XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices

UG687 (v 13.1) March 1, 2011
Xilinx is disclosing this user guide, manual, release note, and/or specification (the “Documentation”) to you solely for use in the development of designs to operate with Xilinx hardware devices. You may not reproduce, distribute, republish, download, display, post, or transmit the Documentation in any form or by any means including, but not limited to, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Xilinx. Xilinx expressly disclaims any liability arising out of your use of the Documentation. Xilinx reserves the right, at its sole discretion, to change the Documentation without notice at any time. Xilinx assumes no obligation to correct any errors contained in the Documentation, or to advise you of any corrections or updates. Xilinx expressly disclaims any liability in connection with technical support or assistance that may be provided to you in connection with the Information.

THE DOCUMENTATION IS DISCLOSED TO YOU “AS-IS” WITH NO WARRANTY OF ANY KIND. XILINX MAKES NO OTHER WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, REGARDING THE DOCUMENTATION, INCLUDING ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT OF THIRD-PARTY RIGHTS. IN NO EVENT WILL XILINX BE LIABLE FOR ANY CONSEQUENTIAL, INDIRECT, EXEMPLARY, SPECIAL, OR INCIDENTAL DAMAGES, INCLUDING ANY LOSS OF DATA OR LOST PROFITS, ARISING FROM YOUR USE OF THE DOCUMENTATION.

© Copyright 2002-2011 Xilinx Inc. All Rights Reserved. XILINX, the Xilinx logo, the Brand Window and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners. The PowerPC name and logo are registered trademarks of IBM Corp., and used under license. All other trademarks are the property of their respective owners.
Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>03/01/2011</td>
<td>13.1</td>
<td>• Changed guide title</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Synthesis support of the 7 Series FPGA families.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Improved inference support of block RAMs with byte-write enable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The recommended Single-Process Description Style has been generalized</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to any number of write columns beyond a single BRAM primitive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The message filtering capability, already available in the ISE Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Suite, is now available to command-line users. See Running XST as a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Standalone Tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Made additional minor changes throughout.</td>
</tr>
</tbody>
</table>
# Table of Contents

Revision History .......................................................................................................................... 3

## Chapter 1  Introduction ........................................................................................................... 11

  - About XST .......................................................................................................................... 11
  - Acronyms ........................................................................................................................... 11

## Chapter 2  Creating and Synthesizing an XST Project .............................................................. 13

  - HDL Synthesis Project File Coding Example .................................................................... 13
  - Creating an HDL Synthesis Project .................................................................................. 14
  - Running XST in ISE Design Suite .................................................................................... 15
  - Running XST in Command Line Mode .............................................................................. 15
  - XST Output Files ................................................................................................................ 22

## Chapter 3  XST VHDL Language Support ............................................................................... 25

  - Advantages of VHDL .......................................................................................................... 25
  - VHDL IEEE Support .......................................................................................................... 25
  - VHDL Data Types .............................................................................................................. 25
  - VHDL Objects .................................................................................................................... 30
  - VHDL Operators ................................................................................................................. 31
  - VHDL Entity and Architecture Descriptions .................................................................... 32
  - VHDL Combinatorial Circuits ............................................................................................ 40
  - VHDL Sequential Logic .................................................................................................... 49
  - VHDL Functions and Procedures ....................................................................................... 52
  - VHDL Assert Statements .................................................................................................... 55
  - VHDL Libraries and Packages .......................................................................................... 58
  - VHDL File Type Support ................................................................................................... 62
  - VHDL Constructs ................................................................................................................. 67
  - VHDL Reserved Words ........................................................................................................ 71

## Chapter 4  XST Verilog Support ............................................................................................. 73

  - Verilog Variable Part Selects ............................................................................................. 74
  - Structural Verilog Features ................................................................................................. 74
  - Verilog Parameters .............................................................................................................. 76
  - Verilog Parameter and Attribute Conflicts ......................................................................... 78
  - Verilog Usage Restrictions in XST .................................................................................... 79
  - Verilog–2001 Attributes and Meta Comments ................................................................... 81
  - Verilog Constructs ................................................................................................................. 83
  - Verilog System Tasks and Functions ................................................................................... 85
Verilog Primitives ................................................................. 89
Verilog User Defined Primitive (UDP) ....................................... 90
Verilog Reserved Keywords ...................................................... 92
Verilog 2001 Support in XST ..................................................... 93

Chapter 5  XST Behavioral Verilog Support ........................................ 95
  Initial Values ........................................................................ 95
  Arrays Coding Examples ...................................................... 96
  Multi-Dimensional Arrays ..................................................... 96
  Data Types .......................................................................... 96
  Legal Statements ................................................................... 97
  Expressions ......................................................................... 98
  Blocks ................................................................................. 100
  Modules .............................................................................. 101
  Continuous Assignments ...................................................... 102
  Procedural Assignments ....................................................... 103
  Tasks and Functions ............................................................ 109
  Blocking Versus Non-Blocking Procedural Assignments ........... 111
  Constants .......................................................................... 112
  Macros ................................................................................. 112
  Include Files ....................................................................... 112
  Comments .......................................................................... 113
  Generate Statements ............................................................ 113

Chapter 6  XST Mixed Language Support ......................................... 117
  VHDL and Verilog Boundary Rules ......................................... 118
  Generics Support ................................................................. 119
  Port Mapping ....................................................................... 120
  Library Search Order (LSO) Files ........................................... 121

Chapter 7  XST HDL Coding Techniques .......................................... 125
  Relative Advantages and Disadvantages of VHDL and Verilog .......... 125
  Macro Inference Flow Overview ............................................. 126
  Flip-Flops and Registers ........................................................ 127
  Latches ................................................................................ 131
  Tristates .............................................................................. 134
  Counters and Accumulators .................................................. 138
  Shift Registers ..................................................................... 142
  Dynamic Shift Registers ....................................................... 151
  Multiplexers ....................................................................... 155
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic Operators</td>
<td>160</td>
</tr>
<tr>
<td>Comparators</td>
<td>163</td>
</tr>
<tr>
<td>Dividers</td>
<td>165</td>
</tr>
<tr>
<td>Adders, Subtractors, and Adders/Subtractors</td>
<td>167</td>
</tr>
<tr>
<td>Multipliers</td>
<td>171</td>
</tr>
<tr>
<td>Multiply-Add and Multiply-Accumulate</td>
<td>176</td>
</tr>
<tr>
<td>Extended DSP Inferencing</td>
<td>180</td>
</tr>
<tr>
<td>Resource Sharing</td>
<td>183</td>
</tr>
<tr>
<td>RAMs</td>
<td>186</td>
</tr>
<tr>
<td>ROMs</td>
<td>247</td>
</tr>
<tr>
<td>FSM Components</td>
<td>254</td>
</tr>
<tr>
<td>Black Boxes</td>
<td>266</td>
</tr>
</tbody>
</table>

### Chapter 8 XST FPGA Optimization

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping Logic to Block RAM</td>
<td>269</td>
</tr>
<tr>
<td>Flip-Flop Implementation Guidelines</td>
<td>270</td>
</tr>
<tr>
<td>Flip-Flop Retiming</td>
<td>271</td>
</tr>
<tr>
<td>Speed Optimization Under Area Constraint</td>
<td>272</td>
</tr>
<tr>
<td>Implementation Constraints</td>
<td>273</td>
</tr>
<tr>
<td>Xilinx Device Primitive Support</td>
<td>274</td>
</tr>
<tr>
<td>Using the UniMacro Library</td>
<td>280</td>
</tr>
<tr>
<td>Cores Processing</td>
<td>281</td>
</tr>
<tr>
<td>Mapping Logic to LUTs</td>
<td>282</td>
</tr>
<tr>
<td>Controlling Placement on the Device</td>
<td>284</td>
</tr>
<tr>
<td>Inserting Buffers</td>
<td>285</td>
</tr>
<tr>
<td>Using the PCI Flow With XST</td>
<td>285</td>
</tr>
</tbody>
</table>

### Chapter 9 XST Design Constraints

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifying Constraints</td>
<td>287</td>
</tr>
<tr>
<td>Constraints Precedence Rules</td>
<td>288</td>
</tr>
<tr>
<td>Synthesis Options in ISE Design Suite</td>
<td>288</td>
</tr>
<tr>
<td>VHDL Attributes</td>
<td>289</td>
</tr>
<tr>
<td>Verilog-2001 Attributes</td>
<td>290</td>
</tr>
<tr>
<td>XST Constraint File (XCF)</td>
<td>293</td>
</tr>
</tbody>
</table>

### Chapter 10 XST General Constraints

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add I/O Buffers (-iobuf)</td>
<td>297</td>
</tr>
<tr>
<td>BoxType (BOX_TYPE)</td>
<td>299</td>
</tr>
<tr>
<td>Bus Delimiter (-bus_delimiter)</td>
<td>301</td>
</tr>
<tr>
<td>Case (-case)</td>
<td>302</td>
</tr>
</tbody>
</table>
Case Implementation Style (--vlgcase)................................................. 303
Verilog Macros (-define) .................................................................... 305
Duplication Suffix (--duplication_suffix)........................................... 306
Full Case (FULL_CASE) ....................................................................... 307
Generate RTL Schematic (--rtlview) .................................................... 309
Generics (-generics) ........................................................................... 310
Hierarchy Separator (--hierarchy_separator) ...................................... 312
I/O Standard (IOSTANDARD) ............................................................... 313
Keep (KEEP) ....................................................................................... 314
Keep Hierarchy (KEEP_HIERARCHY) ................................................ 316
Library Search Order (--iso) ............................................................... 319
LOC ....................................................................................................... 320
Netlist Hierarchy (-netlist_hierarchy) ................................................ 321
Optimization Effort (OPT_LEVEL) ...................................................... 322
Optimization Goal (OPT_MODE) ......................................................... 324
Parallel Case (PARALLEL_CASE) ...................................................... 325
RLOC ..................................................................................................... 327
Save (S or SAVE) .................................................................................. 327
Synthesis Constraint File (--uc).............................................................. 328
Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON) ......................................................................................... 329
Ignore Synthesis Constraints File (--iuc) ............................................. 330
Verilog Include Directories (--vlgincdir) ........................................... 331
HDL Library Mapping File (--xsthdpini) ............................................. 332
Work Directory (--xsthdpdir) .............................................................. 334

Chapter 11  XST HDL Constraints ...................................................... 337
Automatic FSM Extraction (FSM_EXTRACT) .................................... 337
Enumerated Encoding (ENUM_ENCODING) ....................................... 339
Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) ......................................................................................... 340
FSM Encoding Algorithm (FSM_ENCODING) ..................................... 342
Mux Minimal Size (MUX_MIN_SIZE) .................................................. 344
Resource Sharing (RESOURCE_SHARING) ...................................... 346
Safe Recovery State (SAFE_RECOVERY_STATE) .............................. 348
Safe Implementation (SAFE_IMPLEMENTATION) ............................. 349

Chapter 12  XST FPGA Constraints (Non-Timing) ............................ 351
Asynchronous to Synchronous (ASYNC_TO_SYNC) ......................... 351
Automatic BRAM Packing (AUTO_BRAM_PACKING) ....................... 353
BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) ........................................... 354
Buffer Type (BUFFER_TYPE) ................................................................................. 356
Extract BUFGCE (BUFGCE) ................................................................................. 357
Cores Search Directories (~sd) ............................................................................ 358
DSP Utilization Ratio (DSP_UTILIZATION_RATIO) ............................................. 359
FSM Style (FSM_STYLE) ...................................................................................... 361
Power Reduction (POWER) ................................................................................... 363
Read Cores (READ_CORES) ................................................................................. 365
LUT Combining (LC) ............................................................................................. 367
Map Logic on BRAM (BRAM_MAP) ................................................................. 368
Max Fanout (MAX_FANOUT) ................................................................................ 369
Move First Stage (MOVE_FIRST_STAGE) ............................................................ 371
Move Last Stage (MOVE_LAST_STAGE) ............................................................... 373
Multiplier Style (MULTI_STYLE) ....................................................................... 375
Number of Global Clock Buffers (~bufg) ............................................................ 377
Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) ......................... 378
Pack I/O Registers Into IOBs (IOB) ................................................................. 380
RAM Extraction (RAM_EXTRACT) ................................................................. 381
RAM Style (RAM_STYLE) .................................................................................. 383
BRAM Read-First Implementation
(RDADDR_COLLISION_HWCONFIG) .............................................................. 386
Reduce Control Sets (REDUCE_CONTROL_SETS) ........................................... 388
Register Balancing (REGISTER_BALANCING) ............................................... 389
Register Duplication (REGISTER_DUPLICATION) ............................................. 393
ROM Extraction (ROM_EXTRACT) ................................................................. 395
ROM Style (ROM_STYLE) .................................................................................. 397
Shift Register Extraction (SHREG_EXTRACT) ............................................... 399
Shift Register Minimum Size (SHREG_MIN_SIZE) ......................................... 401
Use Low Skew Lines (USELOWSKEWLINES) ............................................... 402
Slice (LUT-FF Pairs) Utilization Ratio
(SLICE_UTILIZATION_RATIO) ...................................................................... 403
Slice (LUT-FF Pairs) Utilization Ratio Delta
(SLICE_UTILIZATION_RATIO_MAXMARGIN) .................................................... 405
Map Entity on a Single LUT (LUT_MAP) .......................................................... 407
Use Carry Chain (USE_CARRY_CHAIN) ........................................................... 408
Convert Tristates to Logic (TRISTATE2LOGIC) ............................................ 410
Use Clock Enable (USE_CLOCK_ENABLE) ....................................................... 412
Use Synchronous Set (USE_SYNC_SET) ............................................................ 414
Use Synchronous Reset (USE_SYNC_RESET) .................................................... 416
Use DSP Block (USE_DSP48) .................................................................................. 418

| Chapter 13 | XST Timing Constraints ................................................................. 421 |
| Global Optimization Goal ........................................................................ 421 |
| User Constraints File (UCF) .................................................................... 421 |
| XST Constraint File (XCF) ....................................................................... 422 |
| Clock Signal (CLOCK_SIGNAL) .................................................................. 423 |
| Cross Clock Analysis (~cross_clock_analysis) ......................................... 424 |
| From-To (FROM-TO) ................................................................................ 425 |
| Global Optimization Goal (~glob_opt) ....................................................... 426 |
| Offset (OFFSET) ...................................................................................... 428 |
| Period (PERIOD) ...................................................................................... 429 |
| Timing Name (TNM) ................................................................................ 430 |
| Timing Name on a Net (TNM_NET) .............................................................. 431 |
| Timegroup (TIMEGRP) .............................................................................. 432 |
| Timing Ignore (TIG) ................................................................................ 433 |
| Write Timing Constraints (~write_timing_constraints) ............................ 434 |

| Chapter 14 | XST-Supported Third-Party Constraints ........................................... 435 |

| Chapter 15 | XST Synthesis Report ....................................................................... 439 |
| XST Synthesis Report Contents .................................................................. 439 |
| XST Synthesis Report Navigation ................................................................. 445 |
| XST Synthesis Report Information ............................................................... 446 |

| Chapter 16 | XST Naming Conventions ................................................................. 449 |
| XST Naming Conventions Coding Examples ............................................... 449 |
| XST Net Naming Conventions .................................................................... 452 |
| XST Instance Naming Conventions .............................................................. 453 |
| XST Case Preservation .............................................................................. 453 |
| XST Name Generation Control ................................................................... 453 |

| Appendix | Additional Resources ........................................................................ 455 |
Chapter 1

Introduction

About XST

The Xilinx Synthesis Technology (XST) software:
- Is the Xilinx® proprietary logic synthesis solution
- Is available in:
  - ISE® Design Suite
  - PlanAhead™
- Can run as a standalone tool in command-line mode

The XST software:
1. Takes the description of a design in a Hardware Description Language (HDL) (VHDL or Verilog)
2. Converts it to a synthesized netlist of Xilinx technology-specific logical resources

The synthesized netlist, representing a logical view of the design, is then:
1. Processed by the design implementation tool chain
2. Converted into a physical representation
3. Converted to a bitstream file to program Xilinx devices

For more information, see:
Xilinx Synthesis Technology (XST) - Frequently Asked Questions (FAQ)

Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>LRM</td>
<td>Language Reference Manual</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>EDIF</td>
<td>Electronic Data Interchange Format</td>
</tr>
<tr>
<td>LSO</td>
<td>Library Search Order</td>
</tr>
<tr>
<td>XST</td>
<td>Xilinx® Synthesis Technology</td>
</tr>
<tr>
<td>XCF</td>
<td>XST Constraint File</td>
</tr>
</tbody>
</table>
Creating and Synthesizing an XST Project

Unlike other synthesis tools, XST separates information about the design from information specifying how XST should process it.

- Information about the design is stored in a Hardware Description Language (HDL) Synthesis Project.
- Synthesis parameters are provided in an XST Script file.

An HDL Synthesis Project:

- Is an ASCII text file
- Lists the various HDL source files that compose the design
- Specifies a separate HDL source file on each line
- Usually has a .prj extension

The syntax is:

\[
\text{<hdh_language>} \text{<compilation_library>} \text{<source_file>}
\]

where

- \text{hdh_language} specifies whether the designated HDL source file is written in VHDL or Verilog. This field allows you to create mixed VHDL and Verilog language projects.
- \text{compilation_library} specifies the logic library where the HDL is compiled. The default logic library is \text{work}.
- \text{source_file} specifies the HDL source file. It can use an absolute or a relative path. A relative path is relative to the location of the HDL synthesis project file.

HDL Synthesis Project File Coding Example

The following HDL Synthesis Project file example uses relative paths:

```
vhdl   work       my_vhdl1.vhd
verilog work       my_vlog1.v
vhdl   my_vlog_lib  ../my_other_srsdir/my_vlog2.vhd
verilog my_vlog_lib my_vlog2.v
```

When run from XST in ISE® Design Suite, XST automatically creates an HDL Synthesis Project with a .prj extension in the project directory. Entries are added to the project file whenever you add an HDL source file to the project.

For more information, see:

ISE Design Suite Help
Creating an HDL Synthesis Project

Unlike other synthesis tools, XST separates information about the design from information specifying how XST should process it.

- Information about the design is stored in a Hardware Description Language (HDL) Synthesis Project.
- Synthesis parameters are provided in an XST Script file.

An HDL Synthesis Project:

- Is an ASCII text file
- Lists the various HDL source files that compose the design
- Specifies a separate HDL source file on each line
- Usually has a .prj extension

The syntax is:

```
<hdl_language> <compilation_library> <source_file>
```

where

- `hdl_language` specifies whether the designated HDL source file is written in VHDL or Verilog. This field allows you to create mixed VHDL and Verilog language projects.
- `compilation_library` specifies the logic library where the HDL is compiled. The default logic library is `work`.
- `source_file` specifies the HDL source file. It can use an absolute or a relative path. A relative path is relative to the location of the HDL synthesis project file.

HDL Synthesis Project File Coding Example

The following HDL Synthesis Project file example uses relative paths:

```
vhdl  work  my_vhdl1.vhd
verilog work my_vlg1.v
vhdl  my_vhdl_lib ../my_other_srcdir/my_vhdl2.vhd
verilog my_vlg_lib my_vlg2.v
```

When run from XST in ISE® Design Suite, XST automatically creates an HDL Synthesis Project with a .prj extension in the project directory. Entries are added to the project file whenever you add an HDL source file to the project.

For more information, see:

ISE Design Suite Help

If you run XST from the command line, you must create the HDL synthesis project file manually. For XST to load the HDL synthesis project, you must provide an Input File Name (\texttt{-ifn}) switch on the run command line. The switch tells XST the location of the HDL synthesis project file.
Running XST in ISE Design Suite

To run XST in ISE® Design Suite:
1. Create a new project (File > New Project)
2. Import Hardware Description Language (HDL) source files (Project > Add Copy of Source)
3. In Design > Hierarchy, select the top-level block of the design
4. If ISE Design Suite did not select the correct block as the top-level block:
   a. Select the correct block
   b. Right-click Select Set as Top Module
   c. Right-click Processes > Synthesize-XST
5. To view all available synthesis options, select Process > Properties.
6. To start synthesis:
   a. Right-click
   b. Select Run (Or double-click Synthesize–XST).

For more information, see:
ISE Design Suite Help

Running XST in Command Line Mode

This section discusses Running XST in Command Line Mode, and includes:
- Running XST as a Standalone Tool
- Running XST Interactively
- Running XST in Scripted Mode
- XST Script Files
- XST Commands
- Improving Readability of an XST Script File

Running XST as a Standalone Tool

XST can run as a standalone tool. In command line mode, XST runs as part of a scripted design implementation instead of running in the graphical environment of ISE® Design Suite.

Setting Environment Variables

Before running XST in command line mode, set the following environment variables to point to the correct Xilinx® software installation directory. The following example is for 64-bit Linux.

```
setenv XILINX
setenv PATH $XILINX/bin/lin64:$PATH
setenv LD_LIBRARY_PATH $XILINX/lib/lin64:$LD_LIBRARY_PATH
```

Running XST in Command Line Mode

To run XST in command line mode:
- On Linux, run xst
- On Windows, run xst.exe

The XST command line syntax is:

XST Command Line Options

XST command line options include:

- **–ifn**
  Designates the XST script file containing the commands to execute.
  - If the –ifn switch is omitted, XST runs interactively.
  - If the –ifn switch is specified, XST runs in scripted mode.

- **–ofn**
  Forces redirection of the XST log to a directory and file of your choice. By default, the XST log is written to an .srp file in the work directory.

- **-intstyle**
  Controls reporting on the standard output. If you run XST in command line mode, see Silent Mode.

- **-filter**
  Enables limited message filtering in command line mode.

Using Message Filtering in Command Line Mode

To use message filtering in command line mode:

1. Synthesize your design once in command line mode without any message filtering.
2. Run the Xilinx xreport graphical tool as follows:

   ```
   xreport -config example.xreport -reports_dir . -filter example.filter example &
   ```

   The directory defined by `-reports_dir` should be the same directory in which the XST log file was created. The above example assumes that the XST log file generated from the initial run is named `example.srp`, and that the log file is located in the same directory in which XST was invoked. For more information about this command, run `xreports –h`.

3. Select Design Overview > Summary.
5. Select Design Overview > Synthesis Messages to display messages from the XST log file.
6. Select the messages to be filtered.
7. Right click.
8. Select either Filter All Instances of This Message, or Filter This Instance Only
   The message filter configuration is saved in the file `example.filter`.
9. Run XST again in command line mode using the –filter switch:

   ```
   xst ... -filter example.filter ...
   ```

For more advanced filtering, or to re-enable previously disabled messages:

1. Right click in the Synthesis Messages pane.
2. Select Edit Message Filters.
Running XST Interactively

If you run XST without the \texttt{–ifn} option, you can enter instructions at the XST command prompt. The \texttt{–ifn} option has no effect in interactive mode, since no XST log file is created.

Running XST in Scripted Mode

Rather than typing or pasting commands at the command prompt, Xilinx\textregistered{} recommends that you create an XST script file containing the desired commands and options. When you run XST as part of a scripted design implementation flow, you must either manually prepare an XST script file in advance, or automatically generate it on the fly.

XST Script Files

An XST script file:

\begin{itemize}
  \item Is an ASCII text file
  \item Contains one or more XST commands. Each command can include various options.
  \item Is passed to XST by the \texttt{–ifn} option:
  \begin{verbatim}
  xst \texttt{–ifn} myscript.xst
  \end{verbatim}
\end{itemize}

There is no mandatory file extension for XST script files. ISE\textregistered{} Design Suite creates XST script files with an \texttt{xst} extension.

XST Commands

XST recognizes the following commands:

\begin{itemize}
  \item XST Run Command
  \item XST Set Command
  \item XST Script Command
  \item XST Help Command
\end{itemize}

You can control some commands with options. If you use an option incorrectly, XST issues an error message.

\texttt{ERROR:Xst:1361 - Syntax error in command run for option "–ofmt" : parameter "EDN" is not allowed.}

XST Run Command

The XST \texttt{run} command:

\begin{itemize}
  \item Is the main synthesis command.
  \item Allows you to run synthesis in its entirety, beginning with the parsing of the Hardware Description Language (HDL) source files, and ending with the generation of the final netlist.
  \item Can be used to run HDL Parsing and Elaboration only in order to verify language compliance, or to pre-compile HDL files.
  \item Can be used only once per script file.
\end{itemize}

The syntax is:

\begin{verbatim}
run option_1 value option_2 value ...
\end{verbatim}

The \texttt{run} command is not case sensitive except for option values that designate elements of the HDL description (for example the top-level module of the design). You can specify an option in either lowercase or uppercase. For example, option values of \texttt{yes} and \texttt{YES} are treated identically.
The following tables list mandatory and optional settings for the run command. For additional command line options, see:

- Chapter 10, XST General Constraints
- Chapter 11, XST HDL Constraints
- Chapter 12, XST FPGA Constraints (Non-Timing)
- Chapter 13, XST Timing Constraints

### XST Run Command Mandatory Settings

<table>
<thead>
<tr>
<th>Option</th>
<th>Command Line Name</th>
<th>Option Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input File Name</td>
<td>-ifn</td>
<td>Relative or absolute path to an HDL Synthesis Project file.</td>
</tr>
<tr>
<td>Output File Name</td>
<td>-ofn</td>
<td>Relative or absolute path to a file where the post-synthesis NGC netlist is to be saved. The .ngc extension may be omitted.</td>
</tr>
<tr>
<td>Target Device</td>
<td>-P</td>
<td>A specific device, such as xc6vlx240t-ff1759-1, or a generic device family, such as Virtex®-6 devices.</td>
</tr>
<tr>
<td>Top Module Name</td>
<td>-top</td>
<td>Name of the VHDL entity or Verilog module describing the top level of your design. If you are using a separate VHDL configuration declaration to bind component instantiations to design entities and architectures, the value should be the name of the configuration.</td>
</tr>
</tbody>
</table>

### XST Run Command Optional Settings

<table>
<thead>
<tr>
<th>Option</th>
<th>Command Line Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VHDL Top Level Architecture (name of the specific VHDL architecture to be tied to the top level VHDL entity. Not applicable if the top level of your design is described in Verilog.)</td>
<td>-ent</td>
</tr>
<tr>
<td>Optimization Goal</td>
<td>-opt_mode</td>
</tr>
<tr>
<td>Optimization Effort</td>
<td>-opt_level</td>
</tr>
<tr>
<td>Power Reduction</td>
<td>-power</td>
</tr>
<tr>
<td>Use Synthesis Constraints File</td>
<td>-iuc</td>
</tr>
<tr>
<td>Synthesis Constraints File</td>
<td>-uc</td>
</tr>
<tr>
<td>Keep Hierarchy</td>
<td>-keep_hierarchy</td>
</tr>
<tr>
<td>Netlist Hierarchy</td>
<td>-netlist_hierarchy</td>
</tr>
<tr>
<td>Global Optimization Goal</td>
<td>-glob_opt</td>
</tr>
<tr>
<td>Generate RTL Schematic</td>
<td>-rtlview</td>
</tr>
<tr>
<td>Read Cores</td>
<td>-read_cores</td>
</tr>
<tr>
<td>Cores Search Directories</td>
<td>-sd</td>
</tr>
<tr>
<td>Write Timing Constraints</td>
<td>-write_timing_constraints</td>
</tr>
<tr>
<td>Cross Clock Analysis</td>
<td>-cross_clock_analysis</td>
</tr>
<tr>
<td>Hierarchy Separator</td>
<td>-hierarchy_separator</td>
</tr>
<tr>
<td>Bus Delimiter</td>
<td>-bus_delimiter</td>
</tr>
<tr>
<td>LUT-FF Pairs Utilization Ratio</td>
<td>-slice_utilization_ratio</td>
</tr>
<tr>
<td>BRAM Utilization Ratio</td>
<td>-bram_utilization_ratio</td>
</tr>
<tr>
<td>Option</td>
<td>Command Line Name</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>DSP Utilization Ratio</td>
<td>-dsp_utilization_ratio</td>
</tr>
<tr>
<td>Case</td>
<td>-case</td>
</tr>
<tr>
<td>Library Search Order</td>
<td>-lso</td>
</tr>
<tr>
<td>Verilog Include Directories</td>
<td>-vlgincdir</td>
</tr>
<tr>
<td>Generics</td>
<td>-generics</td>
</tr>
<tr>
<td>Verilog Macros</td>
<td>-define</td>
</tr>
<tr>
<td>FSM Extraction</td>
<td>-fsm_extract</td>
</tr>
<tr>
<td>FSM Encoding Algorithm</td>
<td>-fsm_encoding</td>
</tr>
<tr>
<td>Safe Implementation</td>
<td>-safe_implementation</td>
</tr>
<tr>
<td>Case Implementation Style</td>
<td>-vlgcase</td>
</tr>
<tr>
<td>FSM Style</td>
<td>-fsm_style</td>
</tr>
<tr>
<td>RAM Extraction</td>
<td>-ram_extract</td>
</tr>
<tr>
<td>RAM Style</td>
<td>-ram_style</td>
</tr>
<tr>
<td>ROM Extraction</td>
<td>-rom_extract</td>
</tr>
<tr>
<td>ROM Style</td>
<td>-rom_style</td>
</tr>
<tr>
<td>Automatic BRAM Packing</td>
<td>-auto_bram_packing</td>
</tr>
<tr>
<td>Shift Register Extraction</td>
<td>-shreg_extract</td>
</tr>
<tr>
<td>Shift Register Minimum Size</td>
<td>-shreg_min_size</td>
</tr>
<tr>
<td>Resource Sharing</td>
<td>-resource_sharing</td>
</tr>
<tr>
<td>Use DSP Block</td>
<td>-use_dsp48</td>
</tr>
<tr>
<td>Asynchronous To Synchronous</td>
<td>-async_to_sync</td>
</tr>
<tr>
<td>Add I/O Buffers</td>
<td>-iobuf</td>
</tr>
<tr>
<td>Max Fanout</td>
<td>-max_fanout</td>
</tr>
<tr>
<td>Number of Clock Buffers</td>
<td>-bufg</td>
</tr>
<tr>
<td>Register Duplication</td>
<td>-register_duplication</td>
</tr>
<tr>
<td>Equivalent Register Removal</td>
<td>-equivalent_register_removal</td>
</tr>
<tr>
<td>Register Balancing</td>
<td>-register_balancing</td>
</tr>
<tr>
<td>Move First Flip-Flop Stage</td>
<td>-move_first_stage</td>
</tr>
<tr>
<td>Move Last Flip-Flop Stage</td>
<td>-move_last_stage</td>
</tr>
<tr>
<td>Pack I/O Registers into IOBs</td>
<td>-iob</td>
</tr>
<tr>
<td>LUT Combining</td>
<td>-lc</td>
</tr>
<tr>
<td>Reduce Control Sets</td>
<td>-reduce_control_sets</td>
</tr>
<tr>
<td>Use Clock Enable</td>
<td>-use_clock_enable</td>
</tr>
<tr>
<td>Use Synchronous Set</td>
<td>-use_sync_set</td>
</tr>
<tr>
<td>Use Synchronous Reset</td>
<td>-use_sync_reset</td>
</tr>
<tr>
<td>Optimize Instantiated Primitives</td>
<td>-optimize_primitives</td>
</tr>
</tbody>
</table>
Chapter 2: Creating and Synthesizing an XST Project

XST Set Command

Use the set command to set preferences before invoking the run command.

```plaintext
set -option_name [option_value]
```

XST set command options are shown in the following table.

For more information, see:
Chapter 9, XST Design Constraints

### XST Set Command Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>-tmpdir</td>
<td>Location of all temporary files generated by XST during a session</td>
<td>Any valid path to a directory</td>
</tr>
<tr>
<td>-xsthdpdir</td>
<td>Work Directory (location of all files resulting from HDL compilation)</td>
<td>Any valid path to a directory</td>
</tr>
<tr>
<td>-xsthdpini</td>
<td>HDL Library Mapping File (.INI file)</td>
<td>file_name</td>
</tr>
</tbody>
</table>

XST Script Command

In interactive mode, the script command loads and executes an XST script file.

The syntax is:

```plaintext
script script_file_name
```

The script command provides an absolute or relative path to the XST script files.

XST Help Command

Use the help command to view:

- Supported Families
- All Commands for a Specific Device
- Specific Commands for a Specific Device

### Supported Families

For a list of supported families, type help at the command line with no argument.

```plaintext
help
```

XST issues the following message:

```plaintext
---> help ERROR:Xst:1356 - Help : Missing "-arch ".
Please specify what family you want to target
available families:
spartan6
virtex6
```

### All Commands for a Specific Device

For a list of all commands for a specific device, type the following at the command line:

```plaintext
help -arch family_name
```

where

family_name is a supported device family
Chapter 2: Creating and Synthesizing an XST Project

For example, to view a list of all commands for Virtex®-6 devices, type:

```
help -arch virtex6
```

**Specific Commands for a Specific Device**

For information about a specific command for a specific device, type the following at the command line:

```
help -arch family_name -command command_name
```

where

- `family_name` is a supported device family
- `command_name` is one of the following commands:
  - `run`
  - `set`
  - `time`

For example, to see information about the `run` command for Virtex-6 devices type:

```
help -arch virtex6 -command run
```

**Improving Readability of an XST Script File**

Observe the following rules to improve the readability of an XST script file, especially if you use many options to run synthesis:

- Each option-value pair is on a separate line.
- The first line contains only the `run` command without any options.
- There are no blank lines in the middle of the command.
- Each line containing an option-value pair begins with a dash (`-`).
  - `-ifn`
  - `-ifmt`
  - `-ofn`
- Each option has one value.
- There are no options without a value.
- The value for a given option can be:
  - Predefined by XST (for example, `yes` or `no`)
  - Any string, such as a file name or a name of the top level entity
    Options such as `-vlgincdir` accept several directories as values.
    Separate the directories with spaces, and enclose them in braces `{...}`.
    `-vlgincdir {c:\vlgl c:\vlg2}`
    For more information, see:
    Names With Spaces in Command Line Mode
  - An integer
- Use the pound (`#`) character to comment out options, or to place additional comments in the script file.
Example XST Script File

```
run
-ifn myproject.prj
-ofn myproject.ngc
-ofmt NGC
-p virtex6
# -opt_mode area
-opt_mode speed
-opt_level 1
```

**XST Output Files**

This section discusses XST Output Files, including XST Typical Output Files, XST Temporary Output Files, and Names With Spaces in XST Command Line Mode.

**XST Typical Output Files**

XST typically generates the following output files:

- Output NGC netlist (.ngc)
  - In ISE® Design Suite, the NGC file is created in the project directory.
  - In command line mode, the NGC file is created in the current directory, or in any other directory specified by `run -ofn`.
- Register Transfer Level (RTL) netlist for the RTL Viewer (.ngr)
- Synthesis log file (.srp)
- Temporary files

**XST Temporary Output Files**

In command line mode, XST generates temporary files in the XST temp directory.

**Default XST temp Directory**

<table>
<thead>
<tr>
<th>System</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workstations</td>
<td>/tmp</td>
</tr>
<tr>
<td>Windows</td>
<td>The directory specified by either the TEMP or TMP environment variable</td>
</tr>
</tbody>
</table>

To change the XST temp directory, run `set -tmpdir <directory>` at the XST prompt, or in an XST script file.

Hardware Description Language (HDL) compilation files are generated in the temp directory. The default temp directory is the xst subdirectory of the current directory.

**Tip** The temp directory contains the files resulting from the compilation of all VHDL and Verilog files during all XST sessions. Eventually, the number of files stored in the temp directory can severely impact CPU performance. Since XST does not automatically clean the temp directory, Xilinx® recommends that you manually clean the temp directory regularly.

**Names With Spaces in Command Line Mode**

XST supports file and directory names with spaces in command line mode. Enclose file or directory names containing spaces in double quotes:

```
“C:\my project”
```
The command line syntax for options supporting multiple directories (-sd and -vlgincdir) has changed. Enclose multiple directories in braces {...}:

```sh
e-vlgincdir {"C:\my project" C:\temp}
```

In earlier releases of XST, multiple directories were included in double quotes. XST still supports this convention, provided directory names do not contain spaces. Xilinx® recommends that you change existing scripts to the new syntax.
XST VHDL Language Support

For more information, see:

• IEEE VHDL Language Reference Manual (LRM)
• Chapter 9, XST Design Constraints, especially VHDL Attributes

Advantages of VHDL

VHDL offers a broad set of constructs to compactly describe complicated logic. VHDL allows you to:

• Describe the structure of a system — how it is decomposed into subsystems, and how those subsystems are interconnected.
• Specify the function of a system using familiar programming language forms.
• Simulate a system design before it is implemented and programmed in hardware.
• Easily produce a detailed, device-dependent version of a design to be synthesized from a more abstract specification. This allows you to concentrate on more strategic design decisions, and reduce the overall time to market.

VHDL IEEE Support

XST features a VHDL IEEE 1076-1993 fully compliant parsing and elaboration engine.

XST supports non-LRM compliant constructs when the construct:

• Is supported by most synthesis and simulation tools
• Greatly simplifies coding
• Does not cause problems during synthesis
• Does not negatively impact quality of results

For example, the LRM does not allow instantiation with a port map where a formal port is a buffer and the corresponding effective port is an out (and vice-versa). XST does support such instantiation.

VHDL Data Types

This section discusses VHDL Data Types, including VHDL Unsupported Data Types and VHDL Supported Data Types.

Some types are part of predefined packages. For information on where they are compiled, and how to load them, see VHDL Predefined Packages.
VHDL Unsupported Data Types

VHDL supports the real type defined in the standard package only for the purpose of performing calculations, such as the calculation of generics values. You cannot define a synthesizable object of type real.

VHDL Supported Data Types

This section discusses VHDL Supported Data Types, and includes:

- VHDL Predefined Enumerated Types
- VHDL User-Defined Enumerated Types
- VHDL Bit Vector Types
- VHDL Integer Types
- VHDL Multi-Dimensional Array Types
- VHDL Record Types

VHDL Predefined Enumerated Types

The following predefined VHDL enumerated types are supported for hardware description:

- The bit type, defined in the standard package
  Allowed values are 0 (logic zero) and 1 (logic 1)
- The boolean type, defined in the standard package
  Allowed values are false and true
- The std_logic type defined in the IEEE std_logic_1164 package
  The following table lists allowed values and their interpretation by XST.

### std_logic Allowed Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
<th>What XST does</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>uninitialized</td>
<td>Not accepted by XST</td>
</tr>
<tr>
<td>X</td>
<td>unknown</td>
<td>Treated as don’t care</td>
</tr>
<tr>
<td>0</td>
<td>low</td>
<td>Treated as logic zero</td>
</tr>
<tr>
<td>1</td>
<td>high</td>
<td>Treated as logic one</td>
</tr>
<tr>
<td>Z</td>
<td>high impedance</td>
<td>Treated as high impedance</td>
</tr>
<tr>
<td>W</td>
<td>weak unknown</td>
<td>Not accepted by XST</td>
</tr>
<tr>
<td>L</td>
<td>weak low</td>
<td>Treated identically to 0</td>
</tr>
<tr>
<td>H</td>
<td>weak high</td>
<td>Treated identically to 1</td>
</tr>
<tr>
<td>-</td>
<td>don’t care</td>
<td>Treated as don’t care</td>
</tr>
</tbody>
</table>
XST-Supported Overloaded Enumerated Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Defined In IEEE Package</th>
<th>SubType Of</th>
<th>Contains Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>std_ulogic</td>
<td>std_logic_1164</td>
<td>N/A</td>
<td>• same nine values as std_logic&lt;br&gt;• does not contain predefined resolution functions</td>
</tr>
<tr>
<td>X01</td>
<td>std_logic_1164</td>
<td>std_ulogic</td>
<td>X, 0, 1</td>
</tr>
<tr>
<td>X01Z</td>
<td>std_logic_1164</td>
<td>std_ulogic</td>
<td>X, 0, 1, Z</td>
</tr>
<tr>
<td>UX01</td>
<td>std_logic_1164</td>
<td>std_ulogic</td>
<td>U, X, 0, 1</td>
</tr>
<tr>
<td>UX01Z</td>
<td>std_logic_1164</td>
<td>std_ulogic</td>
<td>U, X, 0, Z</td>
</tr>
</tbody>
</table>

VHDL User-Defined Enumerated Types

You can create your own enumerated types, usually to describe the states of a Finite State Machine (FSM).

VHDL User-Defined Enumerated Types Coding Example

type STATES is (START, IDLE, STATE1, STATE2, STATE3) ;

VHDL Bit Vector Types

XST supports the following VHDL bit vector types for hardware description:

- The bit_vector type, defined in the standard package, models a vector of bit elements.
- The std_logic_vector type, defined in the IEEE std_logic_1164 package, models a vector of std_logic elements.

The following overloaded types are also available:

- The std_ulogic_vector type, defined in the IEEE std_logic_1164 package
- The unsigned type, defined in the IEEE std_logic_arith package
- The signed type, defined in the IEEE std_logic_arith package

VHDL Integer Types

The integer type is a predefined VHDL type. By default, XST implements an integer on 32 bits. For a more compact implementation, define the exact range of applicable values as follows:

type MSB is range 8 to 15

You can also take advantage of the predefined natural and positive types, overloading the integer type.
VHDL Multi-Dimensional Array Types

XST supports multi-dimensional array types, with no restriction on the number of dimensions. However, Xilinx® recommends that you describe no more than three dimensions. Objects of multi-dimensional array type that you can describe are:

- Signals
- Constants
- Variables

Objects of multi-dimensional array type can be:

- Passed to functions
- Used in component instantiations

Fully Constrained Array Type Coding Examples

An array type must be fully constrained in all dimensions.

```vhdl
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB12 is array (11 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB12;
```

You can also declare an array as a matrix.

```vhdl
subtype TAB13 is array (7 downto 0,4 downto 0) of STD_LOGIC_VECTOR (8 downto 0);
```

The following coding examples demonstrate the uses of multi-dimensional array signals and variables in assignments.

Consider the following declarations:

```vhdl
subtype WORD8 is STD_LOGIC_VECTOR (7 downto 0);
type TAB05 is array (4 downto 0) of WORD8;
type TAB03 is array (2 downto 0) of TAB05;
signal WORD_A : WORD8;
signal TAB_A, TAB_B : TAB05;
signal TAB_C, TAB_D : TAB03;
constant CNST_A : TAB03 := (
("00000000","01000001","01000010","10000011","00001100"),
("00100000","00100001","00101010","10100011","00101100"),
("01000010","01000101","01000111","01000100","01000100"));
```

You can now specify:

- A multi-dimensional array signal or variable
  ```vhdl
  TAB_A <= TAB_B; TAB_C <= TAB_D; TAB_C <= CNST_A;
  ```
- An index of one array
  ```vhdl
  TAB_A (5) <= WORD_A; TAB_C (1) <= TAB_A;
  ```
- Indexes of the maximum number of dimensions
  ```vhdl
  TAB_A (5) (0) <= '1'; TAB_C (2) (5) (0) <= '0'
  ```
- A slice of the first array
  ```vhdl
  TAB_A (4 downto 1) <= TAB_B (3 downto 0);
  ```
- An index of a higher level array and a slice of a lower level array
  ```vhdl
  TAB_C (2) (5) (3 downto 0) <= TAB_B (3) (4 downto 1); TAB_D (0) (4) (2 downto 0)
  \ \ <= CNST_A (5 downto 3)
  ```
Add the following declaration:

```vhdl
subtype MATRIX15 is array(4 downto 0, 2 downto 0) of STD_LOGIC_VECTOR(7 downto 0);

signal MATRIX_A : MATRIX15;
```

You can now specify:

- A multi-dimensional array signal or variable
  ```vhdl
  MATRIXA <= CNST_A
  ```
- An index of one row of the array
  ```vhdl
  MATRIXA (5) <= TAB_A;
  ```
- Indexes of the maximum number of dimensions
  ```vhdl
  MATRIXA (5,0) (0) <= '1';
  ```

indexes can be variable.

### VHDL Record Types

XST supports VHDL record types. A record type can be described as:

```vhdl
type mytype is record
  field1 : std_logic;
  field2 : std_logic_vector(3 downto 0)
end record;
```

- A field of a record types can also be of type record.
- Constants can be record types.
- Record types cannot contain attributes.
- XST supports aggregate assignments to record signals.
VHDL Objects

This section discuss VHDL objects, including VHDL Signals, VHDL Variables, and VHDL Constants.

VHDL Signals

Declare VHDL signals in:

- An architecture declarative part
  Use VHDL signals anywhere within that architecture.
- A block
  Use VHDL signals within that block.

Assign VHDL signals with the <= signal assignment operator:

```vhdl
signal sig1 : std_logic;
sig1 <= '1';
```

VHDL Variables

VHDL variables are:

- Declared in a process or a subprogram.
- Used within that process or subprogram.
- Assigned with the := assignment operator.

```vhdl
variable var1 : std_logic_vector (7 downto 0); var1 := "01010011";
```

VHDL Constants

You can declare VHDL constants in any declarative region, which can then be used within that region. Their values cannot be changed once declared.

```vhdl
signal sig1 : std_logic_vector (5 downto 0);constant init0 : std_logic_vector (5 downto 0) := "010111";sig1 <= init0;
```
VHDL Operators

Supported VHDL operators are shown in Supported/Unsupported VHDL Operators later in this chapter. This section provides examples on how to use each shift operator.

- The **SLL (Shift Left Logic)** operator
  
  ```vhdl
  sig1 <= A(4 downto 0) sll 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= A(2 downto 0) & "00";
  ```

- The **SRL (Shift Right Logic)** operator
  
  ```vhdl
  sig1 <= A(4 downto 0) srl 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= "00" & A(4 downto 2);
  ```

- The **SLA (Shift Left Arithmetic)** operator
  
  ```vhdl
  sig1 <= A(4 downto 0) sla 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= A(2 downto 0) & A(0) & A(0);
  ```

- The **SRA (Shift Right Arithmetic)** operator
  
  ```vhdl
  sig1 <= A(4 downto 0) sra 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= A(2 downto 0) & A(4 downto 2);
  ```

- The **ROL (Rotate Left)** operator
  
  ```vhdl
  sig1 <= A(4 downto 0) rol 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= A(2 downto 0) & A(4 downto 3);
  ```

- The **ROR (Rotate Right)** operator
  
  ```vhdl
  A(4 downto 0) ror 2
  ```

  is logically equivalent to:
  
  ```vhdl
  sig1 <= A(1 downto 0) & A(4 downto 2);```
VHDL Entity and Architecture Descriptions

This section discusses VHDL Entity and Architecture Descriptions, and includes:

- VHDL Circuit Descriptions
- VHDL Entity Declarations
- VHDL Architecture Declarations
- VHDL Component Instantiation
- VHDL Recursive Component Instantiation
- VHDL Component Configuration
- VHDL Generics
- Conflicts Among VHDL Generics and Attributes

VHDL Circuit Descriptions

A VHDL circuit description (design unit) consists of two parts:

- Entity declaration
  - Provides the external view of the circuit
  - Describes objects visible from the outside, including the circuit interface, such as the I/O ports and generics
- Architecture
  - Provides the internal view of the circuit
  - Describes the circuit behavior or structure

VHDL Entity Declarations

The I/O ports of the circuit are declared in the entity. Each port has a:

- name
- mode
  - in
  - out
  - inout
  - buffer
- type

While ports are usually constrained, they can also be left unconstrained in the entity declaration. If left unconstrained, their width is defined at instantiation when the connection between formal ports and actual signals is made. Unconstrained ports allow you to create different instantiations of the same entity, defining different port widths.

However, Xilinx® recommends that you:

- Define ports that are constrained through generics.
- Apply different values of those generics at instantiation.
- Do not have an unconstrained port on the top-level entity.

Array types of more than one-dimension are not accepted as ports.

The entity declaration can also declare generics.

For more information, see:

VHDL Generics
NOT RECOMMENDED Coding Example WITH Buffer Port Mode

Xilinx does not recommend using buffer port mode. Although VHDL allows buffer port mode when a signal is used both internally and as an output port (when there is only one internal driver), buffer ports are a potential source of errors during synthesis, and complicate validation of post-synthesis results through simulation.

```vhdl
entity alu is
  port(
    CLK : in   STD_LOGIC;
    A   : in   STD_LOGIC_VECTOR(3 downto 0);
    B   : in   STD_LOGIC_VECTOR(3 downto 0);
    C   : out  STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
begin
  process
    if rising_edge(CLK) then
      C <= UNSIGNED(A) + UNSIGNED(B) UNSIGNED(C);
    end if;
  end process;
end behavioral;
```

RECOMMENDED Coding Example WITHOUT Buffer Port Mode

In the NOT RECOMMENDED Coding Example WITH Buffer Port Mode above, signal C, used both internally and as an output port, has been modeled with a buffer mode. Every level of hierarchy in the design that can be connected to C must also be declared as a buffer. To drop the buffer mode in this example, insert a dummy signal and declare port C as an output, as shown in the following coding example.

```vhdl
entity alu is
  port(
    CLK : in   STD_LOGIC;
    A   : in   STD_LOGIC_VECTOR(3 downto 0);
    B   : in   STD_LOGIC_VECTOR(3 downto 0);
    C   : out  STD_LOGIC_VECTOR(3 downto 0));
end alu;

architecture behavioral of alu is
  -- dummy signal
  signal C_INT : STD_LOGIC_VECTOR(3 downto 0);
begin
  C <= C_INT;
  process
    if rising_edge(CLK) then
      C_INT <= A and B and C_INT;
    end if;
  end process;
end behavioral;
```

VHDL Architecture Declarations

You can declare internal signals in the architecture. Each internal signal has a:

- name
- type
Chapter 3: XST VHDL Language Support

VHDL Architecture Declaration Coding Example

library IEEE;
use IEEE.std_logic_1164.all;

entity EXAMPLE is
  port ( 
    A, B, C : in std_logic;
    D, E : out std_logic );
end EXAMPLE;

architecture ARCHI of EXAMPLE is 
  signal T : std_logic;
begin
  ...
end ARCHI;

VHDL Component Instantiation

Component instantiation allows you to instantiate a design unit (component) inside another design unit in order to create a hierarchically structured design description.

To perform component instantiation:

1. Create the design unit (entity and architecture) modeling the functionality to be instantiated.
2. Declare the component to be instantiated in the declarative region of the parent design unit architecture.
3. Instantiate and connect this component in the parent design unit's architecture body.
4. Map (connect) formal ports of the component to actual signals and ports of the parent design unit.

The main elements of a component instantiation statement are:

- **label**
  - Identifies the instance

- **association list**
  - Introduced by the reserved **port map** keyword
  - Ties formal ports of the component to actual signals or ports of the parent design unit

- **optional association list**
  - Introduced by the reserved **generic map** keyword
  - Provides actual values to formal generics defined in the component

XST supports unconstrained vectors in component declarations.
Chapter 3: XST VHDL Language Support

VHDL Component Instantiation Coding Example

The following coding example shows the structural description of a half adder composed of four `nand2` components:

```vhdl
--
-- A simple component instantiation example
-- Involves a component declaration and the component instantiation itself
--
-- File: VHDL_Language_Support/instantiation/instantiation_simple.vhd
--
entity sub is
  generic (WIDTH : integer := 4);
  port (A, B : in BIT_VECTOR(WIDTH-1 downto 0);
        O : out BIT_VECTOR(2*WIDTH-1 downto 0));
end sub;
architecture archi of sub is
begin
  O <= A & B;
end ARCHI;

entity top is
  generic (WIDTH : integer := 2);
  port (X, Y : in BIT_VECTOR(WIDTH-1 downto 0);
        Z : out BIT_VECTOR(2*WIDTH-1 downto 0));
end top;
architecture ARCHI of top is
  component sub -- component declaration
    generic (WIDTH : integer := 2);
    port (A, B : in BIT_VECTOR(WIDTH-1 downto 0);
          O : out BIT_VECTOR(2*WIDTH-1 downto 0));
  end component;
begin
  inst_sub : sub -- component instantiation
    generic map (WIDTH => WIDTH
    )
    port map (A => X,
              B => Y,
              O => Z);
end ARCHI;

VHDL Recursive Component Instantiation

XST supports recursive component instantiation. XST does not support direct instantiation for recursion. To prevent endless recursive calls, the number of recursions is limited by default to 64. Use `-recursion_iteration_limit` to control the number of allowed recursive calls, as shown in the following coding example.

```
Chapter 3: XST VHDL Language Support

VHDL Recursive Component Instantiation Coding Example

```vhdl
-- Recursive component instantiation
-- File: VHDL_Language_Support/instantiation/instantiation_recursive.vhd
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity single_stage is
  generic (
    sh_st: integer:=4);
  port (
    CLK : in std_logic;
    DI : in std_logic;
    DO : out std_logic );
end entity single_stage;

architecture recursive of single_stage is
  component single_stage
    generic (sh_st: integer);
    port (CLK : in std_logic;
          DI : in std_logic;
          DO : out std_logic );
  end component;
  signal tmp : std_logic;
begins
  GEN_FD_LAST: if sh_st=1 generate
    inst_fd: FD port map (D=>DI, C=>CLK, Q=>DO);
  end generate;
  GEN_FD_INTERM: if sh_st /= 1 generate
    inst_fd: FD port map (D=>DI, C=>CLK, Q=>tmp);
    inst_sstage: single_stage
      generic map (sh_st => sh_st-1)
      port map (DI=>tmp, CLK=>CLK, DO=>DO);
  end generate;
end recursive;
```

VHDL Component Configuration

Use a component configuration to explicitly link a component with the appropriate model (entity and architecture pair). XST supports component configuration in the declarative part of the architecture. Use the following syntax:

```vhdl
for instantiation_list : component_name use LibName.entity_Name(Architecture_Name);
```

For example, the following statement indicates that all NAND2 components use the design unit consisting of entity NAND2 and architecture ARCHI, and that is compiled in the work library.

```vhdl
For all : NAND2 use entity work.NAND2(ARCHI);
```

When the configuration clause is missing for a component instantiation, XST links the component to the entity with the same name (and same interface), and the selected architecture to the most recently compiled architecture. If no entity or architecture is found, a black box is generated during synthesis.

In command line mode, you may also use a dedicated configuration declaration to link component instantiations in your design to design entities and architectures. In this case, the value of the mandatory Top Module Name (-top) option in the run command is the configuration name instead of the top level entity name.
For more information, see:
XST Run Command

**VHDL Generics**

VHDL generics:
- Are the equivalent of Verilog parameters.
- Help you create scalable design modelizations.
- Allow you to parameterize functionality such as bus sizes, or the amount of certain repetitive elements in the design unit.
- Allow you to write compact, factorized VHDL code.

For example, for the same functionality that must be instantiated multiple times, but with different bus sizes, you need describe only one design unit with generics, as shown in the VHDL Generic Parameters Coding Example below.

You can declare generic parameters in the entity declaration part. XST supports all types for generics including:
- `integer`
- `boolean`
- `string`
- `real`
- `std_logic_vector`

Declare a generic with a default value.
Chapter 3: XST VHDL Language Support

VHDL Generic Parameters Coding Example

--
-- VHDL generic parameters example
--
-- File: VHDL_Language_Support/generics/generics_1.vhd

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;

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 (17 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);
begin
  U1 : addern generic map (width=>13) port map (X,Y,C1);
  C2 <= C1 & A;
  C3 <= Z & B;
  U2 : addern generic map (width=>18) port map (C2,C3,S);
end bhv;

Conflicts Among VHDL Generics and Attributes

Conflicts can occasionally arise among VHDL generics and attributes since:

- You can apply VHDL generics and attributes to both instances and components in the Hardware Description Language (HDL) source code, and
- You can specify attributes in a constraints file.
XST resolves these conflicts using the following rules of precedence:

- Specifications on an instance (lower level) take precedence over specifications on a component (higher level).

- If a generic and an attribute are applicable to the same instance or the same component, the attribute is considered, regardless of where the generic was specified. Xilinx® does not recommend using both mechanisms to define the same constraint. XST flags such occurrences.

- An attribute specified in the XST Constraint File (XCF) always takes precedence over attributes or generics specified in the VHDL code.

- Security attributes on the block definition always have higher precedence than any other attribute or generic.
VHDL Combinatorial Circuits

XST supports the following VHDL combinatorial circuits:

- VHDL Concurrent Signal Assignments
- VHDL Generate Statements
- VHDL Combinatorial Processes

VHDL Concurrent Signal Assignments

Combinatorial logic can be described using concurrent signal assignments that can be specified in the body of an architecture.

VHDL supports three types of concurrent signal assignments:

- Simple
- Selected (with-select-when)
- Conditional (when-else)

The following principles apply:

- You can describe as many concurrent statements as needed.
- The order of appearance in the architecture is irrelevant.
- All statements are concurrently active.
- The concurrent assignment is re-evaluated when any signal on the right side of the assignment changes value.
- The re-evaluated result is assigned to the signal on the left-hand side.

Simple Signal Assignment VHDL Coding Example

T <= A and B;
Concurrent Selection Assignment VHDL Coding Example

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

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

entity concurrent_selected_assignment is
  generic (
    width: integer := 8);
  port (a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_selected_assignment;

architecture bhv of concurrent_selected_assignment is
begin
  with sel select
    T <= a when "00",
     b when "01",
     c when "10",
     d when others;
end bhv;
```

Concurrent Conditional Assignment (When-Else) VHDL Coding Example

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

entity concurrent_conditional_assignment is
  generic (width: integer := 8);
  port (a, b, c, d : in std_logic_vector (width-1 downto 0);
        sel : in std_logic_vector (1 downto 0);
        T : out std_logic_vector (width-1 downto 0) );
end concurrent_conditional_assignment;

architecture bhv of concurrent_conditional_assignment is
begin
  T <= a when sel = "00" else
       b when sel = "01" else
       c when sel = "10" else
       d;
end bhv;
```

VHDL Generate Statements

This section discusses VHDL For-Generate Statements and VHDL If-Generate Statements.
Chapter 3: XST VHDL Language Support

VHDL For-Generate Statements

Use a for-generate statement to describe repetitive structures. In the following coding example, the for-generate statement describes the calculation of the result and carry out for each bit position of this 8-bit adder.

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

For-Generate Statement VHDL Coding Example

```vhdl
entity for_generate is
  port (  
    A, B : in BIT_VECTOR (0 to 7); 
    CIN : in BIT; 
    SUM : out BIT_VECTOR (0 to 7); 
    COUT : out BIT );
end for_generate;

architecture archi of for_generate is
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;
```

VHDL If-Generate Statements

A typical use of the if-generate statement is to activate distinct parts of the Hardware Description Language (HDL) source code based on the result of a test, such as a test of a generic value. For example, a generic may indicate which Xilinx® FPGA device family is being targeted. An if-generate statement tests the value of this generic against a specific device family, and activates a section of the HDL source code that was written specifically for this device family.

The if-generate statement is supported for static (non-dynamic) conditions.

In the following coding example, a generic N-bit adder with a width ranging between 4 and 32 is described with an if-generate and a for-generate statement.

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
For-Generate Nested in an If-Generate Statement VHDL Coding Example

--
-- A for-generate nested in a if-generate
--
-- File: VHDL_Language_Support/combinatorial/if_for_generate.vhd
--
entity if_for_generate 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 if_for_generate;
architecture archi of if_for_generate is
signal C : BIT_VECTOR (N+1 downto 0);
begin
  IF_N: 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 if_for_generate;
end archi;
VHDL Combinatorial Processes

This section discusses VHDL Combinatorial Processes, and includes:

- About VHDL Combinatorial Processes
- VHDL Variable and Signal Assignments
- VHDL If-Else Statements
- VHDL Case Statements
- VHDL For-Loop Statements

About VHDL Combinatorial Processes

Combinatorial logic can be modeled with a process. A process is combinatorial when signals assigned in the process are explicitly assigned a new value every time the process is executed. No such signal should implicitly retain its current value.

Hardware inferred from a combinatorial process does not involve any memory elements. When all assigned signals in a process are always explicitly assigned in all possible paths within a process block, the process is combinatorial. A signal that is not explicitly assigned in all branches of an if or case statement typically leads to a latch inference. When XST infers unexpected latches, examine the Hardware Description Language (HDL) source code and look for a signal that is not explicitly assigned.

A combinatorial process has a sensitivity list appearing within parentheses after the process keyword. A process is activated if an event (value change) appears on one of the sensitivity list signals. For a combinatorial process, this sensitivity list must contain:

- All signals in conditions (for example, if and case)
- All signals on the right-hand side of an assignment

If one or more signals is missing from the sensitivity list, XST:

- Issues a warning message
- Adds the missing signals to the sensitivity list

In this case, the synthesis results can differ from the initial design specification. To avoid problems during simulation, explicitly add all missing signals in the HDL source code and re-run synthesis.

A process can contain local variables.

VHDL Variable and Signal Assignments

This section gives coding examples for VHDL Variable and Signal Assignments.
Variable and Signal Assignment VHDL Coding Example One

The following coding example illustrates how to assign a signal within a process.

```
--
-- Signal assignment in a process
--
-- File: VHDL_Language_Support/signals_variables/signal_in_process.vhd
--
entity signal_in_process is
  port (A, B : in BIT; S : out BIT);
end signal_in_process;
architecture archi of signal_in_process is
begin
  process (A, B)
  begin
    S <= '0';
    if ((A and B) = '1') then
      S <= '1';
    end if;
  end process;
end archi;
```

Variable and Signal Assignment VHDL Coding Example Two

A process can also contain local variables. Variables are declared and used within a process. They are generally not visible outside the process.

```
--
-- Variable and signal assignment in a process
--
-- File: VHDL_Language_Support/signals_variables/variable_in_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity variable_in_process is
  port (A,B : in std_logic_vector (3 downto 0);
        ADD_SUB : in std_logic;
        S : out std_logic_vector (3 downto 0));
end variable_in_process;
architecture archi of variable_in_process is
begin
  process (A, B, ADD_SUB)
  variable AUX : std_logic_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;
```

VHDL If-Else Statements

If-else and if-elsif-else statements use true-false conditions to execute statements.

- If the expression evaluates to true, the if branch is executed.
- If the expression evaluates to false, x, or z, the else branch is executed.
A block of multiple statements can be executed in an if or else branch, using begin and end keywords.

If-else statements can be nested.

**If-Else Statement VHDL Coding Example**

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

entity mux4 is
  port (
    a, b, c, d : in std_logic_vector (7 downto 0);
    sel1, sel2 : in std_logic;
    outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
  process (a, b, c, d, sel1, sel2)
  begin
    if (sel1 = '1') then
      if (sel2 = '1') then
        outmux <= a;
      else
        outmux <= b;
      end if;
    else
      if (sel2 = '1') then
        outmux <= c;
      else
        outmux <= d;
      end if;
    end if;
  end process;
end behavior;
```

**VHDL Case Statements**

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The case statement evaluates the branches in the order they are written. The first branch that evaluates to true is executed. If none of the branches match, the default branch is executed.
Case Statement VHDL Coding Example

library IEEE;
use IEEE.std_logic_1164.all;

entity mux4 is
  port ( 
    a, b, c, d : in std_logic_vector (7 downto 0); 
    sel : in std_logic_vector (1 downto 0); 
    outmux : out std_logic_vector (7 downto 0));
end mux4;

architecture behavior of mux4 is
begin
process (a, b, c, d, sel)
begin
  case sel is
    when "00" => outmux <= a;
    when "01" => outmux <= b;
    when "10" => outmux <= c;
    when others => outmux <= d; -- case statement must be complete
  end case;
end process;
end behavior;

VHDL For-Loop Statements

The for statement is supported for:

• Constant bounds
• Stop test condition using any of the following operators:
  – <
  – <=
  – >
  – >=
• Next step computation falling within one of the following specifications:
  – var = var + step
  – var = var - step

  where
  ♦ var is the loop variable
  ♦ step is a constant value
• Next and exit statements
Chapter 3: XST VHDL Language Support

For-Loop Statement VHDL Coding Example

```vhdl
--
-- For-loop example
--
-- File: VHDL_Language_Support/combinatorial/for_loop.vhd
--
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 countzeros;

architecture behavior of countzeros is
  signal Count_Aux: std_logic_vector (2 downto 0);
begin
  process (a, Count_Aux)
  begin
    Count_Aux <= "000";
    for i in a'range loop
      if (a(i) = '0') then
        Count_Aux <= Count_Aux + 1;
      end if;
    end loop;
    Count <= Count_Aux;
  end process;
end behavior;
```

---

XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices

VHDL Sequential Logic

This section discusses VHDL Sequential Logic and includes:

- VHDL Sequential Processes With a Sensitivity List
- VHDL Sequential Processes Without a Sensitivity List
- VHDL Initial Values and Operational Set/Reset
- VHDL Default Initial Values on Memory Elements

VHDL Sequential Processes With a Sensitivity List

A process is sequential (as opposed to combinatorial) when some assigned signals are not explicitly assigned in all paths within the process. In this case, the hardware generated has an internal state or memory (flip-flops or latches). Describing sequential logic using a process with a sensitivity list includes:

- A sensitivity list that contains the clock signal and any optional signal controlling the sequential element asynchronously (asynchronous set/reset)
- An if statement that models the clock event
- Modelization of any asynchronous control logic (asynchronous set/reset) is done before the clock event statement
- Modelization of the synchronous logic (data, optional synchronous set/reset, optional clock enable) is done in the clock event if branch

The syntax is:

```
process (<sensitivity list>)
begin
  <asynchronous part>
  <clock event>
  <synchronous part>
end;
```

The clock event statement can be described for a **rising edge** clock as:

```
If clk'event and clk = '1' then
```

The clock event statement can be described for a **falling edge** clock as:

```
If clk'event and clk = '0' then
```

For greater clarity, you can instead use the VHDL'93 IEEE standard **rising_edge** and **falling_edge** functions. The above statements become:

```
If rising_edge(clk) then
If falling_edge(clk) then
```

If XST detects that a signal has been omitted from the sensitivity list, it issues a warning. Missing signals are automatically added to the list. Xilinx® recommends adding the missing signals to the HDL source code. Failure to do so can cause difficulties when validating your synthesized solution through simulation.

Xilinx recommends using the sensitivity-list based description style to describe sequential logic.

For more information, see:

*Chapter 7, XST HDL Coding Techniques*, which describes macro inference of such functions as registers and counters.
VHDL Sequential Processes Without a Sensitivity List

XST allows the description of sequential logic using a wait statement. In this case, the sequential processes is described without a sensitivity list.

- The same process cannot have both a sensitivity list and a wait statement.
- There can be only one wait statement in the process.
- The wait statement must be the first statement of the process.
- The condition in the wait statement describes the sequential logic clock.

VHDL Sequential Process Using a Wait Statement Coding Example

```vhdl
process
begin
  wait until rising_edge(clk);
  q <= d;
end process;
```

Describing a Clock Enable in the Wait Statement Coding Example

A clock enable can be described in the wait statement together with the clock.

```vhdl
process
begin
  wait until rising_edge(clk) and clken = '1';
  q <= d;
end process;
```

Describing a Clock Enable After the Wait Statement Coding Example

You can also describe the clock enable separately.

```vhdl
process
begin
  wait until rising_edge(clk);
  if clken = '1' then
    q <= d;
  end if;
end process;
```

Besides the clock enable, this coding method also allows you to describe synchronous control logic, such as a synchronous reset or set. You cannot describe a sequential element with asynchronous control logic using a process without a sensitivity list. Only a process with a sensitivity list allows such functionality. XST does not allow description of a latch based on a wait statement. For greater flexibility, Xilinx® recommends describing synchronous logic using a process with a sensitivity list.

VHDL Initial Values and Operational Set/Reset

In VHDL, you can initialize registers when you declare them.

The initialization value:

- Is a constant.
- May be generated from a function call (for example, loading initial values from an external data file).
- Cannot depend on earlier initial values.
- Can be a parameter value propagated to a register.
Initializing Registers VHDL Coding Example One

The following coding example specifies a power-up value, to which the sequential element is initialized when the circuit goes live, and the circuit global reset is applied.

```vhdl
signal arb_onebit : std_logic := '0';
signal arb_priority : std_logic_vector(3 downto 0) := "1011";
```

Initializing Registers VHDL Coding Example Two

You can also initialize sequential elements operationally, describing set/reset values and local control logic. To do so, assign a value to a register when the register reset line goes to the appropriate value, as shown in the following coding example.

```vhdl
process (clk, rst)
begin
  if rst='1' then
    arb_onebit <= '0';
  end if;
end process;
```

For more information about the advantages and disadvantages of operational set/reset, and the advantages and disadvantages of asynchronous versus synchronous set/reset, see Flip-Flops and Registers.

Initializing Registers VHDL Coding Example Three

The following coding example mixes power-up initialization and operational reset.

```vhdl
--
-- Register initialization
-- Specifying initial contents at circuit powes-up
-- Specifying an operational set/reset
--
-- File: VHDL_Language_Support/initial/initial_1.vhd
--
luggage ieee;
use ieee.std_logic_1164.all;

entity initial_1 is
  Port (  
    clk, rst : in std_logic;
    din : in std_logic;
    dout : out std_logic);
end initial_1;

architecture behavioral of initial_1 is
  signal arb_onebit : std_logic := '1'; -- power-up to vcc
begin

  process (clk)
  begin
    if (rising_edge(clk)) then
      if rst='1' then -- local synchronous reset
        arb_onebit <= '0';
      else
        arb_onebit <= din;
      end if;
    end if;
  end process;

  dout <= arb_onebit;
end behavioral;
```
Chapter 3: XST VHDL Language Support

VHDL Default Initial Values on Memory Elements

Every memory element in a Xilinx® FPGA device must come up in a known state. For this reason, in certain cases XST does not apply IEEE standards for initial values. In the previous coding example, if arb_onebit is not initialized to 1 (one), XST assigns it a default of 0 (zero) as its initial state. In this case, XST does not follow the IEEE standard, where U is the default for std_logic. This process of initialization is the same for both registers and RAMs.

Where possible, XST adheres to the IEEE VHDL standard when initializing signal values. If no initial values are supplied in the VHDL code, XST uses the default values (where possible) as shown in the XST column in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>IEEE</th>
<th>XST</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>std_logic</td>
<td>U</td>
<td>0</td>
</tr>
<tr>
<td>bit_vector (3 downto 0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>std_logic_vector (3 downto 0)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>integer (unconstrained)</td>
<td>integer'left</td>
<td>integer'left</td>
</tr>
<tr>
<td>integer range 7 downto 0</td>
<td>integer'left = 7</td>
<td>integer'left = 7 (coded as 111)</td>
</tr>
<tr>
<td>integer range 0 to 7</td>
<td>integer'left = 0</td>
<td>integer'left = 0 (coded as 000)</td>
</tr>
<tr>
<td>Boolean</td>
<td>FALSE</td>
<td>FALSE (coded as 0)</td>
</tr>
<tr>
<td>enum (S0,S1,S2,S3)</td>
<td>type'left = S0</td>
<td>type'left = S0 (coded as 000)</td>
</tr>
</tbody>
</table>

Unconnected output ports default to the values shown in the XST column. If the output port has an initial condition, XST ties the unconnected output port to the explicitly defined initial condition.

According to the IEEE VHDL specification, input ports cannot be left unconnected. As a result, XST issues an error message if an input port is not connected. Even the open keyword is not sufficient for an unconnected input port.

VHDL Functions and Procedures

VHDL functions and procedures let you handle blocks that are used multiple times in a design. Functions and procedures can be declared in:

- The declarative part of an entity
- An architecture
- A package

A function or procedure consists of:

- A declarative part
- A body
The declarative part specifies:

- Input parameters
- Output and inout parameters (procedures only)
- Output and inout parameters (procedures only)

These parameters can be unconstrained. They are not constrained to a given bound. The content is similar to the combinatorial process content. Resolution functions are not supported except the one defined in the IEEE std_logic_1164 package.

### Function Declared Within a Package VHDL Coding Example

The following coding example shows a function declared within a package. The ADD function declared here is a single bit adder. This function is called four times with the proper parameters in the architecture to create a 4-bit adder.

```vhdl
--
-- Declaration of a function in a package
--
-- File: VHDL_Language_Support/functions_procedures/function_package_1.vhd
--
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;
```

---

Chapter 3: XST VHDL Language Support

The XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices

UG687 (v 13.1) March 1, 2011

www.xilinx.com

53
Chapter 3: XST VHDL Language Support

Procedure Declared Within a Package VHDL Coding Example

Following is the same example using a procedure instead.

```vhdl
--
-- Declaration of a procedure in a package
--
-- File: VHDL_Language_Support/functions_procedures/procedure_package_1.vhd
--
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;

desity 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;
```
VHDL Assert Statements

XST supports VHDL Assert statements. Assert statements help you debug your design, enabling you to detect undesirable conditions such as:

- Bad values for generics, constants, and generate conditions
- Bad values for parameters in called functions

For any failed condition in an Assert statement, depending on the severity level, XST either:

- Issues a warning message, or
- Rejects the design and issues an error message

XST supports the Assert statement only with static condition.

The following coding example contains a block (SINGLE_SRL) which describes a shift register. The size of the shift register depends on the SRL_WIDTH generic value. The Assert statement ensures that the implementation of a single shift register does not exceed the size of a single Shift Register LUT (SRL).

Since the size of the SRL is 16 bit, and XST implements the last stage of the shift register using a flip-flop in a slice, the maximum size of the shift register cannot exceed 17 bits. The SINGLE_SRL block is instantiated twice in the entity named TOP, the first time with SRL_WIDTH equal to 13, and the second time with SRL_WIDTH equal to 18.

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Use of an Assert Statement for Design Rule Checking VHDL Coding Example

-- Use of an assert statement for design rule checking
-- File: VHDL_Language_Support/asserts/asserts_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity SINGLE_SRL is
  generic (SRL_WIDTH : integer := 24);
  port (       clk : in std_logic;
              inp : in std_logic;
              outp : out std_logic);
end SINGLE_SRL;

architecture beh of SINGLE_SRL is
  signal shift_reg : std_logic_vector (SRL_WIDTH-1 downto 0);
beging
  assert SRL_WIDTH <= 17
    report "The size of Shift Register exceeds the size of a single SRL"
    severity FAILURE;
  process (clk)
  begin
    if rising_edge(clk) then
      shift_reg <= shift_reg (SRL_WIDTH-2 downto 0) & inp;
    end if;
  end process;
  outp <= shift_reg(SRL_WIDTH-1);
end beh;

library ieee;
use ieee.std_logic_1164.all;

entity TOP is
  port (        clk : in std_logic;
              inp1, inp2 : in std_logic;
              outp1, outp2 : out std_logic);
end TOP;

architecture beh of TOP is
  component SINGLE_SRL is
    generic (SRL_WIDTH : integer := 16);
    port (        clk : in std_logic;
                inp : in std_logic;
                outp : out std_logic);
  end component;
begin
  inst1: SINGLE_SRL
    generic map (SRL_WIDTH => 13)
    port map(
      clk => clk,
      inp => inp1,
      outp => outp1);
  inst2: SINGLE_SRL
    generic map (SRL_WIDTH => 18)
    port map(
      clk => clk,
      inp => inp2,
      outp => outp2);
end beh;
XST issues the following error message.

```
HDL Elaboration

Elaborating entity <TOP> (architecture <beh>) from library <work>.

Elaborating entity <SINGLE_SRL> (architecture <beh>) with generics from library <work>.

ERROR:HDLC:1242 - "VHDL_Language_Support/asserts/asserts_1.vhd"

Line 15: "The size of Shift Register exceeds the size of a single SRL": exiting elaboration
"VHDL_Language_Support/asserts/asserts_1.vhd"

Line 4. netlist SINGLE_SRL(18)(beh) remains a blackbox, due to errors in its contents
```
VHDL Libraries and Packages

This section discusses VHDL Libraries and Packages and includes:

- VHDL Libraries
- VHDL Predefined Packages
- Defining Your Own VHDL Packages
- Accessing VHDL Packages

VHDL Libraries

A library is a directory in which design units (entity or architectures and packages) are compiled. Each VHDL and Verilog source file is compiled into a designated library.

Creating an HDL Synthesis Project describes the syntax of the Hardware Description Language (HDL) synthesis project file, and explains how to specify the library into which the contents of an HDL source file is compiled.

A design unit that was compiled into a library can be invoked from any VHDL source file, provided that you have referenced it through a library clause.

The syntax is:

```
library library_name;
```

The work library is the default library, and does not require a library clause. To change the name of the default library, use run -work_lib.

The physical location of the default library, and of any other user-defined library, is a subdirectory with the same name located under a directory defined by the Work Directory (–xsthdpdir) constraint.

VHDL Predefined Packages

XST supports the following VHDL predefined packages:

- VHDL Predefined Standard Packages
- VHDL Predefined IEEE Packages
- VHDL Predefined IEEE Real Type and IEEE math_real Packages

These packages:

- Are defined in the std and ieee standard libraries.
- Are pre-compiled.
- Need not be user-compiled.
- Can be directly included in the VHDL code.

VHDL Predefined Standard Packages

The standard package defines basic VHDL types, including:

- bit
- bit_vector
- integer
- natural
- real
- boolean

The standard package is included by default.
VHDL Predefined IEEE Packages

XST supports the following IEEE packages, which define more common data types, functions, and procedures:

- **numeric_bit**
  Defines unsigned and signed vector types based on *bit*, as well as all overloaded arithmetic operators, conversion functions, and extended functions for these types.

- **std_logic_1164**
  Defines *std_logic*, *std_ulogic*, *std_logic_vector*, and *std_ulogic_vector* types, as well as conversion functions based on these types.

- **std_logic_arith** (Synopsys)
  Defines *unsigned* and *signed* vector types based on *std_logic*. Also defines overloaded arithmetic operators, conversion functions, and extended functions for these types.

- **numeric_std**
  Defines unsigned and signed vector types based on *std_logic*. Also defines overloaded arithmetic operators, conversion functions, and extended functions for these types. Equivalent to *std_logic_arith*.

- **std_logic_unsigned** (Synopsys)
  Defines unsigned arithmetic operators for *std_logic* and *std_logic_vector*

- **std_logic_signed** (Synopsys)
  Defines signed arithmetic operators for *std_logic* and *std_logic_vector*

- **std_logic_misc** (Synopsys)
  Defines supplemental types, subtypes, constants, and functions for the *std_logic_1164* package, such as *and_reduce* and *or_reduce*

IEEE packages are pre-compiled in the ieee library.

VHDL Predefined IEEE Fixed and Floating Point Packages

XST supports VHDL predefined IEEE fixed and floating point packages.

- The **fixed_pkg** package:
  - Contains functions for fixed point math.
  - Is already precompiled into the *ieee_proposed* library.
  - Is invoked as follows:
    ♦ use ieee.std_logic_1164.all;
    ♦ use ieee.numeric_std.all;
    ♦ library ieee_proposed;
    ♦ use ieee_proposed.fixed_pkg.all;

- The **float_pkg** package:
  - Contains functions for floating point math.
  - Is already precompiled into the *ieee_proposed* library.
  - Is invoked as follows:
    ♦ use ieee.std_logic_1164.all;
    ♦ use ieee.numeric_std.all;
    ♦ library ieee_proposed;
    ♦ use ieee_proposed.float_pkg.all;
VHDL Predefined IEEE Real Type and IEEE math_real Packages

The \texttt{real} type, as well as functions and procedures in the IEEE \texttt{math_real} package, are supported only for calculations (such as the calculation of generics values). They cannot be used to describe synthesizable functionality.

### VHDL Real Number Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>math_e</td>
<td>$e$</td>
<td>math_log_of_2</td>
<td>ln2</td>
</tr>
<tr>
<td>math_1_over_e</td>
<td>$1/e$</td>
<td>math_log_of_10</td>
<td>ln10</td>
</tr>
<tr>
<td>math_pi</td>
<td>$\pi$</td>
<td>math_log2_of_e</td>
<td>log$_e$</td>
</tr>
<tr>
<td>math_2_pi</td>
<td>$2\pi$</td>
<td>math_log10_of_e</td>
<td>log$_{10}$</td>
</tr>
<tr>
<td>math_1_over_pi</td>
<td>$1/\pi$</td>
<td>math_sqrt_2</td>
<td>$\sqrt{2}$</td>
</tr>
<tr>
<td>math_pi_over_2</td>
<td>$\pi/2$</td>
<td>math_1_oversqrt_2</td>
<td>$1/\sqrt{2}$</td>
</tr>
<tr>
<td>math_pi_over_3</td>
<td>$\pi/3$</td>
<td>math_sqrt_pi</td>
<td>$\sqrt{\pi}$</td>
</tr>
<tr>
<td>math_pi_over_4</td>
<td>$\pi/4$</td>
<td>math_deg_to_rad</td>
<td>$2\pi/360$</td>
</tr>
<tr>
<td>math_3_pi_over_2</td>
<td>$3\pi/2$</td>
<td>math_rad_to_deg</td>
<td>$360/2\pi$</td>
</tr>
</tbody>
</table>

### VHDL Real Number Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>math_e</th>
<th>floor(x)</th>
<th>ceil(x)</th>
<th>exp(x)</th>
<th>cos(x)</th>
<th>cosh(x)</th>
<th>log(x)</th>
<th>tan(x)</th>
<th>tanh(x)</th>
<th>log2(x)</th>
<th>arccos(x)</th>
<th>arccosh(x)</th>
<th>arcsin(x)</th>
<th>arccosh(x)</th>
<th>sinh(x)</th>
<th>sinh(x)</th>
<th>arctanh(x)</th>
<th>log10(x)</th>
<th>arctan(x)</th>
<th>arccosh(x)</th>
<th>log(x,y)</th>
<th>arctan(y,x)</th>
<th>arctanh(x)</th>
<th>sqrt(x)</th>
<th>sqrt(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil(x)</td>
<td>realmax(x,y)</td>
<td>log(x)</td>
<td>exp(x)</td>
<td>cos(x)</td>
<td>cosh(x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>floor(x)</td>
<td>realmin(x,y)</td>
<td>tan(x)</td>
<td>ceil(x)</td>
<td>cos(x)</td>
<td>cosh(x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>round(x)</td>
<td>sqrt(x)</td>
<td>log2(x)</td>
<td>exp(x)</td>
<td>cos(x)</td>
<td>cosh(x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trunc(x)</td>
<td>cbrt(x)</td>
<td>arccos(x)</td>
<td>arccosh(x)</td>
<td>arcsin(x)</td>
<td>arccosh(x)</td>
<td>arcsin(x)</td>
<td>arccosh(x)</td>
<td>arcsin(x)</td>
<td>arccosh(x)</td>
<td>arctanh(x)</td>
<td>log(x,y)</td>
<td>arctan(x)</td>
<td>arccosh(x)</td>
<td>log(x,y)</td>
<td>arctan(y,x)</td>
<td>arctanh(x)</td>
<td>sqrt(x)</td>
<td>sqrt(x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Defining Your Own VHDL Packages

You can create your own VHDL packages to define:

- Types and subtypes
- Constants
- Functions and procedures
- Component declarations

Defining your own VHDL packages allows access to shared definitions and models from other parts of your project.

Defining a VHDL package requires a:

- Package declaration
  Declares each of the elements listed above
- Package body
  Describes the functions and procedures declared in the package declaration
Package Declaration Syntax

```vhdl
package mypackage is

    type mytype is
        record
            first : integer;
            second : integer;
        end record;

    constant myzero : mytype := (first => 0, second => 0);

    function getfirst (x : mytype) return integer;

end mypackage;
```

Package Body Syntax

```vhdl
package body mypackage is

    function getfirst (x : mytype) return integer is
        begin
            return x.first;
        end function;

end mypackage;
```

Accessing VHDL Packages

To access definitions of a VHDL package:

- Include the library in which the package has been compiled with a library clause, and
- Designate the package, or a specific definition contained in the package, with a use clause.

Use the following syntax:

```vhdl
library library_name;
use library_name.package_name.all;
```

Insert these lines immediately before the entity or architecture in which you use the package definitions. Because the work library is the default library, you can omit the library clause if the designated package has been compiled into this library.
VHDL File Type Support

This section discusses VHDL File Type Support, and includes:

- VHDL File Read and File Write Capability
- Loading Memory Contents from an External File
- Writing to a File for Debugging
- Rules for Debugging Using Write Operations

VHDL File Read and File Write Capability

XST supports a limited VHDL File Read and File Write capability. File Read capability can be used for initializing memories from an external data file. For more information, see:

Specifying Initial Contents in an External Data File

File Write capability can be used for:

- Debugging
- Writing a specific constant or generic value to an external file

The `textio` package:

- Is available in the `std` library
- Provides basic text-based File I/O capabilities
- Defines the following procedures for file I/O operations
  - `readline`
  - `read`
  - `writeline`
  - `write`

The `std_logic_textio` package:

- Is available in the IEEE library
- Provides extended text I/O support for other data types, overloading the `read` and `write` procedures as shown in the following table.
### VHDL File Type Support

<table>
<thead>
<tr>
<th>Function</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>file (type text only)</td>
<td>standard</td>
</tr>
<tr>
<td>access (type line only)</td>
<td>standard</td>
</tr>
<tr>
<td>file_open (file, name, open_kind)</td>
<td>standard</td>
</tr>
<tr>
<td>file_close (file)</td>
<td>standard</td>
</tr>
<tr>
<td>endfile (file)</td>
<td>standard</td>
</tr>
<tr>
<td>text</td>
<td>std.textio</td>
</tr>
<tr>
<td>line</td>
<td>std.textio</td>
</tr>
<tr>
<td>width</td>
<td>std.textio</td>
</tr>
<tr>
<td>readline (text, line)</td>
<td>std.textio</td>
</tr>
<tr>
<td>readline (line, bit, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, bit)</td>
<td>std.textio</td>
</tr>
<tr>
<td>readline (line, bit_vector, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, bit_vector)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, boolean, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, character, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, character)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, string, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, string)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (file, line)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, bit, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, bit)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, bit_vector, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, bit_vector)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, boolean, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, character, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, character)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, integer, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, integer)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, string, boolean)</td>
<td>std.textio</td>
</tr>
<tr>
<td>write (line, string)</td>
<td>std.textio</td>
</tr>
<tr>
<td>read (line, std_ulogic, boolean)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>read (line, std_ulogic)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>read (line, std_ulogic_vector, boolean)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>read (line, std_ulogic_vector)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>read (line, std_logic_vector, boolean)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>Function</td>
<td>Package</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>read (line, std_logic_vector)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>write (line, std_ulogic, boolean)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>write (line, std_ulogic)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>write (line, std_logic_vector, boolean)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>write (line, std_logic_vector)</td>
<td>ieee.std_logic_textio</td>
</tr>
<tr>
<td>hread</td>
<td>ieee.std_logic_textio</td>
</tr>
</tbody>
</table>

XST supports both implicit and explicit file open and close operations. A file is implicitly opened when declared as follows:

```vhdl
file myfile : text open write_mode is "myfilename.dat"; -- declaration and implicit open
```

Explicitly open and close an external file as follows:

```vhdl
file myfile : text; -- declaration
variable file_status : file_open_status;
...
file_open (file_status, myfile, "myfilename.dat", write_mode); -- explicit open
...
file_close(myfile); -- explicit close
```

### Loading Memory Contents from an External File

For information on loading memory contents from an external file, see:

*Specifying Initial Contents in an External Data File*

### Writing to a File for Debugging Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from `ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip`. Each directory contains a `summary.txt` file listing all examples together with a brief overview.
Writing to a File (Explicit Open/Close) VHDL Coding Example

File write capability is often used for debugging. In the following coding example, write operations are performed to a file that has been explicitly opened.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

entity filewrite_explicitopen is
  generic (data_width: integer:= 4);
  port (clk : in std_logic;
          di : in std_logic_vector (data_width - 1 downto 0);
          do : out std_logic_vector (data_width - 1 downto 0));
end filewrite_explicitopen;

architecture behavioral of filewrite_explicitopen is
  file results : text;
  constant base_const: std_logic_vector(data_width - 1 downto 0):= conv_std_logic_vector(3,data_width);
  constant new_const: std_logic_vector(data_width - 1 downto 0):= base_const + "0100"
begin
  process(clk)
  variable txtline : line;
  variable file_status : file_open_status;
  begin
    file_open (file_status, results, "explicit.dat", write_mode);
    writeln(results, txtline);
    write(txtline,string'("--------------------"));
    writeln(results, txtline);
    write(txtline,string'("Base Const: "));
    write(txtline, base_const);
    writeln(results, txtline);
    write(txtline,string'("New Const: "));
    write(txtline,new_const);
    writeln(results, txtline);
    write(txtline,string'("--------------------"));
    writeln(results, txtline);
    file_close(results);
    if rising_edge(clk) then
      do <= di + new_const;
    end if;
  end process;
end behavioral;
```
Writing to a File (Implicit Open/Close) VHDL Coding Example

You can also rely on an implicit file open.

```vhdl
-- Writing to a file. Implicit open/close
-- File: VHDL_Language_Support/file_type_support/filewrite_implicitopen.vhd

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_TEXTIO.all;
use STD.TEXTIO.all;

datefile implicitopen

entity filewrite_implicitopen is
  generic (data_width: integer := 4);
  port (clk : in std_logic;
        di : in std_logic_vector (data_width - 1 downto 0);
        do : out std_logic_vector (data_width - 1 downto 0));
end filewrite_implicitopen;

architecture behavioral of filewrite_implicitopen is
  file results : text open write_mode is "implicit.dat";
  constant base_const: std_logic_vector(data_width - 1 downto 0):= conv_std_logic_vector(3,data_width);
  constant new_const: std_logic_vector(data_width - 1 downto 0):= base_const + "0100";

begin
  process(clk)
  variable txtline : LINE;
  begin
    write(txtline,string'(------------------------'));
    writeln(results, txtline);
    write(txtline,string'(Base Const: '));
    writeln(txtline,base_const);
    writeln(results, txtline);
    write(txtline,string'(New Const: '));
    writeln(txtline,new_const);
    writeln(results, txtline);
    write(txtline,string'(------------------------'));
    writeln(results, txtline);
    if rising_edge(clk) then
      do <= di + new_const;
    end if;
  end process;
end behavioral;
```

Debugging Using Write Operations

During a std_logic read operation, the only allowed characters are 0 and 1. Other values such as X and Z are not allowed. XST rejects the design if the file includes characters other than 0 and 1, except that XST ignores a blank space character.

Do not use identical names for files in different directories.

Do not use conditional calls to read procedures:

```vhdl
if SEL = '1' then
  read (MY_LINE, A(3 downto 0));
else
  read (MY_LINE, A(1 downto 0));
end if;
```
Chapter 3: XST VHDL Language Support

VHDL Constructs

This section discusses VHDL Constructs, and includes:
• VHDL Design Entities and Configurations
• VHDL Expressions
• VHDL Statements

VHDL Design Entities and Configurations

XST supports VHDL design entities and configurations except as noted below.
• VHDL Entity Headers
  – Generics
    Supported
  – Ports
    Supported, including unconstrained ports
  – Entity Statement Part
    Unsupported
• VHDL Packages
  – STANDARD
  – Type TIME is not supported
• VHDL Physical Types
  – TIME
    Ignored
  – REAL
    Supported, but only in functions for constant calculations
• VHDL Modes
  Linkage
  Unsupported
• VHDL Declarations
  Type
  Supported for
  ♦ Enumerated types
  ♦ Types with positive range having constant bounds
  ♦ Bit vector types
  ♦ Multi-dimensional arrays
• VHDL Objects
  – Constant Declaration
    Supported except for deferred constant
  – Signal Declaration
    Supported except for register and bus type signals
  – Attribute Declaration
    Supported for some attributes, otherwise skipped.
For more information, see:
Chapter 9, XST Design Constraints
Chapter 3: XST VHDL Language Support

- VHDL Specifications
  Supported for some predefined attributes only:
  - HIGHLOW
  - LEFT
  - RIGHT
  - RANGE
  - REVERSE_RANGE
  - LENGTH
  - POS
  - ASCENDING
  - EVENT
  - LAST_VALUE

- Configuration
  Supported only with the all clause for instances list. If no clause is added, XST looks for the entity or architecture compiled in the default library

- Disconnection
  Unsupported

Object names can contain underscores in general (for example, DATA_1), but XST does not allow signal names with leading underscores (for example, _DATA_1).

**VHDL Expressions**

This section discusses VHDL Expressions, including Supported and Unsupported VHDL Operators and Supported and Unsupported VHDL Operands.

**Supported and Unsupported VHDL Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Operators: and, or, nand, nor, xor, xnor, not</td>
<td>Supported</td>
</tr>
<tr>
<td>Relational Operators: =, /=, &lt;, &lt;=, &gt;, &gt;=</td>
<td>Supported</td>
</tr>
<tr>
<td>&amp; (concatenation)</td>
<td>Supported</td>
</tr>
<tr>
<td>Adding Operators: +, -</td>
<td>Supported</td>
</tr>
<tr>
<td>*</td>
<td>Supported</td>
</tr>
<tr>
<td>/</td>
<td>Supported if the right operand is a constant power of 2, or if both operands are constant</td>
</tr>
<tr>
<td>rem</td>
<td>Supported if the right operand is a constant power of 2</td>
</tr>
<tr>
<td>mod</td>
<td>Supported if the right operand is a constant power of 2</td>
</tr>
<tr>
<td>Shift Operators: sll, srl, sla, sra, rol, ror</td>
<td>Supported</td>
</tr>
<tr>
<td>abs</td>
<td>Supported</td>
</tr>
<tr>
<td>**</td>
<td>Supported if the left operand is 2</td>
</tr>
<tr>
<td>Sign: +, -</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Supported and Unsupported VHDL Operands

<table>
<thead>
<tr>
<th>Operand</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Literals</td>
<td>Only integer literals are supported</td>
</tr>
<tr>
<td>Physical Literals</td>
<td>Ignored</td>
</tr>
<tr>
<td>Enumeration Literals</td>
<td>Supported</td>
</tr>
<tr>
<td>String Literals</td>
<td>Supported</td>
</tr>
<tr>
<td>Bit String Literals</td>
<td>Supported</td>
</tr>
<tr>
<td>Record Aggregates</td>
<td>Supported</td>
</tr>
<tr>
<td>Array Aggregates</td>
<td>Supported</td>
</tr>
<tr>
<td>Function Call</td>
<td>Supported</td>
</tr>
<tr>
<td>Qualified Expressions</td>
<td>Supported for accepted predefined attributes</td>
</tr>
<tr>
<td>Types Conversions</td>
<td>Supported</td>
</tr>
<tr>
<td>Allocators</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Static Expressions</td>
<td>Supported</td>
</tr>
</tbody>
</table>

VHDL Statements

VHDL Wait Statements

<table>
<thead>
<tr>
<th>Wait Statement</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait on sensitivity_list until Boolean_expression.</td>
<td>Supported with one signal in the sensitivity list and in the Boolean expression. Multiple Wait statements not supported. Note XST does not support Wait statements for latch descriptions.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait for time_expression.</td>
<td>Unsupported</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Assertion Statement</td>
<td>Supported (only for static conditions)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Signal Assignment Statement</td>
<td>Supported (delay is ignored)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable Assignment Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure Call Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>If Statement</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Case Statement</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Note: XST does not support Wait statements for latch descriptions.
## VHDL Loop Statements

<table>
<thead>
<tr>
<th>Loop Statement</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>for... loop... end loop</td>
<td>Supported for constant bounds only. Disable statements are not supported.</td>
</tr>
<tr>
<td>while... loop... end loop</td>
<td>Supported</td>
</tr>
<tr>
<td>loop ... end loop</td>
<td>Only supported in the particular case of multiple Wait statements</td>
</tr>
<tr>
<td>Next Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Exit Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Return Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Null Statement</td>
<td>Supported</td>
</tr>
</tbody>
</table>

## VHDL Concurrent Statements

<table>
<thead>
<tr>
<th>Concurrent Statement</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>Concurrent Procedure Call</td>
<td>Supported</td>
</tr>
<tr>
<td>Concurrent Assertion Statement</td>
<td>Ignored</td>
</tr>
<tr>
<td>Concurrent Signal Assignment Statement</td>
<td>Supported (no after clause, no transport or guarded options, no waveforms) UNAFFECTED is supported.</td>
</tr>
<tr>
<td>Component Instantiation Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>for-generate</td>
<td>Statement supported for constant bounds only</td>
</tr>
<tr>
<td>if-generate</td>
<td>Statement supported for static condition only</td>
</tr>
</tbody>
</table>
### VHDL Reserved Words

<table>
<thead>
<tr>
<th>abs</th>
<th>access</th>
<th>after</th>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>and</td>
<td>architecture</td>
<td>array</td>
</tr>
<tr>
<td>assert</td>
<td>attribute</td>
<td>begin</td>
<td>block</td>
</tr>
<tr>
<td>body</td>
<td>buffer</td>
<td>bus</td>
<td>case</td>
</tr>
<tr>
<td>component</td>
<td>configuration</td>
<td>constant</td>
<td>disconnect</td>
</tr>
<tr>
<td>downto</td>
<td>else</td>
<td>elsif</td>
<td>end</td>
</tr>
<tr>
<td>entity</td>
<td>exit</td>
<td>file</td>
<td>for</td>
</tr>
<tr>
<td>function</td>
<td>generate</td>
<td>generic</td>
<td>group</td>
</tr>
<tr>
<td>guarded</td>
<td>if</td>
<td>impure</td>
<td>in</td>
</tr>
<tr>
<td>inertial</td>
<td>inout</td>
<td>is</td>
<td>label</td>
</tr>
<tr>
<td>library</td>
<td>linkage</td>
<td>literal</td>
<td>loop</td>
</tr>
<tr>
<td>map</td>
<td>mod</td>
<td>nand</td>
<td>new</td>
</tr>
<tr>
<td>next</td>
<td>nor</td>
<td>not</td>
<td>null</td>
</tr>
<tr>
<td>of</td>
<td>on</td>
<td>open</td>
<td>or</td>
</tr>
<tr>
<td>others</td>
<td>out</td>
<td>package</td>
<td>port</td>
</tr>
<tr>
<td>postponed</td>
<td>procedure</td>
<td>process</td>
<td>pure</td>
</tr>
<tr>
<td>range</td>
<td>record</td>
<td>register</td>
<td>reject</td>
</tr>
<tr>
<td>rem</td>
<td>report</td>
<td>return</td>
<td>rol</td>
</tr>
<tr>
<td>ror</td>
<td>select</td>
<td>severity</td>
<td>signal</td>
</tr>
<tr>
<td>shared</td>
<td>sla</td>
<td>sll</td>
<td>sra</td>
</tr>
<tr>
<td>srl</td>
<td>subtype</td>
<td>then</td>
<td>to</td>
</tr>
<tr>
<td>transport</td>
<td>type</td>
<td>unaffected</td>
<td>units</td>
</tr>
<tr>
<td>until</td>
<td>use</td>
<td>variable</td>
<td>wait</td>
</tr>
<tr>
<td>when</td>
<td>while</td>
<td>with</td>
<td>xnor</td>
</tr>
<tr>
<td>xor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4

XST Verilog Support

Complex circuits are commonly designed using a top-down methodology. Various specification levels are required at each stage of the design process. For example, at the architectural level, a specification can correspond to a block diagram or an Algorithmic State Machine (ASM) chart. A block or ASM stage corresponds to a register transfer block where the connections are \textbf{N-bit} wires, such as:

- Register
- Adder
- Counter
- Multiplexer
- Glue logic
- Finite State Machine (FSM)

A Hardware Description Language (HDL) such as Verilog allows the expression of notations such as ASM charts and circuit diagrams in a computer language.

Verilog provides both behavioral and structural language structures. These structures allow expressing design objects at high and low levels of abstraction. Designing hardware with a language such as Verilog allows using software concepts such as parallel processing and object-oriented programming. Verilog has a syntax similar to C and Pascal, and is supported by XST as IEEE 1364.

Verilog support in XST allows you to describe the global circuit and each block in the most efficient style. Synthesis is then performed with the best synthesis flow for each block. Synthesis in this context is the compilation of high-level behavioral and structural Verilog HDL statements into a flattened gate-level netlist, which can then be used to custom program a programmable logic device such as a Virtex® device. Different synthesis methods are used for arithmetic blocks, glue logic, and Finite State Machine (FSM) components.

This guide assumes that you are familiar with basic Verilog concepts. For more information, see the \textit{IEEE Verilog HDL Reference Manual}.

For more information about XST support for Verilog constructs and meta comments, see:

- Verilog design constraints and options
  \textit{Chapter 9, XST Design Constraints}
- Verilog attribute syntax
  \textit{Verilog 2001 Attributes and Meta Comments} in \textit{Chapter 4, XST Verilog Support}.
- Setting Verilog options in the Process window of ISE® Design Suite
  \textit{Chapter 10, XST General Constraints}

For information about Behavioral Verilog, see \textit{Chapter 5, XST Behavioral Verilog Support}. 
Chapter 4: XST Verilog Support

Verilog Variable Part Selects

Verilog–2001 allows you to use variables to select a group of bits from a vector. A variable part select is defined by the starting point of its range and the width of the vector, instead of being bounded by two explicit values. The starting point of the part select can vary, but the width of the part select remains constant.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ (plus)</td>
<td>The part select increases from the starting point</td>
</tr>
<tr>
<td>- (minus)</td>
<td>The part select decreases from the starting point</td>
</tr>
</tbody>
</table>

Variable Part Selects Verilog Coding Example

```verilog
reg [3:0] data;
reg [3:0] select; // a value from 0 to 7
wire [7:0] byte = data[select +: 8];
```

Structural Verilog Features

Structural Verilog descriptions assemble several blocks of code and allow the introduction of hierarchy in a design. The basic concepts of hardware structure are as follows.

- Component
  Building or basic block
- Port
  Component I/O connector
- Signal
  Corresponds to a wire between components

In Verilog, a component is represented by a design module.

<table>
<thead>
<tr>
<th>Item</th>
<th>View</th>
<th>Describes</th>
</tr>
</thead>
<tbody>
<tr>
<td>declaration</td>
<td>external</td>
<td>What can be seen from the outside, including the component ports</td>
</tr>
<tr>
<td>body</td>
<td>internal</td>
<td>The behavior or the structure of the component</td>
</tr>
</tbody>
</table>

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). The list specifies which actual signals or ports are associated with which local ports of the component declaration.
Verilog provides a large set of built-in logic gates which can be instantiated to build larger logic circuits. The set of logical functions described by the built-in gates includes:

- AND
- OR
- XOR
- NAND
- NOR
- NOT

2-Input XOR Function Verilog Coding Example

```verilog
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

Half Adder Verilog Coding Example

The following coding example shows the structural description of a half adder composed of four, 2-input `nand` modules.

```verilog
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
```

The structural features of Verilog also allow you to design circuits by instantiating pre-defined primitives such as gates, registers and Xilinx® specific primitives such as `CLKDLL` and `BUFG`. These primitives are other than those included in Verilog. These pre-defined primitives are supplied with the XST Verilog libraries (`unisim_comp.v`).
Instantiating an FDC and a BUFG Primitive Verilog Coding Example

module example (sysclk, in, reset, out);
    input sysclk, in, reset;
    output reg out;
    wire sysclk_out;

    FDC register (out, sysclk_out, reset, in); //position based referencing
    BUFG clk (.O(sysclk_out),.I(sysclk)); //name based referencing
    ...

The unisim_comp.v library file supplied with XST includes the definitions for FDC and BUFG.

Verilog Parameters

Verilog parameters:

- Allow you to create parameterized code that can be easily reused and scaled.
- Make code more readable, more compact, and easier to maintain.
- Can be used to describe such functionality as bus sizes, or the amount of certain repetitive elements in the modeled design unit.
- Are constants. For each instantiation of a parameterized module, default parameter values can be overridden.
- Are the equivalent of VHDL generics.

Null string parameters are not supported.

Use Generics (-generics) to redefine Verilog parameters values defined in the top-level design block. This allows you to modify the design configuration without modifying the source code. This feature is useful for such processes as IP core generation and flow testing.

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Verilog Parameters Coding Example

```
//
// A Verilog parameter allows to control the width of an instantitated
// block describing register logic
//
// File: Verilog_Language_Support/parameter/parameter_1.v
//
module myreg (clk, clken, d, q);
    parameter SIZE = 1;
    input         clk, clken;
    input [SIZE-1:0] d;
    output reg [SIZE-1:0] q;

    always @(posedge clk)
        begin
            if (clken)
                q <= d;
        end

endmodule

module parameter_1 (clk, clken, di, do);
    parameter SIZE = 8;
    input          clk, clken;
    input [SIZE-1:0] di;
    output [SIZE-1:0] do;

    myreg #8 inst_reg (clk, clken, di, do);
endmodule
```

Instantiation of the module `lpm_reg` with a instantiation width of 8 causes the instance `buf_373` to be 8 bits wide.
Chapter 4: XST Verilog Support

Verilog Parameters and Generate-For Coding Example

The following example illustrates how to control the creation of repetitive elements using parameters and generate-for constructs.

```verilog
parameter SIZE = 8;
input  clk;
input  si;
output so;
reg [0:SIZE-1] s;
assign so = s[SIZE-1];
always @(posedge clk)
s[0] <= si;
genvar i;
generate
for (i = 1; i < SIZE; i = i+1)
begin : shreg
    always @(posedge clk)
    begin
        s[1] <= s[i-1];
    end
end
generate
endmodule
```

For more information, see:

Generate Loop Statements

Verilog Parameter and Attribute Conflicts

Conflicts occasionally arise since:

- Parameters and attributes can be applied to both instances and modules in the Verilog code, and
- Attributes can also be specified in a constraints file

To resolve these conflicts, XST uses the following rules of precedence:

1. Specifications on an instance (lower level) take precedence over specifications on a module (higher level).
2. If a parameter and an attribute are specified on either the same instance or the same module, the parameter takes precedence. XST issues a warning message.
3. An attribute specified in the XST Constraint File (XCF) takes precedence over attributes or parameters specified in the Verilog code.
When an attribute specified on an instance overrides a parameter specified on a module in XST, a simulation tool can still use the parameter. If that occurs, there will be a simulation mismatch with post-synthesis results.

### Verilog Parameter and Attribute Conflicts Precedence

<table>
<thead>
<tr>
<th></th>
<th>Parameter on an Instance</th>
<th>Parameter on a Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute on an Instance</td>
<td>Apply Parameter (XST issues warning)</td>
<td>Apply Attribute (possible simulation mismatch)</td>
</tr>
<tr>
<td>Attribute on a Module</td>
<td>Apply Parameter</td>
<td>Apply Parameter (XST issues warning)</td>
</tr>
<tr>
<td>Attribute in XCF</td>
<td>Apply Attribute (XST issues warning)</td>
<td>Apply Attribute</td>
</tr>
</tbody>
</table>

Security attributes on the module definition always have higher precedence than any other attribute or parameter.

### Verilog Usage Restrictions in XST

This section discusses Verilog usage restrictions in XST, and supported features for which Xilinx® recommends restrictions. This section includes:

- **Case Sensitivity**
- **Blocking and Nonblocking Assignments**
- **Integer Handling**

#### Verilog Case Sensitivity

XST fully supports Verilog case sensitivity despite the potential of name collisions.

Since Verilog is case sensitive, the names of *modules*, *instances*, and *signals* can theoretically be made unique by changing capitalization. XST can successfully synthesize a design in which *instance* and *signal* names differ only by capitalization. However, when *module* names differ only by capitalization, XST errors out.

Xilinx® recommends that you do not rely on capitalization to make object names unique. Doing so can cause problems in mixed language projects. You may also be unable to apply constraints by means of an XST Constraint File (XCF) file.

#### Blocking and Nonblocking Assignments

XST supports both blocking and non-blocking assignments.

Do not mix blocking and non-blocking assignments. Although synthesized without error by XST, they can cause errors during simulation.

**Unacceptable Coding Example One**

Do not mix blocking and non-blocking assignments to the same signal.

```verilog
always @ (in1)
begin
  if (in2)
    out1 = in1;
  else
    out1 <= in2;
end
```
Unacceptable Coding Example Two
Do not mix blocking and non-blocking assignments for different bits of the same signal.

```verilog
if (in2)
begin
    out1[0] = 1'b0;
    out1[1] <= in1;
end
else
begin
    out1[0] = in2;
    out1[1] <= 1'b1;
end
```

### Integer Handling

XST handles integers differently from other synthesis tools in several instances. They must be coded in a particular way.

#### Integer Handling in Verilog Case Statements

Unsized integers in `case` item expressions can cause unpredictable results. In the following coding example, the `case` item expression `4` is an unsized integer that causes unpredictable results. To avoid problems, size the `4` to `3` bits as follows.

```verilog
reg [2:0] condition1;
always @(condition1)
begin
    case(condition1)
    4 : data_out = 2; // < will generate bad logic
    3'd4 : data_out = 2; // < will work
    endcase
end
```

#### Integer Handling in Verilog Concatenations

Unsized integers in concatenations can cause unpredictable results. If you use an expression that results in an unsized integer, assign the expression to a temporary signal, and use the temporary signal in the concatenation as follows.

```verilog
reg [31:0] temp;
assign temp = 4'b1111 % 2;
assign dout = {12/3,temp,din};
```
Verilog–2001 Attributes and Meta Comments

This section discusses Verilog-2001 Attributes and Verilog Meta Comments.

Verilog-2001 Attributes

XST supports Verilog-2001 attribute statements. Attributes are comments that pass specific information to programs such as synthesis tools. Xilinx® recommends Verilog-2001 Attributes since they are more generally accepted. You can specify Verilog-2001 attributes anywhere for operators or signals within module declarations and instantiations. Although the compiler may support other attribute declarations, XST ignores them.

Use attributes to:

• Set constraints on individual objects, such as:
  – module
  – instance
  – net

• Set the following synthesis constraints
  – Full Case (FULL_CASE)
  – Parallel Case (PARALLEL_CASE)

Verilog Meta Comments

Verilog meta comments:

• Are understood by the Verilog parser
• Are used to:
  – Set constraints on individual objects, such as:
    ♦ module
    ♦ instance
    ♦ net
  – Set directives on synthesis:
    ♦ parallel_case and full_case
    ♦ translate_on and translate_off
    ♦ All tool specific directives (for example, syn_sharing)

• Can be written using the following styles:
  – C-style /* ... */
    C-style comments can be multiple line.
  – Verilog style /// ...
    Verilog style comments end at the end of the line.
XST supports:

- Both C-style and Verilog style meta comments
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)
  ```
  // synthesis translate_on
  // synthesis translate_off
  ```
- Parallel Case (PARALLEL_CASE)
  ```
  // synthesis parallel_case full_case // synthesis parallel_case
  // synthesis full_case
  ```
- Constraints on individual objects

The general syntax is:

```
// synthesis attribute [of] ObjectName [is] AttributeValue
```

**Syntax 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 more information, see:

Chapter 9, XST Design Constraints
Verilog Constructs

This section discusses supported and unsupported Verilog constructs, and includes:

- Verilog Constants
- Verilog Data Types
- Verilog Continuous Assignments
- Verilog Procedural Assignments
- Verilog Design Hierarchies
- Verilog Compiler Directives

Note XST does not allow underscores as the first character of signal names (for example, _DATA_1)

### Verilog Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Supported</td>
</tr>
<tr>
<td>Real</td>
<td>Supported</td>
</tr>
<tr>
<td>Strings</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

### Verilog Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Category</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net types</td>
<td>tri0, tri1, trireg</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Drive strengths</td>
<td>All</td>
<td>Ignored</td>
</tr>
<tr>
<td>Registers</td>
<td>Real and realtime registers</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Named events</td>
<td>All</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

### Verilog Continuous Assignments

<table>
<thead>
<tr>
<th>Continuous Assignment</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive Strength</td>
<td>Ignored</td>
</tr>
<tr>
<td>Delay</td>
<td>Ignored</td>
</tr>
</tbody>
</table>
### Verilog Procedural Assignments

<table>
<thead>
<tr>
<th>Procedural Assignment</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>Supported with limitations. See Behavioral Verilog Assign and Deassign Statements.</td>
</tr>
<tr>
<td>deassign</td>
<td>Supported with limitations. See Behavioral Verilog Assign and Deassign Statements.</td>
</tr>
<tr>
<td>force</td>
<td>Unsupported</td>
</tr>
<tr>
<td>release</td>
<td>Unsupported</td>
</tr>
<tr>
<td>forever statements</td>
<td>Unsupported</td>
</tr>
<tr>
<td>repeat statements</td>
<td>Supported, but repeat value must be constant</td>
</tr>
<tr>
<td>for statements</td>
<td>Supported, but bounds must be static</td>
</tr>
<tr>
<td>delay (#)</td>
<td>Ignored</td>
</tr>
<tr>
<td>event (@)</td>
<td>Unsupported</td>
</tr>
<tr>
<td>wait</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Named Events</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Parallel Blocks</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Specify Blocks</td>
<td>Ignored</td>
</tr>
<tr>
<td>Disable</td>
<td>Supported except in For and Repeat Loop statements.</td>
</tr>
</tbody>
</table>

### Verilog Design Hierarchies

<table>
<thead>
<tr>
<th>Design Hierarchy</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>module definition</td>
<td>Supported</td>
</tr>
<tr>
<td>macromodule definition</td>
<td>Unsupported</td>
</tr>
<tr>
<td>hierarchical names</td>
<td>Unsupported</td>
</tr>
<tr>
<td>defparam</td>
<td>Supported</td>
</tr>
<tr>
<td>array of instances</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Verilog Compiler Directives

<table>
<thead>
<tr>
<th>Compiler Directive</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘celldefine ‘endcelldefine</td>
<td>Ignored</td>
</tr>
<tr>
<td>‘default_nettype</td>
<td>Supported</td>
</tr>
<tr>
<td>‘define</td>
<td>Supported</td>
</tr>
<tr>
<td>‘ifdef ‘else ‘endif</td>
<td>Supported</td>
</tr>
<tr>
<td>‘undef, ‘ifndef, ‘elsif,</td>
<td>Supported</td>
</tr>
<tr>
<td>‘include</td>
<td>Supported</td>
</tr>
<tr>
<td>‘resetall</td>
<td>Ignored</td>
</tr>
<tr>
<td>‘timescale</td>
<td>Ignored</td>
</tr>
<tr>
<td>‘unconnected_drive ‘nounconnected_drive</td>
<td>Ignored</td>
</tr>
<tr>
<td>‘uselib</td>
<td>Unsupported</td>
</tr>
<tr>
<td>‘file, ‘line</td>
<td>Supported</td>
</tr>
</tbody>
</table>

Verilog System Tasks and Functions

This section discusses Verilog System Tasks and Functions and includes:

- Verilog System Tasks and Functions Supported in XST
- Using Conversion Functions
- Loading Memory Contents with File I/O Tasks
- Display Tasks
- Creating Design Rule Checks with $finish
## Verilog System Tasks and Functions

<table>
<thead>
<tr>
<th>System Task or Function</th>
<th>Supported/Unsupported</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$display</td>
<td>Supported</td>
<td>Escape sequences are limited to %d, %b, %h, %o, %c and %s</td>
</tr>
<tr>
<td>$fclose</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$fdisplay</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$fgets</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$finish</td>
<td>Supported</td>
<td>$finish is supported for statically never active conditional branches only</td>
</tr>
<tr>
<td>$fopen</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$fscanf</td>
<td>Supported</td>
<td>Escape sequences are limited to %b and %d</td>
</tr>
<tr>
<td>$fwrite</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$monitor</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>$random</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>$readmemb</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$readmemh</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$signed</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$stop</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>$strobe</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>$time</td>
<td>Ignored</td>
<td></td>
</tr>
<tr>
<td>$unsigned</td>
<td>Supported</td>
<td></td>
</tr>
<tr>
<td>$write</td>
<td>Supported</td>
<td>Escape sequences are limited to %d, %b, %h, %o, %c and %s</td>
</tr>
<tr>
<td>all others</td>
<td>Ignored</td>
<td></td>
</tr>
</tbody>
</table>

The XST Verilog compiler ignores unsupported system tasks.

### Using Conversion Functions

Use the following syntax to call $signed and $unsigned system tasks on any expression:

```
$signed(expr) or $unsigned(expr)
```

The return value from these calls is the same size as the input value. Its sign is forced regardless of any previous sign.

### Loading Memory Contents With File I/O Tasks

The $readmemb and $readmemh system tasks can be used to initialize block memories.

For more information, see:

*Specifying Initial Contents in an External Data File*
Use $readmemb for binary and $readmemh for hexadecimal representation. To avoid possible differences between XST and simulator behavior, Xilinx® recommends that you use index parameters in these system tasks.

$readmemb("rams_20c.data",ram, 0, 7);

## Display Tasks

Display tasks can be used to print information to the console or write it to an external file. You must call these tasks from within initial blocks. XST supports the following subset of escape sequences:

- `%h`
- `%d`
- `%o`
- `%b`
- `%c`
- `%s`

### Verilog $display Syntax Example

The following example shows the syntax for $display that reports the value of a binary constant in decimal.

```verilog
parameter c = 8'b00101010;
initial
begin
  $display("The value of c is %d", c);
end
```

The following information is written to the log file during the HDL Analysis phase:

```
Analyzing top module <example>.
c = 8'b00101010
"foo.v" line 9: $display : The value of c is 42
```

## Creating Design Rule Checks with $finish

Although the $finish simulation control task is primarily intended for simulation, XST partially supports it. This allows you to use $finish to create built-in design rule checks. Design rule checking detects design configurations that are syntactically correct, but which may result in unworkable or otherwise undesired implementations. Using $finish can save significant synthesis and implementation time by forcing an early exit of XST when it detects undesired conditions.

XST ignores $finish if its execution depends on the occurrence of specific dynamic conditions during simulation or operation of the circuit on the board. Only simulation tools can detect such situations. Synthesis tools, including XST, ignore them.

Ignored Use of $finish Verilog Coding Example

//
// Ignored use of $finish for simulation purposes only
//
// File: Verilog_Language_Support/system_tasks/finish_ignored_1.v
//
module finish_ignored_1 (clk, di, do);

    input       clk;
    input      [3:0] di;
    output reg  [3:0] do;

initial
    begin
    do = 4'b0;
    end

always @(posedge clk)
    begin
    if (di < 4'b1100)
        do <= di;
    else
        begin
        $display("%t, di value %d should not be more than 11", $time, di);
        $finish;
        end
    end
endmodule

Occurrences of the $finish system task in dynamically active situations are flagged and ignored.

XST considers a $finish if its execution depends only on static conditions that can be fully evaluated during elaboration of the Verilog source code. Such statically evaluated conditions mainly involve comparison of parameters against expected values. This is typically done in a module initial block as shown below. Use the $display system task in conjunction with $finish to create exit messages to help you locate the root cause of an early exit by XST.
Supported Use of $finish for Design Rule Checking Verilog Coding Example

```verilog
// Supported use of $finish for design rule checking
//
// File: Verilog_Language_Support/system_tasks/finish_supported_1.v
// module finish_supported_1 (clk, di, do);
// parameter integer WIDTH = 4;
// parameter DEVICE = "virtex6";
input clk;
input [WIDTH-1:0] di;
output reg [WIDTH-1:0] do;

initial begin
  if (DEVICE != "virtex6")
    $display("DRC ERROR: Unsupported device family: %s.", DEVICE);
  $finish;
end
if (WIDTH < 8)
  $display("DRC ERROR: This module not tested for data width: %d. Minimum allowed width is 8.", WIDTH);
  $finish;
end

always @(posedge clk)
begin
  do <= di;
end
endmodule
```

XST ignores the $stop Verilog simulation control task.

Verilog Primitives

XST supports Verilog primitives as follows.

- Certain gate-level primitives. The supported syntax is:
  ```verilog
gate_type instance_name (output, inputs,...);
```
  Following is a gate-level primitive instantiations coding example.
  ```verilog
  and U1 (out, in1, in2); bufif1 U2 (triout, data, trienable);
  ```

- All Verilog gate level primitives except as shown in the following table.

### Verilog Gate Level Primitives Supported in XST

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pulldown and pullup</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Drive strength and delay</td>
<td>Ignored</td>
</tr>
<tr>
<td>Arrays of primitives</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

XST does not support Verilog Switch-Level primitives, such as:

- cmos, nmos, pmos, rcmos, rnmos, rpmos
- rtran, rtranif0, rtranif1, tran, tranif0, tranif1
Verilog User Defined Primitive (UDP)

XST supports the Verilog User Defined Primitive (UDP).

With a UDP, the Verilog language offers a modeling technique to describe functionality in the form of a state table. The state table:

- Enumerates all combinations of input values
- Specifies the corresponding values on the circuit’s unique output

The functionality modeled with a UDP can be Combinatorial or Sequential. A UDP can be a convenient technique to model low-complexity functionality, such as simple combinational functions or basic sequential elements. For more elaborate circuit descriptions, Xilinx® recommends using Behavioral Verilog modeling techniques, and leveraging the inference capabilities of XST instead. For an extensive discussion of inference capabilities and coding guidelines in both Verilog and VHDL, see Chapter 7, XST HDL Coding Techniques.

UDP Definition and Instantiation

As with the defining and instantiating of a Verilog module, UDPs must be properly defined before they can be instantiated. A UDP definition is described between the primitive and endprimitive keywords, and may be found anywhere outside the scope of any module-endmodule section. UDPs are instantiated the same as gate-level primitives and user-defined modules. For more information on the Verilog User Defined Primitive (UDP), including syntax rules, see your Verilog Language Reference manual.

Combinatorial UDP

A combinatorial UDP uses the value of its inputs to determine the next value of its output, allowing it to describe any combinatorial function.

Combinatorial UDP Coding Example

```verbatim
// Description and instantiation of a user defined primitive
// combinatorial function
// File: Verilog_Language_Support/user_defined_primitives/udp_combinatorial_1.v
//
primitive myand2 (o, a, b);
   input a, b;
   output o;

   table
      // a b : o
      0 0 : 0;
      0 1 : 0;
      1 0 : 0;
      1 1 : 1;
   endtable
endprimitive

module udp_combinatorial_1 (a, b, c, o);
   input a, b, c;
   output o;
   wire s;
   myand2 i1 (.a(a), .b(b), .o(s));
   myand2 i2 (.a(s), .b(c), .o(o));
endmodule
```
Sequential UDP

A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of its output. A sequential UDP can model both level-sensitive and edge-sensitive behavior, allowing it to describe such sequential elements as flip-flops and latches. An initial value may be specified.

Sequential UDP Coding Example

```verilog
// Description and instantiation of a user defined primitive
// Sequential function
// File: Verilog_Language_Support/user_defined_primitives/udp_sequential_2.v
// primitive mydff (q, d, c);
// input c, d;
// output reg q;
// initial q = 1'b0;
// table
// c d : q : q+
// 0 0 : ? : 0;
// 1 0 : ? : 1;
// 0 1 : ? : 1;
// 1 1 : ? : -;
endtable

endprimitive
module udp_sequential_2 (clk, si, so);
// input clk, si;
// output so;
wire s1, s2;
mydff i1 (.c(clk), .d(si), .q(s1));
mydff i2 (.c(clk), .d(s1), .q(s2));
mydff i3 (.c(clk), .d(s2), .q(so));
endmodule
```
### Verilog Reserved Keywords

Keywords marked with an asterisk (*) are reserved by Verilog, but are not supported by XST.

<table>
<thead>
<tr>
<th>always</th>
<th>and</th>
<th>assign</th>
<th>automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>buf</td>
<td>bufif0</td>
<td>bufif1</td>
</tr>
<tr>
<td>case</td>
<td>casez</td>
<td>casez</td>
<td>cell*</td>
</tr>
<tr>
<td>cmos</td>
<td>config*</td>
<td>deassign</td>
<td>default</td>
</tr>
<tr>
<td>defparam</td>
<td>design*</td>
<td>disable</td>
<td>edge</td>
</tr>
<tr>
<td>else</td>
<td>end</td>
<td>endcase</td>
<td>endconfig*</td>
</tr>
<tr>
<td>endfunction</td>
<td>endgenerate</td>
<td>endmodule</td>
<td>endprimitive</td>
</tr>
<tr>
<td>endspecify</td>
<td>endtable</td>
<td>endtask</td>
<td>event</td>
</tr>
<tr>
<td>for</td>
<td>force</td>
<td>forever</td>
<td>fork</td>
</tr>
<tr>
<td>function</td>
<td>generate</td>
<td>genvar</td>
<td>highz0</td>
</tr>
<tr>
<td>highz1</td>
<td>if</td>
<td>ifnone</td>
<td>incdir*</td>
</tr>
<tr>
<td>include*</td>
<td>initial</td>
<td>inout</td>
<td>input</td>
</tr>
<tr>
<td>instance*</td>
<td>integer</td>
<td>join</td>
<td>large</td>
</tr>
<tr>
<td>liblist*</td>
<td>library*</td>
<td>localparam</td>
<td>macromodule</td>
</tr>
<tr>
<td>medium</td>
<td>module</td>
<td>nand</td>
<td>negedge</td>
</tr>
<tr>
<td>nmos</td>
<td>nor</td>
<td>noshow-cancelled*</td>
<td>not</td>
</tr>
<tr>
<td>notif0</td>
<td>notif1</td>
<td>or</td>
<td>output</td>
</tr>
<tr>
<td>parameter</td>
<td>pmos</td>
<td>posedge</td>
<td>primitive</td>
</tr>
<tr>
<td>pull0</td>
<td>pull1</td>
<td>pullup</td>
<td>pulldown</td>
</tr>
<tr>
<td>pulsestyle- _ondetect*</td>
<td>pulsestyle- _onevent*</td>
<td>rcmos</td>
<td>real</td>
</tr>
<tr>
<td>realtime</td>
<td>reg</td>
<td>release</td>
<td>repeat</td>
</tr>
<tr>
<td>rmos</td>
<td>rpamos</td>
<td>rtran</td>
<td>rtranif0</td>
</tr>
<tr>
<td>rtranif1</td>
<td>scalared</td>
<td>show-cancelled*</td>
<td>signed</td>
</tr>
<tr>
<td>small</td>
<td>specify</td>
<td>specparam</td>
<td>strong0</td>
</tr>
<tr>
<td>strong1</td>
<td>supply0</td>
<td>supply1</td>
<td>table</td>
</tr>
<tr>
<td>task</td>
<td>time</td>
<td>tran</td>
<td>tranif0</td>
</tr>
<tr>
<td>tranif1</td>
<td>tri</td>
<td>tri0</td>
<td>tri1</td>
</tr>
<tr>
<td>triand</td>
<td>trior</td>
<td>trireg</td>
<td>use*</td>
</tr>
<tr>
<td>vectored</td>
<td>wait</td>
<td>wand</td>
<td>weak0</td>
</tr>
<tr>
<td>weak1</td>
<td>while</td>
<td>wire</td>
<td>wor</td>
</tr>
<tr>
<td>xnor</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Verilog 2001 Support in XST

XST supports the following Verilog-2001 features.

- Generate statements
- Combined port/data type declarations
- ANSI-style port list
- Module parameter port lists
- ANSI C style task/function declarations
- Comma separated sensitivity list
- Combinatorial logic sensitivity
- Default nets with continuous assigns
- Disable default net declarations
- Indexed vector part selects
- Multi-dimensional arrays
- Arrays of net and real data types
- Array bit and part selects
- Signed reg, net, and port declarations
- Signed-based integer numbers
- Signed arithmetic expressions
- Arithmetic shift operators
- Automatic width extension past 32 bits
- Power operator
- N sized parameters
- Explicit in-line parameter passing
- Fixed local parameters
- Enhanced conditional compilation
- File and line compiler directives
- Variable part selects
- Recursive Tasks and Functions
- Constant Functions

For more information, see:

XST Behavioral Verilog Support

Variables in Behavioral Verilog can be declared as integer or real. These declarations are intended for use in test code only. Verilog provides data types such as reg and wire for actual hardware description.

The difference between reg and wire depends on whether the variable is given its value in a procedural block (reg) or in a continuous assignment (wire). Both reg and wire have a default width of one bit (scalar). To specify an N-bit width (vectors) for a declared reg or wire, the left and right bit positions are defined in square brackets separated by a colon. In Verilog-2001, both reg and wire data types can be signed or unsigned.

Variable Declarations Coding Example

```verilog
reg [3:0] arb_priority;
wire [31:0] arb_request;
wire signed [8:0] arb_signed;
```

Initial Values

In Verilog-2001, you can initialize registers when you declare them. The initial value specified:

- Is a constant
- Cannot depend on earlier initial values
- Cannot be a function or task call
- Can be a parameter value propagated to the register
- Specifies all bits of a vector

When you give a register an initial value in a declaration, XST sets this value on the output of the register at global reset, or at power up. A value assigned this way is carried in the NGC file as an INIT attribute on the register, and is independent of any local reset.

Initial Values Coding Example One

```verilog
reg arb_onebit = 1'b0;
reg [3:0] arb_priority = 4'b1011;
```

You can also assign a set/reset (initial) value to a register in your Behavioral Verilog code. Assign a value to a register when the register reset line goes to the appropriate value as shown in the following coding example.

Initial Values Coding Example Two

```verilog
always @(posedge clk)
begin
  if (rst)
    arb_onebit <= 1'b0;
end
```
When you set the initial value of a variable in the behavioral code, it is implemented in the design as a flip-flop whose output can be controlled by a local reset. As such, it is carried in the NGC file as an FDP or FDC flip-flop.

Arrays Coding Examples

Verilog allows arrays of reg and wire to be defined as shown below.

Arrays Coding Example One

The following coding example describes an array of 32 elements. Each element is 4-bits wide.

```verilog
reg [3:0] mem_array [31:0];
```

Arrays Coding Example Two

The following coding example describes an array of 64 8-bit wide elements which can be assigned only in structural Verilog code.

```verilog
wire [7:0] mem_array [63:0];
```

Multi-Dimensional Arrays

XST supports multi-dimensional array types of up to two dimensions. Multi-dimensional arrays can be any net or any variable data type. You can code assignments and arithmetic operations with arrays, but you cannot select more than one element of an array at one time. You cannot pass multi-dimensional arrays to system tasks or functions, or to regular tasks or functions.

Multi-Dimensional Array Coding Example One

The following Verilog coding example describes an array of 256 x 16 wire elements of 8-bits each, which can be assigned only in structural Verilog code

```verilog
wire [7:0] array2 [0:255][0:15];
```

Multi-Dimensional Array Coding Example Two

The following Verilog coding example describes an array of 256 x 8 register elements, each 64 bits wide, which can be assigned in Behavioral Verilog code.

```verilog
reg [63:0] regarray2 [255:0][7:0];
```

Data Types

The Verilog representation of the bit data type contains the following values:

- **0**
  - logic zero

- **1**
  - logic one

- **x**
  - unknown logic value

- **z**
  - high impedance
XST supports the following Verilog data types:

- net
- wire
- tri
- triand/wand
- trior/wor
- registers
- reg
- integer
- supply nets
- supply0
- supply1
- constants
- parameter
- Multi-Dimensional Arrays (Memories)

Net and registers can be either:

- Single bit (scalar)
- Multiple bit (vectors)

**Behavioral Verilog Data Types Coding Example**

The following coding example shows sample Verilog data types found in the declaration section of a Verilog module.

```verilog
wire net1;  // single bit net
reg r1;     // single bit register
tri [7:0] bus1; // 8 bit tristate bus
reg [15:0] bus1; // 15 bit register
reg [7:0] mem[0:127]; // 8x128 memory register
parameter state1 = 3'b001; // 3 bit constant
parameter component = "TMS380C16"; // string
```

**Legal Statements**

The following statements (variable and signal assignments) are legal in Behavioral Verilog:

- `variable = expression`
- `if (condition) statement`
- `else statement`
- `case (expression)`
  ```verilog
  expression: statement
  ...
  default: statement
  endcase
  ```
- `for (variable = expression; condition; variable = variable + expression) statement`
- `while (condition) statement`
- `forever statement`
- functions and tasks
All variables are declared as integer or reg. A variable cannot be declared as a wire.

Expressions

An expression involves constants and variables with arithmetic, logical, relational, and conditional operators. Logical operators are further divided into bit-wise and logical, depending on whether they are applied to an expression involving several bits or a single bit.

Supported Operators

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Logical</th>
<th>Relational</th>
<th>Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>&amp;</td>
<td>&lt;</td>
<td>?</td>
</tr>
<tr>
<td>-</td>
<td>&amp;&amp;</td>
<td>==</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
<td>==</td>
</tr>
<tr>
<td>**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>^</td>
<td>&gt;=</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>~</td>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>^=</td>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>^~</td>
<td>~!=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Supported Expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Symbol</th>
<th>Supported/Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replication</td>
<td>{{}}</td>
<td>Supported</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+, -, *, **</td>
<td>Supported</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>Supported only if second operand is a power of 2, or if both operands are constant</td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>Supported only if second operand is a power of 2</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>Supported</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
<td>Supported</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>Supported</td>
</tr>
<tr>
<td>Expression</td>
<td>Symbol</td>
<td>Supported/Unsupported</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>Power</td>
<td>**</td>
<td>Supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Both operands are constants, with the second operand being non-negative.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the first operand is a 2, then the second operand can be a variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• XST does not support the real data type. Any combination of operands that results in a real type causes an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The values X (unknown) and Z (high impedance) are not allowed.</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;, &lt;, &gt;=, &lt;=</td>
<td>Supported</td>
</tr>
<tr>
<td>Logical Negation</td>
<td>!</td>
<td>Supported</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
<td>Supported</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical Equality</td>
<td>==</td>
<td>Supported</td>
</tr>
<tr>
<td>Logical Inequality</td>
<td>!=</td>
<td>Supported</td>
</tr>
<tr>
<td>Case Equality</td>
<td>===</td>
<td>Supported</td>
</tr>
<tr>
<td>Case Inequality</td>
<td>!==</td>
<td>Supported</td>
</tr>
<tr>
<td>Bitwise Negation</td>
<td>~</td>
<td>Supported</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>&amp;</td>
<td>Supported</td>
</tr>
<tr>
<td>Bitwise Inclusive OR</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Bitwise Exclusive OR</td>
<td>^</td>
<td>Supported</td>
</tr>
<tr>
<td>Bitwise Equivalence</td>
<td><del>^, ^</del></td>
<td>Supported</td>
</tr>
<tr>
<td>Reduction AND</td>
<td>&amp;</td>
<td>Supported</td>
</tr>
<tr>
<td>Reduction NAND</td>
<td>~&amp;</td>
<td>Supported</td>
</tr>
<tr>
<td>Reduction OR</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Reduction NOR</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>Reduction XOR</td>
<td></td>
<td>Supported</td>
</tr>
<tr>
<td>Reduction XNOR</td>
<td><del>^, ^</del></td>
<td>Supported</td>
</tr>
<tr>
<td>Left Shift</td>
<td>&lt;&lt;</td>
<td>Supported</td>
</tr>
<tr>
<td>Right Shift Signed</td>
<td>&gt;&gt;&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>Left Shift Signed</td>
<td>&gt;&gt;&gt;&lt;</td>
<td>Supported</td>
</tr>
<tr>
<td>Right Shift</td>
<td>&gt;&gt;</td>
<td>Supported</td>
</tr>
<tr>
<td>Conditional</td>
<td>?:</td>
<td>Supported</td>
</tr>
<tr>
<td>Event OR</td>
<td>or, ', '</td>
<td>Supported</td>
</tr>
</tbody>
</table>
## Results of Evaluating Expressions in Behavioral Verilog

The following table shows evaluated expressions based on the most frequently used operators. The (===) and (!==) operators are special comparison operators. Use them in simulation to see if a variable is assigned a value of (x) or (z). They are treated as (==) or (!=) by synthesis.

| 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   | 1    | 1  | 1  | 1   |
| 0 x | x    | x     | 0    | 0     | 0   | x    | x  | x  | x   |
| 0 z | x    | 0     | x    | 1     | 0   | x    | x  | x  | x   |
| 1 0 | 0    | 0     | 1    | 1     | 0   | 1    | 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   | x    | 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   | x    | 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 group statements together. XST supports sequential blocks only. Within these blocks, the statements are executed in the order listed. Block statements are designated by **begin** and **end** keywords. XST does not support parallel blocks.

All procedural statements occur in blocks that are defined inside modules. The two kinds of procedural blocks are:

- **Initial block**
- **Always block**

Within each block, Verilog uses **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:

```verbatim
always
begin
statement
....
end
```

Each statement is a procedural assignment line terminated by a semicolon.
In Verilog, a design component is represented by a module. This section discusses Behavioral Verilog Modules, including Behavioral Verilog Module Declaration and Behavioral Verilog Module Instantiation.

**Behavioral Verilog Module Declaration**

A Behavioral Verilog module is declared as illustrated in the following coding examples.

**Behavioral Verilog Module Declaration Coding Example One**

```verilog
global example (A, B, O);
input A, B;
output O;
assign O = A & B;
endmodule
```

The module declaration consists of:

- The module name
- A list of I/O ports
- The module body where you define the intended functionality

The end of the module is signalled by a mandatory endmodule statement.

The I/O ports of the circuit are declared in the module declaration. Each port is characterized by:

- A name
- A mode:
  - `input`
  - `output`
  - `inout`
- Range information if the port is of array type

**Behavioral Verilog Module Declaration Coding Example Two**

```verilog
global example (A, B, O);
input A;
input B
output O
endmodule
```

```verilog
global example (A, B, O);
input A, B;
output O;
assign O = A & B;
endmodule
```
Behavioral Verilog Module Instantiation

A Behavioral Verilog module is instantiated in another module as follows.

Behavioral Verilog Module Instantiation Coding Example

```verilog
module top (A, B, C, O);
    input  A, B, C;
    output O;
    wire  tmp;

example inst_example (.A(A), .B(B), .O(tmp));
    assign O = tmp | C;
endmodule
```

A module instantiation statement:

- Defines an instance name.
- Contains a port association list that specifies how the instance is connected in the parent module.

Each element of the list ties a formal port of the module declaration, to an actual net of the parent module.

Continuous Assignments

Continuous assignments model combinatorial logic in a concise way. Both explicit and implicit continuous assignments are supported.

- Explicit continuous assignments start with an `assign` keyword after the net has been separately declared.
  ```verilog
  wire mysignal;
  ...
  assign mysignal = select ? b : a;
  ```

- Implicit continuous assignments combine declaration and assignment.
  ```verilog
  wire misignal = a | b;
  ```

XST ignores delays and strengths given to a continuous assignment. Continuous assignments are allowed on `wire` and `tri` data types only.
Procedural Assignments

This section discusses Behavioral Verilog Procedural Assignments, and includes:

- Combinatorial Always Blocks
- If-Else Statements
- Case Statements
- For and Repeat Loops
- While Loops
- Sequential Always Blocks
- Assign and Deassign Statements
- Assignment Extension Past 32 Bits

Behavioral Verilog procedural assignments are:

- Used to assign values to variables declared as \texttt{reg}.
- Introduced by \texttt{always} blocks, tasks, and functions.
- Usually used to model registers and Finite State Machine (FSM) components.

XST supports:

- Combinatorial functions
- Combinatorial and sequential tasks
- Combinatorial and sequential \texttt{always} blocks

Combinatorial Always Blocks

Combinatorial logic can be modeled efficiently using two forms of Verilog time control statements:

- Delay: \# (pound)
- Event control: @ (at)

The delay time control statement is relevant for simulation only and is ignored by synthesis tools.

Since the \# (pound) time control statement is ignored for synthesis, this discussion describes modeling combinatorial logic with the @ (at) time control statement.

A combinatorial \texttt{always} block has a sensitivity list appearing within parentheses after \texttt{always@}.

An \texttt{always} block is activated if an event (value change or edge) appears on one of the sensitivity list signals. This sensitivity list can contain any signal that appears in conditions (if or case, for example), and any signal appearing on the right-hand side of an assignment. By substituting an @ (at) without parentheses for a list of signals, the \texttt{always} block is activated for an event in any of the \texttt{always} block’s signals as described above.

In combinatorial processes, if a signal is not explicitly assigned in all branches of if or case statements, XST generates a latch to hold the last value. To the creation of latches, make sure that all assigned signals in a combinatorial process are always explicitly assigned in all paths of the process statements.
The following statements can be used in a process:

- Variable and signal assignments
- if-else statements
- case statements
- for-while loop statements
- Function and task calls

**If-Else Statements**

If-else statements use true-false conditions to execute statements.

- If the expression evaluates to true, the first statement is executed.
- If the expression evaluates to false, x or z, the else statement is executed.

A block of multiple statements can be executed using begin and end keywords. If-else statements can be nested.

**If-Else Statement Coding Example**

The following coding example shows how a multiplexer can be described using an if-else statement:

```verilog
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 Statements**

Case statements perform a comparison to an expression to evaluate one of a number of parallel branches. The case statement evaluates the branches in the order they are written. The first branch that evaluates to true is executed. If none of the branches matches, the default branch is executed.

- Do not use unsized integers in case statements. Always size integers to a specific number of bits. Otherwise, results can be unpredictable.
- Casez treats all z values in any bit position of the branch alternative as a don’t care.
- Casex treats all x and z values in any bit position of the branch alternative as a don’t care.
- The question mark (?) can be used as a don’t care in either the casez or casex case statements.
Case Statement Coding Example

The following coding example shows how a multiplexer can be described using a case statement.

```verilog
module mux4 (sel, a, b, c, d, outmux);
    input [1:0] sel;
    input [1:0] a, b, c, d;
    output [1:0] outmux;
    reg [1:0] outmux;

    always @(sel or a or b or c or d)
        begin
            case (sel)
                2'b00: outmux = a;
                2'b01: outmux = b;
                2'b10: outmux = c;
                default: outmux = d;
            endcase
        end

endmodule
```

The preceding case statement evaluates the values of input sel in priority order. To avoid priority processing, Xilinx® recommends that you use a parallel-case Verilog attribute to ensure parallel evaluation of the sel inputs.

Replace the case statement above with:

```verilog
(* parallel_case *) case(sel)
```

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 the following operators:
  - `<`
  - `<=`
  - `>`
  - `>=`
- Next step computation falling in one of the following specifications:
  - `var = var + step`
  - `var = var - step`
    where
    - `var` is the loop variable
    - `step` is a constant value

The repeat statement is supported for constant values only.
Disable statements are not supported.

```verilog
module countzeros (a, Count);
    input [7:0] a;
    output [2:0] Count;
    reg [2:0] Count;
    reg [2:0] Count_Aux;
    integer i;
    always @(a)
        begin
            Count_Aux = 3'b0;
            for (i = 0; i < 8; i = i+1)
                begin
                    if (!a[i])
                        Count_Aux = Count_Aux+1;
                    end
            Count = Count_Aux;
        end
endmodule
```

**While Loops**

When using `always` blocks, use the `while` statement to execute repetitive procedures.

- A `while` loop:
  - Executes other statements until its test expression becomes `false`.
  - Is not executed if the test expression is initially `false`.
- The test expression is any valid Verilog expression.
- To prevent endless loops, use the `-loop_iteration_limit` option.
- **While** loops can have `disable` statements. The `disable` statement is used inside a labeled block, since the syntax is:

```verilog
disable <blockname>
```

```verilog
parameter P = 4;
always @ (ID_complete)
begin : UNIDENTIFIED
    integer i;
    reg found;
    unidentified = 0;
    i = 0;
    found = 0;
    while (!found && (i < P))
        begin
            found = !ID_complete[i];
            unidentified[i] = !ID_complete[i];
            i = i + 1;
        end
end
```

**Sequential Always Blocks**

Describe a sequential circuit with an `always` block and a sensitivity list that contains the following edge-triggered (with `posedge` or `negedge`) events:

- A mandatory clock event
- Optional set/reset events (modeling asynchronous set/reset control logic)
If no optional asynchronous signal is described, the **always** block is structured as follows:

```verilog
always @(posedge CLK)
begin
    <synchronous_part>
end
```

If optional asynchronous control signals are modeled, the **always** block is instead structured as follows:

```verilog
always @(posedge CLK or posedge ACTRL1 or à )
begin
    if (ACTRL1)
        <$asynchronous_part>
    else
        <$synchronous_part>
end
```

**Sequential Always Block Coding Example One**

The following example describes an 8-bit register with a rising-edge clock. There are no other control signals.

```verilog
module seql (DI, CLK, DO);
    input [7:0] DI;
    input CLK;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK)
        DO <= DI ;
endmodule
```

**Sequential Always Block Coding Example Two**

The following example adds an active-High asynchronous reset.

```verilog
module EXAMPLE (DI, CLK, ARST, DO);
    input [7:0] DI;
    input CLK, ARST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge ARST)
        if (ARST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DI;
endmodule
```
Sequential Always Block Coding Example Three

The following example shows an active-High asynchronous reset, an active-Low asynchronous set.

```verilog
module EXAMPLE (DI, CLK, ARST, ASET, DO);
    input [7:0] DI;
    input CLK, ARST, ASET;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK or posedge ARST or negedge ASET)
        if (ARST == 1'b1)
            DO <= 8'b00000000;
        else if (ASET == 1'b1)
            DO <= 8'b11111111;
        else
            DO <= DI;
endmodule
```

Sequential Always Block Coding Example Four

The following example describes a register with no asynchronous set/reset, but with a synchronous reset.

```verilog
module EXAMPLE (DI, CLK, SRST, DO);
    input [7:0] DI;
    input CLK, SRST;
    output [7:0] DO;
    reg [7:0] DO;

    always @(posedge CLK)
        if (SRST == 1'b1)
            DO <= 8'b00000000;
        else
            DO <= DI;
endmodule
```

Assign and Deassign Statements

XST does not support assign and deassign statements.

Assignment Extension Past 32 Bits

If the expression on the left-hand side of an assignment is wider than the expression on the right-hand side, the left-hand side is padded to the left according to the following rules:

- If the right-hand expression is signed, the left-hand expression is padded with the sign bit.
- If the right-hand expression is unsigned, the left-hand expression is padded with 0s (zeros).
- For unsized x or z constants only, the following rule applies. If the value of the right-hand expression’s leftmost bit is z (high impedance) or x (unknown), regardless of whether the right-hand expression is signed or unsigned, the left-hand expression is padded with that value (z or x, respectively).

These rules follow the Verilog-2001 standard. They are not backwardly compatible with Verilog-1995.
Tasks and Functions

This section discusses Behavioral Verilog Tasks and Functions, and includes:

- Tasks and Functions Coding Examples
- Recursive Tasks and Functions
- Constant Functions

When the same code is used multiple times across a design, using tasks and functions reduces the amount of code and facilitates maintenance.

Tasks and functions must be declared and used in a module. The heading contains the following parameters:

- Input parameters (only) for functions
- Input/output/inout parameters for tasks

The return value of a function can be declared either signed or unsigned. The contents are similar to the contents of the combinatorial always block.

Tasks and Functions Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Tasks and Functions Coding Example One

```verilog
//
// An example of a function in Verilog
// File: Verilog_Language_Support/functions_tasks/functions_1.v
//
module functions_1 (A, B, CIN, S, COUT);
    input [3:0] A, B;
    input CIN;
    output [3:0] S;
    output COUT;
    wire [1:0] S0, S1, S2, S3;

    function signed [1:0] ADD;
        input A, B, CIN;
        reg S, COUT;
        begin
            S = A ^ B ^ CIN;
            COUT = (A&B) | (A&CIN) | (B&CIN);
            ADD = {COUT, S};
        end
    endfunction

    assign S0 = ADD (A[0], B[0], CIN),
            S1 = ADD (A[1], B[1], S0[1]),
            S2 = ADD (A[2], B[2], S1[1]),
            S3 = ADD (A[3], B[3], S2[1]),
            S = {S3[0], S2[0], S1[0], S0[0]},
            COUT = S3[1];

endmodule
```
Tasks and Functions Coding Example Two

In the following coding example, the same functionality is described with a task.

```verilog
// Verilog tasks
// File: Verilog_Language_Support/functions_tasks/tasks_1.v
//
module tasks_1 (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
task
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
```

Recursive Tasks and Functions

Verilog-2001 supports recursive tasks and functions. You can use recursion only with the `automatic` keyword. To prevent endless recursive calls, the number of recursions is limited by default to 64. Use `-recursion_iteration_limit` to control the number of allowed recursive calls.

Recursive Tasks and Functions Coding Example

```verilog
function automatic [31:0] fac;
    input [15:0] n;
    if (n == 1)
        fac = 1;
    else
        fac = n * fac(n-1); //recursive function call
endfunction
```

Constant Functions

XST supports function calls to calculate constant values.
Constant Functions Coding Example

//
// A function that computes and returns a constant value
//
// File: Verilog_Language_Support/functions_tasks/functions_constant.v
//
module functions_constant (clk, we, a, di, do);

parameter ADDRWIDTH = 8;
parameter DATAWIDTH = 4;
input clk;
input we;
input [ADDRWIDTH-1:0] a;
input [DATAWIDTH-1:0] di;
output [DATAWIDTH-1:0] do;

function integer getSize;
    input addrwidth;
    begin
        getSize = 2**addrwidth;
    end
endfunction

reg [DATAWIDTH-1:0] ram [getSize(ADDRWIDTH)-1:0];
always @(posedge clk)
begin
    if (we)
        ram[a] <= di;
end
assign do = ram[a];
endmodule

Blocking Versus Non-Blocking Procedural Assignments

The pound (#) and at sign (@) time control statements delay execution of the statement following them until the specified event is evaluated as true. Blocking and non-blocking procedural assignments have time control built into their respective assignment statement. The pound (#) delay is ignored for synthesis.

Blocking Procedural Assignment Syntax Coding Example One
reg a;
a = #10 (b | c);

Blocking Procedural Assignment Syntax Coding Example Two (Alternate)
if (in1) out = 1’b0;
else out = in2;

As the name implies, these types of assignments block the current process from continuing to execute additional statements at the same time. These should mainly be used in simulation.

Non-blocking assignments, on the other hand, evaluate the expression when the statement executes, but allow other statements in the same process to execute as well at the same time. The variable change occurs only after the specified delay.

Non-Blocking Procedural Assignment Syntax Coding Example One
variable <= @(posedge_or_negedge_bit) expression;
Non-Blocking Procedural Assignment Coding Example Two

The following example shows how to use a non-blocking procedural assignment.

```verilog
if (in1) out <= 1'b1;
else out <= in2;
```

Constants

Constants in Verilog are assumed to be decimal integers. To specify constants explicitly in binary, octal, decimal, or hexadecimal, prefix them with the appropriate syntax. For example, the following constant expressions represent the same value:

- `4'b1010`
- `4'o12`
- `4'd10`
- `4'ha`

Macros

Verilog defines macros as follows:

```verilog
'define TESTEQ1 4'b1101
```

The defined macro is referenced later in the design code as follows:

```verilog
if (request == 'TESTEQ1)
```

Macros Coding Example One

```verilog
'define myzero 0
assign mysig = 'myzero;
```

The Verilog `ifdef` and `endif` constructs determine whether or not a macro is defined. These constructs are used to define conditional compilation. If the macro called out by the `ifdef` command has been defined, that code is compiled. If not, the code following the `else` command is compiled. The `else` is not required, but `endif` must complete the conditional statement.

Macros Coding Example Two

```verilog
'ifdef MYVAR
module if_MYVAR_isDeclared;
...
endmodule
'else
module if_MYVAR_isNotDeclared;
...
endmodule
'endif
```

Use Verilog Macros (`define) to define (or redefine) Verilog macros. This allows you to modify the design configuration without modifying the source code. This feature is useful for such processes as IP core generation and flow testing.

Include Files

Verilog allows you to separate Hardware Description Language (HDL) source code into more than one file. To reference the code contained in another file, use the following syntax in the current file:
'include "path/file-to-be-included"

The path can be relative or absolute.

Multiple 'include statements are allowed in the same Verilog file. This feature makes your code more manageable in a team design environment where different files describe different modules of the design.

To enable the file in your 'include statement to be recognized, identify the directory in which it resides, either to ISE® Design Suite or to XST.

- Since ISE Design Suite searches the project directory by default, adding the file to your project directory identifies the file to ISE Design Suite.
- To direct ISE Design Suite to a different directory, include a path (relative or absolute) in the 'include statement in the HDL source code.
- To point XST directly to your include file directory, use Verilog Include Directories (-vigincdir)
- If the include file is required for ISE Design Suite to construct the design hierarchy, this file must either reside in the project directory, or be referenced by a relative or absolute path. The file need not be added to the project.

The XST design project file provides another way to make a Verilog file contents visible to the rest of your project. Xilinx® recommends the XST design project file method. If you use the file inclusion method, be aware of a potential conflict. Do not include a Verilog file with the mechanism described here, and, at the same time, list that file in your XST design project file. Doing so results in an error as follows:

ERROR: HDLCompiler:687 - "include_sub.v" line 1: Illegal redeclaration of module <sub>.

You may encounter this error if you add Verilog files with such inclusions to an ISE Design Suite project. Because ISE Design Suite adds them to the XST design project file, a multiple-definition conflict can result.

Comments

XST supports both forms of Behavioral Verilog comments:

- One-line comments, starting with a double forward slash (//)
  
  // This is a one-line comment

- Multiple-line block comments, starting with /* and ending with */

  /* This is a
     Multiple-line
     comment
  */

Behavioral Verilog comments are similar to those used in such languages as C++.

Generate Statements

This section discusses Behavioral Verilog Generate Statements, including Generate Loop Statements, Generate Conditional Statements, and Generate Case Statements.

Verilog generate statements allow you to create parameterized and scalable code. The contents of a generate statement is conditionally instantiated into your design. Generate statements are resolved during Verilog elaboration.
Generate statements are a powerful way to create repetitive or scalable structures, or to create functionality conditional to a particular criteria being met. Structures likely to be created using a generate statement are:

- Primitive or module instances
- Initial or always procedural blocks
- Continuous assignments
- Net and variable declarations
- Parameter redefinitions
- Task or function definitions

Describe generate statements within a module scope. They start with a generate keyword, and end with an endgenerate keyword.

XST supports all three forms of Verilog generate statements:

- generate-loop (generate-for)
- generate-conditional (generate-if-else)
- generate-case (generate-case)

### Generate Loop Statements

Use a generate-for loop to create one or more instances that can be placed inside a module. Use the generate-for loop the same way you use a normal Verilog for loop, with the following limitations:

- The index for a generate-for loop has a genvar variable.
- The assignments in the for loop control refers to the genvar variable.
- The contents of the for loop are enclosed by begin and end statements. The begin statement is named with a unique qualifier.

#### Generate Loop Statement 8-Bit Adder Coding Example

```verilog
generate genvar i;
  for (i=0; i<=7; i=i+1)
  begin : for_name
    adder add (a[8*i+7 : 8*i], b[8*i+7 : 8*i], ci[i], sum_for[8*i+7 : 8*i], c0_or[i+1]);
  end
endgenerate
```

### Generate Conditional Statements

Use a generate-if-else statement to conditionally control which objects are generated.

- The contents of each branch of the if-else statement are enclosed by begin and end statements.
- The begin statement is named with a unique qualifier.
Generate Conditional Statement Coding Example

The following example instantiates two different implementations of a multiplier based on the width of data words.

```verilog
generate
  if (IF_WIDTH < 10)
    begin : if_name
      multiplier_impl1 # (IF_WIDTH) u1 (a, b, sum_if);
    end
  else
    begin : else_name
      multiplier_impl2 # (IF_WIDTH) u2 (a, b, sum_if);
    end
endgenerate
```

Behavioral Verilog Generate Case Statements

Use a `generate-case` statement to conditionally control which objects are generated under different conditions.

- Each branch in a `generate-case` is enclosed by `begin` and `end` statements.
- The `begin` statement is named with a unique qualifier.

Behavioral Verilog Generate Case Statements Coding Example

The following coding example instantiates more than two different implementations of an adder based on the width of data words.

```verilog
generate
  case (WIDTH)
    1:
      begin : case1_name
        adder #(WIDTH*8) x1 (a, b, ci, sum_case, c0_case);
      end
    2:
      begin : case2_name
        adder #(WIDTH*4) x2 (a, b, ci, sum_case, c0_case);
      end
    default:
      begin : d_case_name
        adder x3 (a, b, ci, sum_case, c0_case);
      end
  endcase
endgenerate
```
XST Mixed Language Support

XST supports mixed VHDL and Verilog projects.
- Mixing VHDL and Verilog is restricted to design unit (cell) instantiation only.
- A Verilog module can be instantiated from VHDL code.
- A VHDL entity can be instantiated from Verilog code.
- No other mixing between VHDL and Verilog is supported. For example, you cannot embed Verilog source code directly in VHDL code.
- In a VHDL design, a restricted subset of VHDL types, generics, and ports is allowed on the boundary to a Verilog module.
- In a Verilog design, a restricted subset of Verilog types, parameters, and ports is allowed on the boundary to a VHDL entity or configuration.
- XST binds VHDL design units to a Verilog module during HDL Elaboration.
- Component instantiation based on default binding is used for binding Verilog modules to a VHDL design unit.
- Configuration specification, direct instantiation and component configurations are not supported for a Verilog module instantiation in VHDL.
- VHDL and Verilog files making up your project are specified in a unique XST HDL project file.

For more information, see:
Chapter 2, Creating and Synthesizing an XST Project
- VHDL and Verilog libraries are logically unified.
- The default work directory for compilation (xsthdpdir) is available for both VHDL and Verilog.
- The xhdp.ini mechanism for mapping a logical library name to a physical directory name on the host file system, is available for both VHDL and Verilog.
- Mixed language projects accept a search order used for searching unified logical libraries in design units (cells). During Elaboration, XST follows this search order for picking and binding a VHDL entity or a Verilog module to the mixed language project.
VHDL and Verilog Boundary Rules

The boundary between VHDL and Verilog is enforced at the design unit level. A VHDL entity or architecture can instantiate a Verilog module. A Verilog module can instantiate a VHDL entity.

Instantiating a VHDL Design Unit in a Verilog Design

To instantiate a VHDL entity:

1. Declare a module name with the same as name as the VHDL entity that you want to instantiate (optionally followed by an architecture name).
2. Perform a normal Verilog instantiation.

The only VHDL construct that can be instantiated in a Verilog design is a VHDL entity. No other VHDL constructs are visible to Verilog code. When you do so, XST uses the entity-architecture pair as the Verilog-VHDL boundary.

XST performs the binding during elaboration. During binding, XST searches for a Verilog module name using the name of the instantiated module in the user-specified list of unified logical libraries in the user-specified order. XST ignores any architecture name specified in the module instantiation.

For more information, see:

Library Search Order (LSO) Files

If found, XST binds the name. If XST cannot find a Verilog module, it treats the name of the instantiated module as a VHDL entity, and searches for it using a case sensitive search for a VHDL entity. XST searches for the VHDL entity in the user-specified list of unified logical libraries in the user-specified order, assuming that a VHDL design unit was stored with extended identifier.

For more information, see:

Library Search Order (LSO) Files

If found, XST binds the name. XST selects the first VHDL entity matching the name, and binds it.

XST has the following limitations when instantiating a VHDL design unit from a Verilog module:

- Use explicit port association. Specify formal and effective port names in the port map.
- All parameters are passed at instantiation, even if they are unchanged.
- The parameter override is named and not ordered. The parameter override occurs through instantiation, and not through defparams.

Accepted Coding Example

XST DOES accept the following coding example.

```vhdl
ff #(.init(2'b01)) ul (.sel(sel), .din(din), .dout(dout));
```

NOT Accepted Coding Example

**Caution!** XST DOES NOT accept the following coding example.

```vhdl
ff ul (.sel(sel), .din(din), .dout(dout));
defparam ul.init = 2'b01;
```
Instantiating a Verilog Module in VHDL

To instantiate a Verilog module in a VHDL design:

1. Declare a VHDL component with the same name (observing case sensitivity) as the Verilog module to be instantiated. If the Verilog module name is not all lowercase, use the case property to preserve the case of the Verilog module.
   - ISE® Design Suite
     Select Process > Properties > Synthesis Options > Case > Maintain
   - Command Line
     Set -case to maintain

2. Instantiate the Verilog component as if you were instantiating a VHDL component.

Using a VHDL configuration declaration, you could attempt to bind this component to a particular design unit from a particular library. Such binding is not supported. Only default Verilog module binding is supported.

The only Verilog construct that can be instantiated in a VHDL design is a Verilog module. No other Verilog constructs are visible to VHDL code.

- During elaboration, all components subject to default binding are regarded as design units with the same name as the corresponding component name.
- During binding, XST treats a component name as a VHDL design unit name and searches for it in the logical library work.
  - If XST finds a VHDL design unit, XST binds it.
  - If XST cannot find a VHDL design unit, it treats the component name as a Verilog module name, and searches for it using a case sensitive search.

XST searches for the Verilog module in the user-specified list of unified logical libraries in the user-specified search order.

For more information, see:

Library Search Order (LSO) Files

XST selects the first Verilog module matching the name, and binds it.

Since libraries are unified, a Verilog cell having the same name as that of a VHDL design unit cannot co-exist in the same logical library. A newly compiled cell or unit overrides a previously compiled cell or unit.

Generics Support

XST supports the following VHDL generic types, and their Verilog equivalents for mixed language designs:
- integer
- real
- string
- boolean
Port Mapping

This section discusses Port Mapping for VHDL Instantiated in Verilog and Verilog Instantiated in VHDL.

VHDL Instantiated in Verilog

When a VHDL entity is instantiated in a Verilog module, formal ports may have the following characteristics:

- Allowed directions
  - in
  - out
  - inout
- Unsupported directions
  - buffer
  - linkage
- Allowed data types
  - bit
  - bit_vector
  - std_logic
  - std_ulogic
  - std_logic_vector
  - std_ulogic_vector

Verilog Instantiated in VHDL

When a Verilog module is instantiated in a VHDL entity or architecture, formal ports may have the following characteristics:

- Allowed directions:
  - input
  - output
  - inout
- XST does not support connection to bi-directional pass options in Verilog.
- XST does not support unnamed Verilog ports for mixed language boundaries.
- Allowed data types:
  - wire
  - reg

Use an equivalent component declaration to connect to a case sensitive port in a Verilog module. XST assumes Verilog ports are in all lowercase.
Chapter 6: XST Mixed Language Support

Library Search Order (LSO) Files

The Library Search Order (LSO) file specifies the search order that XST uses to link the libraries used in VHDL and Verilog mixed language designs. XST searches the files specified in the project file in the order in which they appear in that file.

XST uses the default search order when:

- The `DEFAULT_SEARCH_ORDER` keyword is used in the LSO file, or
- The LSO file is not specified

Specifying LSO Files in ISE Design Suite

In ISE® Design Suite, the default name for the Library Search Order (LSO) file is `project_name.lso`. If a `project_name.lso` file does not exist, ISE Design Suite creates one. If ISE Design Suite detects an existing `project_name.lso` file, this file is preserved and used as is. The name of the project is the name of the top-level block. When creating a default LSO file, ISE Design Suite places the `DEFAULT_SEARCH_ORDER` keyword in the first line of the file.

Specifying LSO Files in Command Line Mode

The Library Search Order (LSO) `-ls0` option specifies the Library Search Order (LSO) file when using XST from the command line. If the `-ls0` option is omitted, XST automatically uses the default library search order without using an LSO file.

LSO Rules

When processing a mixed language project, XST obeys the following search order rules, depending on the contents of the Library Search Order (LSO) file:

- Empty Library Search Order (LSO) File
- `DEFAULT_SEARCH_ORDER` Keyword Only
- `DEFAULT_SEARCH_ORDER` Keyword and List of Libraries
- List of Libraries Only
- `DEFAULT_SEARCH_ORDER` Keyword and Non-Existential Library Name

Empty LSO Files

When the Library Search Order (LSO) file is empty, XST:

- Warns that the LSO file is empty.
- Searches the files specified in the project file using the default library search order.
- Updates the LSO file by adding the list of libraries in the order that they appear in the project file.

`DEFAULT_SEARCH_ORDER` Keyword Only

When the Library Search Order (LSO) file contains only the `DEFAULT_SEARCH_ORDER` keyword, XST:

- Searches the specified library files in the order in which they appear in the project file
- Updates the LSO file by:
  - Removing the `DEFAULT_SEARCH_ORDER` keyword
  - Adding the list of libraries to the LSO file in the order in which they appear in the project file
For a project file, `my_proj.prj`, with the following contents:

```
vhdl  vhlib1  f1.vhd
verilog  rtfllib  f1.v
vhdl  vhlib2  f3.vhd
```

and an LSO file, `my_proj.lso`, created by ISE® Design Suite, with the following contents:

```
DEFAULT_SEARCH_ORDER
```

XST uses the following search order. The same contents appear in the updated `my_proj.lso` file after processing:

```
vhlib1
rtfllib
vhlib2
```

**DEFAULT_SEARCH_ORDER Keyword and List of Libraries**

When the Library Search Order (LSO) file contains the `DEFAULT_SEARCH_ORDER` keyword, and a list of libraries, XST:

- Searches the specified library files in the order in which they appear in the project file
- Ignores the list of library files in the LSO file
- Leaves the LSO file unchanged

For a project file `my_proj.prj` with the following contents:

```
vhdl  vhlib1  f1.vhd
verilog  rtfllib  f1.v
vhdl  vhlib2  f3.vhd
```

and an LSO file `my_proj.lso` with the following contents:

```
rtfllib
vhlib2
vhlib1
```

XST uses the following search order:

```
vhlib1
rtfllib
vhlib2
```

After processing, the contents of `my_proj.lso` remains unchanged:

```
rtfllib
vhlib2
vhlib1
```

**List of Libraries Only**

When the Library Search Order (LSO) file contains a list of the libraries without the `DEFAULT_SEARCH_ORDER` keyword, XST:

- Searches the library files in the order in which they appear in the LSO file
- Leaves the LSO file unchanged
For a project file `my_proj.prj` with the following contents:

```vhdl
vhdl vhlib1 f1.vhd
verilog rtfllib f1.v
vhdl vhlib2 f3.vhd
```

and an LSO file `my_proj.lso` with the following contents:

```text
rtflllib
vhlib2
vhlib1
```

XST uses the following search order:

```text
rtflllib
vhlib2
vhlib1
```

After processing, the contents of `my_proj.lso` is:

```text
rtflllib
vhlib2
vhlib1
```

**DEFAULT_SEARCH_ORDER Keyword and Non-Existent Library Name**

When the Library Search Order (LSO) file contains a library name that does not exist in the project or INI file, and the LSO file does not contain the `DEFAULT_SEARCH_ORDER` keyword, XST ignores the library.

For a project file `my_proj.prj` with the following contents:

```vhdl
vhdl vhlib1 f1.vhd
verilog rtfllib f1.v
vhdl vhlib2 f3.vhd
```

and an LSO file `my_proj.lso` created with the following contents:

```text
personal_lib
rtflllib
vhlib2
vhlib1
```

XST uses the following search order:

```text
rtflllib
vhlib2
vhlib1
```

After processing, the contents of `my_proj.lso` is:

```text
rtflllib
vhlib2
vhlib1
```
Chapter 7

XST HDL Coding Techniques

Hardware Description Language (HDL) coding techniques allow you to:

- Describe the most common functionalities found in digital logic circuits.
- Take advantage of the architectural features of Virtex®-6 and Spartan®-6 devices.

Most sections of this chapter include:

- A general description of the functionality
- Guidelines to model the functionality in the HDL source code
- Information on how XST implements the functionality on Virtex-6 and Spartan-6 devices. For more information, see Chapter 8, XST FPGA Optimization.
- A list of constraints to control how XST processes such functionality
- Reporting examples
- VHDL and Verilog coding examples

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

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Relative Advantages and Disadvantages of VHDL and Verilog

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enforces stricter rules, in particular strongly typed, less permissive and error-prone</td>
<td>Extension to System Verilog (currently not supported by XST)</td>
<td></td>
</tr>
<tr>
<td>Initialization of RAMs in the HDL source code is easier (Verilog initial blocks are less convenient)</td>
<td>C-like syntax</td>
<td></td>
</tr>
<tr>
<td>Package support</td>
<td>Results in more compact code</td>
<td></td>
</tr>
<tr>
<td>Custom types</td>
<td>Block commenting</td>
<td></td>
</tr>
<tr>
<td>Enumerated types</td>
<td>No heavy component instantiation as in VHDL</td>
<td></td>
</tr>
<tr>
<td>No reg versus wire confusion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Macro Inference Flow Overview

Macro inferences can occur at three stages of the XST synthesis flow:

- Basic macros are inferred during Hardware Description Language (HDL) Synthesis.
- Complex macros are inferred during Advanced HDL Synthesis.
- Some macros can be inferred at an even later stage, during Low-Level Optimizations, when timing information is available to make more fully-informed decisions.

Macros inferred during Advanced HDL Synthesis are usually the result of an aggregation of several basic macros previously inferred during HDL Synthesis. In most cases, the XST inference engine can perform this grouping regardless of hierarchical boundaries, unless Keep Hierarchy (KEEP_HIERARCHY) has been set to yes in order to prevent it.

For example, a block RAM can be inferred by combining RAM core functionality described in one user-defined hierarchical block, with a register described in a different user-defined hierarchy. This allows you to structure the HDL project in a modular way, ensuring that XST can recognize relationships among design elements described in different VHDL entities and Verilog modules.

Do not describe every basic bit-level element in its own separate hierarchy. Doing so may prevent you from leveraging the RTL inference capabilities of the synthesis tool. See the design projects in Extended DSP Inferencing for ideas on how to structure the HDL source code.
Flip-Flops and Registers

This section discusses XST HDL Coding Techniques for Flip-Flops and Registers, and includes:

- About Flip-Flops and Registers
- Flip-Flops and Registers Initialization
- Flip-Flops and Registers Control Signals
- Flip-Flops and Registers Related Constraints
- Flip-Flops and Registers Reporting
- Flip-Flops and Registers Coding Examples

About Flip-Flops and Registers

XST recognizes flip-flops and registers with the following control signals:

- Rising or falling-edge clocks
- Asynchronous Set/Reset
- Synchronous Set/Reset
- Clock Enable

Flip-flops and registers are described with:

- sequential process (VHDL)
- always block (Verilog)

For more information on describing sequential logic in HDL, see:

- Chapter 3, XST VHDL Language Support
- Chapter 4, XST Verilog Support

The process or always block sensitivity list should list the clock signal and all asynchronous control signals.

Flip-Flops and Registers Initialization

To initialize the contents of a register at circuit power-up, specify a default value for the signal modeling it.

To do so in VHDL, declare a signal such as:

```vhdl
signal example1 : std_logic := '1';
signal example2 : std_logic_vector(3 downto 0) := (others => '0');
signal example3 : std_logic_vector(3 downto 0) := "1101";
```

In Verilog, initial contents is described as follows:

```verilog
reg example1 = 'b1;
reg [15:0] example2 = 16'b1111111011011100;
reg [15:0] example3 = 16'hFEDC;
```

The synthesized flip-flops are initialized to the specified value on the target device upon activation of the circuit global reset at circuit power-up.
Chapter 7: XST HDL Coding Techniques

Flip-Flops and Registers Control Signals

Control signals include:

- Clocks
- Asynchronous and synchronous set and reset signals
- Clock enable

Observe the coding guidelines below to:

- Minimize slice logic utilization
- Maximize circuit performance
- Utilize device resources such as block RAMs and DSP blocks

The coding guidelines are as follows:

- Do not set or reset registers asynchronously. Use synchronous initialization. Although possible on Xilinx® devices, Xilinx does not recommend this practice for the following reasons:
  - Control set remapping is made impossible
  - Sequential functionality in device resources such as block RAMs and DSP blocks can only be set or reset synchronously. You will either be unable to leverage those resources, or they will be configured sub-optimally.

- If your coding guidelines require registers to be set or reset asynchronously, try running XST with Asynchronous to Synchronous (ASYNC_TO_SYNC). This allows you to assess the benefits of a synchronous set/reset approach.

- Do not describe flip-flops with both a set and a reset. Starting with the Virtex®-6 and Spartan®-6 devices, none of the available flip-flop primitives features both a set and a reset, whether synchronous or asynchronous. If not rejected by the software, such combinations can lead to implementations that can adversely affect area and performance.

- XST rejects flip-flops described with both an asynchronous reset and an asynchronous set, rather than retargeting them to a costly equivalent model.

- Whenever possible, avoid operational set/reset logic altogether. There may be other, less expensive, ways to achieve the desired effect, such as taking advantage of the circuit global reset by defining an initial contents.

- The clock enable, set and reset control inputs of Xilinx flip-flop primitives are always active-High. If described to be active-Low, such functionality inevitably leads to inverter logic that penalizes the performance of the circuit.

Flip-Flops and Registers Related Constraints

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

For more ways to control implementation of flip-flops and registers, see:

Mapping Logic to LUTs
Flip-Flops and Registers Reporting

Registers are inferred and reported during HDL Synthesis. After Advanced HDL Synthesis, they are expanded to individual flip-flops, as reflected by subsequent reporting.

* HDL Synthesis *

Synthesizing Unit registers_5>. Found 4-bit register for signal Q>. Summary: inferred 4 D-type flip-flop(s). Unit registers_5> synthesized.

* Advanced HDL Synthesis *

(...)

Advanced HDL Synthesis Report

The number of registers inferred during HDL Synthesis may not directly translate into a precisely matching number of flip-flop primitives in the Design Summary section. The latter is dependent on the outcome of a number of processing steps during Advanced HDL Synthesis and Low Level Synthesis. They include:

- Absorption of registers into DSP blocks or block RAMs
- Register duplication
- Removal of constant or equivalent flip-flops
- Register balancing

Flip-Flops and Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Flip-Flops and Registers VHDL Coding Example

```vhdl
-- -- Flip-Flop with
--   Rising-edge Clock
--   Active-high Synchronous Reset
--   Active-high Clock Enable
--   Initial Value

-- File: HDL_Coding_Techniques/registers/registers_6.vhd

library ieee;
use ieee.std_logic_1164.all;

entity registers_6 is
  port(
    clk : in std_logic;
    rst : in std_logic;
    clken : in std_logic;
    D : in std_logic;
    Q : out std_logic);
end registers_6;

architecture behavioral of registers_6 is
  signal S : std_logic := '0';
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        S <= '0';
      elsif clken = '1' then
        S <= D;
      end if;
    end if;
  end process;
  Q <= S;
end behavioral;
```

Flip-Flops and Registers Verilog Coding Example

```verilog
// // 4-bit Register with
//   Rising-edge Clock
//   Active-high Synchronous Reset
//   Active-high Clock Enable
// // File: HDL_Coding_Techniques/registers/registers_6.v
//
module v_registers_6 (clk, rst, clken, D, Q);
  input clk, rst, clken;
  input [3:0] D;
  output reg [3:0] Q;
  always @(posedge clk)
  begin
    if (rst)
      Q <= 4'ob0011;
    else if (clken)
      Q <= D;
  end
endmodule
```
Chapter 7: XST HDL Coding Techniques

Latches

This section discusses XST HDL Coding Techniques for Latches, and includes:

- About Latches
- Describing Latches
- Latches Related Constraints
- Latches Reporting
- Latches Coding Examples

About Latches

Latches inferred by XST have:

- A data input
- An enable input
- A data output
- An optional Set/Reset

Describing Latches

Latches are usually created from an HDL description when a signal modeling the latch output is not assigned any new contents in a branch of an if-else construct.

A latch can be described as follows:

- Concurrent signal assignment (VHDL)
  \[ Q <= D \text{ when } G = '1'; \]
- Process (VHDL)
  ```
  process (G, D)
  begin
    if G = '1' then
      Q <= D;
  end process;
  ```
- Always block (Verilog)
  ```
  always @(G or D)
  begin
    if (G)
      Q <= D;
  end
  ```

In VHDL, XST can infer latches from descriptions based on a `wait` statement.

Latches Related Constraints

Pack I/O Registers Into IOBs (IOB)
Latches Reporting

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

* HDL Synthesis *

---------------------------------------------------------------

Sythesizing Unit example>

WARNING:Xst:737 - Found 1-bit latch for signal <Q>.

Latches may be generated from incomplete case or if statements.
We do not recommend the use of latches in FPGA/CPLD designs,
as they may lead to timing problems.

Summary:
inferred 1 Latch(s).
Unit example> synthesized.

---------------------------------------------------------------

HDL Synthesis Report

Macro Statistics
# Latches : 1
1-bit latch : 1

---------------------------------------------------------------

Unlike for other macros, XST issues a warning in this instance. Inferred latches are often the result of HDL coding mistakes, such as incomplete case or if constructs. This warning alerts you to potential problems, allowing you to verify that the inferred latch functionality was intended.

Latches Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Latch with Positive Gate and Asynchronous Reset VHDL Coding Example

--
-- Latch with Positive Gate and Asynchronous Reset
--
--- File: HDL_Coding_Techniques/latches/latches_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;

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

architecture archi of latches_2 is
begin
  process (CLR, D, G)
  begin
    if (CLR='1') then
      Q <= '0';
    elsif (G='1') then
      Q <= D;
    end if;
  end process;
end archi;
Latch with Positive Gate Verilog Coding Example

//
// Latch with Positive Gate
//
// File: HDL_Coding_Techniques/latches/latches_1.v
//
module v_latches_1 (G, D, Q);
    input G, D;
    output Q;
    reg Q;
    always @(G or D)
        begin
            if (G)
                Q = D;
        end
endmodule
Tristates

This section discusses XST HDL Coding Techniques for Tristates, and includes:

- About Tristates
- Tristates Implementation
- Tristates Related Constraints
- Tristates Reporting
- Tristates Coding Examples

About Tristates

Tristate buffers are usually modeled by a signal and an if-else construct whether driving:

- An internal bus
- An external bus on the board on which the Xilinx® device resides

The signal is assigned a high impedance value in one branch of the if-else. This description can be achieved with different coding styles.

- Concurrent signal assignment (VHDL)
  
  ```vhdl
  <= I when T = '0' else (others => 'Z');
  ```

- Concurrent signal assignment (Verilog)

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

- Combinatorial process (VHDL)

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

- Always block (Verilog)

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

Tristates Implementation

Inferred tristate buffers are implemented with different device primitives when driving an internal bus (BUFT) or an external pin of the circuit (OBUFT).

Tristates Related Constraints

Convert Tristates to Logic (TRISTATE2LOGIC)
Tristates Reporting

Tristate buffers are inferred and reported during HDL Synthesis.

* HDL Synthesis *

Synthesizing Unit example>

  Found 1-bit tristate buffer for signal S> created at line 22

  Summary:
  inferred 8 Tristate(s).
  Unit example> synthesized.

Macro Statistics
  # Tristates : 8
  1-bit tristate buffer : 8

Tristates Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Tristate Description Using Combinatorial Process VHDL Coding Example

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

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

architecture archi of three_st_1 is
begin
  process (I, T)
  begin
    if (T='0') then
      O <= I;
    else
      O <= 'Z';
    end if;
  end process;
end archi;
```
Tristate Description Using Concurrent Assignment VHDL Coding Example

--
-- Tristate Description Using Concurrent Assignment
--
-- File: HDL_Coding_Techniques/tristates/tristates_2.vhd
library ieee;
use ieee.std_logic_1164.all;

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

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

Tristate Description Using Combinatorial Process VHDL Coding Example

--
-- Tristate Description Using Combinatorial Process
-- Implemented with an OBUF (internal buffer)
--
-- File: HDL_Coding_Techniques/tristates/tristates_3.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity example is
  generic (WIDTH : integer := 8);
  port(
      T : in std_logic;
      I : in std_logic_vector(WIDTH-1 downto 0);
      O : out std_logic_vector(WIDTH-1 downto 0));
end example;

architecture archi of example is
  signal S : std_logic_vector(WIDTH-1 downto 0);
begin
  process (I, T)
  begin
    if (T = '1') then
      S <= I;
    else
      S <= (others => 'Z');
    end if;
  end process;
  O <= not(S);
end archi;
Chapter 7: XST HDL Coding Techniques

Tristate Description Using Combinatorial Always Block Verilog Coding Example

```verilog
// Tristate Description Using Combinatorial Always Block
// File: HDL_Coding_Techniques/tristates/tristates_1.v
//
module v_three_st_1 (T, I, O);
  input T, I;
  output O;
  reg   O;

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

Tristate Description Using Concurrent Assignment Verilog Coding Example

```verilog
// Tristate Description Using Concurrent Assignment
// File: HDL_Coding_Techniques/tristates/tristates_2.v
//
module v_three_st_2 (T, I, O);
  input T, I;
  output O;

  assign O = (~T) ? I: 'Z;
endmodule
```
Counters and Accumulators

This section discusses XST HDL Coding Techniques for Counters and Accumulators, and includes:

- About Counters and Accumulators
- Counters and Accumulators Implementation
- Counters and Accumulators Related Constraints
- Counters and Accumulators Reporting
- Counters and Accumulators Coding Examples

About Counters and Accumulators

XST provides inference capability for counters and accumulators. Besides the core functionality, you can describe such optional features as:

- Asynchronous set, reset or load
- Synchronous set, reset or load
- Clock enable
- Up, down, or up/down direction

An accumulator differs from a counter (also known as incrementer or decrementer) in the nature of the operands of the add or subtract operation, or both.

In a counter description, the destination and first operand is a signal or variable, and the other operand is a constant equal to 1.

\[ A \leftarrow A + 1; \]

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

- A signal or variable
  \[ A \leftarrow A + B; \]
- A constant not equal to 1
  \[ A \leftarrow A + \text{Constant}; \]

Direction of an inferred counter or accumulator can be up, down, or updown. For an updown accumulator, the accumulated data can differ between the up and down mode.

\[
\begin{align*}
\text{if updown} &= '1' \text{ then} \\
& a \leftarrow a + b; \\
\text{else} \\
& a \leftarrow a - c;
\end{align*}
\]

XST supports description of both unsigned and signed counters and accumulators.

Whether described with a signal of type integer or array of bits, XST determines the minimal number of bits needed to implement an inferred counter or accumulator. Unless explicitly otherwise specified in the HDL description, a counter can potentially take all values allowed by this number during circuit. You can count up to a specific value using a modulo operator as follows.

**VHDL Syntax Example**

\[ \text{cnt} \leftarrow (\text{cnt} + 1) \mod \text{MAX}; \]
Verilog Syntax Example

cnt <= (cnt + 1) × MAX;

Counters and Accumulators Implementation

Counters and accumulators can be implemented on:

- Slice logic
- DSP block resources

A DSP block can absorb up to two levels of registers, provided the counter or accumulator fits in a single DSP block. If a counter or accumulator macro does not fit in a single DSP block, XST implements the entire macro using slice logic.

Macro implementation on DSP block resources is controlled by Use DSP Block (USE_DSP48) with a default value of auto.

In auto mode, XST implements counters and accumulators considering such factors as:

- DSP block resources available on the device
- Contextual information such as the source of the data being accumulated
- Whether implementation in a DSP block allows the leveraging of the high-performance cascading capabilities of the Xilinx® DSP blocks.

For most standalone counters and accumulators, slice logic is favored by default in auto mode. Change it to yes in order to force implementation onto DSP blocks.

In auto mode, DSP Utilization Ratio (DSP_UTILIZATION_RATIO) controls DSP block resource utilization. XST tries to utilize all DSP block resources available on the targeted device.

For more information, see:

Arithmetic Operators DSP Block Resources

Counters and Accumulators Related Constraints

- Use DSP Block (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
Counters and Accumulators Reporting

Counters and accumulators are identified during Advanced HDL Synthesis, by a combination of a register and an adder/subtractor macro previously inferred during HDL Synthesis. The following report example shows this sequence of events.

```
Synthesizing Unit <example>.
  Found 4-bit register for signal <cnt>.
  Found 4-bit register for signal <acc>.
  Found 4-bit adder for signal <n0005> created at line 29.
  Found 4-bit adder for signal <n0006> created at line 30.
Summary:
inferred 2 Adder/Subtractor(s).
inferred 8 D-type flip-flop(s).
Unit <example> synthesized.
```

```
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors : 2
4-bit adder : 2
# Registers : 2
4-bit register : 2
```

```
Synthesizing (advanced) Unit <example>.
The following registers are absorbed into counter <cnt>: 1 register on signal <cnt>.
The following registers are absorbed into accumulator <acc>: 1 register on signal <acc>.
Unit <example> synthesized (advanced).
```

```
Advanced HDL Synthesis Report

Macro Statistics
# Counters : 1
4-bit up counter : 1
# Accumulators : 1
4-bit up accumulator : 1
```

Counters and Accumulators Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
4-Bit Unsigned Up Accumulator with Synchronous Reset VHDL Coding Example

```vhdl
-- 4-bit Unsigned Up Accumulator with synchronous Reset
-- File: HDL_Coding_Techniques/accumulators/accumulators_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity accumulators_2 is
  generic (WIDTH : integer := 4);
  port (clk : in std_logic;
         rst : in std_logic;
         D : in std_logic_vector(WIDTH-1 downto 0);
         Q : out std_logic_vector(WIDTH-1 downto 0));
end accumulators_2;

architecture archi of accumulators_2 is
  signal cnt : std_logic_vector(WIDTH-1 downto 0);
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if (rst = '1') then
        cnt <= (others => '0');
      else
        cnt <= cnt + D;
      end if;
    end if;
  end process;
  Q <= cnt;
end archi;
```

4-Bit Unsigned Down Counter With a Synchronous Load Verilog Coding Example

```verilog
// 4-bit unsigned down counter with a synchronous load.
// File: HDL_Coding_Techniques/counters/counters_31.v
//
module v_counters_31 (clk, load, Q);

parameter WIDTH = 4;
input clk;
input load;
output [WIDTH-1:0] Q;
reg [WIDTH-1:0] cnt;

always @(posedge clk)
begin
  if (load)
    cnt <= {WIDTH{1'b1}};
  else
    cnt <= cnt - 1'b1;
end
assign Q = cnt;
endmodule
```
Shift Registers

This section discusses XST HDL Coding Techniques for Shift Registers, and includes:

- About Shift Registers
- Describing Shift Registers
- Shift Registers Implementation
- Shift Registers Related Constraints
- Shift Registers Reporting
- Shift Registers Coding Examples

About Shift Registers

A shift register is a chain of flip-flops allowing propagation of data across a fixed (static) number of latency stages. In Dynamic Shift Registers, the length of the propagation chain dynamically varies during circuit operation.

A static shift register usually involves:

- A clock
- An optional clock enable
- A serial data input
- A serial data output

You can include additional functionality, such as reset, set, or parallel load logic. In this case however, XST may not always be able to take advantage of dedicated SRL-type primitives for reduced device utilization and optimized performance. Xilinx® recommends removing such logic, and loading the desired contents serially instead.

Describing Shift Registers

The two general approaches to describing the core functionality of a shift register are shown below.

Concatenation Operator VHDL Coding Example

In a compact way, using a concatenation operator.

\[
\text{shreg} <= \text{shreg (6 downto 0)} \& \text{SI};
\]

For Loop VHDL Coding Example

Using a for loop construct.

\[
\begin{align*}
\text{for } i \text{ in } 0 \text{ to } 6 \text{ loop} \\
\quad & \text{shreg}(i+1) <= \text{shreg}(i); \\
\quad & \text{end loop;} \\
\quad & \text{shreg}(0) <= \text{SI};
\end{align*}
\]
Shift Registers Implementation

This section discusses Shift Registers Implementation, and includes:

- Shift Registers SRL-Based Implementation
- Implementing Shift Registers on Block RAM
- Implementing Shift Registers on LUT RAM

Shift Registers SRL-Based Implementation

XST implements inferred shift registers on SRL-type resources such as SRL16, SRL16E, SRLC16, SRLC16E, and SRLC32E.

Depending on the length of the shift register, XST implements it on a single SRL-type primitive, or takes advantage of the cascading capability of SRLC-type primitives. XST also tries to take advantage of this cascading capability if the rest of the design uses some intermediate positions of the shift register.

You can also implement delay lines on RAM resources (block RAM or LUT RAM), instead of SRL-type resources. This technique brings significant benefits, especially with respect to power savings, when delay lines become relatively long.

However, on block RAM or LUT RAM resources, XST cannot implement a shift register as outlined in Describing Shift Registers. You must explicitly describe the RAM-based implementation, as shown in the following coding examples.

Implementing Shift Registers on Block RAM

One of the key block RAM features being leveraged is the read-first synchronization mode. Another important element of this technique is a counter that sequentially scans the addressable space, and counts back to zero when reaching the delay line length minus two. In order to ensure maximal performance, use the block RAM output latch and optional output register stage. As a result, a 512-deep delay line, for example, uses 510 addressable data words in the RAM, while the data output latch and optional output register provide the last two stages.

For more information, see:

RAMs

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Chapter 7: XST HDL Coding Techniques

512-Deep 8-Bit Delay Line Implemented on Block RAM VHDL Coding Example

---
-- A 512-deep 8-bit delay line implemented on block RAM
-- 510 stages implemented as addressable memory words
-- 2 stages implemented with output latch and optional output register for
-- optimal performance
--
-- File: HDL_Coding_Techniques/shift_registers/delayline_bram_512.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity srl_512_bram is
  generic(
    LENGTH : integer := 512;
    ADDRWIDTH : integer := 9;
    WIDTH : integer := 8);
  port(
    CLK : in std_logic;
    SHIFT_IN : in std_logic_vector(WIDTH-1 downto 0);
    SHIFT_OUT : out std_logic_vector(WIDTH-1 downto 0));
end srl_512_bram;

architecture behavioral of srl_512_bram is
  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  signal SHIFT_TMP : std_logic_vector(WIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-3) of std_logic_vector(WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));
  begin
    counter : process (CLK)
    begin
      if CLK'event and CLK = '1' then
        if CNTR = conv_std_logic_vector(LENGTH-3, ADDRWIDTH) then
          CNTR <= (others => '0');
        else
          CNTR <= CNTR + '1';
        end if;
      end if;
    end process counter;

    memory : process (CLK)
    begin
      if CLK'event and CLK = '1' then
        RAM(conv_integer(CNTR)) <= SHIFT_IN;
        SHIFT_TMP <= RAM(conv_integer(CNTR));
        SHIFT_OUT <= SHIFT_TMP;
      end if;
    end process memory;
  end behavioral;
514-Deep 8-Bit Delay Line Implemented on Block RAM VHDL Coding Example

```
--
-- A 514-deep 8-bit delay line implemented on block RAM
-- 512 stages implemented as addressable memory words
-- 2 stages implemented with output latch and optional output register for
-- optimal performance
--
-- File: HDL_Coding_Techniques/shift_registers/delayline_bram_514.vhd
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity srl_514_bram is
  generic
    LENGTH : integer := 514;
    ADDRWIDTH : integer := 9;
    WIDTH : integer := 8;
  port
    (CLK : in std_logic;
     SHIFT_IN : in std_logic_vector(WIDTH-1 downto 0);
     SHIFT_OUT : out std_logic_vector(WIDTH-1 downto 0));
end srl_514_bram;

architecture behavioral of srl_514_bram is

  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  signal SHIFT_TMP : std_logic_vector(WIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-3) of std_logic_vector(WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

begin

  counter : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      CNTR <= CNTR + '1';
    end if;
  end process counter;

  memory : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      RAM(conv_integer(CNTR)) <= SHIFT_IN;
      SHIFT_TMP <= RAM(conv_integer(CNTR));
      SHIFT_OUT <= SHIFT_TMP;
    end if;
  end process memory;

end behavioral;
```

Implementing Shift Registers on LUT RAM

You can also implement such a shift register on distributed RAM, with the last stage implemented with a separate register. For example, a 128-deep delay line uses a LUT RAM with 127 addressable data words, and a final register stage.

For more information, see:

RAMs

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Chapter 7: XST HDL Coding Techniques

128-Deep 8-Bit Delay Line Implemented on LUT RAM VHDL Coding Example

--
-- A 128-deep 8-bit delay line implemented on LUT RAM
-- 127 stages implemented as addressable memory words
-- Last stage implemented with an external register
--
-- File: HDL_Coding_Techniques/shift_registers/delayline_lutram_128.vhd
--
library IEEE;
use IEEE.STD_LOGIC_UNSIGNED.all;
use IEEE.STD_LOGIC_ARITH.all;

entity srl_128_lutram is
  generic (
    LENGTH : integer := 128;
    ADDRWIDTH : integer := 7;
    WIDTH : integer := 8);
  port (  
    CLK : in std_logic;
    SHIFT_IN : in std_logic_vector(WIDTH-1 downto 0);
    SHIFT_OUT : out std_logic_vector(WIDTH-1 downto 0));
end srl_128_lutram;

architecture behavioral of srl_128_lutram is

  signal CNTR : std_logic_vector(ADDRWIDTH-1 downto 0);
  type ram_type is array (0 to LENGTH-2) of std_logic_vector(WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

  attribute ram_style : string;
  attribute ram_style of RAM : signal is "distributed";

begin

  counter : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      if CNTR = conv_std_logic_vector(LENGTH-2, ADDRWIDTH) then
        CNTR <= (others => '0');
      else
        CNTR <= CNTR + '1';
      end if;
    end if;
  end process counter;

  memory : process (CLK)
  begin
    if CLK'event and CLK = '1' then
      RAM(conv_integer(CNTR)) <= SHIFT_IN;
      SHIFT_OUT <= RAM(conv_integer(CNTR));
    end if;
  end process memory;

end behavioral;

Shift Registers Related Constraints

Shift Register Extraction (SHREG_EXTRACT)
Shift Registers Reporting

During HDL Synthesis, XST initially identifies individual flip-flops. Actual recognition of shift registers occurs during Low Level Synthesis. The following report example shows this sequence of events.

=================================================================
* HDL Synthesis *
=================================================================
Synthesizing Unit <example>.
   Found 8-bit register for signal <tmp>.
      Summary:
         inferred 8 D-type flip-flop(s).
Unit <example> synthesized.

(...) 
=================================================================
* Advanced HDL Synthesis *
=================================================================
Advanced HDL Synthesis Report
Macro Statistics
   # Registers : 8
   Flip-Flops : 8
=================================================================

(...) 
=================================================================
* Low Level Synthesis *
=================================================================
Processing Unit <example> :
   Found 8-bit shift register for signal <tmp_7>.
Unit <example> processed.

(...) 
=================================================================
Final Register Report
Macro Statistics
   # Shift Registers : 1
   8-bit shift register : 1
=================================================================

Shift Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
32-Bit Shift Register VHDL Coding Example One

The following coding example uses the concatenation coding style.

```vhdl
--
-- 32-bit Shift Register
--    Rising edge clock
--    Active high clock enable
--    Concatenation-based template
--
-- File: HDL_Coding_Techniques/shift_registers/shift_registers_0.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_0 is
    generic (DEPTH: integer := 32);
    port (clk : in std_logic;
          clken : in std_logic;
          SI : in std_logic;
          SO : out std_logic);
end shift_registers_0;

architecture archi of shift_registers_0 is
    signal shreg: std_logic_vector(DEPTH-1 downto 0);
begin
    process (clk)
    begin
        if rising_edge(clk) then
            if clken = '1' then
                shreg <= shreg(DEPTH-2 downto 0) & SI;
            end if;
        end if;
    end process;
    SO <= shreg(DEPTH-1);
end archi;
```
32-Bit Shift Register VHDL Coding Example Two

The same functionality can also be described as follows.

```vhdl
--
-- 32-bit Shift Register
--    Rising edge clock
--    Active high clock enable
--    for loop-based template
--
-- File: HDL_Coding_Techniques/shift_registers/shift_registers_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity shift_registers_1 is
    generic (DEPTH: integer := 32);
    port (clk : in std_logic;
            clken : in std_logic;
            SI : in std_logic;
            SO : out std_logic);
end shift_registers_1;

architecture archi of shift_registers_1 is
    signal shreg: std_logic_vector(DEPTH-1 downto 0);
begin
    process (clk)
    begin
        if rising_edge(clk) then
            if clken = '1' then
                for i in 0 to DEPTH-2 loop
                    shreg(i+1) <= shreg(i);
                end loop;
                shreg(0) <= SI;
            end if;
        end if;
    end process;
    SO <= shreg(DEPTH-1);
end archi;
```
Chapter 7: XST HDL Coding Techniques

8-Bit Shift Register Verilog Coding Example One

The following coding example uses a concatenation to describe the register chain.

```verilog
//
// 8-bit Shift Register
//  Rising edge clock
//  Active high clock enable
//  Concatenation-based template
//
// File: HDL_Coding_Techniques/shift_registers/shift_registers_0.v
//
module v_shift_registers_0 (clk, clken, SI, SO);

  parameter WIDTH = 8;
  input  clk, clken, SI;
  output SO;
  reg [WIDTH-1:0] shreg;

  always @(posedge clk)
  begin
    if (clken)
      shreg = {shreg[WIDTH-2:0], SI};
  end

  assign SO = shreg[WIDTH-1];
endmodule
```

8-Bit Shift Register Verilog Coding Example Two

```verilog
//
// 8-bit Shift Register
//  Rising edge clock
//  Active high clock enable
//  For-loop based template
//
// File: HDL_Coding_Techniques/shift_registers/shift_registers_1.v
//
module v_shift_registers_1 (clk, clken, SI, SO);

  parameter WIDTH = 8;
  input  clk, clken, SI;
  output SO;
  reg [WIDTH-1:0] shreg;

  integer i;

  always @(posedge clk)
  begin
    if (clken)
      begin
        for (i = 0; i < WIDTH-1; i = i+1)
          shreg[i+1] <= shreg[i];
        shreg[0] <= SI;
      end
  end

  assign SO = shreg[WIDTH-1];
endmodule
```
Dynamic Shift Registers

This section discusses XST HDL Coding Techniques for Dynamic Shift Registers, and includes:

- About Dynamic Shift Registers
- Dynamic Shift Registers Related Constraints
- Dynamic Shift Registers Reporting
- Dynamic Shift Registers Coding Examples

About Dynamic Shift Registers

A dynamic shift register is a shift register the length of which can dynamically vary during circuit operation. Considering the maximal length that it can take during circuit operation, a shift register can be seen as a chain of flip-flops of that length, with a multiplexer selecting, in a given clock cycle, at which stage data is to be extracted from the propagation chain. The following figure summarizes this concept.

![Dynamic Shift Registers Diagram](image)

XST can infer dynamic shift registers of any maximal length, and implement them optimally using the SRL-type primitives available in the targeted device family.

Dynamic Shift Registers Related Constraints

Shift Register Extraction (SHREG_EXTRACT)
Dynamic Shift Registers Reporting

During HDL Synthesis, XST initially identifies flip-flops and multiplexers. Actual recognition of a dynamic shift register happens during Advanced HDL Synthesis, where XST determines the dependency between those basic macros. The following report example shows this sequence of events.

```
* HDL Synthesis *

Synthesizing Unit <example>.
   Found 1-bit 16-to-1 multiplexer for signal <Q>.
   Found 16-bit register for signal <SRL_SIG>.
   Summary:
      inferred 16 D-type flip-flop(s).
      inferred 1 Multiplexer(s).
Unit <example> synthesized.

(…)

* Advanced HDL Synthesis *

Synthesizing (advanced) Unit <example>.
   Found 16-bit dynamic shift register for signal <Q>.
Unit <example> synthesized (advanced).
```

```
HDL Synthesis Report
Macro Statistics
# Shift Registers : 1
16-bit dynamic shift register : 1
```

Dynamic Shift Registers Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from http://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
### 32-Bit Dynamic Shift Registers VHDL Coding Example

```vhdl
-- 32-bit dynamic shift register.
-- File: HDL_Coding_Techniques/dynamic_shift_registers/dynamic_shift_registers_1.vhd
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity example is
  generic (
    DEPTH : integer := 32;
    SEL_WIDTH : integer := 5
  );
  port(
    CLK : in std_logic;
    SI : in std_logic;
    CE : in std_logic;
    A : in std_logic_vector(SEL_WIDTH-1 downto 0);
    DO : out std_logic
  );
end example;

architecture rtl of example is
  type SRL_ARRAY is array (0 to DEPTH-1) of std_logic;
  signal SRL_SIG : SRL_ARRAY;
begin
  process (CLK)
  begin
    if rising_edge(CLK) then
      if CE = '1' then
        SRL_SIG <= SI & SRL_SIG(0 to DEPTH-2);
      end if;
    end if;
  end process;
  DO <= SRL_SIG(conv_integer(A));
end rtl;
```

XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices

UG687 (v 13.1) March 1, 2011  www.xilinx.com
32-Bit Dynamic Shift Registers Verilog Coding Example

//
// 32-bit dynamic shift register.
//
// File: HDL_Coding_Techniques/dynamic_shift_registers/dynamic_shift_registers_1.v
//
module v_dynamic_shift_registers_1 (CLK, CE, SEL, SI, DO);

parameter SELWIDTH = 5;
input CLK, CE, SI;
input [SELWIDTH-1:0] SEL;
output DO;

localparam DATAWIDTH = 2**SELWIDTH;
reg [DATAWIDTH-1:0] data;

assign DO = data[SEL];

always @(posedge CLK)
begin
  if (CE == 1'b1)
    data <= {data[DATAWIDTH-2:0], SI};
end
endmodule
Chapter 7: XST HDL Coding Techniques

Multiplexers

This section discusses XST HDL Coding Techniques for Multiplexers, and includes:

- About Multiplexers
- Multiplexers Implementation
- Multiplexers Verilog Case Implementation Style Parameter
- Multiplexers Related Constraints
- Multiplexers Reporting
- Multiplexers Coding Examples

About Multiplexers

Multiplexer macros can be inferred from different coding styles, involving either concurrent assignments, description in combinatorial processes or `always` blocks, or descriptions within sequential processes or `always` blocks. Description of multiplexers usually involve:

- `if-elsif` constructs
- `case` constructs

When using a `case` statement, make sure that all selector values are enumerated, or that a default statement defines what data is selected for selector values not explicitly enumerated. Failing to do so creates undesired latches. Similarly, if the multiplexer is described with an `if-elsif` construct, a missing `else` can also create undesired latches.

When the same data is to be selected for different values of the selector, you can use `don't care` to describe those selector values in a compact way.

Multiplexers Implementation

The decision to explicitly infer a multiplexer macro may depend on the nature of the multiplexer inputs, especially the amount of common inputs.

Multiplexers Verilog Case Implementation Style Parameter

You can use a Case Implementation Style Parameter to further characterize a `case` statement that you have described.

For more information, see:

Chapter 9, XST Design Constraints

Accepted values for a Case Implementation Style Parameter are:

- `none` (default)
  XST implements the behavior of the `case` statement as written.
- `full`
  XST considers that case statements are complete, and avoids latch creation, even if not all possible selector values are enumerated.
- `parallel`
  XST considers that the branches cannot occur simultaneously, and does not create priority encoding logic.
- `full-parallel`
  XST considers that `case` statements are complete and that the branches cannot occur simultaneously, and avoids latch creation and priority encoding logic.
XST issues an information message when a Case Implementation Style Parameter is actually taken advantage of. No message is issued if the statement is not needed given the characteristics of the case itself (for example, a full case parameter when the case it relates to enumerates all possible values of the selector).

Specifying full, parallel, or full-parallel can result in an implementation with a behavior that differs from the behavior of the initial model.

**Multiplexers Related Constraints**

**Enumerated Encoding (ENUM_ENCODING)**

**Multiplexers Reporting**

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

```
* HDL Synthesis *
*---------------------------------------------------------------*

Synthesizing Unit <example>.
  Found 1-bit 8-to-1 multiplexer for signal <o> created at line 11.
  Summary:
    inferred 1 Multiplexer(s).
  Unit <example> synthesized.

---------------------------------------------------------------

HDL Synthesis Report

Macro Statistics
# Multiplexers            : 1
  1-bit 8-to-1 multiplexer : 1

---------------------------------------------------------------
```

Explicit inference and reporting of multiplexers can vary depending on the targeted devices and the size of the multiplexer. For example, 4-to-1 multiplexers are not reported for Virtex®-6 or Spartan®-6 devices. For those devices, they are inferred for sizes of 8-to-1 and above.

**Multiplexers Coding Examples**

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Chapter 7: XST HDL Coding Techniques

8-to-1 1-Bit MUX Using an If Statement VHDL Coding Example

```vhdl
// 8-to-1 1-bit MUX using an If statement.
// File: HDL_Coding_Techniques/multiplexers/multiplexers_1.v
//
// module v_multiplexers_1 (di, sel, do);
// input [7:0] di;
// input [2:0] sel;
// output reg do;

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

8-to-1 1-Bit MUX Using an If Statement Verilog Coding Example

```verilog
// 8-to-1 1-bit MUX using an If statement.
// File: HDL_Coding_Techniques/multiplexers/multiplexers_1.v
//
// module v_multiplexers_1 (di, sel, do);
// input [7:0] di;
// input [2:0] sel;
// output reg do;

always @(sel or di)
begin
    if (sel == 3'b000) do = di[7];
    else if (sel == 3'b001) do = di[6];
    else if (sel == 3'b010) do = di[5];
    else if (sel == 3'b011) do = di[4];
    else if (sel == 3'b100) do = di[3];
    else if (sel == 3'b101) do = di[2];
    else if (sel == 3'b110) do = di[1];
    else
        do = di[0];
end
endmodule
```
8-to-1 1-Bit MUX Using a Case Statement VHDL Coding Example

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

entity multiplexers_2 is
    port (di : in std_logic_vector(7 downto 0);
          sel : in std_logic_vector(2 downto 0);
          do : out std_logic);
end multiplexers_2;

architecture archi of multiplexers_2 is
begin
    process (sel, di)
    begin
        case sel is
            when "000" => do <= di(7);
            when "001" => do <= di(6);
            when "010" => do <= di(5);
            when "011" => do <= di(4);
            when "100" => do <= di(3);
            when "101" => do <= di(2);
            when "110" => do <= di(1);
            when others => do <= di(0);
        end case;
    end process;
end archi;
```

8-to-1 1-Bit MUX Using a Case Statement Verilog Coding Example

```verilog
module v_multiplexers_2 (di, sel, do);
    input [7:0] di;
    input [2:0] sel;
    output reg do;
begin
    always @(sel or di)
    begin
        case (sel)
            3'b000 : do = di[7];
            3'b001 : do = di[6];
            3'b010 : do = di[5];
            3'b011 : do = di[4];
            3'b100 : do = di[3];
            3'b101 : do = di[2];
            3'b110 : do = di[1];
            default : do = di[0];
        endcase
    end
endmodule
```
8-to-1 1-Bit MUX Using Tristate Buffers Verilog Coding Example

//
// 8-to-1 1-bit MUX using tristate buffers.
//
// File: HDL_Coding_Techniques/multiplexers/multiplexers_3.v
//
module v_multiplexers_3 (di, sel, do);
    input [7:0] di;
    input [7:0] sel;
    output do;

    assign do = sel[0] ? di[0] : 1’bz;
endmodule
Arithmetic Operators

This section discusses XST HDL Coding Techniques for Arithmetic Operators, and includes:

- About Arithmetic Operators
- Arithmetic Operators Signed and Unsigned Support in XST
- Arithmetic Operators Implementation

About Arithmetic Operators

XST supports the following arithmetic operators:

- Adders, Subtractors, and Adders/Subtractors
- Multipliers
- Dividers
- Comparators

These basic arithmetic macros can serve as building blocks to more complex macros such as accumulators, multiply-add, and DSP filters.

Arithmetic Operators Signed and Unsigned Support in XST

XST supports signed and unsigned representation for the following operators:

- Adders
- Subtractors
- Comparators
- Multipliers

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

Verilog Signed/Unsigned Support

Without explicit specification of the representation, Verilog defines the following convention:

- Port, wire and reg vector types are treated as unsigned, unless explicitly declared to be signed.
- Integer variables are treated as signed, unless specified otherwise.
- Decimal numbers are signed.
- Based numbers are unsigned, unless specified otherwise.

Use the unsigned and signed keywords to explicitly force the representation of data types.

Verilog Signed/Unsigned Support Coding Example One

```verilog
input signed [31:0] example1;
reg unsigned [15:0] example2;
wire signed [31:0] example3;
```

Verilog Signed/Unsigned Support Coding Example Two

You can also force a based number to be signed, using the s notation in the base specifier.

```verilog
4'sd87
```
Verilog Signed/Unsigned Support Coding Example Three

In addition, you can ensure proper type casting with the $signed and $unsigned conversion functions.

```verilog
wire [7:0] udata;
wire [7:0] sdata;
sassign sdata = $signed(udata);
```

In Verilog, the type of an expression is defined only by its operands. It does not depend on the type of an assignment left-hand part. An expression type is resolved according to the following rules:

- Bit-select results are unsigned, regardless of the operands.
- Part-select results are unsigned, regardless of the operands, even if the part-select specifies the entire vector.
- Concatenate results are unsigned, regardless of the operands.
- Comparison results are unsigned, regardless of the operands.
- The sign and bit length of any self-determined operand is determined by the operand itself and is independent of the rest of the expression. If you use context-determined operands, review the additional guidelines in the Verilog LRM.

**VHDL Signed/Unsigned Support**

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

### Unsigned Arithmetic

<table>
<thead>
<tr>
<th>PACKAGE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric_std</td>
<td>unsigned</td>
</tr>
<tr>
<td>std_logic_arith</td>
<td>unsigned</td>
</tr>
<tr>
<td>std_logic_unsigned</td>
<td>std_logic_vector</td>
</tr>
</tbody>
</table>

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

### Signed Arithmetic

<table>
<thead>
<tr>
<th>PACKAGE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>numeric_std</td>
<td>signed</td>
</tr>
<tr>
<td>std_logic_arith</td>
<td>signed</td>
</tr>
<tr>
<td>std_logic_signed</td>
<td>std_logic_vector</td>
</tr>
</tbody>
</table>

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

This section discusses Arithmetic Operators Implementation.

Arithmetic Operators Slice Logic

When implementing arithmetic macros on slice logic, XST leverages some of the features of the Xilinx® CLB structure, in particular the dedicated carry logic available to implement fast, efficient arithmetic functions.

Arithmetic Operators DSP Block Resources

Virtex®-6 and Spartan®-6 devices contain dedicated high-performance arithmetic blocks (DSP blocks). DSP blocks are available in varying quantities depending on the targeted device. They can be configured to implement various arithmetic functions. If leveraged to their full potential, DSP blocks can implement a fully pipelined preadder-multiply-add or preadder-multiply-accumulate function.

XST tries to leverage those resources as much as possible for high-performance and power-efficient implementation of arithmetic logic. Multipliers and Multiply-Add and Multiply-Accumulate discuss the details of implementation on DSP blocks.

- Implementation of arithmetic macros on either slice logic or DSP block resources is controlled by Use DSP Block (USE_DSP48), with a default value of auto.

- In automatic mode, XST takes into account actual availability of DSP block resources, in order to avoid overmapping the targeted device. XST may use all DSP resources available on the device. DSP Utilization Ratio (DSP_UTILIZATION_RATIO) forces XST to leave some of those resources unallocated.

- Some arithmetic macros, such as standalone adders, accumulators, and counters, are not implemented on DSP blocks by default. To force implementation, apply Use DSP Block (USE_DSP48) with a value of yes.

- To take advantage of registers for pipelining of arithmetic functions implemented on DSP blocks, describe the registers with an optional clock enable. Registers are also optionally synchronously resettable. Asynchronous reset logic prevents such implementation and should be avoided.

- When describing unsigned arithmetic, remember that DSP block resources assume signed operands. You cannot map unsigned operands to the full width of a single DSP block. For example, XST can implement up to a 25x18-bit signed multiplication on a single Virtex-6 DSP48E1 block. It can implement up to a 24x17-bit unsigned product only on that same single block, with most significant bits of the DSP block inputs set to 0.

For more information about DSP block resources, and the advantages of proper Hardware Description Language (HDL) coding practices, see:

Comparators

This section discusses XST HDL Coding Techniques for Comparators, and includes:

- About Comparators
- Comparators Related Constraints
- Comparators Reporting
- Comparators Coding Examples

About Comparators

XST recognizes comparators of all possible types:

- equal
- not equal
- larger than
- larger than or equal
- less than
- less than or equal

Comparators Related Constraints

None

Comparators Reporting

Equal or not equal comparison of a signal or a variable to a constant does not lead to an explicit comparator macro inference, since it is directly optimized to Boolean logic by XST. For all other comparison situations, comparator macro inference are reported as shown below:

```
*-----------------------------------------------*
| HDL Synthesis                             *
*-----------------------------------------------*

Synthesizing Unit <example>.
  Found 8-bit comparator lessequal for signal <n0000> created at line 8
  Found 8-bit comparator greater for signal <cmp2> created at line 15
  Summary:
    inferred  2 Comparator(s).
  Unit <example> synthesized.
```

HDL Synthesis Report

Macro Statistics

<table>
<thead>
<tr>
<th># Comparators</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit comparator greater</td>
<td>2</td>
</tr>
<tr>
<td>8-bit comparator lessequal</td>
<td>1</td>
</tr>
</tbody>
</table>

Comparators Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a `summary.txt` file listing all examples together with a brief overview.
Chapter 7: XST HDL Coding Techniques

Unsigned 8-Bit Greater or Equal Comparator VHDL Coding Example

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

entity comparators_1 is
  generic (WIDTH : integer := 8);
  port (A,B : in std_logic_vector(WIDTH-1 downto 0);
        CMP : out std_logic);
end comparators_1;

architecture archi of comparators_1 is
begin
  CMP <= '1' when A >= B else '0';
end archi;
```

Unsigned 8-Bit Less Than Comparator Verilog Coding Example

```verilog
module v_comparators_1 (A, B, CMP);
  parameter WIDTH = 8;
  input [WIDTH-1:0] A;
  input [WIDTH-1:0] B;
  output CMP;
  assign CMP = (A < B) ? 1'b1 : 1'b0;
endmodule
```

File: HDL_Coding_Techniques/comparators/comparators_1.vhd
Dividers

This section discusses XST HDL Coding Techniques for Dividers, and includes:

- About Dividers
- Dividers Related Constraints
- Dividers Reporting
- Dividers Coding Examples

About Dividers

Dividers are supported only when:

- The divisor is constant and a power of 2. Such a description is implemented as a shifter.
- Both operands are constant.

In all other cases, XST exits with a specific error message.

Dividers Related Constraints

None

Dividers Reporting

None

Dividers Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Division By Constant 2 VHDL Coding Example

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

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

architecture archi of divider_1 is
begin
    DO <= DI / 2;
end archi;
```
Division By Constant 2 Verilog Coding Example

```verilog
// Division By Constant 2
// File: HDL_Coding_Techniques/dividers/dividers_1.v
//
module v_divider_1 (DI, DO);
  input [7:0] DI;
  output [7:0] DO;
  assign DO = DI / 2;
endmodule
```
Adders, Subtractors, and Adders/Subtractors

This section discusses XST HDL Coding Techniques for Adders, Subtractors, and Adders/Subtractors, and includes:

- About Adders, Subtractors, and Adders/Subtractors
- Describing a Carry Output
- Adders, Subtractors, and Adders/Subtractors Implementation
- Adders, Subtractors, and Adders/Subtractors Related Constraints
- Adders, Subtractors, and Adders/Subtractors Reporting
- Adders, Subtractors, and Adders/Subtractors Coding Examples

About Adders, Subtractors, and Adders/Subtractors

XST recognizes adders, subtractors, and adders/subtractors. Adders can be described with an optional carry input, and an optional carry output. Subtractors can be described with an optional borrow input.

Describing a Carry Output

A carry output is usually modeled by assigning the result of the described addition to a signal with an extra bit over the longest operand.

Describing a Carry Output VHDL Coding Example One

```vhdl
input [7:0] A;
input [7:0] B;
wire [8:0] res;
wire carryout;
assign res = A + B;
assign carryout = res[8];
```

Describing a Carry Output VHDL Coding Example Two

If you intend to describe an adder with a carry output in VHDL, carefully review the arithmetic package you plan to use. For example, the method above is not applicable when using std_logic_unsigned, because the size of the result is necessarily equal to the size of the longest argument. In this case, you can adjust the size of the operands as shown in the following example.

```vhdl
signal A : std_logic_unsigned(7 downto 0);
signal B : std_logic_unsigned(7 downto 0);
signal res : std_logic_unsigned(8 downto 0);
signal carryout : std_logic;
res <= ("0" & A) + ("0" & B);
carryout <= res[8];
```
You can also convert the operands to type `integer`, and convert the result of the addition back to `std_logic_vector` as follows. The `conv_std_logic_vector` conversion function is contained in package `std_logic_arith`. The unsigned + operation is contained in `std_logic_unsigned`.

```vhdl
signal A            : std_logic_vector(7 downto 0);
signal B            : std_logic_vector(7 downto 0);
signal res          : std_logic_vector(8 downto 0);
signal carryout     : std_logic;

res <= conv_std_logic_vector(conv_integer(A) + conv_integer(B)),9);
carryout <= res[8];
```

### Adders, Subtractors, and Adders/Subtractors Implementation

Standalone adders, subtractors, or adder/subtractors are not implemented on DSP block resources by default. They are instead synthesized using carry logic.

To force the implementation of a simple adder, subtractor, or adder/subtractor to the DSP block, apply `Use DSP Block (USE_DSP48)` with a value of `yes`.

XST supports the one level of output registers into DSP48 blocks. If the Carry In or Add/Sub operation selectors are registered, XST pushes these registers into the DSP48 as well.

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

Macro implementation on DSP48 blocks is controlled by `DSP Utilization Ratio (DSP_UTILIZATION_RATIO)` with a default value of `auto`. In `auto` mode, if an adder/subtractor is a part of a more complex macro such as a filter, XST automatically places it on the DSP block. Otherwise, XST implements adders/subtractors using LUTs.

To force XST to push these macros into a DSP48, set the value of `Use DSP Block (USE_DSP48)` to `yes`. When placing an Adder/Subtractor on a DSP block, XST checks to see if it is connected to other DSP chains. If so, XST tries to take advantage of fast DSP connections, and connects this adder/subtractor to the DSP chain using these fast connections. When implementing adders/subtractors on DSP48 blocks, XST performs automatic DSP48 resource control.

To deliver the best performance, XST tries to infer and implement the maximum macro configuration, including as many registers in the DSP48 as possible. Use `Keep (KEEP)` to shape a macro in a specific way. For example, to exclude the first register stage from the DSP48, `Keep (KEEP)` on the outputs of these registers.

### Adders, Subtractors, and Adders/Subtractors Related Constraints

- Use DSP Block (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)
Adders, Subtractors, and Adders/Subtractors Reporting

Synthesizing Unit <example>.

- Found 8-bit adder for signal <sum> created at line 9.
- Summary:
  - inferred 1 Adder/Subtractor(s).
- Unit <example> synthesized.

HDL Synthesis Report

Macro Statistics

- # Adders/Subtractors: 1
- 8-bit adder: 1

For adders with a carry input, two separate adder macros are initially inferred and reported in HDL Synthesis. They are later grouped together into a single adder macro with carry input during Advanced HDL Synthesis, as reflected in the Advanced HDL Synthesis Report. Similarly, for descriptions of subtractors with borrow input, two separate subtractor macros are initially inferred and later grouped together in Advanced HDL Synthesis. Carry output logic is not explicitly reported.

Adders, Subtractors, and Adders/Subtractors Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Unsigned 8-Bit Adder VHDL Coding Example

```vhdl
-- -- Unsigned 8-bit Adder
-- -- File: HDL_Coding_Techniques/adders/adders_1.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity adders_1 is
  generic (WIDTH : integer := 8);
  port (A, B : in std_logic_vector(WIDTH-1 downto 0);
        SUM : out std_logic_vector(WIDTH-1 downto 0));
end adders_1;
architecture archi of adders_1 is
begin
  SUM <= A + B;
end archi;
```
Unsigned 8-Bit Adder with Carry In Verilog Coding Example

```verbatim
//
// Unsigned 8-bit Adder with Carry In
//
// File: HDL_Coding_Techniques/adders/adders_2.v
//
module v_adders_2 (A, B, CI, SUM);

    parameter WIDTH = 8;
    input [WIDTH-1:0] A;
    input [WIDTH-1:0] B;
    input CI;
    output [WIDTH-1:0] SUM;

    assign SUM = A + B + CI;

endmodule
```
Multipliers

This section discusses XST HDL Coding Techniques for Multipliers, and includes:

- About Multipliers
- Multipliers Implementation
- Multipliers Related Constraints
- Multipliers Reporting
- Multipliers Coding Examples

About Multipliers

XST infers multiplier macros from product operators found in the Hardware Description Language (HDL) source code.

The size of the resulting signal is equal to the sum of the two operand sizes. For example, multiplying a 16-bit signal by an 8-bit signal produces a result on 24 bits. If you do not intend to use all most significant bits of a product, consider reducing the size of operands to the minimum needed, in particular if the multiplier macro will be implemented on slice logic.

Multipliers Implementation

This section discusses Multipliers Implementation, and includes:

- About Multipliers Implementation
- DSP Block Implementation
- Slice Logic Implementation
- Multiplication to a Constant

About Multipliers Implementation

Multiplier macros can be implemented on the following device resources:

- Slice logic
- DSP blocks

Implementing a multiplier on either slice logic or with DSP block resources is controlled by Use DSP Block (USE_DSP48) with a default value of auto.

In auto mode:

- XST seeks to implement a multiplier on DSP block resources, provided that its operands have a minimal size. The minimal size can vary depending on the targeted device family. Implementation on DSP block can be forced with a value of yes.
- XST takes into account the actual availability of DSP block resources, in order to avoid overmapping the targeted device. XST may use all DSP resources available on the device. DSP Utilization Ratio (DSP_UTILIZATION_RATIO) forces XST to leave some of those resources unallocated.

To specifically force implementation of a multiplier to slice logic or DSP block, set Use DSP Block (USE_DSP48) to no (slice logic) or yes (DSP block) on the appropriate signal, entity, or module.
Chapter 7: XST HDL Coding Techniques

DSP Block Implementation

When implementing a multiplier in a single DSP block, XST seeks to take advantage of pipelining capabilities of DSP blocks, pulling up to:

- Two levels of registers present on the multiplication operands
- Two levels of registers present behind the multiplication

When a multiplier does not fit in a single DSP block, XST automatically decomposes the macro to implement it using either several DSP blocks, or a hybrid solution involving both DSP blocks and slice logic. The implementation choice is driven by the size of operands, and is aimed at maximizing performance.

The performance of implementations based on multiple DSP blocks can be further improved if XST is instructed to perform pipelining. To do so, apply Multiplier Style (MULT_STYLE) with a value of pipe_block. XST automatically calculates the ideal number of register stages needed to maximize performance of a given multiplier. If they are available, XST moves them in order to achieve the desired goal. If an insufficient amount of latencies is found, XST issues the following HDL Advisor message during Advance HDL Synthesis. You can insert the suggested amount of additional register stages behind the multiplication.

INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the multiplier Mmult_n0005 by adding 2 register level(s).

Use Keep (KEEP) to restrict absorption of registers into DSP blocks. For example, to exclude a register present on an operand of the multiplier from absorption into the DSP block, place Keep (KEEP) on the output of the register.

Slice Logic Implementation

When Use DSP Block (USE_DSP48) is set to auto, most multipliers are implemented on DSP block resources, provided that one or more latency stages is available, and within the limits of available DSP blocks on the targeted device.

To force a multiplier to be implemented on slice logic, apply Use DSP Block (USE_DSP48) with a value of no.

For a multiplier implemented on slice logic, XST looks for pipelining opportunities around the operator, and moves those registers in order to reduce data path length. Pipelining can therefore greatly increase the performance of large multipliers. The effect of pipelining is similar to Flip-Flop Retiming.

To insert pipeline stages:
1. Describe the registers.
2. Place them after the multiplier.
3. Set Multiplier Style (MULT_STYLE) to:
   pipe_lut

Multiplication to a Constant

XST can select between the following dedicated implementation methods when one argument of the multiplication is a constant:

- Constant Coefficient Multiplier (CCM) implementation
- Canonical Signed Digit (CSD) implementation

These methods are applicable only if the multiplication is implemented on slice logic.
The level of optimization depends on the characteristics of the constant operand. In some cases, the CCM implementation may not be better than the default slice logic implementation. Therefore, XST automatically chooses between CCM or standard multiplier implementation. The CSD method cannot be automatically chosen. Use Multiplier Style (MULT STYLE) to:

- Force CSD implementation
- Force CCM implementation

XST does not use the CCM or CSD implementations if:

- The multiplication is signed.
- One of the operands is larger than 32 bits.

### Multipliers Related Constraints

- Use DSP Block (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)
- Multiplier Style (MULT STYLE)
Multipliers Reporting

Multipliers are inferred during HDL Synthesis. Absorption of registers by a multiplier macro can occur during Advanced HDL Synthesis, as shown by the following HDL Synthesis Report.

--- HDL Synthesis ---

Synthesizing Unit <v_multipliers_11>.
- Found 8-bit register for signal <RB>.
- Found 24-bit register for signal <RES>.
- Found 16-bit register for signal <RA>.
- Found 16x8-bit multiplier for signal <n0005> created at line 20.

Summary:
- inferred 1 Multiplier(s).
- inferred 48 D-type flip-flop(s).

Unit <v_multipliers_11> synthesized.

--- HDL Synthesis Report ---

Macro Statistics
# Multipliers : 1
16x8-bit multiplier : 1
# Registers : 3
16-bit register : 1
24-bit register : 1
8-bit register : 1

--- Advanced HDL Synthesis ---

Synthesizing (advanced) Unit <v_multipliers_11>.
- Found pipelined multiplier on signal <n0005>:
  - 1 pipeline level(s) found in a register connected to the multiplier macro output.
  - Pushing register(s) into the multiplier macro.
  - 1 pipeline level(s) found in a register on signal <RA>.
  - Pushing register(s) into the multiplier macro.
  - 1 pipeline level(s) found in a register on signal <RB>.
  - Pushing register(s) into the multiplier macro.

INFO:Xst:2385 - HDL ADVISOR - You can improve the performance of the multiplier Mmult_n0005 by adding 1 register level(s).

Unit <v_multipliers_11> synthesized (advanced).

--- Advanced HDL Synthesis Report ---

Macro Statistics
# Multipliers : 1
16x8-bit registered multiplier : 1

--- Multipliers Coding Examples ---

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Unsigned 8x4-Bit Multiplier VHDL Coding Example

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

entity multipliers_1 is
    generic (WIDTHA : integer := 8;
              WIDTHB : integer := 4);
    port (A : in std_logic_vector(WIDTHA-1 downto 0);
          B : in std_logic_vector(WIDTHB-1 downto 0);
          RES : out std_logic_vector(WIDTHA+WIDTHB-1 downto 0));
end multipliers_1;

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

Unsigned 32x24-Bit Multiplier Verilog Coding Example

module v_multipliers_11 (clk, A, B, RES);
    parameter WIDTHA = 32;
    parameter WIDTHB = 24;
    input clk;
    input [WIDTHA-1:0] A;
    input [WIDTHB-1:0] B;
    output [WIDTHA+WIDTHB-1:0] RES;
    reg [WIDTHA-1:0] rA;
    reg [WIDTHB-1:0] rB;
    reg [WIDTHA+WIDTHB-1:0] M [3:0];
    integer i;
    always @(posedge clk)
    begin
        rA <= A;
        rB <= B;
        M[0] <= rA * rB;
        for (i = 0; i < 3; i = i+1)
            M[i+1] <= M[i];
        end
        assign RES = M[3];
    endmodule
Multiply-Add and Multiply-Accumulate

This section discusses XST HDL Coding Techniques for Multiply-Add and Multiply-Accumulate, and includes:

- About Multiply-Add and Multiply-Accumulate
- Multiply-Add and Multiply-Accumulate Implementation
- Multiply-Add and Multiply-Accumulate Related Constraints
- Multiply-Add and Multiply-Accumulate Reporting
- Multiply-Add and Multiply-Accumulate Coding Examples

About Multiply-Add and Multiply-Accumulate

A multiply-add, multiply-sub, multiply-add/sub or multiply-accumulate macro is inferred during Advanced HDL Synthesis by aggregation of a multiplier, an adder/subtractor, and registers previously inferred during the HDL Synthesis phase.

Multiply-Add and Multiply-Accumulate Implementation

An inferred multiply-add or multiply-accumulate macro can be implemented on DSP block resources available on Xilinx® devices. In this case, XST tries to take advantage pipelining capabilities of DSP blocks, pulling up to:

- Two register stages present on the multiplication operands
- One register stage present behind the multiplication
- One register stage found behind the adder, subtractor, or adder/subtractor
- One register stage on the add/sub selection signal
- One register stage on the adder optional carry input

XST can implement a multiply accumulate in a DSP48 block if its implementation requires only a single DSP48 resource. If the macro exceeds the limits of a single DSP48, XST processes it as two separate Multiplier and Accumulate macros, making independent decisions on each macro.

Macro implementation on Xilinx DSP block resources is controlled by Use DSP Block (USE_DSP48) with a default value of auto. In auto mode, XST implements multiply-add and multiply-accumulate macros taking into account DSP block resources availability in the targeted device. XST may use up to all available DSP resources. DSP Utilization Ratio (DSP_UTILIZATION_RATIO) forces XST to leave some of those resources unallocated.

XST tries to maximize circuit performance by leveraging all pipelining capabilities of DSP blocks, looking for all opportunities to absorb registers into a multiply-add or multiply-accumulate macro. Use Keep (KEEP) to restrict absorption of registers into DSP blocks. For example, to exclude a register present on an operand of the multiplier from absorption into the DSP block, apply Keep (KEEP) on the output of this register.

Multiply-Add and Multiply-Accumulate Related Constraints

- Use DSP Block (USE_DSP48)
- DSP Utilization Ratio (DSP_UTILIZATION_RATIO)
- Keep (KEEP)
Multiply-Add and Multiply-Accumulate Reporting

XST reports the details of inferred multipliers, accumulators and registers at HDL Synthesis. Information about the composition of those macros into a multiply-add or multiply-accumulate macro can be found in the Advanced HDL Synthesis section. Both types of functionalities are accounted for under the unified MAC denomination.

---

HDL Synthesis

Synthesizing Unit <v_multipliers_7a>.
Found 16-bit register for signal <accum>.
Found 16-bit register for signal <mult>.
Found 16-bit adder for signal <n0058> created at line 26.
Found 8x8-bit multiplier for signal <n0005> created at line 18.
Summary:
inferred 1 Multiplier(s).
inferred 1 Adder/Subtractor(s).
inferred 32 D-type flip-flop(s).
Unit <v_multipliers_7a> synthesized.

HDL Synthesis Report

Macro Statistics
# Multipliers : 1
8x8-bit multiplier : 1
# Adders/Subtractors : 1
16-bit adder : 1
# Registers : 2
16-bit register : 2

---

Advanced HDL Synthesis

Synthesizing (advanced) Unit <v_multipliers_7a>.
The following registers are absorbed into accumulator <accum>: 1 register on signal <accum>.
Multiplier <Mmult_n0005> in block <v_multipliers_7a> and accumulator <accum> in block <v_multipliers_7a> are combined into a MAC<Mmac_n0005>.
The following registers are also absorbed by the MAC: <mult> in block <v_multipliers_7a>.
Unit <v_multipliers_7a> synthesized (advanced).

Advanced HDL Synthesis Report

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

---

Multiply-Add and Multiply-Accumulate Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
# Multiplier Up Accumulate with Register After Multiplication

**VHDL Coding Example**

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

department multipliers_7a is
    generic (p_width: integer:=8);
    port (clk, reset: in std_logic;
          A, B: in std_logic_vector(p_width-1 downto 0);
          RES: out std_logic_vector(p_width*2-1 downto 0));
end multipliers_7a;

architecture beh of multipliers_7a is
    signal mult, accum: std_logic_vector(p_width*2-1 downto 0);
begin
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (reset = '1') then
                accum <= (others => '0');
                mult <= (others => '0');
            else
                accum <= accum + mult;
                mult <= A * B;
            end if;
        end if;
    end process;
    RES <= accum;
end beh;
```

---

**Download:** ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip

**File:** HDL_Coding_Techniques/multipliers/multipliers_7a.vhd
Multiplier Up Accumulate Verilog Coding Example

//
// Multiplier Up Accumulate with:
//  Registered operands
//  Registered multiplication
//  Accumulation
//
// File: HDL_Coding_Techniques/multipliers/multiply_accum_2.v
//
module v_multiply_accum_2 (clk, rst, A, B, RES);
  parameter WIDTH = 8;
  input clk;
  input rst;
  input [WIDTH-1:0] A, B;
  output [2*WIDTH-1:0] RES;
  reg [WIDTH-1:0] rA, rB;
  reg [2*WIDTH-1:0] mult, accum;

  always @(posedge clk)
  begin
    if (rst)
      begin
        rA <= {WIDTH{1'b0}};
        rB <= {WIDTH{1'b0}};
        mult <= (2*WIDTH{1'b0});
        accum <= (2*WIDTH{1'b0});
      end
    else begin
      rA <= A;
      rB <= B;
      mult <= rA * rB;
      accum <= accum + mult;
    end
  end

  assign RES = accum;
endmodule
Extended DSP Inferencing

This section discusses Extended DSP Inferencing, and includes:

- About Extended DSP Inferencing
- Symmetric Filters
- Extended DSP Inferencing Coding Examples

About Extended DSP Inferencing

In addition to finer grained inferencing capabilities of such basic functionalities as latency stages (registers), multiply, multiply-add/subtract, accumulate, multiply-accumulate, and ROM, XST offers extended inferencing capabilities for describing filters with portable behavioral source code.

XST attempts to understand the existence of any contextual relationship between basic functional elements, and to leverage the powerful features of the DSP block resources available on Xilinx® devices (pipelining stages, cascade paths, pre-adder stage, time multiplexing), for high performance implementation and power reduction.

To optimally leverage DSP block capabilities, use an adder chain instead of an adder tree as the backbone of the filter description. Some HDL language features, such as for generate in VHDL, facilitate describing a filter in this way, and ensure maximal readability and scalability of the code.

For more information on DSP block resources and how to leverage them, see:

- Virtex®-6 FPGA DSP48E1 Slice User Guide, 
- Spartan®-6 FPGA DSP48A1 Slice User Guide, 

Symmetric Filters

The optional pre-adder capability in Xilinx® DSP Blocks was designed for symmetric filters. If you describe a symmetric coefficients filter, leverage the pre-adder to reduce the number of required DSP blocks by half.

Since XST does not automatically identify and factor symmetric coefficients, Xilinx does not recommend that you describe the filter in a generic manner, and assume that XST will be able to determine the symmetry. You must manually code the factorized form in order for XST to see the pre-adder opportunity and configure DSP blocks accordingly.

The SymSystolicFilter and SymTransposeConvFilter coding examples below show how to do so.

Note For the pre-adder capability to be leveraged, XST must identify a description that exactly matches the pre-adder size characteristics, even though your data may be of a lower width. This requirement is specific to the pre-adder. Use signed or unsigned extensions for explicit padding of the pre-adder operands, as illustrated in the following example, to ensure proper inference and implementation on DSP resources.
Pre-Adder Description With Explicit Data Extensions Example

--
-- Explicit padding of pre-adder operands
--
-- File: HDL_Coding_Techniques/dsp/preadder_padding.vhd

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

entity preadder_padding is
    generic (
        DATA_WIDTH : integer := 16
    );
    port (  
            clk : in std_logic;
            a : in signed(DATA_WIDTH-1 downto 0);
            b : in signed(DATA_WIDTH-1 downto 0);
            c : in signed(DATA_WIDTH-1 downto 0);
            d : in signed(2*DATA_WIDTH-1 downto 0);
            o : out signed(2*DATA_WIDTH-1 downto 0)
    );
end preadder_padding;

architecture behavioral of preadder_padding is
    constant PREADD_WIDTH : integer := 18;
    signal a_resized : signed(PREADD_WIDTH-1 downto 0);
    signal b_resized : signed(PREADD_WIDTH-1 downto 0);
    signal pre : signed(PREADD_WIDTH-1 downto 0);
    signal m : signed(DATA_WIDTH+PREADD_WIDTH-1 downto 0);
    signal p : signed(DATA_WIDTH+PREADD_WIDTH-1 downto 0);

begin
    assert DATA_WIDTH <= PREADD_WIDTH report "DATA_WIDTH exceeds limit of 18 bits" severity ERROR;

    a_resized <= RESIZE(a, PREADD_WIDTH);
    b_resized <= RESIZE(b, PREADD_WIDTH);

    process (clk)
    begin
        if rising_edge(clk) then
            pre <= a_resized + b_resized;
            m <= pre * c;
            p <= m + d;
            end if;
        end process;
    o <= RESIZE(p, 2*DATA_WIDTH);
end behavioral;

Extended DSP Inferencing Coding Examples

For Extended DSP Inferencing Coding Examples, go to the directory HDL_Coding_Techniques/dsp in
ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip, where each design is stored in its own subdirectory.
### DSP Reference Designs

<table>
<thead>
<tr>
<th>Design</th>
<th>Language</th>
<th>Description</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyDecFilter</td>
<td>VHDL</td>
<td>A polyphase decimating filter</td>
<td>Spartan®-6, Virtex®-6, 7 series</td>
</tr>
<tr>
<td>PolyIntrpFilter</td>
<td>VHDL</td>
<td>A polyphase interpolator filter</td>
<td>Spartan-6, Virtex-6, 7 series</td>
</tr>
<tr>
<td>EvenSymSystFIR</td>
<td>VHDL</td>
<td>A symmetric systolic filter with an even number of taps. Symmetric coefficients have been factorized to take advantage of pre-adder capabilities of DSP blocks.</td>
<td>Virtex-6, 7 series</td>
</tr>
<tr>
<td>OddSymSystFIR</td>
<td>VHDL</td>
<td>A symmetric systolic filter with an odd number of taps. Symmetric coefficients have been factorized to take advantage of pre-adder capabilities of DSP blocks.</td>
<td>Virtex-6, 7 series</td>
</tr>
<tr>
<td>EvenSymTranspConvFIR</td>
<td>VHDL</td>
<td>A symmetric transpose convolution filter with an even number of taps. Symmetric coefficients have been factorized to take advantage of pre-adder capabilities of DSP blocks.</td>
<td>Virtex-6, 7 series</td>
</tr>
<tr>
<td>OddSymTranspConvFIR</td>
<td>VHDL</td>
<td>A symmetric transpose convolution filter with an odd number of taps. Symmetric coefficients have been factorized to take advantage of pre-adder capabilities of DSP blocks.</td>
<td>Virtex-6, 7 series</td>
</tr>
<tr>
<td>AlphaBlender</td>
<td>VHDL, Verilog</td>
<td>Implements an alpha blending function, commonly used in image composition, on a single DSP block, taking advantage of the pre-adder, multiplier and post-adder features.</td>
<td>Spartan-6, Virtex-6, 7 series</td>
</tr>
<tr>
<td>ComplexMult</td>
<td>VHDL</td>
<td>A simple way to describe complex multiplier functionality</td>
<td>Spartan-6, Virtex-6, 7 series</td>
</tr>
<tr>
<td>ComplexMultAcc</td>
<td>VHDL</td>
<td>A simple way to describe complex multiply-accumulate functionality</td>
<td>Spartan-6, Virtex-6, 7 series</td>
</tr>
</tbody>
</table>
Resource Sharing

This section discusses Resource Sharing, and includes:

- About Resource Sharing
- Resource Sharing Related Constraints
- Resource Sharing Reporting
- Resource Sharing Coding Examples

About Resource Sharing

XST implements high-level optimizations known as resource sharing. Resource sharing minimizes the number of arithmetic operators, resulting in reduced device utilization. Resource sharing is based on the principle that two similar arithmetic operators can be implemented with common resources on the device, provided their respective outputs are never used simultaneously. Resource sharing usually involves creating additional multiplexing logic to select between factorized inputs. Factorization is performed in a way that minimizes this logic.

XST supports resource sharing for:

- Adders
- Subtractors
- Adders/Subtractors
- Multipliers

Resource sharing is enabled by default, no matter which overall optimization strategy you have selected. If circuit performance is your primary optimization goal, and you are unable to meet timing goals, disabling resource sharing may help. An HDL Advisor message informs you when resource sharing has taken place.

Resource Sharing Related Constraints

Resource Sharing (RESOURCE_SHARING)
Resource Sharing Reporting

Arithmetic resource sharing is performed during HDL Synthesis, and is reflected by arithmetic macro statistics, and by a specific HDL Advisor message, as shown below.

* HDL Synthesis *

Synthesizing Unit <resource_sharing_1>.
Found 8-bit adder for signal <n0017> created at line 18.
Found 8-bit subtractor for signal <n0004> created at line 18.
Found 8-bit 2-to-1 multiplexer for signal <RES> created at line 18.
Summary:
inferred 1 Adder/Subtractor(s).
inferred 1 Multiplexer(s).
Unit <resource_sharing_1> synthesized.

HDL Synthesis Report

Macro Statistics
# Adders/Subtractors : 1
  8-bit add/sub : 1
# Multiplexers : 1
  8-bit 2-to-1 multiplexer : 1

INFO:Xst:1767 - HDL ADVISOR - Resource sharing has identified that some arithmetic operations in this design can share the same physical resources for reduced device utilization.
For improved clock frequency you may try to disable resource sharing.

Resource Sharing Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

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

Resource Sharing Diagram
Resource Sharing VHDL Coding Example

--
-- Resource Sharing
--
-- File: HDL_Coding_Techniques/resource_sharing/resource_sharing_1.vhd

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

entity resource_sharing_1 is
  port(A, B, C : in std_logic_vector(7 downto 0);
      OPER : in std_logic;
      RES : out std_logic_vector(7 downto 0));
end resource_sharing_1;

architecture archi of resource_sharing_1 is
begin
  RES <= A + B when OPER='0' else A - C;
end archi;

Resource Sharing Verilog Coding Example

//--
//-- Resource Sharing
//--
//-- File: HDL_Coding_Techniques/resource_sharing/resource_sharing_1.v
//--
module v_resource_sharing_1 (A, B, C, OPER, RES);
  input [7:0] A, B, C;
  input OPER;
  output [7:0] RES;
  wire [7:0] RES;

  assign RES = !OPER ? A + B : A - C;
endmodule
Chapter 7: XST HDL Coding Techniques

RAMs

This section discusses XST HDL Coding Techniques for RAMs, and includes:

- About RAMs
- Distributed RAMs vs. Block RAMs
- RAMS Supported Features
- RAMs Hardware Description Language (HDL) Coding Guidelines
- Block RAMs Optimization Strategies
- Distributed RAMs Pipelining
- RAMs Related Constraints
- RAMs Reporting
- RAMs Coding Examples

About RAMs

XST features extended RAM inferencing capabilities that can spare you from having to manually instantiate Xilinx® RAM primitives. Those capabilities save time and keep Hardware Description Language (HDL) source code portable and scalable.

Distributed RAMs vs. Block RAMs

RAM resources on Virtex®-6 and Spartan®-6 devices are of two types:

- Distributed RAMs, implemented on properly configured slice logic
- Dedicated block RAM resources

For both types, data is synchronously written into the RAM. The key difference between distributed and block RAMs lies in the way data is read from the RAM:

- Asynchronously in the case of distributed RAM
- Synchronously in the case of block RAM

XST can take advantage of both types of resources. The implementation choice may depend on:

- The exact characteristics of the RAM you have described in the Hardware Description Language (HDL) source code
- Whether you have forced a specific implementation style
- Availability of block RAM resources on the targeted device.

However, considering the key difference mentioned above:

- RAM descriptions with asynchronous read will necessarily be implemented with distributed RAM resources. They cannot be implemented in block RAM.
- RAM descriptions with synchronous read generally go into block RAM. However, if you have so requested, or for device resource utilization considerations, they can also be implemented using distributed RAM plus additional registers. Use RAM Style (RAM_STYLE) to control RAM implementation.

For more information about RAM resources on Virtex-6 and Spartan-6 devices, see:

- Virtex-6 FPGA Memory Resources User Guide
- Distributed RAM topics in the Virtex-6 FPGA Configurable Logic Block User Guide
- Spartan-6 FPGA Block RAM Resources User Guide
- Distributed RAM topics in the Spartan-6 FPGA Configurable Logic Block User Guide
Chapter 7: XST HDL Coding Techniques

RAMs Supported Features

XST RAM inferencing capabilities include:

- Support for any size and data width. XST automatically handles mapping the RAM description to one or several RAM primitives.
- Single-port, simple-dual port, true dual port
- Up to two write ports
- Multiple read ports
  Provided that only one write port is described, XST can identify RAM descriptions with two or more read ports that access the RAM contents at addresses different from the write address.
- Simple-dual port and true dual-port RAM with asymmetric ports.
  For more information, see: Asymmetric Ports Support (Block RAM)
- Write enable
- RAM enable (block RAM)
- Data output reset (block RAM)
- Optional output register (block RAM)
- Byte-Wide Write Enable (block RAM)
- Each RAM port can be controlled by its distinct clock, RAM enable, write enable, and data output reset.
- Initial contents specification

Parity bits are not supported.

XST can use parity bits, available on certain block RAM primitives, as regular data bits, in order to accommodate the described data widths. However, XST does not provide any capability to automatically generate parity control logic, and use those parity bit positions for their intended purpose.

RAMs Hardware Description Language (HDL) Coding Guidelines

This section discusses RAMs Hardware Description Language (HDL) Coding Guidelines, and includes:

- Modeling
- Describing Read Access
- Block RAM Read/Write Synchronization
- Re-Settable Data Outputs (Block RAM)
- Byte-Write Enable Support (Block RAM)
- Asymmetric Ports Support
- RAM Initial Contents

Modeling

RAM is usually modeled with an array of array object.

Modeling a RAM in VHDL

To describe a RAM with a single write port, use a VHDL signal as follows:

type ram_type is array (0 to 255) of std_logic_vector (15 downto 0);
signal RAM : ram_type;

For more information, see: Asymmetric Ports Support (Block RAM)
To describe a RAM with two write ports in VHDL, use a *shared variable* instead of a signal.

```vhdl
type ram_type is array (0 to 255) of std_logic_vector (15 downto 0);
shared variable RAM : ram_type;
```

XST rejects an attempt to use a *signal* to model a RAM with two write ports. Such a model does not behave correctly in simulation tools.

**Note** Shared variables are an extension of variables, allowing inter-process communication. Use them with even greater caution than variables, from which they inherit all basic characteristics. Be aware that:

- The order in which items in a sequential process are described can condition the functionality being modeled.
- Two or more processes making assignments to a shared variable in the same simulation cycle can lead to unpredictable results.

Although shared variables are valid and accepted by XST, Xilinx® does not recommend using a shared variable if the RAM has only one write port. Use a signal instead.

**Modeling a RAM in Verilog**

```verilog
reg [15:0] RAM [0:255];
```

**Describing Write Access**

This section discusses Describing Write Access in VHDL and Verilog.

**Describing Write Access in VHDL**

For a RAM modeled with a VHDL signal, write into the RAM is typically described as follows:

```vhdl
process (clk)
begin
  if rising_edge(clk) then
    if we = '1' then
      RAM(conv_integer(addr)) <= di;
    end if;
  end if;
end process;
```

The address signal is typically declared as follows:

```vhdl
signal addr : std_logic_vector(ADDR_WIDTH-1 downto 0);
```

**Note** In VHDL, you must include `std_logic_unsigned` in order to use the `conv_integer` conversion function. Although `std_logic_signed` also includes a `conv_integer` function, Xilinx® does not recommend using it in this instance. If you do so, XST assumes that address signals have a signed representation, and ignores all negative values. This can result in an inferred RAM of half the desired size. If you need signed data representation in some parts of the design, describe them in units separate from the RAMs.
If the RAM has two write ports and is instead modeled with a VHDL shared variable, a typical write description should look as follows:

```vhdl
process (clk)
begin
  if rising_edge(clk) then
    if we = '1' then
      RAM(conv_integer(addr)) := di;
    end if;
  end if;
end process;
```

**Describing Write Access in Verilog**

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

**Describing Read Access**

This section discusses Describing Read Access in VHDL and in Verilog.

**Describing Read Access in VHDL**

A RAM is typically read-accessed at a given address location as follows:

```vhdl
do <= RAM( conv_integer(addr));
```

Whether this statement is a simple concurrent statement, or is described in a sequential process, determines

- Whether the read is asynchronous or synchronous
- Whether the RAM can be implemented using block RAM resources, or distributed RAM resources on Xilinx® devices.

Following is a coding example for a RAM to be implemented on block resources:

```vhdl
process (clk)
begin
  do <= RAM( conv_integer(addr));
end process;
```

For more information, see Block RAM Read/Write Synchronization below.

**Describing Read Access in Verilog**

An *asynchronous* read is described with an `assign` statement:

```verilog
assign do = RAM[addr];
```

A *synchronous* read is described with a sequential `always` block:

```verilog
always @ (posedge clk)
begin
  do <= RAM[addr];
end
```

For more information, see:

Block RAM Read/Write Synchronization
Block RAM Read/Write Synchronization

Block RAM resources can be configured to provide the following synchronization modes for a given read-and-write port:

- **Read-first**
  Old contents are read before new contents are loaded.
- **Write-first (also known as read-through):**
  New contents are immediately made available for reading.
- **No-change**
  Data output does not change while new contents are loaded into RAM.

XST provides inference support for all of these synchronization modes. You can describe a different synchronization mode for each port of the RAM.

**VHDL Block RAM Read/Write Synchronization Coding Example One**

```vhdl
process (clk)
begin
  if (clk'event and clk = '1') then
    if (we = '1') then
      RAM(conv_integer(addr)) <= di;
    end if;
    do <= RAM(conv_integer(addr));
  end if;
end process;
```

**VHDL Block RAM Read/Write Synchronization Coding Example Two**

The following VHDL coding example describes a write-first synchronized port.

```vhdl
process (clk)
begin
  if (clk'event and clk = '1') then
    if (we = '1') then
      RAM(conv_integer(addr)) <= di;
      do <= di;
    else
      do <= RAM(conv_integer(addr));
    end if;
  end if;
end process;
```

**VHDL Block RAM Read/Write Synchronization Coding Example Three**

The following VHDL coding example describes a no-change synchronization.

```vhdl
process (clk)
begin
  if (clk'event and clk = '1') then
    if (we = '1') then
      RAM(conv_integer(addr)) <= di;
    else
      do <= RAM(conv_integer(addr));
    end if;
  end if;
end process;
```
VHDL Block RAM Read/Write Synchronization Coding Example Four

**Caution!** If you model a dual-write RAM with a VHDL shared variable, be aware that the synchronization described below is not read-first, but write-first.

```vhdl
process (clk)
begin
  if (clk’event and clk = ’1’) then
    if (we = ’1’) then
      RAM(conv_integer(addr)) := di;
      end if;
    do <= RAM(conv_integer(addr));
  end if;
end process;
```

VHDL Block RAM Read/Write Synchronization Coding Example Five

To describe a read-first synchronization, reorder the process body.

```vhdl
process (clk)
begin
  if (clk’event and clk = ’1’) then
    do <= RAM(conv_integer(addr));
    if (we = ’1’) then
      RAM(conv_integer(addr)) := di;
      end if;
  end if;
end process;
```

Re-Settable Data Outputs (Block RAM)

Optionally, you can describe a reset to any constant value of synchronously read data. XST recognizes it and takes advantage of the synchronous set/reset feature of block RAMs. For a RAM port with read-first synchronization, describe the reset functionality as follows.

```vhdl
process (clk)
begin
  if clk’event and clk = ’1’ then
    if en = ’1’ then -- optional RAM enable
      if we = ’1’ then -- write enable
        ram(conv_integer(addr)) <= di;
      end if;
      if rst = ’1’ then -- optional dataout reset
        do <= "00011101";
      else
        do <= ram(conv_integer(addr));
      end if;
    end if;
  end if;
end process;
```

Byte-Write Enable Support (Block RAM)

The byte-wide write enable feature available with block RAM resources offers advanced control for writing data into RAM. It allows you to separately control which portions of 8 bits of an addressed memory location can be written to.

From an HDL modeling and inference standpoint, the concept can be further described as a column-based write. The RAM is seen as a collection of equal size columns. During a write cycle, you separately control writing into each of these columns.
XST provides inference capabilities that allow you to take advantage of the block RAM byte write enable feature. XST now supports two description styles:

- Single-process description style
  Xilinx® recommends that you use the single-process description style in all cases.
- Two-process description style
  The two-process description style continues to be supported, but is no longer the recommended way to describe byte-write enable functionality.

**Single-Process Description Style**

The single-process description style is more intuitive and less error-prone than the two-process description style. The described RAM is implemented on block RAM resources, using the byte-write enable capability, provided that the following requirements are met.

- Write columns of equal widths
- Allowed write column widths: 8-bit, 9-bit, 16-bit, 18-bit
  For other write column widths, such as 5-bit or 12-bit, XST uses distributed RAM resources and creates additional multiplexing logic on the data input.
- Number of write columns: any
- RAM depth: any
  XST implements the RAM using one or several block RAM primitives as needed.
- Supported read-write synchronizations: read-first, write-first, no-change
Single-Process Description Style VHDL Coding Example

The following recommended VHDL coding example uses generics and a for-loop construct for a compact and easily changeable configuration of the desired number and width of write columns.

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

entity bytewrite_ram_1b is
  generic (  
    SIZE  : integer := 1024;
    ADDR_WIDTH : integer := 10;
    COL_WIDTH : integer := 8;
    NB_COL  : integer := 2);
  port (  
    clk : in std_logic;
    we : in std_logic_vector(NB_COL-1 downto 0);
    addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    di : in std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    do : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));
end bytewrite_ram_1b;

architecture behavioral of bytewrite_ram_1b is
  type ram_type is array (SIZE-1 downto 0)  
    of std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

begin
  process (clk)
  begin
    if rising_edge(clk) then  
      do <= RAM(conv_integer(addr));
      for i in 0 to NB_COL-1 loop
        if we(i) = '1' then
          RAM(conv_integer(addr))((i+1)*COL_WIDTH-1 downto i*COL_WIDTH) <= di((i+1)*COL_WIDTH-1 downto i*COL_WIDTH);
        end if;
      end loop;
    end if;
  end process;
end behavioral;
```
Single-Process Description Style Verilog Coding Example

The following recommended Verilog coding example uses parameters and a generate-for construct.

```verilog

parameter SIZE = 1024;
parameter ADDR_WIDTH = 10;
parameter COL_WIDTH = 9;
parameter NB_COL = 4;

input clk;
input [NB_COL-1:0] we;
input [ADDR_WIDTH-1:0] addr;
output [NB_COL*COL_WIDTH-1:0] di;
output reg [NB_COL*COL_WIDTH-1:0] do;

reg [NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];

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

generate

genvar i;
for (i = 0; i < NB_COL; i = i+1)
begin
always @(posedge clk)
begin
if (we[i])
RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH] <= di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];
end
end
endgenerate

endmodule
```
Single-Process Description Style for No-Change VHDL Coding Example

The single-process description style is the only way to correctly model byte-write enable functionality in conjunction with no-change read-write synchronization. This is typically done as follows:

```
-- -- Single-Port BRAM with Byte-wide Write Enable
-- 2x8-bit write
-- No-Change mode
-- Single-process description
-- Compact description of the write with a for-loop statement
-- Column width and number of columns easily configurable
--
-- File: HDL_Coding_Techniques/rams/bytewrite_nochange.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity bytewrite_nochange is

  generic (  
    SIZE : integer := 1024;
    ADDR_WIDTH : integer := 10;
    COL_WIDTH : integer := 8;
    NB_COL : integer := 2);

  port (     
    clk : in std_logic;
    we : in std_logic_vector(NB_COL-1 downto 0);
    addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    di : in std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    do : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));

end bytewrite_nochange;

architecture behavioral of bytewrite_nochange is

  type ram_type is array (SIZE-1 downto 0) of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

begin

  process (clk)
  begin
    if rising_edge(clk) then
      if (we = (we'range => '0')) then
        do <= RAM(conv_integer(addr));
        end if;
        for i in 0 to NB_COL-1 loop
          if we(i) = '1' then
            RAM(conv_integer(addr))((i+1)*COL_WIDTH-1 downto i*COL_WIDTH) <= di((i+1)*COL_WIDTH-1 downto i*COL_WIDTH);
            end if;
            end loop;
            end if;
            end process;
          end behavioral;
```
Single-Process Description Style for No-Change Verilog Coding Example

//
// Single-Port BRAM with Byte-wide Write Enable
// 4x9-bit write
// No-Change mode
// Single-process description
// Compact description of the write with a generate-for statement
// Column width and number of columns easily configurable
//
// File: HDL_Coding_Techniques/rams/bytewrite_nochange.v
//
module v_bytewrite_nochange (clk, we, addr, di, do);

  parameter SIZE = 1024;
  parameter ADDR_WIDTH = 10;
  parameter COL_WIDTH = 9;
  parameter NB_COL = 4;

  input  clk;
  input [NB_COL-1:0] we;
  input [ADDR_WIDTH-1:0] addr;
  input [NB_COL*COL_WIDTH-1:0] di;
  output reg [NB_COL*COL_WIDTH-1:0] do;

  reg [NB_COL*COL_WIDTH-1:0] RAM [SIZE-1:0];

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

  generate
  genvar i;
  for (i = 0; i < NB_COL; i = i+1)
  begin
    always @(posedge clk)
    begin
      if (we[i])
        RAM[addr][(i+1)*COL_WIDTH-1:i*COL_WIDTH] <= di[(i+1)*COL_WIDTH-1:i*COL_WIDTH];
    end
  end
generate

endmodule

Two-Process Description Style

Caution! In order to take advantage of block RAM byte-write enable capabilities, you must provide adequate data read synchronization. If you do not do so, XST implements the described functionality sub-optimally, using distributed RAM resources instead.

The two-process description style does not allow you to properly describe byte-write enable functionality in conjunction with the no-change synchronization mode.

- If you are currently using the two-process description style, Xilinx recommends that you change your HDL code to the single-process description style.
- For new designs, Xilinx recommends that you do not use the two-process description style at all.
However, if you are unable to migrate your code to that style, XST still supports the two-process description style. With the two-process description style:

- A combinatorial process describes which data is loaded and read for each byte. In particular, the write enable functionality is described there, and not in the main sequential process.
- A sequential process describes the write and read synchronization.
- Data widths are more restrictive than with the single-process method:
  - Number of write columns: 2 or 4
  - Write column widths: 8-bit or 9-bit
  - Supported data widths: 2x8-bit (two columns of 8 bits each), 2x9-bit, 4x8-bit, 4x9-bit
Two-Process Description Style VHDL Coding Example

--
-- Single-Port BRAM with Byte-wide Write Enable
-- 2x8-bit write
-- Read-First Mode
-- Two-process description
--
-- File: HDL_Coding_Techniques/rams/rams_24.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_24 is
  generic (
    SIZE : integer := 512;
    ADDR_WIDTH : integer := 9;
    COL_WIDTH : integer := 16;
    NB_COL : integer := 2);
  port (
    clk : in std_logic;
    we : in std_logic_vector(NB_COL-1 downto 0);
    addr : in std_logic_vector(ADDR_WIDTH-1 downto 0);
    di : in std_logic_vector(NB_COL*COL_WIDTH-1 downto 0);
    do : out std_logic_vector(NB_COL*COL_WIDTH-1 downto 0));
end rams_24;

architecture syn of rams_24 is
  type ram_type is array (SIZE-1 downto 0) of std_logic_vector (NB_COL*COL_WIDTH-1 downto 0);
  signal RAM : ram_type := (others => (others => '0'));

  signal di0, di1 : std_logic_vector (COL_WIDTH-1 downto 0);
begin
  process(we, di)
  begin
    if we(1) = '1' then
      di1 <= di(2*COL_WIDTH-1 downto 1*COL_WIDTH);
    else
      di1 <= RAM(conv_integer(addr))(2*COL_WIDTH-1 downto 1*COL_WIDTH);
    end if;
    if we(0) = '1' then
      di0 <= di(COL_WIDTH-1 downto 0);
    else
      di0 <= RAM(conv_integer(addr))(COL_WIDTH-1 downto 0);
    end if;
  end process;
  process(clk)
  begin
    if (clk'event and clk = '1') then
      do <= RAM(conv_integer(addr));
      RAM(conv_integer(addr)) <= di1 & di0;
    end if;
  end process;
end syn;
**Two-Process Description Style Verilog Coding Example**

```verilog
//
// Single-Port BRAM with Byte-wide Write Enable (2 bytes) in Read-First Mode
//
// File: HDL_Coding_Techniques/rams/rams_24.v
//
module v_rams_24 (clk, we, addr, di, do);

parameter SIZE = 512;
parameter ADDR_WIDTH = 9;
parameter DI_WIDTH = 8;

input clk;
input [1:0] we;
input [ADDR_WIDTH-1:0] addr;
input [2*DI_WIDTH-1:0] di;
output [2*DI_WIDTH-1:0] do;

reg [2*DI_WIDTH-1:0] RAM [ADDR_WIDTH-1:0];
reg [2*DI_WIDTH-1:0] do;
reg [DI_WIDTH-1:0] di0, di1;

always @(we or di)
begin
  if (we[1])
    di1 = di[2*DI_WIDTH-1:1*DI_WIDTH];
  else
    di1 = RAM[addr][2*DI_WIDTH-1:1*DI_WIDTH];

  if (we[0])
    di0 = di[DI_WIDTH-1:0];
  else
    di0 = RAM[addr][DI_WIDTH-1:0];
end

always @(posedge clk)
begin
  do <= RAM[addr];
  RAM[addr]<=[di1,di0];
end

endmodule
```

**Asymmetric Ports Support (Block RAM)**

Xilinx® Block RAM resources can be configured with two asymmetric ports. One port accesses the physical memory with a given data width. The other port accesses the same physical memory, but with a different data width. Both ports have physical access to the same memory resources, but see a different logical organization of the RAM. For example, the same 2048 bits of physical memory may be seen as:

- 256x8-bit by port A
- 64x32-bit by port B

Such an asymmetrically configured block RAM is also said to have ports with different aspect ratios.

A typical use of port asymmetry is to create storage and buffering between two flows of data with different data width characteristics, and operating at asymmetric speeds.

**Modeling**

Like RAMs with no port asymmetry, block RAMs with asymmetric ports are modeled with a single array of array object. The key concept is that the depth and width characteristics of the modeling signal or shared variable, match the RAM port with the lower data width (subsequently the larger depth).
As a result of this modeling requirement, describing a read or write access for the port with the larger data width no longer implies one assignment, but several assignments instead. The number of assignments equals the ratio between the two asymmetric data widths.

Each of these assignments may be explicitly described as illustrated in the following coding examples.
Asymmetric Port RAM VHDL Coding Example

--
-- Asymmetric port RAM
-- Port A is 256x8-bit write-only
-- Port B is 64x32-bit read-only
--
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_1a.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_1a is
  generic (
    WIDTHA   : integer := 8;
    SIZEA    : integer := 256;
    ADDRWIDTHA: integer := 8;
    WIDTHB   : integer := 32;
    SIZEB    : integer := 64;
    ADDRWIDTHB: integer := 6
  );
  port (    clkA : in std_logic;
            clkB : in std_logic;
            weA : in std_logic;
            enA : in std_logic;
            enB : in std_logic;
            addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
            addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
            diA   : in std_logic_vector(WIDTHA-1 downto 0);
            doB   : out std_logic_vector(WIDTHB-1 downto 0)
  );
end asymmetric_ram_1a;

architecture behavioral of asymmetric_ram_1a is

  function max(L, R: INTEGER) return INTEGER is
  begin
    if L > R then
      return L;
    else
      return R;
    end if;
  end;

  function min(L, R: INTEGER) return INTEGER is
  begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

  constant minWIDTH : integer := min(WIDTHA,WIDTHB);
  constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
  constant maxSIZE  : integer := max(SIZEA,SIZEB);
  constant RATIO    : integer := maxWIDTH / minWIDTH;

  type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
  signal ram : ramType := (others => (others => '0'));
  signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
  signal regB  : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

Chapter 7: XST HDL Coding Techniques

process (clkA)
begin
  if rising_edge(clkA) then
    if enA = '1' then
      if weA = '1' then
        ram(conv_integer(addrA)) <= diA;
      end if;
    end if;
  end if;
end process;

process (clkB)
begin
  if rising_edge(clkB) then
    if enB = '1' then
      readB(minWIDTH-1 downto 0) <=
      ram(conv_integer(addrB&conv_std_logic_vector(0,2))); 
      readB(2*minWIDTH-1 downto minWIDTH) <=
      ram(conv_integer(addrB&conv_std_logic_vector(1,2))); 
      readB(3*minWIDTH-1 downto 2*minWIDTH) <=
      ram(conv_integer(addrB&conv_std_logic_vector(2,2))); 
      readB(4*minWIDTH-1 downto 3*minWIDTH) <=
      ram(conv_integer(addrB&conv_std_logic_vector(3,2)));
      end if;
      regB <= readB;
  end if;
end process;
doB <= regB;
end behavioral;
Asymmetric Port RAM Verilog Coding Example

//
// Asymmetric port RAM
// Port A is 256x8-bit write-only
// Port B is 64x32-bit read-only
//
// File: HDL_Coding_Techniques/rams/asymmetric_ram_1a.v
//
module v_asymmetric_ram_1a (clkA, clkB, weA, reB, addrA, addrB, diA, doB);

parameter WIDTHA = 8;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
parameter WIDTHB = 32;
parameter SIZEB = 64;
parameter ADDRWIDTHB = 6;

input clkA;
input clkB;
input weA;
input reB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output reg [WIDTHB-1:0] doB;

'define max(a,b) { (a) > (b) ? (a) : (b) }
'define min(a,b) { (a) < (b) ? (a) : (b) }

localparam maxSIZE = 'max(SIZEA, SIZEB);
localparam maxWIDTH = 'max(WIDTHA, WIDTHB);
localparam minWIDTH = 'min(WIDTHA, WIDTHB);
localparam RATIO = maxWIDTH / minWIDTH;

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

always @(posedge clkA)
begin
  if (weA)
    RAM[addrA] <= diA;
end

always @(posedge clkB)
begin
  if (reB)
  begin
    doB <= readB;
    readB[4*minWIDTH-1:3*minWIDTH] <= RAM[addrB, 2'd3];
    readB[3*minWIDTH-1:2*minWIDTH] <= RAM[addrB, 2'd2];
    readB[2*minWIDTH-1:minWIDTH]    <= RAM[addrB, 2'd1];
    readB[minWIDTH-1:0]             <= RAM[addrB, 2'd0];
  end
end
endmodule
Chapter 7: XST HDL Coding Techniques

### VHDL Coding Example Using For-Loop Statement

To make your VHDL code more compact, easier to maintain, and easier to scale, use a for-loop statement as shown in the following coding example.

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

entity asymmetric_ram_1b is
  generic
    WIDTHA : integer := 8;
    SIZEA  : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB  : integer := 64;
    ADDRWIDTHB : integer := 6;
  port
    clkA : in std_logic;
    clkB : in std_logic;
    weA : in std_logic;
    enA : in std_logic;
    enB : in std_logic;
    addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
    diA  : in std_logic_vector(WIDTHA-1 downto 0);
    doB  : out std_logic_vector(WIDTHB-1 downto 0);
end asymmetric_ram_1b;

architecture behavioral of asymmetric_ram_1b is

  function max(L, R: INTEGER) return INTEGER is
    begin
      if L > R then
        return L;
      else
        return R;
      end if;
    end;

  function min(L, R: INTEGER) return INTEGER is
    begin
      if L < R then
        return L;
      else
        return R;
      end if;
    end;

  function log2(val: INTEGER) return natural is
    variable res : natural;
    begin
      for i in 0 to 31 loop
        if (val <= (2**i)) then
          res := i;
          exit;
        end if;
      end loop;
      return res;
    end;

end asymmetric_ram_1b;
```

-- Asymmetric port RAM
-- Port A is 256x8-bit write-only
-- Port B is 64x32-bit read-only
-- Compact description with a for-loop statement

-- File: HDL_Coding_Techniques/rams/asymmetric_ram_1b.vhd

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

entity asymmetric_ram_1b is
  generic
    WIDTHA : integer := 8;
    SIZEA  : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB  : integer := 64;
    ADDRWIDTHB : integer := 6;
  port
    clkA : in std_logic;
    clkB : in std_logic;
    weA : in std_logic;
    enA : in std_logic;
    enB : in std_logic;
    addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
    diA  : in std_logic_vector(WIDTHA-1 downto 0);
    doB  : out std_logic_vector(WIDTHB-1 downto 0);
end asymmetric_ram_1b;

architecture behavioral of asymmetric_ram_1b is

  function max(L, R: INTEGER) return INTEGER is
    begin
      if L > R then
        return L;
      else
        return R;
      end if;
    end;

  function min(L, R: INTEGER) return INTEGER is
    begin
      if L < R then
        return L;
      else
        return R;
      end if;
    end;

  function log2(val: INTEGER) return natural is
    variable res : natural;
    begin
      for i in 0 to 31 loop
        if (val <= (2**i)) then
          res := i;
          exit;
        end if;
      end loop;
      return res;
    end;

end asymmetric_ram_1b;
```
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSIZE : integer := max(SIZEA,SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
signal ram : ramType := (others => (others => '0'));

signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

process (clkA)
begin
  if rising_edge(clkA) then
    if enA = '1' then
      if weA = '1' then
        ram(conv_integer(addrA)) <= diA;
      end if;
    end if;
  end if;
end process;

process (clkB)
begin
  if rising_edge(clkB) then
    if enB = '1' then
      for i in 0 to RATIO-1 loop
        readB((i+1)*minWIDTH-1 downto i*minWIDTH) <=
        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
      end loop;
    end if;
    regB <= readB;
  end if;
end process;

doB <= regB;

end behavioral;
Verilog Coding Example Using Parameters and Generate-For Statement

To make your Verilog code more compact and easier to modify, use parameters and a generate-for statement as shown in the following coding example.

```verilog
module v_asymmetric_ram_1b (clkA, clkB, weA, reB, addrA, addrB, diA, doB);

parameter WIDTHA = 8;
parameter SIZEA = 256;
parameter ADDRWIDTHA = 8;
parameter WIDTHB = 32;
parameter SIZEB = 64;
parameter ADDRWIDTHB = 6;

input clkA;
input clkB;
input weA;
input reB;
input [ADDRWIDTHA-1:0] addrA;
input [ADDRWIDTHB-1:0] addrB;
input [WIDTHA-1:0] diA;
output reg [WIDTHB-1:0] doB;

'define max(a,b) { (a) > (b) ? (a) : (b) }
'define min(a,b) { (a) < (b) ? (a) : (b) }

function integer log2;
    input integer value;
    reg [31:0] shifted;
    integer res;
    begin
        if (value < 2)
            log2 = value;
        else
            begin
                shifted = value-1;
                for (res=0; shifted>0; res=res+1)
                    shifted = shifted>>1;
                log2 = res;
            end
    end
endfunction

localparam maxSIZE = 'max(SIZEA, SIZEB);
localparam maxWIDTH = 'max(WIDTHA, WIDTHB);
localparam minWIDTH = 'min(WIDTHA, WIDTHB);
localparam RATIO = maxWIDTH / minWIDTH;
localparam log2RATIO = log2(RATIO);

reg [minWIDTH-1:0] RAM [0:maxSIZE-1];
reg [WIDTHB-1:0] readB;

genvar i;
always @ (posedge clkA)
    begin
        if (weA)
            RAM[addrA] <= diA;
    end

always @ (posedge clkB)
    begin
        if (reB)
            doB <= readB;
```
Note  These coding examples use min, max, and log2 functions to make the code as generic and clean as possible. Those functions can be defined anywhere in the design, typically in a package.
Chapter 7: XST HDL Coding Techniques

Shared Variable ( VHDL )

When you describe a symmetric port RAM in VHDL, a shared variable is required only if you describe two ports writing into the RAM. Otherwise, a signal is preferred.

When you describe an asymmetric port RAM in VHDL, a shared variable may be required even if only one write port is described. If the write port has the larger data width, several write assignments are needed to describe it, and a shared variable is therefore required as shown in the following coding example.

Shared Variable Required VHDL Coding Example

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

entity asymmetric_ram_4 is
  generic(
    WIDTHA : integer := 8;
    SIZEA : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB : integer := 64;
    ADDRWIDTHB : integer := 6
  );
  port (  
    clkA : in std_logic;
    clkB : in std_logic;
    reA : in std_logic;
    weB : in std_logic;
    addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
    diB : in std_logic_vector(WIDTHB-1 downto 0);
    doA : out std_logic_vector(WIDTHA-1 downto 0)
  );
end asymmetric_ram_4;

architecture behavioral of asymmetric_ram_4 is
func max(L, R: INTEGER) return INTEGER is
begin
  if L > R then
    return L;
  else
    return R;
  end if;
end;

func min(L, R: INTEGER) return INTEGER is
begin
  if L < R then
    return L;
  else
    return R;
  end if;
end;
```
function log2 (val: INTEGER) return natural is
  variable res : natural;
begin
  for i in 0 to 31 loop
    if (val <= (2**i)) then
      res := i;
      exit;
    end if;
  end loop;
  return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
custom maxWIDTH : integer := max(WIDTHA,WIDTHB);
custom maxSIZE : integer := max(SIZEA,SIZEB);
custom RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');

begin
  process (clkA)
  begin
    if rising_edge(clkA) then
      if reA = '1' then
        readA <= ram(conv_integer(addrA));
      end if;
      regA <= readA;
    end if;
  end process;

  process (clkB)
  begin
    if rising_edge(clkB) then
      if weB = '1' then
        for i in 0 to RATIO-1 loop
          ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO)))) := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
        end loop;
      end if;
    end if;
  end process;

doA <= regA;
end behavioral;

Caution! Shared variables are an extension of variables, from which they inherit all basic characteristics, allowing inter-process communication. Use them with great caution.

- The order in which items in a sequential process are described can condition the functionality being modeled.
- Two or more processes making assignments to a shared variable in the same simulation cycle can lead to unpredictable results.
Read-Write Synchronization

Read-Write synchronization is controlled in a similar manner, whether describing a symmetric or asymmetric RAM. The following coding examples describe a RAM with two asymmetric read-write ports, and illustrate how to respectively model write-first, read-first, and no-change synchronization.

Asymmetric Port RAM (Write-First) VHDL Coding Example

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

entity asymmetric_ram_2b is
  generic (
    WIDTHA : integer := 8;
    SIZEA : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB : integer := 64;
    ADDRWIDTHB : integer := 6
  );
  port ( clkA : in std_logic;
         clkB : in std_logic;
         enA : in std_logic;
         enB : in std_logic;
         weA : in std_logic;
         weB : in std_logic;
         addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
         addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
         diA : in std_logic_vector(WIDTHA-1 downto 0);
         diB : in std_logic_vector(WIDTHB-1 downto 0);
         doA : out std_logic_vector(WIDTHA-1 downto 0);
         doB : out std_logic_vector(WIDTHB-1 downto 0)
  );
end asymmetric_ram_2b;

architecture behavioral of asymmetric_ram_2b is

  function max(L, R: INTEGER) return INTEGER is
    begin
    if L > R then
      return R;
    else
      return L;
    end if;
  end;

  function min(L, R: INTEGER) return INTEGER is
    begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

```
function log2 (val: INTEGER) return natural is  
  variable res : natural;
begin  
    for i in 0 to 31 loop  
      if (val <= (2**i)) then  
        res := i;  
        exit;  
      end if;  
    end loop;  
  return res;  
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSIZE : integer := max(SIZEA,SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

process (clkA)
begin  
  if rising_edge(clkA) then  
    if enA = '1' then  
      if weA = '1' then  
        ram(conv_integer(addrA)) := diA;
        end if;  
        readA <= ram(conv_integer(addrA));  
        end if;
        regA <= readA;
        end if;
      end if;
    end process;

process (clkB)
begin  
  if rising_edge(clkB) then  
    if enB = '1' then  
      if weB = '1' then  
        for i in 0 to RATIO-1 loop  
          ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO)))) := diB((i+1)*minWIDTH-1 downto i*minWIDTH);  
          end loop;
        end if;
        for i in 0 to RATIO-1 loop  
          readB((i+1)*minWIDTH-1 downto i*minWIDTH) <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));  
          end loop;
        end if;
        regB <= readB;
        end if;
      end if;
    end process;

  doA <= regA;
  doB <= regB;
end behavioral;

Asymmetric Port RAM (Read-First) VHDL Coding Example

--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-and-write (read-first synchronization)
-- Port B is 64x32-bit read-and-write (read-first synchronization)
-- Compact description with a for-loop statement
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_2c.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_2c is
  generic (
    WIDTHA : integer := 8;
    SIZEA : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB : integer := 64;
    ADDRWIDTHB : integer := 6
  );
  port ( 
    clkA : in std_logic;
    clkB : in std_logic;
    enA  : in std_logic;
    enB  : in std_logic;
    weA : in std_logic;
    weB : in std_logic;
    addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
    diA : in std_logic_vector(WIDTHA-1 downto 0);
    diB : in std_logic_vector(WIDTHB-1 downto 0);
    doA : out std_logic_vector(WIDTHA-1 downto 0);
    doB : out std_logic_vector(WIDTHB-1 downto 0)
  );
end asymmetric_ram_2c;

architecture behavioral of asymmetric_ram_2c is

function max(L, R: INTEGER) return INTEGER is
begin
  if L > R then
    return L;
  else
    return R;
  end if;
end;

function min(L, R: INTEGER) return INTEGER is
begin
  if L < R then
    return L;
  else
    return R;
  end if;
end;

function log2 (val: INTEGER) return natural is
begin
  for i in 0 to 31 loop
    if (val <= (2**i)) then
      res := i;
      exit;
    end if;
  end loop;
end;

end;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSIZE : integer := max(SIZEA,SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

process (clkA)
begin
if rising_edge(clkA) then
  if enA = '1' then
    readA <= ram(conv_integer(addrA));
    if weA = '1' then
      ram(conv_integer(addrA)) := diA;
    end if;
  end if;
  regA <= readA;
end if;
end process;

process (clkB)
begin
if rising_edge(clkB) then
  if enB = '1' then
    for i in 0 to RATIO-1 loop
      readB((i+1)*minWIDTH-1 downto i*minWIDTH) <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
    end loop;
    if weB = '1' then
      for i in 0 to RATIO-1 loop
        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO)))) := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
      end loop;
    end if;
  end if;
  regB <= readB;
end if;
end process;
doA <= regA;
doB <= regB;
end behavioral;
Asymmetric Port RAM (No-Change) VHDL Coding Example

--
-- Asymmetric port RAM
-- Port A is 256x8-bit read-and-write (no-change synchronization)
-- Port B is 64x32-bit read-and-write (no-change synchronization)
-- Compact description with a for-loop statement
-- File: HDL_Coding_Techniques/rams/asymmetric_ram_2d.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity asymmetric_ram_2d is
  generic (
    WIDTHA : integer := 8;
    SIZEA : integer := 256;
    ADDRWIDTHA : integer := 8;
    WIDTHB : integer := 32;
    SIZEB : integer := 64;
    ADDRWIDTHB : integer := 6
  );
  port ( 
    clkA : in std_logic;
    clkB : in std_logic;
    enA : in std_logic;
    enB : in std_logic;
    weA : in std_logic;
    weB : in std_logic;
    addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
    addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
    diA : in std_logic_vector(WIDTHA-1 downto 0);
    diB : in std_logic_vector(WIDTHB-1 downto 0);
    doA : out std_logic_vector(WIDTHA-1 downto 0);
    doB : out std_logic_vector(WIDTHB-1 downto 0)
  );
end asymmetric_ram_2d;

architecture behavioral of asymmetric_ram_2d is
  function max(L, R: INTEGER) return INTEGER is
    begin
      if L > R then
        return L;
      else
        return R;
      end if;
    end;

  function min(L, R: INTEGER) return INTEGER is
    begin
      if L < R then
        return L;
      else
        return R;
      end if;
    end;

  function log2 (val: INTEGER) return natural is
    variable res : natural;
    begin
      for i in 0 to 31 loop
        if (val <= (2**i)) then
          res := i;
          exit;
        end if;
      end loop;
    end log2;
end behavioral;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSize : integer := max(SIZEA,SIZEB);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSize-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));

signal readA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regA : std_logic_vector(WIDTHA-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin
process (clkA)
begin
if rising_edge(clkA) then
  if enA = '1' then
    if weA = '1' then
      ram(conv_integer(addrA)) := diA;
    else
      readA <= ram(conv_integer(addrA));
    end if;
  end if;
end if;
regA <= readA;
end if;
end process;

process (clkB)
begin
if rising_edge(clkB) then
  if enB = '1' then
    for i in 0 to RATIO-1 loop
      if weB = '1' then
        ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO)))) := diB((i+1)*minWIDTH-1 downto i*minWIDTH);
      else
        readB((i+1)*minWIDTH-1 downto i*minWIDTH) <= ram(conv_integer(addrB & conv_std_logic_vector(i,log2(RATIO))));
      end if;
    end loop;
  end if;
regB <= readB;
end if;
end process;
doA <= regA;
doB <= regB;
end behavioral;
Parity Bits

For asymmetric port RAMs, XST can take advantage of the available block RAM parity bits to implement extra data bits for word sizes of 9, 18 and 36 bits, as shown in the following coding example.

Asymmetric Port RAM (Parity Bits) VHDL Coding Example

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

entity asymmetric_ram_3 is
  generic (
    WIDTHA : integer := 18;
    SIZEA : integer := 2048;
    ADDRWIDTHA : integer := 11;
    WIDTHB : integer := 9;
    SIZEB : integer := 4096;
    ADDRWIDTHB : integer := 12
  );
  port (clkA : in std_logic;
        clkB : in std_logic;
        weA : in std_logic;
        reB : in std_logic;
        addrA : in std_logic_vector(ADDRWIDTHA-1 downto 0);
        addrB : in std_logic_vector(ADDRWIDTHB-1 downto 0);
        diA : in std_logic_vector(WIDTHA-1 downto 0);
        doB : out std_logic_vector(WIDTHB-1 downto 0)
  );
end asymmetric_ram_3;

architecture behavioral of asymmetric_ram_3 is

  function max(L, R: INTEGER) return INTEGER is
  begin
    if L > R then
      return L;
    else
      return R;
    end if;
  end;

  function min(L, R: INTEGER) return INTEGER is
  begin
    if L < R then
      return L;
    else
      return R;
    end if;
  end;

  function log2(val: INTEGER) return natural is
    variable res : natural;
  begin
    for i in 0 to 31 loop
      if (val <= (2**i)) then
        res := i;
        return;
      end if;
    end loop;
  end;
```

---

*XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*

www.xilinx.com

UG687 (v 13.1) March 1, 2011
res := i;
exit;
end if;
end loop;
return res;
end function Log2;

constant minWIDTH : integer := min(WIDTHA,WIDTHB);
constant maxWIDTH : integer := max(WIDTHA,WIDTHB);
constant maxSIZE : integer := max(SIZEA,SIZER);
constant RATIO : integer := maxWIDTH / minWIDTH;

type ramType is array (0 to maxSIZE-1) of std_logic_vector(minWIDTH-1 downto 0);
shared variable ram : ramType := (others => (others => '0'));
signal readB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');
signal regB : std_logic_vector(WIDTHB-1 downto 0) := (others => '0');

begin

process (clkA)
begin
  if rising_edge(clkA) then
    if weA = '1' then
      for i in 0 to RATIO-1 loop
        ram(conv_integer(addrA & conv_std_logic_vector(i,log2(RATIO)))) := diA((i+1)*minWIDTH-1 downto i*minWIDTH);
      end loop;
    end if;
  end if;
end process;

process (clkB)
begin
  if rising_edge(clkB) then
    regB <= readB;
    if reB = '1' then
      readB <= ram(conv_integer(addrB));
    end if;
  end if;
end process;

doB <= regB;

end behavioral;
Limitations

Follow these guidelines to ensure that the synthesized solution is implemented optimally on dedicated block RAM resources.

- Support for port asymmetry is available only if the described RAM can be implemented on block RAM resources. Be sure to provide adequate data read synchronization.

- Port asymmetry is supported only if the described RAM fits in a single block RAM primitive.

If the described asymmetric port RAM does not fit in a single block RAM primitive, you must manually instantiate the desired device primitives.

If XST cannot use asymmetrically-configured block RAM resources, the described RAM is implemented on LUT resources, giving suboptimal results and a significant increase in runtime.

Attention The amount of memory accessible from both ports must match exactly. For example, do not try to describe a port seeing the RAM as a 256x8-bit (2048 bits of memory), while the other port sees it as a 64x12-bit (768 bits of memory).

The ratio between both data widths should be a power of two. The ratio between both port depths should also be a power of two.
Synthesizing Unit <asymmetric_ram_1a>.
Found 256x8:64x32-bit dual-port RAM <Mram_ram> for signal <ram>.
Found 32-bit register for signal <doB>.

Summary:
- inferred 1 RAM(s).
- inferred 64 D-type flip-flop(s).

Unit <asymmetric_ram_1a> synthesized.

HDL Synthesis Report

Macro Statistics

<table>
<thead>
<tr>
<th># RAMs</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>256x8:64x32-bit dual-port RAM</td>
<td>1</td>
</tr>
<tr>
<td># Registers</td>
<td>2</td>
</tr>
<tr>
<td>32-bit register</td>
<td>2</td>
</tr>
</tbody>
</table>

Advanced HDL Synthesis Report

INFO:Xst - The RAM <Mram_ram> will be implemented as a BLOCK RAM, absorbing the following register(s): <readB> <doB>

<table>
<thead>
<tr>
<th>ram_type</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port A aspect ratio</td>
<td>256-word x 8-bit</td>
</tr>
<tr>
<td>mode</td>
<td>read-first</td>
</tr>
<tr>
<td>clkA</td>
<td>connected to signal &lt;clkA&gt;</td>
</tr>
<tr>
<td>weA</td>
<td>connected to signal &lt;weA_0&gt;</td>
</tr>
<tr>
<td>addrA</td>
<td>connected to signal &lt;addrA&gt;</td>
</tr>
<tr>
<td>diA</td>
<td>connected to signal &lt;diA&gt;</td>
</tr>
<tr>
<td>optimization</td>
<td>speed</td>
</tr>
<tr>
<td>Port B aspect ratio</td>
<td>64-word x 32-bit</td>
</tr>
<tr>
<td>mode</td>
<td>write-first</td>
</tr>
<tr>
<td>clkB</td>
<td>connected to signal &lt;clkB&gt;</td>
</tr>
<tr>
<td>enB</td>
<td>connected to signal &lt;enB&gt;</td>
</tr>
<tr>
<td>addrB</td>
<td>connected to signal &lt;addrB&gt;</td>
</tr>
<tr>
<td>doB</td>
<td>connected to signal &lt;doB&gt;</td>
</tr>
</tbody>
</table>

Unit <asymmetric_ram_1a> synthesized (advanced).
RAM Initial Contents

This section discusses RAM Initial Contents, and includes:

- Specifying Initial Contents in the HDL Source Code
- Specifying Initial Contents in an External Data File

Specifying Initial Contents in the HDL Source Code

In VHDL, use the signal default value mechanism to describe initial contents of the RAM VHDL directly in the Hardware Description Language (HDL) source code.

VHDL Coding Example One

type ram_type is array (0 to 31) of std_logic_vector(19 downto 0);
signal RAM : ram_type :=
(  X"0200A", X"00300", X"08101", X"04000", X"0233A", X"00300", X"08602",  
  X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",  
  X"00340", X"00241", X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",  
  X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021", X"0030D", X"08201" );

If all addressable words are to be initialized to the same value, you can write:

type ram_type is array (0 to 127) of std_logic_vector (15 downto 0);
signal RAM : ram_type := (others => "0000111100110101");

If all bit positions in the RAM initialize to the same value, you can write:

VHDL Coding Example Two

You can also selectively define particular values for specific address positions or ranges.

Verilog Coding Example One

In Verilog, use an initial block.

reg [19:0] ram [31:0];

initial begin
  (...)
  ram[2] = 20'h02341; ram[1] = 20'h08201; ram[0] = 20'h0400D;
end

Verilog Coding Example Two

If all addressable words initialize to the same value, you can also write:

Reg [DATA_WIDTH-1:0] ram [DEPTH-1:0];

integer i;
initial for (i=0; i<DEPTH; i=i+1) ram[i] = 0;
Verilog Coding Example Three

You can also initialize specific address positions or address ranges.

```verilog
reg [15:0] ram [255:0];

integer index;
initial begin
  for (index = 0 ; index <= 97 ; index = index + 1)
    ram[index] = 16'h8282;
  ram[98] <= 16'h1111;
  ram[99] <= 16'h7778;
  for (index = 100 ; index <= 255 ; index = index + 1)
    ram[index] = 16'hB8B8;
end
```

Specifying Initial Contents in an External Data File

Use the file read function in the HDL source code to load the initial contents from an external data file.

- The external data file is an ASCII text file with any name.
- Each line in the data file describes the initial contents at an address position in the RAM.
- There must be as many lines in the file as there are rows in the RAM array. An insufficient number of lines is flagged.
- The addressable position related to a given line is defined by the direction of the primary range of the signal modeling the RAM.
- RAM contents can be represented in either binary or hexadecimal. You cannot mix both.
- The file cannot contain any other contents, such as comments.

Following is an example of the contents of a file initializing an 8 x 32-bit RAM with binary values:

```
00001111000011110000111100001111
01001010010001110000010000010100
0000000000000011111000000000100001
1111111010100011110010000100010100
0000000000000011111000000000100001
01001010010001110000010000010100
0000000000000011111000000000100001
11111110101000111110001000010100
```

VHDL Coding Example

Load this data as follows in VHDL:

```vhdl
type RamType is array(0 to 127) of bit_vector(31 downto 0);

impure function InitRamFromFile (RamFileName : in string) return RamType is
  FILE RamFile : text is in RamFileName;
  variable RamFileLine : line;
  variable RAM : RamType;
begin
  for I in RamType'range loop
    readline (RamFile, RamFileLine);
    read (RamFileLine, RAM(I));
  end loop;
  return RAM;
end function;

signal RAM : RamType := InitRamFromFile("rams_20c.data");
```
Verilog Coding Example

In Verilog, use a $readmemb or $readmemh system task to load respectively binary-formatted or hexadecimal data.

```verilog
reg [31:0] ram [0:63];

initial begin
    $readmemb("rams_20c.data", ram, 0, 63);
end
```

For more information, see:

- VHDL File Type Support
- Chapter 5, XST Behavioral Verilog Support
Chapter 7: XST HDL Coding Techniques

Block RAM Optimization Strategies

This section discusses Block RAM Optimization Strategies, and includes:

- About Block RAM Optimization Strategies
- Block RAM Performance
- Block RAM Device Utilization
- Block RAM Power Reduction
- Rules for Small RAMs
- Mapping Logic and Finite State Machine (FSM) Components to Block RAM
- Block RAM Resource Management
- Block RAM Packing

About Block RAM Optimization Strategies

When an inferred RAM macro does not fit in a single block RAM, you may consider various strategies to partition it onto several block RAMs. Depending on your choice, the number of involved block RAM primitives and the amount of surrounding logic will vary, leading to different optimization trade-offs among performance, device utilization, and power.

Block RAM Performance

The default block RAM implementation strategy is aimed at maximizing performance. As a result, for a given RAM size requiring multiple block RAM primitives, XST does not seek to achieve the minimal theoretical number of block RAM primitives.

Implementing small RAMs on block resources often does not lead to optimal performance. Furthermore, block RAM resources can be used for those small RAMs at the expense of much larger macros. In order to achieve better design performance, XST implements small RAMs on distributed resources.

For more information, see:

Rules for Small RAMs

Block RAM Device Utilization

XST does not support area-oriented block RAM implementation. Xilinx® recommends the CORE Generator™ software for area-oriented implementation.

For more information, see:

Chapter 8, XST FPGA Optimization

Block RAM Power Reduction

Techniques to reduce block RAM power dissipation can be enabled in XST. They are part of a larger set of optimizations controlled with the Power Reduction (POWER) synthesis option, and can be specifically enabled through the RAM Style (RAM_STYLE) constraint.

RAM power optimization techniques in XST are primarily aimed at reducing the number of simultaneously active block RAMs on the device. They are applicable only to inferred memories that require a decomposition on several block RAM primitives, and take advantage of the enable capability of block RAM resources. Additional enable logic is created to ensure that only one block RAM primitive used to implement an inferred memory is simultaneously enabled. Activating power reduction techniques for an inferred memory that fits in single block RAM primitive has no effect.
Chapter 7: XST HDL Coding Techniques

When enabled, power reduction is sought in conjunction with both Area and Speed optimization goals. Two optimization trade-offs are available through the RAM Style (RAM_STYLE) constraint:

- Mode block_power1 achieves some level of power reduction, while minimally impacting circuit performance. In this mode, the default, performance-oriented, block RAM decomposition algorithm is preserved. XST simply adds block RAM enable logic. Depending on your memory characteristics, power may be impacted in only a limited way.

- Mode block_power2 provides more significant power reduction, but may leave some performance on the table. Additional slice logic may also be induced. This mode uses a different block RAM decomposition method. This method first aims to reduce the number of block RAM primitives required to implement an inferred memory. The number of active block RAMs is then minimized by inserting block RAM enable logic. Multiplexing logic is also created to read the data from active block RAMs.

If your primary concern is power reduction, and you are willing to give up some degree of speed and area optimization, Xilinx® recommends using the block_power2 mode.

Rules for Small RAMs

In order to save block RAM resources, XST does not implement small memories on block RAM. The threshold can vary depending on:

- The targeted device family
- The number of addressable data words (memory depth)
- The total number of memory bits (number of addressable data words * data word width)

Inferred RAMs are implemented on block RAM resources when the criteria in the following table are met.

Criteria for Implementing Inferred RAMs on Block RAM Resources

<table>
<thead>
<tr>
<th>Devices</th>
<th>Depth</th>
<th>Depth * Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan®-6</td>
<td>&gt;= 127 words</td>
<td>&gt; 512 bits</td>
</tr>
<tr>
<td>Virtex®-6</td>
<td>&gt;= 127 words</td>
<td>&gt; 512 bits</td>
</tr>
</tbody>
</table>

Use RAM Style (RAM_STYLE) to override these criteria and force implementation of small RAMs and ROMs on block resources.

Mapping Logic and Finite State Machine (FSM) Components to Block RAM

In addition to RAM inference capabilities, XST can be instructed to implement the following to block RAM resources:

- Finite State Machine (FSM) components
  - For more information, see: Finite State Machine (FSM) Components
- General logic
  - For more information, see: Mapping Logic to Block RAM
Block RAM Resource Management

XST takes into account actual availability of block RAM resources in order to avoid overmapping the targeted device. XST may use all block RAM resources available on the device. BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) forces XST to leave some of those resources unallocated.

XST determines the actual amount of block RAM resources available for inferred RAM macros after subtracting the following amounts from the overall pool theoretically defined by BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO):

- Block RAMs that you have instantiated.
- RAMs and ROMs that you forced to block RAM implementation with RAM Style (RAM_STYLE) or ROM Style (ROM_STYLE). XST honors those constraints before attempting to implement other inferred RAMs to block resources.
- Block RAMs resulting from the mapping of logic or Finite State Machine (FSM) components to Map Logic on BRAM (BRAM_MAP).

The XST block RAM allocation strategy also favors the largest inferred RAMs for block implementation, allowing smaller RAMs to go to block resources if there are any left on the device.

Although XST avoids it in most cases, block RAM over-utilization can happen if the sum of block RAMs created from the three cases listed above exceeds available resources.

Block RAM Packing

XST can attempt to implement more RAMs on block resources by trying to pack small single-port RAMs together. XST can implement two single-port RAMs in a single dual-port block RAM primitive, where each port manages a physically distinct part of the block RAM. This optimization is controlled by Automatic BRAM Packing (AUTO_BRAM_PACKING). It is disabled by default.

Distributed RAM Pipelining

With an adequate number of latency stages, XST can pipeline RAMs implemented on distributed resources for increased performance. The effect of pipelining is similar to Flip-Flop Retiming.

To insert pipeline stages:

1. Describe the necessary number of registers in the HDL source code.
2. Place the registers after the RAM.
3. Set RAM Style (RAM_STYLE) to:
   pipe_distributed

XST automatically calculates the ideal number of register stages needed to maximize operating frequency. If there are fewer than the ideal number, XST issues an HDL Advisor message during Advanced HDL Synthesis. The message reports the number of additional stages needed to achieve the optimum.

XST cannot pipeline distributed RAMs if the registers have asynchronous set/reset logic. XST can pipeline RAMs if registers contain synchronous reset signals.
RAMs Related Constraints

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

XST accepts LOC and RLOC on inferred RAMs that can be implemented in a single block RAM primitive. LOC and RLOC are propagated to the NGC netlist.

RAM Reporting

XST provides detailed information on inferred RAM, including size, synchronization and control signals. As shown in the following log example, RAM recognition consists of two steps:

- During HDL Synthesis, XST recognizes the presence of the memory structure in the Hardware Description Language (HDL) source code.
- During Advanced HDL Synthesis, XST acquires a more accurate picture of each RAM situation, and decides to implement them on distributed or block RAM resources, taking into account resource availability.
An inferred block RAM is generally reported as follows.

* HDL Synthesis *

Synthesizing Unit <rams_27>.
  Found 16-bit register for signal <do>.
  Found 128x16-bit dual-port <RAM Mram_RAM> for signal <RAM>.
Summary:
inferred 1 RAM(s).
inferred 16 D-type flip-flop(s).
Unit <rams_27> synthesized.

Synthesizing (advanced) Unit <rams_27>.
INFO:Xst - The <RAM Mram_RAM> will be implemented as a BLOCK RAM,
absorbing the following register(s): <do>

<table>
<thead>
<tr>
<th>ram_type</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port A</td>
<td></td>
</tr>
<tr>
<td>aspect ratio</td>
<td>128-word x 16-bit</td>
</tr>
<tr>
<td>mode</td>
<td>read-first</td>
</tr>
<tr>
<td>clkA</td>
<td>connected to signal &lt;clk&gt;</td>
</tr>
<tr>
<td>weA</td>
<td>connected to signal &lt;we&gt;</td>
</tr>
<tr>
<td>addrA</td>
<td>connected to signal &lt;waddr&gt;</td>
</tr>
<tr>
<td>diA</td>
<td>connected to signal &lt;di&gt;</td>
</tr>
<tr>
<td>optimization</td>
<td>speed</td>
</tr>
<tr>
<td>Port B</td>
<td></td>
</tr>
<tr>
<td>aspect ratio</td>
<td>128-word x 16-bit</td>
</tr>
<tr>
<td>mode</td>
<td>write-first</td>
</tr>
<tr>
<td>clkB</td>
<td>connected to signal &lt;clk&gt;</td>
</tr>
<tr>
<td>enB</td>
<td>connected to signal &lt;re&gt;</td>
</tr>
<tr>
<td>addrB</td>
<td>connected to signal &lt;addr&gt;</td>
</tr>
<tr>
<td>doB</td>
<td>connected to signal &lt;do&gt;</td>
</tr>
<tr>
<td>optimization</td>
<td>speed</td>
</tr>
</tbody>
</table>

Unit <rams_27> synthesized (advanced).

* Advanced HDL Synthesis *

Advanced HDL Synthesis Report

Macro Statistics
  # RAMs     : 1
  128x16-bit dual-port block RAM : 1
Pipelining of a distributed RAM results in the following specific reporting in the Advanced HDL Synthesis section.

```plaintext
Synthesizing (advanced) Unit <v_rams_22>.
Found pipelined ram on signal <n0006>:
  1 pipeline level(s) found in a register on signal <n0006>.
  Pushing register(s) into the ram macro.
INFO:Xst:2390 - HDL ADVISOR - You can improve the performance of the ram Mram_RAM by adding 1 register level(s) on output signal n0006.
Unit <v_rams_22> synthesized (advanced).
```

**RAMs Coding Examples**

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a `summary.txt` file listing all examples together with a brief overview.

**Single-Port RAM with Asynchronous Read (Distributed RAM) VHDL Coding Example**

```vhdl
-- -- Single-Port RAM with Asynchronous Read (Distributed RAM)
-- File: HDL_Coding_Techniques/rams/rams_04.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

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

architecture syn of rams_04 is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (we = '1') then
        RAM(conv_integer(a)) <= di;
      end if;
    end if;
  end process;
  do <= RAM(conv_integer(a));
end syn;
```
**Dual-Port RAM with Asynchronous Read (Distributed RAM) Verilog Coding Example**

```verilog
// Dual-Port RAM with Asynchronous Read (Distributed RAM)
// File: HDL_Coding_Techniques/rams/rams_09.v
// module v_rams_09 (clk, we, a, dpra, di, spo, dpo);

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

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

**Single-Port Block RAM Read-First Mode VHDL Coding Example**

```vhdl
-- Single-Port Block RAM Read-First Mode
-- File: HDL_Coding_Techniques/rams/rams_01.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rams_01 is
  port (clk : in std_logic;
       we : in std_logic;
       en : in std_logic;
       addr : in std_logic_vector(5 downto 0);
       di : in std_logic_vector(15 downto 0);
       do : out std_logic_vector(15 downto 0));
end rams_01;

architecture syn of rams_01 is
  type ram_type is array (63 downto 0) of std_logic_vector(15 downto 0);
  signal RAM: ram_type;
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
        end if;
        do <= RAM(conv_integer(addr));
      end if;
    end if;
  end process;
end syn;
```
Single-Port Block RAM Read-First Mode Verilog Coding Example

//
// Single-Port Block RAM Read-First Mode
//
// File: HDL_Coding_Techniques/rams/rams_01.v
//
module v_rams_01 (clk, en, we, addr, di, do);

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

    always @(posedge clk)
    begin
        if (en)
        begin
            if (we)
                RAM[addr]<=di;
            do <= RAM[addr];
        end
    end
endmodule
Single-Port Block RAM Write-First Mode VHDL Coding Example

--
-- Single-Port Block RAM Write-First Mode (recommended template)
--
-- File: HDL_Coding_Techniques/rams/rams_02a.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

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

architecture syn of rams_02a is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
          do <= di;
        else
          do <= RAM(conv_integer(addr));
        end if;
      end if;
    end if;
  end process;
end syn;
Single-Port Block RAM Write-First Mode Verilog Coding Example

//
// Single-Port Block RAM Write-First Mode (recommended template)
//
// File: HDL_Coding_Techniques/rams/rams_02a.v
//
module v_rams_02a (clk, we, en, addr, di, do);

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

    always @(posedge clk)
    begin
        if (en)
            begin
                if (we)
                    begin
                        RAM[addr] <= di;
                        do <= di;
                    end
                else
                    do <= RAM[addr];
            end
    end
endmodule
Single-Port Block RAM No-Change Mode VHDL Coding Example

-- -- Single-Port Block RAM No-Change Mode
-- -- File: HDL_Coding_Techniques/rams/rams_03.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

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

architecture syn of rams_03 is
  type ram_type is
    array (63 downto 0) of std_logic_vector (15 downto 0);
  signal RAM : ram_type;
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
        else
          do <= RAM(conv_integer(addr));
        end if;
      end if;
    end if;
  end process;
end syn;
Single-Port Block RAM No-Change Mode Verilog Coding Example

```verilog
//
// Single-Port Block RAM No-Change Mode
//
// File: HDL_Coding_Techniques/rams/rams_03.v
//
module v_rams_03 (clk, we, en, addr, di, do);
    input clk;
    input we;
    input en;
    input [5:0] addr;
    input [15:0] di;
    output [15:0] do;
    reg [15:0] RAM [63:0];
    reg [15:0] do;
    always @(posedge clk)
        begin
            if (en)
                begin
                    if (we)
                        RAM[addr] <= di;
                    else
                        do <= RAM[addr];
                end
        end
endmodule
```
Dual-Port Block RAM with Two Write Ports VHDL Coding Example

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

entity rams_16b is
  port(clka : in std_logic;
       clkb : in std_logic;
       ena : in std_logic;
       enb : in std_logic;
       wea : in std_logic;
       web : in std_logic;
       addra : in std_logic_vector(6 downto 0);
       addrb : in std_logic_vector(6 downto 0);
       dia : in std_logic_vector(15 downto 0);
       dib : in std_logic_vector(15 downto 0);
       doa : out std_logic_vector(15 downto 0);
       dob : out std_logic_vector(15 downto 0));
end rams_16b;

architecture syn of rams_16b is
  type ram_type is array (127 downto 0) of std_logic_vector(15 downto 0);
  shared variable RAM : ram_type;
begin
  process (CLKA)
  begin
    if CLKA'event and CLKA = ‘1’ then
      if ENA = ‘1’ then
        DOA <= RAM(conv_integer(ADDRA));
        if WEA = ‘1’ then
          RAM(conv_integer(ADDRA)) := DIA;
        end if;
      end if;
    end if;
  end process;

  process (CLKB)
  begin
    if CLKB'event and CLKB = ‘1’ then
      if ENB = ‘1’ then
        DOB <= RAM(conv_integer(ADDRB));
        if WEB = ‘1’ then
          RAM(conv_integer(ADDRB)) := DIB;
        end if;
      end if;
    end if;
  end process;
end syn;
Dual-Port Block RAM with Two Write Ports Verilog Coding Example

```verilog
// Dual-Port Block RAM with Two Write Ports
// File: HDL_Coding_Techniques/rams/rams_16.v
module v_rams_16(clka,clkb,ena,enb,wea,web,addra,addrb,dia,dib,doa,dob);

    input clka,clkb,ena,enb,wea,web;
    input [5:0] addra,addrb;
    input [15:0] dia,dib;
    output [15:0] doa,dob;
    reg [15:0] doa,dob;

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

    always @(posedge clkb) begin
        if (enb)
            begin
                if (web)
                    ram[addrb] <= dib;
                dob <= ram[addrb];
            end
    end

endmodule
```
**Block RAM with Resettable Data Output VHDL Coding Example**

```vhdl
-- -- Block RAM with Resettable Data Output --

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

entity rams_18 is
  port (clk : in std_logic;
        en : in std_logic;
        we : in std_logic;
        rst : in std_logic;
        addr : in std_logic_vector(6 downto 0);
        di : in std_logic_vector(15 downto 0);
        do : out std_logic_vector(15 downto 0));
end rams_18;

architecture syn of rams_18 is
  type ram_type is array (127 downto 0) of std_logic_vector (15 downto 0);
  signal ram : ram_type;
begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if en = '1' then -- optional enable
        if we = '1' then -- write enable
          ram(conv_integer(addr)) <= di;
        end if;
      end if;
      if rst = '1' then -- optional reset
        do <= (others => '0');
      else
        do <= ram(conv_integer(addr));
      end if;
    end if;
  end process;
end syn;
```
Block RAM with Resettable Data Output Verilog Coding Example

```verilog
//
// Block RAM with Resettable Data Output
//
// File: HDL_Coding_Techniques/rams/rams_18.v
//
module v_rams_18 (clk, en, we, rst, addr, di, do);

input clk;
input en;
input we;
input rst;
input [6:0] addr;
input [15:0] di;
output [15:0] do;
reg [15:0] ram [127:0];
reg [15:0] do;

always @(posedge clk)
begin
    if (en) // optional enable
        begin
            if (we) // write enable
                ram[addr] <= di;

            if (rst) // optional reset
                do <= 16'b0000111100001101;
            else
                do <= ram[addr];
        end
end
endmodule
```
Block RAM with Optional Output Registers VHDL Coding Example

--
-- Block RAM with Optional Output Registers
--
-- File: HDL_Coding_Techniques/rams/rams_19.vhd
--
library IEEE;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

dentity rams_19 is
  port (clk1, clk2 : in std_logic;
       we, en1, en2 : in std_logic;
       addr1 : in std_logic_vector(5 downto 0);
       addr2 : in std_logic_vector(5 downto 0);
       di : in std_logic_vector(15 downto 0);
       res1 : out std_logic_vector(15 downto 0);
       res2 : out std_logic_vector(15 downto 0));
end rams_19;

architecture beh of rams_19 is
  type ram_type is array (63 downto 0) of std_logic_vector (15 downto 0);
  signal ram : ram_type;
  signal do1 : std_logic_vector(15 downto 0);
  signal do2 : std_logic_vector(15 downto 0);
begin

  process (clk1)
  begin
    if rising_edge(clk1) then
      if we = '1' then
        ram(conv_integer(addr1)) <= di;
      end if;
      do1 <= ram(conv_integer(addr1));
    end if;
  end process;

  process (clk2)
  begin
    if rising_edge(clk2) then
      do2 <= ram(conv_integer(addr2));
    end if;
  end process;

  process (clk1)
  begin
    if rising_edge(clk1) then
      if en1 = '1' then
        res1 <= do1;
      end if;
    end if;
  end process;

  process (clk2)
  begin
    if rising_edge(clk2) then
      if en2 = '1' then
        res2 <= do2;
      end if;
    end if;
  end process;

end beh;
Block RAM with Optional Output Registers Verilog Coding Example

```
module v_rams_19 (clk1, clk2, we, en1, en2, addr1, addr2, di, res1, res2);

    input clk1;
    input clk2;
    input we, en1, en2;
    input [6:0] addr1;
    input [6:0] addr2;
    input [15:0] di;
    output [15:0] res1;
    output [15:0] res2;
    reg [15:0] res1;
    reg [15:0] res2;
    reg [15:0] RAM [127:0];
    reg [15:0] do1;
    reg [15:0] do2;

always @(posedge clk1)
    begin
        if (we == 1'b1)
            RAM[addr1] <= di;
        do1 <= RAM[addr1];
    end

always @(posedge clk2)
    begin
        do2 <= RAM[addr2];
    end

always @(posedge clk1)
    begin
        if (en1 == 1'b1)
            res1 <= do1;
    end

always @(posedge clk2)
    begin
        if (en2 == 1'b1)
            res2 <= do2;
    end
endmodule
```
Initializing Block RAM (Single-Port Block RAM) VHDL Coding Example

--
-- Initializing Block RAM (Single-Port Block RAM)
--
-- File: HDL_Coding_Techniques/rams/rams_20a.vhd

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

entity rams_20a is
  port (clk : in std_logic;
        we : in std_logic;
        addr : in std_logic_vector(5 downto 0);
        di : in std_logic_vector(19 downto 0);
        do : out std_logic_vector(19 downto 0));
end rams_20a;

architecture syn of rams_20a is
  type ram_type is array (63 downto 0) of std_logic_vector (19 downto 0);
  begin
    process (clk)
    begin
      if rising_edge(clk) then
        if we = '1' then
          RAM(conv_integer(addr)) <= di;
        end if;
        do <= RAM(conv_integer(addr));
      end if;
    end process;
  end syn;
Initializing Block RAM (Single-Port Block RAM) Verilog Coding Example

module v_rams_20a (clk, we, addr, di, do);
    input clk;
    input we;
    input [5:0] addr;
    input [19:0] di;
    output [19:0] do;
    reg [19:0] ram [63:0];
    reg [19:0] do;
    initial begin
        ram[63] = 20'h0200A; ram[62] = 20'h00300; ram[61] = 20'h08101;
        ram[60] = 20'h04000; ram[59] = 20'h08601; ram[58] = 20'h0233A;
        ram[57] = 20'h00300; ram[56] = 20'h08602; ram[55] = 20'h02310;
        ram[54] = 20'h0203B; ram[53] = 20'h08300; ram[52] = 20'h04002;
        ram[51] = 20'h08201; ram[50] = 20'h00500; ram[49] = 20'h04001;
        ram[48] = 20'h02500; ram[47] = 20'h00340; ram[46] = 20'h00241;
        ram[45] = 20'h04002; ram[44] = 20'h08300; ram[43] = 20'h08201;
        ram[42] = 20'h00500; ram[41] = 20'h08101; ram[40] = 20'h00602;
        ram[39] = 20'h04003; ram[38] = 20'h0241E; ram[37] = 20'h00301;
        ram[36] = 20'h00102; ram[35] = 20'h02122; ram[34] = 20'h00201;
        ram[33] = 20'h00301; ram[32] = 20'h00102; ram[31] = 20'h02222;
        ram[30] = 20'h00304; ram[29] = 20'h00342; ram[28] = 20'h00232B;
        ram[27] = 20'h00900; ram[26] = 20'h00302; ram[25] = 20'h00102;
        ram[24] = 20'h04002; ram[23] = 20'h00900; ram[22] = 20'h08201;
        ram[18] = 20'h00301; ram[17] = 20'h04004; ram[16] = 20'h00301;
        ram[0] = 20'h0400D;
    end
    always @(posedge clk)
        begin
            if (we)
                ram[addr] <= di;
            do <= ram[addr];
        end
endmodule
Chapter 7: XST HDL Coding Techniques

Initializing Block RAM From an External Data File VHDL Coding Example

```vhdl
-- Initializing Block RAM from external data file
-- File: HDL_Coding_Techniques/rams/rams_20c.vhd

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

entity rams_20c is
  port(clk : in std_logic;
       we : in std_logic;
       addr : in std_logic_vector(5 downto 0);
       din : in std_logic_vector(31 downto 0);
       dout : out std_logic_vector(31 downto 0));
end rams_20c;

architecture syn of rams_20c is
  type RamType is
    array(0 to 63) of bit_vector(31 downto 0);
  impure function InitRamFromFile (RamFileName : in string) return RamType is
    FILE RamFile : text is in RamFileName;
    variable RamFileLine : line;
    variable RAM : RamType;
  begin
    for I in RamType'range loop
      readline (RamFile, RamFileLine);
      read (RamFileLine, RAM(I));
    end loop;
    return RAM;
  end function;

  signal RAM : RamType := InitRamFromFile("rams_20c.data");

begin
  process (clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
        RAM(conv_integer(addr)) <= to_bitvector(din);
      end if;
      dout <= to_stdlogicvector(RAM(conv_integer(addr)));
    end if;
  end process;
end syn;
```

XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices
UG687 (v 13.1) March 1, 2011 www.xilinx.com 243
Initializing Block RAM From an External Data File Verilog Coding Example

//
// Initializing Block RAM from external data file
// Binary data
//
// File: HDL_Coding_Techniques/rams/rams_20c.v
//
module v_rams_20c (clk, we, addr, din, dout);
    input clk;
    input we;
    input [5:0] addr;
    input [31:0] din;
    output [31:0] dout;
    reg [31:0] ram [0:63];
    reg [31:0] dout;

    initial begin
        // $readmemb("rams_20c.data",ram, 0, 63);
        $readmemb("rams_20c.data",ram);
    end

    always @(posedge clk)
    begin
        if (we)
            ram[addr] <= din;
        dout <= ram[addr];
    end
endmodule
Pipelined Distributed RAM VHDL Coding Example

--
-- Pipeline distributed RAM
--
-- File: HDL_Coding_Techniques/rams/rams_22.vhd
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

dentity rams_22 is
  port (clk : in std_logic;
        we : in std_logic;
        addr : in std_logic_vector(8 downto 0);
        di : in std_logic_vector(3 downto 0);
        do : out std_logic_vector(3 downto 0));
end rams_22;

architecture syn of rams_22 is
  type ram_type is array (511 downto 0) of std_logic_vector (3 downto 0);
  signal RAM : ram_type;

  signal pipe_reg: std_logic_vector(3 downto 0);

  attribute ram_style: string;
  attribute ram_style of RAM: signal is "pipe_distributed";
begin

  process (clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
        RAM(conv_integer(addr)) <= di;
      else
        pipe_reg <= RAM( conv_integer(addr));
      end if;
      do <= pipe_reg;
    end if;
  end process;

end syn;
Pipelined Distributed RAM Verilog Coding Example

//
// Pipeline distributed RAM
//
// File: HDL_Coding_Techniques/rams/rams_22.v
//
module v_rams_22 (clk, we, addr, di, do);

    input clk;
    input we;
    input [8:0] addr;
    input [3:0] di;
    output [3:0] do;

    (*ram_style="pipe_distributed"*)
    reg [3:0] RAM [511:0];
    reg [3:0] do;
    reg [3:0] pipe_reg;

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

        do <= pipe_reg;
    end

endmodule
ROMs

This section discusses XST HDL Coding Techniques for ROMs, and includes:
- About ROMs
- ROMs Description
- ROMs Implementation
- ROMs Related Constraints
- ROMs Reporting
- ROMs Coding Examples

About ROMs

Read-Only Memory (ROM) has much in common with RAM in terms of Hardware Description Language (HDL) modeling and implementation. If properly registered, a ROM can also be implemented on block RAM resources by XST.

ROMs Description

This section discusses ROMs Description, and includes:
- ROMs Modeling
- Describing Read Access

ROMs Modeling

ROMs are usually modeled in VHDL with an array of array object. In VHDL, this object can be either a constant or a signal. Xilinx® recommends using a signal. A signal allows you to control implementation of the ROM, either on LUT resources, or on block RAM resources. To control implementation of the ROM, attach a ROM Style (ROM_STYLE) or RAM Style (RAM_STYLE) constraint to the signal.

Constant-Based Declaration VHDL Coding Example

```vhdl
type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
constant ROM : rom_type := (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500", (...)
X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);
```

Signal-Based Declaration VHDL Coding Example

```vhdl
type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
signal ROM : rom_type := (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500", (...)
X"04078", X"01110", X"02500", X"02500", X"0030D", X"02341", X"08201", X"0410D"
);
```
ROM Modeled With Initial Block Verilog Coding Example

A ROM can be modeled in Verilog with an initial block. Verilog does not allow initializing an array with a single statement as allowed by VHDL aggregates. You must enumerate each address value.

```verilog
reg [15:0] rom [15:0];

initial begin
  rom[0] = 16'b0011111110000010;
  rom[1] = 16'b0000000100001001;
  rom[2] = 16'b0001000000111000;
  rom[3] = 16'b0000000001110000;
  rom[4] = 16'b0100001010011000;
  rom[5] = 16'b0000000000000000;
  rom[6] = 16'b0000000110000000;
  rom[7] = 16'b0111111111110000;
  rom[8] = 16'b001000010001001;
  rom[9] = 16'b0101010101011000;
  rom[10] = 16'b0101010101011000;
  rom[11] = 16'b0000000000000000;
  rom[12] = 16'b0111000000000000;
  rom[13] = 16'b0000000110001010;
  rom[14] = 16'b0110011100010000;
  rom[15] = 16'b0000100010000000;
end
```

Describing ROM With a Case Statement Coding Example

You can also describe the ROM with a case statement (or equivalent if-elseif construct).

```verilog
input [3:0] addr
output reg [15:0] data;

always @(posedge clk) begin
  if (en) begin
    case (addr)
      4'b0000: data <= 16'h200A;
      4'b0001: data <= 16'h0300;
      4'b0010: data <= 16'h8101;
      4'b0011: data <= 16'h4000;
      4'b0100: data <= 16'h8601;
      4'b0101: data <= 16'h233A;
      4'b0110: data <= 16'h0300;
      4'b0111: data <= 16'h8602;
      4'b1000: data <= 16'h2222;
      4'b1001: data <= 16'h4001;
      4'b1010: data <= 16'h0342;
      4'b1011: data <= 16'h232B;
      4'b1100: data <= 16'h0900;
      4'b1101: data <= 16'h0302;
      4'b1110: data <= 16'h0102;
      4'b1111: data <= 16'h4002;
    endcase
  end
end
```

Loading the contents of the ROM from an external data file:

- Results in more compact and readable Hardware Description Language (HDL) source code
- Allows more flexibility in generating or altering the ROM data
For more information, see:  
Specifying Initial Contents in an External Data File

**Describing Read Access**

Describing access to ROM is similar to describing access to RAM.

**Describing Read Access VHDL Coding Example**

If you have included the IEEE std_logic_unsigned package defining the conv_integer conversion function, the VHDL syntax is:

```vhdl
signal addr : std_logic_vector(ADDR_WIDTH-1 downto 0);
do <= ROM( conv_integer(addr));
```

**Describing Read Access Verilog Coding Example**

If you have modeled the ROM in an initial block (with data described in the Verilog source code or loaded from an external data file), the Verilog syntax is:

```verilog
do <= ROM[addr];
```

Alternatively in Verilog, use a case construct as shown in Describing ROM With a Case Statement Coding Example.

**ROMs Implementation**

When XST detects that a properly synchronized ROM can be implemented on block RAM resources, the principles discussed in Block RAM Optimization Strategies apply. To override any default XST decision criteria, use ROM Style (ROM_STYLE) instead of RAM Style (RAM_STYLE).

- For more information about ROM Style (ROM_STYLE), see Chapter 9, XST Design Constraints.
- For more information about ROM implementation, see Chapter 8, XST FPGA Optimization.

**ROMs Related Constraints**

ROM Style (ROM_STYLE)
ROMs Reporting

The following report shows how the ROM is identified during HDL Synthesis. Based on the availability of proper synchronization, the decision to implement a ROM on block RAM resources in made during Advanced HDL Synthesis.

```plaintext
Synthesizing Unit <roms_signal>.
Found 20-bit register for signal <data>.
Found 128x20-bit ROM for signal <n0024>.
Summary:
inferred 1 ROM(s).
inferred 20 D-type flip-flop(s).
Unit <roms_signal> synthesized.
```

HDL Synthesis Report

```
Macro Statistics
# ROMs : 1
# Registers : 1
20-bit register : 1
```

```
Synthesizing (advanced) Unit <roms_signal>.
INFO:Xst - The RAM <Mrom_ROM> will be implemented as BLOCK RAM, absorbing the register: <data>.
INFO:Xst - The RAM <Mrom_ROM> will be implemented as BLOCK RAM
```

```
<table>
<thead>
<tr>
<th>ram_type</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port A</td>
<td></td>
</tr>
<tr>
<td>aspect ratio</td>
<td>128-word x 20-bit</td>
</tr>
<tr>
<td>mode</td>
<td>write-first</td>
</tr>
<tr>
<td>clkA</td>
<td>connected to signal &lt;clk&gt;</td>
</tr>
<tr>
<td>enA</td>
<td>connected to signal &lt;en&gt;</td>
</tr>
<tr>
<td>weA</td>
<td>connected to internal node</td>
</tr>
<tr>
<td>addrA</td>
<td>connected to internal node</td>
</tr>
<tr>
<td>diaA</td>
<td>connected to signal &lt;data&gt;</td>
</tr>
<tr>
<td>doA</td>
<td>connected to signal &lt;data&gt;</td>
</tr>
<tr>
<td>optimization</td>
<td>speed</td>
</tr>
</tbody>
</table>

Unit <roms_signal> synthesized (advanced).

Advanced HDL Synthesis Report

```
Macro Statistics
# RAMs : 1
128x20-bit single-port block RAM : 1
```

ROMs Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from http://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Description of a ROM with a VHDL Constant Coding Example

--
-- Description of a ROM with a VHDL constant
--
-- File: HDL_Coding_Techniques/rams/roms_constant.vhd

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

entity roms_constant is
  port (clk : in std_logic;
        en : in std_logic;
        addr : in std_logic_vector(6 downto 0);
        data : out std_logic_vector(19 downto 0));
end roms_constant;

architecture syn of roms_constant is

  type rom_type is array (0 to 127) of std_logic_vector (19 downto 0);
  constant ROM : rom_type :=
    (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
     X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
     X"00340", X"00241", X"04002", X"08100", X"08201", X"00500", X"08101", X"00602",
     X"04003", X"0241E", X"00301", X"00102", X"02122", X"00201", X"00301", X"00102",
     X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
     X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
     X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"04000",
     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
     X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
     X"00340", X"00241", X"04112", X"08300", X"08201", X"00500", X"08101", X"00602",
     X"04003", X"0241E", X"00301", X"00102", X"02122", X"00201", X"00301", X"00102",
     X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
     X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
     X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"04000",
     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
     X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
     X"00340", X"00241", X"04112", X"08300", X"08201", X"00500", X"08101", X"00602",
     X"04003", X"0241E", X"00301", X"00102", X"02122", X"00201", X"00301", X"00102",
     X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
     X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
     X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"04000",
     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
     X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
     X"00340", X"00241", X"04112", X"08300", X"08201", X"00500", X"08101", X"00602",
     X"04003", X"0241E", X"00301", X"00102", X"02122", X"00201", X"00301", X"00102",
     X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
     X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
     X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"04000",
     X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A", X"00300", X"08602",
     X"02310", X"0203B", X"08300", X"04002", X"08201", X"00500", X"04001", X"02500",
     X"00340", X"00241", X"04112", X"08300", X"08201", X"00500", X"08101", X"00602",
     X"04003", X"0241E", X"00301", X"00102", X"02122", X"00201", X"00301", X"00102",
     X"02222", X"04001", X"00342", X"0232B", X"00900", X"00302", X"00102", X"04002",
     X"00900", X"08201", X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
     X"00102", X"02137", X"02036", X"00301", X"00102", X"02237", X"04004", X"00304",
     X"04040", X"02500", X"02500", X"02500", X"0030D", X"02341", X"08201", X"04000"
    );

begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (en = '1') then
        data <= ROM(conv_integer(addr));
      end if;
    end if;
  end process;
end syn;
prolog

ROMs Using Block RAM Resources Verilog Coding Example

//
// ROMs Using Block RAM Resources.
// Verilog code for a ROM with registered output (template 1)
//
// File: HDL_Coding_Techniques/rams/rams_21a.v
//
module v_rams_21a (clk, en, addr, data);

input clk;
input en;
input [5:0] addr;
output reg [19:0] data;

always @(posedge clk) begin
if (en)
  case(addr)
  6'b000000: data <= 20'h02000A;
  6'b000001: data <= 20'h00300;
  6'b000010: data <= 20'h08101;
  6'b000011: data <= 20'h04000;
  6'b000100: data <= 20'h08601;
  6'b000101: data <= 20'h0233A;
  6'b000110: data <= 20'h08300;
  6'b000111: data <= 20'h08602;
  6'b001000: data <= 20'h02310;
  6'b001001: data <= 20'h0203B;
  6'b001010: data <= 20'h08300;
  6'b001011: data <= 20'h00301;
  6'b001100: data <= 20'h08201;
  6'b001101: data <= 20'h00500;
  6'b001110: data <= 20'h04001;
  6'b001111: data <= 20'h02500;
  6'b010000: data <= 20'h00340;
  6'b010001: data <= 20'h08201;
  6'b010010: data <= 20'h00500;
  6'b010011: data <= 20'h00102;
  6'b010100: data <= 20'h08602;
  6'b010101: data <= 20'h08101;
  6'b010110: data <= 20'h00342;
  6'b010111: data <= 20'h00900;
  6'b011000: data <= 20'h02023;
  6'b011001: data <= 20'h0232B;
  6'b011010: data <= 20'h00900;
  6'b011011: data <= 20'h00102;
  6'b011100: data <= 20'h08101;
  6'b011101: data <= 20'h0203B;
  6'b011110: data <= 20'h00301;
  6'b011111: data <= 20'h02341;
  endcase
end
endmodule
Chapter 7: XST HDL Coding Techniques

Dual-Port ROM VHDL Coding Example

-- A dual-port ROM
-- Implementation on LUT or BRAM controlled with a ram_style constraint
-- File: HDL_Coding_Techniques/rams/roms_dualport.vhd

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

entity roms_dualport is
  port (clk : in std_logic;
        ena, enb : in std_logic;
        addra, addrb : in std_logic_vector(5 downto 0);
        dataa, datab : out std_logic_vector(19 downto 0));
end roms_dualport;

architecture behavioral of roms_dualport is
  type rom_type is array (63 downto 0) of std_logic_vector (19 downto 0);
  signal ROM : rom_type := (X"0200A", X"00300", X"08101", X"04000", X"08601", X"0233A",
                           X"00300", X"08602", X"02310", X"0203B", X"08300", X"04002",
                           X"08201", X"00500", X"04001", X"02500", X"00340", X"00241",
                           X"04002", X"08300", X"08201", X"00500", X"08101", X"00602",
                           X"04003", X"0241E", X"00301", X"00102", X"02122", X"02021",
                           X"00301", X"00102", X"02222", X"04001", X"00342", X"0232B",
                           X"00900", X"00302", X"00102", X"04002", X"00900", X"08201",
                           X"02023", X"00303", X"02433", X"00301", X"04004", X"00301",
                           X"00102", X"02137", X"02036", X"00301", X"00102", X"02237",
                           X"04004", X"00304", X"04040", X"02500", X"02500", X"02500",
                           X"0030D", X"02341", X"08201", X"0400D");
  -- attribute ram_style : string;
  -- attribute ram_style : ram_style : "distributed";

begin
  process (clk)
  begin
    if rising_edge(clk) then
      if {ena = '1'} then
        dataa <= ROM(conv_integer(addra));
      end if;
    end if;
  end process;

  process (clk)
  begin
    if rising_edge(clk) then
      if {enb = '1'} then
        datab <= ROM(conv_integer(addrb));
      end if;
    end if;
  end process;

end behavioral;
Chapter 7: XST HDL Coding Techniques

FSM Components

This section discusses XST HDL Coding Techniques for Finite State Machine (FSM) components, and includes:

- About FSM Components
- FSM Components Description
- Implementing FSM Components on block RAM Resources
- FSM Safe Implementation
- FSM Related Constraints
- FSM Reporting
- FSM Coding Examples

About FSM Components

XST features specific inference capabilities for synchronous Finite State Machine (FSM) components, as well as several built-in FSM encoding strategies to accommodate your optimization goals. You may also instruct XST to follow your own encoding scheme.

FSM extraction is enabled by default. To disable it, use Automatic FSM Extraction (FSM_EXTRACT).

FSM Description

This section discusses Finite State Machine (FSM) Description, and includes:

- About FSM Description
- State Register
- Next State Equation
- Unreachable States
- FSM Outputs
- FSM Inputs
- State Encoding Techniques

About FSM Description

XST supports specification of Finite State Machine (FSM) in both Moore and Mealy form. There are many coding variations. The following guidelines:

- Ensure maximum readability
- Maximize the ability of XST to identify the FSM

An FSM consists of:

- State register
- Next state function
- Outputs function

The following figure shows a Mealy-type FSM.
You can choose among the following HDL coding approaches, depending on your goals with respect to code compactness and readability.

- Describe all three components of the FSM in a single sequential process or `always` block.
- Describe the state register and next state function together in a sequential process or `always` block, and describe the outputs function in a separate combinatorial process or `always` block.
- Describe the state register in a sequential process or `always` block, and describe the next state and outputs functions together in a separate combinatorial process or `always` block.
- Describe the state register in a sequential process or `always` block, describe the next state function in a first combinatorial process or `always` block, and describe the outputs function in a second separate combinatorial process or `always` block.

### State Register

Specify a reset or power-up state for XST to identify a Finite State Machine (FSM). The state register can be asynchronously or synchronously reset to a particular state. For an FSM, as for any type of sequential logic, Xilinx® recommends synchronous reset logic over asynchronous.

### State Register VHDL Coding Example

You can specify the state register in VHDL with a standard type such as `integer`, `bit_vector`, or `std_logic_vector`. Another common coding practice is to define an enumerated type containing all possible state values and to declare the state register with that type.

```vhdl
type state_type is (state1, state2, state3, state4);
signal state : state_type;
```
**State Register Verilog Coding Example**

The type of a state register in Verilog can be an integer or a set of defined parameters.

```verilog
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 Equation**

Next state equations can be described:

- Directly in the sequential process, or
- In a separate combinatorial process

The sensitivity list of a separate combinatorial process contains:

- The state signal
- All Finite State Machine (FSM) inputs

The simplest coding example is based on a `case` statement, the selector of which is the current state signal.

**Unreachable States**

XST detects and reports unreachable states.

**FSM Outputs**

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

**Finite State Machine (FSM) Inputs**

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

**State Encoding Techniques**

XST features several encoding techniques that can accommodate different optimization goals, and different Finite State Machine (FSM) patterns. Select the desired encoding technique with `FSM Encoding Algorithm (FSM_ENCODING)`.

**Auto State Encoding**

In automatic mode, XST tries to select the best suited encoding method for a given FSM.

**One-Hot State Encoding**

One-Hot State Encoding is the default encoding scheme. It assigns a distinct bit of code to each FSM state. As a result, the state register is implemented with one flip-flop for each state. In a given clock cycle during operation, one and only one bit of the state register is asserted. Only two bits toggle during a transition between two states. One-Hot State Encoding is usually a good choice for optimizing speed or reducing power dissipation.
Gray State Encoding

Gray State Encoding:

- Guarantees that only one bit switches between two consecutive states. It is appropriate for controllers exhibiting long paths without branching.
- Minimizes hazards and glitches. Good results can be obtained when implementing the state register with T flip-flops.
- Can be used to minimize power dissipation.

Compact State Encoding

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

Johnson State Encoding

Like Gray State Encoding, Johnson State Encoding is beneficial when using state machines containing long paths with no branching.

Sequential State Encoding

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

Speed1 State Encoding

Speed1 State Encoding is oriented for speed optimization. The number of bits for a state register depends on the specific FSM, but is generally greater than the number of FSM states.

User State Encoding

In User State Encoding, XST uses the original encoding specified in the HDL file. For example, if the state register is described based on an enumerated type:

- Use Enumerated Encoding (ENUM_ENCODING) to assign a specific binary value to each state.
- Select User State Encoding to instruct XST to follow your coding scheme.

For more information, see:

Chapter 9, XST Design Constraints

Implementing FSM Components on Block RAM Resources

Finite State Machine (FSM) components are implemented on slice logic. In order to save slice logic resources on the targeted device, you can instruct XST to implement FSM components in block RAM. Such implementation can also favorably impact performance of large FSM components. Use FSM Style (FSM_STYLE) to chose between the default implementation on slice logic and block RAM implementation.

The values for this constraint are:

- lut (default)
- bram

If XST cannot implement an FSM in block RAM, XST:

- Automatically implements the state machine in slice logic.
- Issues a warning during Advanced HDL Synthesis.

Such failure usually occurs if the FSM has an asynchronous reset.
**FSM Safe Implementation**

By default, XST detects and optimizes unreachable states (both logical and physical) and related transition logic. This ensures implementation of a state machine that:

- Uses minimal device resources, and
- Provides optimal circuit performance

This is the standard approach for the great majority of applications. These applications operate in normal external conditions. Their temporary failure due to a single event upset will not have critical consequences.

Other applications may operate in external conditions where the probability and potentially catastrophic impact of soft errors cannot be ignored. These errors are caused primarily by cosmic rays or alpha particles from the chip packaging. State machines are particularly sensible to such errors. A state machine may never resume normal operation after an external condition sends it to an illegal state. For the circuit to detect and be able to recover from those errors, unreachable states should therefore not be optimized away.

Use **Safe Implementation (SAFE_IMPLEMENTATION)** to prevent such optimization. XST creates additional logic allowing the state machine to:

- Detect an illegal transition
- Return to a valid recovery state

By default, XST selects the reset state as the recovery state. If no reset state is available, XST selects the power-up state. Use **Safe Recovery State (SAFE_RECOVERY_STATE)** to manually define a specific recovery state.

Safe Finite State Machine (FSM) design is a subject of debate. There is no single perfect solution. Xilinx® recommends that you carefully review the following sections before deciding on your implementation strategy.

**One-Hot Encoding Versus Binary Encoding**

With binary encoding strategies (such as compact, sequential, and gray), the state register is implemented with a minimum number of flip-flops. One-hot encoding implies a larger number of flip-flops (one for each valid state). This increases the likelihood of a single event upset affecting the state register.

Despite this drawback, one-hot encoding has a significant topological benefit. A Hamming distance of 2 makes all single bit errors easily detectable. An illegal transition resulting from a single bit error always sends the state machine to an invalid state. The XST safe implementation logic ensures that any such error is detected and cleanly recovered from.

An equivalent binary coded state machine has a Hamming distance of 1. As a result, a single bit error may send the state machine to an unexpected – but valid – state. If the number of valid states is a power of 2, all possible code values correspond to a valid state, and a soft error always produces such an outcome. In that event, the circuit does not detect that an illegal transition has occurred, and that the state machine has not executed its normal state sequence. Such a random and uncontrolled recovery may not be acceptable.

**Recovery-Only State**

It may be a good design practice to define a recovery state that is none of the normal operating states of your state machine. Defining a recovery-only state allows you to:

- Detect that the state machine has been affected by a single event upset, and
- Perform specific actions before resuming normal operation. Such actions include flagging the recovery condition to the rest of the circuit or to a circuit output.
Directly recovering to a normal operation state is sufficient, provided that the faulty state machine does not need to:

- Inform the rest of the circuit of its temporary condition, or
- Perform specific actions following a soft error

**FSM Safe Implementation VHDL Coding Example**

```vhdl
-- Finite State Machine Safe Implementation VHDL Coding Example
-- One-hot encoding
-- Recovery-only state
-- File: HDL_Coding_Techniques/state_machines/safe_fsm.vhd
library ieee;
use ieee.std_logic_1164.all;

entity safe_fsm is

    port(
        clk : in std_logic;
        rst : in std_logic;
        c : in std_logic_vector(3 downto 0);
        d : in std_logic_vector(3 downto 0);
        q : out std_logic_vector(3 downto 0));

end safe_fsm;

architecture behavioral of safe fsm is

    type state_t is (idle, state0, state1, state2, recovery);
    signal state, next_state : state_t;

    attribute fsm_encoding : string;
    attribute fsm_encoding of state : signal is "one-hot";
    attribute safe_implementation : string;
    attribute safe_implementation of state : signal is "yes";
    attribute safe_recovery_state of state : signal is "recovery";

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' then
                state <= idle;
            else
                state <= next_state;
            end if;
        end if;
    end process;

    process(state, c, d)
    begin
        next_state <= state;
        case state is
            when idle =>
                if c(0) = '1' then
                    next_state <= state0;
                    q <= "0000";
                end if;
            when state0 =>
                if c(0) = '1' and c(1) = '1' then
                    next_state <= state1;
                    q <= d;
```
Chapter 7: XST HDL Coding Techniques

FSM Safe Implementation Verilog Coding Example

Verilog does not provide enumerated types. Because of this, Verilog support for FSM safe implementation is more restrictive than VHDL. Xilinx recommends that you follow these coding guidelines for proper implementation of the state machine:

- Manually enforce the desired encoding strategy.
  - Explicitly define the code value for each valid state.
  - Set FSM Encoding Algorithm (FSM_ENCODING) to User.
- Use `localparam` or `define` for readability to symbolically designate the various states in the state machine description.
- Hard code the recovery state value as one of the following since it cannot be referred to symbolically in a Verilog attribute specification:
  - A string, directly in the attribute statement, or
  - A `define`, as shown in the following coding example

```verilog
module v_safe_fsm (clk, rst, c, d, q);
  input    clk;
  input    rst;
  input [3:0] c;
  input [3:0] d;
  output reg [3:0] q;

  localparam [4:0] idle = 5'b00001,
                   state0 = 5'b00010,
                   state1 = 5'b00100,
                   state2 = 5'b01000,
                   recovery = 5'b10000;

  `define recovery_attr_val "10000"
(* fsm_encoding = "user",

  when state1 ->
    next_state <= state2;
    q <= "1100";

  when state2 ->
    if c(1) == '0' then
      next_state <= state1;
    elsif c(2) == '1' then
      next_state <= state2;
    elsif c(3) == '1' then
      next_state <= idle;
    end if;
    q <= "0101";

  when recovery ->
    next_state <= state0;
    q <= "1111";

  end case;

end process;

end behavioral;
```
safe_implementation = "yes",
safe_recovery_state = "recovery_attr_val ")
// alternatively:  safe_recovery_state = "10000" ")
reg [4:0] state;
reg [4:0] next_state;

always @ (posedge clk)
begin
  if (rst)
    state <= idle;
  else
    state <= next_state;
end

always @(*)
begin

  next_state <= state;

  case (state)
    idle: begin
      if (c[0])
        next_state <= state0;
      q <= 4'b0000;
    end

    state0: begin
      if (c[0] && c[1])
        next_state <= state1;
      q <= d;
    end

    state1: begin
      next_state <= state2;
      q <= 4'b1100;
    end

    state2: begin
      if (~c[1])
        next_state <= state1;
      else
        if (c[2])
          next_state <= state2;
        else
          if (c[3])
            next_state <= idle;
          q <= 4'b0101;
    end

    recovery: begin
      next_state <= state0;
      q <= 4'b0101;
    end

    default: begin
      next_state <= recovery;
      q <= 4'b1111;
    end

  endcase
end
endmodule
FSM Related Constraints

- Automatic FSM Extraction (FSM_EXTRACT)
- FSM Style (FSM_STYLE)
- FSM Encoding Algorithm (FSM_ENCODING)
- Enumerated Encoding (ENUM_ENCODING)
- Safe Implementation (SAFE_IMPLEMENTATION)
- Safe Recovery State (SAFE_RECOVERY_STATE)
## FSM Reporting

The XST log provides detailed information about identified Finite State Machine (FSM) components, and the encoding of each.

* HDL Synthesis *

---

Synthesizing Unit <fsm_1>.
Found 1-bit register for signal <outp>.
Found 2-bit register for signal <state>.
Found finite state machine <FSM_0> for signal <state>.

| States   | 4 |
| Transitions | 5 |
| Inputs    | 1 |
| Outputs   | 2 |
| Clock     | clk (rising_edge) |
| Reset     | reset (positive) |
| Reset type| asynchronous |
| Reset State| s1 |
| Power Up State| s1 |
| Encoding  | gray |
| Implementation | LUT |

---

Summary:
inferred 1 D-type flip-flop(s).
inferred 1 Finite State Machine(s).
Unit <fsm_1> synthesized.

---

HDL Synthesis Report

Macro Statistics

# Registers : 1
1-bit register : 1
# FSMs : 1

---

* Advanced HDL Synthesis *

---

Advanced HDL Synthesis Report

Macro Statistics

# FSMs : 1
# Registers : 1
Flip-Flops : 1
# FSMs : 1

---

* Low Level Synthesis *

---

Optimizing FSM <state> on signal <state[1:2]> with gray encoding.

<table>
<thead>
<tr>
<th>State</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>00</td>
</tr>
<tr>
<td>s2</td>
<td>11</td>
</tr>
<tr>
<td>s3</td>
<td>01</td>
</tr>
<tr>
<td>s4</td>
<td>10</td>
</tr>
</tbody>
</table>
Chapter 7: XST HDL Coding Techniques

FSM Coding Examples
Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

---

FSM Described with a Single Process VHDL Coding Example
---

-- -- State Machine described with a single process
-- -- File: HDL_Coding_Techniques/state_machines/state_machines_1.vhd
--
library IEEE;
use IEEE.std_logic_1164.all;

entity fsm_1 is
  port ( clk, reset, x1 : IN std_logic;
    outp : OUT std_logic);
end entity;

architecture behavioral of fsm_1 is
  type state_type is (s1,s2,s3,s4);
  signal state : state_type;
begin
  process (clk)
  begin
    if rising_edge(clk) then
      if (reset = '1') then
        state <= s1;
        outp <= '1';
      else
        case state is
          when s1 => if x1='1' then
            state <= s2;
            outp <= '1';
          else
            state <= s3;
            outp <= '0';
          end if;
          when s2 => state <= s4; outp <= '0';
          when s3 => state <= s4; outp <= '0';
          when s4 => state <= s1; outp <= '1';
          end case;
      end if;
    end if;
  end process;
end behavioral;
FSM with Three Always Blocks Verilog Coding Example

//
// State Machine with three always blocks.
//
// File: HDL_Coding_Techniques/state_machines/state_machines_3.v
//
module v_fsm_3 (clk, reset, x1, outp);
  input clk, reset, x1;
  output outp;
  reg outp;
  reg [1:0] state;
  reg [1:0] next_state;

  parameter s1 = 2'b00; parameter s2 = 2'b01;
  parameter s3 = 2'b10; parameter s4 = 2'b11;

  initial begin
    state = 2'b00;
  end

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

  always @(state or x1)
    begin
      case (state)
        s1: if (x1==1'b1)
            next_state = s2;
        else
            next_state = s3;
        s2: next_state = s4;
        s3: next_state = s4;
        s4: next_state = s1;
      endcase
    end

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

This section discusses XST HDL Coding Techniques for Black Boxes, and includes:

- About Black Boxes
- Black Boxes Related Constraints
- Black Boxes Reporting
- Black Boxes Coding Examples

About Black Boxes

A design can contain EDIF or NGC files generated by:

- Synthesis tools
- Schematic text editors
- Any other design entry mechanism

These modules must be instantiated in the code in order to be connected to the rest of the design. To do so in XST, use Black Box instantiation in the Hardware Description Language (HDL) source code. The netlist is propagated to the final top-level netlist without being processed by XST. Moreover, XST enables you to apply specific constraints to these Black Box instantiations, which are passed to the NGC file.

In addition, you may have a design block for which you have a Register Transfer Level (RTL) model, as well as your own implementation of this block in the form of an EDIF netlist. The RTL model is valid for simulation purposes only. Use BoxType (BOX_TYPE) to instruct XST to skip synthesis of this RTL model and create a Black Box. The EDIF netlist is linked to the synthesized design during NGDBuild.

For more information, see:

- Chapter 10, XST General Constraints
- Constraints Guide

Once you make a design a Black Box, each instance of that design is a Black Box. While you can apply constraints to the instance, XST ignores any constraint applied to the original design.

For more information on component instantiation, see the VHDL and Verilog language reference manuals.

Black Boxes Related Constraints

BoxType (BOX_TYPE)

BoxType is used for device primitive instantiation in XST. Before using BoxType, see:

Device Primitive Support

Black Boxes Reporting

XST acknowledges a Black Box instantiation as follows during VHDL elaboration:

WARNING:HDLCOMpiler:89 - "example.vhd" Line 15. <my_bbox> remains a black-box since it has no binding entity.

Verilog elaboration issues the following message:

WARNING:HDLCOMpiler:1498 - "example.v" Line 27: Empty module <v_my_block> remains a black box.
When a black box is explicitly designated using a `BoxType (BOX_TYPE)` constraint, XST processes it silently provided the constraint value is `black_box` or `primitive`. When the constraint value is `user_black_box`, a message is issued as follows, for each instantiation of the designated element:

```
Set property "box_type = user_black_box" for instance <my_inst>.
```

Unit `<my_top>` synthesized.

---

## Black Boxes Coding Examples


### Black Box VHDL Coding Example

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

entity black_box_1 is
    port(DI_1, DI_2 : in std_logic;
        DOUT : out std_logic);
end black_box_1;

architecture archi of black_box_1 is
    component my_block
        port (I1 : in std_logic;
                I2 : in std_logic;
                O : out std_logic);
    end component;

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

### Black Box Verilog Coding Example

```
// // Black Box
// // File: HDL_Coding_Techniques/black_box/black_box_1.v
//
module v_my_block (in1, in2, dout);
    input in1, in2;
    output dout;
endmodule

module v_black_box_1 (DI_1, DI_2, DOUT);
    input DI_1, DI_2;
    output DOUT;

    v_my_block inst (
        .in1(DI_1),
        .in2(DI_2),
        .dout(DOUT));
endmodule
```
Chapter 8

XST FPGA Optimization

During Low Level Synthesis, there are several ways to control XST implementation in order to achieve your design goals. During Low Level Synthesis, XST:

1. Separately maps and optimizes each VHDL entity or Verilog module to the targeted device family resources
2. Globally optimizes the complete design

The output of Low Level Synthesis is an NGC netlist file.

Several options and constraints are available to alter the XST default implementation choices.

For more information, see:
Chapter 12, XST FPGA Constraints (Non-Timing)

Mapping Logic to Block RAM

If you cannot fit the design onto the targeted device, place some of the design logic in unused block RAM. Since XST does not automatically decide which logic can be placed in block RAM, you must instruct XST to do so.

1. Isolate the part of the Register Transfer Level (RTL) description to be placed into block RAM in a separate hierarchical block.
2. Apply Map Logic on BRAM (BRAM_MAP) to the separate hierarchical block, either directly in the HDL source code, or in the XST Constraint File (XCF).

The logic implemented in block RAM must satisfy the following criteria:

- All outputs are registered.
- The block contains only one level of registers, which are output registers.
- All output registers have the same control signals.
- The output registers have a synchronous reset signal.
- The block does not contain multi-source situations or tristate buffers.
- Keep (KEEP) is not allowed on intermediate signals.

XST attempts to map the designated logic onto block RAM during Low Level Synthesis. When successful, XST issues the following message:

Entity <logic_bram_1> mapped on BRAM.

If any of the listed requirements is not satisfied, XST does not map the designated logic onto block RAM, and issues a warning.

INFO:Xst:1789 - Unable to map block <no_logic_bram> on BRAM.

Output FF <RES> must have a synchronous reset.
If the logic cannot be placed in a single block RAM primitive, XST spreads it over several block RAMs.

**Flip-Flop Implementation Guidelines**

Starting with the Virtex®-6 and Spartan®-6 and device families, CLB flip-flops and latches can no longer natively implement both a set and reset. XST enforces the following rules if it finds a flip-flop with both a set and reset, whether the flip-flop is inferred, or retargeted from an older device family primitive instantiation:

- A simultaneous synchronous set and reset is retargeted and additional logic is created.
- A simultaneous asynchronous set and reset is rejected with the following error message.

```plaintext
ERROR:Xst:#### - This design infers one or more latches or registers with both an active asynchronous set and reset. In the Virtex6 and Spartan6 architectures this behaviour creates a sub-optimal circuit in area, power and performance. To synthesis an optimal implementation it is highly recommended to either remove one set or reset or make the function synchronous. To override this error set -retarget_active_async_set_reset option to yes.
```

Follow these additional guidelines:

- Do not set or reset registers asynchronously. Use synchronous initialization instead. Although supported on Xilinx® devices, Xilinx does not recommend this practice for the following reasons:
  - Control set remapping is no longer possible
  - Sequential functionality in several device resources, such as block RAMs and DSP blocks, can only be set or reset synchronously. You will either be unable to leverage those resources, or they will be configured in a suboptimal way.

- If your coding guidelines call for registers to be set or reset asynchronously, consider running XST with Asynchronous to Synchronous (ASYNC_TO_SYNC). This allows you to assess the potential benefits of moving to a synchronous set/reset approach. Asynchronous to Synchronous (ASYNC_TO_SYNC) affects only inferred registers. It does not affect instantiated flip-flops.

- Do not describe flip-flops with both a set and a reset. Starting with the Virtex-6 and Spartan-6 device families, none of the available flip-flop primitives natively features both a set and a reset, whether synchronous or asynchronous. XST rejects flip-flops described with both an asynchronous reset and an asynchronous set.

- Avoid operational set/reset logic whenever possible. There may be other, less expensive, ways to achieve the desired result, such as taking advantage of the circuit global reset by defining an initial contents.

- The clock enable, set and reset control inputs of Xilinx flip-flop primitives are always active-High. If described to be active-Low, such functionality inevitably leads to inverter logic that penalizes circuit performance.
Flip-Flop Retiming

Flip-flop retiming consists of moving flip-flops and latches across logic in order to reduce synchronous paths, thereby increasing clock frequency. This optimization is disabled by default.

Flip-flop retiming can be either forward or backward:
- Forward retiming moves a set of flip-flops that are the input of a LUT to a single flip-flop at its output.
- Backward retiming moves a flip-flop that is at the output of a LUT to a set of flip-flops at its input.
- Backward flip-flop retiming generally increases the number of flip-flop, sometimes significantly.
- Forward flip-flop retiming generally reduces the number of flip-flops.

In either case, the behavior of the design is not changed. Only timing delays are modified.

Flip-flop retiming is part of global optimization. It respects the same constraints as all other optimization techniques. Since retiming is incremental, a flip-flop that is the result of a retiming can be moved again in the same direction (forward or backward) if it results in better timing. Retiming iterations stop when specified timing constraints are satisfied, or if no more timing improvement can be obtained.

For each flip-flop moved, a message specifies:
- The original and new flip-flop names
- Whether it is a forward or backward retiming

Limitations of Flip-Flop Retiming

Flip-flop retiming does not take place under the following circumstances:
- Flip-flop retiming is not applied to flip-flops with an IOB=TRUE property.
- Forward retiming does not take place if a flip-flop or the signal on its output has a Keep (KEEP) property.
- Backward retiming does not take place if the signal on the input of a flip-flop has a Keep (KEEP) property.
- Instantiated flip-flops are moved only if Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) is set to yes.
- Flip-Flops are moved across instantiated primitives only if Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) is set to yes.
- Flip-flops with both a set and a reset are not moved.

Controlling Flip-Flop Retiming

Use the following constraints to control flip-flop retiming:
- Register Balancing (REGISTER_BALANCING)
- Move First Stage (MOVE_FIRST_STAGE)
- Move Last Stage (MOVE_LAST_STAGE)
Speed Optimization Under Area Constraint

The Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) constraint:

- Can be used to achieve some degree of control over circuit performance even when instructing XST to target area reduction as its main goal.
- Is set by default to 100% of the selected device size.
- Influences low level optimization as follows:
  - As long as the estimated area is higher than the constraint requirement, XST tries to further reduce area.
  - When the estimated area falls within the constraint requirement, XST starts to look for timing optimization opportunities, making sure that the solution stays within the area constraint requirement.
- Does not control macro inference.

Low Level Synthesis Report Example One (100%)

In the following example, the target area constraint was set to 100%. The initial area estimation found an actual device utilization of 102%. XST began optimization and reached 95%.

* Low Level Synthesis

* Found area constraint ratio of 100 (+ 5) on block tge, actual ratio is 102.
  * Optimizing block tge> to meet ratio 100 (+ 5) of 1536 slices
  * Area constraint is met for block tge>, final ratio is 95.

Low Level Synthesis Report Example Two (70%)

If the area constraint cannot be met, XST ignores it during timing optimization, and runs low level synthesis to achieve the best frequency. In the following example, the target area constraint was set to 70%. Because XST was unable to satisfy it, XST issued the following warning:

* Low Level Synthesis

* Found area constraint ratio of 70 (+ 5) on block fpga_hm, actual ratio is 64.
  * Optimizing block fpga_hm> to meet ratio 70 (+ 5) of 1536 slices
  * WARNING: Xst - Area constraint could not be met for block tge>, final ratio is 94

(+/5) represents the max margin of the area constraint. If the area constraint is not met, but the difference between the requested area and actual area, achieved during area optimization, is less or equal then 5%, then XST runs timing optimization taking into account the achieved area, and making sure that the final area solution does not exceed that figure.

Low Level Synthesis Report Example Three

In the following example, the area target was set to 55%. XST achieved only 60%. But taking into account that the difference between requested and achieved area is not more than 5%, XST considers that the area constraint was met, and ensures that it is not broken by further optimizations.

* Low Level Synthesis

* Found area constraint ratio of 55 (+ 5) on block fpga_hm, actual ratio is 64.
  * Optimizing block fpga_hm> to meet ratio 55 (+ 5) of 1536 slices
  * Area constraint is met for block fpga_hm>, final ratio is 60.
In some situations, it is important to disable automatic resource management. To do so, specify -1 as the value for `SLICE_UTILIZATION_RATIO`. Slice (LUT-FF Pairs) Utilization Ratio (`SLICE_UTILIZATION_RATIO`) can be applied to a specific block of the design. You can specify an absolute number of slices (or FF-LUT pairs), or a percentage of the total number available on the device.

**Implementation Constraints**

XST writes all implementation constraints found in the Hardware Description Language (HDL) source code or in an XST Constraint File (XCF) to the output NGC file. Keep (KEEP) properties are generated during buffer insertion for maximum fanout control or for optimization.
Xilinx Device Primitive Support

This section discusses Xilinx® Device Primitive Support, and includes:

- About Xilinx Device Primitive Support
- Generating Primitives Through Attributes
- Primitives and Black Boxes
- VHDL and Verilog Xilinx Device Primitives Libraries
- Specifying Primitive Properties
- Reporting of Instantiated Device Primitives
- Primitives Related Constraints
- Primitives Coding Examples

About Device Primitive Support

XST allows you to instantiate any Xilinx® device primitive directly in the Hardware Description Language (HDL) source code. These primitives are:

- Pre-compiled in the UNISIM library.
- Not optimized or changed by XST by default.
- Preserved by XST and made available in the final NGC netlist.

Use **Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)** to let XST try to optimize instantiated primitives with the rest of the design. Timing information is available for most of the primitives, allowing XST to perform efficient timing-driven optimizations.

In order to simplify instantiation of complex primitives such as RAMs, XST supports an additional library called UniMacro.

For more information, see the *Libraries Guides*.

Generating Primitives Through Attributes

Some primitives can be generated through attributes:

- **Buffer Type (BUFFER_TYPE)** can be assigned to the circuit primary I/Os or to internal signals to force the use of a specific buffer type. The same constraints can be used to disable buffer insertion.

- **I/O Standard (IOSTANDARD)** can be used to assign an I/O standard to an I/O primitive. For example, the following assigns **PCI33_5 I/O** standard to the I/O port:

```vhdl
// synthesis attribute IOSTANDARD of in1 is PCI33_5
```

Primitives and Black Boxes

Primitive support is based on the concept of the black box.

For information on the basics of black box support, see:

*Finite State Machine (FSM) Safe Implementation*

There is a significant difference between black box and primitive support. Assume a design with a submodule called **MUXF5**. In general, the **MUXF5** can be your own functional block or a Xilinx® device primitive. To avoid confusion about how XST interprets this module, attach **Box Type (BOX_TYPE)** to the component declaration of **MUXF5**.
If BoxType (BOX_TYPE) is applied to the MUXF5 with a value of:

- **primitive** or **black_box**
  
  XST tries to interpret this module as a Xilinx device primitive and use its parameters, for instance, in critical path estimation.

- **user_black_box**
  
  XST processes it as a regular user black box.

If the name of the **user_black_box** is the same as that of a Xilinx device primitive, XST renames it to a unique name and issues a warning. For example, MUX5 could be renamed to MUX51.

**WARNING:**

- Xst:79 - Model ‘muxf5’ has different characteristics in destination library
- Xst:80 - Model name has been changed to ‘muxf51’

If BoxType (BOX_TYPE) is not applied to the MUXF5, XST processes this block as a user hierarchical block. If the name of the **user_black_box** is the same as that of a Xilinx device primitive, XST renames it to a unique name and issues a warning.
Chapter 8: XST FPGA Optimization

VHDL and Verilog Xilinx Device Primitives Libraries

This section discusses VHDL and Verilog Xilinx® Device Primitives Libraries, and includes:

- About VHDL and Verilog Xilinx Device Primitives Libraries
- VHDL Xilinx Device Primitives Device Libraries
- Verilog Device Primitives Device Libraries
- Primitive Instantiation Guidelines

About VHDL and Verilog Xilinx Device Primitives Libraries

XST provides dedicated VHDL and Verilog libraries to simplify instantiation of Xilinx® device primitives in the Hardware Description Language (HDL) source code. These libraries contain the complete set of Xilinx device primitives declarations with BoxType (BOX_TYPE) applied to each component. If you have properly included those libraries, you need not apply BoxType (BOX_TYPE) itself.

VHDL Xilinx® Device Primitives Device Libraries

In VHDL, declare library UNISIM with its package vcomponents in the Hardware Description Language (HDL) source code.

```vhdl
library unisim;
use unisim.vcomponents.all;
```

The HDL source code for this package is located in the following file of the XST installation:

```vhdl
vhdl\src\unisims\unisims_vcomp.vhd
```

Verilog Device Primitives Device Libraries

In Verilog, the UNISIM library is precompiled. XST automatically links it with your design.

Primitive Instantiation Guidelines

Use UPPERCASE for generic (VHDL) and parameter (Verilog) values when instantiating primitives. For example, the ODDR element has the following component declaration in the UNISIM library:

```vhdl
component ODDR
 generic (
     DDR_CLK_EDGE : string := "OPPOSITE_EDGE";
     INIT : bit := '0';
     SRTYPE : string := "SYNC";
 )
port(
    Q : out std_ulogic;
    C : in std_ulogic;
    CE : in std_ulogic;
    D1 : in std_ulogic;
    D2 : in std_ulogic;
    R : in std_ulogic;
    S : in std_ulogic);
end component;
```

When you instantiate this primitive, the values of DDR_CLK_EDGE and SRTYPE generics must be in UPPERCASE. If not, XST issues a warning stating that unknown values are used.
Some primitives, such as LUT1, enable you to use an INIT during instantiation. The two ways to pass an INIT to the final netlist are:

- Apply INIT to the instantiated primitive.
- Pass INIT with the generics mechanism (VHDL) or the parameters mechanism (Verilog).

This allows you to use the same code for synthesis and simulation.

## Specifying Primitive Properties

Use VHDL generics or Verilog parameters to specify properties on instantiated primitives, such as the INIT of an instantiated LUT.

**Caution!** In order to avoid potential simulation mismatches, XST prevents you from doing so by means of attributes specified in the Hardware Description Language (HDL) source code, or through XST Constraint File (XCF) constraints. In previous releases, a simple warning was advising you against this practice. This warning has now become an error, as illustrated below. Overriding default values of instantiated primitives properties can now only be done through a VHDL generic or Verilog parameter.

**ERROR:** Xst:3003 - "example.vhd". Line 77. Unable to set attribute "A_INPUT" with value "CASCADE" on instance <idsp> of block <DSP48E1>. This property is already defined with value "DIRECT" on the block definition by a VHDL generic or a Verilog parameter. Apply the desired value by overriding the default VHDL generic or Verilog parameter. Using an attribute is not allowed.

Simulation tools recognize generics and parameters, simplifying the circuit validation process.

### Configuring a LUT2 Primitive INIT Property VHDL Coding Example

```
--
-- Instantiating a LUT2 primitive
-- Configured via the generics mechanism (recommended)
--
-- File: FPGA_Optimization/primitive_support/primitive_2.vhd
--
library ieee;
use ieee.std_logic_1164.all;
library unisim;
use unisim.vcomponents.all;

entity primitive_2 is
    port(I0,I1 : in std_logic;
         O : out std_logic);
end primitive_2;

architecture beh of primitive_2 is
begin
    inst : LUT2
        generic map (INIT=>'1')
        port map (I0=>I0, I1=>I1, O=>O);
end beh;
```
Configuring a LUT2 Primitive INIT Property Verilog Coding Example

//
// Instantiating a LUT2 primitive
// Configured via the parameter mechanism
//
// File: FPGA_Optimization/primitive_support/primitive_2.v
//
module v_primitive_2 (I0,I1,O);
    input I0, I1;
    output O;

    LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));
endmodule

Reporting of Instantiated Device Primitives

XST processes instantiated device primitives silently, because BoxType (BOX_TYPE) with its value, primitive, is applied to each primitive in the UNISIM library.

XST issues a warning if either of the following two conditions occurs.

First Warning Condition

- You instantiate a block (non primitive).
  AND
- The block has no contents (no logic description).

Second Warning Condition

- The block has a logic description.
  AND
- You apply BoxType (BOX_TYPE) to it with a value of user_black_box.

Warning Example

Elaborating entity <example> (architecture <archi>) from library <work>.
WARNING:HDLC Compiler:89 - "example.vhd" Line 15: <my_block> remains a black-box since it has no binding entity.

Primitives Related Constraints

- BoxType (BOX_TYPE)
- Constraints for placement and routing that can be passed from Hardware Description Language (HDL) to NGC without any specific XST processing

Primitives Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
Instantiating and Configuring a LUT2 Primitive with a Generic VHDL Coding Example

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

entity primitive_2 is
  port(I0, I1 : in std_logic;
       O : out std_logic);
end primitive_2;

architecture beh of primitive_2 is
begin
  inst : LUT2
    generic map (INIT=>'1')
    port map (I0=>I0, I1=>I1, O=>O);
end beh;
```

Instantiating and Configuring a LUT2 Primitive with a Parameter Verilog Coding Example

```verilog
// input I0, I1;
// output O;
//
// LUT2 #(4'h1) inst (.I0(I0), .I1(I1), .O(O));
//
endmodule
```

Instantiating and Configuring a LUT2 Primitive with a Defparam Verilog Coding Example

```verilog
// input I0, I1;
// output O;
//
// LUT2 inst (.I0(I0), .I1(I1), .O(O));
// defparam inst.INIT = 4'h1;
//
endmodule
```
Using the UniMacro Library

In order to simplify instantiation of such complex primitives as RAMs, XST supports an additional library called UniMacro.

For more information, see the Libraries Guides.

In VHDL, declare library unimacro with its package vcomponents.

```vhdl
library unimacro;
use unimacro.vcomponents.all;
```

The HDL source code of this package can be found in the following file in the Xilinx® software installation:

```
vhdl\src\unisims\unisims_vcomp.vhd
```

In Verilog, the UniMacro library is precompiled. XST automatically links it with your design.
Cores Processing

This section discusses Cores Processing, including Loading Cores, Finding Cores, and Cores Reporting.

Loading Cores

If a design contains cores in the form of EDIF or NGC netlist files, XST can automatically read them for more accurate timing estimation and resource utilization control.

To enable or disable this feature:

- In ISE® Design Suite, select Process > Properties > Synthesis Options > Read Cores.
- In command line mode, use -read_cores.

In this case, an additional optimize value allows XST to integrate the core netlist into the overall design, and try to optimize it.

XST reads cores by default.

Finding Cores

XST automatically finds cores in the ISE® Design Suite project directory. If the cores are located elsewhere, specify the path as follows:

- In ISE Design Suite, select:
  Process > Properties > Synthesis Options > Core Search Directories
- In command line mode, use:
  -sd

Xilinx® recommends that you systematically specify the directories in which the cores reside, and that you keep this information up to date. In addition to better timing and resource estimation, doing so can protect you against unexpected behaviors and hard-to-debug situations.

For example, without knowing the contents of an unloaded core (seen as a black box), XST may have difficulty determining adequate buffer insertions on paths leading to that core. This can negatively impact timing closure.

Cores Reporting

========================================================================
* Low Level Synthesis *
========================================================================
Launcher: Executing edif2ngd -noa "my_add.edn" "my_add.ngo"
INFO:NgdBuild - Release 11.2 - edif2ngd
INFO:NgdBuild - Copyright (c) 1995-2010 Xilinx, Inc. All rights reserved.
Writing the design to "my_add.ngo"...
Loading core <my_add> for timing and area information for instance <inst>.
========================================================================
Mapping Logic to LUTs

Use the UNISIM library to directly instantiate LUT components in the Hardware Description Language (HDL) source code. To specify a function that a LUT must execute, apply INIT to the instance of the LUT. To place an instantiated LUT or register in a particular slice of the chip, attach RLOC to the same instance.

It is not always convenient to calculate INIT functions. Other methods can be used. Alternatively, you can describe the function that you want to map onto a single LUT in the HDL source code in a separate block. Attaching Map Entity on a Single LUT (LUT_MAP) to this block tells XST that this block must be mapped on a single LUT. XST automatically calculates the INIT value for the LUT and preserves this LUT during optimization. For more information, see Map Entity on a Single LUT (LUT_MAP).

XST automatically recognizes the Synplify xc_map attribute.

If a function cannot be mapped on a single LUT, XST errors out.

ERROR:Xst:1349 - Failed to map xcmap entity <v_and_one> in one lut.

Mapping Logic to LUTs Verilog Coding Example

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
In the following example, the top block instantiates two AND gates, respectively described in blocks and_one and and_two. XST generates two LUT2s and does not merge them.

```vhdl
//
// Mapping of Logic to LUTs with the LUT_MAP constraint
// Mapped to 2 distinct LUT2s
// Mapped to 1 single LUT3 if LUT_MAP constraints are removed
//
// File: FPGA_Optimization/lut_mapping/lut_map_1.v
//
(* LUT_MAP="yes" *)
module v_and_one (A, B, REZ);
    input A, B;
    output REZ;
    and and_inst(REZ, A, B);
endmodule

(* LUT_MAP="yes" *)
module v_and_two (A, B, REZ);
    input A, B;
    output REZ;
    or or_inst(REZ, A, B);
endmodule

(* LUT_MAP="yes" *)
module v_lut_map_1 (A, B, C, REZ);
    input A, B, C;
    output REZ;
    wire tmp;
    v_and_one inst_and_one (A, B, tmp);
    v_and_two inst_and_two (tmp, C, REZ);
endmodule
```
Controlling Placement on the Device

You can control placement of the following inferred macros to a specific location on the targeted device:

- Registers
- Block RAMs

To do so, apply RLOC to the signal modeling the register or the RAM, as shown in the following coding examples. When applied on a register, XST distributes the constraint to each flip-flop, and propagates RLOC constraints to the final netlist. RLOC is supported for inferred RAMs that can be implemented with a single block RAM primitive.

RLOC Constraint on a 4-Bit Register VHDL Coding Example

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

The following coding example specifies an RLOC constraint on a 4-bit register:

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

entity inits_rlocs_3 is
  port (CLK : in std_logic;
        DI : in std_logic_vector(3 downto 0);
        DO : out std_logic_vector(3 downto 0));
end inits_rlocs_3;

architecture beh of inits_rlocs_3 is
  signal tmp: std_logic_vector(3 downto 0):="1011";
  attribute RLOC: string;
  attribute RLOC of tmp: signal is "X3Y0 X2Y0 X1Y0 X0Y0";
begin
  process (CLK)
  begin
    if clk'event and clk='1' then
      tmp <= DI;
    end if;
  end process;
  DO <= tmp;
end beh;
```
Inserting Buffers

XST automatically inserts clock and I/O buffers. Insertion of I/O buffers can be enabled or disabled with Add I/O Buffers (-iobuf) It is enabled by default.

You can also manually instantiate clock and I/O buffers. XST does not change instantiated device primitives, but propagates them to the final netlist.

Using the PCI Flow With XST

To satisfy placement constraints and meet timing requirements when using the PCI™ flow with XST:

- For VHDL, ensure that the names in the generated netlist are all in uppercase. The default case is lower. Specify the case in ISE® Design Suite in:
  
  Process > Properties > Synthesis Options > Case

- For Verilog, ensure that the case is set to maintain. The default is maintain. Specify the case in ISE Design Suite in:
  
  Process > Properties > Synthesis Options > Case

- Preserve the hierarchy of the design. Specify the Keep Hierarchy (KEEP_HIERARCHY) setting in ISE Design Suite in:
  
  Process > Properties > Synthesis Options > Keep Hierarchy

- Preserve equivalent flip-flops. XST removes equivalent flip-flops by default. Specify the Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) setting in ISE Design Suite in:
  
  Process > Properties > Xilinx Specific Options > Equivalent Register Removal

Preventing Logic and Flip-Flop Replication

To prevent logic and flip-flop replication caused by a high fanout flip-flop set/reset signal:

- Set a high maximum fanout value for the entire design in ISE® Design Suite in:
  
  Process > Properties > Xilinx Specific Options > Max Fanout

OR

- Use Max Fanout (MAX_FANOUT) to set a high maximum fanout value for the initialization signal connected to the RST port of the PCI™ core, as shown in the following example:
  
  max_fanout=2048

Disabling Read Cores

Disabling Read Cores (READ_CORES) prevents XST from automatically loading the PCI™ cores for timing and area estimation. When reading PCI cores, XST may perform logic optimizations that do not allow the design to meet timing requirements, or which might lead to errors during MAP. To disable Read Cores (READ_CORES), uncheck it in ISE® Design Suite in Process > Properties > Synthesis Options > Read Cores.

By default, XST reads cores for timing and area estimation.
Chapter 9

XST Design Constraints

Constraints can help you meet design goals and obtain the best circuit implementation. Constraints control various aspects of synthesis, as well as placement and routing. Default synthesis algorithms and heuristics have been tuned to provide the best possible results for a large variety of designs. However, if synthesis initially fails to deliver satisfying results, use those constraints to try other, non-default, synthesis alternatives.

Specifying Constraints

Use the following mechanisms to specify constraints:

• Set options to provide global control on most synthesis aspects. You can set options in:
  – ISE® Design Suite
  – The run command in command line mode

  For more information, see:
  Running XST in Command Line Mode

• Directly insert VHDL attributes in VHDL code, and apply them to individual elements of the design. This allows you to control, not only synthesis, but also placement and routing.

• Add constraints as Verilog attributes (preferred) or Verilog meta comments.

• Specify constraints in a separate constraint file.

  Caution! VHDL attributes, Verilog attributes, and XST Constraint File (XCF) constraints are not allowed mechanisms for defining the properties of instantiated device primitives. You must use VHDL generics or Verilog parameters to do so.

  For more information, see:
  Specifying Primitive Properties

Global synthesis settings are typically defined in ISE Design Suite, or from the XST command line. VHDL and Verilog attributes and Verilog meta comments can be inserted into the Hardware Description Language (HDL) source code to specify different choices for individual parts or elements of the design. See Constraints Precedence Rules to understand how the tool determines which constraint applies when set from different sources, and on different HDL objects.
Chapter 9: XST Design Constraints

Constraints Precedence Rules

As a general rule, the local specification of a constraint overrides any other specification that applies more globally. For example, if a constraint is set both on a signal (or an instance) and on the design unit that contains it, the former takes precedence for that signal (or instance). Similarly, a constraint applied to a signal (or an instance, or even an entity or module), takes precedence over its specification on the XST command line, or through ISE® Design Suite.

If a constraint is applied to the same object using different entry methods, the following precedence applies, from the highest to the lowest priority:

1. XST Constraint File (XCF)
2. Hardware Description Language (HDL) attribute
3. ISE Design Suite in Process > Properties, or the command line

Synthesis Options in ISE Design Suite

This section discusses Synthesis Options in ISE® Design Suite, including Setting XST Options in ISE Design Suite, Setting Other XST Command Line Options, and Design Goals and Strategies.

Setting XST Options in ISE Design Suite

To set XST options in ISE® Design Suite:

1. Select a Hardware Description Language (HDL) source file from the Hierarchy panel of the Design window.
   a. Right-click Synthesize-XST in the Processes panel.
   b. Select Process > Properties.
   c. Select a category:
      • Synthesis Options
      • HDL Options
      • Xilinx Specific Options
2. Set the Property display level to:
   a. Standard to see the most common options
   b. Advanced to see all available options
3. Check Display switch names to see the corresponding command-line switch name for each option.

To revert to the XST default options, click Default.

Setting Other XST Command Line Options

In addition to the default options listed in the Process > Properties window, you can specify any other unlisted XST command line options.

1. Go to Process > Properties.
2. Select Synthesis Options.
3. In Other XST Command Line Options, add the desired command line options in the corresponding Value field. Separate multiple options with a space.

Follow the syntax described in:

XST Commands
Design Goals and Strategies

ISE® Design Suite features predefined goals and strategies that allow you to run the software, including XST, with specific options settings that have been tuned for particular optimization goals. This approach may be a good alternative for trying non-default constraints settings, without having to go too much into the details of all XST constraints.

To create and save your own design goals and strategies, select Project > Design Goals & Strategies.

VHDL Attributes

Use VHDL attributes to describe constraints directly in the Hardware Description Language (HDL) source code. Before it can be used, you must declare an attribute as follows:

```vhdl
attribute AttributeName : Type;
```

**VHDL Attribute Syntax Example**

```vhdl
attribute RLOC : string;
```

The attribute type defines the type of the attribute value. The only allowed type for XST is string.

An attribute can be declared in an entity or architecture.

- If 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:

```vhdl
attribute AttributeName of ObjectList : ObjectType is AttributeValue;
```

**VHDL Attribute Example**

```vhdl
attribute RLOC of u123 : label is “R11C1.S0”;
attribute bufg of my_signal : signal is “sr”;
```

The object list is a comma separated list of identifiers. Accepted object types are:

- entity
- architecture
- component
- label
- signal
- variable
- type

If a constraint can be applied on a VHDL entity, it can also be applied on the component declaration.
Verilog-2001 Attributes

XST supports Verilog-2001 attribute statements. Attributes pass specific information to applications such as synthesis tools. You can specify Verilog-2001 attributes anywhere for operators or signals within module declarations and instantiations. Although the compiler may support other attribute declarations, XST ignores them.

Use Verilog attributes to:

- Set constraints on individual objects such as:
  - modules
  - instances
  - nets
- Set the following specific synthesis constraints:
  - Full Case (FULL_CASE)
  - Parallel Case (PARALLEL_CASE)

Verilog-2001 Syntax

Verilog-2001 inline attributes are enclosed between (*) tokens, and use the following general syntax:

(* attribute_name = attribute_value *)

where

- The attribute_value is a string. No integer or scalar values are allowed.
- The attribute_value is enclosed between quotes.
- The default value is 1. (* attribute_name *) is equivalent to (* attribute_name = "1" *).

Place the attribute immediately before the signal, module, or instance declaration to which it refers. This can be done on a separate line, as follows:

(* ram_extract = "yes" *)
reg [WIDTH-1:0] myRAM [SIZE-1:0];

The attribute can also be placed on the same line as the declaration. For example:

(* ram_extract = "yes" *) reg [WIDTH-1:0] myRAM [SIZE-1:0];

A comma-separated list of several attributes may be specified as follows. These attributes will be attached to the same Verilog object.

(* attribute_name1 = attribute_value1, attribute_name2 = attribute_value2 *)

The following style is also acceptable:

(* attribute_name1 = attribute_value1 *) (*attribute_name2 = attribute_value2 *)

For improved readability, the attribute list may span on multiple lines. For example:

(*
    ram_extract = "yes",
    ram_style = "block"
*)
reg [WIDTH-1:0] myRAM [SIZE-1:0];

Verilog-2001 Attribute Coding Example

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.
The following coding example illustrates various ways to specify attributes in Verilog, attaching one or several properties respectively to a module, to a port, and to internal signals. In addition, full_case and parallel_case directives are also attached to a case construct using the attribute syntax.

```verilog
// Verilog 2001 attribute examples
// File: Verilog_Language_Support/attributes/vlgattrib2001_1.v
//
// (* mux_extract = "no" *)
module vlgattrib2001_1 (clk, we1, we2, sel, re1, re2, waddr, raddr, di, do);

// (* max_fanout = "100", buffer_type = "none" *) input clk;
input [1:0] sel;
input we1, we2;
input re1, re2;
input [7:0] waddr;
input [7:0] raddr;
input [15:0] di;
output reg [15:0] do;

// (* mux_extract = "yes", use_clock_enable = "no" *)
reg re;

// (* ram_extract = "yes", ram_style = "block" *)
reg [15:0] RAM [255:0];

// (* keep = "true" *) wire we;
assign we = we1 | we2;
always @ (posedge clk)
bEGIN
  (* full_case *) (* parallel_case *)
  case (sel)
    2'b00 : re <= re1 & re2;
    2'b01 : re <= re1 | re2;
    2'b10 : re <= re1;
    2'b11 : re <= re2;
  endcase
end
always @ (posedge clk)
bEGIN
  if (we)
    RAM[waddr] <= di;
  if (re)
    do <= RAM[raddr];
end
endmodule
```

**Verilog-2001 Limitations**

Verilog-2001 attributes are not supported for:

- Signal declarations
- Statements
- Port connections
- Expression operators
Verilog Meta Comments

You can specify constraints in Verilog code using meta comments. Xilinx® recommends that you use Verilog-2001 attribute syntax. The Verilog meta comment syntax is:

// synthesis attribute AttributeName [of] ObjectName [is] AttributeValue

**Verilog Meta Comment Syntax Examples**

// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
// synthesis attribute bufg of my_clock is "clk"

The following constraints use a different syntax:

- Full Case (FULL_CASE)
- Parallel Case (PARALLEL_CASE)
- Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)

For more information, see:

Verilog 2001 Attributes and Meta Comments
Chapter 9: XST Design Constraints

XST Constraint File (XCF)

This section discusses the XST Constraint File (XCF), and includes:

- About the XCF
- Native and Non-Native UCF Syntax
- Syntax Limitations
- Timing Constraints Applicable Only Through the XCF

About the XCF

Rather than specifying XST constraints in the Hardware Description Language (HDL) source code, you can specify them in the XST Constraint File (XCF). The XCF file has an extension of ".xcf".

To specify an XCF file in ISE® Design Suite:
1. Select an HDL source file from **Design > Hierarchy**.
2. Right-click **Processes > Synthesize-XST**.
3. Select **Process > Properties**.
4. Select **Synthesis Options**.
5. Edit **Synthesis Constraints File**.
6. Check **Synthesis Constraints File**.

To specify the XCF in command line mode, use **Synthesis Constraint File (-uc)** with the **run** command.

For more information about the **run** command and running XST from the command line, see **XST Commands**.

The XCF syntax enables you to specify constraints that are applicable to:

- The entire design
- Specific entities or modules

The XCF syntax is an extension of the User Constraints File (UCF) syntax. You apply constraints to nets or instances in the same manner. In addition, the XCF syntax allows constraints to be applied to specific levels of the design hierarchy. Use the keyword **MODEL** to define the entity or module to which the constraint is applied. If a constraint is applied to an entity or module, the constraint is effective for each instantiation of the entity or module.

Define constraints in ISE Design Suite in **Process > Properties**, or the XST **run** command on the command line. Specify exceptions in the XCF file. The constraints specified in the XCF file are applied only to the module listed, and not to any submodules below it.

To apply a constraint to the entire entity or module use the following syntax:

```
MODEL entityname constraintname = constraintvalue;
```

Coding Example

```
MODEL top mux_extract = false;
MODEL my_design max_fanout = 256;
```

If the entity **my_design** is instantiated several times in the design, the **max_fanout=256** constraint is applied to each instance of **my_design**.

To apply constraints to specific instances or signals within an entity or module, use the **INST** or **NET** keywords. XST does not support constraints that are applied to VHDL variables.
The syntax is:

\[
\text{BEGIN MODEL entityname} \\
\text{INST instancename constraintname = constraintvalue ;} \\
\text{NET signalname constraintname = constraintvalue ;} \\
\text{END;}
\]

Syntax Example

BEGIN MODEL crc32 
  INST stopwatch opt_mode = area ; 
  INST U2 ram_style = block ; 
  NET myclock clock_buffer = true ; 
  NET data_in iob = true ; 
END;

Native and Non-Native UCF Syntax

All XST-supported constraints can be divided into two groups, Native UCF Constraints and Non-Native UCF Constraints.

Native UCF Constraints

Only Timing constraints and Area Group constraints use native User Constraints File (UCF) syntax. Use this syntax, including wildcards and hierarchical names, for such native UCF constraints as:

- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)

Do not use these constraints inside a BEGIN MODEL... END construct. If you do so, XST issues an error.

Non-Native UCF Constraints

For all non-native User Constraints File (UCF) constraints, use the MODEL or BEGIN MODEL... END; constructs.

Non-native UCF constraints include:

- Pure XST constraints such as:
  - Automatic FSM Extraction (FSM_EXTRACT)
  - RAM Style (RAM_STYLE)
- Implementation non-timing constraints such as:
  - RLOC
  - Keep (KEEP)

In XST, the default hierarchy separator is a forward slash (/). Use this separator when specifying timing constraints that apply to hierarchical instance or net names in the XST Constraint File (XCF). Use Hierarchy Separator (-hierarchy_separator) to change the hierarchy separator inserted by XST.
Chapter 9: XST Design Constraints

Syntax Limitations

XST Constraint File (XCF) syntax has the following limitations:

- Nested model statements are not supported.
- Instance or signal names listed between the BEGIN MODEL statement and the END statement are only the ones visible inside the entity. Hierarchical instance or signal names are not supported.
- Wildcards in instance and signal names are not supported, except in timing constraints.
- Not all native User Constraints File (UCF) constraints are supported.

For more information, see the Constraints Guide.

Timing Constraints Applicable Only Through the XCF

The following timing constraints can be applied for synthesis only through the XST Constraint File (XCF):

- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)
- Timing Specifications (TIMESPEC)

  See the Constraints Guide.

- Timing Specification Identifier (TSidentifier)

  See the Constraints Guide.

These timing constraints are not propagated exclusively to implementation tools. They are also understood by XST, and influence synthesis optimization. To pass these constraints to Place and Route (PAR), use Write Timing Constraints (~write_timing_constraints).

For more information as to the value and target of each constraint, see the Constraints Guide.
Add I/O Buffers (–iobuf)

Add I/O Buffers (–iobuf) enables or disables I/O buffer insertion. XST automatically inserts Input/Output Buffers into the design. If you manually instantiate I/O Buffers for some or all the I/Os, XST inserts I/O Buffers only for the remaining I/Os. To prevent XST from inserting any I/O Buffers, set -iobuf to no. Add I/O Buffers is useful for synthesizing a part of a design to be instantiated later.

The values for this constraint are:

- yes (default)
- no

When yes is selected, IBUF and IOBUF primitives are generated. IBUF and OBUF primitives are connected to I/O ports of the top-level module. When XST synthesizes an internal module that is instantiated later in a larger design, you must select no. If I/O buffers are added to a design, the design cannot be used as a submodule of another design.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Applies to design primary I/Os

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```bash
-iobuf {yes|no|true|false|soft}
```

The default is yes.
ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Add I/O Buffers
BoxType (BOX_TYPE)

The Box Type (BOX_TYPE) constraint:

- Is a synthesis constraint.
- Instructs XST not to synthesize the behavior of a module.
- Has the following values:
  - **primitive**
    XST does *not* report inference of a black box in the log file.
  - **black_box**
    Equivalent to **primitive**. This value will eventually become obsolete.
  - **user_black_box**
    XST *does* report inference of a black box in the log file.

If Box Type is applied to at least a single instance of a block of a design, Box Type is propagated to all other instances of the entire design. This feature now supports Verilog and the XST Constraint File (XCF) similar to VHDL, where Box Type can be applied to a component.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to the following design elements:

- VHDL component, entity
- Verilog module, instance
- XCF model, instance

Propagation Rules

Applies to the design element to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute box_type: string;
```

Specify as follows:

```vhdl
attribute box_type of {component_name|entity_name} :
{component|entity} is "(primitive|black_box|user_black_box)";
```
Verilog Syntax Example
Place immediately before the instantiation:

(* box_type = "{primitive|black_box|user_black_box}" *)

XCF Syntax Example One

MODEL "entity_name"
box_type="{primitive|black_box|user_black_box}";

XCF Syntax Example Two

BEGIN MODEL "entity_name"
INST "instance_name"
box_type="{primitive|black_box|user_black_box}"; END;
Bus Delimiter (–bus_delimiter)

Bus Delimiter (–bus_delimiter) defines the format used to write the signal vectors in the result netlist. The available formats are:

- <> (default)
- []
- {}
- ()

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to syntax

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```
–bus_delimiter {<> | [] | {} | ()}
```

The default is <>.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

```
Process > Properties > Synthesis Options > Bus Delimiter
```
Case (–case)

Case (–case) determines whether instance and net names are written in the final netlist using all lower or uppercase letters, or whether the case is maintained from the source. The case can be maintained for either Verilog or VHDL synthesis flow.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to syntax

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

- case {upper|lower|maintain}

The default is maintain.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Case
Case Implementation Style (–vlgcase)

The Case Implementation Style (–vlgcase) constraint:

- Supports Verilog designs only.
- Instructs XST how to interpret Verilog case statements.
- Has three possible values:
  - full
  - parallel
  - full-parallel

The following rules apply:

- Option not specified
  XST implements the exact behavior of the case statements.
- full
  XST assumes that the case statements are complete, and avoids latch creation.
- parallel
  XST assumes that the branches cannot occur in parallel, and does not use a priority encoder.
- full-parallel
  XST assumes that the case statements are complete, and that the branches cannot occur in parallel, therefore saving latches and priority encoders.

For more information, see:

- Full Case (FULL_CASE)
- Parallel Case (PARALLEL_CASE)
- Multiplexers

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

- vlgs_case {full|parallel|full-parallel}

By default, there is no value.
ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > HDL Options > Case Implementation Style

The values for this constraint are:

- full
- parallel
- full-parallel

By default, there is no value.
Verilog Macros (-define)

The Verilog Macros (-define) constraint:

- Is valid for Verilog designs only.
- Is used to define (or redefine) Verilog macros. This allows you to modify the design configuration without modifying the source code.
- Is useful for such processes as IP core generation and flow testing. If the defined macro is not used in the design, no message is given.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```bash
(define {name [=value] name [=value]} -)
```

where

- `name` is a macro name
- `value` is a macro text

The default is an empty definition.

```bash
(define {})
```

- Values for macros are not mandatory.
- Place the values inside curly braces ({}).
- Separate the values with spaces.
- You can specify macro text between quotation marks ("...") or without them. If the macro text contains spaces, you must use quotation marks ("...").

```bash
(define {macro1=Xilinx macro2="Xilinx Virtex6"})
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Verilog Macros

Do not use curly braces ({{}}) when specifying values in ISE® Design Suite.

```bash
acrol=Xilinx macro2="Xilinx Virtex6"
```
**Duplication Suffix (–duplication_suffix)**

The Duplication Suffix (–duplication_suffix) constraint:

- Controls how XST names replicated flip-flops.

  When XST replicates a flip-flop, it creates a name for the new flip-flop by adding \_n to the end of the original flip-flop name, where n is an index number. For example, if the original flip-flop name is my_ff, and the flip-flop is replicated three times, XST generates flip-flops with the following names:
  - my_ff_1
  - my_ff_2
  - my_ff_3

- Specifies a text string to append to the end of the default name.

  Use the \%d escape character to specify where the index number appears. For example, for the flip-flop named my_ff, if you specify \_dupreg\_\%d with the Duplication Suffix option, XST generates the following names:
  - my_ff_dupreg_1
  - my_ff_dupreg_2
  - my_ff_dupreg_3

  The \%d escape character can be placed anywhere in the suffix definition. For example, if the Duplication Suffix value is specified as \_dup\_\%d_reg, XST generates the following names:
  - my_ff_dup_1_reg
  - my_ff_dup_2_reg
  - my_ff_dup_3_reg

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices

**Applicable Elements**

Applies to files

**Propagation Rules**

Not applicable

**Syntax Examples**

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

- duplication_suffix string%dstring

The default is \%d.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Other
Full Case (FULL_CASE)

The Full Case (FULL_CASE) constraint:

- Applies to Verilog designs only.
- Indicates that all possible selector values have been expressed in a `case`, `casex`, or `casez` statement.
- Prevents XST from creating additional hardware for those conditions not expressed.

For more information, see: Multiplexers

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to `case` statements in Verilog meta comments.

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Verilog Syntax Example

The Verilog 2001 syntax is as follows:

```verilog
(* full_case *)
casex select
  4'b1xxx: res = data1;
  4'bx1xx: res = data2;
  4'bxx1x: res = data3;
  4'bxxx1: res = data4;
endcase
```

Since Full Case does not contain a target reference, the attribute immediately precedes the selector.

Full Case is also available as a meta comment in the Verilog code. The syntax differs from the standard meta comment syntax as shown in the following:

```verilog
// synthesis full_case
casex select // synthesis full_case
  4'b1xxx: res = data1;
  4'bx1xx: res = data2;
  4'bxx1x: res = data3;
  4'bxxx1: res = data4;
endcase
```
XST Command Line Syntax Example
Define globally with the run command.
-vlgcase {full|parallel|full-parallel}

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Synthesis Options > Full Case
For Case Implementation Style, select full as a Value.
Generate RTL Schematic (–rtlview)

The Generate RTL Schematic (–rtlview) constraint:

- Allows XST to generate a netlist file, representing a Register Transfer Level (RTL) structure of the design. This netlist can be viewed by the RTL Viewer and the Technology Viewer.

- Has three possible values:
  - yes
  - no
  - only

When only is specified, XST stops synthesis immediately after the RTL view is generated. The file containing the RTL view has an .NGR file extension.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

-rtlview {yes|no|only}

The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Generate RTL Schematic

The default is yes.
Generics (-generics)

Use Generics (-generics) to redefine generics (VHDL) or parameters (Verilog) values defined in the top-level design block. This feature:

- Allows you to modify the design configuration without modifying the source code.
- Is useful for such processes as IP core generation and flow testing.

If the defined value does not correspond to the data type defined in the Hardware Description Language (HDL) source code, then XST tries to detect the situation and issues a warning, ignoring the command line definition.

In some situations, XST may fail to detect a type mismatch. In that case, XST attempts to apply this value by adopting it to the type defined in the HDL file without any warning. Be sure that the value you specified corresponds to the type defined in the HDL source code. If a defined generic or parameter name does not exist in the design, no message is given, and the definition is ignored.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

```bash
run -generics {name=value name=value ...}
```

where

- `name` is the name of a generic or parameter of the design top level block
- `value` is the value of a generic or parameter of the design top level block

The default is an empty definition: `-generics {}`

Guidelines

- Place the `name/value` pairs inside curly braces (`{...}`).
- Separate the `name/value` pairs with spaces.
- XST can accept only constants of scalar types as values. Composite data types (arrays or records) are supported only for the following:
  - `string`
  - `std_logic_vector`
  - `std_ulogic_vector`
  - `signed, unsigned`
  - `bit_vector`
Formatting varies depending on the type of the generic value, as shown in the following table.

**XST Command Line Syntax Example**

<table>
<thead>
<tr>
<th>Type</th>
<th>Generic value syntax examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>“mystring”</td>
</tr>
<tr>
<td>Binary</td>
<td>b0011010</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>h3A</td>
</tr>
<tr>
<td>Decimal (integer)</td>
<td>d58 (or 58)</td>
</tr>
<tr>
<td>Boolean true</td>
<td>TRUE</td>
</tr>
<tr>
<td>Boolean false</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

For binary, hexadecimal, and decimal, there are no spaces between the prefix and the corresponding value.

```
-generics {company="mycompany" width=5 init_vector=b100101}
```

This command sets:

- **company** to mycompany
- **width** to 5
- **init_vector** to b100101

**Note** Values for generics of type std_logic_vector, std_ulogic_vector, or bit_vector, should be specified in binary or hexadecimal form. For example, specifying a binary value without the required base prefix (b), will lead XST to assume that the generic is of type integer, and to report a type mismatch as follows:

```
ERROR: HDLCompiler:839 - "example.vhd" Line 11: Type std_logic_vector does not match with the integer literal
```
Chapter 10: XST General Constraints

Hierarchy Separator (–hierarchy_separator)

The Hierarchy Separator (-hierarchy_separator) constraint:

- Defines the hierarchy separator character that is used in name generation when the design hierarchy is flattened.
- Supports the following characters:
  - _ (underscore)
  - / (forward slash) (default for newly created projects)

If a design contains a sub-block with instance INST1, and this sub-block contains a net called TMP_NET, then the hierarchy separator character is used when the design hierarchy is flattened. The name TMP_NET becomes INST1_TMP_NET. If the hierarchy separator character is / (forward slash), the net name is NST1/TMP_NET.

The / (forward slash) separator is useful in design debugging because it makes it much easier to identify a name if it is hierarchical.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

-hierarchy_separator {_[/]

The default is / (forward slash) for newly created projects.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Hierarchy Separator

The default is / (forward slash).
I/O Standard (IOSTANDARD)

The I/O Standard (IOSTANDARD) constraint:

- Is not applicable globally.
- Assigns an I/O standard to an I/O primitive.
- Can be applied on individual signals or instances, using:
  - a VHDL attribute, OR
  - a Verilog attribute, OR
  - an XST Constraint File (XCF) constraint.

For more information about this constraint, see the Constraints Guide.
Keep (KEEP)

Use the Keep (KEEP) constraint to preserve signals in the netlist.

Keep (KEEP) is an advanced mapping constraint. When a design is mapped, some nets may be absorbed into logic blocks. When a net is absorbed into a block, it can no longer be seen in the physical design database. This may happen, for example, if the components connected to each side of a net are mapped into the same logic block. The net may then be absorbed into the block containing the components. Keep prevents this from happening.

Observe the following limitations of Keep:

- Keep preserves the existence of the designated signal in the final netlist, but not the logic that surrounds it, which may be transformed by one of the optimizations performed by XST. Consider, for example, the 2-bit selector of a 4-to-1 multiplexer. A Keep attached to that signal will ensure its existence in the final netlist. But the multiplexer could be automatically re-encoded by XST using one-hot encoding. As a consequence, the signal preserved in the final netlist will be 4-bits wide instead of the original 2 bits. To preserve the structure of the signal in this case, you must use Enumerated Encoding (ENUM_ENCODING) in addition to Keep.

- In general, in order to preserve both a signal and the elements that directly surround it, use Save (S or SAVE) instead.

- Do not use Keep to control register replication. Use Register Duplication (REGISTER_DUPLICATION) instead.

- Do not use Keep to control removal of equivalent registers. Use Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) instead.

The values for this constraint are:

- **true**
  Preserves the designated signal in the NGC netlist. In addition, the Keep constraint is propagated into the netlist, so that implementation steps will also preserve the signal.

- **soft**
  Preserves the designated signal in the NGC netlist, but Keep is not propagated to implementation where the signal may be optimized away.

- **false**
  XST does not specifically attempt to preserve the designated signal. A signal may still exist in the NGC netlist as a result of internal signal preservation rules, but XST does not put any extra effort beyond those rules.

Keep can be applied to a signal, using a VHDL attribute, a Verilog attribute, or an XCF constraint.

**Note** In an XST Constraint File (XCF) file, the value of Keep (KEEP) may optionally be enclosed in double quotes. Double quotes are mandatory for SOFT.

### XCF Syntax Example One

```
BEGIN MODEL testkeep
    NET aux1 KEEP=true;
END;
```
XCF Syntax Example Two

BEGIN MODEL testkeep
NET aux1 KEEP="soft";
END;

For more information about this constraint, see the Constraints Guide.
Keep Hierarchy (KEEP_HIERARCHY)

Keep Hierarchy (KEEP_HIERARCHY) is a synthesis and implementation constraint. If hierarchy is maintained during synthesis, the implementation tools use Keep Hierarchy to preserve the hierarchy throughout implementation, and allow a simulation netlist to be created with the desired hierarchy.

XST can flatten the design to obtain better results by optimizing entity or module boundaries. If Keep Hierarchy is set to yes, the generated netlist is hierarchical, and respects the hierarchy and interface of any entity or module in the design.

Keep Hierarchy is related to the hierarchical blocks (VHDL entities, Verilog modules) specified in the Hardware Description Language (HDL) design, and does not concern the macros inferred by the HDL synthesizer.

Keep Hierarchy Values

The values for this constraint are:

- **yes**
  Allows the preservation of the design hierarchy, as described in the HDL project. If this value is applied to synthesis, it is also propagated to implementation.

- **no** (default)
  Hierarchical blocks are merged in the top level module.

- **soft**
  Allows the preservation of the design hierarchy in synthesis, but Keep Hierarchy is not propagated to implementation.

Preserving the Hierarchy

In general, a Hardware Description Language (HDL) design is a collection of hierarchical blocks. Preserving the hierarchy enables fast processing since the optimization is done on separate pieces of reduced complexity. Nevertheless, very often, merging the hierarchy blocks improves the fitting results (fewer PTerms and device macrocells, better frequency) because the optimization processes (collapsing, factorization) are applied globally on the entire logic.

In the following figure, if Keep Hierarchy is set to the entity or module I2, the hierarchy of I2 is in the final netlist, but its contents I4, I5 are flattened inside I2. I1, I3, I6, and I7 are also flattened.
Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to logical blocks, including blocks of hierarchy or symbols

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute keep_hierarchy : string;
```

Specify as follows:

```vhdl
attribute keep_hierarchy of architecture_name: architecture is "[yes|no|soft]";
```

The default is **no**.

**Verilog Syntax Example**

```verilog
(* keep_hierarchy = "[yes|no|soft]" *)
```

**XCF Syntax Example**

```xcf
MODEL "entity_name" keep_hierarchy=[yes|no|soft];
```
XST Command Line Syntax Example
Define globally with the run command.
-keep_hierarchy {yes|no|soft}
The default is no.
For more information, see:
Running XST in Command Line Mode

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Synthesis Options > Keep Hierarchy
Library Search Order (–Iso)

Use Library Search Order (–Iso) to specify the order in which library files are used. To invoke Library Search Order:

- Specify the search order file in ISE® Design Suite in:
  Process > Properties > Synthesis Options > Library Search Order, or
- Use the –Iso command line option.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```bash
-lso file_name.lso
```

There is no default file name. If not specified, XST uses the default search order.

For more information, see:
Library Search Order (LSO) Files

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Library Search Order

For more information, see:
Library Search Order (LSO) Files
LOC

LOC defines where a design element can be placed within a device.
For more information about this constraint, see the *Constraints Guide*. 
Netlist Hierarchy (-netlist_hierarchy)

The Netlist Hierarchy (-netlist_hierarchy) constraint:
- Controls the form in which the final NGC netlist is generated.
- Allows you to write the hierarchical netlist even if the optimization was done on a partially or fully flattened design.
- Has the following values:
  - as_optimized
    XST takes Keep Hierarchy (KEEP_HIERARCHY) into account, and generates the NGC netlist in the form in which it was optimized. In this mode, some hierarchical blocks can be flattened, and some can maintain hierarchy boundaries.
  - rebuilt
    XST writes a hierarchical NGC netlist, regardless of Keep Hierarchy (KEEP_HIERARCHY).

The hierarchy is not always fully rebuilt. The main factors to a partial reconstruction are:
- Macro grouping across hierarchy
  During Advanced HDL Synthesis, XST attempts to group basic inferred macros together into higher complexity macros. These composite macros are usually candidates for implementation with DSP or block RAM resources. When grouped macros were inferred in distinct hierarchical blocks, local hierarchical boundaries may be removed and are not rebuilt in the final netlist.
- Optimization of small hierarchical blocks during LUT optimizations
  During Low-Level Synthesis, logic optimizations, such as LUT packing, may send all logic contained in very low complexity hierarchical blocks (typically only a few LUTs) to the surrounding context. Such small hierarchical blocks are eliminated in the process and are not rebuilt in the final netlist.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

```
- netlist_hierarchy {as_optimized|rebuilt}
```

The default is as_optimized.
Optimization Effort (OPT_LEVEL)

Optimization Effort (OPT_LEVEL) defines the synthesis optimization effort level.
The values for this constraint are:

- 1 (Normal)
  Normal optimization effort is the default and recommended effort level for the majority of designs. It provides both a high level of optimizations and fast processing times, especially for hierarchical designs.

- 2 (High)
  In High optimization effort, XST explores additional optimization techniques in order to deliver the best possible solution. This can result in significantly increased synthesis runtimes. A better outcome is not guaranteed. While these optimizations may benefit a particular design, in other cases there may be no improvement at all, or the results may even be degraded. Xilinx® therefore recommends Optimization Effort 1 for most designs.

- 0 (Fast)
  By turning off some of the optimization algorithms used in Normal effort level, Fast optimization effort delivers a synthesized result in minimal runtime. This may result in an optimization trade-off for your particular design. This effort level can be particularly useful at early stages of the design process, when seeking to obtain rapid results from the tools.

Architecture Support
Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements
Applies globally, or to an entity or module

Propagation Rules
Applies to the entity or module to which it is attached

Syntax Examples
The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example
Declare as follows:
attribute opt_level: string;
Specify as follows:
attribute opt_level of entity_name: entity is "(0|1|2)";

Verilog Syntax Example
(* opt_level = "(0|1|2)" *)

XCF Syntax Example
MODEL "entity_name" opt_level={0|1|2};
XST Command Line Syntax Example
Define globally with the run command.
-\texttt{opt\_level \{0|1|2\}}
The default is 1.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
\texttt{Process > Properties > Synthesis Options > Optimization Effort}
Optimization Goal (OPT_MODE)

The Optimization Goal (OPT_MODE) constraint:

- Defines the synthesis optimization strategy.
- Has values of:
  - `speed` (default)
    Reduces the number of logic levels and therefore increases frequency.
  - `area`
    Reduces the total amount of logic used for design implementation and therefore improves design fitting.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity or module

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute opt_mode: string;
```

Specify as follows:

```vhdl
attribute opt_mode of entity_name: entity is "(speed|area)";
```

**Verilog Syntax Example**

(* `opt_mode = "(speed|area)" *)

**XCF Syntax Example**

```xilinx
MODEL "entity_name" opt_mode=(speed|area);
```

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-opt_mode (area|speed)
```

The default is `speed`.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

`Process > Properties > Synthesis Options > Optimization Goal`

The default is `speed`.
Parallel Case (PARALLEL_CASE)

The Parallel Case (PARALLEL_CASE) constraint:

• Is valid for Verilog designs only.
• Forces a case statement to be synthesized as a parallel multiplexer.
• Prevents the case statement from being transformed into a prioritized if-elsif cascade.

For more information, see:

Multiplexers

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to case statements in Verilog meta comments only

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Verilog Syntax Example

The syntax is:

(* parallel_case *)

case select
    4'b1xxx: res = data1;
    4'bx1xx: res = data2;
    4'bxx1x: res = data3;
    4'bxxx1: res = data4;
endcase

Parallel Case is also available as a meta comment in the Verilog code. The syntax differs from the standard meta comment syntax as shown in the following:

// synthesis parallel_case

case select // synthesis parallel_case
    4'b1xxx: res = data1;
    4'bx1xx: res = data2;
    4'bxx1x: res = data3;
    4'bxxx1: res = data4;
endcase
XST Command Line Syntax Example
Define globally with the run command.

-`-vlgcase {full|parallel|full-parallel}`
RLOC

The RLOC constraint:

- Is a basic mapping and placement constraint.
- Groups logic elements into discrete sets.
- Defines the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design.

For more information about this constraint, see the Constraints Guide.

Save (S or SAVE)

Save (S or SAVE) is an advanced mapping constraint. Typically, when the design is mapped, some nets may be absorbed into logic blocks, and some elements such as LUTs may be optimized away. If you need to preserve access to some specific nets and blocks in the post-synthesis netlist, Save prevents such optimizations from happening. Disabled optimization techniques include nets or blocks replication and register balancing.

If Save is applied to a net, XST preserves the net with all elements directly connected to it in the final netlist. This includes nets connected to these elements.

If Save is applied to a block such as a LUT, XST preserves the LUT with all signals connected to it.

Applicable elements are:

- Nets
  - XST preserves the designated net with all elements directly connected to it in the final netlist. As a consequence nets connected to these elements are also preserved.
- Instantiated device primitives
  - If Save is applied to an instantiated primitive, such as a LUT, XST preserves the LUT with all signals connected to it.

For more information about this constraint, see the Constraints Guide.
Synthesis Constraint File (–uc)

Synthesis Constraint File (–uc) specifies a synthesis constraint file for XST to use. The XST Constraint File (XCF) has an extension of .xcf. If the extension is not .xcf, XST errors out and stops processing.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

–uc filename

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Synthesis Constraints File
Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON)

The Translate Off (TRANSLATE_OFF) and Translate On (TRANSLATE_ON) constraints:

- Instruct XST to ignore portions of the Hardware Description Language (HDL) source code that are not relevant for synthesis, such as simulation code.
  - TRANSLATE_OFF marks the beginning of the section to be ignored.
  - TRANSLATE_ON instructs XST to resume synthesis from that point.
- Are Synopsys directives that XST supports in Verilog. Automatic conversion is also available in VHDL and Verilog.
- Can be used with the following words:
  - synthesis
  - synopsys
  - pragma

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies locally

Propagation Rules

Instructs the synthesis tool to enable or disable portions of code

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
-- synthesis translate_off
...code not synthesized...
-- synthesis translate_on
```

Verilog Syntax Example

Translate Off and Translate On are available as HDL meta comments. The Verilog syntax differs from the standard meta comment syntax as shown in the following coding example.

```verilog
// synthesis translate_off
...code not synthesized...
// synthesis translate_on
```
Ignore Synthesis Constraints File (–iuc)

Use Ignore Synthesis Constraints File (–iuc) to ignore the constraint file specified with Synthesis Constraints File (-uc) during synthesis.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

-iuc {yes|no}

The default is no.

ISE Design Suite Syntax Example

Caution! Ignore Synthesis Constraints File is shown as Synthesis Constraints File in ISE® Design Suite. The constraint file is ignored if you uncheck this option. It is checked by default (therefore resulting in a –iuc no command line switch), meaning that any synthesis constraints file you specify is taken into account.

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Synthesis Constraints File
Verilog Include Directories (–vlgincdir)

The Verilog Include Directories (–vlgincdir) constraint:

• Is used in conjunction with ‘include.
• Helps the parser find files referenced by ‘include statements.

When an ‘include statement references a file, XST searches in the following order relative to the:
• current directory
• inc directories
• current file

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to directories

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define Verilog Include Directories globally with the –vlgincdir option of the run command. Allowed values are names of directories.

For more information, see:
Names With Spaces in Command Line Mode
Define globally with the run command.

–vlgincdir {directory_path [directory_path]}

There is no default.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Verilog Include Directories

Allowed values are names of directories.

There is no default.

To view this constraint, select Edit > Preferences > Processes > Property Display Level > Advanced.
HDL Library Mapping File (–xsthdpini)

Use HDL Library Mapping File (–xsthdpini) to define the library mapping.

The library mapping file has two associated parameters:

- `XSTHDPINI`
- `XSTHDPDIR`

The library mapping file contains:

- The library name
- The directory in which the library is compiled

XST maintains two library mapping files:

- The pre-installed file, which is installed during the Xilinx® software installation.
- The user file, which you can define for your own projects.

The pre-installed (default) INI file:

- Is named xhdp.ini.
- Is located in %XILINX%\vhdl\xst.
- Contains information about the locations of the standard VHDL and UNISIM libraries.
- Should not be modified.

**Note** The syntax can be used for user library mapping.

- Appears as follows:

  ```
  -- Default lib mapping for XST
  std=$XILINX/vhdl/xst/std
  ieee=$XILINX/vhdl/xst/unisim
  unisim=$XILINX/vhdl/xst/unisim
  aim=$XILINX/vhdl/xst/aim
  pls=$XILINX/vhdl/xst/pls
  ```

Use the INI file format to define where each of your own libraries will be placed. By default, all compiled VHDL files are stored in the `xst` subdirectory of the project directory.

To place a custom INI file anywhere on a disk:

- Select the VHDL INI file in ISE® Design Suite in:
  
  **Process > Properties > Synthesis Options**, or

- Set the `-xsthdpini` parameter, using the following command in standalone mode:

  ```
  set -xsthdpini file_name
  ```

Although you can give this library mapping file any name you wish, Xilinx recommends keeping the `.ini` classification. The format is:

```ini
library_name=path_to_compiled_directory
```

Use a double dash (`--`) for comments.

**MY.INI Example Text**

```ini
work1=H:\Users\conf\my_lib\work1
work2=C:\mylib\work2
```
Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to files

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```
set -xsthdpini file_name
```

The command can accept a single file only.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > VHDL INI File

To view this constraint, select Edit > Preferences > Processes > Property Display Level > Advanced.
Work Directory (–xsthdmpdir)

Work Directory (–xsthdmpdir) defines the location in which VHDL-compiled files must be placed if the location is not defined by library mapping files. To access Work Directory:

- In ISE® Design Suite select:
  Process > Properties > Synthesis Options > VHDL Work Directory, or
- In standalone mode run:
  \texttt{set –xsthdmpdir directory}

Work Directory Example

Assume the following:

- Three different users are working on the same project.
- They share one standard, pre-compiled library, \texttt{shlib}.
- This library contains specific macro blocks for their project.
- Each user also maintains a local work library.
- User Three places her local work library outside the project directory (for example, in \texttt{c:\temp}).
- User One and User Two share another library (\texttt{lib12}) between them, but not with User Three.

The settings required for the three users are as follows:

Work Directory Example User One

Mapping file:
\texttt{schlib=z:\sharedlibs\shlib}
\texttt{lib12=z:\userlibs\lib12}

Work Directory Example User Two

Mapping file:
\texttt{schlib=z:\sharedlibs\shlib}
\texttt{lib12=z:\userlibs\lib12}

Work Directory Example User Three

Mapping file:
\texttt{schlib=z:\sharedlibs\shlib}

User Three also sets:
\texttt{XSTHDPDIR = c:\temp}

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to directories

Propagation Rules

Not applicable
Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the `run` command.

```
set -xsthdpdir directory
```

Work Directory can accept a single path only. You must specify the directory. There is no default.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

```
Process > Properties > Synthesis Options > VHDL Work Directory
```

To view this constraint, select `Edit > Preferences > Processes > Property Display Level > Advanced`. 
Chapter 11

XST HDL Constraints

Automatic FSM Extraction (FSM_EXTRACT)

The Automatic FSM Extraction (FSM_EXTRACT) constraint:
• Enables or disables Finite State Machine (FSM) extraction and specific synthesis optimizations.
• Must be enabled in order to set values for the FSM Encoding Algorithm and FSM Flip-Flop Type.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

\texttt{attribute fsm\_extract: string;}

Specify as follows:

\texttt{attribute fsm\_extract of \{entity\_name|signal\_name\}:}
\texttt{\{entity|signal is \"(yes|no)\";} 

Verilog Syntax Example

Place immediately before the module or signal declaration:

\texttt{(* fsm\_extract = \"(yes|no)\") */}

XCF Syntax Example One

\texttt{MODEL \"entity\_name\" fsm\_extract=(yes|no|true|false);}
**XCF Syntax Example Two**

```
BEGIN MODEL "entity_name"

NET "signal_name" fsm_extract={yes|no|true|false};
END;
```

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-fsm_extract {yes|no}*
```

The default is `yes`.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

**Process > Properties > HDL Options > FSM Encoding Algorithm**

- If `FSM Encoding Algorithm (FSM_ENCODING)` is set to `none`, and `-fsm_extract` is set to `no`, `-fsm_encoding` does not influence synthesis.
- In all other cases, `-fsm_extract` is set to `yes`, and `-fsm_encoding` is set to the selected value.

For more information, see:

`FSM Encoding Algorithm (FSM_ENCODING)`
Enumerated Encoding (ENUM_ENCODING)

Enumerated Encoding (ENUM_ENCODING) applies a specific encoding to a VHDL enumerated type. The value is a string containing space-separated binary codes. You can specify enumerated Encoding only as a VHDL constraint on the considered enumerated type.

When describing a Finite State Machine (FSM) using an enumerated type for the state register, you can specify a particular encoding scheme with Enumerated Encoding. In order for XST to use this encoding, set FSM Encoding Algorithm (FSM_ENCODING) to user for the considered state register.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to a type or signal. Because Enumerated Encoding must preserve the external design interface, XST ignores Enumerated Encoding when it is used on a port.

Propagation Rules

Applies to the type or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Specify as a VHDL constraint on the considered enumerated type.

...  
architecture behavior of example is  
type statetype is (ST0, ST1, ST2, ST3);  
attribute enum_encoding : string;  
attribute enum_encoding of statetype : type is "001 010 100 111";  
signal statel : statetype;  
signal state2 : statetype;  
begin  
...  

XCF Syntax Example

BEGIN MODEL "entity_name"  
NET "signal_name" enum_encoding="string";  
END;
Equivalent Register Removal
(EQUIVALENT_REGISTER_REMOVAL)

Equivalent Register Removal (EQUIVALENT_REGISTER_REMOVAL) enables or disables removal of equivalent registers described at the Register Transfer Level (RTL) Level. By default, XST does not remove equivalent flip-flops if they are instantiated from a Xilinx® primitive library.

Removal of equivalent flip-flops increases the probability that the design will fit on the targeted device.

The values for this constraint are:
- **yes** (default)
  - Flip-flop optimization is allowed.
- **no**
  - Flip-flop optimization is inhibited. The flip-flop optimization algorithm is time consuming. For fast processing, use **no**.
- **true** (XCF only)
- **false** (XCF only)

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices

**Applicable Elements**

Applies globally, or to an entity, module, or signal

**Propagation Rules**

Removes equivalent flip-flops and flip-flops with constant inputs

**Syntax Examples**

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute equivalent_register_removal: string;
```

Specify as follows:

```vhdl
attribute equivalent_register_removal of
{entity_name|signal_name}: {signal|entity} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* equivalent_register_removal = "{yes|no}" *)
```

**XCF Syntax Example One**

```xilinx
MODEL "entity_name"
equivalent_register_removal={yes|no|true|false};
```
XCF Syntax Example Two

BEGIN MODEL "entity_name"
NET "signal_name"
equivalent_register_removal={yes|no|true|false};
END;

XST Command Line Syntax Example

Define globally with the run command.

-equivalent_register_removal {yes|no}

The default is yes.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Equivalent Register Removal
FSM Encoding Algorithm (FSM_ENCODING)

The FSM Encoding Algorithm (FSM_ENCODING) constraint:

- Selects the Finite State Machine (FSM) coding technique.
- Defaults to auto. The best coding technique is automatically selected for each individual state machine.

In order to select a value for the FSM Encoding Algorithm, Automatic FSM Extraction (FSM_EXTRACT) must be enabled.

The values for this constraint are:

- auto
- one-hot
- compact
- sequential
- gray
- johnson
- speed1
- user

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute fsm_encoding: string;
```

Specify as follows:

```vhdl
attribute fsm_encoding of
{entity_name|signal_name}: {entity|signal} is
"{auto|one-hot|compact|sequential|gray|johnson|speed1|user}";
```

The default is auto.
Verilog Syntax Example
Place immediately before the module or signal declaration:

(*)

\texttt{fsm\_encoding = "\{auto\|one-hot\|compact\|sequential\|gray\|johnson\|speed1\|user\}" *}

The default is auto.

XCF Syntax Example One

\texttt{MODEL "entity\_name" fsm\_encoding\{auto\|one-hot\|compact\|sequential\|gray\|johnson\|speed1\|user\};}

XCF Syntax Example Two

\texttt{BEGIN MODEL "entity\_name"}

\texttt{NET "signal\_name" fsm\_encoding\{auto\|one-hot\|compact\|sequential\|gray\|johnson\|speed1\|user\};}

\texttt{END;}

XST Command Line Syntax Example

Define globally with the \texttt{run} command.

\texttt{-fsm\_encoding }

\texttt{\{auto\|one-hot\|compact\|sequential\|gray\|johnson\|speed1\|user\}}

The default is auto.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

\texttt{Process > Properties > HDL Options > FSM Encoding Algorithm}

- If FSM Encoding Algorithm is set to \texttt{none}, and \texttt{-fsm\_extract} is set to \texttt{no}, 
  \texttt{-fsm\_encoding} has no influence on the synthesis.

- In all other cases, \texttt{-fsm\_extract} is set to \texttt{yes} and \texttt{-fsm\_encoding} is set to the value selected in the menu. For more information, see Automatic FSM Extraction (FSM\_EXTRACT).
Mux Minimal Size (MUX_MIN_SIZE)

Caution! Review this constraint carefully before use.

Mux Minimal Size (MUX_MIN_SIZE) allows you to control the minimal size of multiplexer macros inferred by XST.

Size is the number of multiplexed data inputs. For example, for a 2-to-1 multiplexer, the size, or number of multiplexed inputs, is 2. For a 16-to-1 multiplexer, the size is 16. Selector inputs do not count.

This number is independent of the width of the selected data. Both a 1-bit wide 8-to-1 multiplexer, and a 16-bit wide 8-to-1 multiplexer, have a size of 8.

Mux Minimal Size takes an integer value greater than 1. The default value is 2.

By default, XST infers 2-to-1 multiplexer macros. Explicit inference of 2-to-1 multiplexers can have either a positive or negative impact on final device utilization, depending on the design. Xilinx® does not recommend using Mux Minimal Size if device utilization is satisfactory

If device utilization is not satisfactory, Mux Minimal Size may benefit your design if there are a large number of 2-to-1 multiplexers inferred in the parts of the design that are significantly contributing to the unsatisfactory device utilization. In this case, Xilinx recommends that you try to disable 2-to-1 multiplexer inference, either globally, or for the blocks that are specifically affecting your results. To disable inference of 2-to-1 multiplexers, apply a value of 3.

Mux Minimal Size may prevent inference of multiplexers for sizes above 2, but the benefits are speculative. Xilinx recommends extra caution before using Mux Minimal Size in this situation.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to a designated VHDL entity or Verilog module.

Propagation Rules

Applies to the designated entity or module.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

attribute mux_min_size: string;

Specify as follows:

attribute mux_min_size of entity_name : entity is "integer";

The default is 2.
Verilog Syntax Example

Place immediately before the module declaration

(* mux_min_size= "integer" *)

The default is 2.

XST Command Line Syntax Example

Define globally with the run command.

-mux_min_size integer

Note  Mux Minimal Size is not available in the default XST options set in ISE® Design Suite.
Resource Sharing (RESOURCE_SHARING)

Resource Sharing (RESOURCE_SHARING) enables or disables resource sharing of arithmetic operators.

The values for this constraint are:
- yes (default)
- no
- true (XCF only)
- false (XCF only)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to design elements

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute resource_sharing: string;
```

Specify as follows:

```vhdl
attribute resource_sharing of entity_name: entity is "{yes|no}";
```

Verilog Syntax Example

Place immediately before the module declaration or instantiation:

```verilog
attribute resource_sharing of entity_name: entity is "{yes|no}";
```

XCF Syntax Example One

```xilinx
MODEL "entity_name" resource_sharing={yes|no|true|false};
```

XCF Syntax Example Two

```xilinx
BEGIN MODEL "entity_name"
NET "signal_name" resource_sharing={yes|no|true|false};
END;
```
XST Command Line Syntax Example
Define globally with the run command.

`-resource_sharing {yes|no}`
The default is yes.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.

HDL Options > Resource Sharing
Safe Recovery State (SAFE_RECOVERY_STATE)

Safe Recovery State (SAFE_RECOVERY_STATE) defines a recovery state for use when a Finite State Machine (FSM) is implemented in Safe Implementation mode. If the FSM enters an invalid state, XST uses additional logic to force the FSM to a valid recovery state. By implementing FSM in safe mode, XST collects all code not participating in the normal FSM behavior and treats it as illegal.

XST uses logic that returns the FSM synchronously to the:
- Known state
- Reset state
- Power up state
- State specified using Safe Recovery State

For more information, see:
Safe Implementation (SAFE_IMPLEMENTATION)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to a signal representing a state register

Propagation Rules

Applies to the signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute safe_recovery_state: string;
```

Specify as follows:

```vhdl
attribute safe_recovery_state of {signal_name}:{signal} is "<value>";
```

Verilog Syntax Example

Place immediately before the signal declaration.

```verilog
(* safe_recovery_state = "<value>" *)
```

XCF Syntax Example

```xchp
BEGIN MODEL "entity_name"
NET "signal_name" safe_recovery_state="<value>";
END;
```
Safe Implementation (SAFE_IMPLEMENTATION)

Safe Implementation (SAFE_IMPLEMENTATION) implements Finite State Machine (FSM) components in Safe Implementation mode.

In Safe Implementation mode, XST generates additional logic that forces an FSM to a valid state (recovery state) if the FSM enters an invalid state.

By default, XST automatically selects reset as the recovery state. If the FSM does not have an initialization signal, XST selects power-up as the recovery state.

Define the recovery state manually with Safe Recovery State (SAFE_RECOVERY_STATE).

To activate Safe Implementation in:

• ISE® Design Suite
  Select Process > Properties > HDL Options > Safe Implementation.

• Hardware Description Language (HDL)
  Apply Safe Implementation to the hierarchical block or signal that represents the state register in the FSM.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to an entire design through the XST command line, to a particular block (entity, architecture, component), or to a signal.

Propagation Rules

Applies to an entity, component, module, signal, or instance to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute safe_implementation: string;
```

Specify as follows:

```vhdl
attribute safe_implementation of
{entity_name|component_name|signal_name}:
{entity|component|signal is "{yes|no}"};
```

Verilog Syntax Example

Place immediately before the module or signal declaration:

```verilog
(* safe_implementation = "{yes|no}" *)
```

XCF Syntax Example One

```verilog
MODEL "entity_name" safe_implementation={yes|no|true|false};
```
XCF Syntax Example Two

```
BEGIN MODEL "entity_name"
NET "signal_name" safe_implementation="{yes|no|true|false};
END;
```

XST Command Line Syntax Example

Define globally with the run command.

```
-safe_implementation {yes|no}
```

The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

HDL Options > Safe Implementation
Chapter 12

XST FPGA Constraints (Non-Timing)

Many constraints can be applied:

- **Globally** to an entire entity or model, or
- **Locally** to individual signals, nets or instances

**Asynchronous to Synchronous (ASYNC_TO_SYNC)**

Use Asynchronous to Synchronous (ASYNC_TO_SYNC) to treat asynchronous set and reset signals as synchronous.

The Asynchronous to Synchronous transformation:

- Applies to inferred sequential elements only
- Does not apply to instantiated flip-flops and latches
- Is performed on-the-fly
- Is reflected in the post-synthesis netlist
- Does not change the HDL source code

These set and reset functionality of Xilinx® device resources such as DSP blocks and block RAMs is synchronous by nature. If strict coding practices require you to describe set and reset signals asynchronously, you may not be using those resources to their full potential. Automatic Asynchronous to Synchronous transformation allows you to assess their potential without changing the description of the sequential elements in your HDL source code. By better leveraging registers in your design, you may be able to:

- Improve device utilization
- Increase circuit performance
- Achieve better power reduction

**Caution!** Carefully review the following to assess the potential impact of Asynchronous to Synchronous transformation on your design.

- As a result of Asynchronous to Synchronous transformation, the post-synthesis netlist is **theoretically not functionally equivalent** to your pre-synthesis HDL description. However, if not actually using the asynchronous sets and resets that you have described, or if they are derived from synchronous sources, the post-synthesis solution is functionally equivalent in those cases.
- If you achieve your design goals by using Asynchronous to Synchronous transformation, determine whether you should change the HDL description to enforce synchronous set and reset signals in order to ensure the expected circuit behavior. Changing the HDL description may also ease design validation.
- Xilinx highly recommends that you perform a timing simulation in order to assess the impact of the Asynchronous to Synchronous transformation on your design.
- If you are allowed to change your coding practices, Xilinx recommends that you describe synchronous set and reset signals in your HDL source code.
Architecture Support
Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements
Applies globally

Propagation Rules
Not applicable

Syntax Examples
The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**
Define globally with the `run` command.

```
-async_to_sync {yes|no}
```

The default is `no`.

**ISE Design Suite Syntax Example**
Define globally in ISE Design Suite.

Process > Properties > HDL Options > Asynchronous to Synchronous
Automatic BRAM Packing (AUTO_BRAM_PACKING)

Use Automatic BRAM Packing (AUTO_BRAM_PACKING) to pack two small block RAMs in a single block RAM primitive as dual-port block RAM. XST packs block RAMs together only if they are situated in the same hierarchical level. Automatic BRAM Packing is disabled by default.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

```
-auto_bram_packing {yes|no}
```

The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Automatic BRAM Packing
BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO)

BRAM Utilization Ratio (BRAM_UTILIZATION_RATIO) defines the number of block RAMs that XST must not exceed during synthesis. Block RAMs may come not only from block RAM inference processes, but from instantiation and block RAM mapping optimizations. You can isolate a Register Transfer Level (RTL) description of logic in a separate block, and then direct XST to map this logic to block RAM.

For more information, see Mapping Logic to Block RAM.

Instantiated block RAMs are the primary candidates for available block RAM resources. The inferred RAMs are placed on the remaining block RAM resources. However, if the number of instantiated block RAMs exceeds the number of available resources, XST does not modify the instantiations and implement them as block RAMs. The same behavior occurs if you force specific RAMs to be implemented as block RAMs. If there are no resources, XST respects user constraints, even if the number of block RAM resources is exceeded.

If the number of user-specified block RAMs exceeds the number of available block RAM resources on the target device, XST issues a warning, and uses only available block RAM resources on the chip for synthesis. Use value -1 to disable automatic block RAM resource management. This can be used to see the number of block RAMs XST can potentially infer for a specific design.

Synthesis time may increase if the number of block RAMs in the design significantly exceeds the number of available block RAMs on the target device (hundreds of block RAMs). This may happen due to a significant increase in design complexity when all non-fittable block RAMs are converted to distributed RAMs.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Examples

Define globally with the run command.

-`--bram_utilization_ratio <integer>[%|#]`
where

- The integer value range is -1 to 100.
- % denotes a percentage value, whereas # means an absolute number of block RAMs.
- There must be no space between the integer value and the % or # character.
- If both % and # are omitted, a percentage value is assumed.
- The default value is 100 (XST uses up to 100% of available block RAM resources).
- A value of -1 disables automatic block RAM resource management, and may be useful in assessing the amount of block RAM resources that XST can potentially infer.

**XST Command Line Syntax Example One**

```
-bram_utilization_ratio 50
```

*means*

50% of block RAMs in the target device

**XST Command Line Syntax Example Two**

```
-bram_utilization_ratio 50%
```

*means*

50% of block RAMs in the target device

**XST Command Line Syntax Example Three**

```
-bram_utilization_ratio 50#
```

*means*

50 block RAMs

There must be no space between the integer value and the percent (%) or pound (#) characters.

In some situations, you can disable automatic block RAM resource management (for example, to see how many block RAMs XST can potentially infer for a specific design). To disable automatic resource management, specify -1 (or any negative value) as a constraint value.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

**Process > Properties > Synthesis Options > BRAM Utilization Ratio**

In ISE® Design Suite, you can define the value of BRAM Utilization Ratio only as a percentage. You cannot define the value as an absolute number of Block RAMs.
Buffer Type (BUFFER_TYPE)

The Buffer Type (BUFFER_TYPE) constraint:
- Selects the type of buffer to be inserted on the designated I/O port or internal net.
- Is a new name for CLOCK_BUFFER. Since CLOCK_BUFFER will become obsolete in future releases, Xilinx® recommends that you use this new name.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to signals

Propagation Rules

Applies to the signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

 Declare as follows:

```vhdl
attribute buffer_type: string;
```

Specify as follows:

```vhdl
attribute buffer_type of signal_name: signal is
"(bufp1l|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|ibuf|obuf|buf|none)";
```

**Verilog Syntax Example**

 Place immediately before the signal declaration:

```verilog
(* buffer_type = "(bufp1l|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|ibuf|obuf|buf|none)" *)
```

**XCF Syntax Example**

```c
BEGIN MODEL "entity_name"

NET "signal_name" buffer_type=(bufp1l|ibufg|bufg|bufgp|bufh|bufr|bufio|bufio2fb|bufio2|ibuf|obuf|buf|none);

END;
```
Extract BUFGCE (BUFGCE)

The Extract BUFGCE (BUFGCE) constraint:

- Implements BUFGMUX functionality by inferring a BUFGMUX primitive. This operation reduces the wiring. Clock and clock enable signals are driven to n sequential components by a single wire.
- Must be attached to the primary clock signal.
- Has values of:
  - yes
  - no
- Is accessible through Hardware Description Language (HDL) code.

If \texttt{bufgce=\texttt{yes}}, XST implements \texttt{BUFGMUX} functionality if possible. All flip-flops must have the same clock enable signal.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to clock signals

Propagation Rules

Applies to the signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

\[
\text{attribute bufgce : string;}\\
\]

Specify as follows:

\[
\text{attribute bufgce of signal_name : signal is } \{ \text{yes|no} \};
\]

**Verilog Syntax Example**

Place immediately before the signal declaration:

\[
(* \text{bufgce }= \{ \text{yes|no} \} *)
\]

**XCF Syntax Example One**

\[
\text{BEGIN MODEL } "\text{entity_name}"
\]

\[
\text{NET } "\text{primary_clock_signal} \text{bufgce=\{yes|no|true|false\}}"
\]

\[
\text{END;}
\]
Cores Search Directories (–sd)

Cores Search Directories (–sd) allows you to specify one or several directories, in addition to the default directory, where XST looks for cores. By default, XST searches for cores in the directory designated by the -ifn option. You need to designate only the directories containing the cores to be considered. Do not list individual core files. Core directories may be specified as absolute or relative paths.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```bash
-sd {directory_path [directory_path]}
```

The value may be a single directory name, or a list of several directory names.

- Enclose the list of directories between curly braces. You may omit them if specifying only one directory.
- Multiple directories must be space-separated

Xilinx® does not recommend using directory names containing spaces. However, you may include such directories in your search list, provided that they are enclosed in double quotes as shown in the following example.

```bash
-sd {"./mydir1/mysubdir1" ".mydir2" ".mydir3/mysubdir with space"}
```

For more information, see:

[Names With Spaces in Command Line Mode](https://www.xilinx.com)

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Cores Search Directory
DSP Utilization Ratio (DSP_UTILIZATION_RATIO)

DSP Utilization Ratio (DSP_UTILIZATION_RATIO) allows you to restrict the number of DSP blocks used by XST to implement inferred functions. By default, XST infers DSP blocks within the limit of available resources on the selected device. DSP Utilization Ratio prevents XST from potentially using all those resources.

DSP Utilization Ratio is typically used in a collaborative workflow, where various components are initially designed separately, before being consolidated into the complete project. DSP Utilization Ratio allows you to budget DSP resources for each separate component.

DSP Utilization Ratio defines either:

- An absolute number of DSP slices, or
- A percentage of the total amount of resources available on the device.

The default is 100% of DSP resources available in the selected device.

Defining an absolute number, or a percentage, that exceeds DSP resources available in the selected device, is flagged, and XST uses no more resources than allowed by the device.

Instantiated DSP primitives are served first. XST allocates a corresponding amount from the total budget defined by DSP Utilization Ratio. Unused resources are taken advantage of by XST to implement inferred functions.

The defined budget may be exceeded in the following cases:

- The number of instantiated DSP blocks is higher than the defined budget. All DSP instantiations are always honored by XST, and it is your responsibility to ensure that the selected device can accommodate all instantiated DSP blocks.
- You have forced DSP implementation of inferred macros with Use DSP Block (USE_DSP48) set to yes.

**Caution!** When using Use DSP Block (USE_DSP48) with value yes, XST ignores both the maximum DSP allocation defined by DSP Utilization Ratio, and the amount of DSP resource actually available in the selected device. Your design may not fit in the device as a result. DSP Utilization Ratio works best with the Auto and Automax modes of Use DSP Block (USE_DSP48).

To disable automatic DSP resource management (for example, to see how many DSPs XST can potentially infer for a specific design), set DSP Utilization Ratio to -1 (or any negative value).

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices

**Applicable Elements**

Applies globally

**Propagation Rules**

Not applicable
Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-dsp_utilization_ratio number[%|#]
```

*where*

```
<integer> range is [-1 to 100]
```

*when*

% is used or both % and # are omitted.

To specify a *percent* of total slices use %. To specify an *absolute number* of slices use #.

The default is %.

- To specify 50% of DSP blocks of the target device:
  
  ```
  -dsp_utilization_ratio 50
  ```

- To specify 50% of DSP blocks of the target device:

  ```
  -dsp_utilization_ratio 50%
  ```

- To specify 50 DSP blocks:

  ```
  -dsp_utilization_ratio 50#
  ```

  There must be no space between the integer value and the percent (%) or pound (#) characters.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

**Process > Properties > Synthesis Options > DSP Utilization Ratio**

In ISE Design Suite, you can define the value of DSP Utilization Ratio only as a *percent*. You cannot define the value as an *absolute number* of slices.
FSM Style (FSM_STYLE)

The FSM Style (FSM_STYLE) constraint:

- Is both a global and a local constraint.
- Makes large Finite State Machine (FSM) components more compact and faster by implementing them in the block RAM resources.

Use FSM Style to direct XST to use block RAM resources rather than LUTs (default) to implement FSM Styles.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute fsm_style: string;
```

Specify as follows:

```vhdl
attribute fsm_style of {entity_name|signal_name}:
{entity|signal} is "{lut|bram}";
```

The default is lut.

Verilog Syntax Example

Place immediately before the instance, module, or signal declaration.

```verilog
(* fsm_style = "{lut|bram}" *)
```

XCF Syntax Example One

```xvf
MODEL "entity_name" fsm_style = {lut|bram};
```

XCF Syntax Example Two

```xvf
BEGIN MODEL "entity_name"
NET "signal_name" fsm_style = {lut|bram};
END;
```
XCF Syntax Example Three
BEGIN MODEL "entity_name"
INST "instance_name" fsm_style = {lut|bram};
END;

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Synthesis Options > FSM Style
Power Reduction (POWER)

Power Reduction (POWER) enables synthesis optimization techniques to reduce power consumption. By default, power optimizations are disabled.

Even if Power Reduction is enabled, XST still attempts to honor the primary optimization goal (speed or area) set by Optimization Goal (OPT_MODE). Xilinx® recommends that you carefully review whether the optimizations performed to reduce power consumption negatively impact your primary optimization goal.

Power optimizations available for Spartan®-6 devices and Virtex®-6 devices are primarily related to block RAMs. In particular, XST attempts to minimize the amount of simultaneously active block RAMs by properly using RAM enable features. For information on more precise control of RAM power optimizations, see RAM Style (RAM_STYLE).

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to:
- A component or entity (VHDL)
- A model or label (instance) (Verilog)
- A model or INST (in model) (XCF)
- The entire design (XST command line)

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```
attribute power:  string;
```

Specify as follows:

```
attribute power of {component name|entity_name} :
{component|entity} is "{yes|no}";
```

The default is no.

**Verilog Syntax Example**

Place immediately before the module declaration or instantiation:

```
(* power = "{yes|no}" *)
```

The default is no.
XCF Syntax Example
MODEL "entity_name" power = {yes|no|true|false};
The default is false.

XST Command Line Syntax Example
Define globally with the run command.
-power {yes|no}
The default is no.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Synthesis Options > Power Reduction
Read Cores (READ_CORES)

The Use Read Cores (READ_CORES) constraint:

- Enables or disables the ability of XST to read Electronic Data Interchange Format (EDIF) or NGC core files for timing estimation and device utilization control. By reading a specific core, XST is better able to optimize logic around the core, since it sees how the logic is connected.

- Must be disabled in some cases in order to obtain the desired results. For example, the PCI™ core must not be visible to XST, since the logic directly connected to the PCI core must be optimized differently as compared to other cores. Read Cores allows you to enable or disable read operations on a core by core basis.

For more information, see:

**Cores Processing**

The values for this constraint are:

- **no** (false)
  Disables cores processing

- **yes** (true)
  Enables cores processing, but maintains the core as a black box and does not further incorporate the core into the design.

- **optimize**
  Enables cores processing, and merges the core netlist into the overall design. This value is available through the XST command line mode only.

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices

**Applicable Elements**

Since Read Cores can be used with BoxType (BOX_TYPE), the set of objects on which the both constraints can be applied must be the same.

Applies to:

- A component or entity (VHDL)
- A model or label (instance) (Verilog)
- A model or INST (in model) (XCF)
- The entire design (XST command line)

If Read Cores is applied to at least a single instance of a block, then Read Cores is applied to all other instances of this block for the entire design.

**Propagation Rules**

Not applicable
Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute read_cores: string;
```

Specify as follows:

```vhdl
attribute read_cores of {component_name|entity_name} : {yes|no|optimize};
```

The default is yes.

**Verilog Syntax Example**

Place immediately before the module declaration or instantiation:

```verilog
(* read_cores = "{yes|no|optimize}" *)
```

The default is yes.

**XCF Syntax Example One**

```xcf
MODEL "entity_name" read_cores = {yes|no|true|false|optimize};
```

**XCF Syntax Example Two**

```xcf
BEGIN MODEL "entity_name"
INST "instance_name" read_cores = {yes|no|true|false|optimize};
END;
```

**XST Command Line Syntax Example**

```plaintext
-read_cores {yes|no|optimize}
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Read Cores

The optimize option is not available in ISE® Design Suite.
LUT Combining (LC)

The LUT Combining (LC) constraint:

- Enables the merging of LUT pairs with common inputs into single dual-output LUT6 elements in order to improve design area. This optimization process may reduce design speed.
- Has three values:
  - **auto**
    XST tries to make a trade-off between area and speed. **Auto** is the default for both Virtex®-6 devices and Spartan®-6 devices.
  - **area**
    XST performs maximum LUT combining to provide as small an implementation as possible.
  - **off**
    Disables LUT combining.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-lc {auto|area|off}
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

```
Process > Properties > Xilinx-Specific Options > LUT Combining
```
Chapter 12: XST FPGA Constraints (Non-Timing)

Map Logic on BRAM (BRAM_MAP)

The Map Logic on BRAM (BRAM_MAP) constraint:
- Is both a global and a local constraint.
- Maps an entire hierarchical block on the block RAM resources available in Virtex® and later devices.
- Has values of:
  - yes
  - no (default)

For more information, see Mapping Logic to Block RAM.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to BRAMs

Propagation Rules

Isolate the logic (including output register) to be mapped on RAM in a separate hierarchical level. Logic that does not fit on a single block RAM is not mapped. Ensure that the whole entity fits, not just part of it. The attribute BRAM_MAP is set on the instance or entity. If no block RAM can be inferred, the logic is passed to Global Optimization Goal (–glob_opt) where it is optimized. The macros are not inferred. Be sure that XST has mapped the logic.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example
Declare as follows:

attribute bram_map: string;

Specify as follows:

attribute bram_map of component_name: component is "{yes|no}";

Verilog Syntax Example
Place immediately before the module declaration or instantiation:

(* bram_map = "{yes|no}" *)

XCF Syntax Example One

MODEL "entity_name" bram_map = {yes|no|true|false};

XCF Syntax Example Two

BEGIN MODEL "entity_name"
INST "instance_name" bram_map = {yes|no|true|false};
END;
Max Fanout (MAX_FANOUT)

The Max Fanout (MAX_FANOUT) constraint:
- Is both a global and a local constraint.
- Limits the fanout of nets or signals. The value is an integer.
- Has a default value of 100000 (One Hundred Thousand).

Large fanouts can interfere with routability. XST tries to limit fanout by duplicating gates or by inserting buffers. This limit:
- Is not a technology limit but a guide to XST.
- Is not always observed, especially when this limit is small (less than 30).

In most cases, fanout control is performed by duplicating the gate driving the net with a large fanout. If the duplication cannot be performed, buffers are inserted. These buffers are protected against logic trimming at the implementation level by defining Keep (KEEP) in the NGC file.

If the register replication option is set to no, only buffers are used to control fanout of flip-flops and latches.

Max Fanout is global for the design, but you can use constraints to control maximum fanout independently for each entity or module or for individual signals.

If the actual net fanout is less than the Max Fanout value, XST behavior depends on how Max Fanout is specified.
- If the value of Max Fanout is set in ISE® Design Suite, in the command line, or is applied to a specific hierarchical block, XST interprets its value as a guidance.
- If Max Fanout is applied to a specific net, XST does not perform logic replication. Putting Max Fanout on the net may prevent XST from having better timing optimization.

For example, suppose that the critical path goes through the net, which actual fanout is 80 and set Max Fanout value to 100. If Max Fanout is specified in ISE Design Suite, XST can replicate it, trying to improve timing. If Max Fanout is applied to the net itself, XST does not perform logic replication.

Max Fanout can also take the value reduce. This value has no direct meaning to XST. It is considered only during placement and routing. Until then, fanout control is deferred.

Max Fanout with a value of reduce can be applied only to a net. It cannot be applied globally.

XST disables any logic optimization related to the designated net, meaning that it is preserved in the post-synthesis netlist, and that a MAX_FANOUT=reduce property is attached to it. If a more global Max Fanout constraint was defined with an integer value (either on the command line, or with an attribute attached to the entity or module containing the considered net), then:
- The reduce value takes precedence.
- The integer value is ignored for the designated net.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Exception: When Max Fanout takes the value reduce, it can be applied only to a signal.
Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute max_fanout: string;
```

Specify as follows:

```vhdl
attribute max_fanout of {signal_name | entity_name}: {signal|entity} is "integer";
```

OR

```vhdl
attribute max_fanout of {signal_name}: {signal} is "reduce";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* max_fanout = "integer" *)
```

OR

```verilog
(* max_fanout = "reduce" *)
```

**XCF Syntax Example One**

```xilinx
MODEL "entity_name" max_fanout=integer;
```

**XCF Syntax Example Two**

```xilinx
BEGIN MODEL "entity_name"
NET "signal_name" max_fanout=integer;
END;
```

**XCF Syntax Example Three**

```xilinx
BEGIN MODEL "entity_name"
NET "signal_name" max_fanout="reduce";
END;
```

**XST Command Line Syntax Example**

```xilinx
-max_fanout integer
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Max Fanout
Move First Stage (MOVE_FIRST_STAGE)

Move First Stage (MOVE_FIRST_STAGE) controls the retiming of registers with paths coming from primary inputs. Both Move First Stage and Move Last Stage (MOVE_LAST_STAGE) relate to Register Balancing.

- A flip-flop (FF in the figure) belongs to the First Stage if it is on the paths coming from primary inputs
- A flip-flop belongs to the Last Stage if it is on the paths going to primary outputs.

**Move First Stage Diagram**

During register balancing:
- First Stage flip-flops are moved forward
- Last Stage flip-flops are moved backward

This process can dramatically increase input-to-clock and clock-to-output timing, which is not desirable. To prevent this, use OFFSET_IN_BEFORE and OFFSET_IN_AFTER.

You can use two additional constraints if:
- The design does not have strong requirements, or
- You want to see the first results without touching the first and last flip-flop stages.

The additional constraints are:
- MOVE_FIRST_STAGE
- MOVE_LAST_STAGE

Both constraints can have two values: yes and no.
- MOVE_FIRST_STAGE=no prevents the first flip-flop stage from moving
- MOVE_LAST_STAGE=no prevents the last flip-flop stage from moving

Several constraints influence register balancing.
For more information, see:
Register Balancing (REGISTER_BALANCING)

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices
Applicable Elements

Applies to the following only:
- Entire design
- Single modules or entities
- Primary clock signal

Propagation Rules

For Move First Stage propagation rules, see the figure above.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute move_first_stage : string;
```

Specify as follows:

```vhdl
attribute move_first_stage of {entity_name|signal_name}: {signal|entity} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* move_first_stage = "{yes|no}" *)
```

**XCF Syntax Example One**

```xilinx
MODEL "entity_name" move_first_stage={yes|no|true|false};
```

**XCF Syntax Example Two**

```xilinx
BEGIN MODEL "entity_name"
NET "primary_clock_signal" move_first_stage={yes|no|true|false};
END;
```

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-move_first_stage {yes|no}
```

The default is `yes`.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Move First Flip-Flop Stage
Move Last Stage (MOVE_LAST_STAGE)

Move Last Stage (MOVE_LAST_STAGE) controls the retiming of registers with paths going to primary outputs. Both Move Last Stage and Move First Stage (MOVE_FIRST_STAGE) relate to Register Balancing.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to the following only:
- Entire design
- Single modules or entities
- Primary clock signal

Propagation Rules

See Move First Stage (MOVE_FIRST_STAGE).

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute move_last_stage : string;
```

Specify as follows:

```vhdl
attribute move_last_stage of {entity_name|signal_name}:
{signal|entity} is "{yes|no}"
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* move_last_stage = "{yes|no}" *)
```

**XCF Syntax Example One**

```xilinx
MODEL "entity_name"{move_last_stage={yes|no|true|false}};
```

**XCF Syntax Example Two**

```xilinx
BEGIN MODEL "entity_name"
NET "primary_clock_signal" move_last_stage={yes|no|true|false};
END;
```

**XST Command Line Syntax Example**

Define globally with the run command.

```
-move_last_stage {yes|no}
```

The default is yes.
ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Xilinx-Specific Options > Move Last Flip-Flop Stage
Multiplier Style (MULT_STYLE)

Multiplier Style (MULT_STYLE) controls the way the macrogenerator implements the multiplier macros.
The values for this constraint are:

- **auto** (default)
  XST looks for the best implementation for each considered macro.
- **block**
- **pipe_block**
  Used to pipeline DSP48-based multipliers.
- **kcm**
- **csd**
- **lut**
- **pipe_lut**
  For pipeline slice-based multipliers. The implementation style can be manually forced to use block multiplier or LUT resources.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Multiplier Style is applicable only through an HDL attribute. It is not available as a command line option.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute mult_style: string;
```

Specify as follows:

```vhdl
attribute mult_style of {signal_name|entity_name}: {signal|entity} is "{auto|block|pipe_block|kcm|csd|lut|pipe_lut}";
```

The default is **auto**.
Verilog Syntax Example

Place immediately before the module or signal declaration:

(* mult_style = "(auto|block|pipe_block|kcm|csd|lut|pipe_lut)" *)

The default is auto.

XCF Syntax Example One

MODEL "entity_name"
mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};

XCF Syntax Example Two

BEGIN MODEL "entity_name"

NET "signal_name"
mult_style={auto|block|pipe_block|kcm|csd|lut|pipe_lut};

END;
Number of Global Clock Buffers (–bufg)

Number of Global Clock Buffers (–bufg) controls the maximum number of BUFG elements created by Expressions.

The value is an integer. The default value:
- Depends on the target device.
- Equals the maximum number of available BUFG elements.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

```bash
–bufg integer
```

The value is an integer. The default values are different for different architectures. The defaults for selected architectures are shown in the following table. The number of BUFG elements cannot exceed the maximum number of BUFG elements for the target device.

<table>
<thead>
<tr>
<th>Device</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan®-6</td>
<td>16</td>
</tr>
<tr>
<td>Virtex®-6</td>
<td>32</td>
</tr>
</tbody>
</table>

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Number of Clock Buffers
Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES)

By default, XST does not optimize instantiated primitives in Hardware Description Language (HDL) designs.

The Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) constraint:

- Deactivates the default.
- Allows XST to optimize Xilinx® library primitives that have been instantiated in an HDL design.

Optimization of instantiated primitives is limited by the following:

- If an instantiated primitive has specific constraints such as RLOC applied, XST preserves it as is.
- Not all primitives are considered by XST for optimization. Such hardware elements as MULT18x18, block RAMs, and DSP48 are not optimized (modified) even if optimization of instantiated primitives is enabled.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to the designated hierarchical blocks, components, and instances.

Propagation Rules

Applies to the component or instance to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax Examples

- Attach to a valid instance
- Attribute Name
  OPTIMIZE_PRIMITIVES
- Attribute Values
  - yes
  - no (default)

VHDL Syntax Example

Declare as follows:

```
attribute optimize_primitives: string;
```

Specify as follows:

```
attribute optimize_primitives of
{component_name|entity_name|label_name }: {component|entity|label} is "{yes|no}"
```
Verilog Syntax Example
Place immediately before the instance, module or signal declaration:

(* optimize_primitives = "{yes|no}" *)

XCF Syntax Example
MODEL "entity_name" optimize_primitives = {yes|no|true|false};

XST Command Line Syntax Example
Define globally with the run command.

-optimize_primitives {yes|no}
The default is no.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Xilinx-Specific Options > Optimize Instantiated Primitives
Pack I/O Registers Into IOBs (IOB)

Pack I/O Registers Into IOBs (IOB) packs flip-flops in the I/Os to improve input/output path timing.

When Pack I/O Registers Into IOBs is set to auto, the action XST takes depends on the Optimization setting:

- **area**
  XST packs registers as tightly as possible to the IOBs in order to reduce the number of slices occupied by the design.

- **speed**
  XST packs registers to the IOBs provided they are not covered by timing constraints (in other words, they are not taken into account by timing optimization). For example, if you specify a PERIOD constraint, XST packs a register to the IOB if it is not covered by the period constraint. If a register is covered by timing optimization, but you do want to pack it to an IOB, you must apply the IOB constraint locally to the register.

For more information about this constraint, see the Constraints Guide.
RAM Extraction (RAM_EXTRACT)

RAM Extraction (RAM_EXTRACT) enables or disables RAM macro inference. The values for this constraint are:
- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute ram_extract: string;
```

Specify as follows:

```vhdl
attribute ram_extract of {signal_name|entity_name}:
{signal|entity} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* ram_extract = "{yes|no}" *)
```

**XCF Syntax Example One**

```xilinx
RAM Extraction Syntax MODEL "entity_name"
ram_extract={yes|no|true|false};
```

**XCF Syntax Example Two**

```xilinx
BEGIN MODEL "entity_name"
NET "signal_name" ram_extract={yes|no|true|false};
END;
```
XST Command Line Syntax Example
Define globally with the run command.

-ram_extract {yes|no}
The default is yes.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > HDL Options > RAM Extraction
Chapter 12: XST FPGA Constraints (Non-Timing)

RAM Style (RAM_STYLE)

RAM Style (RAM_STYLE) controls the way the macrogenerator implements the inferred RAM macros.

Constraint Values

The values for this constraint are:

- **auto** (default)
  Instructs XST to look for the best implementation for each inferred RAM, based on:
  - Whether the description style allows block RAM implementation (synchronous data read)
  - Available block RAM resources on the targeted device

- **distributed**
  Manually forces the implementation to distributed RAM resources

- **pipe_distributed**
  When an inferred RAM is implemented on LUT resources, and several distributed RAM primitives are required to accommodate its size, multiplexing logic is created on the RAM data output path. The pipe_distributed value instructs XST to use any latency stages available behind the RAM to pipeline this logic.
  - May be specified only as:
    - VHDL attribute
    - Verilog attribute
    - XST Constraint File (XCF) constraint

- **block**
  Manually forces the implementation to block RAM. Actual implementation on block RAM remains conditional on:
  - A properly synchronized data read, and
  - Available resources on the device
Enabling Two Levels of Optimizations

Use block_power1 and block_power2 to enable two levels of optimizations aimed at reducing power consumption of RAMs implemented on block resources.

- **block_power1**
  - Is intended to have minimal impact on the primary optimization goal defined by Optimization Goal (OPT_MODE) (area or speed)
  - Is the selected mode when general power optimizations are enabled with Power Reduction (POWER)
  - May be specified only as:
    ♦ VHDL attribute
    ♦ Verilog attribute
    ♦ XST Constraint File (XCF) constraint

- **block_power2**
  - Allows further power reduction
  - Can significantly impact area and speed
  - May be specified only as:
    ♦ VHDL attribute
    ♦ Verilog attribute
    ♦ XST Constraint File (XCF) constraint

For more information on those optimization techniques, see Block RAM Power Reduction

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute ram_style: string;
```

Specify as follows:

```vhdl
attribute ram_style of {signal_name|entity_name}: {signal|entity} is "(auto|block|distributed|pipe_distributed|block_power1|block_power2)";
```

The default is **auto**.
Verilog Syntax Example
Place immediately before the module or signal declaration:

(* ram_style = "\{(auto|block|distributed|pipe_distributed|block_power1|block_power2)\} *）

The default is auto.

XCF Syntax Example One
MODEL "entity_name" ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};

XCF Syntax Example Two
BEGIN MODEL "entity_name"
  NET "signal_name" ram_style={auto|block|distributed|pipe_distributed|block_power1|block_power2};
END;

XST Command Line Syntax Example
Define globally with the run command.

-ram_style {auto|block|distributed}

The default is auto.

The pipe_distributed value is not accessible through the command line.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.

Process > Properties > HDL Options > RAM Style
BRAM Read-First Implementation (RDADDR_COLLISION_HWCONFIG)

The BRAM Read-First Implementation (RDADDR_COLLISION_HWCONFIG) constraint:

- Allows you to control implementation of a block RAM described with a read-first synchronization.
- Applies to Virtex®-6 devices only.
- Is ignored if:
  - You are targeting a device family other than Virtex-6 devices, or
  - The described read-write synchronization is not read-first.
- Has two values:
  - delayed_write
    The block RAM is configured to avoid memory collision. While conflicts are avoided, this configuration sacrifices some performance compared to write-first and no-change synchronization.
  - performance
    Maximizes performance of the read-first mode. Performance is comparable to that obtained with write-first and no-change modes. However, you must ensure that memory collisions do not occur.

For inferred RAMs, the default value differs depending on the number of RAM ports:

- Single-port: performance
  Memory collisions are possible only when the RAM is dual-port. The performance mode can therefore be safely enforced when a memory is single-port.
- Dual-port: delayed_write

BRAM Read-First Implementation can be applied both to:

- An instantiated block RAM primitive, and
- An inferred RAM

In the latter case, BRAM Read-First Implementation does not tell XST that the described memory has a read-first synchronization. This is done through proper Hardware Description Language (HDL) coding.

For more information, see:
Block RAM Read/Write Synchronization

Architecture Support

Applies to Virtex-6 devices only. Does not apply to Spartan®-6 devices.

Applicable Elements

Applicable locally, through a VHDL or Verilog attribute, or an XST Constraint File (XCF) constraint, to:

- A particular block (entity, architecture, component)
- A signal describing the RAM

BRAM Read-First Implementation is not available as a command-line or ISE® Design Suite option.
Propagation Rules

Applies to the entity, component, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute RDADDR_COLLISION_HWCONFIG: string;
```

Specify as follows:

```vhdl
attribute RDADDR_COLLISION_HWCONFIG of
"entity_name|component_name|signal_name}:{entity|component|signal}
is "[delayed_write|performance]";
```

Verilog Syntax Example

Place immediately before the instance, module, or signal declaration.

```verilog
(*RDADDR_COLLISION_HWCONFIG = "[delayed_write|performance]" *)
```

XCF Syntax Example One

```xilinx
MODEL "entity_name" RDADDR_COLLISION_HWCONFIG
={delayed_write|performance};
```

XCF Syntax Example Two

```xilinx
BEGIN MODEL "entity_name"

NET "signal_name" RDADDR_COLLISION_HWCONFIG
={delayed_write|performance};

END;
```
Reduce Control Sets (REDUCE_CONTROL_SETS)

Use Reduce Control Sets (REDUCE_CONTROL_SETS) to reduce the number of control sets and, as a consequence, reduce the design area. Reducing the number of control sets improves the packing process in map, and therefore reduces the number of used slices even if the number of LUTs increases.

Reducing the number of unique control sets applies only to synchronous control signals (synchronous set/reset and clock enable). Use Reduce Control Sets has no effect on asynchronous sets/reset logic.

Reduce Control Sets supports two values:

- **auto** (default)
  
  XST optimizes automatically, and reduces the existing control sets in the design.

- **no**
  
  XST performs no control set optimization.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
-reduce_control_sets {auto|no}
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

```
Process > Properties > Xilinx Specific Options > Reduce Control Sets
```
Register Balancing (REGISTER_BALANCING)

Register Balancing (REGISTER_BALANCING) enables flip-flop retiming. The main goal of register balancing is to move flip-flops and latches across logic to increase clock frequency.

The two categories of Register Balancing are:
- Forward Register Balancing
- Backward Register Balancing

Forward Register Balancing

Forward Register Balancing moves a set of flip-flops at the inputs of a LUT to a single flip-flop at its output.

```
Forward Register Balancing

When replacing several flip-flops with one, select the name based on the name of the LUT across which the flip-flops are moving as shown in the following:

$LutName_{FRBId}$
```

Backward Register Balancing

Backward Register Balancing moves a flip-flop at the output of a LUT to a set of flip-flops at its inputs.

```
Backward Register Balancing
```

Chapter 12: XST FPGA Constraints (Non-Timing)

As a consequence the number of flip-flops in the design can be increased or decreased. The new flip-flop has the same name as the original flip-flop with an indexed suffix: OriginalFFName_BRBId

Register Balancing Values

The values for this constraint are:

- **yes**
  Both forward and backward retiming are allowed.

- **no** (default)
  Neither forward nor backward retiming is allowed.

- **forward**
  Only forward retiming is allowed.

- **backward**
  Only backward retiming is allowed.

- **true** (XCF only)

- **false** (XCF only)

Additional Constraints That Affect Register Balancing

Two additional constraints control register balancing:

- **Move First Stage** (MOVE_FIRST_STAGE)
- **Move Last Stage** (MOVE_LAST_STAGE)

Several other constraints also influence register balancing:

- **Keep Hierarchy** (KEEP_HIERARCHY)
  - If the hierarchy is preserved, flip-flops are moved only inside the block boundaries.
  - If the hierarchy is flattened, flip-flops may leave the block boundaries.

- **Pack I/O Registers Into IOBs** (IOB)
  If IOB=TRUE, register balancing is not applied to the flip-flops having this property.

- **Optimize Instantiated Primitives** (OPTIMIZE_PRIMITIVES)
  - Instantiated flip-flops are moved only if OPTIMIZE_PRIMITIVES=YES.
  - Flip-flops are moved across instantiated primitives only if OPTIMIZE_PRIMITIVES=YES.

- **Keep** (KEEP)
  - If applied to the output flip-flop signal, the flip-flop is not moved forward. See the following figure.
  - If applied to the input flip-flop signal, the flip-flop is not moved backward.
  - If applied to both the input and output of the flip-flop, it is equivalent to REGISTER_BALANCING=no.
Architecture Support
Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements
Register Balancing can be applied:
- Globally to the entire design using the command line or ISE® Design Suite
- To an entity or module
- To a signal corresponding to the flip-flop description (RTL)
- To a flip-flop instance
- To the Primary Clock Signal
In this case, the register balancing is performed only for flip-flops synchronized by this clock.

Propagation Rules
Applies to the entity, module, or signal to which it is attached

Syntax Examples
The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example
Declare as follows:
attribute register_balancing: string;
Specify as follows:
attribute register_balancing of {signal_name|entity_name}:
{signal|entity} is "(yes|no|forward|backward)";

Verilog Syntax Example
Place immediately before the module or signal declaration:
* register_balancing = "(yes|no|forward|backward)" *
The default is no.

XCF Syntax Example One
MODEL "entity_name"
register_balancing=(yes|no|true|false|forward|backward);
XCF Syntax Example Two

BEGIN MODEL "entity_name"
NET "primary_clock_signal"
register_balancing={yes|no|true|false|forward|backward};
END;

XCF Syntax Example Three

BEGIN MODEL "entity_name"
INST "instance_name"
register_balancing={yes|no|true|false|forward|backward};
END;

XST Command Line Syntax Example

Define globally with the run command.
-register_balancing {yes|no|forward|backward}
The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.
Process > Properties > Xilinx-Specific Options > Register Balancing
Register Duplication (REGISTER_DUPLICATION)

Register Duplication (REGISTER_DUPLICATION) enables or disables register replication.

The values for this constraint are:
- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

When Register Duplication is set to **yes**, register replication is enabled, and is performed during timing optimization and fanout control.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute register_duplication: string;
```

Specify as follows for an entity:

```vhdl
attribute register_duplication of entity_name: entity is "{yes|no}"
```

Specify as follows for a signal:

```vhdl
attribute register_duplication of signal_name: signal is "{yes|no}"
```

**Verilog Syntax Example**

Place immediately before the module declaration or instantiation, or the signal declaration:

```verilog
(* register_duplication = "{yes|no}" *)
```

**XCF Syntax Example One**

```xilinx
MODEL "entity_name" register_duplication={yes|no|true|false};
```
XCF Syntax Example Two

BEGIN MODEL "entity_name"

NET "signal_name" register_duplication={yes|no|true|false};

END;

XST Command Line Syntax Example

Define globally with the run command.

-register_duplication {yes|no}

The default is yes.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Register Duplication
ROM Extraction (ROM_EXTRACT)

ROM Extraction (ROM_EXTRACT) enables or disables ROM macro inference. The values for this constraint are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

A ROM can usually be inferred from a case statement where all assigned contexts are constant values.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to a design element or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute rom_extract: string;
```

Specify as follows:

```vhdl
attribute rom_extract of {signal_name|entity_name}:
{signal|entity} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* rom_extract = "{yes|no}" *)
```

**XCF Syntax Example One**

```xscript
MODEL "entity_name" rom_extract={yes|no|true|false};*
```

**XCF Syntax Example Two**

```xscript
BEGIN MODEL "entity_name"
NET "signal_name" rom_extract={yes|no|true|false};
END;
```
XST Command Line Syntax Example
Define globally with the run command.

-rom_extract {yes|no}
The default is yes.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > HDL Options > ROM Extraction
Chapter 12: XST FPGA Constraints (Non-Timing)

ROM Style (ROM_STYLE)

ROM Style (ROM_STYLE) controls the manner in which the macrogenerator implements the inferred ROM macros.

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

The values for this constraint are:

- **auto** (default)
- **block**
- **distributed**

XST looks for the best implementation for each inferred ROM. The implementation style can be manually forced to use block RAM or LUT resources.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to an entity, module, or signal

Propagation Rules

Applies to the entity, module, or signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

Declare as follows:

```
attribute rom_style: string;
```

Specify as follows:

```
attribute rom_style of {signal_name|entity_name}: (signal|entity) is 
"{auto|block|distributed}";
```

The default is **auto**.

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```
(* rom_style = "auto|block|distributed" *)
```

The default is **auto**.

**XCF Syntax Example One**

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

```
MODEL "entity_name" rom_style={auto|block|distributed};
```
XCF Syntax Example Two

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

BEGIN MODEL "entity_name"
NET "signal_name" rom_style={auto|block|distributed};
END;

XST Command Line Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

Define globally with the run command.

-rom_style {auto|block|distributed}

The default is auto.

ISE Design Suite Syntax Example

ROM Extraction (ROM_EXTRACT) must be set to yes for ROM Style to take effect.

Define globally in ISE Design Suite.

Process > Properties > HDL Options > ROM Style
Shift Register Extraction (SHREG_EXTRACT)

Shift Register Extraction (SHREG_EXTRACT) enables or disables shift register macro inference.

The values for this constraint are:

- **yes** (default)
- **no**
- **true** (XCF only)
- **false** (XCF only)

Enabling Shift Register Extraction results in the usage of dedicated hardware resources such as SRL16 and SRLC16.

For more information, see:

Chapter 7, XST HDL Coding Techniques

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to a design element or signal

Propagation Rules

Applies to the design elements or signals to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute shreg_extract : string;
```

Specify as follows:

```vhdl
attribute shreg_extract of {signal_name|entity_name}: {signal|entity} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* shreg_extract = "{yes|no}" *)
```

**XCF Syntax Example One**

```verilog
MODEL "entity_name" shreg_extract={yes|no|true|false};
```
XCF Syntax Example Two

BEGIN MODEL "entity_name"

NET "signal_name" shreg_extract={yes|no|true|false};

END;

XST Command Line Syntax Example

Define globally with the run command.

–shreg_extract {yes|no}

The default is yes.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > HDL Options > Shift Register Extraction
Shift Register Minimum Size (SHREG_MIN_SIZE)

The Shift Register Minimum Size (SHREG_MIN_SIZE) constraint:

- Allows you to control the minimum length of shift registers that are inferred and implemented using SRL-type resources. Shift registers below the specified limit are implemented using simple flip-flops.
- Takes a natural value of 2 or higher. The default value is 2.

Using SRL-type resources excessively to implement small shift register macros (such as 2-bit shift registers) may lead to excessive placement restrictions for other design elements. This may eventually degrade circuit performance. Shift Register Minimum Size allows you to avoid this problem by causing XST to force implementation of shift registers below a designated length using simple flip-flop resources. This option may be particularly useful in Spartan®-6 devices, where the availability of a single SliceM for every four Slices (SliceL, SliceM, SliceX, SliceY), makes this element particularly scarce and valuable, and may justify saving it for better use, such as real LUT RAM applications.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Only available as an XST option, defining a global inference threshold for the whole design. If you need to more finely control inference of individual shift registers, use this option in conjunction with the Shift Register Extraction (SHREG_EXTRACT) constraint, which can be applied to designated design elements.

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the run command.

```
-run -shreg_min_size integer
```

The default is 2.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

```
Process > Properties > HDL Options > Shift Register Minimum Size
```
Use Low Skew Lines (USELOWSKEWLINES)

The Use Low Skew Lines (USELOWSKEWLINES) constraint:

- Is a basic routing constraint.
- Prevents XST from using dedicated clock resources and logic replication during synthesis based on the value of Max Fanout (MAX_FANOUT).
- Specifies the use of low skew routing resources for any net.

For more information about this constraint, see the Constraints Guide.
Chapter 12: XST FPGA Constraints (Non-Timing)

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO)

Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO) defines the area size in absolute numbers or percent of total numbers of LUT-FF pairs that XST must not exceed during timing optimization.

If the area constraint cannot be satisfied, XST will make timing optimization regardless of the area constraint. To disable automatic resource management, specify -1 as a constraint value.

For more information, see:

Speed Optimization Under Area Constraint

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to a VHDL entity or Verilog module

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

attribute slice_utilization_ratio: string;

Specify as follows:

attribute slice_utilization_ratio of entity_name: entity is
  "integer";

attribute slice_utilization_ratio of entity_name: entity is
  "integer%";

attribute slice_utilization_ratio of entity_name: entity is
  "integer#";

In the preceding example, XST interprets the integer values in the first two attributes as a percentage and in the last attribute as an absolute number of slices or FF-LUT pairs.

Verilog Syntax Example

Place immediately before the module declaration or instantiation:

(* slice_utilization_ratio = "integer" *)

(* slice_utilization_ratio = "integer%" *)

(* slice_utilization_ratio = "integer#" *)

In the preceding examples XST interprets the integer values in the first two attributes as a percentage and in the last attribute as an absolute number of slices or FF-LUT pairs.
XST FPGA Constraints (Non-Timing)

Chapter 12: XST FPGA Constraints (Non-Timing)

XCF Syntax Example One

MODEL "entity_name" slice_utilization_ratio=integer;

XCF Syntax Example Two

MODEL "entity_name" slice_utilization_ratio="integer%";

XCF Syntax Example Three

MODEL "entity_name" slice_utilization_ratio="integer#"; *

In the preceding examples, XST interprets the integer values in the first two lines as a percentage and in the last line as an absolute number of slices or FF-LUT pairs.

There must be no space between the integer value and the percent (%) or pound (#) characters.

The integer value range is -1 to 100 when percent (%) is used or both percent (%) and pound (#) are omitted.

You must surround the integer value and the percent (%) and pound (#) characters with double quotes ("...") because the percent (%) and pound (#) characters are special characters in the XST Constraint File (XCF).

XST Command Line Syntax Example

Define globally with the run command.

-slice_utilization_ratio integer
-slice_utilization_ratio integer%
-slice_utilization_ratio integer#

In the preceding examples, XST interprets the integer values in the first two lines as a percentage and in the last line as an absolute number of slices or FF-LUT pairs.

The integer value range is -1 to 100 when percent (%) is used or both percent (%) and pound (#) are omitted.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

• Process > Properties > Synthesis Options > Slice Utilization Ratio, or
• Process > Properties > Synthesis Options > LUT-FF Pairs Utilization Ratio.

In ISE® Design Suite, you can define the value of Slice (LUT-FF Pairs) Utilization Ratio only as a percentage. You cannot define the value as an absolute number of slices.
Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN)

The Slice (LUT-FF Pairs) Utilization Ratio Delta (SLICE_UTILIZATION_RATIO_MAXMARGIN) constraint:

- Is closely related to Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO).
- Defines the tolerance margin for Slice (LUT-FF Pairs) Utilization Ratio (SLICE_UTILIZATION_RATIO).

The value of the parameter can be defined as:

- A percentage, or
- An absolute number of slices or LUT-FF Pairs

If the ratio is within the margin set, the constraint is met and timing optimization can continue.

For more information, see: Speed Optimization Under Area Constraint

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to a VHDL entity or Verilog module

Propagation Rules

Applies to the entity or module to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute slice_utilization_ratio_maxmargin: string;
```

Specify as follows:

```vhdl
attribute slice_utilization_ratio_maxmargin of entity_name :
entity is "integer";
attribute slice_utilization_ratio_maxmargin of entity_name :
entity is "integer%";
attribute slice_utilization_ratio_maxmargin of entity_name :
entity is "integer#";
```

In the preceding examples, XST interprets the integer values in the first two attributes as a percentage, and in the last attribute as an absolute number of slices or FF-LUT pairs.

The integer value range is 0 to 100 when percent (%) is used or both percent (%) and pound(#) are omitted.
Verilog Syntax Example
Place immediately before the module declaration or instantiation:

(* slice_utilization_ratio_maxmargin = "integer" *)
(* slice_utilization_ratio_maxmargin = "integer%" *)
(* slice_utilization_ratio_maxmargin = "integer#" *)

In the preceding examples, XST interprets the integer values in the first two attributes as a percentage, and in the last attribute as an absolute number of slices or FF-LUT pairs.

XCF Syntax Example One
MODEL "entity_name" slice_utilization_ratio_maxmargin=integer;

XCF Syntax Example Two
MODEL "entity_name" slice_utilization_ratio_maxmargin="integer%";

XCF Syntax Example Three
MODEL "entity_name"
slice_utilization_ratio_maxmargin="integer#";

In the preceding example, XST interprets the integer values in the first two lines as a percentage and in the last line as an absolute number of slices or FF-LUT pairs.

There must be no space between the integer value and the percent (%) or pound (#) characters.

You must surround the integer value and the percent (%) and pound (#) characters with double quotes because the percent (%) and pound (#) characters are special characters in the XST Constraint File (XCF).

The integer value range is 0 to 100 when percent (%) is used or both percent (%) and pound (#) are omitted.

XST Command Line Syntax Example
Define globally with the run command.

-slice_utilization_ratio_maxmargin integer
-slice_utilization_ratio_maxmargin integer%
-slice_utilization_ratio_maxmargin integer#

In the preceding example, XST interprets the integer values in the first two lines as a percentage and in the last line as an absolute number of slices or FF-LUT pairs.

The integer value range is 0 to 100 when percent (%) is used or both percent (%) and pound (#) are omitted.
Map Entity on a Single LUT (LUT_MAP)

Map Entity on a Single LUT (LUT_MAP) forces XST to map a single block into a single LUT. If a described function on a Register Transfer Level (RTL) description does not fit in a single LUT, XST issues an error message.

Use the UNISIM library to directly instantiate LUT components in the Hardware Description Language (HDL) code. To specify a function that a particular LUT must execute, apply INIT to the instance of the LUT. To place an instantiated LUT or register in a particular slice, apply RLOC to the same instance.

LUT INIT functions and different methods can be used to achieve this. Alternatively, you can describe the function that you want to map onto a single LUT in the HDL source code in a separate block. Attaching LUT_MAP to this block indicates to XST that this block must be mapped on a single LUT. XST automatically calculates the INIT value for the LUT and preserves this LUT during optimization.

For more information, see:

**Mapping Logic to LUTs**

XST automatically recognizes the Synplify xc_map constraint.

**Architecture Support**

Applies to Virtex®-6, Spartan®-6, and 7 series devices

**Applicable Elements**

Applies to an entity or module

**Propagation Rules**

Applies to the entity or module to which it is attached

**Syntax Examples**

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```
attribute lut_map: string;
```

Specify as follows:

```
attribute lut_map of entity_name: entity is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module declaration or instantiation:

```
(* lut_map = "{yes|no}" *)
```

**XCF Syntax Example**

```
MODEL "entity_name" lut_map={yes|no|true|false};
```
Use Carry Chain (USE_CARRY_CHAIN)

The Use Carry Chain constraint:

- Is both a global and a local constraint.
- Can deactivate carry chain use for macro generation.

Although XST uses carry chain resources to implement certain macros, you can sometimes obtain better results by not using carry chain.

The values for this constraint are:

- yes (default)
- no

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies globally, or to signals

Propagation Rules

Applies to the signal to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax Examples

- Attach to a valid instance
- Attribute Name
  USE_CARRY_CHAIN
- Attribute Values
  - yes
  - no

VHDL Syntax Example

Declare as follows:

```vhdl
attribute use_carry_chain:  string;
```

Specify as follows:

```vhdl
attribute use_carry_chain of signal_name:  signal is "(yes|no)";
```

Verilog Syntax Example

Place immediately before the signal declaration:

```verilog
(* use_carry_chain = "(yes|no)" *)
```

XCF Syntax Example One

```verilog
MODEL "entity_name" use_carry_chain={yes|no|true|false};
```
XCF Syntax Example Two
BEGIN MODEL "entity_name"
NET "signal_name" use_carry_chain={yes|no|true|false};
END;

XST Command Line Syntax Example
Define globally with the run command.
-use_carry_chain {yes|no}
The default is yes.
Convert Tristates to Logic (TRISTATE2LOGIC)

Use Convert Tristates to Logic to replace internal tristates with logic.

Since some devices do not support internal tristates, XST automatically replaces tristates with equivalent logic. Because the logic generated from tristates can be combined and optimized with surrounding logic, replacing internal tristates with logic can increase speed, and in some cases, lead to better area optimization. In general, however, replacing tristate with logic increases area. If your optimization goal is area, set Convert Tristates to Logic to no.

Convert Tristates to Logic Limitations

- Only internal tristates are replaced by logic. The tristates of the top module connected to output pads are preserved.
- Internal tristates are not replaced by logic for modules when incremental synthesis is active.
- XST is unable to replace a tristate by logic when:
  - The tristate is connected to a black box.
  - The tristate is connected to the output of a block, and the hierarchy of the block is preserved.
  - The tristate is connected to a top-level output.
  - Convert Tristates to Logic is set to no on the block where tristates are placed, or on the signals to which tristates are connected.

The values for this constraint are:

- yes (default)
- no
- true (XCF only)
- false (XCF only)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to an entire design through the XST command line, to a particular block (entity, architecture, component), or to a signal.

Propagation Rules

Applies to an entity, component, module, signal, or instance to which it is attached
Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**VHDL Syntax Example**

Declare as follows:

```vhdl
attribute tristate2logic: string;
```

Specify as follows:

```vhdl
attribute tristate2logic of
{entity_name|component_name|signal_name}:
{entity|component|signal} is "{yes|no}";
```

**Verilog Syntax Example**

Place immediately before the module or signal declaration:

```verilog
(* tristate2logic = "{yes|no}" *)
```

**XCF Syntax Example One**

```xcf
MODEL "entity_name" tristate2logic={yes|no|true|false};
```

**XCF Syntax Example Two**

```xcf
BEGIN MODEL "entity_name"
NET "signal_name" tristate2logic={yes|no|true|false};
END;
```

**XST Command Line Syntax Example**

Define globally with the `run` command.

`-tristate2logic {yes|no}`

The default is yes.

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

`Process > Properties > Xilinx-Specific Options > Convert Tristates to Logic`
Use Clock Enable (USE_CLOCK_ENABLE)

Use Clock Enable (USE_CLOCK_ENABLE) enables or disables the clock enable function in flip-flops. The disabling of the clock enable function is typically used for ASIC prototyping.

When Use Clock Enable is set to no or false, XST avoids using Clock Enable resources in the final implementation. For some designs, putting Clock Enable on the data input of the flip-flop optimizes logic and gives better quality of results.

In auto mode, XST tries to estimate a trade-off between using a dedicated clock enable input of a flip-flop input and putting clock enable logic on the D input of a flip-flop. Where a flip-flop is instantiated by you, XST removes the clock enable only if Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) is set to yes.

The values for this constraint are:
- auto (default)
- yes
- no
- true (XCF only)
- false (XCF only)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to:
- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

Propagation Rules

Applies to an entity, component, module, signal, or instance to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```vhdl
attribute use_clock_enable: string;
```

Specify as follows:

```vhdl
attribute use_clock_enable of
{entity_name|component_name|signal_name|instance_name}: {entity|component|signal|label} is "(auto|yes|no)";
```
Verilog Syntax Example
Place immediately before the instance, module or signal declaration:

(* use_clock_enable = "{auto|yes|no}" *)

XCF Syntax Example One
MODEL "entity_name" use_clock_enable={auto|yes|no|true|false};

XCF Syntax Example Two
BEGIN MODEL "entity_name"
NET "signal_name" use_clock_enable={auto|yes|no|true|false};
END;

XCF Syntax Example Three
BEGIN MODEL "entity_name"
INST "instance_name" use_clock_enable={auto|yes|no|true|false};
END;

XST Command Line Syntax Example
Define globally with the run command.
-use_clock_enable {auto|yes|no}
The default is auto.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.
Process > Properties > Xilinx-Specific Options > Use Clock Enable
Use Synchronous Set (USE_SYNC_SET)

Use Synchronous Set (USE_SYNC_SET) enables or disables the synchronous set function in flip-flops. The disabling of the synchronous set function is generally used for ASIC prototyping. When XST detects Use Synchronous Set with a value of no or false, XST avoids using synchronous reset resources in the final implementation. For some designs, putting synchronous reset function on data input of the flip-flop allows better logic optimization and therefore better QOR.

In auto mode, XST tries to estimate a trade-off between using dedicated Synchronous Set input of a flip-flop input and putting Synchronous Set logic on the D input of a flip-flop. Where a flip-flop is instantiated by you, XST removes the synchronous reset only if Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) is set to yes.

The values for this constraint are:
- auto (default)
- yes
- no
- true (XCF only)
- false (XCF only)

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to:
- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

Propagation Rules

Applies to an entity, component, module, signal, or instance to which it is attached

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

```
attribute use_sync_set: string;
```

Specify as follows:

```
attribute use_sync_set of
{entity_name|component_name|signal_name|instance_name}: {entity|component|signal|label} is "(auto|yes|no)";
```
Verilog Syntax Example
Place immediately before the instance, module or signal declaration:

(* use_sync_set = "\{auto\|yes\|no\}" *)

XCF Syntax Example One

MODEL "entity_name" use_sync_set={auto\|yes\|no\|true\|false};

XCF Syntax Example Two

BEGIN MODEL "entity_name"
NET "signal_name" use_sync_set={auto\|yes\|no\|true\|false};
END;

XCF Syntax Example Three

BEGIN MODEL "entity_name"
INST "instance_name" use_sync_set={auto\|yes\|no\|true\|false};
END;

XST Command Line Syntax Example
Define globally with the run command.

-use_sync_set {auto\|yes\|no}
The default is auto.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Use Synchronous Set
Use Synchronous Reset (USE_SYNC_RESET)

Use Synchronous Reset (USE_SYNC_RESET) enables or disables the usage of synchronous reset function of flip-flops. The disabling of the Synchronous Reset function can be used for ASIC prototyping flow.

When Use Synchronous Reset is set to no or false, XST avoids using synchronous reset resources in the final implementation. For some designs, putting synchronous reset function on the data input of the flip-flop optimizes logic and gives higher quality of results.

In auto mode, XST tries to estimate a trade-off between using a dedicated Synchronous Reset input on a flip-flop input and putting Synchronous Reset logic on the D input of a flip-flop. Where a flip-flop is instantiated by you, XST removes the synchronous reset only if Optimize Instantiated Primitives (OPTIMIZE_PRIMITIVES) is set to yes.

The values for this constraint are:

- auto (default)
- yes
- no
- true (XCF only)
- false (XCF only)

Architecture Support
Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements
Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a flip-flop
- An instance representing an instantiated flip-flop

Propagation Rules
Applies to an entity, component, module, signal, or instance to which it is attached

Syntax Examples
The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example
Declare as follows:

```vhdl
attribute use_sync_reset: string;
```

Specify as follows:

```vhdl
attribute use_sync_reset of {entity_name|component_name|signal_name|instance_name}: is "{entity|component|signal|label}; is {auto|yes|no}";
```
Verilog Syntax Example
Place immediately before the instance, module, or signal declaration.

(* use_sync_reset = "(auto|yes|no)" *)

XCF Syntax Example One
MODEL "entity_name" use_sync_reset={auto|yes|no|true|false};

XCF Syntax Example Two
BEGIN MODEL "entity_name"
NET "signal_name" use_sync_reset={auto|yes|no|true|false};
END;

XCF Syntax Example Three
BEGIN MODEL "entity_name"
INST "instance_name" use_sync_reset={auto|yes|no|true|false};
END;

XST Command Line Syntax Example
Define globally with the run command.

-run use_sync_reset {auto|yes|no}
The default is auto.

ISE Design Suite Syntax Example
Define globally in ISE Design Suite.

Process > Properties > Xilinx-Specific Options > Use Synchronous Reset
Use DSP Block (USE_DSP48)

Use DSP Block (USE_DSP48) enables or disables the use of DSP Block resources.

Constraint Values

The values for this constraint are:

- **auto** (default)
  
  XST selectively implements arithmetic logic to DSP blocks, and seeks to maximize circuit performance. Macros such as multiply, multiply-addsub, and multiply-accumulate are automatically considered for DSP block implementation. XST looks for opportunities to leverage the cascading capabilities of DSP blocks. Other macros such as adders, counters, and standalone accumulators are implemented on slice logic.

- **automax**
  
  XST attempts to maximize DSP block utilization within the limits of available resources on the selected device. In addition to the macros considered in the auto mode, automax considers additional functions, such as adders, counters, and standalone accumulators, as candidates for DSP block implementation. Xilinx® recommends that you use automax when a tightly packed device is your primary concern, and you are attempting to free up LUT resources. **Attention** Using automax may degrade circuit performance compared to the default auto mode. Do not use automax when performance is your primary implementation goal.

- **yes** [or true (XCF only)]
  
  Allows you to manually force implementation of arithmetic logic to DSP blocks. Use yes primarily to force individual functions to DSP resources. Xilinx does not recommend applying yes globally, since XST does not check actual DSP resources availability in this mode, and may oversubscribe DSP blocks.

  **Attention** With a value of yes, the decision to implement a function in a DSP block ignores both the actual availability of DSP resources on the selected device, and any maximum allocation defined with the DSP Utilization Ratio (DSP_UTILIZATION_RATIO) constraint. As a result, the design may use more DSP resources than are available or budgeted.

- **no** [or false (XCF only)]
  
  Allows you to manually prevent implementation of designated logic on DSP resources.

DSP Block Resources

In the auto and automax modes, you can further control the number of DSP block resources exploited by synthesis with DSP Utilization Ratio (DSP_UTILIZATION_RATIO). By default, XST assumes that all available DSP blocks on the selected device can be used.

Macros such as multiply-addsub and multiply-accumulate are treated as a composition of simpler macros such as multipliers, accumulators, and registers. To maximize performance, XST performs these aggregations aggressively. It attempts in particular to use all pipelining stages in the DSP block.

Use the Keep (KEEP) constraint to control how XST aggregates those basic macros into a DSP block. For example, when two register stages are available before a multiplication operand, insert a Keep (KEEP) constraint between them if you want to prevent one of them from being implemented in the DSP block.
Chapter 12: XST FPGA Constraints (Non-Timing)

For more information on supported macros and their implementation control, see:
Chapter 7, XST HDL Coding Techniques

Architecture Support
Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements
Applies to:

- An entire design through the XST command line
- A particular block (entity, architecture, component)
- A signal representing a macro described at the RTL level

Propagation Rules
Applies to the entity, component, module, or signal to which it is attached

Syntax Examples
The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example
Declare as follows:

```vhdl
attribute use_dsp48: string;
```

Specify as follows:

```vhdl
attribute use_dsp48 of "entity_name" component_name | signal_name: {entity|component|signal} is "{auto|automax|yes|no}";
```

Verilog Syntax Example
Place immediately before the instance, module or signal declaration:

```verilog
(* use_dsp48 = "{auto|automax|yes|no}" *)
```

XCF Syntax Example One

```cflow
MODEL "entity_name" use_dsp48=(auto|automax|yes|no|true|false);
```

XCF Syntax Example Two

```cflow
BEGIN MODEL "entity_name"
NET "signal_name" use_dsp48=(auto|automax|yes|no|true|false);
END;
```

XST Command Line Syntax Example
Define globally with the run command.

```bash
-use_dsp48 {auto|automax|yes|no}
```
The default is auto.
ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > HDL Options > Use DSP Block
Apply XST timing constraints with:

- Global Optimization Goal (~glob_opt)
- User Constraints File (UCF)
- XST Constraint File (XCF)

The following additional options affect timing constraint processing, regardless of how the timing constraints are applied:

- Clock Signal (CLOCK_SIGNAL)
- Cross Clock Analysis (~cross_clock_analysis)
- Write Timing Constraints (~write_timing_constraints)

Global Optimization Goal

Use Global Optimization Goal (~glob_opt) to apply the five global timing constraints:

- ALLCLOCKNETS
- OFFSET_IN_BEFORE
- OFFSET_OUT_AFTER
- INPAD_TO_OUTPAD
- MAX_DELAY

These constraints:

- Are applied globally to the entire design.
- Cannot have a value specified for them, since XST optimizes them for the best performance.
- Are overridden by constraints specified in the User Constraints File (UCF).

User Constraints File (UCF)

Use the User Constraints File (UCF) to apply timing constraints using native UCF syntax. XST supports constraints such as:

- Timing Name (TNM)
- Timegroup (TIMEGRP)
- Period (PERIOD)
- Timing Ignore (TIG)
- From-To (FROM-TO)

XST supports wildcards and hierarchical names with these constraints.
XST Constraint File (XCF)

If you apply timing constraints using an XST Constraint File (XCF), Xilinx® recommends that you use a forward slash (/) as a hierarchy separator instead of an underscore (_).

For more information, see:
Hierarchy Separator

If XST does not support all or part of a specified timing constraint, XST:

- Issues a warning.
- Ignores the unsupported timing constraint (or unsupported part of it) in the Timing Optimization step.

If Write Timing Constraints (--write_timing_constraints) is set to yes, XST propagates the entire constraint to the final netlist, even if it was ignored at the Timing Optimization step.

The following timing constraints are supported in an XCF:

- Period (PERIOD)
- Offset (OFFSET)
- From-To (FROM-TO)
- Timing Name (TNM)
- Timing Name on a Net (TNM_NET)
- Timegroup (TIMEGRP)
- Timing Ignore (TIG)
Clock Signal (CLOCK_SIGNAL)

If a clock signal goes through combinatorial logic before being connected to the clock input of a flip-flop, XST cannot identify which input pin or internal signal is the real clock signal. Use Clock Signal (CLOCK_SIGNAL) to define the clock signal.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to signals

Propagation Rules

Applies to clock signals

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax Example

Declare as follows:

attribute clock_signal : string;

Specify as follows:

attribute clock_signal of signal_name : signal is "{yes|no}";

Verilog Syntax Example

Place immediately before the signal declaration:

(* clock_signal = "{yes|no}" *)

XCF Syntax Example

BEGIN MODEL "entity_name"

NET "primary_clock_signal" clock_signal={yes|no|true|false};

END;
Cross Clock Analysis (–cross_clock_analysis)

Cross Clock Analysis (–cross_clock_analysis) allows XST to perform timing optimizations across clock domains. By default, those optimizations are disabled, since they may not always be desirable. When optimizations are disabled, XST optimizes timing only within each separate clock domain.

If you use Register Balancing (REGISTER_BALANCING) to enable flip-flop retiming, Cross Clock Analysis also defines the scope of the retiming.

- If Cross Clock Analysis is enabled, logic may be moved from one clock domain to the other when beneficial.
- If Cross Clock Analysis is disabled, register balancing takes place only within each clock domain.

Obtaining Cross Clock Domain Timing Information describes where to find inter-clock domain timing information. It is available by default in the XST Synthesis Report. Inter-clock domain optimizations do not need to be activated to access it.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to an entire design through the XST command line

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

–cross_clock_analysis (yes|no)

The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Cross Clock Analysis
Chapter 13: XST Timing Constraints

From-To (FROM-TO)

From-To (FROM-TO) defines a timing constraint between two groups.
A group can be user-defined or predefined:
- FF
- PAD
- RAM
For more information, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XCF Syntax Example

```
TIMESPEC TSname = FROM group1 TO group2 value;
```
Global Optimization Goal (-glob_opt)

The Global Optimization Goal (-glob_opt) constraint:
- Defines how XST optimizes the entire design for best performance.
- Allows XST to optimize the following design regions:
  - Register to register
  - Inpad to register
  - Register to outpad
  - Inpad to outpad
- Lets you select among one of the following global timing constraints:
  - **ALLCLOCKNETS**
    Optimizes the period of the entire design
  - **OFFSET_BEFORE**
    Optimizes the maximum delay from input pad to clock, either for a specific clock or for an entire design.
  - **OFFSET_OUT_AFTER**
    Optimizes the maximum delay from clock to output pad, either for a specific clock or for an entire design.
  - **INP_AD_OUTP**
    Optimizes the maximum delay from input pad to output pad throughout an entire design.
  - **MAX_DELAY**
    Incorporates all previously mentioned constraints

These constraints are applied globally to the entire design. They apply only if no timing constraints are specified in the constraint file. You cannot specify a value for these constraints, since XST optimizes them for the best performance. These constraints are overridden by constraints specified in the User Constraints File (UCF).

You can apply Global Optimization Goal in ISE® Design Suite in Process > Properties > Synthesis Options > Global Optimization Goal
Global Optimization Goal Domain Definitions

The possible domains are shown in the following schematic.

- **ALLCLOCKNETS** (register to register)
  Identifies all paths from register to register on the same clock for all clocks in a design. To take inter-clock domain delays into account, set Cross Clock Analysis (`-cross_clock_analysis`) to yes.

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

- **OFFSET_OUT_AFTER** (register to outpad)
  Similar to **OFFSET_IN_BEFORE**, but sets the constraint from the sequential elements to all primary output ports

- **INP_TO_OUTPAD** (inpad to outpad)
  Sets a maximum combinatorial path constraint.

- **MAX_DELAY**
  - **ALLCLOCKNETS**
  - **OFFSET_IN_BEFORE**
  - **OFFSET_OUT_AFTER**
  - **INP_TO_OUTPAD**

Global Optimization Goal Domain Diagram

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XST Command Line Syntax Example**

Define globally with the `run` command.

```
glob_opt {allclocknets|offset_in_before|offset_out_after|inpad_to_outpad|max_delay} -
```

**ISE Design Suite Syntax Example**

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Global Optimization Goal
Offset (OFFSET)

The Offset (OFFSET) constraint:

- Is a basic timing constraint.
- Specifies the timing relationship between an external clock and its associated data-in or data-out pin.
- Is used only for pad-related signals.
- Cannot be used to extend the arrival time specification method to the internal signals in a design.
- Calculates whether a setup time is being violated at a flip-flop whose data and clock inputs are derived from external nets.
- Specifies the delay of an external output net derived from the Q output of an internal flip-flop being clocked from an external device pin.

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XCF Syntax Example

```
OFFSET = (IN|OUT) offset_time [units] {BEFORE|AFTER} clk_name [TIMEGRP group_name];
```
Period (PERIOD)

Period (PERIOD) is a basic timing constraint and synthesis constraint. A clock period specification checks timing between all synchronous elements within the clock domain as defined in the destination element group. The group may contain paths that cross between clock domains if the clocks are defined as a function of one or the other.

For an MMCM block, apply the PERIOD constraints to the appropriate clock input signals. Manually creating related PERIOD constraints for the MMCM clock output signals is not necessary. XST automatically derives them so synthesis decisions can be made on the basis of accurate timing data.

XST writes out only those PERIOD constraints specified in the NGC netlist. XST does not explicitly write out derived constraints. Derived constraints are written out later during implementation.

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XCF Syntax Example

```
NET netname PERIOD = value [HIGH|LOW] value;
```
Timing Name (TNM)

The Timing Name (TNM) constraint:
- Is a basic grouping constraint.
- Identifies the elements that make up a group for use in a timing specification.
- Tags specific FF, RAM, LATCH, PAD, BRAM_PORTA, BRAM_PORTB, CPU, HSIO, and MULT elements as members of a group to simplify the application of timing specifications.

You can use the RISING and FALLING keywords with Timing Name.

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XCF Syntax Example

```
{INST|NET|PIN} inst_net_or_pin_name TNM = [predefined_group:] identifier;
```
Timing Name on a Net (TNM_NET)

Timing Name on a Net (TNM_NET) is essentially equivalent to Timing Name (TNM) on a net except for input pad nets. Special rules apply when using Timing Name (TNM) and TNM_NET with Period (PERIOD) for DLLs, DCMs, and PLLs.

For more information, see:

“PERIOD Specifications on CLKDLLs, DCMs, and PLLs” in the Constraints Guide.

A Timing Name on a Net is a property that you normally use in conjunction with a Hardware Description Language (HDL) design to tag a specific net. All downstream synchronous elements and pads tagged with the TNM_NET identifier are considered a group.

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XCF Syntax Example

```plaintext
NET netname TNM_NET = [predefined_group:] identifier;
```
Timegroup (TIMEGRP)

Timegroup (TIMEGRP) is a basic grouping constraint. In addition to naming groups using the TNM identifier, you can also:

- Define groups in terms of other groups.
- Create a group that is a combination of existing groups by defining a Timegroup constraint.

You can:

- Place Timegroup constraints in:
  - An XST Constraint File (XCF), or
  - A Netlist Constraints File (NCF)
- Use Timegroup attributes to create groups by:
  - Combining multiple groups into one, or
  - Defining flip-flop subgroups by clock sense

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XCF Syntax Example**

```
TIMEGRP newgroup = existing_grp1 existing_grp2 [existing_grp3 ...
```
Timing Ignore (TIG)

The Timing Ignore (TIG) constraint:

- Causes all paths going through a specific net to be ignored for timing analysis and optimization.
- Can be applied to the name of the affected signal.

For more information about this constraint, see the Constraints Guide.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

See the Constraints Guide.

Propagation Rules

See the Constraints Guide.

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

**XCF Syntax Example**

```
NET net_name TIG;
```
Write Timing Constraints (–write_timing_constraints)

Timing constraints are written to the NGC file only when:

- Write Timing Constraints is checked yes in ISE® Design Suite in Process > Properties > Synthesis Options > Write Timing Constraints, or
- -write_timing_constraints is specified in the command line.

By default, timing constraints are not written to the NGC file.

Architecture Support

Applies to Virtex®-6, Spartan®-6, and 7 series devices

Applicable Elements

Applies to an entire design through the XST command line

Propagation Rules

Not applicable

Syntax Examples

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

XST Command Line Syntax Example

Define globally with the run command.

–write_timing_constraints {yes|no}

The default is no.

ISE Design Suite Syntax Example

Define globally in ISE Design Suite.

Process > Properties > Synthesis Options > Write Timing Constraints
XST supports many third-party constraints. The following table shows the XST equivalents for these constraints. For more information on specific constraints, see your vendor documentation.

Constraints marked yes are fully supported. If a constraint is only partially supported, the support conditions are shown in the Automatic Recognition column.

Third-party constraints can be applied in a similar way as Xilinx® constraints, using one of the following mechanisms:

- VHDL attribute
- Verilog attribute
- XCF constraint

VHDL uses standard attribute syntax. No changes are needed to the Hardware Description Language (HDL) code.

For Verilog with third-party metacomment syntax, you must change the metacomment syntax to conform to XST conventions. The constraint name and its value can be used as shown in the third-party tool.

For Verilog 2001 attributes, no changes are needed to the HDL code. The constraint is automatically translated as in the case of VHDL attribute syntax.

### XST Equivalents to Third-Party Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Vendor</th>
<th>XST Equivalent</th>
<th>Automatic Recognition</th>
<th>Available For</th>
</tr>
</thead>
<tbody>
<tr>
<td>black_box</td>
<td>Synopsys</td>
<td>BoxType</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>black_box_pad_pin</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>black_box_tri_pins</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>cell_list</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>clock_list</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>enum</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>full_case</td>
<td>Synopsys</td>
<td>Full Case</td>
<td>N/A</td>
<td>Verilog</td>
</tr>
<tr>
<td>ispad</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>map_to_module</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Chapter 14: XST-Supported Third-Party Constraints

<table>
<thead>
<tr>
<th>Name</th>
<th>Vendor</th>
<th>XST Equivalent</th>
<th>Automatic Recognition</th>
<th>Available For</th>
</tr>
</thead>
<tbody>
<tr>
<td>net_name</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>parallel_case</td>
<td>Synopsys</td>
<td>Parallel Case</td>
<td>N/A</td>
<td>Verilog</td>
</tr>
<tr>
<td>return_port_name</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>resource_sharing directives</td>
<td>Synopsys</td>
<td>Resource Sharing</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>set_dont_touch_network</td>
<td>Synopsys</td>
<td>not required</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>set_dont_touch</td>
<td>Synopsys</td>
<td>not required</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>set_dont_use_cell_name</td>
<td>Synopsys</td>
<td>not required</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>set_prefer</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>state_vector</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_allow_retiming</td>
<td>Synopsys</td>
<td>Register Balancing</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>syn_black_box</td>
<td>Synopsys</td>
<td>BoxType</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td>syn_direct_enable</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_edif_bit_format</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_edif_scalar_format</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_encoding</td>
<td>Synopsys</td>
<td>FSM Encoding Algorithm</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The value safe is not supported for automatic recognition. Use Safe Implementation in XST to activate this mode.</td>
<td></td>
</tr>
<tr>
<td>syn_enum_encoding</td>
<td>Synopsys</td>
<td>Enumerated Encoding</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>syn_hier</td>
<td>Synopsys</td>
<td>Keep Hierarchy</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>syn_hier = hard is recognized as keep_hierarchy = soft</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>syn_hier = remove is recognized as keep_hierarchy = no</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>XST supports only the values hard and remove for syn_hier in automatic recognition.</td>
<td></td>
</tr>
<tr>
<td>syn_isclock</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Name</td>
<td>Vendor</td>
<td>XST Equivalent</td>
<td>Automatic Recognition</td>
<td>Available For</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------</td>
<td>---------------------------------</td>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>syn_keep</td>
<td>Synopsys</td>
<td>Keep</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_maxfan</td>
<td>Synopsys</td>
<td>Max Fanout</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_netlist_hierarch</td>
<td>Synopsys</td>
<td>Netlist Hierarchy</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_noarrayports</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_noclockbuf</td>
<td>Synopsys</td>
<td>Buffer Type</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_noprune</td>
<td>Synopsys</td>
<td>Optimize Instantiated Primitives</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_pipeline</td>
<td>Synopsys</td>
<td>Register Balancing</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_preserve</td>
<td>Synopsys</td>
<td>Equivalent Register Removal</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_ramstyle</td>
<td>Synopsys</td>
<td>RAM Extraction and RAM Style</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_reference_clock</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_replicate</td>
<td>Synopsys</td>
<td>Register Duplication</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_ramstyle</td>
<td>Synopsys</td>
<td>ROM Extraction and ROM Style</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_sharing</td>
<td>Synopsys</td>
<td>Resource Sharing</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_state_machine</td>
<td>Synopsys</td>
<td>Automatic FSM Extraction</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>syn_tco</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_tpd</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_tristate</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_tristatetomux</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_tsu</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>syn_useenables</td>
<td>Synopsys</td>
<td>Use Clock Enable</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

XST implements RAMs in **no_rw_check** mode whether or not **no_rw_check** is specified. The area value is ignored.
<table>
<thead>
<tr>
<th>Name</th>
<th>Vendor</th>
<th>XST Equivalent</th>
<th>Automatic Recognition</th>
<th>Available For</th>
</tr>
</thead>
<tbody>
<tr>
<td>syn_useioff</td>
<td>Synopsys</td>
<td>Pack I/O Registers Into IOBs (IOB)</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>synthesis_translate_off</td>
<td>Synopsys</td>
<td>Translate Off and Translate On</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td>synthesis_translate_on</td>
<td>Synopsys</td>
<td>Translates Off and Translates On</td>
<td></td>
<td>Verilog</td>
</tr>
<tr>
<td>xc_alias</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>xc_clockbuftype</td>
<td>Synopsys</td>
<td>Buffer Type</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_fast</td>
<td>Synopsys</td>
<td>FAST</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_fast_auto</td>
<td>Synopsys</td>
<td>FAST</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_global_buffers</td>
<td>Synopsys</td>
<td>BUFG (XST)</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_ioff</td>
<td>Synopsys</td>
<td>Pack I/O Registers Into IOBs</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_isgsr</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>xc_loc</td>
<td>Synopsys</td>
<td>LOC</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_map</td>
<td>Synopsys</td>
<td>Map Entity on a Single LUT</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_ncf_auto_relax</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>xc_nodelay</td>
<td>Synopsys</td>
<td>NODELAY</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_padtype</td>
<td>Synopsys</td>
<td>I/O Standard</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_props</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>xc_pullup</td>
<td>Synopsys</td>
<td>PULLUP</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_rloc</td>
<td>Synopsys</td>
<td>RLOC</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_fast</td>
<td>Synopsys</td>
<td>FAST</td>
<td>N/A</td>
<td>VHDL</td>
</tr>
<tr>
<td>xc_slow</td>
<td>Synopsys</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>xc_uset</td>
<td>Synopsys</td>
<td>U_SET</td>
<td>Yes</td>
<td>VHDL</td>
</tr>
</tbody>
</table>
XST Synthesis Report

The XST Synthesis Report:
- Is an ASCII text file
- Is a hybrid between a report and a log
- Contains information about the XST synthesis run

During synthesis, the XST Synthesis Report allows you to:
- Control the progress of the synthesis
- Review what has occurred so far

After synthesis, the XST Synthesis Report allows you to:
- Determine whether the Hardware Description Language (HDL) description has been processed according to expectations.
- Determine whether device resources utilization and optimization levels are likely to meet design goals once the synthesized netlist has been run through the implementation chain.

XST Synthesis Report Contents

The XST Synthesis Report contains the following sections:
- Table of Contents
- Synthesis Options Summary
- HDL Parsing and Elaboration
- HDL Synthesis
- Advanced HDL Synthesis
- Low Level Synthesis
- Partition Report
- Design Summary

Table of Contents

Use the Table of Contents to navigate through the XST Synthesis Report.
For more information, see:
XST Synthesis Report Navigation

Synthesis Options Summary

The Synthesis Options Summary section summarizes the parameters and options used for the current synthesis run.
**HDL Parsing and Elaboration**

During Hardware Description Language (HDL) parsing and elaboration, XST:

- Parses the VHDL and Verilog files that make up the synthesis project
- Interprets the contents of the VHDL and Verilog files
- Recognizes the design hierarchy
- Flags HDL coding mistakes
- Points out potential problems such as:
  - Simulation mismatches between post-synthesis and HDL
  - Potential multi-source situations

If problems occur at later stages of synthesis, the HDL parsing and elaboration sections may reveal the root cause of these problems.

**HDL Synthesis**

During HDL Synthesis, XST:

- Attempts to recognize basic macros such as registers, adders, and multipliers for which a technology-specific implementation might later be possible.
- Looks for Finite State Machine (FSM) descriptions on a block by block basis.
- Issues the HDL Synthesis Report, providing statistics on inferred macros.

For more information about the processing of each macro and the corresponding messages issued during synthesis, see Chapter 7, XST HDL Coding Techniques.

**Advanced HDL Synthesis**

During Advanced HDL Synthesis, XST:

- Attempts to combine basic macros inferred during the HDL Synthesis phase into larger macro blocks such as counters, pipelined multipliers, and multiply-accumulate functions.
- Reports on the selected encoding scheme for each inferred Finite State Machine (FSM).

The Advanced HDL Synthesis Report summarizes the recognized macros in the overall design, sorted by macro type.

For more information, see:

Chapter 7, XST HDL Coding Techniques

**Low Level Synthesis**

The Low Level Synthesis section displays information about the low-level optimizations performed by XST, including the removal of equivalent flip-flops, register replication, or the optimization of constant flip-flops.

**Partition Report**

If the design is partitioned, the Partition Report displays information about the design partitions.
Design Summary

The Design Summary section helps you determine whether synthesis has been successful, especially whether device utilization and circuit performance has met design goals.

The Design Summary section contains the following subsections:

- Primitive and Black Box Usage
- Device Utilization Summary
- Partition Resource Summary
- Timing Report
- Clock Information
- Asynchronous Control Signals Information
- Timing Summary
- Timing Details
- Encrypted Modules

Primitive and Black Box Usage

The Primitive and Black Box Usage subsection displays usage statistics for all device primitives and identified black boxes.

The primitives are classified in the following groups:

- **BELS**
  - All basic logical primitives such as **LUT**, **MUXCY**, **XORCY**, **MUXF5**, and **MUXF6**
- Flip-flops and latches
- Block and distributed RAM
- Shift register primitives
- Tristate buffers
- Clock buffers
- I/O buffers
- Other logical, more complex, primitives such as **AND2** and **OR2**
- Other primitives

Device Utilization Summary

The Device Utilization Summary subsection displays XST device utilization estimates for such functions as:

- Slice logic utilization
- Slice logic distribution
- Number of flip-flops
- I/O utilization
- Number of block RAMs
- Number of DSP blocks

A similar report is generated when you later run **MAP**.

Partition Resource Summary

If partitions have been defined, the Partition Resource Summary subsection displays information similar to the Device Utilization Summary on a partition-by-partition basis.
Timing Report

The Timing Report subsection displays XST timing estimates to help you:

- Determine whether the design meets performance and timing requirements.
- Locate bottlenecks if performance and timing requirements are not met.

Clock Information

The Clock Information subsection displays information about the number of clocks in the design, how each clock is buffered, and their respective fanout.

Clock Information Report Example

Clock Information:
-----------------------------------+------------------------+-------+
Clock Signal | Clock buffer (FF name) | Load |
-----------------------------------+------------------------+-------+
CLK          | BUFGP                  | 11    |

Asynchronous Control Signals Information

The Asynchronous Control Signals Information subsection displays information about the number of asynchronous set/reset signals in the design, how each signal is buffered, and their respective fanout.

Asynchronous Control Signals Report Information Example

Asynchronous Control Signals Information:
-------------------------------------+-------------------------------+-------+
Control Signal | Buffer (FF name) | Load |
rstint (MACHINE/current_state_Out01:O) | NONE (sixty/lsbcount/qoutsig_3) | 4 |
RESET          | IBUF               | 3    |
sixty/msbclr(sixty/msbclr:O)           | NONE (sixty/msbcount/qoutsig_3) | 4 |
-------------------------------------+-------------------------------+-------+

Timing Summary

The Timing Summary subsection shows timing information for all four possible clock domains of a netlist:

- Minimum period (register to register paths)
- Minimum input arrival time before clock (input to register paths)
- Maximum output required time after clock (register to outpad paths)
- Maximum combinatorial path delay (inpad to outpad paths)

This timing information is an estimate. For precise timing information, see the TRACE Report generated after placement and routing.

Timing Summary Report Example

Timing Summary:
---------------------
Speed Grade: -1

Minimum period: 2.644ns (Maximum Frequency: 378.165MHz)
Minimum input arrival time before clock: 2.148ns
Maximum output required time after clock: 4.803ns
Maximum combinatorial path delay: 4.473ns
Timing Details

The Timing Details subsection displays information about the most critical path in each clock region, including:

- Start point
- End point
- Maximum delay
- Levels of logic
- Detailed breakdown of the path into individual net and component delays, also providing valuable information on net fanouts.
- Distribution between routing and logic

When XST writes out a hierarchical netlist with Netlist Hierarchy (-netlist_hierarchy) and the reported path crosses hierarchical boundaries, the detailed path breakdown uses the `begin scope` and `end scope` keywords to indicate when the path enters and exits a hierarchical block.

Timing Details Report Example

Timing Details:
----------------
All values displayed in nanoseconds (ns)

=========================================================================  
Timing constraint: Default period analysis for Clock ‘CLK’
Clock period: 2.644ns (frequency: 378.165MHz)
Total number of paths / destination ports: 77 / 11
=========================================================================  
Delay: 2.644ns (Levels of Logic = 3)
Source: MACHINE/current_state_FFd3 (FF)
Destination: sixty/msbcount/qoutsig_3 (FF)
Source Clock: CLK rising
Destination Clock: CLK rising

Data Path: MACHINE/current_state_FFd3 to sixty/msbcount/qoutsig_3

<table>
<thead>
<tr>
<th>Cell:in-&gt;out</th>
<th>fanout</th>
<th>Gate</th>
<th>Net</th>
<th>Delay</th>
<th>Logical Name (Net Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FDC:C-&gt;Q</td>
<td>8</td>
<td>0.272</td>
<td>ctrl/state_FFd3 (ctrl/state_FFd3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUT3:10--O</td>
<td>3</td>
<td>0.147</td>
<td>Ker81 (clkenable)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUT4_D:11-&gt;O</td>
<td>1</td>
<td>0.147</td>
<td>sixty/msbce (sixty/msbce)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LUT3:12-&gt;O</td>
<td>1</td>
<td>0.147</td>
<td>sixty/msbcount/qoutsig_3_rstpot (N43)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDC:D</td>
<td>0.297</td>
<td></td>
<td>sixty/msbcount/qoutsig_3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total 2.644ns (1.010ns logic, 1.634ns route)
(38.2% logic, 61.8% route)
Timing Details Report For a Hierarchy Crossing Path Example

Timing constraint: Default path analysis
Total number of paths / destination ports: 36512 / 16

Delay: 4.326ns (Levels of Logic = 14)
Source: a<0> (PAD)
Destination: out<3> (PAD)

Data Path: a<0> to out<3>

<table>
<thead>
<tr>
<th>Cell: in-&gt;out</th>
<th>fanout</th>
<th>Gate Delay</th>
<th>Net Delay</th>
<th>Logical Name (Net Name)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBUF: I-&gt;O</td>
<td>5</td>
<td>0.003</td>
<td>0.376</td>
<td>a_0_IBUF (a_0_IBUF)</td>
</tr>
</tbody>
</table>
| begin scope: 'm'
| LUT2: I0->O  | 1      | 0.053      | 0.000     | Madd_out_Madd_lut<0> (Madd_out_Madd_lut<0>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<1> (Madd_out_Madd_cy<1>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<2> (Madd_out_Madd_cy<2>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<3> (Madd_out_Madd_cy<3>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<4> (Madd_out_Madd_cy<4>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<5> (Madd_out_Madd_cy<5>) |
| MUXCY: CI->O | 1      | 0.015      | 0.000     | Madd_out_Madd_cy<6> (Madd_out_Madd_cy<6>) |
| XORCY: CI->O | 1      | 0.180      | 0.279     | Madd_out_Madd_xor<7> (out<7>) |
| end scope: 'a1'
| DSP48E1: A7->P2 | 1 | 2.843 | 0.279 | Maddsub_out (out_2_OBUF) |
| end scope: 'm'
| OBUF: I->O   | 0.003  | out_2_OBUF (out<2>) |

Total 4.326ns (3.391ns logic, 0.935ns route)
(78.4% logic, 21.6% route)

Obtaining Cross Clock Domain Timing Information

Clock Domain Crossing (CDC) paths are reported in the Cross Domains Crossing Report section of the XST Synthesis Report. The Cross Domains Crossing Report section:

- Is provided by default
- Follows the Timing Details section
- Is available whether or not XST has performed cross clock domain optimization
- Is available whether or not you have specified timing constraints in an XST Constraint File (XCF)

You do not need to enable Cross Clock Analysis (~cross_clock_analysis) to obtain cross clock domain timing information. Use Cross Clock Analysis (~cross_clock_analysis) only if you want XST to take advantage of cross clock domain timing information in order to seek timing optimizations across clock domains.
Cross Domains Crossing Report Example

Clock Domains Crossing Report:
---------------------------------

Clock to Setup on destination clock clk2
----------------------------------------

<table>
<thead>
<tr>
<th>Source Clock</th>
<th>Src:Rise</th>
<th>Src:Fall</th>
<th>Dest:Rise</th>
<th>Dest:Fall</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk1</td>
<td></td>
<td>0.804</td>
<td></td>
<td></td>
</tr>
<tr>
<td>clk2</td>
<td></td>
<td>0.661</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clock to Setup on destination clock clk3
----------------------------------------

<table>
<thead>
<tr>
<th>Source Clock</th>
<th>Src:Rise</th>
<th>Src:Fall</th>
<th>Dest:Rise</th>
<th>Dest:Fall</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk2</td>
<td></td>
<td></td>
<td>0.809</td>
<td></td>
</tr>
<tr>
<td>clk3</td>
<td></td>
<td></td>
<td>0.651</td>
<td></td>
</tr>
</tbody>
</table>

Encrypted Modules

XST hides all information about encrypted modules.

XST Synthesis Report Navigation


Command Line Mode

In command line mode, XST generates an SRP (.srp) file. The SRP file:
- Contains the full XST Synthesis Report
- Is an ASCII text file
- Can be opened in a text editor

Entries in the SRP file Table of Contents are not hyperlinked. Use the text editor Find function to navigate.

ISE Design Suite

In ISE® Design Suite, XST generates an SYR (.syr) file. The SYR file:
- Contains the full XST Synthesis Report.
- Is located in the directory in which the ISE Design Suite project resides.
- Allows you to navigate to the different sections of the XST Synthesis Report using a navigation pane.
XST Synthesis Report Information

Use the following to reduce the information displayed in the XST Synthesis Report:

- Message Filtering
- Quiet Mode
- Silent Mode

Message Filtering

When running XST in ISE® Design Suite, use the Message Filtering wizard to select specific messages to filter out of the XST Synthesis Report. You can filter out individual messages, or a category of messages.

For more information, see:
“Using the Message Filters” in the ISE Design Suite Help.

Quiet Mode

Quiet Mode:
- Limits the number of messages printed to the computer screen (stdout).
- Does not affect the contents of the XST Synthesis Report itself. The report contains the full, unfiltered, synthesis information.

To invoke Quiet Mode, set -intstyle to either of the following:
- ise
  Formats messages for ISE® Design Suite
- xflow
  Formats messages for XFLOW

XST normally prints the entire report to stdout. In Quiet Mode, XST does not print the following sections of the XST Synthesis Report to stdout:
- Copyright Message
- Table of Contents
- Synthesis Options Summary
- The following portions of the Design Summary:
  - Final Results section
  - A note in the Timing Report stating that the timing numbers are only a synthesis estimate
  - Timing Details
  - CPU (XST runtime)
  - Memory usage

In Quiet Mode, XST prints the following sections of the XST Synthesis Report to stdout:
- Device Utilization Summary
- Clock Information
- Timing Summary
Silent Mode

Silent Mode prevents messages from being sent to the computer screen (stdout). The entire XST Synthesis Report is written to the log file.

To invoke Silent Mode, set -intstyle to silent.
Chapter 16

XST Naming Conventions

Synthesis tools must use a naming strategy for objects written to the synthesized netlist that is logical, consistent, predictable, and repeatable. Whether you wish to control implementation of a design with constraints, or to reduce timing closure cycles, XST naming conventions help you achieve those goals.

XST Naming Conventions Coding Examples

Coding examples are accurate as of the date of publication. Download updates and other examples from ftp://ftp.xilinx.com/pub/documentation/misc/xstug_examples.zip. Each directory contains a summary.txt file listing all examples together with a brief overview.

Reg in Labelled Always Block Verilog Coding Example

```verilog
//
// A reg in a labelled always block
//
// File: Naming_Conventions/reg_in_labelled_always.v
//
module top (input clk, input di, output do);
reg data;
always @(posedge clk)
begin : mylabel
    reg tmp;
    tmp <= di;     // Post-synthesis name : mylabel.tmp
    data <= ~tmp;  // Post-synthesis name : data
end
assign do = ~data;
endmodule
```
Chapter 16: XST Naming Conventions

Primitve Instantiation in If-Generate Without Label Verilog Coding Example

//
// A primitive instantiation in a if-generate without label
//
// File: Naming_Conventions/if_generate_nolabel.v
//
module top (input clk, input di, output do);

parameter TEST_COND = 1;

generate

if (TEST_COND) begin
    FD myinst (.C(clk), .D(di), .Q(do)); // Post-synthesis name : myinst
end

endgenerate

endmodule

Primitve Instantiation in If-Generate With Label Verilog Coding Example

//
// A primitive instantiation in a labelled if-generate
//
// File: Naming_Conventions/if_generate_label.v
//
module top (input clk, input rst, input di, output do);

// parameter TEST_COND = 1;
// parameter TEST_COND = 0;

generate

if (TEST_COND)
    begin : myifname
        FDR myinst (.C(clk), .D(di), .Q(do), .R(rst));
        // Post-synthesis name : myifname.myinst
    end
else
    begin : myelsename
        FDS myinst (.C(clk), .D(di), .Q(do), .S(rst));
        // Post-synthesis name : myelsename.myinst
    end

endgenerate

endmodule
Variable in Labelled Process VHDL Coding Example

--
-- A variable in a labelled process
--
-- File: Naming_Conventions/var_in_labelled_process.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity top is
  port(
    clk : in std_logic;
    di : in std_logic;
    do : out std_logic
  );
end top;

architecture behavioral of top is
  signal data : std_logic;
begin
  mylabel: process (clk)
    variable tmp : std_logic;
  begin
    if rising_edge(clk) then
      tmp := di;  -- Post-synthesis name : mylabel.tmp
      end if;
      data <= not(tmp);
      end process;
      do <= not(data);
  end behavioral;

Flip-Flop Modeled With a Boolean VHDL Coding Example

--
-- Naming of boolean type objects
--
-- File: Naming_Conventions/boolean.vhd
--
library ieee;
use ieee.std_logic_1164.all;

entity top is
  port(
    clk : in std_logic;
    di : in boolean;
    do : out boolean
  );
end top;

architecture behavioral of top is
  signal data : boolean;
begin
  process (clk)
  begin
    if rising_edge(clk) then
      data <= di;  -- Post-synthesis name : data
      end if;
      end process;
      do <= not(data);
  end behavioral;
XST Net Naming Conventions

XST creates net names based on the following rules, listed in order of naming priority:

1. Maintain external pin names.
2. Keep hierarchy in signal names, using the hierarchy separator defined by Hierarchy Separator. The default hierarchy separator is a forward slash (/).
3. Maintain output signal names of registers, including state bits. Use the hierarchical name from the level where the register was inferred.
4. For output signals of clock buffers, a _clockbuffertype suffix (such as _BUFGP or _IBUFG) is appended to the clock signal name.
5. Maintain input nets to registers and tristates names.
6. Maintain names of signals connected to primitives and black boxes.
7. The output net of an IBUF is named <signal_name>_IBUF. Assuming for example that an IBUF output drives signal DIN, the output net of this IBUF is named DIN_IBUF.
8. The input net to an OBUF is named <signal_name>_OBUF. Assuming for example that an OBUF input is driven by signal DOUT, the input net of this OBUF is named DOUT_OBUF.
9. Base names for internal (combinatorial) nets on user HDL signal names where possible.
10. Nets resulting from the expansion of buses are formatted as <bus_name><left_delimiter><position>#<right_delimiter>. The default left and right delimiters are respectively < and >. Use Bus Delimiter (~bus_delimiter) to change this convention.
XST Instance Naming Conventions

XST creates instance names based on the following rules, listed in order of naming priority:

1. Keep hierarchy in instance names, using the hierarchy separator defined by Hierarchy Separator. The default hierarchy separator is a slash (/).

2. When instance names are generated from HDL generate statements, labels from generate statements are used in composition of instance names.
   
   For the following VHDL generate statement:
   
   ```vhdl
   i1_loop: for i in 1 to 10 generate
   inst_lut: LUT2 generic map (INIT => "00")
   ```
   
   XST generates the following instance names for LUT2:
   
   ```vhdl
   i1_loop[1].inst_lut
   i1_loop[2].inst_lut
   ...
   i1_loop[9].inst_lut
   i1_loop[10].inst_lut
   ```

3. Match the flip-flop instance name to the name of the signal it drives. This principle also applies to state bits.

4. Name clock buffer instances _clockbuffertype (such as _BUFGP or _IBUFG) after the output signal.

5. Names of black box instances are maintained.

6. Name of library primitive instances are maintained.

7. Name input and output buffers using the form _IBUF or _OBUF after the pad name.

8. Name Output instance names of IBUFs using the form instance_name_IBUF.

9. Name input instance names to OBUFs using the form instance_name_OBUF.

XST Case Preservation

Verilog is case sensitive. Unless instructed otherwise through the Case (-case) option, XST enforces the exact capitalization found in the Hardware Description Language (HDL) source code.

For more information on XST support for Verilog case sensitivity, see Case Sensitivity.

VHDL is case insensitive. Unless instructed otherwise through the Case (-case) option, object names based on names defined in the HDL source code are converted to all lowercase in the synthesized netlist.

XST Name Generation Control

The following constraints allow some control over the naming of objects in the synthesized netlist.

- Hierarchy Separator (~hierarchy_separator)
- Bus Delimiter (~bus_delimiter)
- Case (~case)
- Duplication Suffix (~duplication_suffix)

Apply these constraints in ISE® Design Suite in Synthesize - XST Process > Properties, or use the appropriate command line options.
For more information, see:

Chapter 9, XST Design Constraints
Appendix

Additional Resources

- Xilinx Documentation - [http://www.xilinx.com/support/documentation](http://www.xilinx.com/support/documentation)
- Xilinx Support - [http://www.xilinx.com/support](http://www.xilinx.com/support)