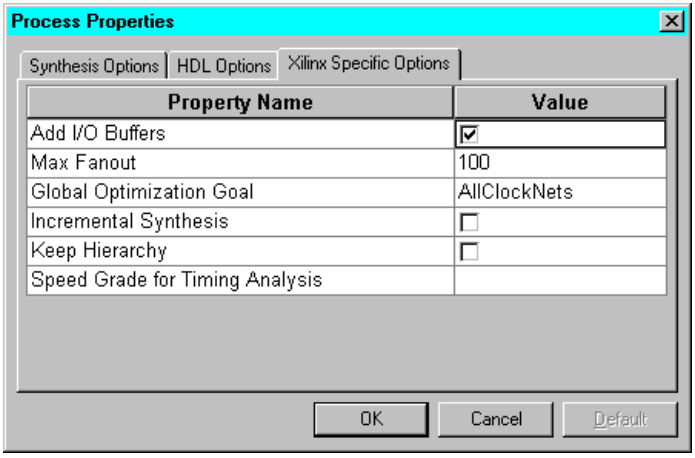


Figure 5-5 Xilinx Specific Options (FPGAs)

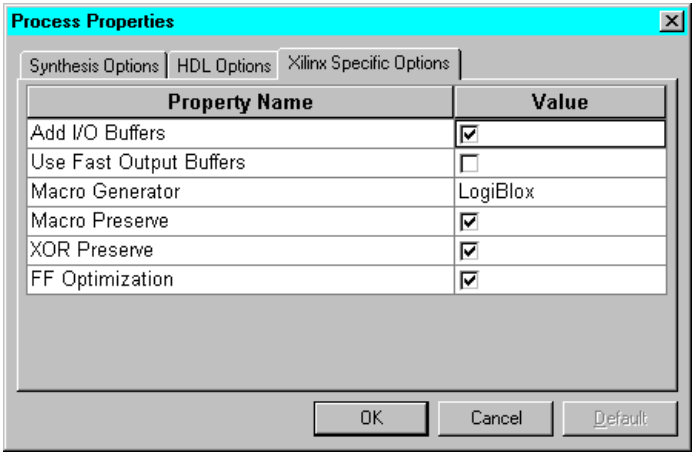


Figure 5-6 Xilinx Specific Options (CPLDs)

Following is a list of the Xilinx Specific Options:

- "Add IO Buffers"
- "Max Fanout" (FPGA Only)
- "Global Optimization Goal" (FPGA Only)
- "Keep Hierarchy" (FPGA Only)
- "Speed Grade for Timing Analysis" (FPGA Only)
- "Macro Preserve" (CPLD Only)
- "XOR Preserve" (CPLD Only)
- "FF Optimization" (CPLD Only)

Command Line Options

Options can be invoked in command-line mode using the following syntax:

```
-OptionName OptionValue
```

Example:

```
run -ifn mydesign.v -ifmt verilog -ofn mydesign.edn
-o fmt edif -opt_mode speed -opt_level 2 -fsm_encoding
compact
```

For more details, refer to the "Command Line Mode" chapter.

VHDL Attribute Syntax

In your VHDL code, constraints can be described with VHDL attributes. Before it can be used, an attribute must be declared with the following syntax.

```
attribute AttributeName : Type ;
```

Example:

```
attribute RLOC : string ;
```

The attribute type defines the type of the attribute value. Allowed types are **string**, **boolean** and **integer**. An attribute can be declared in an entity or architecture. If declared in the entity, it is visible both in the entity and the architecture body. If the attribute is declared in the architecture, it cannot be used in the entity declaration. Once declared a VHDL attribute can be specified as follows:

```
attribute AttributeName of ObjectList : ObjectType is
AttributeValue ;
```

Examples:

```
attribute RLOC of u123 : label is "R11C1.S0" ;
attribute bufg of signal_name: signal is {“clk”
|”sr”|”oe”};
```

The object list is a comma separated list of identifiers. Accepted object types are entity, component, label, signal, variable and type.

Verilog Meta Comment Syntax

Constraints can be specified as follows in Verilog code:

```
// synthesis attribute AttributeName [of]
ObjectName [is] AttributeValue
```

Example:

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
//synthesis attribute bufg of signal_name is {“clk”
|”sr”|”oe”};
```

Note The parallel_case, full_case, translate_on and translate_off directives follow a different syntax described later in the section on XST language level constraints.

Constraint File Syntax and Utilization

The constraint file syntax is derived from the VHDL attribute syntax with a few differences pointed out below. The main difference is that no attribute declaration is required. An attribute can be directly specified using the following syntax:

```
attribute AttributeName of ObjectName : ObjectType is
"AttributeValue" [;]
```

A statement only applies to one object. A list of object identifiers cannot be specified in the same statement. Allowed object types are entity, label and signal. Attribute values are not typed and should always be strings. In a hierarchical design, use the following begin and end statements to access objects in hierarchical units. They are not required if the considered object is in the top-level unit.

```
begin UnitName
end UnitName [;]
```

Example:

```
begin alu
attribute resource_sharing of result : signal is
   "yes" ;
end alu ;
```

Note that begin and end statements only apply to design units. They cannot refer to unit instances. As a result, begin and end statements should never appear inside another begin/end section.

A constraint file can be specified in the Constraint File section of the Process Properties dialog box in the Project Navigator, or with the -attribfile command line option. The option value is a relative or absolute path to the file.

XST Constraints

This section discusses various constraints that can be used with XST. These constraints apply to FPGAs, CPLDs, VHDL, and Verilog. With the exception of Box Type, these options can all be set within the Synthesis Options tab of the Process Properties dialog box within the Project Navigator.

General

This section describes the opt_mode, opt_level, and box_type constraints. These constraints can be used with FPGAs, CPLDs, VHDL, and Verilog

Optimization Goal

The opt_mode constraint defines the synthesis optimization strategy. Available strategies are *speed* and *area*. By default, XST optimizations are speed-oriented.

- *speed*: priority is to reduce the number of logic levels, therefore to increase frequency;
- *area*: priority is to reduce the total amount of logic used for design implementation, therefore to improve design fitting.

The constraint can be globally defined with the Optimization Goal option in the Synthesis Options tab of the Process Properties dialog box in the Project Navigator, or with the -opt_mode command line option. Optimization can also be controlled locally at the VHDL entity / architecture or Verilog module level with a VHDL attribute or Verilog meta comment.

Optimization Effort

The opt_level constraint defines the synthesis optimization effort level. Allowed values are *1* (normal optimization) and *2* (higher optimization). The default optimization effort level is *1* (Normal).

- **`1 (Normal)`**: very fast processing, especially for hierarchical designs
- **`2 (Higher Optimization)`**: time consuming processing, with better results in number of macrocells or maximum frequency.

The constraint can be globally defined with the Optimization Effort option in the Synthesis Options tab of the Process Properties dialog box in the Project Navigator, or with the -opt_level command line option. Optimization can also be controlled locally at the VHDL entity / architecture or Verilog module level with a VHDL attribute or Verilog meta comment.

Note Selecting optimization level 2 usually results in increased synthesis run times.

Box Type

The box_type constraint characterizes an instance. It currently takes only one possible value: *black_box*. The black_box value instructs XST not to synthesize the instance and to propagate generic INIT values to the output EDIF netlist.

The constraint can be attached to a VHDL or Verilog instance through an attribute or a meta comment.

Case Implementation Style

This section describes the translate_on, translate_off, parallel_case, and full_case directives. These directives can be globally defined with the Case Implementation Style option in the Synthesis Options tab of the Process Properties dialog box in the Project Navigator.

Translate Off/Translate On (Verilog/VHDL)

The translate_off and translate_on directives can be used to instruct XST to ignore portions of your VHDL or Verilog code that are not relevant for synthesis—for example, simulation code. The translate_off directive marks the beginning of the section to be ignored and the translate_on directive instructs XST to resume synthesis from that point.

The directives are available as VHDL or Verilog meta comments. The Verilog syntax differs from the standard meta comment syntax presented earlier in this chapter, as shown below.

```
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```

In your VHDL code, the directives should be written as follows:

```
-- synthesis translate_off
 ...code not synthesized...
-- synthesis translate_on
```

Parallel Case (Verilog)

The parallel_case directive is a Verilog meta comment used to force a case statement to be synthesized as a parallel multiplexer and prevents the case statement from being transformed into a prioritized if / elsif cascade.

The directive is exclusively available as a meta comment in your Verilog code and cannot be specified in a VHDL description or in a separate constraint file. The syntax differs from the standard meta comment syntax presented earlier in this chapter, and shown in the following:

```
// synthesis parallel_case
```

Since the directive does not contain a target reference, the meta comment immediately follows the selector.

Example:

```
casex select // synthesis parallel_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

Full Case (Verilog)

The full_case directive is a Verilog meta comment used to indicate that all possible selector values have been expressed in a case, casex or casez statement. Values that are not expressed are simply considered as never being reached during normal circuit operation, and the directive prevents XST from creating additional hardware for those conditions.

The directive is available as a meta comment in your Verilog code and cannot be specified in a VHDL description or in a separate constraint file. The syntax differs from the standard meta comment syntax presented earlier in this chapter, as shown in the following line.

```
// synthesis full_case
```

Since the directive does not contain a target reference, the meta comment immediately follows the selector.

Example:

```
casex select / / synthesis full_case
4'b1xxx: res = data1;
4'bx1xx: res = data2;
4'bxx1x: res = data3;
4'bxxx1: res = data4;
endcase
```

Both the parallel_case and full_case directives may be applied to a case selector. A single meta comment is necessary and both directives should be separated by a space, as follows:

```
// synthesis parallel_case full_case
```

Add IO Buffers

XST automatically inserts Input/Output Buffers into the design. You can manually instantiate I/O Buffers for some or all the I/Os, and XST will insert I/O Buffers only for the remaining I/Os. If you do not want XST to insert any I/O Buffers, then set this option to NO. This option is useful to synthesize a part of a design to be instantiated later on.

The iobuf constraint enables or disables IO buffer insertion. Allowed values are *yes* and *no*. By default, buffer insertion is enabled.

When the *Yes* value is selected, IBUF and OBUF primitives will be generated. IBUF/OBUF primitives are connected to I/O ports of the top-level module. When XST is called to synthesize an internal module which will be instantiated later in a larger design, you must select NO for this option. If I/O buffers are added to a design, this design cannot be used as a submodule of another design.

The constraint can only be specified globally with the Add IO Buffers option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -iobuf command line option.

HDL Inference and Optimization

This section describes encoding and extraction constraints. Most of the constraints can be set globally in the HDL Options tab of the Process Properties dialog box in the Project Navigator. Constraints that *cannot* be set in this dialog box are Automatic FSM Extraction and Enumeration Encoding. The constraints described in this section apply to FPGAs, CPLDs, VHDL, and Verilog.

Automatic FSM Extraction

The fsm_extract constraint enables or disables finite state machine extraction and specific synthesis optimizations. Allowed values are *yes* and *no*. By default, FSM synthesis is enabled (*yes*). This option must be enabled in order to set values for the FSM Encoding Algorithm and FSM Flip-Flop Type.

The constraint can be set with the -fsm_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control FSM extraction at the VHDL entity / architecture or Verilog module level, or for an individual signal.

FSM Encoding Algorithm

The fsm_encoding constraint selects the finite state machine coding technique to be used. Available property values are *auto*, *one-hot*, *compact*, *sequential*, *gray*, *johnson*, and *user*. The constraint defaults to *auto*, meaning that the best coding technique is automatically selected for each individual state machine. The Automatic FSM Extraction option must be enabled in order to select a value for FSM Encoding Algorithm.

The constraint can be globally set with the FSM Encoding Algorithm option in the Process Properties dialog box in the Project Navigator, or with the -fsm_encoding command line option. A VHDL attribute or Verilog meta comment may also be applied on a VHDL entity / architecture or Verilog module level, or on an individual signal.

FSM Flip-Flop Type

The fsm_fftype constraint defines with what type of flip-flops the state register should be implemented in a FSM. Allowed values are: *d* and *t*. By default, D flip-flops are used. The Automatic FSM Extraction option must be enabled in order to select a value for FSM Flip-Flop Type.

The constraint can be globally set with the FSM Flip-Flop Type option in the Process Properties dialog box in the Project Navigator, or with the -fsm_fftype command line option. A VHDL attribute or Verilog meta comment may also be applied on a VHDL entity / architecture or Verilog module level, or on an individual signal.

Enumeration Encoding

The enum_encoding constraint can be used to apply a specific encoding to a VHDL enumerated type. The constraint value is a string containing space-separated binary codes. The constraint can only be specified as a VHDL attribute on the considered enumerated type, as shown in the example below.

```
...
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
...
```

Note If the constraint is specified in a separate constraints file instead, it should be applied directly to the signal(s) with the considered enumerated type instead, as follows.

```
attribute enum_encoding of state1 : signal is "001 010
100 111";

attribute enum_encoding of state2 : signal is "001 010
100 111";
```

Note When describing a finite state machine using an enumerated type for the state register, a particular encoding scheme may be specified with an enum_encoding constraint. In order for this encoding to be actually used by XST, you must also set the fsm_encoding constraint to user for the considered state register.

Extract RAM

The ram_extract constraint enables or disables RAM macro inference. Allowed values are *yes*, *no* and *force*. By default, RAM inference is enabled (*yes*). For each identified RAM description, based on some internal decision rules, XST actually creates a macro or optimizes it with the rest of the logic. The *force* value allows you to override those decision rules and force XST to create the mux macro.

The constraint can be globally set with the Extract RAM option in the Process Properties dialog box within the Project Navigator, or with the -ram_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Extract Muxes

The mux_extract constraint enables or disables multiplexer macro inference. Allowed values are *yes*, *no* and *force*. 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 allows you to override those decision rules and force XST to create the mux macro.

The constraint can be globally set with the Extract Muxes option in the Process Properties dialog box within the Project Navigator, or with the -mux_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Decoder Extraction

The decoder_extract constraint enables or disables decoder macro inference. Allowed values are *yes* (check box is checked) and *no* (check box in not checked). By default, decoder inference is enabled (check box is checked).

The constraint can be globally set with the Decoder Extraction option in the Process Properties dialog box within the Project Navigator, or with the -decoder_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Priority Encoder Extraction

The priority_extract constraint enables or disables priority encoder macro inference. Allowed values are *yes*, *no* and *force*. By default, priority encoder inference is enabled (*yes*). For each identified priority encoder description, based on some internal decision rules, XST will actually create a macro or optimize it with the rest of the logic. The *force* value allows to override those decision rules and force XST to extract the macro. Priority encoder rules are currently very restrictive. Based on architectural considerations, the *force* value will allow you to override these rules and potentially improve the quality of your results.

The constraint can be globally set with the Priority Encoder Extraction option in the Process Properties dialog box within the Project Navigator, or with the -priority_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Shift Register Extraction

The shreg_extract constraint enables or disables shift register macro inference. Allowed values are *yes* (check box is checked) and *no* (check box is not checked) By default, shift register inference is enabled.

The constraint can be globally set with the Shift Register Extraction option in the Process Properties dialog box within the Project Navigator, or with the -shreg_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Logical Shifter Extraction

The shift_extract constraint enables or disables logical shifter macro inference. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, logical shifter inference is enabled.

The constraint can be globally set with the Logical Shifter Extraction option in the Process Properties dialog box within the Project Navigator, or with the -shift_extract command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

XOR Collapsing

The xor_collapse constraint controls whether cascaded XORs should be collapsed into a single XOR. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, XOR collapsing is enabled.

The constraint can be globally set with the XOR Collapsing option in the Process Properties dialog box within the Project Navigator, or with the -xor_collapse command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Resource Sharing

The resource_sharing constraint enables or disables resource sharing of arithmetic operators. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, resource sharing is enabled.

The constraint can be globally set with the Resource Sharing option in the Process Properties dialog box within the Project Navigator, or with the -resource_sharing command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Complex Clock Enable Extraction

Sequential macro inference in XST generates macros with clock enable functionality whenever possible. The complex_clken constraint instructs or prevents the inference engine to not only consider basic clock enable templates, but also look for less obvious descriptions where the clock enable can be used. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, clock enable extraction is performed with the higher effort.

The constraint can be globally set with the Complex Clock Enable Extraction option in the Process Properties dialog box within the Project Navigator, or with the -complex_clken command line option. A VHDL attribute or Verilog meta comment may also be used to control inference at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Resolution Style

The resolution style constraint controls how multisource situations not protected by tristate logic are handled. Allowed values are *wire_ms*, *wire_or* and *wire_and*. The wire_ms value is assumed by default and instructs XST to exit with an error condition whenever such a multisource situation is found. On the contrary, with the wire_or and wire_and resolution styles, all detected situations are replaced by respectively OR-based logic or AND_based logic, and synthesis continues.

The constraint can only be defined globally with the Resolution Style option in the Process Properties dialog box within the Project Navigator, or with the -resolutionstyle command line option. The constraint is not available as a VHDL attribute or Verilog meta comment.

FPGA Options

This section describes FPGA HDL options. These options apply only to FPGAs—not CPLDs.

Mux Style

The mux_style constraint controls the way the macrogenerator implements the multiplexer macros. Allowed values are *auto*, *muxf* and *muxcy*. The default value is *auto*, meaning that XST looks for the best implementation for each considered macro. Available implementation styles for the Virtex and Spartan2 series are based on either MuxF5 / F6 resources or MuxCY resources.

The constraint can be globally set with the Mux Style option in the HDL Options tab of the Process Properties dialog box within the Project Navigator, or with the -mux_style command line option. A VHDL attribute or Verilog meta comment may also be used to control mux implementation at the VHDL entity / architecture or Verilog module level, or for an individual signal.

RAM Style

The ram_style constraint controls the way the macrogenerator implements the inferred RAM macros. Allowed values are *auto*, *block* and *distributed*. The default value is *auto*, meaning that XST looks for the best implementation for each inferred RAM. The implementation style can be manually forced to use block RAM or distributed RAM resources available in the Virtex and Spartan2 series.

The constraint can be globally set with the RAM Style option in the HDL Options tab of the Process Properties dialog box within the Project Navigator, or with the -ram_style command line option. A VHDL attribute or Verilog meta comment may also be used to control RAM implementation at the VHDL entity / architecture or Verilog module level, or for an individual signal a RAM is inferred from.

Speed Grade for Timing Analysis

For the timing analysis, XST takes into account the speed grade. By specifying a different speed grade, XST will perform a different optimization due to different timing information. The default is the fastest speed grade available. The constraint value is an integer. Values that can be selecteddepend on the architecture. Following is a list of valid integers for each architecture:

- Virtex: 4, 5, and 6
- VirtexE: 6, 7, and 8
- Virtex2: 4, 5, and 6
- Spartan2: 5 and 6

The constraint can only be defined globally with the Speed Grade for Timing Analysis option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -speedgrade command line option.

Max Fanout

The maxfanout constraint can be used to limit the fanout of nets. The constraint value is an integer and is equal to 100 by default.

The constraint can be globally set with the Max Fanout option in the Xilinx Specific Options tab of the Process Properties dialog box in the Project Navigator, or with the -maxfanout command line option. A VHDL attribute or Verilog meta comment may also be used to control maximum fanout at the VHDL entity / architecture or Verilog module level, or for an individual signal.

Large fanouts can cause routability problems, therefore 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 (below 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, then buffers will be inserted. These buffers will be protected against logic trimming at the implementation level by defining a KEEP attribute in the .NCF file.

If the register replication option is set to NO then only buffers will be used to control fanout of flip-flops and latches.

This option is global for the design, but you can control maximum fanout independently for each entity / module or for given individual signals by using attributes.

Add Generic Clock Buffer

The bufg constraint controls the maximum number of BUFG created by XST. The constraint value is an integer and is equal to 4 by default.

The constraint can only be specified with the -bufg command line option.

Maximum Number of Clock Buffers Created by XST

XST automatically inserts clock buffers for clock signals. If you want XST to use less than the maximum number of clock buffers available, then specify a lower number. This option is useful if the design being synthesized is not complete and if the missing part contains clock buffers. The default value is 4. The type of clock buffer used for a given port can be controlled using the clock_buffer constraint.

Clock Buffer Type

The clock_buffer constraint selects the type of clock buffer to be inserted on the clock port. Allowed values are **`bufgdll`**,**`ibufg`**, **`bufgp`**,**`ibuf`** and **`none`**. By default, a BUFGP is inserted.

Specifying a Port as a Clock

The sig_isclock constraint can be used to indicate if an input port on a black box is a clock.

Packing Flip-Flops and Latches in IOBs

XST considers the IOB constraints as an implementation constraint and will therefore propagate them in the generated .ncf file.

XST also duplicates the flip-flops and latches driving the Enable pin of output buffers, so that the corresponding flip-flops and latches can be packed in the IOB.

The clock_buffer constraint selects the type of clock buffer to be inserted on the clock port. Allowed values are *bufgdll*, *ibufg*, *bufgp*, *ibuf* and *none*. By default, a BUFGP is inserted.

The constraint is available as a VHDL attribute or a Verilog meta comment and should be attached to a port signal.

Sig_isclock

The sig_isclock constraint indicates if an input port on a black box is a clock. Allowed values are *yes* and *no*.

The constraint is only available as a VHDL attribute or a Verilog meta comment and should be attached to a port signal.

Register Duplication

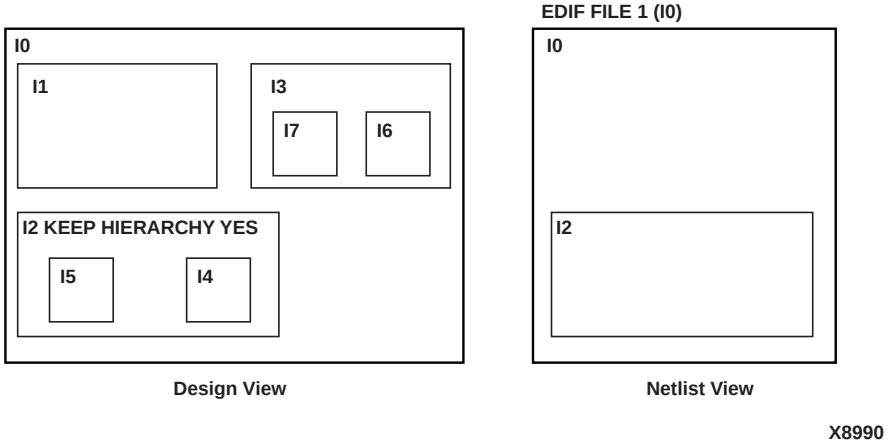
The register_duplication constraint enables or disables register replication. Allowed values are *yes* and *no*. By default, register replication is enabled and is performed during timing optimization and fanout control.

The constraint can be set with the -register_duplication command line option. A VHDL attribute or Verilog meta comment may also be used to control register replication at the VHDL entity / architecture or Verilog module level.

Keep Hierarchy

XST may automatically flatten the design to get better results by optimizing entity / module boundaries. You can set the keep_hierarchy option to YES (check box is checked) so that the generated netlist is hierarchical and respects the hierarchy and interface of any Entity / Module of your design. This option is global for the design, but it can also be specified independently for each entity / module by using attributes. If the attribute keep_hierarchy is set on a entity / module then its hierarchy and interface will be preserved, while the remaining portion of the design may be flattened.

In the following figure, if the attribute keep_hierarchy is set to the entity / module I2, then the hierarchy of I2 will be in the final netlist, but its contents I4, I5 will be flattened inside I2. Also I1, I3, I6, I7 will be flattened.



The constraint can be globally set with the Keep Hierarchy option in the Process Properties dialog box within the Project Navigator, or with the -keep_hierarchy command line option. A VHDL attribute or Verilog meta comment may also be used to control maximum fanout at the VHDL entity / architecture or Verilog module level.

Incremental Synthesis

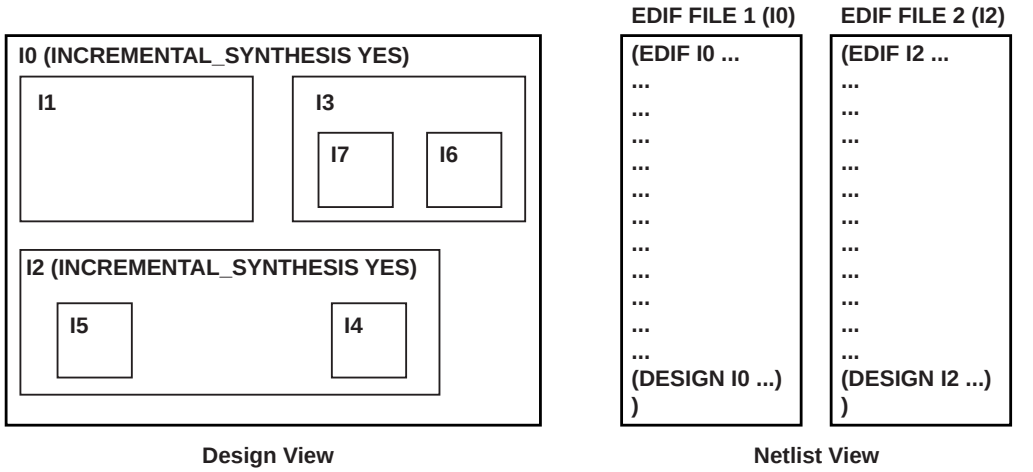
XST allows incremental synthesis. This feature is based on two constraints:

- Incremental_synthesis
- Resynthesis

If the incremental_synthesis option is set to YES (check box is checked), then each entity / module of the design will be generated in a single separate file.

When you synthesize a given design again, you can add the resynthesize attribute only on the entity / module you have changed. XST will automatically read the EDIF netlist of the unchanged entity / module to get its timing information but will not modify it. Only the entity / module with the Resynthesize attribute will be resynthesized. Note that by default (check box is not checked), XST will consider that a given entity module has not been modified, you must manually insert the Resynthesis attribute to force resynthesis.

The incremental_synthesis option can be applied independently on any entity / module. In the following figure, the incremental_synthesis attribute has been set to the entities I0 and I2. Therefore two netlists will be generated, one for I0 and one for I2. Because the entities / modules I1, I3, I4, I5, I6, I7 do not have the incremental_synthesis attribute, they will not be in a separate file. In the netlist describing I0, there will be the contents of I1, I3, I6 and I7; their hierarchy will be preserved according to the option keep_hierarchy, but by default, they will not be preserved. In the next run, if you have changed the entity I4, you have to manually set the attribute Resynthesize to this entity / module. As I4 belongs to the set (I2, I4, I5), all these entity / module will be resynthesized, but the netlist I0 will remain unchanged.



The incremental_synthesis constraint can be applied on a VHDL entity or Verilog module so that XST generates a single and separate EDIF netlist file for it and its descendents. Allowed values are *yes (*check box is checked*)* and *no (*check box is not checked*)* By default, the constraint is globally set to *no* and XST creates a unique netlist for the complete design.

The constraint can be globally set with the Incremental Synthesis option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -incremental_synthesis command line option. A VHDL attribute or Verilog meta comment can also be attached respectively to a VHDL entity / architecture or to a Verilog module.

Resynthesis

The resynthesis constraint forces or prevents resynthesis of an entity or module. Allowed values are *yes* and *no*. By default, your design is always resynthesized whenever XST is run.

With a VHDL attribute or Verilog meta comment, the constraint can be attached respectively to a VHDL entity / architecture or to a Verilog module. No global resynthesis option is available.

Global Optimization Goal

XST can optimize different regions (register to register, inpad to register, register to outpad and inpad to outpad) of the design depending on this option. By default, XST optimizes the design for Maximum Frequency (register to register). Please refer to the “Timing Constraints” section of the “FPGA Optimization” chapter for a detailed description of supported timing constraints.

The glob_opt constraint selects the global optimization goal. Allowed values are *allclocknets*, *inpad_to_outpad*, *offset_in_before*, *offset_out_after*, *max_delay*. By default, global optimization is tuned for clock frequency maximization (ALLCLOCKNETS).

XST also supports a set of specific timing constraints allowing full control of timing optimization. See the “Timing Constraints” section of the “FPGA Optimization” chapter for details.

The constraint can only be specified globally with the Global Optimization option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -glob_opt command line option.

CPLD Options

This section describes options that only apply to CPLDs.

Macro Generator

The macrogen constraint controls the generation of inferred macros and provides a choice between the XST macro generator (Macro+) and the LogiBLOX macro generator.

A macro inferred by the HDL synthesizer is passed to a CPLD low-level synthesizer which calls a macrogenerator to create its implementation. Two macrogenerators are available:

- *Macro+*: XST internal Macrogenerator
- *LogiBlOX*: M1 macrogenerator (the default)

A macro submitted by the HDL synthesizer may be accepted or rejected by the CPLD synthesizer. An accepted macro becomes a hierarchical block in the final netlist, its logic being generated by Macro+ or, later, by CPLD fitter (LogiBlox macro).

Allowed values are *macro+*, *logiblox* and *auto*. By default, LogiBLOX is automatically selected.

- *Macro+*: the accepted macros will be generated by Macro+
- *LogiBlOX*: the accepted macros will be replaced by black boxes which will be expanded by CPLD fitter
- *Auto*: the best implementation of the macro, between Macro+ and LogiBlOX, is selected

The Macro Generator option is closely related to the Macro Preserve option: the macros are generated only if Macro Preserve is *yes*. Otherwise, the macros are replaced by equivalent logic units generated by the HDL synthesizer.

The constraint can only be defined globally with the Macro Generator option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -macrogen command line option.

Flatten Hierarchy

This option is related to the hierarchical blocks (VHDL entities, Verilog modules) specified in the HDL design and does not concern the macros inferred by the HDL synthesizer. Two values are available for this option:

- *yes* (check box is checked): hierarchical blocks are merged in the top level module
- *no* (check box is not checked): allows the preservation of the design hierarchy, as described in the HDL project. (the default)

Regardless of the Flatten hierarchy option, XST generates a single EDIF file. In general, an HDL design is a collection of hierarchical blocks and 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 (less number of PTerms and device macrocells, better frequency) because the optimization processes (collapsing, factorization) are applied globally on the entire logic.

The merge constraint enables or disables hierarchical flattening of user-defined design units. Allowed values are yes and no. By default, the user hierarchy is preserved.

The constraint can only be defined globally with the Flatten Hierarchy option in the Synthesis Options tab of the Process Properties dialog box within the Project Navigator, or with the -merge command line option.

Macro Preserve

This option is useful for making the macro handling independent of design hierarchy processing (see Flatten Hierarchy option). So you can merge all hierarchical blocks in the top module, but you can still keep the macros as hierarchical modules. Also, you can keep the design hierarchy excepting the macros which are merged with the surrounded logic. Sometimes, merging the macros gives better results for design fitting. Two values are available for this option:

- *yes* (check box is checked): macros are preserved and generated by Macro+ or LogiBlox. This is the default.
- *no* (check box is not checked): macros are rejected and generated by HDL synthesizer

Depending on the Flatten Hierarchy value, a rejected macro becomes a hierarchical block (Flatten Hierarchy=*no*) or is merged in the design logic (Flatten Hierarchy=*yes*). Please note that very small macros (2-bit adders, 4-bit multiplexers) are always merged, independent of the Macro Preserve or Flatten Hierarchy options.

The pld_mp constraint enables or disables hierarchical flattening of macros.

The constraint can only be defined globally with the Macro Preserve option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -pld_mp command line option.

XOR Preserve

The pld_xp constraint enables or disables hierarchical flattening of XOR macros. Allowed values are *yes* (check box is checked) and *no* (check box is not checked). By default, XOR macros are preserved (check box is checked).

The constraint can only be defined globally with the XOR Preserve option in the Xilinx Specific Options tab of the Process Properties dialog box within the Project Navigator, or with the -pld_xp command line option.

The XORs inferred by HDL synthesis are also considered as macro blocks in the CPLD flow, but they are processed separately to give more flexibility for the use of device macrocells XOR gates. Therefore, you can decide to flatten its design (Flatten Hierarchy *yes*, Macro Preserve *no*) but you want to preserve the XORs. Preserving XORs has a great impact on reducing design complexity. Two values are available for this option:

- *yes*: XOR macros are preserved
- *no*: XOR macros are merged with surrounded logic

The preserved XORs appear in the EDIF netlist as LogiBlOX XOR macros, and will be expanded by the CPLD fitter.

Preserving the XORs, generally, gives better results, that is, the number of PTerms is lower. The *No* value is useful to obtain completely flat netlists. Sometimes, applying the global optimization on a completely flat design improves the design fitting.

A completely flattened design is obtained selecting the following options:

- Flatten Hierarchy *yes*
- Macro Preserve *no*
- XOR Preserve *no*

Note that the *No* value for this option does not guarantee the elimination of the XOR operator from the EDIF netlist. During the netlist generation, the netlist mapper tries to recognize and infer XOR gates in order to decrease the logic complexity. This process is independent of the XOR preservation done by HDL synthesis and is guided only by the goal of complexity reduction.

FF Optimization

The pld_ffopt constraint enables or disables flip-flop optimization. Flip-flop optimization includes the removal of equivalent flip-flops and flip-flops with constant inputs. This processing increases the fitting success as a result of the logic simplification implied by the flip-flops elimination. Two values are available:

- *yes* (check box is checked): flip-flop optimization is allowed. This is the default.
- *no* (check box is not checked): flip-flop optimization is inhibited

Note that the flip-flop optimization algorithm is time consuming, therefore, when fast processing is desired, this option must be invalidated.

The constraint can only be defined globally with the FF Optimization option in the Xilinx Specific Option tab in the Process Properties dialog box within the Project Navigator, or with the -pld_ffopt command line option.

Complex Clock Enable Extraction

The pld_ce constraint specifies how sequential logic should be implemented when it contains a clock enable, either using the specific device resources available for that or generating equivalent logic.

The constraint can only be defined globally with the Clock Enable option in the HDL Options tab of the Process Properties dialog box within the Project Navigator, or with the -pld_ce command line option.

This option allows you to specify the way the clock enable function will be implemented if presented in the design. Two values are available:

- *yes* (check box is checked): the synthesizer implements the use of the Clock Enable signal of the device
- *no* (check box is not checked): the Clock enable function will be implemented through equivalent logic

Keeping or not keeping the clock enable signal depends on the design logic. Sometimes, when the clock enable is the result of a Boolean expression, saying *No* with this option may improve the fitting result because the input data of the flip-flop is simplified when it is merged with the clock enable expression.

Summary

The following summarizes all available XST-specific options, with allowed values for each, the type of objects they can be applied to, and usage restrictions. Defaults are indicated in bold.

Name	Values	Target	Comment
box_type	black_box	label	All
bufg	*integer*	global	FPGA only
clock_buffer	bufgdll, ibufg, **bufgp**, ibuf, none	global	FPGA only
decoder_extract	**yes**, no	global, entity, signal	All
enum_encoding	*string*	type, signal	All
fsm_extract	**yes**, no	global, entity, signal	All

Name	Values	Target	Comment
fsm_encoding	**auto**, one-hot, compact, sequential, gray, johnson, user	global, entity, signal	All
fsm_fftype	d, t	global, entity, signal	All
full_case	*no value*	case statement	Verilog meta comments only
glob_opt	**allclocknets**, inpad_to_outpad, offset_in_before, offset_out_after, max_delay	global	FPGA only
incremental_synthesis	yes, **no**	global, entity	FPGA only
iobuf	**yes**, no	global	All
keep_hierarchy	yes, **no**	global, entity	FPGA only
macrogen	**auto**, macro+, logiblox	global	CPLD only
maxfanout	*integer*	global, entity, signal	FPGA only
merge	yes, **no**	global	CPLD only
mux_extract	**yes**, no	global, entity, signal	All
mux_style	**auto**, muxf, muxcy	global, entity, signal	FPGA only
opt_level	**1**, 2	global	All
opt_mode	**speed**, area	global	All
parallel_case	*no value*	case statement	Verilog meta comments only
pld_ce	**yes**, no	global	CPLD only
pld_ffopt	**yes**, no	global	CPLD only
pld_mp	yes, **no**	global	CPLD only
pld_xp	yes, **no**	global	CPLD only
priority_extract	**yes**, no	global, entity, signal	All
ram_extract	**yes**, no	global, entity, signal	FPGA only
ram_style	**auto**, block, distributed	global, entity, signal	FPGA only

Name	Values	Target	Comment
register_duplication	**yes**, no	global, entity	FPGA only
resolution_style	**wire_ms**, wire_or, wire_and	global	All
resource_sharing	**yes**, no	global, entity, signal	All
resynthesis	**yes**, no	entity	FPGA only
shift_extract	**yes**, no	global, entity, signal	All
shreg_extract	**yes**, no	global, entity, signal	All
sig_isclock	yes, **no**	signal	FPGA only
speedgrade	*integer*	global	FPGA only
translate_on	*no value*	*local, no target*	All
translate_off	*no value*	*local, no target*	All
xor_collapse	**yes**, no	global, entity, signal	All
vlgcase	full, parallel, full-parallel	global	Verilog only

Implementation Constraints

This section explains how XST handles implementation constraints.

Handling by XST

Implementation constraints control placement and routing. They are not directly useful to XST and are simply propagated and made available to the implementation tools. The constraints are written either in the output EDIF netlist or in the associated NCF file. In addition, the object an implementation constraint is attached to will be preserved. XST does not check the validity of an implementation constraint.

Implementation constraints appear in an NCF/UCF file according to one of the two following syntaxes:

{**NET**|**INST**|**PIN**} {*NetName*|*InstName*|*PinName*}
PropertyName;
{**NET**|**INST**|**PIN**} {*NetName*|*InstName*|*PinName*}
PropertyName=*PropertyValue*;

When written in VHDL code, they should be specified as follows respectively:

```
attribute PropertyName of
   {NetName|InstName|PinName} : {signal|label} is
   "true";
attribute PropertyName of
   {NetName|InstName|PinName} : {signal|label} is
   "PropertyValue";
```

In a Verilog description, they should be written as follows respectively:

```
// synthesis attribute PropertyName [of]
   {NetName|InstName|PinName} [is] "true"
// synthesis attribute PropertyName [of]
   {NetName|InstName|PinName} [is] "PropertyValue"
```

Examples

Following are three examples.

Example 1

When targeting an FPGA device, the RLOC constraint can be used to indicate the placement of a design element on the FPGA die relatively to other elements. Assuming a SRL16 instance of name srl1 to be placed at location R9C0.S0, you may specify it as follows in your Verilog code:

```
// synthesis attribute RLOC of srl1 : label is
"R9C0.S0";
```

The following line will be written the output NCF file:

```
INST srl1 RLOC=R9C0.S0;
```

Example 2

The NOREDUCE constraint, available with CPLDs, prevents the optimization of the boolean equation generating a given signal. Assuming a local signal is being assigned the arbitrary function below and a NOREDUCE constraint attached to s:

```
signal s : std_logic;
attribute NOREDUCE : boolean;
```

```
attribute NOREDUCE of s : signal is “true”;
...
 s <= a or (a and b);
```

The following statements are written in the NCF file:

```
NET s NOREDUCE;
NET s KEEP;
```

Example 3

The PWR_MODE constraint, available when targeting CPLD families, controls the power consumption characteristics of macrocells. The following VHDL statement specifies that the function generating signal s should be optimized for low power consumption.

```
attribute PWR_MODE : string;
attribute PWR_MODE of s : signal is "LOW";
```

The following statement is written in the NCF file by XST:

```
NET s PWR_MODE=LOW;
NET s KEEP;
```

If the NCF attribute applies to an instance (for example, IOB, DRIVE, IOSTANDARD) and if the instance is not available (not instantiated) in the HDL source, then the HDL attribute can be applied to the signal on which XST will infer the instance.

Third Party Constraints

This section describes constraints of third-party synthesis vendors that are supported by XST. For each of the constraints, the following table gives the XST equivalent and indicates when automatic conversion is available. For information on what these constraints actually do, please refer to the corresponding vendor documentation. Note that “NA” stands for “Not Available” in the following table.

Name	Vendor	XST Equivalent	Available For
black_box	Synplicity	box_type	NA
black_box_pad_pin	Synplicity	NA	NA
black_box_tri_pins	Synplicity	NA	NA
cell_list	Synopsys	NA	NA

Name	Vendor	XST Equivalent	Available For
clock_list	Synopsys	NA	NA
Directives for inferring FF and latches	Synopsys	NA	NA
Enum	Synopsys	NA	NA
full_case	Synplicity Synopsys	full_case	Verilog
ispad	Synplicity	NA	NA
map_to_module	Synopsys	NA	NA
net_name	Synopsys	NA	NA
parallel_case	Synplicity Synopsys	parallel_case	Verilog
return_port_name	Synopsys	NA	NA
resource_sharing directives	Synopsys	resource_sharing directives	NA
set_dont_touch_network	Synopsys	NA	NA
set_dont_touch	Synopsys	NA	NA
set_dont_use_cel_name	Synopsys	NA	NA
set_prefer	Synopsys	NA	NA
state_vector	Synopsys	NA	NA
syn_encoding	Synplicity	fsm_encoding	NA
syn_hier	Synplicity	keep_hierarchy	NA
syn_isclock	Synplicity	sig_isclock	VHDL
syn_keep	Synplicity	keep*	VHDL
syn_netlist_hierarchy	Synplicity	keep_hierarchy	NA
syn_noarrayports	Synplicity	NA	NA
syn_noclockbuf	Synplicity	clock_buffer	VHDL
syn_preserve	Synplicity	NA	NA
syn_ramstyle	Synplicity	NA	NA
syn_sharing	Synplicity	resource_sharing	NA
syn_state_machine	Synplicity	fsm_extract	NA

Name	Vendor	XST Equivalent	Available For
syn_tristate	Synplicity	NA	NA
translate_off / translate_on	Synplicity / Synopsys	translate_off / translate_on	VHDL / Verilog
translate_on / translate_off	Synplicity	translate_on / translate_off	Verilog
xc_clockbuftype	Synplicity	clock_buffer	VHDL
xc_fast	Synplicity	fast	VHDL
xc_ioff	Synplicity	iob**	NA
xc_isgsr	Synplicity	NA	NA
xc_loc	Synplicity	loc	VHDL
xc_nodelay	Synplicity	nodelay	VHDL
xc_padtype	Synplicity	iostandard	NA
xc_pullup	Synplicity	pullup	VHDL
xc_pulldown	Synplicity	pulldown	VHDL
xc_slow	Synplicity	NONE	NA

* You must use the Keep constraint instead of SIGNAL_PRESERVE.

Verilog example:

```
module testkeep (in1, in2, out1);
input in1;
input in2;
output out1;

wire aux1;
wire aux2;

// synthesis attribute keep of aux1 is "true"
// synthesis attribute keep of aux2 is "true"

assign aux1 = in1;
assign aux2 = in2;
assign out1 = aux1 & aux2;

endmodule
```

The KEEP constraint can also be applied through the separate synthesis constraint file:

Example syntax:

```
attribute keep of aux1 : signal is "true";
```

These are the only two ways of preserving a signal/net in an HDL design and preventing optimization on the signal or net during synthesis.

** The IOB=TRUE constraint currently must be applied to the internal signals driven by flip-flops in order to be processed. If the constraint is placed on the port signal, it is applied to the I/O buffer.

Constraints Precedence

Priority depends on the file in which the constraint appears. A constraint in a file accessed later in the design flow overrides a constraint in a file accessed earlier in the design flow. Priority is as follows (first listed is the highest priority, last listed is the lowest).

1. Constraints in a Physical Constraints File (PCF)
2. Constraints in a User Constraints File (UCF)
3. Synthesis Constraint File
4. HDL file
5. Command Line/Process Properties dialog box in the Project Navigator

Chapter 6

VHDL Language Support

This chapter explains how VHDL is supported for XST. The chapter provides details on the VHDL language, supported constructs, and synthesis options in relationship to XST. The sections in this chapter are as follows:

- “Introduction”
- “Data Types in VHDL”
- “Objects in VHDL”
- “Operators”
- “Entity and Architecture Descriptions”
- “Combinatorial Circuits”
- “Sequential Circuits”
- “Functions and Procedures”
- “Packages”
- “VHDL Language Support”
- “VHDL Reserved Words”

For a complete specification of VHDL, refer to the IEEE VHDL Language Reference Manual.

For a detailed description of supported design constraints, refer to the “Design Constraints” chapter. For a description of the VHDL attribute syntax, see the “VHDL Attribute Syntax” section of the “Design Constraints” chapter

Introduction

VHDL is a hardware description language that offers a broad set of constructs for describing even the most complicated logic in a compact fashion. The VHDL language is designed to fill a number of requirements throughout the design process:

- Allows the description of the structure of a system—how it is decomposed into subsystems and how those subsystems are interconnected.
- Allows the specification of the function of a system using familiar programming language forms.
- Allows the design of a system to be simulated prior to being implemented and manufactured. This feature allows you to test for correctness without the delay and expense of hardware prototyping.
- Provides a mechanism for easily producing a detailed, device-dependent version of a design to be synthesized from a more abstract specification. This feature allows you to concentrate on more strategic design decisions and reduce the overall time to market for the design.

Data Types in VHDL

XST accepts the following VHDL basic types:

- Enumerated Types:
  - BIT ('0','1')
  - BOOLEAN (false, true)
  - STD_LOGIC ('U','X','0','1','Z','W','L','H','-') where:

    'U' means uninitialized

    'X' means unknown

    '0' means low

    '1' means high

    'Z' means high impedance

    'W' means weak unknown

'L' means weak low

'H' means weak high

'-' means don't care

For XST synthesis, the '0' and 'L', '1' and 'H' values are treated identically. The 'X', and '-' values are treated as don't care. The 'U' and 'W' values are not accepted by XST. The 'Z' value is treated as high impedance.

- User defined enumerated type:

  type COLOR is (RED,GREEN,YELLOW);

- Bit Vector Types:
  - BIT_VECTOR
  - STD_LOGIC_VECTOR

    Unconstrained types (types whose length is not defined) are not accepted

- Integer Type: INTEGER

The following types are VHDL predefined types:

- BIT
- BOOLEAN
- BIT_VECTOR
- INTEGER

The following types are declared in the STD_LOGIC_1164 IEEE package.

- STD_LOGIC
- STD_LOGIC_VECTOR

This package is compiled in the IEEE library. In order to use one of these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

Overloaded Data Types

The following basic types can be overloaded.

- Enumerated Types:
  - STD_ULOGIC: contains the same nine values as the STD_LOGIC type, but does not contain predefined resolution functions.
  - X01: subtype of STD_ULOGIC containing the 'X', '0' and '1' values
  - X01Z: subtype of STD_ULOGIC containing the 'X', '0', '1' and 'Z' values
  - UX01: subtype of STD_ULOGIC containing the 'U', 'X', '0' and '1' values
  - UX01Z: subtype of STD_ULOGIC containing the 'U', 'X', '0','1' and 'Z' values
- Bit Vector Types:
  - STD_ULOGIC_VECTOR
  - UNSIGNED
  - SIGNED

    Unconstrained types (types whose length is not defined) are not accepted.
- Integer Types:
  - NATURAL
  - POSITIVE

    Any integer type within a user-defined range. As an example, "type MSB is range 8 to 15;" means any integer greater than 7 or less than 16.

The types NATURAL and POSITIVE are VHDL predefined types.

The types STD_ULOGIC (and subtypes X01, X01Z, UX01, UX01Z), STD_LOGIC, STD_ULOGIC_VECTOR and STD_LOGIC_VECTOR are declared in the STD_LOGIC_1164 IEEE package. This package is compiled in the library IEEE. In order to use one of these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
```

The types UNSIGNED and SIGNED (defined as an array of STD_LOGIC) are declared in the STD_LOGIC_ARITH IEEE package. This package is compiled in the library IEEE. In order to use these types, the following two lines must be added to the VHDL specification:

```
library IEEE;
use IEEE.STD_LOGIC_ARITH.all;
```

Bi-dimensional Array Types

XST supports bi-dimensional array types. The array must be fully constrained in both dimensions. An example is shown below:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
```

The following examples demonstrate the various uses of bi-dimensional array signals and variables.

Consider the declarations:

```
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB12;
```

A bi-dimensional array signal or variable can be completely used:

```
TAB_A <= TAB_B;
```

Just an index of the first array can be specified:

```
TAB_A (8) <= WORD_A;
```

Just indexes of the first and second arrays can be specified:

```
TAB_A (8) (0) <= '1';
```

Just a slice of the first array can be specified:

```
TAB_A (11 downto 8) <= TAB_B (3 downto 0);
```

Just an index of the first array and a slice of the second array can be specified:

```
TAB_A (6) (3 downto 0) <= TAB_B (7) (4 downto 1);
```

Note also that the indices may be variable.

Objects in VHDL

VHDL objects include signals, variables and constants.

Signals can be declared in an architecture declarative part and used anywhere within the architecture. Signals can be also declared in a block and used within that block. Signals can be assigned by the assignment operator "<=".

Example:

```
signal sig1: std_logic;
sig1 <= '1';
```

Variables are declared in a process, or a subprogram, and used within that process or that subprogram. Variables can be assigned by the assignment operator ":=".

Example:

```
variable var1: std_logic_vector (7 downto 0);
var1 := "01010011";
```

Constants can be declared in any declarative region and can be used within that region. Their value cannot be changed once declared.

Example:

```
signal sig1: std_logic_vector (5 downto 0);
constant init0 : std_logic_vector (5 downto 0) :=
"010111";
sig1 <= init0;
```

Operators

Supported operators are listed in Table 6-7. This section provides an example of how to use each shift operator.

Example: sll (Shift Left Logical)

```
A(4 downto 0) sll 2 à A(2 downto 0) & "00")
```

Example: srl (Shift Right Logical)

```
A(4 downto 0) srl 2 à “00” & A(4 downto 2)
```

Example: sla (Shift Left Arithmetic)

```
A(4 downto 0) sla 2 à A(2 downto 0) & A(0) & A(0)
```

Example: sra (Shift Right Arithmetic)

```
A(4 downto 0) sra 2 à A(4) & A(4) & A(4 downto 2)
```

Example: rol (Rotate Left)

```
A(4 downto 0) rol 2 à A(2 downto 0) & A(4 downto 3)
```

Example: ror (Rotate Right)

```
A(4 downto 0) ror 2 à A(1 downto 0) & A(4 downto 2)
```

Entity and Architecture Descriptions

A circuit description consists of two parts: the interface (defining the I/O ports) and the body. In VHDL, the entity corresponds to the interface and the architecture describes the behavior.

Entity Declaration

In the entity, the I/O ports of the circuit are declared. Each port has a name, a mode (in, out, inout or buffer) and a type (ports A, B, C, D, E in the Example 6-1).

Note that types of ports must be constrained, and not more than one-dimensional array types are accepted as ports.

Architecture Declaration

In the architecture, internal signals may be declared. Each internal signal has a name and a type (signal T in Example 6-1).

Example 6-1: Entity and Architecture Declaration

```
Library IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
  port ( A,B,C : in std_logic;
         D,E : out std_logic );
end EXAMPLE;
```

```
architecture ARCHI of EXAMPLE is
  signal T : std_logic;

begin
...
end ARCHI;
```

Component Instantiation

Structural descriptions assemble several blocks and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are the component, the port and the signal. The component is the building or basic block. A port is a component I/O connector. A signal corresponds to a wire between components.

In VHDL, a component is represented by a design entity. This is actually a composite consisting of an entity declaration and an architecture body. The entity declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The architecture body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within component instantiation statements. These statements specify an instance of a component occurring inside an architecture of an other 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 following the reserved word port map) that specifies which actual signals or ports are associated with which local ports of the component declaration.

Example 6-2 gives the structural description of a half adder composed of four nand2 components.

Example 6-2: Structural Description of a Half Adder

```
entity NAND2 is
   port ( A,B : in BIT;
          Y : out BIT );
end NAND2;
architecture ARCHI of NAND2 is
begin
```

```
   Y <= A nand B;
end ARCHI;

entity HALFADDER is
   port ( X,Y : in BIT;
          C,S : out BIT );
end HALFADDER;
architecture ARCHI of HALFADDER is
   component NAND2
      port ( A,B : in BIT;
             Y : out BIT );
   end component;
   for all : NAND2 use entity work.NAND2(ARCHI);
   signal S1, S2, S3 : BIT;
begin
   NANDA : NAND2 port map (X,Y,S3);
   NANDB : NAND2 port map (X,S3,S1);
   NANDC : NAND2 port map (S3,Y,S2);
   NANDD : NAND2 port map (S1,S2,S);
   C <= S3;
 end ARCHI;
```

The synthesized top level netlist is shown in the following figure.

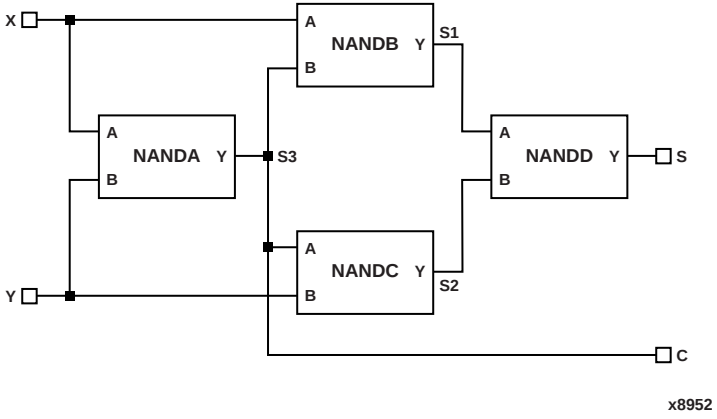


Figure 6-1 Synthesized Top Level Netlist

Component Configuration

Associating an entity / architecture pair to a component instance provides the means of linking components with the appropriate model (entity / architecture pair). XST supports component configuration in the declarative part of the architecture:

for *instantiation_list*: *component_name* use
LibName.entity_Name(*Architecture_Name)*;

Example 6-2 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);
```

which dictates that all NAND2 components will use the entity NAND2 and Architecture ARCHI.

Note When the configuration clause is missing for a component instantiation, XST links the component to the entity with the same name (and same interface) and the selected architecture to the most recently compiled architecture. If no entity / architecture is found, a black box is generated during the synthesis.

Generic Parameter Declaration

Generic parameters may also be declared in the entity declaration part. An example use of generic parameters would be setting the width of the design. In VHDL, describing circuits with generic ports has the advantage that the same component can be repeatedly instantiated with different values of generic ports as shown in Example 6-3.

Example 6-3: Generic Instantiation of Components

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

entity addern is
generic (width : integer := 8);
port (A,B : in std_logic_vector (width-1  downto  0);
      Y : out std_logic_vector (width-1 downto 0));
end addern;
architecture bhv of addern is
begin
  Y <= A + B;
end bhv;

Library IEEE;
use IEEE.std_logic_1164.all;

entity top is
  port (X, Y, Z : in std_logic_vector (12 downto 0);
      A, B : in std_logic_vector (4 downto 0);
      S :out std_logic_vector (16 downto 0) );
end top;
architecture bhv of top is
  component addern
    generic (width : integer := 8);
    port (A,B : in std_logic_vector (width-1 downto 0);
        Y : out std_logic_vector (width-1 downto 0));
  end component;

  for all : addern use entity work.addern(bhv);
  signal C1 : std_logic_vector (12 downto 0);
```

```
  signal C2, C3 : std_logic_vector (16 downto 0);
begin
  U1 : addern generic map (n=>13), port map (X,Y,C1);
  C2 <= C1 & A;
  C3 <= Z & B;
  U2 : addern generic map (n=>17), port map (C2,C3,S);
end bhv;
```

Combinatorial Circuits

The following subsections describes XST usage with various VHDL constructs for combinatorial circuits.

Concurrent Signal Assignments

Combinatorial logic may be described using concurrent signal assignments which can be defined within the body of the architecture. VHDL offers three types of concurrent signal assignments: simple, selected, and conditional. You can describe as many concurrent statements as needed; the order of concurrent signal definition in the architecture is irrelevant.

A concurrent assignment is made of two parts: left hand side, and right hand side. The assignment changes when any signal in the right part changes; in this case, the result is assigned to the signal on the left part.

Simple Signal Assignment

T <= A and B;

Selected Signal Assignment

Example 6-4: Mux Description Using Selected Signal Assignment

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

entity select_bhv is
  generic (width: integer := 8);
  port (a, b, c, d: in std_logic_vector (width-1 downto 0);
        selector: in std_logic_vector (1 downto 0);
        T: out std_logic_vector (width-1 downto 0) );
```

```
end select_bhv;
architecture bhv of select_bhv is
begin
  with selector select
    T <= a when "00",
         b when "01",
         c when "10",
         d when others;
end bhv;
```

Conditional Signal Assignment

Example 6-5: Mux Description Using Conditional Signal Assignment

```
entity when_ent is
  generic (width: integer := 8);
  port (a, b, c, d: in std_logic_vector (width-1 downto 0);
        selector: in std_logic_vector (1 downto 0);
        T: out std_logic_vector (width-1 downto 0) );
end when_ent;
architecture bhv of when_ent is
begin
  T <= a when selector = "00" else
       b when selector ="01" else
       c when selector ="10" else
       d;
end bhv;
```

Generate Statement

The 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 will be repeated N times. As an example, Example 6-6 gives the description of an 8-bit adder by declaring the bit slice structure.

Example 6-6: 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 also supported for static (non-dynamic) conditions. Example 6-7 shows such an example. It is a generic N-bit adder with a width ranging between 4 and 32.

Example 6-7: N Bit Adder Described with an "if...generate" and a "for… generate" Statement

```
entity EXAMPLE is
  generic ( N : INTEGER := 8);
  port ( A,B : in BIT_VECTOR (N downto 0);
        CIN : in BIT;
        SUM : out BIT_VECTOR (N downto 0);
        COUT : out BIT
);
end EXAMPLE;
architecture ARCHI of EXAMPLE is
  signal C : BIT_VECTOR (N+1 downto 0);
begin
  L1: if (N>=4 and N<=32) generate
    C(0) <= CIN;
    COUT <= C(N+1);
    LOOP_ADD : for I in 0 to N generate
      SUM(I) <= A(I) xor B(I) xor C(I);
      C(I+1) <= (A(I)and B(I))or (A(I) and C(I)) or (B(I) and C(I));
    end generate;
  end generate;
end ARCHI;
```

Combinatorial Process

A process assigns values to signals in a different way than when using concurrent signal assignments. The value assignments are made in a sequential mode. The latest assignments may cancel previous ones. See Example 6-8. First the signal S is assigned to 0, but later on (for (A and B) =1), the value for S is changed to 1.

Example 6-8: Assignments in a Process

```
entity EXAMPLE is
  port ( A, B : in BIT;
         S : out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( A, B )
  begin
    S <= '0' ;
    if ((A and B) = '1') then
      S <= '1' ;
    end if;
  end process;
end ARCHI;
```

A process is called combinatorial when its inferred hardware does not involve any memory elements. Said differently, when all assigned signals in a process are always explicitly assigned in all paths of the process statements, then the process in combinatorial.

A combinatorial process has a sensitivity list appearing within parenthesis after the word "process". A process is activated if an event (value change) appears on one of the sensitivity list signals. For a combinatorial process, this sensitivity list must contain all signals which appear in conditions (if, case, etc.) and any signal appearing on the right hand side of a assignment.

If one or more signals are missing from the sensitivity list, XST generates a warning for the missing signals and adds them to the sensitivity list. In this case, the result of the synthesis may be different from the initial design specification.

A process may contain local variables. The variables are handled in a similar manner as signals (but are not, of course, outputs to the design).

In Example 6-9, a variable named AUX is declared in the declarative part of the process and is assigned to a value (with ":=") in the statement part of the process. Examples 9 and 10 are two examples of a VHDL design using combinatorial processes.

Example 6-9: Combinatorial Process

```
library ASYL;
use ASYL.ARITH.all;

entity ADDSUB is
  port ( A,B : in BIT_VECTOR (3 downto 0) ;
         ADD_SUB : in BIT;
         S : out BIT_VECTOR (3 downto 0));
end ADDSUB;
architecture ARCHI of ADDSUB is
begin
  process ( A, B, ADD_SUB )
    variable AUX : BIT_VECTOR (3 downto 0);
  begin
    if ADD_SUB = '1' then
      AUX := A + B ;
    else
      AUX := A - B ;
    end if;
    S <= AUX;
  end process;
end ARCHI;
```

Example 6-10: Combinatorial Process

```
entity EXAMPLE is
  port ( A, B : in BIT;
    S : out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( A,B )
    variable X, Y : BIT;
  begin
    X := A and B;
    Y := B and A;
    if X = Y then
      S <= '1' ;
```

```
    end if;
  end process;
end ARCHI;
```

Note In combinatorial processes, if a signal is not explicitly assigned in all branches of "if" or "case" statements, XST will generate a latch to hold the last value. To avoid latch creation, assure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the process statements.

Different statements can be used in a process:

- Variable and signal assignment
- If statement
- Case statement
- For ... Loop statement
- Function and procedure call

The following sections provide examples of each of these statements.

If .. Else Statement

If ... else statements use true / false conditions to execute statements. If the expression evaluates to true, the first statement is executed. If the expression evaluates to false (or x or z), the else statement is executed. A block of multiple statements may be executed using begin and end keywords. If ... else statements may be nested.

Example 6-11: Mux Description Using If ... Else Statement

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

entity mux4 is
  port (a, b, c, d: in std_logic_vector (7 downto 0);
    sel1, sel2: in std_logic;
    outmux: out std_logic_vector (7 downto 0));
end mux4;
architecture behavior of mux4 is
begin
  process (a, b, c, d, sel1, sel2)
  begin
    if (sel1 = '1') then
```

```
      if (sel2 = '1' ) then
        outmux <= a;
      else
        outmux <= b;
      endif;
    else
      if (sel2 = '1' ) then
        outmux <= c;
      else
        outmux <= d;
      end if;
    end if;
  end process;
end behavior;
```

Case Statement

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The case statement evaluates the branches in the order they are written; the first branch that evaluates to true is executed. If none of the branches match, the default branch is executed

Example 6-12: Mux Description Using the Case Statement

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

entity mux4 is
  port (a, b, c, d: in std_logic_vector (7 downto 0);
    sel: in std_logic_vector (1 downto 0);
    outmux: out std_logic_vector (7 downto 0));
end mux4;
architecture behavior of mux4 is
begin
  process (a, b, c, d, sel)
  begin
    case sel is
      when "00" =>  outmux <= a;
      when "01" =>  outmux <= b;
      when "10" =>  outmux <= c;
      when others =>
                    outmux <= d;-- case statement must be complete
```

```
    end case;
  end process;
end behavior;
```

For .. Loop Statement

The "for" statement is supported for :

- Constant bounds
- Stop test condition using operators <, <=, > or >=
- Next step computation falling in one of the following specifications:
  - *var* = *var* + step
  - *var* = *var* - *step*

(where *var* is the loop variable and *step* is a constant value).

Example 6-13: For ... Loop Description

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

entity countzeros is
  port (a: in std_logic_vector (7 downto 0);
        Count: out std_logic_vector (2 downto 0));
end mux4;
architecture behavior of mux4 is
  signal Count_Aux: std_logic_vector (2 downto 0);
begin
  process (a)
  begin
    Count_Aux <= "000";
    for i in a'rangeloop
      if (a[i] = '0') then
        Count_Aux <= Count_Aux + 1; -- operator "+" defined
                                    --in std_logic_unsigned
      end if;
    end loop;
    Count <= Count_Aux;
  end process;
end behavior;
```

Sequential Circuits

Sequential circuits can be described using sequential processes. The following two types of descriptions are allowed by XST:

- sequential processes with a sensitivity list
- sequential processes without a sensitivity list

Sequential Process with a Sensitivity List

A process is sequential when it is not a combinatorial process. In other words, a process is sequential when some assigned signals are not explicitly assigned in all paths of the statements. In this case, the hardware generated has an internal state or memory (flip-flops or latches).

Example 6-14 provides a template for describing sequential circuits. Also refer to the chapter describing macro inference for additional details (registers, counters, etc.).

Example 6-14: Sequential Process with Asynchronous, Synchronous Parts

```
process ( CLK, RST ) ...
begin
  if RST = <'0' | ‘ 1’ > then
    -- an asynchronous part may appear here
    -- optional part
    .......
  elsif <CLK'EVENT | not CLK’ STABLE>
    and CLK = <'0' | ‘ 1’ > then
    -- synchronous part
    -- sequential statements may appear here
  end if;
end process;
```

Note Asynchronous signals must be declared in the sensitivity list. Otherwise, XST generates a warning and adds them to the sensitivity list. In this case, the behavior of the synthesis result may be different from the initial specification.

Sequential Process without a Sensitivity List

Sequential processes without a sensitivity list must contain a "wait" statement. The "wait" statement must be the first statement of the process and must be the only "wait" statement in the process. The condition in the "wait" statement must be a condition on the clock signal. Several "wait" statements in the same process are not accepted. An asynchronous part can not be specified within processes without a sensitivity list.

Example 6-15 shows the skeleton of such a process. The clock condition may be a falling or a rising edge.

Example 6-15: Sequential Process Without a Sensitivity List

```
process ...
begin
  wait until <CLK'EVENT | not CLK' STABLE> and CLK = <' 0' |'1'>;
    ... -- a synchronous part may be specified here.
end process;
```

Examples of Register and Counter Descriptions

Example 6-16 gives the description an 8-bit register using a process with a sensitivity list. In Example 6-17, the same example is described using a process without a sensitivity list containing a "wait" statement.

Example 6-16: 8 bit Register Description Using a Process with a Sensitivity List

```
entity EXAMPLE is
  port ( DI : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( CLK )
  begin
    if CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

Example 6-17: 8 bit Register Description Using a Process without a Sensitivity List

```
entity EXAMPLE is
  port ( DI : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process begin
    wait until CLK'EVENT and CLK = '1';
    DO <= DI ;
  end process;
end ARCHI;
```

Example 6-18 gives the description of an 8-bit register with a clock signal and an asynchronous reset signal.

Example 6-18: 8 bit Register Description Using a Process with a Sensitivity List

```
entity EXAMPLE is
  port ( DI : in BIT_VECTOR (7 downto 0);
         CLK : in BIT;
         RST : in BIT;
         DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( CLK, RST )
  begin
    if RST = '1' then
      DO <= "00000000";
    elsif CLK'EVENT and CLK = '1' then
      DO <= DI ;
    end if;
  end process;
end ARCHI;
```

Example 6-19: 8 bit Counter Description Using a Process with a Sensitivity List

```
library ASYL;
use ASYL.PKG_ARITH.all;

entity EXAMPLE is
  port ( CLK : in BIT;
         RST : in BIT;
         DO : out BIT_VECTOR (7 downto 0) );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process ( CLK, RST )
    variable COUNT : BIT_VECTOR (7 downto 0);
  begin
    if RST = '1' then
      COUNT := "00000000";
    elsif CLK'EVENT and CLK = '1' then
      COUNT := COUNT + "00000001";
    end if;
    DO <= COUNT;
  end process;
end ARCHI;
```

Multiple Wait Statements Descriptions

Sequential circuits can be described with multiple wait statements in a process. When using XST, several rules must be respected to use multiple wait statements. These rules are as follows:

- The process must only contain one loop statement.
- The first statement in the loop must be a wait statement.
- After each wait statement, a next or exit statement must be defined.
- The condition in the wait statements must be the same for each wait statement.
- This condition must use only one signal—the clock signal.
- This condition must have the following form:

```
"wait [on <clock_signal>] until [(<clock_signal>'EVENT |
    not <clock_signal>'STABLE) and ] <clock_signal> = <'0' | '1'>;"
```

Example 6-20 uses multiple wait statements. This example describes a sequential circuit performing four different operations in sequence. The design cycle is delimited by two successive rising edges of the clock signal. A synchronous reset is defined providing a way to restart the sequence of operations at the beginning. The sequence of operations consists of assigning each of the four inputs: DATA1, DATA2, DATA3 and DATA4 to the output RESULT.

Example 6-20: Sequential Circuit Using Multiple Wait Statements

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity EXAMPLE is
  port ( DATA1, DATA2, DATA3, DATA4 : in
   STD_LOGIC_VECTOR (3 downto 0);
        RESULT : out STD_LOGIC_VECTOR (3 downto 0);
        CLK : in STD_LOGIC;
        RST : in STD_LOGIC := '0' );
end EXAMPLE;
architecture ARCH of EXAMPLE is
begin
  process begin
    SEQ_LOOP : loop
      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA1;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA2;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA3;

      wait until CLK'EVENT and CLK = '1';
      exit SEQ_LOOP when RST = '1';
      RESULT <= DATA4;
    end loop;
```

```
  end process;
end ARCH;
```

Functions and Procedures

The declaration of a function or a procedure provides a mechanism for handling blocks used multiple times in a design. Functions and procedures can be declared in the declarative part of an entity, in an architecture, or in packages. The heading part contains the parameters: input parameters for functions and input, output and inout parameters for procedures. These parameters can be unconstrained; it means that they are not constrained to a given bound. The content is similar to the combinatorial process content.

Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package.

Recursive function and procedure calls are also not supported.

Example 6-21 shows a function declared within a package. The "ADD" function declared here is a single bit adder. This function is called 4 times with the proper parameters in the architecture to create a 4-bit adder. The same example described using a procedure is shown in Example 6-22.

Example 6-21: Function Declaration and Function Call

```
package PKG is
  function ADD  (A,B, CIN : BIT )
  return BIT_VECTOR;
end PKG;

package body PKG is
  function ADD  (A,B, CIN : BIT )
  return BIT_VECTOR is
    variable S, COUT : BIT;
    variable RESULT : BIT_VECTOR (1 downto 0);
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    RESULT := COUT & S;
    return RESULT;
  end ADD;
end PKG;
```

```
use work.PKG.all;

entity EXAMPLE is
  port ( A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT: out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
  signal S0, S1, S2, S3 : BIT_VECTOR (1 downto 0);
  begin
    S0 <= ADD ( A(0), B(0), CIN );
    S1 <= ADD ( A(1), B(1), S0(1) );
    S2 <= ADD ( A(2), B(2), S1(1) );
    S3 <= ADD ( A(3), B(3), S2(1) );
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3(1);
end ARCHI;
```

Example 6-22: Procedure Declaration and Procedure Call

```
package PKG is
  procedure ADD
    (A,B, CIN : in BIT;
      C : out BIT_VECTOR (1 downto 0) );
end PKG;
package body PKG is
  procedure ADD
    (A,B, CIN : in BIT;
      C : out BIT_VECTOR (1 downto 0) ) is
    variable S, COUT : BIT;
  begin
    S := A xor B xor CIN;
    COUT := (A and B) or (A and CIN) or (B and CIN);
    C := COUT & S;
  end ADD;
end PKG;

use work.PKG.all;
```

```
entity EXAMPLE is
  port ( A,B : in BIT_VECTOR (3 downto 0);
    CIN : in BIT;
    S : out BIT_VECTOR (3 downto 0);
    COUT : out BIT );
end EXAMPLE;
architecture ARCHI of EXAMPLE is
begin
  process (A,B,CIN)
    variable S0, S1, S2, S3 : BIT_VECTOR (1 downto
   0);
  begin
    ADD ( A(0), B(0), CIN, S0 );
    ADD ( A(1), B(1), S0(1), S1 );
    ADD ( A(2), B(2), S1(1), S2 );
    ADD ( A(3), B(3), S2(1), S3 );
    S <= S3(0) & S2(0) & S1(0) & S0(0);
    COUT <= S3(1);
  end process;
end ARCHI;
```

Packages

VHDL models may be defined using packages. Packages contain type and subtype declarations, constant definitions, function and procedure definitions, and component declarations.

This mechanism provides the ability to change parameters and constants of the design (for example, constant values, function definitions). Packages may contain two declarative parts: package declaration and body declaration. The body declaration includes the description of function bodies declared in the package declaration.

XST provides full support for packages. To use a given package, the following lines must be included at the beginning of the VHDL design:

```
library lib_pack; -- lib_pack is the name of the library specified
where the package has been compiled (work by default)
use lib_pack.pack_name.all; -- pack_name is the name of the defined
package.
```

XST also supports predefined packages; these packages are pre-compiled and can be included in VHDL designs. These packages

are intended for use during synthesis, but may also used for simulation.

STANDARD Package

The Standard package contains basic types (bit, bit_vector, and integer). The STANDARD package is included by default.

IEEE Packages

The following IEEE packages are supported.

- std_logic_1164: defines types std_logic, std_ulogic, std_logic_vector, std_ulogic_vector, and conversion functions based on these types.
- std_logic_arith: supports types unsigned, signed vectors, and all overloaded arithmetic operators on these types. It also defines conversion and extended functions for these types.
- std_logic_unsigned: defines arithmetic operators on std_ulogic_vector and considers them as unsigned operators.
- std_logic_signed: defines arithmetic operators on std_logic_vector and considers them as signed operators.
- std_logic_misc: defines supplemental types, subtypes, constants, and functions for the std_logic_1164 package (and_reduce, or_reduce, ...)

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.
```

IEEE Numeric Packages

Numeric packages are the new IEEE standard packages for synthesis.

- numeric_bit: supports types unsigned, signed vectors based on type bit, and all overloaded arithmetic operators on these types. It also defines conversion and extended functions for these types.
- numeric_std: supports types unsigned, signed vectors based on type std_logic. This package is equivalent to std_logic_arith.
- numeric_unsigned: defines arithmetic operators on std_ulogic_vector and considers them as unsigned operators. This package is equivalent to std_logic_unsigned.
- numeric_signed: defines arithmetic operators on std_logic_vector and considers them as signed operators. This package is equivalent to std_logic_signed.
- numeric_extra: defines supplemental functions for the std_logic_1164 package (and_reduce, or_reduce, ...). This package is equivalent to std_logic_misc.

VHDL Language Support

The following tables indicate which VHDL constructs are supported in VHDL. For more information about these constructs, refer to the sections following the tables.

Table 6-1 Design Entities and Configurations

Entity Header	Generics	Supported
	Ports	Supported (no unconstrained ports)
	Entity Declarative Part	Supported
	Entity Statement Part	Unsupported
Architecture Bodies	Architecture Declarative Part	Supported
	Architecture Statement Part	Supported
Configuration Declarations	Block Configuration	Ignored
	Component Configuration	Ignored

Table 6-1 Design Entities and Configurations

Subprograms	Functions	Supported
	Procedures	Supported
Packages	STANDARD	Types TIME and REAL are not supported
	TEXTIO	Unsupported
	STD_LOGIC_1164	Supported
	STD_LOGIC_ARITH	Supported
	STD_LOGIC_SIGNED	Supported
	STD_LOGIC_UNSIGNED	Supported
	STD_LOGIC_MISC	Supported
	NUMERIC_BIT	Supported
	NUMERIC_EXTRA	Supported
	NUMERIC_SIGNED	Supported
	NUMERIC_UNSIGNED	Supported
	NUMERIC_STD	Supported
	ASYL.ARITH	Supported
	ASYL.SL_ARITH	Supported
	ASYL.PKG_RTL	Supported
	ASYL.ASYL1164	Supported
Enumeration Types	BOOLEAN, BIT	Supported
	STD_ULOGIC, STD_LOGIC	Supported
	XO1, UX01, XO1Z, UX01Z	Supported
	Character	Supported
Integer Types	INTEGER	Supported
	POSITIVE	Supported
	NATURAL	Supported
	Physical	Unsupported
	Floating	Unsupported

Table 6-1 Design Entities and Configurations

Composite	BIT_VECTOR	Supported
	STD_ULOGIC_VECTOR	Supported
	STD_LOGIC_VECTOR	Supported
	UNSIGNED	Supported
	SIGNED	Supported
	Record	Unsupported
	Access	Unsupported
	File	Unsupported

Table 6-2 Mode

In, Out, Inout	Supported
Buffer	Supported
Linkage	Unsupported

Table 6-3 Declarations

Type	Supported for enumerated types, types with positive range having constant bounds, bit vector types and bi-dimensional arrays
Subtype	Supported

Table 6-4 Objects

Constant Declaration	Supported (deferred constants are not supported)
Signal Declaration	Supported (“register” or “bus” type signals are not supported)
Variable Declaration	Supported (no initial value except in functions and procedures)
File Declaration	Unsupported
Alias Declaration	Supported
Attribute Declaration	Supported for some attributes, otherwise skipped (see the “Design Constraints” chapter)
Component Declaration	Supported

Table 6-5 Specifications

Attribute	Only supported for some predefined attributes: HIGH, LOW, LEFT, RIGHT, RANGE, REVERSE_RANGE, LENGTH, POS, ASCENDING, EVENT, STABLE, LAST_VALUE, DRIVING_VALUE. Otherwise, ignored.
Configuration	Supported only with the “all” clause for instances list. If no clause is added, XST looks for the entity / architecture compiled in the default library.
Disconnection	Unsupported

Table 6-6 Names

Simple Names	Supported
Selected Names	Supported
Indexed Names	Supported
Slice Names	Supported for constant ranges (ranges with constant bounds)

Note XST does not allow underscores in signal names (for example, _DATA_1).

Table 6-7 Expressions

Operators	Logical Operators: and, or, nand, nor, xor, xnor, not	Supported
	Relational Operators: =, /=, <, <=, >, >=	Supported
	& (concatenation)	Supported
	Adding Operators: +, -	Supported
	*	Supported
	/, mod, rem	Supported if the right operand is a constant power of 2
	Shift Operators: sll, srl, sla, sra, rol, ror	Supported
	abs	Supported for constant operands
	**	Only supported if the left operand is 2
	Sign: +, -	Supported
Operands	Abstract Literals	Only integer literals are supported
	Physical Literals	Ignored
	Enumeration Literals	Supported
	String Literals	Supported
	Bit String Literals	Supported
	Record Aggregates	Unsupported
	Array Aggregates	Supported
	Function Call	Supported
	Qualified Expressions	Supported for accepted predefined attributes
	Types Conversions	Unsupported
	Allocators	Unsupported
	Static Expressions	Supported

Table 6-8 Sequential Statements

Wait Statement	Wait on *sensitivity_list* until *Boolean_expression*. See the "Sequential Circuits" section for details.	Supported with one signal in the sensitivity list and in the Boolean expression. In case of multiple wait statements, the sensitivity list and the Boolean expression must be the same for each wait statement.
	Wait for t*ime_expression*. . See the "Sequential Circuits" section for details.	Unsupported
	Assertion Statement	Ignored
	Signal Assignment Statement	Supported (delay is ignored)
	Variable Assignment Statement	Supported
	Procedure Call Statement	Supported
	If Statement	Supported
	Case Statement	Supported
Loop Statement	"for ... loop ... end ... loop"	Supported for constant bounds only
	"while ... loop ... end loop"	Unsupported
	"loop ... end loop"	Only supported in the particular case of multiple wait statements
	Next Statement	Only supported in a loop statement in case of multiple wait statements
	Exit Statement	Only supported in a loop statement in case of multiple wait statements
	Return Statement	Only supported in function bodies
	Null Statement	Supported

Table 6-8 Sequential Statements

Concurrent Statement	Process Statement	Supported
	Concurrent Procedure Call	Supported
	Concurrent Assertion Statement	Ignored
	Concurrent Signal Assignment Statement	Supported (no “after” clause, no “transport” or “guarded” options, no waveforms)
	Component Instantiation Statement	Supported
	“For ... Generate”	Statement supported for constant bounds only
	“If ... Generate”	Statement supported for static condition only

VHDL Reserved Words

The following table shows the VHDL reserved words.

abs	configuration	impure	null	rem	type
access	constant	in	of	report	unaffected
after	disconnect	inertial	on	return	units
alias	downto	inout	open	rol	until
all	else	is	or	ror	use
and	elsif	label	others	select	variable
architecture	end	library	out	severity	wait
array	entity	linkage	package	signal	when
assert	exit	literal	port	shared	while
attribute	file	loop	postponed	sla	with
begin	for	map	procedure	sll	xnor
block	function	mod	process	sra	xor
body	generate	nand	pure	srl	
buffer	generic	new	range	subtype	
bus	group	next	record	then	
case	guarded	nor	register	to	
component	if	not	reject	transport	

Chapter 7

Verilog Language Support

This chapter contains the following sections.

- “Introduction”
- “Behavioral Features of Verilog”
- “Structural Verilog Features”
- “Parameters”
- “Verilog Limitations in XST”
- “Verilog Meta Comments”
- “Language Support Tables”
- “Primitives”
- “Verilog Reserved Keywords”

For detailed information about Verilog design constraints and options, refer to the “Design Constraints” chapter. For information about the Verilog attribute syntax, see the “Verilog Meta Comment Syntax” section of the “Design Constraints” chapter.

For information on setting Verilog options in the Process window of the Project Navigator, refer to the “Setting Constraints and Options” section of the “Design Constraints” chapter.

Introduction

Complex circuits are commonly designed using a top down methodology. Various specification levels are required at each stage of the design process. As an example, at the architectural level, a specification may correspond to a block diagram or an Algorithmic State Machine (ASM) chart. A block or ASM stage corresponds to a register transfer block (register, adder, counter, multiplexer, glue logic, finite state machine, for example.) where the connections are N-bit wires. Use of an HDL language like Verilog allows expressing notations such as ASM charts and circuit diagrams in a computer language. Verilog provides both behavioral and structural language structures which allow expressing design objects at high and low levels of abstraction. Designing hardware with a language like Verilog allows usage of software concepts such as parallel processing and object-oriented programming. Verilog has a syntax similar to C and Pascal and is supported by XST as IEEE 1364.

The Verilog support in XST provides an efficient way to describe both the global circuit and each block according to the most efficient "style". Synthesis is then performed with the best synthesis flow for each block. Synthesis in this context is the compilation of high-level behavioral and structural Verilog HDL statements into a flattened gate-level netlist which can then be used to custom program programmable logic device such as the Virtex FPGA family from Xilinx. Different synthesis methods will be used for arithmetic blocks, glue logic, and finite state machines.

This manual assumes that you are familiar with the basic notions of Verilog. Please refer to the IEEE Verilog HDL Reference Manual for a complete specification.

Behavioral Features of Verilog

This section contains descriptions of the behavioral features of Verilog.

Variable Declaration

Variables in Verilog may be declared as integers or real. These declarations are intended only for use in test code. Verilog provides data types such as reg and wire for actual hardware description.

The difference between reg and wire is whether the variable is given its value by behavioral (reg) or structural (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.

Example:

```
reg [3:0] arb_priority;
wire [31:0] arb_request;
```

where arb_request[31] is the MSB and arb_request[0] is the LSB.

Verilog supports concatenation of bits to form a wider wire or reg. Example {arb_priority[2], arb_priority[1]} is a two bit reg. Verilog also allows assignments to a set of bits from a declared reg or wire.

Example:

```
arb_priority[2:1] = arb_request[1:0];
```

Verilog allows arrays of reg and wires to be defined as follows:

```
reg [3:0] mem_array [31:0];
```

describes an array of 32 Elements each 4 bits wide which can be assigned via behavioral verilog code or

```
wire [7:0] mem_array [63:0];
```

describes an array of 64 elements each 8 bits wide which can only be assigned via structural Verilog code.

Note XST does not allow underscores in signal names (for example, _DATA_1).

Data Types

The Verilog representation of the bit data type contains the following four values:

- 0: logic zero
- 1: logic one
- x: unknown logic value
- z: high impedance

XST includes support for the following Verilog data types:

- Net: wire, tri, triand / wand, trior / wor
- Registers: reg, integer
- Supply nets: supply0, supply1
- Constants: parameter
- Memories

Net and registers can be either single bit (scalar) or multiple bit (vectors).

Example 3-1 gives some examples of Verilog data types (as found in the declaration section of a Verilog module).

Example 3-1: Basic Data Types

```
wire net1;        // single bit net
reg r1;           // single bit register
tri [7:0] bus1;   // 8 bit tristate bus
reg [15:0] bus1;  // 15 bit register
reg [7:0] mem[0:127];   // 8x128 memory register
parameter state1 = 3'b001;  // 3 bit constant
```

Legal Statements

The following are statements that are legal in behavioral Verilog.

Variable and signal assignment:

- Variable = expression
- if (condition) statement
- if (condition) else statement

- case (expression)

  constant: statement

  …

  default: statement

  endcase

- for (variable = expression; condition; variable = variable + expression) statement
- while (condition) statement
- forever statement
- functions and tasks

Note All variables are declared as integer or reg. A variable cannot be declared as a wire.

Expressions

An expression involves constants and variables with arithmetic (+, -, *, /, %), logical (&, &&, |, | |, ^, ~, <<, >>), relational (<, ==, ===, <=, >=, !=, !==, >) and conditional (?) operators. The logical operators are further divided as bit-wise versus logical depending on whether it is applied to an expression involving several bits or a single bit. The following table lists the expressions supported by XST.

Table 7-1 Expressions

Concatenation	{}	Supported
Replication	{{}}	Supported
Arithmetic	+, -, *	Supported
	/	Supported only if second operand is a power of 2
Modulus	%	Supported only if second operand is a power of 2
Addition	+	Supported
Subtraction	-	Supported
Multiplication	*	Supported

Table 7-1 Expressions

Division	/	Supported XST generates incorrect logic for the division operator between signed and unsigned constants. Example: -1235 / 3'bill
Remainder	%	Supported
Relational	>, <, >=, <=	Supported
Logical Negation	!	Supported
Logical AND	&&	Supported
Logical OR	||	Supported
Logical Equality	==	Supported
Logical Inequality	!=	Supported
Case Equality	===	Unsupported
Case Inequality	!==	Unsupported
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	>>	Supported
Conditional	?:	Supported
Event OR	or	Supported

Note The (===) and (!==) are special comparison operators useful in simulations to check if a variable is assigned a value of (x) or (z). They have no meaning in hardware.

Table 7-2 Results of Evaluating Expressions

a b	a==b	a===b	a!=b	a!==b	a&b	a&&b	a|b	a||b	a^b
0 0	1	1	0	0	0	0	0	0	0
0 1	0	0	1	1	0	0	1	1	1
0 x	x	0	x	1	0	0	x	x	x
0 z	x	0	x	1	0	0	x	x	x
1 0	0	0	1	1	0	0	1	1	1
1 1	1	1	0	0	1	1	1	1	0
1 x	x	0	x	1	x	x	1	1	x
1 z	x	0	x	1	x	x	1	1	x
x 0	x	0	x	1	0	0	x	x	x
x 1	x	0	x	1	x	x	1	1	x
x x	x	1	x	0	x	x	x	x	x
x z	x	0	x	1	x	x	x	x	x
z 0	x	0	x	1	0	0	x	x	x
z 1	x	0	x	1	x	x	1	1	x
z x	x	0	x	1	x	x	x	x	x
z z	x	1	x	0	x	x	x	x	x

Blocks

Block statements are used to group statements together. XST only supports sequential blocks. Within these blocks, the statements are executed in the order listed. Parallel blocks are not supported by XST. Block statements are designated by **`begin`** and **`end`** keywords, and are discussed within examples later in this chapter.

Modules

In Verilog a design component is represented by a module. The connections between components are specified within module instantiation statements. Such a statement specifies an instance of a module. Each module instantiation statement must be given a name (instance name). In addition to the name, a module instantiation statement contains an association list that specifies which actual nets or ports are associated with which local ports (formals) of the module declaration.

All procedural statements occur in blocks that are defined inside modules. There are two kinds of procedural blocks: the initial block and the always block. Within each block, Verilog uses a begin and end to enclose the statements. Since initial blocks are ignored during synthesis, only always blocks are discussed. Always blocks usually take the following format:

```
always
  begin
  statement
  .....
end
```

where each statement is a procedural assignment line terminated by a semicolon.

Module Declaration

In the module declaration, the I/O ports of the circuit are declared. Each port has a name and a mode (in, out, and inout) as shown in the example below. Following is an example of a module declaration

```
module EXAMPLE (A, B, C, D, E);
 input A, B, C;
 output D;
 inout E;
 wire D, E;
  ...
  assign E = oe ? A : 1'bz;
  assign D = B & E;
  ...
endmodule
```

The input and output ports defined in the module declaration called EXAMPLE are the basic input and output I/O signals for the design. The inout port in Verilog is analogous to a bi-directional I/O pin on the device with the data flow for output versus input being controlled by the enable signal to the tristate buffer. The preceding example describes E as a tristate buffer with a high-true output enable signal. If oe = 1, the value of signal A will be output on the pin represented by E. If oe = 0, then the buffer is in high impedance (Z) and any input value driven on the pin E (from the external logic) will be brought into the device and fed to the signal represented by D.

Verilog Assignments

There are two forms of assignment statements in the Verilog language:

- Continuous Assignments
- Procedural Assignments

Continuous Assignments

Continuous assignments are used to model combinatorial logic in a concise way. Both explicit and implicit continuous assignments are supported. Explicit continuous assignments are introduced by the **`assign`** keyword after the net has been separately declared. Implicit continuous assignments combine declaration and assignment.

Note Delays and strengths given to a continuous assignment are ignored by XST.

Example of an explicit continuous assignment:

```
wire par_eq_1;
…..
assign par_eq_1 = select ? b : a;
```

Example of an implicit continuous assignment:

```
wire temp_hold = a | b;
```

Note Continuous assignments are only allowed on wire and tri data types.

Procedural Assignments

Procedural assignments are used to assign values to variables declared as regs and are introduced by always blocks, tasks, and functions. Procedural assignments are usually used to model registers and FSMs.

XST includes support for combinatorial functions, combinatorial and sequential tasks, and combinatorial and sequential always blocks.

Combinatorial always blocks

Combinatorial logic can be modeled efficiently using two forms of time control, namely the # and @ Verilog time control statements. The # time control is ignored for synthesis and hence this section describes modeling combinatorial logic with the @ statement.

A combinatorial always block has a sensitivity list appearing within parenthesis 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 contains all signals which appear in conditions (if, case, for example) and any signal appearing on the right hand side of an assignment.

Note In combinatorial processes, if a signal is not explicitly assigned in all branches of “if" or "case" statements, XST will generate a latch to hold the last value. To avoid latch creation, assure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the process statements.

Different statements can be used in a process:

- Variable and signal assignment
- If... else statement
- Case statement
- For loop statement
- Function and task call

The following sections provide examples of each of these statements.

if ... else statement

If ... else statements use true / false conditions to execute statements. If the expression evaluates to true, the first statement is executed. If the expression evaluates to false (or x or z), the else statement is executed. A block of multiple statements may be executed using begin and end keywords. If ... else statements may be nested.

Example 3-2: Mux Description Using If .. Else Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
  begin
    if (sel[1])
      if (sel[0])
        outmux = d;
      else
        outmux = c;
    else
      if (sel[0])
        outmux = b;
      else
        outmux = a;
    end
endmodule
```

case statement

case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The case statement evaluates the branches in the order they are written; the first branch that evaluates to true is executed. If none of the branches match, the default branch is executed.

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 any of the preceding case statements.

Example 3-3: Mux Description Using CASE Statement

```
module mux4 (sel, a, b, c, d, outmux);
input [1:0] sel;
input [1:0] a, b, c, d;
output [1:0] outmux;
reg [1:0] outmux;

always @(sel or a or b or c or d)
  begin
    case (sel)
      2’b00:  outmux = a;
      2’b01:  outmux = b;
      2’b10:  outmux = c;
      default: outmux = d;
    endcase
  end
endmodule
```

The preceding CASE statement will evaluate the values of the input sel in priority order. To avoid priority processing, it is recommended that you use a parallel-case Verilog meta comment which will ensure parallel evaluation of the sel inputs.

Example:

```
always @(sel or a or b or c or d) //synthesis
   parallel_case
```

for and repeat loops

When using always blocks, repetitive or bit slice structures can also be described using the "for" statement or the "repeat" statement.

The "for" statement is supported for:

- Constant bounds
- Stop test condition using operators <, <=, > or >=

- Next step computation falling in one of the following specifications:
  - *var* = *var* + step
  - *var* = *var* - *step*

(where *var* is the loop variable and *step* is a constant value).

The repeat statement is only supported for constant values.

Example 3-4: For Loop Description

```
module countzeros (a, Count);
input [7:0] a;
output [2:0] Count;
reg [2:0] Count;
reg [2:0] Count_Aux;
integer i;

always @(a)
  begin
    Count_Aux = 3'b0;
  for (i = 0; i < 8; i = i+1)
    begin
      if (!a[i])
        Count_Aux = Count_Aux+1;
    end
  Count = Count_Aux;
  end

endmodule
```

Sequential Always Blocks

Sequential circuit description is based on always blocks with a sensitivity list.

The sensitivity list contains a maximum of three edge-triggered events: the clock signal event (which is mandatory) and possibly a reset signal event and a set signal event. One and only one "if-else" statement is accepted in such an always block.

An asynchronous part may appear before the synchronous part in the first and the second branch of the "if-else" statement. Signals assigned in the asynchronous part must be assigned to the constant values ’0’, ’1’, ’X’ or ’Z’ or any vector composed of these values.

These same signals must also be assigned in the synchronous part (that is, the last branch of the "if-else" statement). The clock signal condition is the condition of the last branch of the "if-else" statement.

Example 3-5: 8 Bit Register Using an Always Block

```
module seq1 ( DI, CLK, DO);
  input [7:0] DI;
  input CLK;
  output [7:0] DO;
  reg [7:0] DO;

  always @( posedge CLK )
    DO = DI ;

endmodule
```

Example 3-6 gives the description of an 8-bit register with a clock signal and an asynchronous reset signal. Example 3-7 describes an 8-bit counter.

Example 3-6: 8 Bit Register with Asynchronous Reset (high-true) Using an Always Block

```
module EXAMPLE ( DI, CLK, RST, DO);
  input [7:0] DI;
  input CLK, RST;
  output [7:0] DO;
  reg [7:0] DO;

  always @( posedge CLK or posedge RST)
    if (RST == 1’b1)
      DO = 8’b00000000;
    else
      DO = DI;
endmodule
```

Example 3-7: 8 Bit Counter with Asynchronous Reset (low-true) Using an Always Block

```
module seq2 ( CLK, RST, DO);
  input CLK, RST;
  output [7:0] DO;
  reg [7:0] DO;

  always @( posedge CLK or posedge RST )
    if (RST == 1'b1)
      DO = 8'b00000000;
    else
      DO = DO + 8'b00000001;
endmodule
```

Assign and Deassign Statements

Assign and deassign statements are supported within simple templates.

The following is an example of the general template for assign / deassign statements:

```
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'b 0;
  end else
  begin
    deassign STATE;
  end

always @ ( posedge CLOCK )
```

```
  begin
    STATE = DATA_IN;
  end

endmodule
```

Main limitations on support of the assign / deassign statement in XST are as follows:

- For a given signal, there must be only one assign / deassign statement. For example, the following design will be rejected:

```
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'b 0;
   else
     deassign STATE;

always @ ( SET ) // block b1
   if( SET )
     assign STATE  = 1'b 1;
   else
     deassign STATE;

always @ ( posedge CLOCK ) // block b2
   begin
     STATE = DATA_IN;
   end

endmodule
```

- The assign / deassign statement must be performed in the same always block through an if / else statement. For example, the following design will be rejected:

```
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'b 0;
   2'b01:     assign STATE = 1'b 0;
   2'b10:     assign STATE = 1'b 1;
   2'b11:     deassign STATE;
endcase

always @ ( posedge CLOCK ) // block b2
begin
   STATE = DATA_IN;
end

endmodule
```

- You cannot assign a bit / part select of a signal through an assign / deassign statement. For example, the following design will be rejected:

```
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'b 0;
      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
```

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 content is similar to the combinatorial always block content. Recursive function and task calls are not supported.

Example 3-8 shows a function declared within a module. The ADD function declared is a single-bit adder. This function is called 4 times with the proper parameters in the architecture to create a 4-bit adder. The same example, described with a task, is shown in Example 3-9.

Example 3-8: Function Declaration and Function Call

```
module comb15 ( A, B, CIN, S, COUT);
  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;
  wire [1:0] S0, S1, S2, S3;
  function [1:0] ADD;
    input A, B, CIN;
    reg S, COUT;
    begin
      S = A ^ B ^ CIN;
      COUT = (A&B) | (A&CIN) | (B&CIN);
      ADD = {COUT, S};
    end
  endfunction
```

```
  assign S0 = ADD ( A[0], B[0], CIN),
    S1 = ADD ( A[1], B[1], S0[1]),
    S2 = ADD ( A[2], B[2], S1[1]),
    S3 = ADD ( A[3], B[3], S2[1]),
    S = {S3[0], S2[0], S1[0], S0[0]},

    COUT = S3[1];
endmodule
```

Example 3-9: Task Declaration and Task Enable

```
module EXAMPLE ( A, B, CIN, S, COUT);
  input [3:0] A, B;
  input CIN;
  output [3:0] S;
  output COUT;
  reg [3:0] S;
  reg COUT;
  reg [1:0] S0, S1, S2, S3;

  task ADD;
    input A, B, CIN;
    output [1:0] C;
    reg [1:0] C;
    reg S, COUT;

    begin
      S = A ^ B ^ CIN;
      COUT = (A&B) | (A&CIN) | (B&CIN);
      C = {COUT, S};
    end
  endtask

  always @( A or B or CIN)
  begin
    ADD ( A[0], B[0], CIN, S0);
    ADD ( A[1], B[1], S0[1], S1);
    ADD ( A[2], B[2], S1[1], S2);
    ADD ( A[3], B[3], S2[1], S3);
    S = {S3[0], S2[0], S1[0], S0[0]};
    COUT = S3[1];
```

```
  end

endmodule
```

Blocking Versus Non-Blocking Procedural Assignments

The # and @ time control statements delay execution of the statement following them until the specified event is evaluated as true. Use of blocking and non-blocking procedural assignments have time control built into their respective assignment statement.

The # delay is ignored for synthesis.

The syntax for a blocking procedural assignment is shown in the following example:

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 at the time when the statement executes but allow other statements in the same process to execute as well at the same time. The variable change only occurs after the specified delay.

The syntax for a non-blocking procedural assignment is as follows:

```
variable <= @(posedge or negedge bit) expression;
```

Example:

```
if (in1) out <= 1'b1;
else out <= in2;
```

Constants, Macros, Include Files and Comments

This section discusses constants, macros, include files, and comments

Constants

By default, constants in Verilog are assumed to be decimal integers. They can be specified explicitly in binary, octal, decimal or hexadecimal by prefacing them with the appropriate syntax.

Example: 4'b1010, 4'o12, 4'd10 and 4'ha all represent the same value.

Macros

Verilog provides a way to define macros as shown in the following example:

```
`define TESTEQ1 4'b1101
```

Later in the design code a reference to the defined macro is made as follows:

```
if (request == `TESTEQ1)
```

Example:

```
'define myzero 0
assign mysig = 'myzero;
```

Verilog provides the 'ifdef and 'endif constructs to determine whether a macro is defined or not. These constructs are used to define conditional compilation. If the macro called out by the 'ifdef command has been defined, that code will be compiled. If not, the code following the 'else command is compiled. The 'else is not required, but the 'endif must complete the conditional statement.

Example:

```
'ifdef MYVAR
module if_MYVAR_is_declared;
...
endmodule
'else
module if_MYVAR_is_not_declared;
...
endmodule
'endif
```

Include Files

Verilog allows separating source code into more than one file. To use the code contained in another file, the current file has the following syntax:

```
‘include "path/file-name-to-be-included"
```

Multiple ‘include statements are allowed in a single Verilog file. This is a great feature to make code modular and manageable in a team design environment where different files describe different modules of the design.

If files are referenced by an ‘include statement, they must not be manually added to the project. For example, at the top of a Verilog file you might see this:

```
‘timescale 1ns/1ps
‘include "modules.v"
...
```

If the specified file (in this case, modules.v) has been added to a Foundation ISE project *and* is specified with an ‘include, conflicts will occur and an error message displays:

```
ERROR:   (VLG__5002). fifo.v Line 2. Duplicate
   declarations of module
’RAMB4_S8_S8’
```

Comments

There are two forms of comments in Verilog similar to the two forms found in a language like C++.

- // Allows definition of a one-line comment.
- /* You can define a multi-line comment by enclosing it as illustrated by this sentence*/

Structural Verilog Features

Structural Verilog descriptions assemble several blocks of code and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are the module, the port and the signal. The component is the building or basic block. A port is a component I/O connector. A signal corresponds to a wire between components.

In Verilog, a component is represented by a design module. The module declaration provides the "external" view of the component; it describes what can be seen from the outside, including the component ports. The module body provides an "internal" view; it describes the behavior or the structure of the component.

The connections between components are specified within component instantiation statements. These statements specify an instance of a component occurring within another component or the circuit. Each component instantiation statement is labeled with an identifier. Besides naming a component declared in a local component declaration, a component instantiation statement contains an association list (the parenthesized list) that specifies which actual signals or ports are associated with which local ports of the component declaration.

The Verilog language provides a large set of built-in logic gates which can be instantiated to build larger logic circuits. The set of logical functions described by the built-in gates include AND, OR, XOR, NAND, NOR and NOT.

Here is an example of building a basic XOR function of two single bit inputs a and b.

```
module build_xor (a, b, c);
 input a, b;
 output c;
 wire c, a_not, b_not;
  not a_inv (a_not, a);
  not b_inv (b_not, b);
  and a1 (x, a_not, b);
  and a2 (y, b_not, a);
  or out (c, x, y);
endmodule
```

Each instance of the built-in modules has a unique instantiation name such as a_inv, b_inv, out. The wiring up of the gates describes an XOR gate in structural Verilog.

Example 3-12 gives the structural description of a half adder composed of four, 2 input nand modules.

Example 3-12: Structural Description of a Half Adder

```
module halfadd (X, Y, C, S);
input X, Y;
output C, S;
wire S1, S2, S3;
nand NANDA (S3, X, Y);
nand NANDB (S1, X, S3);
nand NANDC (S2, S3, Y);
nand NANDD (S, S1, S2);
assign C = S3;

endmodule
```

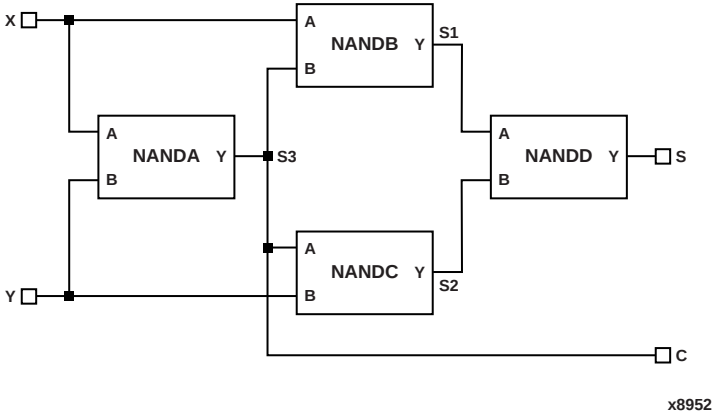


Figure 7-1 Synthesized Top Level Netlist

The structural features of Verilog HDL also allow designing circuits by instantiating pre-defined primitives such as gates, registers and Xilinx specific primitives like CLKDLL and BUFGs. These primitives are other than those included in the Verilog language. These pre-defined primitives are supplied with the XST Verilog libraries (virtex.v, xc9500.v, spartan2.v).

XST also allows instantiation of complex pre-defined macros from a user developed library to build a larger design. Both name-based and position-based referencing of I/O ports is supported during module instantiation.

Example: Structural Instantiation of Register and BUFG

```
module foo (sysclk, in, reset,out);
   input sysclk, in, reset;
   output out;
   reg out;
   wire sysclk_out;

   FDC register (sysclk, reset, in, out);  //position based
                                           //referencing
   BUFG clk (.O(sysclk_out), .I(sysclk));  //name based referencing
  ….
  endmodule
```

The virtex.v library file supplied with XST, includes the definitions for FDC and BUFG.

```
module FDC ( C, CLR, D, Q);
   input C;
   input CLR;
   input D;
   output Q;
endmodule
// synthesis attribute BOX_TYPE of FDC is "BLACK_BOX"

module BUFG ( O, I);
   output O;
   input I;
endmodule
// synthesis attribute BOX_TYPE of BUFG is "BLACK_BOX"
```

Parameters

Verilog modules support defining constants known as PARAMETERS which can be passed to module instances to define circuits of arbitrary widths. PARAMETERS form the basis of creating and using LPM blocks in a design to achieve hierarchy and stimulate modular design techniques.

Example of using PARAMETERS

```
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 <= SIZE'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 will cause the instance buf_373 to be 8 bits wide.

assign lpm_reg.out = 8'hFF;

Verilog Limitations in XST

This section describes Verilog limitations in XST support for case sensitivity and blocking and nonblocking assignments.

Case Sensitivity

XST supports case sensitivity as follows:

- Designs using case equivalent names for IO ports, nets, regs and memories
- Equivalent names are renamed using a postfix ("rnm<Index>)
- A rename construct is generated in the EDIF file
- Verilog identifiers that differ only in case

  Following is an example.

```
module upperlower4 (input1, INPUT1, output1,
   output2);
input input1;
input INPUT1;
```

The following restrictions apply for Verilog within XST:

- Designs using equivalent names (named blocks, tasks, and functions) are rejected.

  Example:

```
...
always @(clk)
begin: fir_main5
reg [4:0] fir_main5_w1;
reg [4:0] fir_main5_W1;
```

  This code generates the following error message:

```
ERROR :(VLG_6004). Name conflict (<fir_main5/
   fir_main5_w1> and <fir_main5/fir_main5_W1>)
```

- Designs using case equivalent module names are also rejected.

  Example:

```
module UPPERLOWER10 (AA, aa, YT, yt);
...
```

```
module upperlower10 (a1, a2, yt1, yt2);
...
```

This example generates the following error message:

```
ERROR   : (VLG__0106). Module name conflict
   (UPPERLOWER10 and upperlower10).
```

Blocking and Nonblocking Assignments

XST rejects Verilog designs if a given signal is assigned through both blocking and nonblocking assignments.

Example:

```
always @(in1) begin
  if (in2)  out1 = in1;
  else out1 <= in2;
end
```

If a variable is assigned in both a blocking and nonblocking assignment, the following error message is generated:

```
ERROR   : (VLG__4600). "design.v", line n:
 Cannot mix blocking and non blocking assignments
 on signal <out1>.
```

There are also restrictions when mixing blocking and nonblocking assignments on bits and slices.

The following example is rejected even if there is no real mixing of blocking and non blocking assignments:

```
if (in2) begin
  out1[0] = 1'b0;
  out1[1] <= in1;
end
else begin
  out1[0] = in2;
  out1[1] <= 1'b1;
end
```

Errors are checked at the signal level, not at the bit level. Note that this design is also rejected by Synplicity 5.14.

If there is more than a single VLG_4600 error, only the first one will be 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 Meta Comments

XST supports meta comments in Verilog. Because Verilog does not offer a method for attribute definition such as VHDL, meta comments (comments that are understood by the Verilog parser) are used.

Meta comments can be used as follows:

- Set constraints on individual objects (for example, module, instance, net)
- Set directives on synthesis
  - parallel_case and full_case directives
  - translate_on translate_off directives
  - all tool specific directives (for example, syn_sharing), refer to the “Design Constraints” chapter for details.

Meta comments can be written using the C-style (/* ... */) or the Verilog style (// ...) for comments. C-style comments can be multiple line. Verilog style comments end at the end of the line.

XST supports the following:

- Both C-style and Verilog style meta comments
- translate_on translate_off directives

```
// synthesis translate_on
// synthesis translate_off
```

- parallel_case, full_case directives

```
// synthesis parallel_case full_case
// synthesis parallel_case
// synthesis full_case
```

- Constraints on individual objects

  The general syntax is:

  // synthesis attribute *AttributeName* [**of**] *ObjectName* [**is**] *AttributeValue*

Examples:

// synthesis attribute RLOC of u123 is R11C1.S0

// synthesis attribute HUSET u1 MY_SET

// synthesis attribute fsm_extract of State2 is "yes"

// synthesis attribute fsm_encoding of State2 is "gray"

For a full list of constraints, refer to the “Design Constraints” chapter.

Language Support Tables

The following tables indicate which Verilog constructs are supported in XST. Previous sections in this chapter describe these constructs and their use within XST.

Note XST does not allow underscores in signal names (for example, _DATA_1).

Table 7-3 Constants

Integer Constants	Supported
Real Constants	Unsupported
Strings Constants	Unsupported

Table 7-4 Data Types

Nets	net type	wire	Supported
		tri	Supported
		supply0, supply1	Supported
		wand, wor, triand, trior	Supported
		tri0, tri1, trireg	Unsupported
	drive strength		Ignored

Table 7-4 Data Types

Registers	reg		Supported
	integer		Supported
	real		Unsupported
	realtime		Unsupported
Vectors	net		Supported
	reg		Supported
	vectored		Supported
	scalared		Supported
Memories			Supported
Parameters			Supported
Named Events			Unsupported

Table 7-5 Continuous Assignments

Drive Strength	Ignored
Delay	Ignored

Table 7-6 Procedural Assignments

Blocking Assignments		Supported
Non-Blocking Assignments		Supported
Continuous Procedural Assignments	assign	Supported with limitations See the “Assign and Deassign Statements” section
	deassign	
	force	Unsupported
	release	Unsupported
if Statement	if, if else	Supported
case Statement	case, casex, casez	Supported

Table 7-6 Procedural Assignments

forever Statement		Unsupported
repeat Statement		Unsupported
while Statement		Unsupported
for Statement		Supported (bounds must be static)
fork / join Statement		Unsupported
Timing Control on Procedural Assignments	delay (#)	Ignored
	event (@)	Unsupported
	wait	Unsupported
	named events	Unsupported
Sequential Blocks		Supported
Parallel Blocks		Unsupported
Specify Blocks		Ignored
initial Statement		Ignored
always Statement		Supported
task		Supported (Recursion Unsupported)
functions		Supported (Recursion Unsupported)
disable Statement		Unsupported

Table 7-7 System Tasks and Functions

System Tasks	Ignored
System Functions	Unsupported

Table 7-8 Design Hierarchy

Module definition	Supported
Macromodule definition	Unsupported
Hierarchical names	Unsupported
defparam	Supported
Array of instances	Unsupported

Table 7-9 Compiler Directives

‘celldefine ‘endcelldefine	Ignored
‘default_nettype	Ignored
‘define	Supported
‘undef	Unsupported
‘ifdef ‘else ‘endif	Supported
‘include	Supported
‘resetall	Ignored
‘timescale	Ignored
‘unconnected_drive ‘nounconnected_drive	Ignored
‘uselib	Unsupported

Primitives

XST supports certain gate level primitives. The supported format is as follows:

```
gate_type instance_name (output, inputs, ...);
```

Example 3-10: Gate Level Primitive Instantiations

```
and U1 (out, in1, in2);
bufif1 U2 (triout, data, trienable);
```

The following table shows which primitives are supported.

Table 7-10 Primitives

Gate Level Primitives	and nand nor or xnor xor	Supported
	buf not	Supported
	bufif0 bufif1 notif0 notif1	Supported
	pulldown pullup	Unsupported
	drive strength	Ignored
	delay	Ignored
	array of primitives	Unsupported
Switch Level Primitives	cmos nmos pmos rcmos rnmos rpmos	Unsupported
	rtran rtranif0 rtranif1 tran tranif0 tranif1	Unsupported
User Defined Primitives		Unsupported

Verilog Reserved Keywords

The following table shows the Verilog reserved keyword

always	end	ifnone	or	rpmos	tranif1
and	endcase	initial	output	rtran	tri
assign	endmodule	inout	parameter	rtranif0	tri0
begin	endfunction	input	pmos	rtranif1	tri1
buf	endprimitive	integer	posedge	scalared	triand
bufif0	endspecify	join	primitive	small	trior
bufif1	endtable	large	pull0	specify	trireg
case	endtask	macromodule	pull1	specparam	vectored
casex	event	medium	pullup	strong0	wait
casez	for	module	pulldown	strong1	wand
cmos	force	nand	rcmos	supply0	weak0
deassign	forever	negedge	real	supply1	weak1
default	for	nmos	realtime	table	while
defparam	function	nor	reg	task	wire
disable	highz0	not	release	time	wor
edge	highz1	notif0	repeat	tran	xnor
else	if	notif1	rnmos	tranif0	xor

Chapter 8

Command Line Mode

This chapter describes how to run XST using the command line. The chapter contains the following sections.

- “Introduction”
- “Launching XST”
- “Setting Up an XST Script”
- “Run Command”
- “Set Command”
- “Elaborate Command”
- “Time Command”
- “Example 1: How to Synthesize VHDL Designs”
- “Example 2: How to Synthesize Verilog Designs”

Introduction

With XST, you can run synthesis in command line mode instead of from the Process window in the Project Navigator. To run synthesis from the command line, you must use special executable files. If you work on a workstation, the name of the executable is "xst". On a PC, the name of the executable is "xst.exe".

Before attempting to run XST, it must be properly installed. The XST installation directory must be specified using the XST_HOME environment variable or through the -home option.

The XST_EXE and XST_HOME variables must be set as follows:

XST_EXE=*path_to_ISE***\bin\nt\xst.exe**

XST_HOME=*path_to_ISE***\data\HDLSynthesis\nt**

XST will generate 3 types of files:

- Design output files include EDIF (.EDN) and .NCF.

  These files are generated in the current output directory (see the -ofn option)

- Temporary files

  Temporary file are generated in the XST temp directory. By default the XST temp directory is /tmp on workstations and the directory specified by either the TEMP or TMP environment variables under Windows. The XST temp directory can be changed using the "set -tmpdir" command.

- VHDL compilation files

  Note There are no compilation files for Verilog.

  VHDL compilation files are generated in the VHDL dump directory. The default dump directory is the XST temp directory. The VHDL dump directory can be changed using the "set -dumpdir" command.

Note It is strongly suggested that you *clean the VHDL dump directory* regularly because the directory contains the files resulting from the compilation of *all VHDL* files during all XST sessions. Eventually the number of files stored in the VHDL dump directory may severely impact CPU performances. This directory is not automatically cleaned by XST.

Launching XST

You can run XST in two ways.

- XST Shell—You can type **xst** and then enter directly into an XST shell. You enter your commands and execute them. In fact, in order to run synthesis you have to specify a complete command with all required options before running. XST does not accept a mode where you can first enter **set** *option_1*, then **set** *option_2*, and then enter **run**.

  All the options must be set up at once. Therefore, this method is very cumbersome and Xilinx suggests the use of the next described method.

- Script File—You can store your commands in a separate script file and run all of them at once. To execute your script file, run the following workstation or PC command:

  xst -ifn *script_file_name* [*-ofn*] *log_file_name*

  Note The -ofn option is not mandatory. If you omit it, then all messages will display on the screen.

Besides the -ifn and -ofn options, XST also accepts the -home option. The -home option allows you to specify the XST installation directory and overwrites the XST_HOME definition.

For example, assume that the text below is contained in a file foo.scr.

```
run
-ifn tt1.vhd
-ifmt VHDL
-opt_mode SPEED
-opt_level 1
-ofn tt1.edn
-family virtex
```

This script file can be executed under XST using the following command:

Workstation:

```
xst -ifn foo.scr
```

PC:

```
xst.exe -ifn foo.scr
```

You can also generate a log file with the following command:

Workstation:

```
xst -ifn foo.scr -ofn foo.log
```

PC:

```
xst.exe -ifn foo.scr -ofn foo.log
```

Besides the -ifn and -ofn options, XST also accepts the -home option. This option allows you to specify the XST installation directory and will overwrite the XST_HOME definition.

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.

Setting Up an XST Script

An XST script is a set of commands, each command having various options. XST recognizes the following commands:

- run
- set
- elaborate
- time short

Run Command

Following is a description of the run command.

- The command begins with a keyword **run**, which is followed by a set of options and its values

  run *option_1_value option_2_value ...*

- Each option name starts with dash (-). For instance: -ifn, -ifmt, -ofn.
- Each option has one value. There are no options without a value.
- The value for a given option can be one of the following:
  - Predefined by XST (for instance, YES or NO)
  - Any string (for instance, a file name or a name of the top level entity)
  - An integer
- Options and values are case sensitive.

In the following tables, you can find the name of each option and its values.

- First column—the name of the options you can use in command line mode. If the option is in bold, then it means that it must be present in the command line.
- Second column—the option description
- Third column—the possible values of this option. The values in bold are the default values.

Table 8-1 Global Options

Run Command Options	Description	Values
-ifn	Input / Project File Name	*file_name*
-ifmt	Input Format	VHDL, Verilog, NSR, EDIF
-ofn	Output File Name	*file_name*
-ofmt	Output File Format	**EDIF**
-family	Target Technology	Virtex, VirtexE, Virtex2, Spartan2, 9500, 9500XL, 9500XV
-opt_mode	Optimization Criteria	AREA, **SPEED**
-opt_level	Optimization Effort	**1**, 2
-attribfile	Constraint File Name	*file_name*

Table 8-2 VHDL Source Options

Run Command Options	Description	Values
-work_lib	Work Library	*name*
-ent	Entity Name	*name*
-arch	Architecture	*name*

Table 8-3 Verilog Source Options

Run Command Options	Description	Values
-top	Top Module name	*name*
-vlgcase	Case Implementation Style	Full, Parallel, Full-Parallel

Table 8-4 HDL Options (VHDL and Verilog)

Run Command Options	Description	Values
-fsm_extract	Automatic FSM Extraction	**YES**, NO
-fsm_encoding	Encoding Algorithm	**Auto**, One-Hot, Compact, Sequential, Gray, Johnson, User
-fsm_fftype	FSM Flip-Flop Type	**D**, T
-ram_extract	FSM Flip-Flop Type	**YES**, NO
-ram_style	RAM Style	**Auto**, distributed, block
-mux_extract	Mux Extraction	**YES**, NO, FORCE
-mux_style	Mux Style	**Auto**, MUXF, MUXCY
-decoder_extract	Decoder Extraction	**YES**, NO
-priority_extract	Priority Encoder Extraction	**YES**, NO, FORCE
-shreg_extract	Shift Register Extraction	**YES**, NO
-shift_extract	Logical Shift Extraction	**YES**, NO
-xor_collapse	XOR Collapsing	**YES**, NO
-resource_sharing	Resource Sharing	**YES**, NO
-complex_clken	Complex Clock Enable Extraction	**YES**, NO
-resolutionStyle	Resolution Style	WIRE_OR, WIRE_AND, **WIRE_MS**

Table 8-5 Target Options (9500, 9500XL, 9500XV)

Run Command Options	Description	Values
-iobuf	Add I/O Buffers	**YES**, NO
-macrogen	Macro Generator	Macro+, LogiBLOX, **Auto**
-pld_mp	Macro Preserve	**YES**, NO
-pld_xp	XOR Preserve	**YES**, NO
-merge	Flatten Hierarchy	**YES**, NO
-pld_ce	Clock Enable	**YES**, NO
-pld_ffopt	Flip-Flop Optimization	**YES**, NO
-wysiwyg	What You See Is What You Get	YES, **NO**
-ofmt	Output Format	**EDIF**

Table 8-6 Target Options (Virtex, VirtexE, Virtex2, Spartan2)

Run Command Options	Description	Values
-iobuf	Add I/O Buffers	**YES**, NO
-speedgrade	Speed Grade for Timing Analysis	*integer* (Default is the fastest speed grade available)
-macrogen	Macro Generator	**Macro**+
-bufg	Maximum Number of BUFGs created by XST	*integer* (Default **4**)
-maxfanout	Maximum Fanout	*integer* (Default **100**)
-keep_hierarchy	Keep Hierarchy	YES, **NO**
-glob_opt	Global Optimization Goal	**AllClockNets**, Inpad_to_Outpad, Offset_in_Before, Offset_out_after, Max_Delay
-incremental_synthesis	Incremental Synthesis	YES, **NO**
-register_duplication	Register Duplication	**YES**, NO
-ofmt	Output Format	**EDIF**

Set Command

In addition to the run command, XST also recognizes the set command. This command accepts the following options:

Table 8-7 Set Command Options

Set Command Options	Description	Values
-tmpdir	Location of all temporary files generated by XST during a session	Any valid path to a directory
-dumpdir	Location of all files resulting from VHDL compilation	Any valid path to a directory
-overwrite	Overwrite existing files. When NO, if XST generates a file that already exists, the previous file will be saved using .000, .001 suffixes	YES, **NO**

Elaborate Command

The goal of this command is to pre-compile VHDL files in a specific library or to verify Verilog files. Taking into account that the compilation process is included in the "run", this command remains optional.

The elaborate command accepts the following options:

Table 8-8 Elaborate Command Options

Elaborate Command Options	Description	Values
-ifn	VHDL file or project Verilog file	*filename*
-ifmt	Format	**VHDL**, VERILOG
-work_lib	VHDL working library, not available for Verilog	*name*

Time Command

The time command displays some information about CPU utilization.

Use the command **`time short`** to enable the CPU information. Use the command **`time off`** to remove reporting of CPU utilization. By default, CPU utilization is not reported.

Example 1: How to Synthesize VHDL Designs

The goal of this example is to synthesize a hierarchical VHDL design for a Virtex FPGA.

Following are two main cases:

- Case 1—all design blocks (entity / architecture pairs) are located in a single VHDL file.
- Case 2—each design block (entity / architecture pair) is located in a separate VHDL file.

The example uses a VHDL design, called watchvhd. The files for watchvhd can be found in the ISEexamples \ watchvhd directory of the Foundation ISE installation directory.

This design contains 6entities:

- stopwatch
- statmach
- tenths (a CORE Generator core)
- decode
- cnt60
- hex2led

Case 1

For Case 1, all design blocks will be located in a single VHDL file.

1. Create a new directory called vhdl_s.
2. Copy the following files from the ISEexamples \ watchvhd directory of the Foundation ISE installation directory to the vhdl_s directory.

- stopwatch.vhd
- statmach.vhd
- decode.vhd
- cnt60.vhd
- hex2led.vhd

3. Copy and paste the contents of the files into a single file called 'watchvhd.vhd'. Make sure the contents of 'stopwatch.vhd' appear last in the file.
4. To synthesize this design for Speed with optimization effort 1 (Low), execute the following command:

```
run -ifn watchvhd.vhd -ifmt VHDL -ofn watchvhd.edn
-ofmt EDIF -family Virtex -opt_mode Speed -
opt_level 1
```

Please note that all options in this command except the -opt_mode and -opt_level ones are mandatory. All other options will be used by default.

This command can be launched in two ways:

- Directly from an XST shell
- Script mode

XST Shell

To use the XST shell, perform the following steps:

1. In the tcsh or other shell type "xst". XST will start and prompt you with the following message:

```
XST D-19
Copyright (c) 1995-2000 Xilinx, Inc. All rights
reserved.
-->
```

2. Enter the following command at the - -> prompt to start synthesis.

```
run -ifn watchvhd.vhd -ifmt VHDL -ofn watchvhd.edn
-ofmt EDIF -family Virtex -opt_mode Speed
-opt_level 1
```

3. When the synthesis is complete, XST shows the prompt -->, you can type **`quit`** to exit the XST shell.

During this run, XST creates the following files:

- watchvhd.edn: an EDIF netlist ready for the implementation tools
- watchvhd.ncf: a NCF file

Note All messages issued by XST are displayed on the screen only. If you want to save your messages in a separate log file, then the best way is to use script mode to launch XST.

In the previous run, XST synthesized entity "stopwatch" as a top level module of the design. The reason is that this block was placed at the end of the VHDL file. XST picks up the latest block in the VHDL file and treats it as a top level one. Suppose you would like to synthesize just "hex2led" and check its performance independently of the other blocks. This can be done by specifying the top level entity to synthesize in the command line using the -ent option (please refer to Table 8-2 of this chapter or more information):

```
run -ifn watchvhd.vhd -ifmt VHDL -ofn watchvhd.edn
-ofmt EDIF -family Virtex -opt_mode Speed -opt_level 1
-ent hex2led
```

Script Mode

It can be very tedious work to enter XST commands directly in the XST shell, especially when you have to specify several options and execute the same command several times. You can run XST in a script mode as follows:

1. Open a new file named xst.cmd in the current directory. Put the previously executed XST shell command into this file and save it.

   ```
   run -ifn watchvhd.vhd -ifmt VHDL -ofn watchvhd.edn
   -ofmt EDIF -family Virtex -opt_mode Speed
   -opt_level 1
   ```

2. From the tcsh or other shell, enter the following command to start synthesis.

   ```
   xst -ifn xst.cmd
   ```

3. Please note that in this case all XST messages will display on the screen. If you want to save them in a separate log file, for example, watchvhd.log, you must execute the following command.

```
xst -ifn xst.cmd -ofn watchvhd.log
```

You can improve the readability of the xst.cmd file, especially if you use many options to run synthesis. You can place each option with its value on a separate line, respecting the following rules:

- The first line must contain only the run command without any options.
- There must be no empty lines in the middle of the command.
- Each line (except the first one) must start with a dash (-)

For the previously used command you may have the xst.cmd file in the following form:

```
run
-ifn watchvhd.vhd
-ifmt VHDL
-ofn watchvhd.edn
-ofmt EDIF
-family Virtex
-opt_mode Speed
-opt_level 1
```

Case 2

For Case 2, each design block is located in a separate VHDL file.

1. Create a new directory, named vhdl_m.
2. Copy the following files from the ISEexamples\watchvhd directory of the Foundation ISE installation directory to the newly created vhdl_m directory.
   - stopwatch.vhd
   - statmach.vhd
   - decode.vhd
   - cnt60.vhd
   - hex2led.vhd

To synthesize the design, which is now represented by three VHDL files, you can use the project approach supported in XST. A VHDL project file contains a list of VHDL files from the project. The order of the files is not important. XST is able to recognize the hierarchy and compile VHDL files in the correct order. Moreover, XST automatically detects the top level block for synthesis.

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:

```
statmach.vhd
decode.vhd
stopwatch.vhd
cnt60.vhd
hex2led.vhd
```

3. To synthesize the design, execute the following command from XST shell or via script file:

```
run -ifn watchvhd.prj -ifmt VHDL -ofn watchvhd.edn
-ofmt EDIF -family Virtex -opt_mode Speed
-opt_level 1
```

If you want to synthesize just "hex2led" and check its performance independently of the other blocks, you can specify the top level entity to synthesize in the command line, using the -ent option (please refer to Table 8-2 for more details):

```
run -ifn watchvhd.prj -ifmt VHDL -ofn watchvhd.edn
-ofmt EDIF -family Virtex -opt_mode Speed -opt_level 1
-ent hex2led
```

During VHDL compilation, XST uses the library "work" as the default. If some VHDL files must be compiled to different libraries, then you can add the name of the library just before the file name. Suppose that "hexl2led" must be compiled into the library, called "my_lib", then the project file must be:

```
statmach.vhd
decode.vhd
stopwatch.vhd
cnt60.vhd
my_lib hex2led.vhd
```

Sometimes, XST is not able to recognize the order and issues the following message.

```
WARNING: (VHDL_3204). The sort of the vhdl files
   failed, they will be compiled in the order of
   the project file.
```

In this case you must do the following:

- Put all VHDL files in the correct order.
- Add at the end of the list on a separate line the keyword "nosort". XST will then use your predefined order during the compilation step.

```
statmach.vhd
decode.vhd
cnt60.vhd
hex2led.vhd
stopwatch.vhd
nosort
```

Example 2: How to Synthesize Verilog Designs

The goal of this example is to synthesize a hierarchical Verilog design for a Virtex FPGA.

Two main cases are considered:

- All design blocks (modules) are located in a single Verilog file.
- Each design block (modules) is located in a separate Verilog file.

Example 2 uses a Verilog design, called watchver. These files can be found in the ISEexamples\watchver directory of the Foundation ISE installation directory.

- stopwatch.v
- statmach.v
- decode.v

- cnt60.v
- hex2led.v

This design contains six modules:

- stopwatch
- statmach
- tenths (a CORE Generator core)
- decode
- cnt60
- HEX2LED

Case 1

All design blocks will be located in a single Verilog file.

1. Create a new directory called vlg_s.
2. Copy the following files from the ISEexamples\watchver directory of the Foundation ISE installation directory to the newly created vlg_s directory.
3. Copy and paste the contents of the files into a single file called 'watchver.ver'. Make sure the contents of 'stopwatch.v' appear last in the file.

To synthesize this design for Speed with optimization effort 1 (Low), execute the following command:

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.edn
-ofmt EDIF -family Virtex -opt_mode Speed -opt_level 1
```

Note All options in this command except -opt_mode and -opt_level ones are mandatory. All other options are used by default.

This command can be launched in two ways:

- Directly from the XST shell
- Script mode

XST Shell

To use the XST shell, perform the following steps.

1. In the tcsh or other shell, enter **xst**. XST begins and prompts you with the following message:

```
XST D-19
Copyright (c) 1995-2000 Xilinx, Inc. All rights
   reserved.
-->
```

2. Enter the following command at the - -> prompt to start synthesis:

```
run -ifn watchver.v -ifmt Verilog -ofn
watchver.edn -ofmt EDIF -family Virtex -opt_mode
Speed -opt_level 1
```

3. When the synthesis is complete, XST displays the - -> prompt. Enter **quit** to exit the XST shell.

During this run, XST creates the following files:

- watchver.edn: an EDIF netlist ready for the implementation tools
- watchver.ncf: a NCF file

Note All messages issued by XST are displayed on the screen only. If you want to save your messages in a separate log file, then the best way is to use script mode to launch XST.

In the previous run, XST synthesized module, stopwatch, as the top level module of the design. XST automatically recognizes the hierarchy and detects the top level module. If you would like to synthesize just HEX2LED and check its performance independently of the other blocks, you can specify the top level module to synthesize in the command line, using the -top option (please refer to Table 8-3)

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.edn
-ofmt EDIF -family Virtex -opt_mode Speed -opt_level 1
-top HEX2LED
```

Script Mode

It can be very tedious work entering XST commands directly into the XST shell, especially when you have to specify several options and execute the same command several times. You can run XST in a script mode as follows.:

1. Open a new file called xst.cmd in the current directory. Put the previously executed XST shell command into this file and save it.

   ```
   run -ifn watchver.v -ifmt Verilog -ofn
   watchver.edn -ofmt EDIF -family Virtex -opt_mode
   Speed -opt_level 1
   ```

2. From the tcsh or other shell, enter the following command to start synthesis.

   ```
   xst -ifn xst.cmd
   ```

3. Please note that in this case all XST messages display on the screen. If you want to save these messages in a separate log file, for example, watchver.log, execute the following command.

   ```
   xst -ifn xst.cmd -ofn watchver.log
   ```

You can improve the readability of the xst.cmd file, especially if you use many options to run synthesis. You can place each option with its value on a separate line, respecting the following rules:

- The first line must contain only the run command without any options.
- There must be no empty lines in the middle of the command
- Each line (except the first one) must start with a dash (-)

For the previously used command, you may have the xst.cmd file in the following form:

```
run
-ifn watchver.v
-ifmt Verilog
-ofn watchver.edn
-ofmt EDIF
-family Virtex
-opt_mode Speed
-opt_level 1
```

Case 2

Each design block is located in a separate Verilog file.

1. Create a new directory named vlg_m.
2. Copy the watchver design files from the ISEexamples\watchver directory of the Foundation ISE installation directory to the newly created vlg_m directory.

To synthesize the design, which is now represented by four Verilog files, you can use the project approach supported in XST. A Verilog project file contains a set of "include" Verilog statements (one each per Verilog module). The order of the files in the project is not important. XST is able to recognize the hierarchy and compile Verilog files in the correct order. Moreover, XST automatically detects the top level module for synthesis.

For our example:

1. Open a new file, called watchver.v
2. Enter the names of the Verilog files in any order into this file and save it:

```
‘include "decode.v"
‘include "statmach.v"
‘include "stopwatch.v"
‘include "cnt60.v"
‘include "hex2led.v"
```

3. To synthesize the design, execute the following command from the XST shell or via a script file:

```
run -ifn watchver.v -ifmt Verilog -ofn
watchver.edn -ofmt EDIF -family Virtex -opt_mode
Speed -opt_level 1
```

If you want to synthesize just HEX2LED and check its performance independently of the other blocks, you can specify the top level module to synthesize in the command line, using the -top option (please refer to Table 8-3 for more information):

```
run -ifn watchver.v -ifmt Verilog -ofn watchver.edn
-ofmt EDIF -family Virtex -opt_mode Speed -opt_level 1
-top HEX2LED
```

Appendix A

XST Naming Conventions

This appendix discusses net naming and instance naming conventions.

Net Naming Conventions

These rules are listed in order of naming priority.

1. External pin names maintained.
2. Keep hierarchy in signal names, using underscores as hierarchy designators.
3. Output signal names of registers, including state bits, will be maintained. Use the hierarchical name from the level where the register was inferred.
4. Output signals of clock buffers get *_clockbuffertype* (like _BUFGP or _IBUFG) following the clock signal name.
5. Input nets to registers and tristates maintain their names.
6. Output net names of IBUFs are named *net_name*_IBUF. For example, for an IBUF with an output net name of DIN, the output IBUF net name is DIN_IBUF.

   Input net names to OBUFs are named *net_name*_OBUF. For example, for an OBUF with an input net name of DOUT, the input OBUF net name is DOUT_OBUF.

Instance Naming Conventions

These rules are listed in order of naming priority.

1. Keep hierarchy in instance names, using underscores as hierarchy designators.
2. Register instances, including state bits, will be named for the output signal.
3. Clock buffer instances are named *_clockbuffertype* (like _BUFGP or _IBUFG) after the output signal.
4. Instantiation instance names of black boxes are maintained.
5. Instantiation instance names of library primitives are maintained.
6. Input and output buffers are named _IBUF or _OBUF after the pad name.
7. Output instance names of IBUFs are named *instance_name*_IBUF.

   Input instance names to OBUFs are named *instance_name*_OBUF.