Constraints Guide

10.1

XILINX®

About This Guide

The Xilinx® *Constraints Guide* describes constraints and attributes that can be attached to designs for Xilinx FPGA and CPLD devices. This chapter contains the following sections:

- "Guide Contents"
- "Additional Resources"
- "Conventions"

Guide Contents

This Guide contains the following chapters:

- Chapter 1, "Introduction," discusses What's New in this Guide for ISE™ Release 10.1, and provides a Supported Architectures table showing the Xilinx devices supported for each constraint.
- Chapter 2, "Constraint Types," discusses the various types of constraints documented within this Guide, including CPLD fitter, grouping constraints, logical constraints, physical constraints, mapping directives, placement constraints, placement constraints, routing directives, synthesis constraints, timing constraints
- Chapter 3, "Entry Strategies for Xilinx Constraints," discusses entry strategies for Xilinx constraints, including which feature of the ISE software to use to enter a given constraint type.
- Chapter 4, "Timing Constraint Strategies," provides general guidelines that explain how to constrain the timing on designs when using the implementation tools for FPGA devices.
- Chapter 5, "Xilinx Constraints," describes the individual constraints that can be used with Xilinx FPGA and CPLD devices, including, for each constraint, architecture support, applicable elements, description, propagation rules, syntax examples, and, where necessary, additional information for particular constraints.

Additional Resources

To find additional documentation, see the Xilinx website at:

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

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

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

Conventions

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

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File →Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**`ngdbuild`** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **`bus[7:0]`**, they are required.	**`ngdbuild`** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**`lowpwr =`**{**`on`**|**`off`**}
Vertical bar |	Separates items in a list of choices	**`lowpwr =`**{**`on`**|**`off`**}

Vertical ellipsis
.
.
.
	Repetitive material that has been omitted	`IOB #1: Name = QOUT'`
		`IOB #2: Name = CLKIN'`
		.
		.
		.

Horizontal ellipsis ...	Repetitive material that has been omitted	**`allow block`** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

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

Table of Contents

Chapter 4: Timing Constraint Strategies

Chapter 5: Xilinx Constraints

Chapter 1

Introduction

This chapter discusses What's New in this Guide for ISE™ Release 9.1i, and provides a Supported Architectures table showing the Xilinx® devices supported for each constraint. This chapter contains the following sections:

- "What's New"
- "Supported Architectures"

What's New

The following changes have been made to this edition (ISE Release 9.1i) of the Xilinx *Constraints Guide*.

- "DCI_CASCADE" constraint added (Virtex™-5)
- "Hierarchical Lookup Table Name (HLUTNM)" constraint added (Virtex-5)
- "Enable Suspend (ENABLE_SUSPEND)" constraint added (Spartan™-3A)
- "Stepping (STEPPING)" constraint added
- "Attributes and Constraints" and "Configuration Constraints" sections added to Chapter 2, "Constraint Types"

Supported Architectures

The Supported Architectures table shows the Xilinx devices supported for each constraint. Contact Xilinx Technical Support if you need information for Xilinx architectures not shown.

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500XL\XV	CoolRunnerXPLA3	CoolRunner-II
Constraints A															
Area Group (AREA_GROUP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500\XL\XV	CoolRunnerXPLA3	CoolRunner-II
Asynchronous Register (ASYNC_REG)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints B															
BEL	No	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	No	No	No
Block Name (BLKNM)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
BUFG (CPLD)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Constraints C															
Clock Dedicated Route	No	No	No	No	No	Yes	Yes	No	No	Yes	Yes	Yes	No	No	No
Collapse (COLLAPSE)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Component Group (COMPGRP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Configuration Mode (CONFIG_MODE)	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No	No	No	No	No
CoolCLOCK (COOL_CLK)	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
Constraints D															
Data Gate (DATA_GATE)	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
DCI_CASCADE	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No
DCI_VALUE	No	No	Yes	Yes	Yes	Yes	Yes	No	No	Yes	No	No	No	No	No
Directed Routing (DIRECTED_ROUTING)	No	No	Yes	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes	No	No	No
Disable (DISABLE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
D rive (DRIVE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Drop Specifications (DROP_SPEC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Constraints E															
Enable (ENABLE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Enable Suspend (ENABLE_SUSPEND)	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
Constraints F															

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500\XL\XV	CoolRunnerXPLA3	CoolRunner-II
Fast (FAST)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Feedback (FEEDBACK)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
File (FILE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Float (FLOAT)	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
From Thru T o (FROM-THRU-TO)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
From To (FROM-TO)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Constraints H															
Hierarchical Block Name (HBLKNM)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Hierarchical Lookup Table Name (HLUTNM)	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No
HU_SET	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints I															
Input Buffer Delay Value (IBUF_DELAY_VALUE)	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes	No	No	No
IFD_DELAY_VALUE	No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes	No	No	No
Input Registers (INREG)	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes
IOB	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Input Output Block Delay (IOBDELAY)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Input Output Standard (IOSTANDARD)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
Constraints K															
Keep (KEEP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Keeper (KEEPER)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Keep Hierarchy (KEEP_HIERARCHY)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Constraints L															
Location (LOC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500\XL\XV	CoolRunnerXPLA3	CoolRunner-II
Locate (LOCATE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Lock Pins (LOCK_PINS)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Lookup Table Name (LUTNM)	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No
Constraints M															
Map (MAP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Maximum Delay (MAXDELAY)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Maximum Product Terms (MAXPT)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Maximum Skew (MAXSKEW)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints N															
No Delay (NODELAY)	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No
No Reduce (NOREDUCE)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Constraints O															
Offset In (OFFSET IN)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Offset Out (OFFSET OUT)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Open Drain (OPEN_DRAIN)	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
Optimizer Effort (OPT_EFFORT)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Optimize (OPTIMIZE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints P															
Period (PERIOD)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Pin (PIN)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
POST_CRC	No	No	No	No	No	No	Yes	No	No	No	Yes	No	No	No	No
POST_CRC_ACTION	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
POST_CRC_FREQ	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500\XL\XV	CoolRunnerXPLA3	CoolRunner-II
POST_CRC_SIGNAL	No	No	No	No	No	No	Yes	No	No	No	No	No	No	No	No
Priority (PRIORITY)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Prohibit (PROHIBIT)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Pulldown (PULLDOWN)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Pullup (PULLUP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Power Mode (PWR_MODE)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No
Constraints R															
Registers (REG)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	Yes	Yes
Relative Location (RLOC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Relative Location Origin (RLOC_ORIGIN)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Relative Location Range (RLOC_RANGE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints S															
Save Net Flag (SAVE NET FLAG)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Schmitt Trigger (SCHMITT_TRIGGER)	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
Slew (SLEW)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Slow (SLOW)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Stepping (STEPPING)	No	No	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes	No	No	No
Suspend (SUSPEND)	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
System Jitter (SYSTEM_JITTER)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Constraints T															
Temperature (TEMPERATURE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Timing Ignore (TIG)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Timing Group (TIMEGRP)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 1-1: **Supported Architectures**

Constraint	Architecture														
	Virtex	Virtex-E	Virtex-II	Virtex-II Pro	Virtex-II Pro X	Virtex-4	Virtex-5	Spartan-II	Spartan-IIE	Spartan-3	Spartan-3A	Spartan-3E	XC9500\XL\XV	CoolRunnerXPLA3	CoolRunner-II
Timing Specifications (TIMESPEC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Timing Name (TNM)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Timing Name Net (TNM_NET)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Timing Point Synchronization (TPSYNC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Timing Thru Points (TPTHRU)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Timing Specification Identifier (TSidentifier)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Constraints U															
U_SET	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Use Relative Location (USE_RLOC)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Use Low Skew Lines (USELOWSKEWLINES)	Yes	Yes	No	No	No	No	No	Yes	Yes	No	No	No	No	No	No
Constraints V-X															
VCCAUX	No	No	No	No	No	No	No	No	No	No	Yes	No	No	No	No
Voltage (VOLTAGE)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
VREF	No	No	No	No	No	No	No	No	No	No	No	No	No	No	Yes
Wire And (WIREAND)	No	No	No	No	No	No	No	No	No	No	No	No	Yes	No	No
XBLKNM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No

Chapter 2

Constraint Types

This chapter discusses the various types of constraints documented within this Guide. This chapter contains the following sections:

Attributes and Constraints

The terms *attribute* and *constraint* have been used interchangeably by some in the engineering community, while others ascribe different meanings to these terms. In addition, language constructs use the terms *attribute* and *directive* in similar yet different senses. For the purpose of clarification, the Xilinx® documentation refers to the terms *attributes* and *constraints* as defined below.

Attributes

An attribute is a property associated with a device architecture primitive component that generally affects an instantiated component's functionality or implementation. Attributes are passed as follows:

- In VHDL, by means of generic maps
- In Verilog, by means of defparams or inline parameter passing while instantiating the primitive component

Examples of attributes are:

- The INIT property on a LUT4 component
- The CLKFX_DIVIDE property on a DCM

All attributes are described in the appropriate Xilinx *Libraries Guide* as a part of the primitive component description.

Synthesis Constraints

Synthesis constraints direct the synthesis tool optimization technique for a particular design or piece of HDL code. They are either embedded within the VHDL or Verilog code, or within a separate synthesis constraints file. Examples of synthesis constraints are:

- USE_DSP48 (XST)
- RAM_STYLE (XST)

Synthesis constraints are documented as follows:

- XST constraints are documented in the Xilinx *XST User Guide*.
- Synthesis constraints for other synthesis tools are documented in the vendor's documentation for the tool. For more information on synthesis constraints for your synthesis tool, see the vendor documentation.

Implementation Constraints

Implementation constraints are instructions given to the FPGA implementation tools to direct the mapping, placement, timing or other guidelines for the implementation tools to follow while processing an FPGA design. Implementation constraints are generally placed in the UCF file, but may exist in the HDL code, or in a synthesis constraints file. Examples of implementation constraints are:

- LOC (placement) constraints
- PERIOD (timing) constraints

Implementation constraints are documented in the Xilinx *Constraints Guide*.

CPLD Fitter

The following constraints apply to CPLD devices:

"BUFG (CPLD)"	"Collapse (COLLAPSE)"	"CoolCLOCK (COOL_CLK)"
"Data Gate (DATA_GATE)"	"Fast (FAST)"	"Input Registers (INREG)"
"Input Output Standard (IOSTANDARD)"	"Keep (KEEP)"	"Keeper (KEEPER)"
"Location (LOC)"	"Maximum Product Terms (MAXPT)"	"No Reduce (NOREDUCE)"
"Offset In (OFFSET IN)" "Offset Out (OFFSET OUT)"	"Open Drain (OPEN_DRAIN)"	"Period (PERIOD)"
"Prohibit (PROHIBIT)"	"Pullup (PULLUP)"	"Power Mode (PWR_MODE)"
"Registers (REG)"	"Schmitt Trigger (SCHMITT_TRIGGER)"	"Slow (SLOW)"
"Timing Group (TIMEGRP)"	"Timing Specifications (TIMESPEC)"	"Timing Name (TNM)"

"Timing Specification Identifier (TSidentifier)" "VREF" "Wire And (WIREAND)"

Grouping Constraints

In a TS TIMESPEC attribute, specify the set of paths to be analyzed by grouping start and end points in one of the following ways.

- Refer to a predefined group by specifying one of the corresponding keywords: CPUS, DSPS, FFS, HSIOS, LATCHES, MULTS, PADS, RAMS, BRAMS_PORTA, or BRAMS_PORTB.
- Create your own groups within a predefined group by tagging symbols with "Timing Name (TNM)" (pronounced tee-name) and "Timing Name Net (TNM_NET)" constraints.
- Create groups that are combinations of existing groups using "Timing Group (TIMEGRP)" symbols.
- Create groups by pattern matching on net names. For more information, see "Creating Groups by Pattern Matching" in the "Timing Group (TIMEGRP)" constraint.

Using Predefined Groups

Using predefined groups, you can refer to a group of flip-flops, input latches, pads, or RAMs by using the corresponding keywords. See the following table.

Table 2-1: **Predefined Groups**

Keyword	Description
CPUS	PPC405 in Virtex™-II Pro, Virtex-II Pro X and Virtex-4 FX
DSPS	DSP48 and any DSP48 derivative in Virtex-4, Virtex-5 and Spartan-3A Extended
FFS	• All CLB and IOB edge-triggered flip-flops • Shift Register LUTs in Virtex and derived • Dual-data-rate registers in Virtex-II and derived (includes both flip-flops in the DDR)
HSIOS	GT and GT10 in Virtex-II Pro and Virtex-II Pro X
LATCHES	All CLB and IOB level-sensitive latches
MULTS	Multipliers, both sync and async, in Virtex-II and derived
PADS	All I/O pads (typically inferred from top level HDL ports)
RAMS	• All CLB LUT RAMs, both single- and dual-port (includes both ports of dual-port) • All block RAMs, both single-and dual-port (includes both ports of dual-port)
BRAMS_PORTA	Port A of all dual-port block RAMs
BRAMS_PORTB	Port B of all dual-port block RAMs

From-To statements enable you to define timing specifications for paths between predefined groups. The following examples are TS attributes that are entered in the UCF. This method enables you to easily define default timing specifications for the design, as illustrated by the following examples.

Predefined Group Examples

Following is a UCF syntax example.

```
TIMESPEC "TS01"=FROM FFS TO FFS 30;
TIMESPEC "TS02"=FROM LATCHES TO LATCHES 25;
TIMESPEC "TS03"=FROM PADS TO RAMS 70;
TIMESPEC "TS04"=FROM FFS TO PADS 55;
TIMESPEC "TS01" = FROM BRAMS_PORTA TO BRAMS_PORTB(gork*);
```

For BRAMS_PORTA and BRAM_PORTB, the specification TS01 controls paths that begin at any A port and end at a B port, which drives a signal matching the pattern **`gork*`**.

BRAMS_PORTA and BRAMS_PORTB Examples

Following are additional examples of BRAMS_PORTA and BRAMS_PORTB.

```
NET "X" TNM_NET = BRAMS_PORTA groupA;
```

The TNM group groupA contains all A ports that are driven by net X. If net X is traced forward into any B port inputs, any single-port block RAM elements, or any Select RAM elements, these do not become members of groupA.

```
NET "X" TNM_NET = BRAMS_PORTB( dob* ) groupB;
```

The TNM group groupB contains each B port driven by net X, if at least one output on that B port drives a signal matching the pattern dob*.

```
INST "Y" TNM = BRAMS_PORTB groupC;
```

The TNM group groupC contains all B ports found under instance Y. If instance Y is itself a dual-port block RAM primitive, then groupC contains the B port of that instance.

```
INST "Y" TNM = BRAMS_PORTA( doa* ) groupD;
```

The TNM group groupD contains each A port found under instance Y, if at least one output on that A port drives a signal matching the pattern doa*.

```
TIMEGRP "groupE" = BRAMS_PORTA;
```

The user group groupE contains the A ports of all dual-port block RAM elements in the design. This is equivalent to BRAMS_PORTA(*).

```
TIMEGRP "groupF" = BRAMS_PORTB( mem/dob* );
```

The user group groupF contains all B ports in the design, which drives a signal matching the pattern mem/dob*.

A predefined group can also carry a name qualifier. The qualifier can appear any place the predefined group is used. This name qualifier restricts the number of elements referred to. The syntax is:

predefined group **(***name_qualifier* [*name_qualifier*]**)**

name_qualifier is the full hierarchical name of the net that is sourced by the primitive being identified.

The name qualifier can include the following wildcard characters:

- An asterisk (*) to show any number of characters
- A question mark (?) to show a single character

Wildcard characters allow you to:

- Specify more than one net
- Shorten and simplify the full hierarchical name

For example, specifying the group FFS(MACRO_A/Q?) selects only the flip-flops driving the Q0, Q1, Q2 and Q3 nets.

The following constraints are grouping constraints:

"Component Group (COMPGRP)"	"Pin (PIN)"	"Timing Group (TIMEGRP)"
"Timing Name (TNM)"	"Timing Name Net (TNM_NET)"	"Timing Point Synchronization (TPSYNC)"
"Timing Thru Points (TPTHRU)"		

Logical Constraints

Logical constraints are constraints that are attached to elements in the design prior to mapping or fitting. Applying logical constraints helps you to adapt your design's performance to expected worst-case conditions. Later, when you choose a specific Xilinx® architecture, and place and route or fit your design, the logical constraints are converted into physical constraints.

You can attach logical constraints using attributes in the input design, which are written into the Netlist Constraints File (NCF) or NGC netlist, or with a User Constraints File (UCF).

Three categories of logical constraints are:

- "Placement Constraints"
- "Relative Location (RLOC) Constraints"
- "Timing Constraints"

For FPGA devices, relative location constraints (RLOCs) group logic elements into discrete sets. They allow you to define the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design. For more information, see "Relative Location (RLOC) Constraints" in this chapter.

Timing constraints allow you to specify the maximum allowable delay or skew on any given set of paths or nets in your design.

Physical Constraints

Constraints can also be attached to the elements in the physical design, that is, the design after mapping has been performed. These constraints are referred to as physical constraints. They are defined in the Physical Constraints File (PCF), which is created during mapping.

Xilinx recommends that you place any user-generated constraint in the UCF file, not in an NCF or PCF file.

Note: The information in this section applies to FPGA device families only.

When a design is mapped, the logical constraints found in the netlist and the UCF file are translated into physical constraints, that is, constraints that apply to a specific architecture. These constraints are found in a mapper-generated file called the Physical Constraints File (PCF).

The PCF file contains two sections:

- The schematic section, which contains the physical constraints based on the logical constraints found in the netlist and the UCF file
- The user section, which can be used to add any physical constraints

Mapping Directives

Mapping directives instruct the mapper to perform specific operations. The following constraints are mapping directives:

"Area Group (AREA_GROUP)"	"BEL"	"Block Name (BLKNM)"
"DCI_VALUE"	"D rive (DRIVE)"	"Fast (FAST)"
"Hierarchical Block Name (HBLKNM)"	"Hierarchical Lookup Table Name (HLUTNM)"	"HU_SET"
"IOB"	"Input Output Block Delay (IOBDELAY)"	"Input Output Standard (IOSTANDARD)"
"Keep (KEEP)"	"Keeper (KEEPER)"	"Lookup Table Name (LUTNM)"
"Map (MAP)"	"No Delay (NODELAY)"	"Optimize (OPTIMIZE)"
"Pulldown (PULLDOWN)"	"Pullup (PULLUP)"	"Relative Location (RLOC)"
"Relative Location Origin (RLOC_ORIGIN)"	"Relative Location Range (RLOC_RANGE)"	"Save Net Flag (SAVE NET FLAG)"
"Slew (SLEW)"	"U_SET"	"Use Relative Location (USE_RLOC)"
"XBLKNM"		

Placement Constraints

This section describes the placement constraints for each type of logic element in FPGA designs, such as:

- Flip-flops
- ROMs and RAMs
- BUFTs
- CLBs

- IOBs
- I/Os
- Edge decoders
- Global buffers

Individual logic gates, such as AND or OR gates, are mapped into CLB function generators before the constraints are read, and therefore cannot be constrained.

The following constraints control mapping and placement of symbols in a netlist:

- "Block Name (BLKNM)"
- "Hierarchical Block Name (HBLKNM)"
- "Hierarchical Lookup Table Name (HLUTNM)"
- "Location (LOC)"
- "Lookup Table Name (LUTNM)"
- "Prohibit (PROHIBIT)"
- "Relative Location (RLOC)"
- "Relative Location Origin (RLOC_ORIGIN)"
- "Relative Location Range (RLOC_RANGE)"
- "XBLKNM"

Most constraints can be specified either in the HDL or in the UCF file. In a constraints file, each placement constraint acts upon one or more symbols. Every symbol in a design carries a unique name, which is defined in the input file. Use this name in a constraint statement to identify the symbol.

The UCF and NCF files are case sensitive. Identifier names (names of objects in the design, such as net names) must exactly match the case of the name as it exists in the source design netlist. However, any Xilinx constraint keyword (for example, LOC, PROHIBIT, RLOC, BLKNM) can be entered in either all upper-case or all lower-case letters. Mixed case is not allowed.

Relative Location (RLOC) Constraints

The RLOC constraint groups logic elements into discrete sets. You can define the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design. For example, if RLOC constraints are applied to a group of eight flip-flops organized in a column, the mapper maintains the columnar order and moves the entire group of flip-flops as a single unit. In contrast, absolute location (LOC) constraints constrain design elements to specific locations on the FPGA die with no relation to other design elements.

Placement Constraints

The following constraints are placement constraints:

"Area Group (AREA_GROUP)"
"BEL"
"Location (LOC)"
"Locate (LOCATE)"
"Optimizer Effort (OPT_EFFORT)"
"Prohibit (PROHIBIT)"
"Relative Location (RLOC)"
"Relative Location Origin (RLOC_ORIGIN)"
"Relative Location Range (RLOC_RANGE)"
"Use Relative Location (USE_RLOC)"

Routing Directives

Routing directives instruct PAR to perform specific operations. The following constraints are routing directives:

- Area Group (AREA_GROUP)
- Configuration Mode (CONFIG_MODE)
- Lock Pins (LOCK_PINS)
- Optimizer Effort (OPT_EFFORT)
- Use Low Skew Lines (USELOWSKEWLINES)

Synthesis Constraints

Synthesis constraints instruct the synthesis tool to perform specific operations. The following constraints are synthesis constraints:

"From To (FROM-TO)"
"IOB"
"Keep (KEEP)"
"Map (MAP)"
"Offset In (OFFSET IN)"
"Period (PERIOD)"
"Offset Out (OFFSET OUT)"
"Timing Ignore (TIG)"
"Timing Name (TNM)"
"Timing Name Net (TNM_NET)"

Timing Constraints

Xilinx software enables you to specify precise timing constraints for your Xilinx designs. You can specify the timing constraints for any nets or paths in your design, or you can specify them globally. One way of specifying path requirements is to first identify a set of paths by identifying a group of start and end points. The start and end points can be flip-flops, I/O pads, latches, or RAMs. You can then control the worst-case timing on the set of paths by specifying a single delay requirement for all paths in the set.

The primary way to specify timing constraints is to enter them in your design (HDL and schematic). However, you can also specify timing constraints in constraints files (UCF, NCF, PCF, XCF). For more information about each constraint, see the later chapters in this Guide.

Once you define timing specifications and map the design, PAR places and routes your design based on these requirements.

To analyze the results of your timing specifications, use the command line tool, TRACE (TRCE) or the ISE™ Timing Analyzer.

XST Timing Constraints

XST supports an XCF (XST Constraints File) syntax to define synthesis and timing constraints. The constraint syntax in use prior to the ISE 7.1i release is no longer supported.

Timing constraints supported by XST can be applied via either:

- The **`-glob_opt`** command line switch
- The constraints file

Command Line Switch

Using the **`-glob_opt`** command line switch is the same as selecting **Process Properties > Synthesis Options > Global Optimization Goal**. Using this method allows you to apply global timing constraints to the entire design. You cannot specify a value for these constraint; XST optimizes them for the best performance. These constraints are overridden by constraints specified in the constraints file.

Constraints File

Using the constraint file method, you can use the native UCF timing constraint syntax. Using the XCF syntax, XST supports constraints such as TNM_NET, TIMEGRP, PERIOD, TIG, FROM-TO, including wildcards and hierarchical names.

> ***Note:*** Timing constraints are written to the NGC file only when the Write Timing Constraints property is checked in the Process Properties dialog box in Project Navigator, or the *-write_timing_constraints* option is specified when using the command line. By default, they are not written to the NGC file.

Independent of the way timing constraints are specified, the Clock Signal option affects timing constraint processing. In the case where a clock signal goes through which input pin is the real clock pin. The CLOCK_SIGNAL constraint allows you to define the clock pin. For more information, see the Xilinx *XST User Guide*.

UCF Timing Constraint Support

> ***Caution!*** If you specify timing constraints in the XCF file, Xilinx strongly suggests that you to use the '/' character as a hierarchy separator instead of '_'.

The following timing constraints are supported in the XST Constraints File (XCF).

From-To

FROM-TO defines a timing constraint between two groups. A group can be user-defined or predefined (FFS, PADS, RAMS). For more information, see the "From To (FROM-TO)" constraint. Following is an example of XCF Syntax:

```
TIMESPEC "TSname"=FROM "group1" TO "group2" value;
```

OFFSET IN

The OFFSET IN constraint is used to specify the timing requirements of an input interface to the FPGA. The constraint specifies the clock and data timing relationship at the external pads of the FPGA. An OFFSET IN constraint specification checks the setup and hold timing requirements of all synchronous elements associated with the constraint. The following image shows the paths covered by the OFFSET IN constraint. For more information, see the "Offset In (OFFSET IN)" constraint.

OFFSET OUT

The OFFSET OUT constraint is used to specify the timing requirements of an output interface from the FPGA. The constraint specifies the time from the clock edge at the input pin of the FPGA until data becomes valid at the outp pin of the FPGA. For more information, see the"Offset Out (OFFSET OUT)" constraint.

TIG

The "Timing Ignore (TIG)" constraint causes all paths going through a specific net to be ignored for timing analyses and optimization purposes. This constraint can be applied to the name of the signal affected.

XCF Syntax:

```
NET "netname" TIG;
```

TIMEGRP

"Timing Group (TIMEGRP)" is a basic grouping constraint. In addition to naming groups using the TNM identifier, you can also define groups in terms of other groups. You can create a group that is a combination of existing groups by defining a TIMEGRP constraint.

You can place TIMEGRP constraints in a constraints file (XCF or NCF). You can use TIMEGRP attributes to create groups using the following methods.

- Combining multiple groups into one
- Defining flip-flop subgroups by clock sense

XCF Syntax:

```
TIMEGRP "newgroup"="existing_grp1" "existing_grp2"
["existing_grp3" . . .];
```

TNM

"Timing Name (TNM)" is a basic grouping constraint. Use TNM (Timing Name) to identify the elements that make up a group, which you can then use in a timing specification. TNM tags specific FFS, RAMs, LATCHES, PADS, BRAMS_PORTA, BRAMS_PORTB, CPUS, HSIOS, and MULTS as members of a group to simplify the application of timing specifications to the group.

The RISING and FALLING keywords may also be used with TNMs.

XCF Syntax:

```
{NET|INST|PIN} "net_or_pin_or_inst_name" TNM=[predefined_group]
identifier;
```

TNM Net

"Timing Name Net (TNM_NET)" is essentially equivalent to TNM on a net *except* for input pad nets. Special rules apply when using TNM_NET with the "Period (PERIOD)" constraint for DLL/DCM/PLLs in the following devices:

- Virtex
- Virtex-E
- Virtex-II
- Virtex-II Pro
- Virtex-II Pro X
- Virtex-4
- Virtex-5
- Spartan-II
- Spartan-IIE
- Spartan-3
- Spartan-3A
- Spartan-3E

For more information, see "PERIOD Specifications on CLKDLLs, DCMs and PLLs" in the "Period (PERIOD)" constraint.

A TNM_NET is a property that you normally use in conjunction with an HDL design to tag a specific net. All downstream synchronous elements and pads tagged with the TNM_NET identifier are considered a group. For more information, see the "Timing Name (TNM)"constraint.

XCF Syntax:

```
NET "netname" TNM_NET=[predefined_group] identifier;
```

Timing Model

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

Priority

Constraints are processed in the following order:

- Specific constraints on signals
- Specific constraints on top module
- Global constraints on top module

For example, constraints on two different domains or two different signals have the same priority (that is, PERIOD clk1 can be applied with PERIOD clk2).

Timing and Grouping Constraints

The following are timing constraints and associated grouping constraints:

- "Asynchronous Register (ASYNC_REG)"
- "Disable (DISABLE)"
- "Drop Specifications (DROP_SPEC)"
- "Enable (ENABLE)"
- "From Thru T o (FROM-THRU-TO)"
- "From To (FROM-TO)"
- "Maximum Skew (MAXSKEW)"
- "Offset In (OFFSET IN)"
- "Offset Out (OFFSET OUT)"
- "Period (PERIOD)"
- "Priority (PRIORITY)"
- "System Jitter (SYSTEM_JITTER)"
- "Temperature (TEMPERATURE)"
- "Timing Ignore (TIG)"
- "Timing Group (TIMEGRP)"
- "Timing Specifications (TIMESPEC)"
- "Timing Name (TNM)"
- "Timing Name Net (TNM_NET)"
- "Timing Point Synchronization (TPSYNC)"
- "Timing Thru Points (TPTHRU)"
- "Timing Specification Identifier (TSidentifier)"
- "Voltage (VOLTAGE)"

Configuration Constraints

The following are configuration constraints:

- "Configuration Mode (CONFIG_MODE)"
- "DCI_CASCADE"
- "Stepping (STEPPING)"
- "POST_CRC"
- "POST_CRC_ACTION"
- "POST_CRC_FREQ"
- "POST_CRC_SIGNAL"
- "VCCAUX"
- "VREF"

Chapter 3

Entry Strategies for Xilinx Constraints

This chapter discusses entry strategies for Xilinx® constraints, including which feature of the ISE™ software to use to enter a given constraint type. This chapter contains the following sections:

Constraints Entry Methods

The following table shows which feature of the ISE software to use to enter a given constraint type.

Table 3-1: **Constraints Entry Methods**

ISE Tool	Constraint Type	Devices
Constraints Editor	Timing	All CPLD and FPGA device families
Floorplanner	Non-timing placement constraints	All FPGA device families

Table 3-1: **Constraints Entry Methods**

ISE Tool	Constraint Type	Devices
PACE	IO placement and area-group constraints	All CPLD and FPGA device families except Virtex™-5 and Spartan™-3A
Floorplan Editor	IO placement and area-group constraints	Virtex-4, Virtex-5 and Spartan-3A devices only
Schematic and Symbol Editors	IO placement and RLOC constraints	All CPLD and FPGA device families

Constraints Entry Table

The following table lists the constraints and their associated entry strategies. See the individual constraint for syntax examples.

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Constraints A													
Area Group (AREA_GROUP)	√			√	√	√			√	√	√		
Asynchronous Register (ASYNC_REG)		√		√	√	√							
Constraints B													
BEL	√	√		√	√				√				
Block Name (BLKNM)	√	√		√	√			√					
BUFG (CPLD)	√	√	√	√	√			√					
Constraints C													
Clock Dedicated Route	√			√	√								
Collapse (COLLAPSE)	√	√		√	√								
Component Group (COMPGRP)							√						
Configuration Mode (CONFIG_MODE)					√								
CoolCLOCK (COOL_CLK)	√	√	√	√	√								

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Constraints D													
Data Gate (DATA_GATE)	√	√	√	√	√								
DCI_CASCADE				√	√		√						
DCI_VALUE				√	√								
Directed Routing (DIRECTED_ROUTING)				√	√							√	
Disable (DISABLE)				√	√		√						
D rive (DRIVE)	√	√		√	√			√		√	√		
Drop Specifications (DROP_SPEC)				√	√		√						
Constraints E													
Enable (ENABLE)				√	√		√						
Enable Suspend (ENABLE_SUSPEND)				√	√								
Constraints F													
Fast (FAST)	√	√	√	√	√			√		√	√		
Feedback (FEEDBACK)					√	√	√	√					
File (FILE)	√	√											
Float (FLOAT)	√	√	√	√	√			√					
From Thru T o (FROM-THRU-TO)				√	√	√	√						
From To (FROM-TO)				√	√	√	√	√					
Constraints H													
Hierarchical Block Name (HBLKNM)	√	√		√	√								
Hierarchical Lookup Table Name (HLUTNM)	√	√		√	√	√		√		√			
HU_SET	√	√		√	√			√					

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Constraints I													
Input Buffer Delay Value (IBUF_DELAY_VALUE)	√	√		√	√								
IFD_DELAY_VALUE	√	√		√	√								
Input Registers (INREG)	√		√		√								
IOB	√	√		√	√			√		√	√		√
Input Output Block Delay (IOBDELAY)	√	√		√	√					√	√		
Input Output Standard (IOSTANDARD)	√	√	√	√	√			√		√	√		
Constraints K													
Keep (KEEP)	√	√	√	√	√			√					
Keeper (KEEPER)	√	√	√	√	√	√		√					
Keep Hierarchy (KEEP_HIERARCHY)	√	√		√	√			√					√
Constraints L													
Location (LOC)	√	√	√*	√	√		√	√	√	√	√		
Note: * Pin assignments are specified in ABEL PIN declarations without using the LOC keyword.													
Locate (LOCATE)							√					√	
Lock Pins (LOCK_PINS)		√		√	√								
Lookup Table Name (LUTNM)	√	√		√	√								
Constraints M													
Map (MAP)	√			√	√								
Maximum Delay (MAXDELAY)	√	√		√	√	√	√					√	
Maximum Product Terms (MAXPT)		√	√	√	√								
Maximum Skew (MAXSKEW)	√	√		√	√	√	√					√	

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Constraints													
No Delay (NODELAY)	√	√		√	√			√					
No Reduce (NOREDUCE)	√	√	√*	√	√			√					
Note: * Specified using ABEL-specific keyword RETAIN.													
Constraints O													
Offset In (OFFSET IN)	√			√	√	√	√	√					
Offset Out (OFFSET OUT)	√			√	√	√	√	√					
Open Drain (OPEN_DRAIN)	√	√	√	√	√			√					
Optimizer Effort (OPT_EFFORT)	√			√	√								√
Optimize (OPTIMIZE)	√	√		√	√								√
Constraints P													
Period (PERIOD)	√	√		√	√	√	√	√				√	
Pin (PIN)					√								
POST_CRC					√		√						
POST_CRC_ACTION					√		√						
POST_CRC_FREQ					√		√						
POST_CRC_SIGNAL					√		√						
Priority (PRIORITY)				√	√		√						
Prohibit (PROHIBIT)					√		√		√	√	√	√	
Pulldown (PULLDOWN)	√	√		√	√			√		√	√		
Pullup (PULLUP)	√	√	√	√	√			√		√	√		
Power Mode (PWR_MODE)	√	√	√	√	√			√					
Constraints R													
Registers (REG)	√	√	√	√	√			√					
Relative Location (RLOC)	√	√		√	√			√	√				

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Relative Location Origin (RLOC_ORIGIN)	√	√		√	√		√		√				
Relative Location Range (RLOC_RANGE)	√	√		√	√		√	√					
Constraints S													
Save Net Flag (SAVE NET FLAG)	√	√		√	√			√					
Schmitt Trigger (SCHMITT_TRIGGER)	√	√	√	√	√			√					
Slew (SLEW)	√	√		√	√			√		√	√		
Slow (SLOW)	√	√	√	√	√			√		√	√		
Stepping (STEPPING)					√								
Suspend (SUSPEND)	√	√		√	√						√		
System Jitter (SYSTEM_JITTER)	√	√		√	√			√					
Constraints T													
Temperature (TEMPERATURE)				√	√	√	√						
Timing Ignore (TIG)	√			√	√	√	√	√					
Timing Group (TIMEGRP)				√	√	√	√	√					
Timing Specifications (TIMESPEC)				√	√	√		√					
Timing Name (TNM)	√		√	√	√	√		√					
Timing Name Net (TNM_NET)	√			√	√	√		√					
Timing Point Synchronization (TPSYNC)	√			√	√								
Timing Thru Points (TPTHRU)	√			√	√	√							
Timing Specification Identifier (TSidentifier)				√	√	√	√	√				√	
Constraints U													
U_SET	√	√		√	√			√					

Table 3-2: Constraints Entry Table

Constraint	Schematic	VHDL\Verilog	ABEL	NCF	UCF	Constraints Editor	PCF	XCF	Floorplanner	Floorplan Editor	PACE	FPGA Editor	Project Navigator
See Table 3-1, "Constraints Entry Methods," above for the Constraint Type and Devices with which each of these tools can be used.													
Use Relative Location (USE_RLOC)	√	√		√	√			√					
Use Low Skew Lines (USELOWSKEWLINES)	√	√		√	√	√	√	√					
Constraints V													
VCCAUX				√	√								
Voltage (VOLTAGE)				√	√	√	√						
VREF	√			√	√								
Constraints W													
Wire And (WIREAND)	√	√		√	√								
Constraints X													
XBLKNM	√	√		√	√			√					

Schematic Design

To add Xilinx constraints as attributes within a symbol or schematic drawing, follow these rules:

- If a constraint applies to a net, add it as an attribute to the net.
- If a constraint applies to an instance, add it as an attribute to the instance.
- You cannot add global constraints such as PART and PROHIBIT.
- You cannot add any timing specifications that would be attached to a TIMESPEC or TIMEGRP.
- Enter attribute names and values in either all upper case or all lower case. Mixed upper and lower case is not allowed.

For more information about creating, modifying, and displaying attributes, see the Schematic and Symbol Editors help.

In the this Guide, the syntax for any constraint that can be entered in a schematic is described in the individual section for the constraint. For an example of correct schematic syntax, see "Schematic Syntax Example" in the "BEL" constraint.

VHDL

In VHDL code, constraints can be specified with VHDL attributes. Before it can be used, a constraint must be declared with the following syntax:

```
attribute attribute_name : string;
```

Example:

```
attribute RLOC : string;
```

An attribute can be declared in an entity or architecture.

- If the attribute is declared in the entity, it is visible both in the entity and the architecture body.
- If the attribute is declared in the architecture, it cannot be used in the entity declaration.

Once the attribute is declared, you can specify a VHDL attribute as follows:

```
attribute attribute_name of
{component_name|label_name|entity_name|signal_name
|variable_name|type_name}: {component|label|entity|signal
|variable|type} is attribute_value;
```

Accepted *attribute_values* depend on the attribute type.

Examples:

```
attribute RLOC of u123 : label is "R11C1.S0";
attribute bufg of my_clock: signal is “clk”;
```

For Xilinx, the most common objects are **signal**, **entity**, and **label**. A label describes an instance of a component.

VHDL is case insensitive.

In some cases, existing Xilinx constraints cannot be used in attributes, since they are also VHDL keywords. To avoid this naming conflict, use a constraint alias. Each Xilinx constraint has its own alias. The alias is the original constraint name prepended with the prefix "XIL_". For example, the "RANGE" constraint cannot be used in an attribute directly. Use "XIL_RANGE" instead.

Verilog

You can specify constraints as follows in Verilog code:

```
(* ATTRIBUTE_NAME = "attribute_value" *)
```

The *attribute_value* is case sensitive.

Examples:

```
(* RLOC = "R11C1.S0" *)
(* HU_SET = "MY_SET" *)
(* BUFG = "clk" *)
```

ABEL

Xilinx supports the use of ABEL for CPLD devices.

Following is an example of the basic syntax.

```
XILINX PROPERTY 'bufg=clk my_clock';
```

UCF

The UCF file is an ASCII file specifying constraints on the logical design. You can create this file and enter your constraints with any text editor. You can also use the Constraints Editor to create constraints within a UCF file. For more information, see "Constraints Editor" in this chapter.

These constraints affect how the logical design is implemented in the target device. You can use the file to override constraints specified during design entry.

UCF Flow

The following figure illustrates the UCF flow.

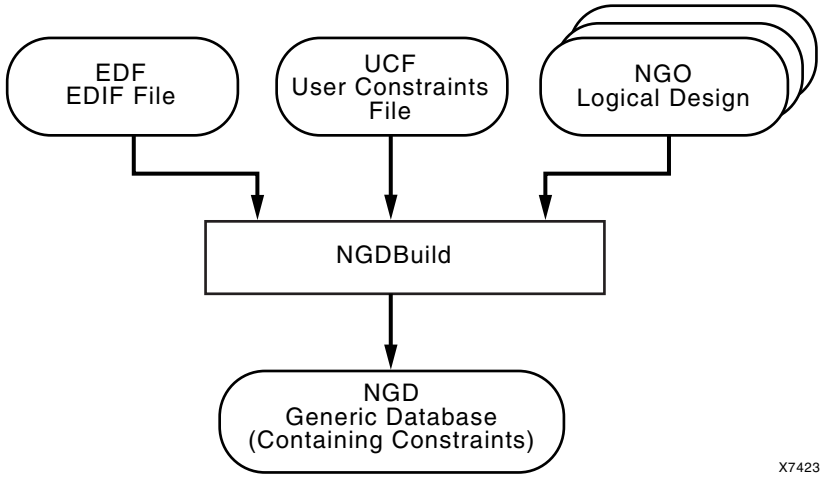

Figure 3-1: **UCF File Flow**

The UCF file is an input to NGDBuild (see the preceding figure). The constraints in the UCF file become part of the information in the NGD file produced by NGDBuild. For FPGA devices, some of these constraints are used when the design is mapped by MAP and some of the constraints are written into the PCF (Physical Constraints File) produced by MAP.

The constraints in the PCF file are used by each of the physical design tools (for example, PAR and the timing analysis tools), which are run after the design is mapped.

Manual Entry of Timing Constraints

You can manually enter timing specifications as constraints in a UCF file. When you run NGDBuild on the design, the timing constraints are added to the design database as part of the NGD file. To avoid manually entering timing constraints in a UCF file, use the Xilinx Constraints Editor.

UCF and NCF File Syntax

Logical constraints are found in:

- The Netlist Constraint File (NCF), an ASCII file generated by synthesis programs
- The User Constraint File (UCF), an ASCII file generated by the user

Xilinx recommends that you place user-generated constraints in the UCF file — *not* in an NCF or PCF file.

General Rules

Following are some general rules for the UCF and NCF files.

- The UCF and NCF files are case sensitive. Identifier names (names of objects in the design, such as net names) must exactly match the case of the name as it exists in the source design netlist. However, any Xilinx constraint keyword (for example, LOC, PERIOD, HIGH, LOW) may be entered in all upper-case, all lower-case, or mixed case.
- Each statement is terminated by a semicolon (;).
- No continuation characters are necessary if a statement exceeds one line, since a semicolon marks the end of the statement.
- Xilinx recommends that you group similar blocks, or components, as a single timing constraint, and not as separate timing constraints.
- To add comments to the UCF and NCF file, begin each comment line with a pound (#) sign, as in the following example.

  ```
  # file TEST.UCF
  # net constraints for TEST design
  NET “$SIG_0“ MAXDELAY = 10;
  NET “$SIG_1“ MAXDELAY = 12 ns;
  ```

  C and C++ style comments (`/* */` and respectively) are also supported.
- Statements need not be placed in any particular order in the UCF and NCF file.
- Enclose NET and INST names in double quotes (recommended but not mandatory).
- Enclose inverted signal names that contain a tilde (for example, ~OUTSIG1) in double quotes (mandatory).
- You can enter multiple constraints for a given instance. For more information, see “Entering Multiple Constraints” in this chapter.

Conflict in Constraints

The constraints in the UCF and NCF files and the constraints in the schematic or synthesis file are applied equally. It does not matter whether a constraint is entered in the schematic or synthesis file, or in the UCF and NCF files. If the constraints overlap, UCF overrides NCF and schematic constraints. NCF overrides schematic constraints.

If by mistake two or more elements are locked onto a single location, the mapper detects the conflict, issues an error message, and stops processing so that you can correct the mistake.

Syntax

The UCF file supports a basic syntax that can be expressed as:

```
{NET|INST|PIN} "full_name" constraint;
```

or as

```
SET set_name set_constraint;
```

where

- *full_name* is a full hierarchically qualified name of the object being referred to. When the name refers to a pin, the instance name of the element is also required.
- *constraint* is a constraint in the same form as it would be used if it were attached as an attribute on a schematic object. For example, LOC=P38 and FAST.
- *set_name* is the name of an RLOC set. For more information, see "RLOC Description" in the "Relative Location (RLOC)" constraint.
- *set_constraint* is an RLOC_ORIGIN or RLOC_RANGE constraint.

Specifying Attributes for TIMEGRP and TIMESPEC

To specify attributes for TIMEGRP, the keyword TIMEGRP precedes the attribute definitions in the constraints files.

```
TIMEGRP "input_pads"=PADS EXCEPT output_pads;
```

Using Reserved Words

In all of the constraints files (NCF, UCF, and PCF), instance or variable names that match internal reserved words may be rejected unless the names are enclosed in double quotes. It is good practice to enclose all names in double quotes.

For example, the following entry would not be accepted because the word "net" is a reserved word.

```
NET net OFFSET=IN 20 BEFORE CLOCK;
```

Following is the recommended way to enter the constraint.

```
NET "net" OFFSET=IN 20 BEFORE CLOCK;
```

or

```
NET "$SIG_0" OFFSET=IN 20 BEFORE CLOCK;
```

Enclose inverted signal names that contain a tilde (for example, ~OUTSIG1) in double quotes (mandatory) as follows:

```
NET "~OUTSIG1" OFFSET=IN 20 BEFORE CLOCK;
```

Wildcards

You can use the wildcard characters, asterisk (*) and question mark (?) in constraint statements as follows:

- The asterisk (*) represents any string of zero or more characters.
- The question mark (?) indicates a single character.

In net names, the wildcard characters enable you to select a group of symbols whose output net names match a specific string or pattern. For example, the constraint shown

below increases the output speed of pads to which nets are connected with names that meet the following patterns:

- They begin with any series of characters (represented by an asterisk [*]).
- The initial characters are followed by "AT."
- The net names end with one single character (represented by a question mark [?].

```
NET "*AT?" FAST;
```

In an instance name, a wildcard character by itself represents every symbol of the appropriate type. For example, the following constraint initializes an entire set of ROMs to a particular hexadecimal value, 5555.

```
INST "$1I3*/ROM2" INIT=5555;
```

If the wildcard character is used as part of a longer instance name, the wildcard represents one or more characters at that position.

In a location, you can use a wildcard character for either the row number or the column number. For example, the following constraint specifies placement of any instance under the hierarchy of loads_of_logic in any CLB in column 8.

```
INST "/loads_of_logic/*" LOC=CLB_r*c8;
```

Wildcard characters can be used in dot extensions.

```
CLB_R1C3.*
```

Wildcard characters cannot be used for both the row number and the column number in a single constraint, since such a constraint is meaningless.

Traversing Hierarchies

Top-level block names (design names) are ignored when searching for instance name matches. You can use the asterisk wildcard character (*) to traverse the hierarchy of a design within a UCF and NCF file. The following syntax applies (where level1 is an example hierarchy level name).

*	Traverses all levels of the hierarchy
level1/*	Traverses all blocks in level1 and below
level1/*/	Traverses all blocks in the level1 hierarchy level but no further

Consider the following design hierarchy.

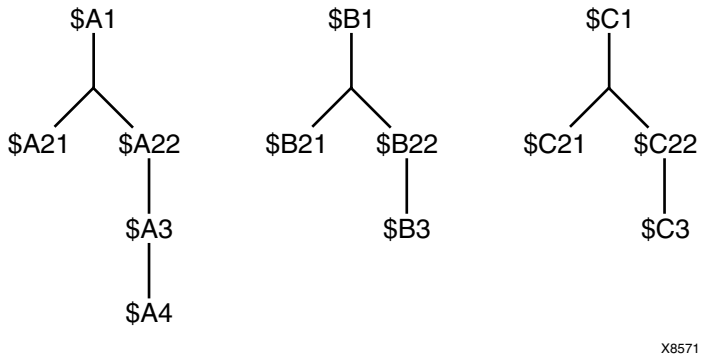

Figure 3-2: **UCF Design Hierarchy**

With the example design hierarchy, the following specifications illustrate the scope of the wildcard.

```
INST *          => <everything>
INST /*         => <everything>
INST /*/        => <$A1,$B1,$C1>
INST $A1/*      => <$A21,$A22,$A3,$A4>
INST $A1/*/     => <$A21,$A22>
INST $A1/*/*    => <$A3,$A4>
INST $A1/*/*/   => <$A3>
INST $A1/*/*/*  => <$A4>
INST $A1/*/*/*/ => <$A4>
INST /*/*22/    => <$A22,$B22,$C22>
INST /*/*22     => <$A22,$A3,$A4,$B22,$B3,$C3>
```

Entering Multiple Constraints

You can cascade multiple constraints for a given instance in the UCF file:

INST *instanceName constraintName* **=** *constraintValue* | *constraintName* **=** *constraintValue*;

For example:

```
INST myInst LOC = P53 | IOSTANDARD = LVPECL33 | SLEW = FAST;
```

File Name

By default, NGDBuild reads the constraints file that carries the same name as the input design with a .ucf extension. However, you can specify a different constraints file name with the **-uc** option when running NGDBuild. NGDBuild automatically reads in the NCF file if it has the same base name as the input EDIF file and is in the same directory as the EDIF file.

The implementation tools (for example, NGDBuild, MAP, and PAR) require file name extensions in all lowercase (for example, .ucf) in command lines.

Instances and Blocks

The statements in the constraints file concern instances and blocks, which are defined as follows.

- An *instance* is a symbol on the schematic.
- An *instance name* is the symbol name as it appears in the EDIF netlist.
- A *block* is a CLB, an IOB, or a TBUF.
- Specify the *block name* with the BLKNM, HBLKNM, or XBLKNM attributes. By default, the software assigns a block name on the basis of a signal name associated with the block.

PCF

The NGD file produced when a design netlist is read into the Xilinx Development System may contain a number of logical constraints. These constraints originate in any of these sources.

- An attribute assigned within a schematic or HDL file
- A constraint entered in a UCF (User Constraints File)
- A constraint appearing in an NCF (Netlist Constraints File) produced by a CAE vendor toolset

Logical constraints in the NGD file are read by MAP. MAP uses some of the constraints to map the design and converts logical constraints to physical constraints. MAP then writes these physical constraints into a Physical Constraints File (PCF).

The PCF file is an ASCII file containing two separate sections:

- A section for those physical constraints created by the mapper
- A section for physical constraints entered by the user

The mapper section is rewritten every time you run the mapper.

Mapper-generated physical constraints appear first in the file, followed by user physical constraints. In the event of conflicts between mapper-generated and user constraints, user constraints are read last, and override mapper-generated constraints.

The mapper-generated section of the file is preceded by a **`SCHEMATIC START`** notation on a separate line. The end of this section is indicated by **`SCHEMATIC END`**, also on a separate line. Enter user-generated constraints, such as timing constraints, after **`SCHEMATIC END`**.

You can write user constraints directly into the file or you can write them indirectly (or undo them) from within the FPGA Editor. For more information on constraints in the FPGA Editor, see the FPGA Editor help.

Note: Whenever possible, you should add design constraints to the HDL, schematic, or UCF, instead of PCF. This simplifies design archiving and improves design role checking.

The PCF file is an optional input to PAR, FPGA Editor, TRACE, NetGen, and BitGen.

The file may contain any number of constraints, and any number of comments, in any order. A comment consists of either a pound sign (#) or double slashes (//) ,followed by any number of other characters up to a new line. Each comment line must begin with # or //.

The structure of the PCF file is as follows.

```
schematic start;
translated schematic and UCF and NCF constraints in PCF format
schematic end;
user-entered physical constraints
```

Caution! Put all user-entered physical constraints after the “schematic end” statement. *Any constraints preceding this section or within this section may be overwritten or ignored.*

Do not edit the schematic constraints. They are overwritten every time the mapper generates a new PCF file.

Global constraints need not be attached to any object, but should be entered in a constraints file.

Indicate the end of each constraint statement with a semi-colon.

In all of the constraints files (NCF, UCF, and PCF), instance or variable names that match internal reserved words are rejected unless the names are enclosed in double quotes. It is good practice to enclose all names in double quotes. For example, the following entry would not be accepted because the word *net* is a reserved word.

```
NET net FAST;
```

Following is the recommended way to enter the constraint.

```
NET “net” FAST;
```

NCF

The syntax rules for NCF files are the same as those for the UCF file. For more information, see “UCF and NCF File Syntax” in this chapter.

Constraints Editor

Use Constraints Editor to enter timing constraints . The user interface simplifies constraint entry by guiding you through constraint creation without your needing to understand UCF file syntax.

For the constraints and devices with which Constraints Editor can be used, see “Constraints Entry Methods” above. For information on running Constraints Editor, see the ISE Help

Used in the implementation phase of the design after the translation step (NGDBuild), Constraints Editor allows you to create and manipulate constraints without any direct editing of the UCF. After the constraints are created or modified with Constraints Editor, NGDBuild must be run again, using the new UCF and design source netlist files as input and generating a new NGD file as output.

Input/Output

Constraints Editor requires:

- A User Constraints File (UCF)
- A Native Generic Database (NGD) file

Constraints Editor uses the NGD to provide names of logical elements for grouping. As output, it uses the UCF.

After you open Constraints Editor, you must first open a UCF file. If the UCF and NGD root names are not the same, you must select the appropriate NGD file to open. For more information, see the Constraints Editor help.

Upon successful completion, Constraints Editor writes out a UCF. NGDBuild (translation) uses the UCF, along with design source netlists, to produce an NGD file. The NGD file is read by the MAP program. MAP generates a physical design database in the form of an NCD (Native Circuit Description) file and also generates a PCF (Physical Constraints File). The implementation tools use these files to ultimately produce a bitstream.

Not all Xilinx constraints are accessible through Constraints Editor. Constraints supported in Constraints Editor and the associated UCF syntax are described in "UCF Syntax."

Starting Constraints Editor

Constraints Editor runs on PCs and workstations. You can start Constraints Editor:

- "From Project Navigator"
- "As a Standalone"
- "From the Command Line"

From Project Navigator

Within Project Navigator, launch Constraints Editor from the Processes window.

1. Select a design file in the Sources window.
2. Double-click *Create Timing Constraints* in the Processes window, which is located within User Constraints underneath Design Utilities.

As a Standalone

If you installed Constraints Editor as a standalone tool on your PC, either:

- Click the Constraints Editor icon on the Windows desktop, or
- Select **Start** > **Programs** > **Xilinx ISE** > **Accessories** > **Constraints Editor**

From the Command Line

Below are several ways to start Constraints Editor from the command line.

With No Data Loaded

To start Constraints Editor from the command line with no data loaded, type:

```
constraints_editor
```

With the NGD File Loaded

To start Constraints Editor from the command line with the NGD file loaded, type:

```
constraints_editor ngdfile_name
```

where

- *ngdfile_name* is the name of the NGD file

It is not necessary to use the .ngd extension.

If a UCF file with the same base name as the NGD file exists, it is loaded also. Otherwise, you are prompted for a UCF file.

With the NGD File and the UCF File Loaded

To start Constraints Editor from the command line with the NGD file and the UCF file loaded, type:

```
constraints_editor ngdfile_name -uc ucf_file_name
```

where

- *ngdfile_name* is the name of the NGD file
- *ucf_file_name* is the name of the UCF file

It is not necessary to use the .ucf extension.

As a Background Process

To run Constraints Editor as a background process on a workstation, enter:

```
constraints_editor &
```

UCF Syntax

This section describes the UCF syntax for constraints that are supported by Constraints Editor. For more information, see the Constraints Editor help. This chapter contains the following:

Group Elements Associated by Nets (TNM_NET)

Definition

A TNM_NET (timing name for nets) is an attribute that can be used to identify the elements that make up a group which can then be used in a timing specification. Essentially TNM_NET is equivalent to TNM on a net except for pad nets.

UCF Syntax

```
NET "netname" TNM_Net=identifier;
```

where

- *netname* is the name of a net
- *identifier* is a value that consists of any combination of letters, numbers, or underscores

Group Elements by Instance Name (TNM)

Definition

Identifies the instances that make up a group which can then be used in a timing specification. A TNM (pronounced tee-name) is a flag that you place directly on your schematic to tag a specific net, element pin, primitive or macro. All symbols tagged with the TNM identifier are considered a group.

UCF Syntax

```
INST "instance_name" TNM=identifier;
```

where

- *instance_name* can be FFs, All Pads, Input Pads, Output Pads, Bi-directional Pads, 3-stated Output Pads, RAMs, or Latches
- *identifier* is a value that consists of any combination of letters, numbers, or underscores

Keep *identifier* short for convenience and clarity.

Group Elements by Element Output Net Name Schematic Users (TIMEGRP)

Definition

Specifies a new group with instances of FFs, PADs, RAMs, LATCHES, or User Groups by output net name.

UCF Syntax

```
TIMEGRP identifier=element (output_netname);
```

where

- *identifier* is the name for the new time group
- *element* can be FFS, All Pads, Input Pads, Output Pads, Bi-directional Pads, 3-stated Output Pads, RAMs, LATCHES, or User Groups
- *output_netname* is the name of the net attached to the element

Timing THRU Points (TPTHRU)

Definition

Identifies an intermediate point on a path.

UCF Syntax

```
INST "instance_name" TPTHRU=identifier;
NET "netname" TPTHRU=identifier;
```

where

- *identifier* is a unique name

Pad to Setup

Definition

Specifies the timing relationship between an external clock and data at the pins of a device. Operates on pads or predefined groups of pads.

UCF Syntax

```
OFFSET=IN time unit BEFORE pad_clock_netname [TIMEGRP
"reg_group_name"];
[NET "pad_netname"] OFFSET=IN time unit BEFORE pad_clock_netname
[TIMEGRP "reg_group_name"];
[TIMEGRP "padgroup_name"] OFFSET=IN time unit BEFORE pad_clock_netname
[TIMEGRP "reg_group_name"];
```

where

- *padgroup_name* is the name of a group of pads predefined by the user
- *reg_group_name* is the name of a group of registers predefined by the user
- *pad_clock_netname* is the name of the clock at the port

For more information on Pad to Setup, see "Global Tab" in the Constraints Editor help.

Clock to Pad

Definition

Specifies the timing relationship between an external clock and data at the pins of a device. Operates on pads or predefined groups of pads.

UCF Syntax

```
OFFSET=OUT time unit AFTER pad_clock_netname [TIMEGRP
"reg_group_name"];
NET "pad_netname" OFFSET=OUT time unit AFTER pad_clock_netname [TIMEGRP
"reg_group_name"];
TIMEGRP "padgroup_name" OFFSET=OUT time unit AFTER pad_clock_netname
[TIMEGRP "reg_group_name"];
```

where

- *padgroup_name* is the name of a group of pads predefined by the user
- *reg_group_name* is the name of a group of registers predefined by the user
- *pad_clock_netname* is the name of the clock at the port

For more information on Clock to Pad, see "Global Tab" in the Constraints Editor help.

FROM TO

Definition

Establishes an explicit maximum acceptable time delay between groups of elements.

UCF Syntax

```
TIMESPEC "TSid"=FROM "source_group" TO "destination_group" time [unit];
```

where

- *source_group* and *destination_group* are FFS, RAMS, PADS, LATCHES, or user-created groups

FROM/THRU/TO

Definition

Establishes a maximum acceptable time delay between groups of elements relative to another timing specification.

UCF Syntax

```
TIMESPEC "TSid"=FROM "source_group" THRU "timing_point" TO
"destination_group" time [unit];
```

where

- *source_group* and *destination_group* are FFS, RAMS, PADS, LATCHES, or user-created groups
- *timing_point* is an intermediate point as specified by the TPTHRU constraint on the Advanced tab window

FROM TO TIG

Definition

Marks paths between a source group and a destination group that are to be ignored for timing purposes.

UCF Syntax

```
TIMESPEC "TSid"=FROM "source_group" TO "destination_group" TIG;
TIMESPEC "TSid"=FROM "source_group" THRU "timing_point(s)" TO
"destination_group" TIG;
```

where

- *source_group* and *destination_group* are FFS, RAMS, PADS, LATCHES, or user-created groups
- *timing_point* is an intermediate point as specified by the TPTHRU Points constraint on the Advanced tab window

Net TIG

Definition

Marks nets that are to be ignored for timing purposes.

UCF Syntax

```
NET "netname" TIG;
NET "netname" TIG="TSid1" ... "TSidn";
```

Period

Definition

Defines a clock period.

UCF Syntax

```
TIMESPEC "TSid"=PERIOD {timegroup_name time | TSid
[+/- phase [units]} [HIGH | LOW high_or_low_time unit];
```

where

- *id* is a unique identifier. The identifier can consist of letters, numbers, or the underscore character (_).
- *unit* is picoseconds, nanoseconds, microseconds, or milliseconds
- **HIGH** | **LOW** indicates the state of the first pulse of the clock
- **phase** is the amount of time that the clock edges are offset when describing the time requirement as a function of another clock
- *units* are in ms, us, ns, and ps

VOLTAGE

Definition

Specifies operating voltage and provides a means of prorating delay characteristics based on the specified voltage.

UCF Syntax

```
VOLTAGE=value[units];
```

where

- *value* is an integer or real number specifying the voltage in volts and *units* is an optional parameter specifying the unit of measure

TEMPERATURE

Definition

Allows the specification of the operating temperature which provides a means of prorating device delay characteristics based on the specified junction temperature. Prorating is a linear scaling operation on existing speed file delays and is applied globally to all delays.

UCF Syntax

```
TEMPERATURE=value [units];
```

where

- *value* is an integer or real number specifying the temperature in Celsius as the default. F and K are also accepted.

Project Navigator

To set implementation constraints in Project Navigator:

- For FPGA devices, the implementation process properties specify how a design is translated, mapped, placed, and routed. You can set multiple properties to control the implementation processes for the design.
- For CPLD devices, the implementation process properties specify how a design is translated and fit.

For more information, see the Project Navigator help for the Process Properties dialog box.

Floorplanner

The following sections explain how to set area and IOB constraints using Floorplanner.

For the constraints and devices with which Floorplanner can be used, see "Constraints Entry Methods" above. For information on running Floorplanner, see the ISE Help.

Using Area Constraints

Area constraints are a way of restricting where PAR can place a particular piece of logic. By reducing PAR's search area for placing logic, PAR's performance may be improved.

To create an area constraint in Floorplanner.

1. Select a hierarchical group in the Design Hierarchy window.
2. Select **Floorplan** > **Assign Area Constraint**.

Use the mouse to drag a rectangular box where you want to locate the area constraint.

The area constraint includes all the tiles inside the drag box.

Area constraints may overlap each other. Select **Floorplan** > **Bring Area To Front** or **Floorplan** > **Push Area To Back** to move a selected area constraint in front of or behind another.

Creating UCF Constraints from IOB Placement

You can also add constraints to the UCF file through Floorplanner and iteratively implement your design to achieve optimal placement.

To begin with, you need only the NGD file generated in a previous flow. In Floorplanner, you manually make IOB assignments which are automatically written into the UCF file. Floorplanner edits the UCF file by adding the newly created placement constraints. The placement constraints you create in Floorplanner take precedence over existing constraints in the UCF.

Next, go through the steps of implementing your design by running NGDBuild, MAP, and PAR.

Floorplan Editor

For the constraints and devices with which Floorplan Editor can be used, see "Constraints Entry Methods" above. For information on running Floorplan Editor, see the ISE Help.

PACE

You can set constraints in the Pinout & Area Constraints Editor (PACE). Within PACE, the Pin Assignments Editor is mainly used to assign location constraints to IOs. It is also used to assign IO properties such as IO Standards.

For the constraints and devices with which PACE can be used, see "Constraints Entry Methods" above. For more information about accessing and using PACE, see the ISE Help.

LOC Constraints

This section refers to LOC constraints for IOs (including Bank and Edge constraints) and global logic.

IOs

```
NET "name" LOC = "A23";
NET "name" LOC = "BANK0";
NET "name" LOC = "TL"; //half-edge constraint
NET "name" LOC = "T"; //edge constraint
```

Global Logic

```
INST "gt_name" LOC = GT_X0Y0;
INST "bram_name" LOC = RAMB16_X0Y0; (or RAMB4_C0R0)
INST "dcm_name" LOC = DCM_X0Y0;
INST "ppc_name" LOC = PPC405_X0Y0;
INST "mult_name" LOC = MULT18X18_X0Y0;
```

IOSTANDARD Constraints

```
NET "name" IOSTANDARD = "LVTTL";
```

PROHIBIT Constraints

```
CONFIG PROHIBIT = A23;
CONFIG PROHIBIT = SLICE_X1Y6;
CONFIG PROHIBIT = TBUF_X0Y0; (RAMs, MULTs, GTs, DCMs also)
```

AREA Constraints Editor

The AREA Constraints Editor is mainly used to assign areas to hierarchical blocks of logic. The following UCF examples show AREA_GROUP constraints that can be set in the AREA Constraints Editor.

```
INST "name" AREA_GROUP = group_name;
AREA_GROUP "group_name" RANGE=SLICE_X1Y1:SLICE_X5Y5;
AREA_GROUP "group_name" RANGE = SLICE_X6Y6:SLICE_X10Y10,
SLICE_X1Y1:SLICE_X4Y4;
AREA_GROUP "group_name" COMPRESSION = 0;
AREA_GROUP "group_name" ROUTE_AREA = FIXED;
```

Note: SLICE_ equals CLB_ for Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices.

Partial Design Pin Preassignment

Note: This section deals with Pin Preassignment when a design is partially completed. For information on Pin Preassignment in which an HDL template is built by adding constraints to pins that are defined within PACE, see the ISE Help.

Designs that are not yet fully coded might still have layout requirements. Pin assignments, voltage standards, banking rules, and other board requirements might be in place long before the design has reached the point where these constraints can be applied. The Pin Preassignment feature allows the pin-out rules of the design to be determined before the design logic has been completed.

To use the Pin Preassignment feature in PACE:

1. Provide the complete list of ports in your top-level design
2. Assign I/O constraints to them

Even if the ports are not used by any logic in the design (that is, no loads for input pins, no sources for output pins), they can still receive constraints and be kept through implementation.

Assign LOC or IOSTANDARD constraints in the UCF just like for any I/O pin. These requirements are annotated in the database. PACE can be used to assign pin locations, banking groups or voltage standards, and DRC checks can be run. The final PAD report contains any pins that have logic or constraints associated with them.

This implementation of the design is incomplete and cannot be downloaded to the hardware. You should expect these errors during the DRC phase of bitstream generation (BitGen):

- ERROR: PhysDesignRules:368 - The signal <D_OBUF> is incomplete. The signal is not driven by any source pin in the design.
- ERROR: PhysDesignRules:10 - The network <D_OBUF> is completely unrouted.

To trim any unused ports from the design, remove the associated constraints. The Translate (NGDBuild) phase trims these unused pins.

In this example, there are six top-level ports. Only three (clk, A, C) are currently used in the design. Of the remaining three ports:

- B is kept because it has a LOC constraint.
- D is kept because it has an IOSTANDARD constraint.
- E is trimmed because it is completely unused and unconstrained.

Verilog Example

```
--------------
module design_top(clk, A, B, C, D, E);
input clk, A, B;
output reg C, D, E;

always@(posedge clk)
C <= A;

endmodule
```

UCF Example

```
--------

NET "A" LOC = "E2" ;
NET "B" LOC = "E3" ;
NET "C" LOC = "B15" ;
NET "D" IOSTANDARD = SSTL2_II ;
```

FPGA Editor

You can add certain constraints to, or delete certain constraints from, the PCF file in the FPGA Editor. In the FPGA Editor, net, site, and component constraints are supported as property fields in the individual nets and components. Properties are set with the **Setattr** command, and are read with the **Getattr** command.

All Boolean constraints (BLOCK, LOCATE, LOCK, OFFSET IN, OFFSET OUT, and PROHIBIT) have values of On or Off; offset direction has a value of either In or Out; and offset order has a value of either Before or After. All other constraints have a numeric value. They can also be set to Off to delete the constraint. All values are case-insensitive (for example, “On” and “on” are both accepted).

When you create a constraint in the FPGA Editor, the constraint is written to the PCF file whenever you save your design. When you use the FPGA Editor to delete a constraint and then save your design file, the line on which the constraint appears in the PCF file remains

in the file but it is automatically commented out. Some of the constraints supported in the FPGA Editor are listed in the following table.

Table 3-3: **Constraints Supported in FPGA Editor**

Constraint	Accessed Through
block paths	Component Properties and Path Properties property sheet
define path	Viewed with Path Properties property sheet
location range	Component Properties Constraints page
locate macro	Macro Properties Constraints page
lock placement	Component Properties Constraints page
lock routing of this net	Net Properties Constraints page
lock routing	Net Properties Constraints page
maxdelay allnets	Main Properties Constraints page
maxdelay allpaths	Main Properties Constraints page
maxdelay net	Net Properties Constraints page
maxdelay path	Path Properties property sheet
maxskew	Main Properties Constraints page
maxskew net	Net Properties Constraints page
offset comp	Component Properties Offset page
penalize tilde	Main Properties Constraints page
period	Main Properties Constraints page
period net	Net Properties Constraints page
prioritize net	Net Properties Constraints page
prohibit site	Site Properties property sheet

Locked Nets and Components

If a net is locked, you cannot unroute any portion of the net, including the entire net, a net segment, a pin, or a wire. To unroute the net, you must first unlock it. You can add pins or routing to a locked net.

A net is displayed as locked in the FPGA Editor if the Lock Net [*net_name*] constraint is enabled in the PCF file. You can use the Net Properties property sheet to remove the lock constraint.

When a component is locked, one of the following constraints is set in the PCF file.

```
lock comp [comp_name]
locate comp [comp_name]
lock macro [macro_name]
lock placement
```

If a component is locked, you cannot unplace it, but you can unroute it. To unplace the component, you must first unlock it.

Interaction Between Constraints

Schematic constraints are placed at the beginning of the PCF file by MAP. The start and end of this section is indicated with **SCHEMATIC START** and **SCHEMATIC END**, respectively. Because of a "last-read" order, all constraints that you enter in this file should come after **SCHEMATIC END**.

You are not prohibited from entering a user constraint before the schematic constraints section, but if you do, a conflicting constraint in the schematic-based section may override your entry.

Every time a design is remapped, the schematic section of the PCF file is overwritten by the mapper. The user constraints section is left intact, but certain constraints may be invalid because of the new mapping.

Constraints Priority

In some cases, two timing specifications cover the same path. For cases where the two timing specifications on the path are mutually exclusive, the following constraint rules apply.

File Priorities

Priority depends on the file in which the constraint appears. A constraint in a file accessed later in the design flow replaces a constraint in a file accessed earlier in the design flow (Last One Wins) if the constraint name is the same in both files. If the two constraints have different names, the last one in the PCF file has priority.

Priority is as follows. The first listed is the highest priority, the last listed is the lowest.

- Constraints in a Physical Constraints File (PCF)
- Constraints in a User Constraints File (UCF)
- Constraints in a Netlist Constraints File (NCF)
- Attributes in a schematic

Timing Specification Priorities

If two timing specifications cover the same path, the priority is as follows. The first listed is the highest priority, the last listed is the lowest.

- Timing Ignore (TIG)
- FROM THRU TO
- FROM TO
- Specific OFFSET
- Group OFFSET
- Global OFFSET
- PERIOD

FROM THRU TO and FROM TO Statement Priorities

FROM THRU TO and FROM TO statements have a priority order that depends on the type of source and destination groups included in a statement. The priority is as follows (first listed is the highest priority, last listed is the lowest).

- Both the source group and the destination group are user-defined groups
- Either the source group or the destination group is a predefined group
- Both the source group and the destination group are predefined groups

OFFSET constraints take precedence over more global constraints.

OFFSET Priorities

If two specific OFFSET constraints at the same level of precedence interact, an OFFSET with a register qualifier takes precedence over an OFFSET without a qualifier; if otherwise equivalent, the latter in the constraint file takes precedence.

Net Delay and Net Skew Priorities

Net delay and net skew specifications are analyzed independently of path delay analysis and do not interfere with one another.

Constraints Priority Exceptions

There are circumstances in which constraints priority may not operate as expected. These cases include supersets, subsets, and intersecting sets of constraints. See the following diagram.

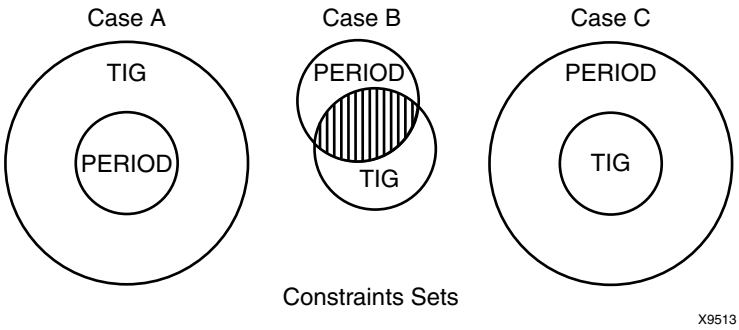

Figure 3-3: **Interaction Between Constraints Sets**

- In Case A, the TIG superset conflicts with the PERIOD set.
- In Case B, the intersection of the PERIOD and TIG sets creates an ambiguous circumstance. In this instance, constraints may sometimes be considered as part of TIG, and at other times part of PERIOD.
- In Case C, the TIG subset works normally within the PERIOD superset.

Chapter 4

Timing Constraint Strategies

This chapter provides general guidelines that explain how to constrain the timing on designs when using the implementation tools for FPGA devices. This chapter contains the following sections:

- "Basic Implementation Tools Constraining Methodology"
- "Global Timing Assignments"
- "Specific Timing Assignments"
- "Multi-Cycle and Fast or Slow Timing Assignments"
- "Special Case Path Constraining"
- "Path Coverage Statistics"
- "Static Timing Analysis"
- "Synchronous Timing"
- "Directed Routing"

For more information about timing constraints and strategies:

1. Go to the Xilinx® home page.
2. Click Support.
3. Click Tech Tips.
4. Click Timing & Constraints.

Basic Implementation Tools Constraining Methodology

Creating global constraints for a design is the easiest way to provide coverage of the constrainable connections in a design, and to guide the tools to meet timing requirements for all paths. The global constraints constrain the whole design. If there are multi-cycle or static paths, you can constrain them using more specific constraints. A multi-cycle path is a path between two registers with a timing requirement that is a multiple of the clock period for the registers. A static path does not include clocked elements, for example, pad-to-pad paths.

Xilinx recommends that you specify the exact value required for a path, as opposed to over-tightening a specification. Specifying tighter constraints than required is not recommended. Tighter constraints can lengthen PAR runtimes and cause degradation in the quality of results.

The Constraints Editor is based on the methodology discussed in this chapter. The group names and TSids in the examples show how the Constraints Editor populates the grids and creates new groups and constraints. The Constraints Editor provides additional help. The clocks and IOs are supplied, so you need not know the exact spelling of the names. You

only need to define the timing, and not the syntax, of the constraints. For more specific grouping, element names are provided, and exceptions to the global constraints can be made using those groups.

The first tab of the Constraints Editor shows all the global paths that need to be covered. If this tab is completed, all synchronous paths are covered.

All examples in this chapter show the UCF syntax.

Global Timing Assignments

Global timing assignments are overall constraints that cover all constrainable paths in a design. These assignments include:

- Clock definitions
- Input and output timing requirements
- Combinatorial path requirements

Following are some recommendations for assigning definitions.

Assigning Definitions for Clocks Driven by Pads

Define each clock in the design. Defining each clock covers all synchronous paths within each clock domain and paths that cross between related clock domains. Use a TNM_NET on each clock net (on the net attached to the pad, usually the port name in HDL,) and then use the TIMESPEC PERIOD syntax with the TNM_NET group created. Using the TIMESPEC version of the PERIOD definition allows for greater path control later on when constraining paths between clock domains.

For more information if you are using a Virtex™ DLL/DCM, see "Assigning Definitions for DLL/DCM/PLL Clocks" in this chapter.

Related Clocks Example

The following example design has two clocks. TNM_NETs identify the synchronous elements of each clock domain. TIMESPEC PERIOD gives the flexibility to describe inter clock domain path requirements. The clock "clock2_in" has twice the period of "clock1_in," which is shown in the following UCF example with the clock2_in PERIOD definition using a function of the "TS_clock1_in" specification ("TS_clock1_in" * 2).

```
NET "clock1_in" TNM_NET =  "clock1_in";
TIMESPEC "TS_clock1_in" = PERIOD "clock1_in" 20 ns HIGH 10;
NET "clock2_in" TNM_NET = "clock2_in";
TIMESPEC "TS_clock2_in" = PERIOD "clock2_in" "TS_clock1_in" * 2;
```

The Constraints Editor uses the clock pad net name for the group name and the TSid as show in the previous example. This feature is important if you want to override a constraint that was entered in the source.

PHASE Related Clocks Example

The following example shows how to specify two clocks related by a phase difference. The clock "clock" has a period of 10ns. The clock "clock_90" is also 10 ns, but is shifted 90 degrees out of phase, or is lagging "clock's" rising edge by 2.5 ns.

Use the keyword PHASE to identify this relationship. The timing tools use this information in OFFSET and cross-clock domain paths. See the following example.

```
NET "clock" TNM_NET = "clock";
TIMESPEC "TS_clock" = PERIOD "clock" 10 ns HIGH 50%;
NET "clock_90" TNM_NET = "clock_90";
TIMESPEC "TS_clock_90" = PERIOD "clock_90" "TS_clock" * 1 PHASE + 2.5ns;
```

Assigning Definitions for DLL/DCM/PLL Clocks

TRANSLATION (NGDBuild) propagates TNM_NET tags through DLLs, DCMs and PLLs. NGDBuild creates new TNM_NETs for each of the DLL, DCM and PLL output taps and associated PERIOD statements. The code takes into account the phase relationship factor of the outputs for the DLL, and also performs the appropriate multiplication or division of the PERIOD value.

The code also takes into account any of the PHASE taps adjustments. This means that for OFFSETs and cross-clock domain paths, the timing tools now know the relationship for PHASE shifts also.

DCM PERIOD Propagation Example

In this example, you only need to define the input clock to the DCM. The tools generate all of the correct PERIODs for the output taps. Assume that the input clock (net "clock_in" with PERIOD 30 ns) DCM in this example uses the CLK0 (net "clock0") and CLK2X180 (net "clock2x180") output taps. When you define the input clock, the system performs all of the transformations.

For input clock “clock_in”:

```
NET "clock_in" TNM_NET = "clock_in";
TIMESPEC "TS_clock_in" = PERIOD "clock_in" 30 ns HIGH 50%;
```

Generated clock definitions:

```
NET "clock0" TNM_NET = "clock0";
TIMESPEC "TS_clock0" = PERIOD "TS_clock_in" * 1;
NET "clock2x180" TNM_NET = "clock2x180";
TIMESPEC "TS_clock2x180" = PERIOD "TS_clock_in" / 2 PHASE + 7.50 ns;
```

Assigning Definitions for Derived and Gated Clocks

For clocks that are created in the FPGA, such as the output of a register or a gated clock (the output of combinatorial logic), the net name from the output of the register or gate should be the name used for the TNM_NET group name and TS*id*. For more information, see “OFFSETs with Derived or Gated Clocks” in this chapter.

Assigning Input and Output Requirements

Constrain input and output timing requirements using the OFFSET constraints. Pad to Setup requirements use OFFSET IN BEFORE and for Clock to Out requirements use OFFSET OUT AFTER.

You can specify OFFSETs in three levels of coverage.

- The first, global OFFSET applies to all inputs or outputs for a specific clock.
- The second, a group OFFSET form, identifies a group of inputs or outputs clocked by a common clock that have the same timing requirements.
- The third, a specific OFFSET form, specifies the timing by each input or output.

OFFSET constraints of a more specific scope override a more general scope.

A group OFFSET overrides a global OFFSET specified for the same IOs. A specific OFFSET overrides both global and group OFFSETs if used. This priority rule allows you to start with global OFFSETs, then create group or specific OFFSETs for IOs with special timing requirements.

For memory usage and runtime considerations, use global and group OFFSETs and avoid specific OFFSETs whenever possible. Using wildcards in the specific OFFSET form creates multiple specific OFFSET constraints, not a group OFFSET.

Example:

```
NET bob* OFFSET = IN 5 AFTER clock;
```

Global Inputs Requirements

Use OFFSET IN BEFORE to define Pad to Setup timing requirements. OFFSET IN BEFORE is an external clock-to-data relationship specification and takes into account the clock delay, clock edge and DLL/DCM/PLL introduced clock phase when analyzing the setup requirements (data delay + setup - clock delay-clock arrival).). Clock arrival takes into account any clock phase generated by the DLL/DCM/PLL or clock edge. This strategy constrains all of the inputs clocked by the same clock to identical requirements.

Following is a global OFFSET IN BEFORE example:

```
OFFSET = IN value units BEFORE clock_pad_net;
OFFSET = IN 10 ns BEFORE "clock_in";
```

where

- *value* is the time allowed for the data to propagate from the pad to meet a setup requirement to the clock. This value is in relationship to the clocks initial edge at the pin of the chip. (The PERIOD constraint defines the clock initial edge.)
- *units* is ms, micro, ns (default) or ps
- *clock_pad_net* is the name of the clock using the net name attached to the pad (This or the port name for HDL designs)

Global Outputs Requirements

Use OFFSET OUT AFTER to define Clock to Pad timing requirements. OFFSET OUT AFTER is an external clock-to-data specification and takes into account the clock delay, clock edge and DLL/DCM/PLL introduced clock phase when analyzing the setup requirements (clock delay + clock to out + data delay +clock arrival). Clock arrival takes into account any clock phase generated by the DLL/DCM/PLL or clock edge. This strategy constrains all of the outputs clocked by the same clock to the same requirement.

The following is a global OFFSET OUT AFTER example:

```
OFFSET = OUT value units AFTER clock_pad_net;
OFFSET = OUT 10 ns AFTER "clock_in";
```

where

- *value* is the time allowed for the data to propagate from the synchronous element (clock to out, T_{CKO}) to the pad. This value is in relationship to the clocks initial edge at the pin of the chip. (The PERIOD constraint defines the clock initial edge.)
- *units* is ms, micro, ns (default) or ps
- *clock_pad_net* is the name of the clock using the net name attached to the pad or the port name for HDL designs

Assigning Global Pad to Pad Requirements

Use a FROM PADS TO PADS constraint to globally constrain all combinatorial pin-to-pin paths. If you do not have any combinatorial pin-to-pin paths, ignore this constraint.

Following a global pad to pad example:

```
TIMESPEC "TSid" = FROM "PADS" TO "PADS" value units;
TIMESPEC "TS_P2P" = FROM "PADS" TO "PADS" 10 ns;
```

where

- *id* is a user-specified unique identifier for the constraint
- *value* is the time allowed for the data to propagate from an input pad to an output pad
- *units* is ms, micro, ns (default) or ps

Specific Timing Assignments

If there are paths that are static in nature, you can use TIG to eliminate the paths from timing consideration in Place and Route (PAR) and TRCE. If there are paths that require faster or slower specifications than the global requirements, you can create fast or slow exceptions for those paths. If multi-cycle paths exist, identify and constrain them.

The TIG paths still show the longest delay for that constraint in the verbose timing report. Net TIGs can be turned off in the Timing Analyzer to see the actual timing on these nets.

You can specify false paths (paths to ignore) in two different ways: by nets and elements or by timing paths. Identifying false paths allows PAR to concentrate on more critical paths when placing components and when using routing resources. There might be less runtime because PAR does not need to meet a specific timing requirement. Creating a large number of path TIGs can increase memory usage and possibly increase runtime due to the extra paths models that are created.

These paths are ignored by both PAR and timing analysis and do not show up in the timing report. Also these paths are not included in the Connection Coverage statistic. For more information, see "Ignored Paths (TIG)" in this chapter.

False Paths by Net

You can define false paths for *all* paths that pass through a particular net using the following UCF syntax:

```
NET "net_name" TIG;
```

You can also define false paths for a specified set of paths that pass through a particular net using the following UCF syntax:

```
NET "net_name" TIG = TSid_list;
```

where

- *net_name* is the name of the net that the paths are passing through
- TS*id_list* is a comma-delimited list of TIMESPEC identifiers to which the TIG applies

False Paths by Instance

You can define false paths for *all* paths that pass through a particular instance using the following UCF syntax:

```
INST "inst_name" TIG;
```

You can also define false paths for a specified set of paths that pass through a particular instance using the following UCF syntax:

```
INST "inst_name" TIG = TSid_list;
```

where

- *inst_name* is the name of the instance that the paths are passing through
- TS*id_list* is a comma-delimited list of TIMESPEC identifiers to which the TIG should apply

False Paths by Pin

You can define false paths for *all* paths that pass through a particular instance pin using the following UCF syntax:

```
PIN "instance.pin_name" TIG;
```

You can also define false paths for a specified set of paths that pass through a particular instance pin using the following UCF syntax:

```
PIN "instance.pin_name" TIG = TSid_list;
```

where

- *instance.pin_name* is the name of the instance and the pin identifier separated by a period that the paths are passing through
- TS*id_list* is a comma-delimited list of TIMESPEC identifiers to which the TIG should apply

False Paths by Timing Path

You can create groups, use the FROM TO, FROM THRU TO, or open FROM or TO constraints, and then specify TIG as the path value. For more information on syntax usage, see "False Paths by Path"in this chapter. These paths show up in a timing analysis report,

but the timing is not considered. These paths are also included in the connection coverage statistics.

FROM TO TIG

Following is a FROM TO TIG example:

```
TIMESPEC "TSid" = FROM "from_grp" TO "to_grp" TIG;
```

where

- *id* is a user-specified unique identifier for the constraint
- *from_grp* and *to_grp* are TIMEGRPs

FROM THRU TO TIG

Following is a FROM THRU TO TIG example:

```
TIMESPEC "TSid" = FROM "from_grp" THRU "thru_pt" TO "to_grp" TIG;
```

where

- *id* is a user-specified unique identifier for the constraint
- *from_grp* and *to_grp* are TIMEGRPs
- *thru_pt* is a net, instance or pin

For more information on defining TPTHRU points, see "TPTHRU" in this chapter.

Asynchronous Set/Reset Paths

The tools do not automatically analyze asynchronous set/reset paths. Automatic analysis is controlled by the path tracing controls. For more information, see the "Disable (DISABLE)" and "Enable (ENABLE)" constraints.

Multi-Cycle and Fast or Slow Timing Assignments

These path assignments include multi-cycle paths and fast or slow exceptions. First create timing groups to define start point and end points for the paths. These groups are used in the FROM TO timing constraints to override the PERIOD constraints for these specific paths. The following sections describe different exception types.

Cross-Clock Domain Constraining

The timing tools no longer include domain paths in the destination register clock domain if the clocks are not defined as related. Related clock domains are defined in the system as a function of other clock TIMESPECs. The TRANSLATE (NGDBuild) phase automatically relates clocks from the outputs of a DLL/DCM. If the paths between two "related" clocks are false, or if they require a different time requirement than calculated, create a FROM:TO constraint with a TIG or the correct value.

If the clocks are unrelated but have valid paths between them, create FROM TO constraints to constrain them. To constrain paths between two clocks and use the groups created by each clock domain, create a FROM TO for each direction that paths pass between the two clock domains, then specify the time requirement according to the path requirement. For information about how the groups were created, see "Related Clocks Example" in this chapter.

Following is a cross-clock domain TIMESPEC example:

```
TIMESPEC "TS_clock1_in_2_clock2_in" = FROM "clock1_in" TO "clock2_in"
10 ns;
```

User Group Creation

You can create groups to identify path end points. There are three basic methods for creating groups:

- "Identifying Groups by Connectivity"
- "Identifying Groups by Hierarchy"
- "Identifying Groups by Element"

The types of elements that can be grouped are:

- FFS
- PADS
- RAMS
- BRAMS_PORTA
- BRAMS_PORTB
- CPUS
- MULTS
- HSIOS
- LATCHES

These are considered reserved keywords that define the types of synchronous elements in FPGA devices and pads.

Identifying Groups by Connectivity

Identifying groups by connectivity allows you to group elements by specifying nets that eventually drive synchronous elements and pads. This method is a good way to identify multi-cycle paths elements that are controlled by a clock enable. This method uses TNM_NET on a net.

The TNM_NET syntax for identifying groups by connectivity is:

```
NET "net_name" TNM_NET = qualifier "tnm_name";
```

where

- *net_name* is the name of a net propagated by the tools to the element ends
- *tnm_name* is the user-assigned name for the group created by the TNM_NET. Multiple nets can be assigned the same *tnm_name*
- An optional *qualifier* of FFS, PADS, RAMS, BRAMS_PORTA, BRAMS_PORTB, CPUS, MULTS, HSIOS or LATCHES may be used when the *net_name* contains wildcards

Identifying Groups by Hierarchy

Identifying groups by hierarchy allows you to group by traversing the hierarchy of a module and tagging all predefined elements with the TNM. This method uses a TNM on a block.

The TNM syntax for identifying groups by hierarchy is:

```
INST "inst_name" TNM = qualifier "tnm_name";
```

where

- *inst_name* is the hierarchical name of a macro or module to be traversed by the tools to identify underlying elements for the group labeled by the *tnm_name* label
- An optional *qualifier* of FFS, PADS, RAMS, BRAMS_PORTA, BRAMS_PORTB, CPUS, MULTS, HSIOS or LATCHES may be used

Identifying Groups by Element

You can identify groups by element in the following ways:

- "Identifying Specific Elements by Instance Name"
- "Identifying Elements for Groups Using Element Output Net Names"

Identifying Specific Elements by Instance Name

Identifying elements directly allows you to group by tagging predefined elements with a TNM. Multiple instances can be given the same *tnm_name*.

The TNM syntax for identifying groups by instance is:

```
INST "inst_name" TNM = qualifier "tnm_name";
```

where

- *inst_name* is the predefined instance name for the group labeled by the *tnm_name* label
- An optional *qualifier* of FFS, PADS, RAMS, BRAMS_PORTA, BRAMS_PORTB, CPUS, MULTS, HSIOS or LATCHES may be used when the *inst_name* contains wildcards

Identifying Elements for Groups Using Element Output Net Names

This method is mainly used by schematic users who generally name nets, not instances. Identifying elements individually is used for singling out elements or identifying elements by output net name. This method uses TIMEGRP and allows the use of wildcards (*, ?) for filtering elements. This method is best used for schematics where the instance names are rarely known but the output nets generally are.

The TIMEGRP syntax for identifying groups by element output net name is:

```
TIMEGRP "tgrp_name" = qualifier (output_net_name);
```

where

- *tgrp_name* is the name assigned by you to the group
- *qualifier* is a (FFS, PADS, RAMS, BRAMS_PORTA, BRAMS_PORTB, CPUS, MULTS, HSIOS, LATCHES) keyword
- *output_net_name* is the output net name for each element that you would like to group. You can use wildcards with *output_net_name*

Specific OFFSET Constraints Using PAD and or Register Groups

You can use grouping with OFFSET. Grouping includes both register groups and pad groups. Grouping allows you to group pads to set the same path delay requirements and group registers for identifying paths that have different requirements from or to single pads. You can group and constrain the single pads and registers all at once. This is useful

if a clock is used on the rising and falling edge for inputs or outputs. These two groups require different constraints.

Group OFFSET IN Example

```
TIMEGRP "pad_group" OFFSET = IN time units BEFORE "clock_pad_net"
TIMEGRP "register_group";
```

where

- *pad_group* is the user- created group of input pads
- *time* is the time allowed for the data to propagate from the pad to meet a setup requirement to the clock. This value is in relationship to the clocks initial edge at the pin of the chip. (The PERIOD constraint defines the clock initial edge.)
- *units* is ms, micro, ns (default) or ps
- *clock_pad_net* is the name of the clock using the net name attached to the pad
- *register_group* is the user-created group of synchronous elements

Group OFFSET OUT Example

```
TIMEGRP "pad_group" OFFSET = OUT time units AFTER "clock_pad_net"
TIMEGRP "register_group";
```

where

- *pad_group* is the user- created group of output pads
- *time* is the time allowed for the data to propagate from the pad to meet a setup requirement to the clock. This value is in relationship to the clocks initial edge at the pin of the chip. (The PERIOD constraint defines the clock initial edge.)
- *units* is ms, micro, ns (default) or ps
- *clock_pad_net* is the name of the clock using the net name attached to the pad
- *register_group* is the user-created group of synchronous elements

FROM TO Syntax

This group includes FROM, TO, and FROM TO. FROM specifies the source group, and TO specifies the destination group. Using just a FROM assumes all destinations are TO points and using just a TO assumes all sources are FROM points.

The FROM TO syntax is used in the following path assignments, and is defined as follows in the UCF:

```
TIMESPEC "TSid" = FROM "from_grp" TO "to_grp" value units;
```

where

- *id* is a user-specified unique identifier for the constraint
- *from_grp* and *to_grp* are TIMEGRPs
- *value* is a specific time, a (*,?) function of another TS*id* (that is, TS_01 *2), or TIG. The allowable operations are: "*" (multiply) and "/" (divide).
- *units* is ms, micro, ns (default) or ps

Open FROM to TO Example

```
TIMESPEC "TSid" = FROM "from_grp" value units;
```

where

- *id* is a user specified unique identifier for the constraint
- *from_grp* is TIMEGRP
- *value* is the time requirement
- *units* is ms, micro, ns (default) or ps

FROM THRU TO Syntax

To further narrow down paths, use TPTHRU and FROM THRU TO. You can also specify multiple THRUs. For more information, see "TPTHRU" in this chapter. FROM or TO are optional.

Multi-Cycle Paths Assignments

To specify multi-cycle path assignments:

1. Identify the start point and end point groups.
2. Apply a FROM TO constraint for that path.

For elements controlled by clock enables, use a TNM_NET on the clock enable to identify all of the elements. You can specify timing requirements as a function of the clock.

The units you use on the originating TSid affect the speed of the new clock specification:

- "MHz", "*" used as multiplication makes the new clock specification faster.
- "ns", "*" used as multiplication makes the new clock specification slower.

```
TIMESPEC "TSid" = FROM "from_grp" TO "to_grp" TS_01*2;
```

Slow or Fast Exception Paths

To specify slow or fast path assignments:

1. Identify the start point and end point groups.
2. Apply a FROM TO constraint with a specific value for that path.

```
TIMESPEC "TSid" = FROM "from_grp" TO "to_grp" value units;
```

False Paths by Path

Create groups, specify the FROM TO constraint, and then use TIG as the path value.

```
TIMESPEC "TSid" = FROM "from_grp" TO "to_grp" TIG;
```

Special Case Path Constraining

Special case path constraining allows you to further refine path specifications, or define asynchronous points as a path endpoint. TPTHRU allows the further refinement of a FROM TO path. With TPSYNC, you can specify an asynchronous point as a path start or end point.

TPTHRU

TPTHRU narrows the paths constrained by a FROM TO constraint. It specifies nets or instances that the paths must pass through. You can specify multiple TPTHRU points for a set of paths.

TPTHRU Syntax

This section discusses TPTHRU syntax.

Forms

There are three forms of the TPTHRU syntax. These forms identify:

- THRU points that pass through nets
- THRU points through instances
- THRU points of specific instance pins

Be careful when placing TPTHRU points. They can become subsumed into components, and may not resolve uniquely. You may need to use the KEEP attribute on the net to preserve the TPTHRU tag.

NET Form (UCF)

```
NET "net_name" TPTHRU = "thru_name";
```

where

- *net_name* is the name of the net the paths pass through
- *thru_name* is the user name for the THRU point

INSTANCE Form (UCF)

```
INST "inst_name" TPTHRU = "thru_name";
```

where

- *inst_name* is the name of the instance the paths pass through
- *thru_name* is the user name for the THRU point

Pin Form (UCF)

```
PIN "instance.pin_name" TPTHRU = "thru_name";
```

where

- *instance.pin_name* is the name of the specific instance pin the paths pass through
- *thru_name* is the user name for the THRU point

FROM THRU TO Syntax (UCF)

```
TIMESPEC "TSid" = FROM "from_grp" THRU "thru_point" TO "to_grp" value
units;
```

where

- *id* is a user specified unique identifier for the constraint
- *from_grp* and *to_grp* are TIMEGRPs

- *thru_point* is specified by the TPTHRU tag
- *value* is a number or a (*,/) function of another TSid (that is, TS_01 *2) or a TIG
- *units* is (ms, micro, ns (default) or ps)

You can specify multiple sequential THRU points for any FROM TO specification.

TPSYNC

TPSYNC identifies asynchronous points in the design as endpoints for paths. You may want to use TPSYNC when specifying timing to a non-synchronous point in a path, such as a TBUF or to black box macro pins. You can identify non-synchronous elements or pins as a group, and then use either FROM or TO points.

TPSYNC Syntax

```
INST "inst_name" TPSYNC = "tpsync_name";
PIN "inst_name.pin_name" TPSYNC = "tpsync_name";
```

where

- *tpsync_name* represents the user label for the group that is created by the TPSYNC statement
- *pin_name* must match the name used in the HDL code or from the library

Output Slew Rate Constraint

You can use a slew rate of FAST in architectures that support this feature. Outputs are defined as SLOW by default. You can speed up timing by using the FAST property, but this may cause ringing or noise problems.

Following is the slew rate syntax:

```
INST "pad_inst_name" FAST;
NET "pad_net_name" FAST;
```

where

- *pad_inst_name* is the name of the pad instance
- *pad_net_name* is the name of the pad net. (The port name in HDL code.)

Path Coverage Statistics

A connection is a driver/load pin combination, which is connected by a signal. There are situations where connections are not valid, or do not show up in the coverage statistic.

Ignored Paths (TIG)

The most common reason for connection coverage not reaching 100% is that elements in the design have NET TIGs. If the timing tool encounters a TIG'd element when tracing a path, the trace stops there, possibly leaving connections on the "other side" of the element uncovered. On the other hand, a FROM TO TIG on a path has all of its connections accounted for in the coverage statistic, since those paths are enumerated in the timing report.

STARTUP Paths

There are other reasons for less than 100% coverage. One is that the total number of connections in a design includes some which cannot be covered by constraints. An example is the connections on the STARTUP component.

Static Paths

A static pin can drive a LUT which combines with no other signals and then drives other logic. This can happen at the start of a carry chain where a FORCE mode is used from a logic 1 or 0.

In addition, if terms for carry logic are connected to a CLB, but are not used within the CLB, these connections are never traced. These are just obscure cases that are not handled.

Certain categories of paths are turned off using path tracing controls. Paths that are turned off due to path tracing controls are not covered. For more information, see the "Enable (ENABLE)" constraint.

OFFSETs with Derived or Gated Clocks

If the clock that clocks a synchronous element does not come through an input pad -- for example, it is derived from another clock -- then OFFSET does not return any paths. Use FROM TOs for these paths, taking into account the clock delay.

Following is an example for pad to setup:

If the global clock delay is 1 ns, and the Pad to Setup requirement is 30 ns, then identify the PADs and registers that are clocked by a derived or gated clock, and group them accordingly.

Then create a timing constraint similar to the following:

```
TIMESPEC "TS_P2S_halfclock" = FROM "halfclock_pads" TO "halfclock_ffs"
31 ns;
```

Static Timing Analysis

You can perform timing analysis at several stages in the implementation flow to show your design delays. You create or generate the following:

- A post-map timing report to evaluate the effects of logic delays on timing constraints
- A post-place-and-route timing report that incorporates both block and routing delays as a final analysis of the design's timing constraints

The Interactive Timing Analyzer tool produces detailed timing constraint, clock, and path analysis for post-map or post-place-and-route implementations.

Static Timing Analysis After Map

Post-map timing reports can be very useful in evaluating timing performance. Although route delays are not accounted for, the logic delays can provide valuable information about the design.

If logic delays account for a significant portion (> 50%) of the total allowable delay of a path, the path may not be able to meet your timing requirements when routing delays are added.

Routing Delays

Routing delays typically account for 45% to 65% of the total path delays. By identifying problem paths, you can mitigate potential problems before investing time in place and route. You can:

- Redesign the logic paths to use fewer levels of logic
- Tag the paths for specialized routing resources
- Move to a faster device
- Allocate more time for the path

Logic-Only Delays

If logic-only delays account for much less (<35%) than the total allowable delay for a path or timing constraint, the place-and-route software can use very low placement effort levels. In these cases, reducing effort levels allows you to decrease runtimes while still meeting performance requirements.

Static Timing Analysis After PAR

Post-PAR timing reports incorporate all delays to provide a comprehensive timing summary. If a placed and routed design has met all of your timing constraints, you can proceed by creating configuration data and downloading a device.

If you identify problems in the timing reports, you can:

- Increase the placer effort level
- Use re-entrant routing
- Use multi-pass place and route

You can also:

- Redesign the logic paths to use fewer levels of logic
- Tag the paths for specialized routing resources
- Move to a faster device
- Allocate more time for the paths

Detailed Timing Analysis

To perform detailed timing analysis:

1. Open Project Navigator.
2. Select your project in the Sources window.
3. Double click Timing Analyzer under Launch Tools in the Processes window.

This allows you to:

- Specify specific paths for analysis
- Discover paths not affected by timing constraints
- Analyze the timing performance of the implementation based on another speed grade

For more information, see the Timing Analyzer help.

Synchronous Timing

Xilinx supports system synchronous and source synchronous timing. This section describes both types of timing.

This section also describes the following keywords:

- INPUT_JITTER
- SYSTEM_JITTER

Table 4-1: **Keyword Usage with Synchronous Timing**

Keyword	Can be used with system synchronous timing	Can be used with source synchronous timing
INPUT_JITTER	Yes	Yes
SYSTEM_JITTER	Yes	Yes

Note: For a syntax example of INPUT_JITTER and SYSTEM_JITTER, see "Syntax Examples," page 74.

System Synchronous Timing

In system synchronous timing, one clock source controls the data transmission and reception of all devices. See the following figure.

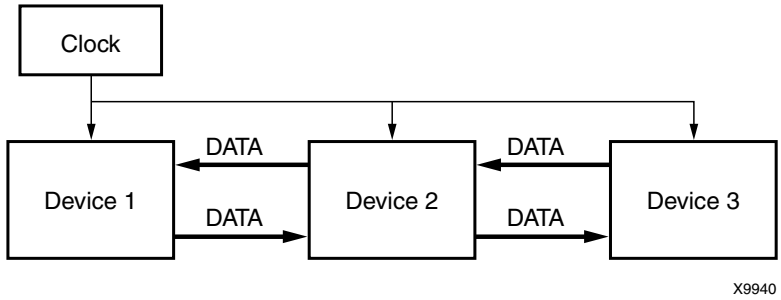

Figure 4-1: **System Synchronous Timing**

Source Synchronous Timing

The section describes how to use SYSTEM_JITTER, and INPUT_JITTER for source synchronous timing.

In the following example of source synchronous timing, one clock source controls the data transmission of devices. The derived clocks control data reception. See the following figure.

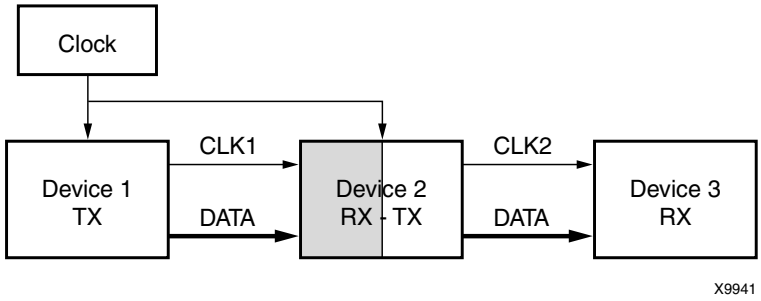

Figure 4-2: **Example of Source Synchronous Timing**

In the preceding example, CLK1 and CLK2 are derived clocks that control data reception of Device 2 and Device 3. The primary clock controls the data transmission for all three devices.

You can use source synchronous timing constraints for Double Data Rate (DDR) or Single Data Rate (SDR) inputs or outputs. The following figure shows an example of a timing diagram for Dual Data Rate inputs for two flip-flops, one with an active High input, and one with an active Low input.

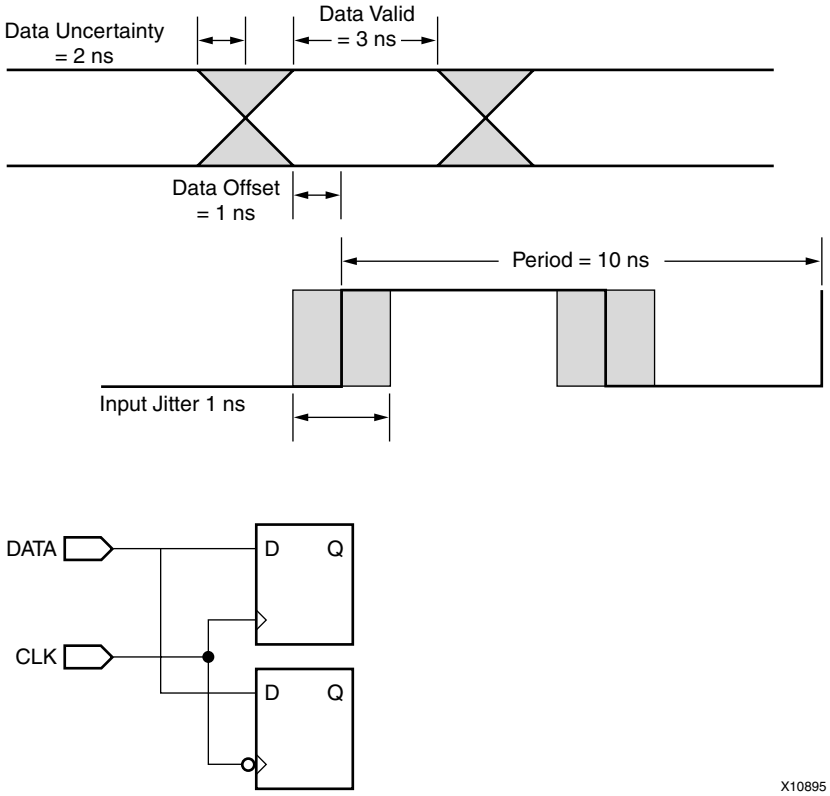

Figure 4-3: **Example of Timing Diagram for Dual Data Rate Inputs**

Syntax Examples

The following steps show an example of how to use the PERIOD, OFFSET, and SYSTEM_JITTER constraints for source synchronous timing for the example circuit.

1. Create Period

```
NET CLK TNM_NET = CLK_GRP;
TIMESPEC "TS_CLK" = PERIOD ”CLK_GRP" 10 ns INPUT_JITTER 1;
```

2. Create Groups

```
INST DATA_IN[*] TNM = DATA_IN;
TIMEGRP FF_RISING = RISING CLK_GRP ;
TIMEGRP FF_FALLING = FALLING CLK_GRP;
```

3. Create OFFSET constraint

```
TIMEGRP DATA_IN OFFSET = IN 1 VALID 3 BEFORE CLK TIMEGRP FF_RISING;
TIMEGRP DATA_IN OFFSET = IN -4 VALID 3 BEFORE CLK  TIMEGRP FF_FALLING;
```

4. Create SYSTEM_JITTER constraint.

```
SYSTEM_JITTER=0.456 ns;
```

Directed Routing

Directed Routing is a means of supporting repeatable, locked routing functionality similar to “exact guide” for a limited number of critical signals in a design via UCF constraints. Directed Routing is also used on signals with a limited fanout between comps in close proximity to one another, thereby avoiding the use of long-line resources.

Avoiding long-line resources in Directed Routing constraints is important for two reasons:

- Using such an “expensive” routing resource for a low fanout net is generally a bad practice.
- Using long-line resources reduces routing flexibility as the design changes and grows in the design process.

You set the value of the constraint in the FPGA Editor for Directed Routing.

About Directed Routing

Directed Routing is:

- A mechanism of locking the routing in order to maintain timing of nets in a design
- A potential work around for routing limitations
- A means of controlling route delays to a tighter tolerance than is possible via timing constraints

How Directed Routing Works

A constraint describing the exact routing resources used to route between source COMP pins and load COMP pins for the selected NET is created.

COMP placement constraints are required to maintain the relative positioning between all COMPs attached to the NET. For SLICE COMPs, BEL constraints are also required.

When To Use Directed Routing

Use Directed Routing when:

- Timing must be maintained (less than 200 ps variation) between implementations on a few nets
- Both the source and destination comps can be an (R)LOC and BEL constrained to maintain relative placement
- Skew must be controlled between nets
- Creating a high-speed macro to limit timing variation between instances of the MACRO
- Creating a high-speed macro to use in other devices of the same device family

When NOT To Use Directed Routing

Do not use Directed Routing when:

- Creating a MACRO that uses global resources, and which will be relocated in the device or other devices in the same device family.
- Routing for hundreds of nets between COMPs must be maintained. Directed Routing is NOT a replacement for Guide.

Related Constraints

- "BEL"
- "Location (LOC)"
- "Relative Location (RLOC)"
- "Relative Location Origin (RLOC_ORIGIN)"
- "U_SET"

Chapter 5

Xilinx Constraints

This chapter describes the individual constraints that can be used with Xilinx® FPGA and CPLD devices, including, for each constraint, architecture support, applicable elements, description, propagation rules, syntax examples, and, where necessary, additional information for particular constraints. This chapter contains the following sections:

Constraint Information

This chapter gives the following information for each constraint:

- Architecture Support

  A device table shows whether the constraint may be used with that device.
- Applicable Elements

  The elements to which the constraint may be applied.
- Description

  A brief description of the constraint, including its usage and behavior.
- Propagation Rules

  How the constraint is propagated.
- Syntax Examples

  Syntax examples for using the constraint with particular tools or methods. Not every tool or method is listed for every constraint. If a tool or method is not listed, the constraint may not be used with it. Following are the available tools and methods.

Schematic	VHDL
Verilog	ABEL
NCF	UCF
XCF	Constraints Editor
PCF	Floorplanner
PACE	FPGA Editor
Project Navigator	

- Additional Information
  Additional information is provided for certain constraints.

Alphabetized List of Xilinx Constraints

This chapter contains information on the following constraints:

- Timing Ignore (TIG)
- Timing Group (TIMEGRP)
- Timing Specifications (TIMESPEC)
- Timing Name (TNM)
- Timing Name Net (TNM_NET)
- Timing Point Synchronization (TPSYNC)
- Timing Thru Points (TPTHRU)
- Timing Specification Identifier (TSidentifier)
- U_SET
- Use Relative Location (USE_RLOC)
- Use Low Skew Lines (USELOWSKEWLINES)
- VCCAUX
- Voltage (VOLTAGE)
- VREF
- Wire And (WIREAND)
- XBLKNM

Area Group (AREA_GROUP)

AREA_GROUP Architecture Support

The AREA_GROUP constraint applies to FPGA devices only.

AREA_GROUP Applicable Elements

- Logic blocks
- Timing groups

For more information, see "Defining From Timing Groups" in this chapter.

AREA_GROUP Description

AREA_GROUP is a design implementation constraint that enables partitioning of the design into physical regions for mapping, packing, placement, and routing.

AREA_GROUP is attached to logical blocks in the design, and the string value of the constraint identifies a named group of logical blocks that are to be packed together by mapper and placed in the ranges if specified by PAR. If AREA_GROUP is attached to a hierarchical block, all sub-blocks in the block are assigned to the group.

Once defined, an AREA_GROUP can have a variety of additional constraints associated with it to control its implementation. For more information, see "AREA_GROUP Syntax" in this chapter.

AREA_GROUP Propagation Rules

The following rules apply to AREA_GROUP.

- When attached to a design element, AREA_GROUP is propagated to all applicable elements in the hierarchy below the component.
- It is illegal to attach AREA_GROUP to a net, signal, or pin.

AREA_GROUP Syntax

The basic UCF syntax for defining an area group is:

```
INST "X" AREA_GROUP=groupname;
```

The syntax to be used in attaching constraints to an area group is:

```
AREA_GROUP "groupname" RANGE=range;
```

or

```
AREA_GROUP "groupname" COMPRESSION=percent;
```

or

```
AREA_GROUP "groupname" IMPLEMENT={FORCE|AUTO};
```

or

```
AREA_GROUP "groupname" GROUP={OPEN|CLOSED};
```

or

```
AREA_GROUP "groupname" PLACE={OPEN|CLOSED};
```

or

AREA_GROUP "*groupname***" MODE={RECONFIG};**

where

- *groupname* is the name assigned to an implementation partition to uniquely define the group

Each of these additional AREA_GROUP constraints is described below.

RANGE

RANGE defines the range of device resources that are available to place logic contained in the AREA_GROUP, in the same manner ranges are defined for the LOC constraint.

For Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices, RANGE syntax is as follows:

RANGE=CLB_R*m1***Cn1:CLB_***rm2***C***n2***

RANGE=TBUF_R*m1***Cn1:TBUF_***rm2***C***n2***

RANGE=RAMB4_R*m1***Cn1:RAMB4_***rm2***C***n2***

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, RANGE syntax is as follows:

RANGE=SLICE_X*m1***Yn1:SLICE_X***m2***Y***n2***

RANGE=TBUF_X*m1***Yn1:TBUF_X***m2***Y***n2***

RANGE=MULT18X18_X*m1***Yn1:MULT18X18_X***m2***Y***n2***

RANGE=RAMB16_X*m1***Yn1:RAMB16_X***m2***Y***n2***

TBUF is not supported by Spartan-3, Spartan-3A, Spartan-3E, Virtex-4, and Virtex-5 devices.

All FPGA devices and CLBs/SLICEs are supported. If an AREA_GROUP contains both TBUFs (not applicable for Spartan-3, Spartan-3A, and Spartan-3E) and one for CLBs/SLICEs, two separate AREA_GROUP RANGEs can be specified: one for TBUFs and one for CLBs/SLICEs.

You can use the wildcard character for either the row number or column number. For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, you can use the wildcard character for either the X coordinate or the Y coordinate.

COMPRESSION

COMPRESSION defines the compression factor for the AREA_GROUPs. The percent values can be from 0 to 100. If the AREA_GROUP does not have a RANGE, only 0 (no compression) and 1 (maximum compression) are meaningful. The mapper computes the number of CLBs in the AREA_GROUP from the RANGE and attempts to compress the logic into the percentage specified. Compression does not apply to TBUFs, BRAMs, or multipliers.

The compression factor is similar to the **-c** option in MAP, except that it operates on the AREA_GROUP instead of the whole design. AREA_GROUP compression interacts with the **-c** map option as follows:

- Area groups with a compression factor are not affected by the **-c** option. (Logic that is not part of an area group is not merged with grouped logic if the AREA_GROUP has its own compression factor.)

- Area groups without a compression factor are affected by the **-c** option. The mapper may attempt to combine ungrouped logic with logic that is part of an area group without a compression factor.
- At no time is the logic from two separate area groups combined.
- The **-c** map option does not force compression among slices in the same area group.

The Map Report (MRP) includes a section that summarizes AREA_GROUP processing.

If a symbol that is part of an AREA_GROUP contains a LOC constraint, the mapper removes the symbol from the area group and processes the LOC constraint.

Logic that does not belong to any AREA_GROUP can be pulled into the region of logic belonging to an area group, as well as being packed or merged with such logic to form SLICES.

IMPLEMENT

For IMPLEMENT, the string value must be one of the following.

FORCE

Forces the AREA_GROUP logic to be re-implemented.

AUTO

Determines if the AREA_GROUP logic has changed and, if so, the logic is reimplemented. The default is AUTO.

GROUP

GROUP controls the packing of logic into physical components (that is, slices) as follows.

CLOSED

Do not allow logic *outside* the AREA_GROUP to be combined with logic *inside* the AREA_GROUP.

OPEN

Allow logic *outside* the AREA_GROUP to be combined with logic *inside* the AREA_GROUP.

Default

The default value is GROUP=OPEN.

PLACE

PLACE controls the allocation of resources in the area group's RANGE, as follows.

CLOSED

Do not allow comps that are not members of the AREA_GROUP to be placed within the RANGE defined for the AREA_GROUP.

OPEN

Allow comps that are not members of the AREA_GROUP to be placed within the RANGE defined for the AREA_GROUP.

Default

The default value is PLACE=OPEN.

MODE

MODE is used to define a reconfigurable area group, as in the following example:

MODE=RECONFIG

AREA_GROUP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach AREA_GROUP=*groupname* to a valid instance.
- Attach RANGE =*range* to a CONFIG symbol.
- Attach COMPRESSION=*percent* to a CONFIG symbol.
- Attach IMPLEMENT={**FORCE** | **AUTO**} to a CONFIG symbol.
- Attach GROUP={**OPEN** | **CLOSED**} to a CONFIG symbol.
- Attach PLACE={**OPEN** | **CLOSED**} to a CONFIG symbol.
- Attach to a CONFIG symbol. For a value of TRUE, PLACE, and GROUP must both be CLOSED.
- Attribute Names: AREA_GROUP, RANGE *range*, COMPRESSION *percent*, IMPLEMENT={**FORCE** | **AUTO**}, GROUP={**OPEN** | **CLOSED**}, PLACE={**OPEN** | **CLOSED**}, and MODE={**RECONFIG**}.
- Attribute Values: *groupname*, *range*, *percent*, IMPLEMENT={**FORCE** | **AUTO**}, GROUP={**OPEN** | **CLOSED**}, PLACE={**OPEN** | **CLOSED**}, MODE={**RECONFIG**}

UCF and NCF Syntax Example

For architectures with slice-based XY designations (Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only).

The following example assigns all the logical blocks in state_machine_X to the area group "group1" and places CLB logic in the physical area between CLB 1,1 and CLB 10,10. It places TBUFs in the physical area between TBUF 1,0 and TBUF 10,10. Unrelated logic within "group1" is not compressed. Because compression is defined, ungrouped logic is not combined with logic in "group1."

```
INST "state_machine_X" AREA_GROUP=group1;
AREA_GROUP "group1" COMPRESSION=0;
AREA_GROUP "group1" RANGE=CLB_R1C1:CLB_R10C10;
AREA_GROUP "group1" RANGE=TBUF_X6Y0:TBUF_X10Y22;
```

Note: Not applicable for architectures that do not contain internal tristate buffers.

The following example assigns all the logical blocks in state_machine_X to the area group, "group1," and places logic in the physical area bounded by SLICE_X3Y1 in the lower left corner and SLICE_X33Y33 in the upper left corner. It places TBUFs in the physical area bounded by TBUF_X6Y0 and TBUF_X10Y22. Unrelated logic within "group1" *not*

compressed. Because compression is defined, ungrouped logic *not* combined with logic in "group1."

```
INST “state_machine_X” AREA_GROUP=group1;
AREA_GROUP “group1” COMPRESSION=0;
AREA_GROUP “group1” RANGE=SLICE_X3Y1:SLICE_X33Y33;
AREA_GROUP “group1” RANGE=TBUF_X6Y0:TBUF_X10Y22;
```

The following example assigns I$1, I$2, I$3, and I$4 to the area group "group2." Because there is no compression, ungrouped logic may be combined within this area group.

```
INST “I$1” AREA_GROUP=group2;
INST “I$2” AREA_GROUP=group2;
INST “I$3” AREA_GROUP=group2;
INST “I$4” AREA_GROUP=group2;
```

Floorplanner Syntax Example

See the following topics in the Floorplanner help:

- “Using a Floorplanner UCF File in Project Navigator”
- “Creating and Editing Area Constraints”

PACE Syntax Example

The Pin AREA Constraints Editor (PACE) is mainly used to identify and assign areas to hierarchical blocks of logic. You can access PACE from the Processes window in the Project Navigator. Double-click Create Area Constraints. For more information, see the PACE help, especially “Editing Area Constraints.”

Defining From Timing Groups

To create an area group based on a timing group, use the following UCF and NCF syntax:

```
TIMEGRP timing_group_name AREA_GROUP = area_group_name;
```

where

- *timing_group_name* is the name of a previously defined timing group
- *area_group_name* is the name of a new area group to be defined from the TIMEGRP contents

This is equivalent to manually assigning each member of the timing group to *area_group_name*. The area group name defined by this statement can be used in RANGE constraints, just like any other area group name.

In the AREA_GROUP definition, the *timing_group_name* is generally TNM_NET group, which allows area groups to be formed based on the loads of clock or other control nets. Defining AREA_GROUPs from TIMEGRPs is useful for improving placement of designs with many different clock domains in devices that have more clocks than clock regions.

You can also specify a TNM group name, or the name of a user group defined by a TIMEGRP statement. Edge qualifiers used in the TIMEGRP definition are ignored when determining area group membership. In all cases, the AREA_GROUP members are determined after the TIMEGRP has been propagated to its target elements.

Since TIMEGRPs can contain only synchronous elements and pads, area groups defined from timing groups also contain only these element types. If an AREA_GROUP is defined

by a TIMEGRP that contains only flip-flops or latches, assigning a RANGE to that group makes sense only if ungrouped logic is also allowed within the area. Therefore, COMPRESSION should not be defined for such groups.

If a TNM_NET is used by a PERIOD specification, and is traced into a Virtex, Virtex-E, Spartan-II, Spartan-IIE, CLKDLL or Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex, Virtex-4, or Virtex-5 devices, DCM, new TNM_NET groups and PERIOD specifications are created at the CLKDLL or DCM outputs. If the original TNM_NET is used to define an area group, and if more than one clock tap is used on the CLKDLL or DCM, the area group is split into separate groups at each clock tap.

For example, assume you have the following UCF constraints:

```
NET "clk" TNM_NET="clock";
TIMESPEC "TS_clk" = PERIOD "clock" 10 MHz;
TIMEGRP "clock" AREA_GROUP="clock_area";
```

If the net clk is traced into a CLKDLL or DCM, a new group and PERIOD specification is created at each clock tap. Likewise, a new area group is created at each clock tap, with a suffix indicating the clock tap name. If the CLK0 and CLK2X taps were used, the AREA_GROUPS clock_area_CLK0 and clock_area_CLK2X are defined automatically.

When AREA_GROUP definitions are split in this manner, NGDBuild issues an informational message, showing the names of the new groups. These new group names, rather than the originally specified one, should be used in RANGE constraints.

Defining from Area Groups

To create an area group based on an area group, use the following UCF and NCF syntax:

```
AREAGRP timing_group_name AREA_GROUP = area_group_name;
```

where

- *area_group_name* is the name of a previously defined timing group
- *area_group_name* is the name of a new area group to be defined from the TIMEGRP contents

Asynchronous Register (ASYNC_REG)

ASYNC_REG Architecture Support

The ASYNC_REG constraint applies to FPGA devices only.

ASYNC_REG Applicable Elements

The ASYNC_REG constraint can be attached to registers and latches only. It should be used only on registers or latches with asynchronous inputs (D input or the CE input).

ASYNC_REG Description

The ASYNC_REG timing constraint improves the behavior of asynchronously clocked data for simulation. Specifically, it disables 'X' propagation during timing simulation. In the event of a timing violation, the previous value is retained on the output instead of going unknown.

ASYNC_REG Propagation Rules

Applies to the register or latch to which it is attached

ASYNC_REG Syntax Examples

Following are syntax examples using the ASYNC_REG constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute ASYNC_REG : string;
```

Specify the VHDL constraint as follows:

```
attribute ASYNC_REG of instance_name: label is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* ASYNC_REG = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

```
INST "instance_name" ASYNC_REG = {TRUE|FALSE};
```

The default (if constraint is not applied) is FALSE. If no boolean value is supplied it is considered TRUE.

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. You can set using the Misc tab.

BEL

BEL Architecture Support

The BEL constraint applies to the following devices:

- Virtex™-II
- Virtex-4
- Virtex-5
- Spartan™-3
- Spartan-3A
- Spartan-3E

BEL Applicable Elements

- Registers
- LUTRAMs
- RAMB18s
- Latches
- SRLs

BEL Description

BEL is an advanced placement constraint. It locks a logical symbol to a particular BEL site in a slice, or an IOB. BEL differs from "Location (LOC)" in that LOC allows specification to the comp level. BEL allows specification as to which particular BEL site of the slice or IOB slice is to be used. The BEL constraint should always be used with an appropriate LOC or RLOC attribute.

An IOB BEL constraint does not direct the mapper to pack the register into an IOB component. Some other feature (the **`-pr`** switch, for example) must cause the packing. Once the register is directed to an IOB, the BEL constraint causes the proper placement within the IOB.

BEL Propagation Rules

It is only legal to place a BEL constraint on an appropriate instance with a valid LOC or RLOC.

BEL Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: BEL
- Attribute Values: F, G, FFA, FFB, FFC, FFD, FFX, FFY, XORF, XORG, A6LUT, B6LUT, C6LUT, D6LUT A5LUT, B5LUT, C5LUT, D5LUT

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute bel : string;
```

Specify the VHDL constraint as follows:

```
attribute bel of {component_name|label_name}: {component|label} is
"{F|G|FFA|FFB|FFC|FFD|FFX|FFY|XORF|XORG|A6LUT|B6LUT|C6LUT|D6LUT|A5LUT|
B5LUT|C5LUT|
D5LUT}";
```

For a description of BEL values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* BEL =
"{F|G|FFA|FFB|FFC|FFD|FFX|FFY|XORF|XORG|A6LUT|B6LUT|C6LUT|D6LUT|A5LUT|
B5LUT|C5LUT|
D5LUT}" *)
```

For a description of BEL values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The syntax is:

```
INST "instance_name" BEL={F | G | FFA | FFB | FFC | FFD | FFX | FFY |
XORF | XORG | A6LUT | B6LUT | C6LUT | D6LUT | A5LUT | B5LUT | C5LUT |
D5LUT};
```

where

- F , G, A6LUT, B6LUT, C6LUT, D6LUT, A5LUT, B5LUT, C5LUT and D5LUT identify specific LUTs, SRL16s, distributed RAM components in the slice
- FFX, FFY, FFA, FFB, FFC and FFD identify specific flip-flops, latches, and other elements in a slice
- XORF and XORG identify XORCY elements in a slice

The syntax for the RAMB BEL instance is:

```
INST "upper_BRAM_instance_name" LOC = RAMB36_XnYn | BEL = UPPER;
INST "lower_BRAM_instance_name" LOC = RAMB36_XnYn | BEL = LOWER;
```

Example:

```
INST "ramb18_inst0" LOC = RAMB36_X0Y2 | BEL = UPPER;
INST "ramb18_inst1" LOC = RAMB36_X0Y2 | BEL = LOWER;
```

The following statement locks **xyzzy** to the FFX site on the slice.

```
INST "xyzzy" BEL=FFX;
```

Block Name (BLKNM)

BLKNM Architecture Support

The BLKNM constraint applies to FPGA devices only.

BLKNM Applicable Elements

The BLKNM constraint may be used with an FPGA device in one or more of the following design elements, or categories of design elements. Not all device families support all these elements. To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

- Flip-flop and latch primitives
- Any I/O element or pad
- FMAP
- BUFT
- ROM primitives
- RAMS and RAMD primitives
- Carry logic primitives

You can also attach BLKNM to the net connected to the pad component in a UCF file. NGDBuild transfers the constraint from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following syntax:

```
NET "net_name" BLKNM=property_value;
```

BLKNM Description

BLKNM is an advanced mapping constraint. BLKNM assigns block names to qualifying primitives and logic elements. If the same BLKNM constraint is assigned to more than one instance, the software attempts to map them into the same block. Conversely, two symbols with different BLKNM names are not mapped into the same block. Placing similar BLKNM constraints on instances that do not fit within one block creates an error.

Specifying identical BLKNM constraints on FMAP tells the software to group the associated function generators into a single SLICE. Using BLKNM, you can partition a complete SLICE without constraining the SLICE to a physical location on the device.

BLKNM constraints, like LOC constraints, are specified from the design. Hierarchical paths are not prefixed to BLKNM constraints, so BLKNM constraints for different SLICEs must be unique throughout the entire design. For information on attaching hierarchy to block names, see the "Hierarchical Block Name (HBLKNM)" constraint.

BLKNM allows any elements except those with a different BLKNM to be mapped into the same physical component. Elements without a BLKNM can be packed with those that have a BLKNM. For information on allowing only elements with the same XBLKNM to be mapped into the same physical component, see the "XBLKNM" constraint.

BLKNM Propagation Rules

When attached to a design element, it is propagated to all applicable elements in the hierarchy within the design element.

BLKNM Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: BLKNM
- Attribute Value: *block_name*

VHDL Syntax Example

Declare the VHDL constraint with the following syntax:

```
attribute blknm: string;
```

Specify the VHDL constraint as follows:

```
attribute blknm of
{component_name|signal_name|entity_name|label_name}:
{component|signal|entity|label} is "block_name";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* BLKNM = "blk_name" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" BLKNM=block_name;
```

where

- *block_name* is a valid block name for that type of symbol

For information on assigning hierarchical block names, see the "Hierarchical Block Name (HBLKNM)"constraint.

The following statement assigns an instantiation of an element named block1 to a block named U1358.

```
INST "$1I87/block1" BLKNM=U1358;
```

XCF Syntax Example

```
MODEL "entity_name" blknm = block_name;


BEGIN MODEL "entity_name"
 INST "instance_name" blknm = block_name;
END;
```

BUFG (CPLD)

BUFG (CPLD) Architecture Support

The BUFG (CPLD) applies to CPLD devices only.

BUFG (CPLD) Applicable Elements

Any input buffer (IBUF), input pad net, or internal net that drives a CLK, OE, SR, DATA_GATE pin

BUFG (CPLD) Description

BUFG is an advanced fitter constraint and a synthesis constraint. When applied to an input buffer or input pad net, the BUFG attribute maps the tagged signal to a global net. When applied to an internal net, the tagged signal is either routed directly to a global net or brought out to a global control pin to drive the global net, as supported by the target device family architecture.

BUFG (CPLD) Propagation Rules

When attached to a net, BUFG has a net or signal form and so no special propagation is required. When attached to a design element, BUFG is propagated to all applicable elements in the hierarchy within the design element.

BUFG (CPLD) Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an IBUF instance of the input pad connected to an IBUF input
- Attribute Name: BUFG
- Attribute Values: CLK, OE, SR, DATA_GATE
- BUFG=CLK: maps to a global clock (GCK) line
- BUFG=OE: maps to a global 3-state control (GTS) line
- BUFG=SR: maps to a global set/reset control (GSR) line
- BUFG=DATA_GATE: maps to the DataGate latch enable control line

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute BUFG: string;
```

Specify the VHDL constraint as follows:

```
attribute BUFG of signal_name: signal is "{CLK|OE|SR|DATA_GATE}";
```

BUFG=CLK: maps to a global clock (GCK) line.

BUFG=OE: maps to a global 3-state control (GTS) line.

BUFG=SR: maps to a global set/reset control (GSR) line.

BUFG=DATA_GATE: maps to the DataGate latch enable control line.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify BUFG as follows:

```
(* BUFG = "{CLK|OE|SR|DATA_GATE}" *)
```

BUFG=CLK: maps to a global clock (GCK) line.

BUFG=OE: maps to a global 3-state control (GTS) line.

BUFG=SR: maps to a global set/reset control (GSR) line.

BUFG=DATA_GATE: maps to the DataGate latch enable control line.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'bufg={clk|oe|sr|DATA_GATE} signal_name';
```

UCF and NCF Syntax Example

The basic UCF syntax is

```
NET "net_name" BUFG={CLK | OE | SR | DATA_GATE};
INST "instance_name" BUFG={CLK | OE | SR | DATA_GATE};
```

where

- CLK designates a global clock pin (all CPLD families)
- OE designates a global 3-state control pin (all CPLD devices except CoolRunner) or internal global 3-state control line (CoolRunner-II only).
- SR designates a global set/reset pin (all CPLD devices except CoolRunner)
- DATA_GATE maps to the DataGate latch enable control line

The following statement maps the signal named **fastclk** to a global clock net.

```
NET "fastclk" BUFG=CLK;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" BUFG = {CLK|OE|SR|DATA_GATE};
END;
```

Clock Dedicated Route

CLOCK_DEDICATED_ROUTE Architecture Support

The CLOCK_DEDICATED_ROUTE constraint applies to the following devices:

- Spartan™-3
- Spartan-3E
- Spartan-3A
- Spartan-3A DSP
- Virtex™-4
- Virtex-5

CLOCK_DEDICATED_ROUTE Applicable Elements

- Nets

Input and output pins of the follwing primitives:

- BUFG
- BUFR
- DCM
- PLLPMCD
- GT11
- GT11 DUAL
- GT11 CLK
- GTP DUAL

CLOCK_DEDICATED_ROUTE Description

CLOCK_DEDICATED_ROUTE constraint is an advanced constraint that directs the tools whether or not to follow clock placement rules for a specific architecture. If the constraint is not used or set to TRUE, clock placement rules must be followed. Otherwise, placement will error. If the constraint is set to FALSE, it directs the tools to ignore the specific clock placement rule and continue with place and route. If possible, all clock placement rule violations should be fixed in a design in order to ensure the best clocking performance. This constraint is intended to be used only in limited situations when it is absolutely necessary to violate a clock placement rule. Please see the *Hardware User's Guide* for more details about specific clock placement rules.

CLOCK_DEDICATED_ROUTE Propagation Rules

Applies to the NET or INSTANCE PIN.

CLOCK_DEDICATED_ROUTE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: CLOCK_DEDICATED_ROUTE

- TRUE, FALSE

UCF and NCF Syntax Example

The syntax is:

```
PIN "<BEL_INSTANCE_NAME.PIN>" CLOCK_DEDICATED_ROUTE = (TRUE|FALSE);
```

where

```
<BEL_INSTANCE_NAME.PIN>
```

is the specific input/output pin of the instance you want to constrain. An example is the CLKIN input pin of a DCM instance.

Collapse (COLLAPSE)

COLLAPSE Architecture Support

The COLLAPSE constraint applies to CPLD devices only.

COLLAPSE Applicable Elements

Any internal net.

COLLAPSE Description

COLLAPSE is an advanced fitter constraint. It forces a combinatorial node to be collapsed into all of its fanouts.

COLLAPSE Propagation Rules

COLLAPSE is a net constraint. Any attachment to a design element is illegal.

COLLAPSE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a logic symbol or its output net
- Attribute Name: COLLAPSE
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute collapse: string;
```

Specify the VHDL constraint as follows:

```
attribute collapse of signal_name: signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* COLLAPSE = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
NET "net_name" COLLAPSE;
```

The following statement forces net $1N6745 to collapse into all its fanouts.

```
NET "$1I87/$1N6745" COLLAPSE;
```

Component Group (COMPGRP)

COMPGRP Architecture Support

The COMPGRP constraint applies to FPGA devices only.

COMPGRP Applicable Elements

Groups of components

COMPGRP Description

COMPGRP is an advanced grouping constraint that identifies a group of components.

COMPGRP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

PCF Syntax Example

```
COMPGRP "group_name"=comp_item1... comp_itemn [EXCEPT comp_group];
```

where

- *comp_item* is one of the following
  - **COMP** "*comp_name*"
  - **COMPGRP** "*group_name*"

CoolCLOCK (COOL_CLK)

COOL_CLK Architecture Support

The COOL_CLK constraint applies to Coolrunner™-II devices only.

COOL_CLK Applicable Elements

Applies to any input pad or internal signal driving a register clock.

COOL_CLK Description

You can save power by combining clock division circuitry with the DualEDGE circuitry. This capability is called COOL_CLK. It is designed to reduce clocking power within a CPLD. Because the clock net can be a significant power drain, the clock power can be reduced by driving the net at half frequency, then doubling the clock rate using DualEDGE triggered macrocells.

COOL_CLK Propagation Rules

Applying COOL_CLK to a clock net is equivalent to passing the clock through a divide-by-two clock divider (CLK_DIV2) and replacing all flip-flops controlled by that clock with DualEDGE flip-flops. Using the COOL_CLK attribute does not alter your overall design functionality.

Some restrictions apply:

- You cannot use COOL_CLK on a clock that triggers any flip-flop on the low-going edge. The CoolRunner-II clock divider can be triggered only on the high-rising edge of the clock signal.
- If there are any DualEDGE flip-flops in your design source, the clock that controls any of them cannot be specified as a COOL_CLK.
- If there is already a clock divider in your design source, you cannot also use COOL_CLK. CoolRunner-II devices contain only one clock divider.

COOL_CLK Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a input pad or internal signal driving a register clock
- Attribute Name: COOL_CLK
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute cool_clk: string;
```

Specify the VHDL constraint as follows:

```
attribute cool_clk of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* COOL_CLK = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'COOL_CLK signal_name';
```

UCF and NCF Syntax Example

```
NET "signal_name" COOL_CLK;
```

Configuration Mode (CONFIG_MODE)

CONFIG_MODE Architecture Support

The CONFIG_MODE constraint applies to the following devices:

- Virtex™
- Virtex-E
- Virtex-II
- Virtex-II Pro
- Virtex-4
- Spartan™-II
- Spartan-IIE
- Spartan-3

CONFIG_MODE Applicable Elements

Attaches to the CONFIG symbol.

CONFIG_MODE Description

This constraint communicates to PAR which of the dual purpose configuration pins can be used as general purpose IOs.

This constraint is used by PAR to prohibit the use of Dual Purpose IOs if they are required for CONFIG_MODE: S_SELECTMAP+READBACK OR M_SELECTMAP+READBACK.

In the case of CONFIG_MODE: S_SELECTMAP OR M_SELECTMAP, PAR uses the Dual Purpose IOs as General Purpose IOs only if necessary.

CONFIG_MODE Propagation Rules

Applies to dual-purpose I/Os

CONFIG_MODE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

The basic UCF syntax is:

```
CONFIG CONFIG_MODE=string;
```

where

- *string* can be one of the following:
  - S_SERIAL = Slave Serial Mode
  - M_SERIAL = Master Serial Mode (The default value)
  - S_SELECTMAP = Slave SelectMAP Mode
  - M_SELECTMAP = Master SelectMAP Mode.

- B_SCAN = Boundary Scan Mode
- S_SELECTMAP+READBACK = Slave SelectMAP Mode with Persist set to support Readback and Reconfiguration.
- M_SELECTMAP+READBACK = Mater SelectMAP Mode with Persist set to support Readback and Reconfiguration.
- B_SCAN+READBACK = Boundary Scan Mode with Persist set to support Readback and Reconfiguration
- S_SELECTMAP32+READBACK
- S_SELECTMAP32

Data Gate (DATA_GATE)

DATA_GATE Architecture Support

The DATA_GATE constraint only applies to Coolrunner™-II devices with 128 macrocells or more.

DATA_GATE Applicable Elements

I/O pads and pins

DATA_GATE Description

The CoolRunner-II DataGate feature provides direct means of reducing power consumption in your design. Each I/O pin input signal passes through a latch that can block the propagation of incident transitions during periods when such transitions are not of interest to your CPLD design. Input transitions that do not affect the CPLD design function still consume power, if not latched, as they are routed among the device's function blocks. By asserting the DATA_GATE control I/O pin on the device, selected I/O pin inputs become latched, thereby eliminating the power dissipation associated with external transitions on those pins.

Applying the DATA_GATE attribute to any I/O pad indicates that the pass-through latch on that device pin is to respond to the DataGate control line. Any I/O pad (except the DATA_GATE control I/O pin itself), including clock input pads, can be configured to get latched by applying the DATA_GATE attribute. All other I/O pads that do not have a DATA_GATE attribute remain unlatched at all times. The DATA_GATE control signal itself can be received from off-chip via the DATA_GATE I/O pin, or you can generate it in your design based on inputs that remain unlatched (pads without DATA_GATE attributes).

For more information on using DATA_GATE with Verilog and VHDL designs, see the "BUFG (CPLD)" constraint.

DATA_GATE Propagation Rules

See "DATA_GATE Description" in this chapter.

DATA_GATE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to I/O pads and pins
- Attribute Name: DATA_GATE
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute DATA_GATE : string;
```

Specify the VHDL constraint as follows:

```
attribute DATA_GATE of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* DATA_GATE = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'DATA_GATE signal_name';
```

NCF Syntax Example

Same as UCF

UCF Syntax Example

```
NET "signal_name" DATA_GATE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
NET "signal_name" data_gate={TRUE|FALSE}";
END;
```

DCI_CASCADE

DCI_CASCADE Architecture Support

The DCI_CASCADE constraint applies to Virtex™-5 devices only.

DCI_CASCADE Applicable Elements

A DCI_CASCADE attribute on the top level design block.

DCI_CASCADE Description

In Virtex-5 device families, IO banks that need DCI reference voltage can be cascaded with other DCI IO banks. One set of VRN/VRP pins can be used to provide reference voltage to several IO banks. This results in more usable pins and in reduced power usage because fewer VR pins and DCI controllers are used. The DCI_CASCADE constraint is used to identify a DCI master bank and its corresponding slave banks. There can be multiple instances of this constraint for a design in order to specify multiple master-slave pairs. BitGen uses information from this constraint to program DCI controllers for different banks and have them cascade up or down. The placer will also use this information to determine whether VR pins in slave banks can be used for other purposes.

Each instance of the DCI_CASCADE constraint must have one master bank and one or more slave banks that can be entered as a space-separated list. The first value in the list is the master bank and all subsequent values are slave banks that get DCI reference voltage from the master bank. Cascaded banks must be in the same column (left, center or right) and must have the same VCCO setting. See "UCF and NCF Syntax Example" for this constraint for more rules.

DCI_CASCADE Propagation Rules

Placed as an attribute on the CONFIG block, and propagated to the physical design object.

DCI_CASCADE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

Not supported

VHDL Syntax Example

Not supported

Verilog Syntax Example

Not supported

ABEL Syntax Example

Not supported

UCF and NCF Syntax Example

The UCF and NCF syntax is:

```
CONFIG DCI_CASCADE = "<master> <slave1> <slave2> ...";
```

where,

- <*master*> = [1...MAX_NUM_BANKS]
- <*slave1*> = [1...MAX_NUM_BANKS]
- <*slave2*> = [1...MAX_NUM_BANKS]
- All values in the list are legitimate IO banks in the Virtex-5 device.
- The master bank must have an IOB with an IO standard that requires DCI reference voltage.
- All slave banks must have the same VCCO setting as the master bank.
- If there are banks between the master and slave, they should be able to cascade in the required direction.

For Example:

```
CONFIG DCI_CASCADE = "11 13 15 17";
```

XCF Syntax Example

Not supported

Constraints Editor Syntax Example

Not supported

PCF Syntax Example

```
CONFIG DCI_CASCADE = "<master>, <slave1>, <slave2>, ..."
```

where,

- <master> = [1...MAX_NUM_BANKS]
- <slave1> = [1...MAX_NUM_BANKS]
- <slave2> = [1...MAX_NUM_BANKS]

Floorplanner Syntax Example

Not supported

PACE Syntax Example

Not supported

Floorplan Editor Syntax Example

Not supported

FPGA Editor Syntax Example

Not supported

Project Navigator Syntax Example

Not supported

DCI_VALUE

DCI_VALUE Architecture Support

The DCI_VALUE constraint applies to the following devices:

- Virtex™-II
- Virtex-II Pro
- Virtex-II Pro X
- Virtex-4
- Virtex-5
- Spartan™-3

DCI_VALUE Applicable Elements

IOBs

DCI_VALUE Description

DCI_VALUE determines which buffer behavioral models are associated with the IOBs of a design in the generation of an IBS file using IBISWriter.

DCI_VALUE Propagation Rules

Applies to the IOB to which it is attached

DCI_VALUE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
INST pin_name DCI_VALUE = integer;
```

Legal values are integers 25 through 100 with an implied units of ohms. The default value is 50 ohms.

Directed Routing (DIRECTED_ROUTING)

DIRECTED_ROUTING Architecture Support

The DIRECTED_ROUTING constraint applies to the following devices:

- Virtex™-II
- Virtex-II Pro
- Virtex-4
- Virtex-5
- Spartan™-3
- Spartan-3A
- Spartan-3E

DIRECTED_ROUTING Applicable Elements

Applies only to nets.

DIRECTED_ROUTING Description

DIRECTED_ROUTING is a means of maintaining the routing and timing for a small number of loads and sources. Use of directed routing requires that the relative position between the sources and loads be maintained exactly the same.

DIRECTED_ROUTING Propagation Rules

Not applicable

DIRECTED_ROUTING Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

The following examples are for illustration only. They are not valid executables. Formulation of a directed routing constraint requires the placement of the source and load components in a fixed location relative to each other.

FPGA Editor Syntax Example

To generate directed routing constraints with FPGA Editor, select **Tools > Directed Routing Constraints**. FPGA Editor provides the following three settings for the type of placement constraint to be generated automatically on the sources and loads components.

- "Do Not Generate Placement Constraint"
- "Use Relative Location Constraint"
- "Use Absolute Location Constraint"

Do Not Generate Placement Constraint

"Do Not Generate Placement Constraint" generates a constraint for the routing only. It is designed to be used with existing RPMs.

```
NET "net_name" ROUTE="{2;1;-4!-1;-53320;2920;14;90;200;30;13!0;-
2091;1480;24!0;16;-8!}";
```

Use Relative Location Constraint

"Use Relative Location Constraint" generates an RPM for the source and load components along with the routing constraint. The RPM can be relocated around the device letting the Placer make the final decision on placement.

```
NET "net_name" ROUTE="{2;1;-4!-1;-53320;2920;14;90;200;30;13!0;-
2091;1480;24!0;16;-8!}";
INST "inst1" RLOC=X3Y0;
INST "inst1" RPM_GRID=GRID;
INST "inst1" U_SET=macro name;
INST "inst1" BEL="F";
INST "inst2" RLOC=X3Y0;
INST "inst2" U_SET=macro name;
INST "inst2" BEL="G";
```

In the above example, each RLOC reference signals the launch of a new instance. Accordingly, there are three instances encompassed within this example.

Use Absolute Location Constraint

"Use Absolute Location Constraint" causes the source and load components attached to the target net to be locked in place.

```
NET "net_name" ROUTE="{2;1;-4!-1;-53320;2920;14;90;200;30;13!0;-
2091;1480;24!0;16;-8!}";
INST "inst1" RLOC=X3Y0;
INST "inst1" RPM_GRID=GRID;
INST "inst1" RLOC_ORIGIN=X87Y200;
INST "inst1" U_SET=macro name;
INST "inst1" BEL="F";
INST "inst2" RLOC=X0Y1;
INST "inst2" U_SET=macro name;
INST "inst2" BEL="F";
INST "inst3" RLOC=X3Y0;
INST "inst3" U_SET=macro name;
INST "inst3" BEL="G";
```

Disable (DISABLE)

DISABLE Architecture Support

The DISABLE constraint applies to FPGA devices only.

DISABLE Applicable Elements

Global in constraints file.

DISABLE Description

DISABLE is an advanced timing constraint. It controls path tracing. All path tracing control statements from any source (netlist, UCF, or NCF) are passed forward to the PCF. You cannot override a DISABLE in the netlist with an "Enable (ENABLE)" in the UCF.

DISABLE Propagation Rules

Disables timing analysis of specified block delay symbol

DISABLE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
DISABLE=delay_symbol_name;
```

where

- *delay_symbol_name* is the name of one of the standard block delay symbols for path tracing or a specific delay name in the datasheet

These symbols are listed in the following table. Component delay names are also supported in the PCF.

Table 20-1: **Standard Block Delay Symbols for Path Tracing**

Delay Symbol Name	Path Type	Default
reg_sr_o	Asynchronous Set/Reset to output propagation delay	Disabled
reg_sr_r	Asynchronous Set/Reset to recovery path	Disabled
reg_sr_clk	Synchronous Set/Reset to clock setup and hold checks	Enabled
lat_d_q	Data to output transparent latch delay	Disabled
ram_we_o	RAM write enable to output propagation delay	Enabled

Table 20-1: **Standard Block Delay Symbols for Path Tracing**

Delay Symbol Name	Path Type	Default
tbuf_t_o	TBUF 3-state to output propagation delay	Enabled
tbuf_i_o	TBUF input to output propagation delay	Enabled
io_pad_i	IO pad to input propagation delay	Enabled
io_t_pad	IO 3-state to pad propagation delay	Enabled
io_o_i	IO output to input propagation delay. Disabled for 3-stated IOBs.	Enabled
io_o_pad	IO output to pad propagation delay.	Enabled

The following statement prevents timing analysis on any path that includes the I to O delay on any TBUF component in the design.

```
DISABLE=tbuf_i_o;
```

PCF Syntax Example

Same as UCF

Drive (DRIVE)

DRIVE Architecture Support

The DRIVE constraint applies to FPGA devices only.

DRIVE Applicable Elements

If "Yes" is shown next to the device name in the Architecture Support table, the constraint may be used with that device in one or more of the following design elements, or categories of design elements. Not all device families support all these elements. For which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

- IOB output components (such as OBUF and OFD)
- SelectIO output buffers with IOSTANDARD = LVTTL, LVCMOS15, LVCMOS18, LVCMOS25, or LVCMOS33
- Nets

DRIVE Description

DRIVE is a basic mapping directive that selects the output for the following devices:

- Virtex
- Virtex-E
- Virtex-II
- Virtex-II Pro
- Virtex-II Pro X
- Virtex-4
- Virtex-5
- Spartan-II
- Spartan-IIE
- Spartan-3
- Spartan-3A
- Spartan-3E

DRIVE selects output drive strength (mA) for the SelectIO buffers that use the LVTTL, LVCMOS12, LVCMOS15, LVCMOS18, LVCMOS25, or LVCMOS33 interface I/O standard.

You cannot change the LVCMOS drive strengths for Virtex-E devices. Only the variable LVTTL drive strengths are available for Spartan-IIE and Virtex-E devices.

DRIVE Propagation Rules

DRIVE is illegal when attached to a net or signal, except when the net or signal is connected to a pad. In this case, DRIVE is treated as attached to the pad instance. When attached to a design element, DRIVE is propagated to all applicable elements in the hierarchy below the design element.

DRIVE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid IOB output component
- Attribute Name: DRIVE
- Attribute Values: see "UCF and NCF Syntax Example" in this chapter

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute drive: string;
```

Specify the VHDL constraint as follows:

```
attribute drive of {component_name|entity_name|label_name}:
{component|entity|label} is "value";
```

See the "UCF and NCF Syntax Example" section in this chapter for valid *values*. For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* DRIVE = "value" *)
```

See the "UCF and NCF Syntax Example" section in this chapter for valid *values*. For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

IOB Output Components (UCF)

For Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

```
INST "instance_name" DRIVE={2|4|6|8|12|16|24};
```

where

- 12 mA is the default

SelectIO Output Components (IOBUF_SelectIO, OBUF_SelectIO, and OBUFT_SelectIO)

- For the LVTTL standard with Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

  ```
  INST "instance_name" DRIVE={2|4|6|8|12|16|24};
  ```

- For the LVCMOS12, LVCMOS15, and LVCMOS18 standards with Spartan-3, Spartan-3A, Spartan-3E,Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

  ```
  INST "instance_name" DRIVE={2|4|6|8|12|16};
  ```

- For the LVCMOS25 and LVCMOS33 standards with Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

  INST "*instance_name***" DRIVE={2|4|6|8|12|16|24};**

  where

  - 12 mA is the default for all architectures

XCF Syntax Example

```
MODEL "entity_name" drive={2|4|6|8|12|16|24};

BEGIN MODEL "entity_name"
 NET "signal_name" drive={2|4|6|8|12|16|24};
END;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window:

1. Double-click **Create Timing Constraints** under **User Constraints**.
2. In the **Ports** tab grid with I/O **Configuration Options** checked, click the **DRIVE** column in the row with the desired output port name.
3. Choose a value from the drop-down list.

Drop Specifications (DROP_SPEC)

DROP_SPEC Architecture Support

The DROP_SPEC constraint applies to all FPGA and CPLD devices.

DROP_SPEC Applicable Elements

Timing constraints

DROP_SPEC Description

DROP_SPEC is an advanced timing constraint. It allows you to specify that a timing constraint defined in the input design should be dropped from the analysis. You can use DROP_SPEC when new specifications defined in a constraints file do not directly override all specifications defined in the input design, and some of these input design specifications need to be dropped. While this timing command is not expected to be used frequently in an input netlist (or NCF file), it is legal. If defined in an input design DROP_SPEC must be attached to TIMESPEC.

DROP_SPEC Propagation Rules

It is illegal to attach DROP_SPEC to nets or macros. DROP_SPEC removes a specified timing specification.

DROP_SPEC Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
TIMESPEC "TSidentifier"=DROP_SPEC;
```

where

- TS*identifier* is the identifier name used for the timing specification that is to be removed

The following statement cancels the input design specification TS67.

```
TIMESPEC "TS67"=DROP_SPEC;
```

PCF Syntax Example

```
"TSidentifier" DROP_SPEC;
```

Enable (ENABLE)

ENABLE Architecture Support

The ENABLE constraint applies to FPGA devices only.

ENABLE Applicable Elements

Global in constraints file

ENABLE Description

ENABLE is an advanced timing constraint. It controls the types of paths analyzed during static timing. See also "Disable (DISABLE)."

ENABLE Propagation Rules

Enables timing analysis for specified path delays

ENABLE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

ENABLE can be applied only to a global timespec. The path tracing syntax is as follows in the UCF file.

```
ENABLE= delay_symbol_name;
```

where

- *delay_symbol_name* is the name of one of the standard block delay symbols for path tracing symbols shown in the following table, or a specific delay name defined in the datasheet

Table 23-1: **Standard Block Delay Symbols for Path Tracing**

Delay Symbol Name	Path Type	Default
reg_sr_o	Asynchronous Set/Reset to output propagation delay	Disabled
reg_sr_r	Asynchronous Set/Reset to recovery path	Disabled
reg_sr_clk	Synchronous Set/Reset to clock setup and hold checks	Enabled
lat_d_q	Data to output transparent latch delay	Disabled
ram_we_o	RAM write enable to output propagation delay	Enabled
tbuf_t_o	TBUF 3-state to output propagation delay	Enabled
tbuf_i_o	TBUF input to output propagation delay	Enabled
io_pad_i	IO pad to input propagation delay	Enabled
io_t_pad	IO 3-state to pad propagation delay	Enabled

Table 23-1: **Standard Block Delay Symbols for Path Tracing**

Delay Symbol Name	Path Type	Default
io_o_1	IO output to input propagation delay. Disabled for 3-stated IOBs	Enabled
io_o_pad	IO output to pad propagation delay	Enabled

PCF Syntax Example

```
ENABLE=delay_symbol_name;
```

or

```
TIMEGRP name ENABLE=delay_symbol_name;
```

Enable Suspend (ENABLE_SUSPEND)

ENABLE_SUSPEND Architecture Support

The ENABLE_SUSPEND constraint applies to Spartan™-3A devices only.

ENABLE_SUSPEND Applicable Elements

The ENABLE_SUSPEND attribute is a global attribute for the Spartan-3A device and is not attached to any particular element.

ENABLE_SUSPEND Description

The ENABLE_SUSPEND constraint is used to define the behavior of the SUSPEND power-reduction mode for the Spartan-3A device family. The acceptable values for this constraint are NO, FILTERED or UNFILTERED where NO disables this feature, FILTERED activates the suspend feature with the glitch filter being activated (requires longer pulse width to activate), and UNFILTERED activates the feature with the filter bypassed (quicker activation of SUSPEND). The default for this constraint, if not specified, is NO.

ENABLE_SUSPEND Propagation Rules

ENABLE_SUSPEND is a global attribute that is attached to the entire design.

ENABLE_SUSPEND Syntax Examples

The following is the syntax example using the constraint with particular tools or methods. The only syntax example supported is the UCF. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG ENABLE_SUSPEND="value";
```

where

- *value* is NO, FILTERED or UNFILTERED.

Example:

```
CONFIG ENABLE_SUSPEND="FILTERED";
```

Fast (FAST)

FAST Architecture Support

The FAST constraint applies to all FPGA and CPLD devices.

FAST Applicable Elements

- Output primitives
- Output pads
- Bidirectional pads

You can also attach FAST to the net connected to the pad component in a UCF file. NGDBuild transfers the constraint from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following syntax:

```
NET "net_name" FAST;
```

FAST Description

FAST is a basic mapping constraint. It increases the speed of an IOB output. While FAST produces a faster output, it may increase noise and power consumption.

FAST Propagation Rules

FAST is illegal when attached to a net except when the net is connected to a pad. In this instance, FAST is treated as attached to the pad instance. When attached to a macro, module, or entity, FAST is propagated to all applicable elements in the hierarchy below the module.

FAST Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: FAST
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute FAST: string;
```

Specify the VHDL constraint as follows:

```
attribute FAST of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* FAST = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'FAST mysignal';
```

UCF and NCF Syntax Example

The following statement increases the output speed of the element y2:

```
INST "$1I87/y2" FAST;
```

The following statement increases the output speed of the pad to which net1 is connected:

```
NET "net1" FAST;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" fast={TRUE|FALSE};
END;
```

Feedback (FEEDBACK)

FEEDBACK Architecture Support

The FEEDBACK constraint applies to FPGA devices only.

FEEDBACK Applicable Elements

Not applicable.

FEEDBACK Description

The FEEDBACK constraint is associated with the DCM. The constraint specifies the external path delay that occurs when a DCM output drives off-chip and then back on-chip into the DCM CLKFB input. This data is required for the timing tools to properly analyze the path clocked for the DCM.

The basic UCF syntax is:

```
NET feedback_signal FEEDBACK = value units NET output_signal;
```

The FEEDBACK signal is the net that drives the CLKFB input of the DCM and the output signal is the net that drives the output pad. The *value* provides the path delay from the output pad to the input pad. If *units* are not specified, then ns is assumed.

FEEDBACK Propagation Rules

Both the *feedback_signal* and *output_signal* must correspond to pad nets. If attached to any other net, an error results. The *feedback_signal* must be an input pad and *output_signal* must be an output pad.

FEEDBACK Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

The basic UCF syntax is:

```
NET feedback_signal FEEDBACK =value units NEToutput_signal;
```

where

- *feedback_signal* is the name of the input pad net used as the feedback to the DCM
- *value* is the board trace delay calculated or measured by you
- *units* is either ns or ps. The default is ns.
- *output_signal* is the name of the output pad net driven by the DCM

XCF Syntax Example

```
BEGIN MODEL "entity_name"
NET feedback_signal FEEDBACK = value units NET output_signal;
END;
```

For a description of *feedback_signal*, *value*, *units*, and *output_signal*, see "UCF Syntax Example" in this chapter.

PCF Syntax Example

{**BEL** | **COMP**} *feedback_signal_pad* **FEEDBACK =** *value units* {**BEL** | **COMP**} *output_signal*;

File (FILE)

FILE Architecture Support

The FILE constraint applies to all FPGA and CPLD devices.

FILE Applicable Elements

Instance declaration where the definition is defined in the specified file.

FILE Description

When you instantiate a module that resides in another netlist, ngdbuild finds this file by looking it up by the file name. This requires the netlist to have the same name as a module that is defined in the file. If you want to name the netlist differently than the module name, the FILE constraint can be attached to a instance declaration. This tells ngdbuild to look for the module in the file specified.

Some Xilinx® constraints cannot be used in attributes, because they are also VHDL keywords. To avoid this problem, use a constraint alias. Starting from the ISE™ 7.1 release, each constraint has its own alias. The alias name is based on the original constraint name with a "XIL" prefix. For example, the FILE constraint cannot be used in attributes directly. You must use "XIL_FILE" instead. The existing XILFILE alias is still supported.

FILE Propagation Rules

Applicable only on instances

FILE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: FILE
- Attribute Values: *file_name*.*extension*

where

- *file_name* is the name of a file that represents the underlying logic for the element carrying the constraint

Example file types include EDIF, EDN, NGC, and NMC.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute xilfile: string;
```

Specify the VHDL constraint as follows:

```
attribute xilfile of {instance_name|component_name} : {label|component}
is "file_name";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* XIL_FILE = "file_name" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

```
INST <instance definition> FILE= <filename definition is located in>;
```

Note: No valid syntax for UCF.

Float (FLOAT)

FLOAT Architecture Support

The FLOAT constraint applies to Coolrunner™ devices only.

FLOAT Applicable Elements

Applies to nets or pins.

FLOAT Description

FLOAT is a basic mapping constraint. It allows 3-stated pads to float when not being driven. This is useful when the default termination for applicable I/Os is set to PULLUP, PULLDOWN, or KEEPER in Project Navigator.

FLOAT Propagation Rules

Applies to the net or pin to which it is attached.

FLOAT Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic

- Attach to a valid instance
- Attribute Name: FLOAT
- Attribute Value: None required. TRUE, FALSE. If attached, TRUE is assumed.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute FLOAT: string;
```

Specify the VHDL constraint as follows:

```
attribute FLOAT of signal_name : signal is "{TRUE|FALSE}";
```

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* FLOAT = "{TRUE|FALSE}" *)
```

ABEL Syntax Example

```
XILINX PROPERTY 'FLOAT signal_name';
```

UCF and NCF Syntax Example

The basic UCF syntax is:

```
NET "signal_name" FLOAT;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
   NET "signal_name" FLOAT;
END;
```

From Thru To (FROM-THRU-TO)

FROM-THRU-TO Architecture Support

The FROM-THRU-TO constraint applies to FPGA devices only.

FROM-THRU-TO Applicable Elements

Predefined and user-defined groups

FROM-THRU-TO Description

FROM-THRU-TO is an advanced timing constraint, and is associated with the Period constraint of the high or low time. From synchronous paths, a FROM-TO-THRU constraint controls only the setup path, not the hold path. This constraint applies to a specific path that begins at a source group, passes through intermediate points, and ends at a destination group. The source and destination groups can be either user or predefined groups. You must define an intermediate path using TPTHRU before using THRU.

FROM-THRU-TO Propagation Rules

Applies to the specified FROM-THRU-TO path only.

FROM-THRU-TO Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
TIMESPEC "TSidentifier"=FROM "source_group" THRU
"thru_pt1"...[THRU"thru_pt2"...] TO "destination_group" value
[Units]{DATAPATHONLY};
```

where

- *identifier* can consist of characters or underbars
- *source_group* and *destination_group* are user-defined or predefined groups
- *thru_pt1* and *thru_pt2* are intermediate points to define specific paths for timing analysis
- *value* is the delay time
- *units* can be ps, ms, ns, or micro

The DATAPATHONLY keyword indicates that the FROM-TO constraint does not take clock skew or phase information into consideration. This keyword results in only the data path between the groups being constrained and analyzed.

```
TIMESPEC TS_MY_PathB = FROM "my_src_grp" THRU "my_thru_pt" TO
"my_dst_grp" 13.5 ns DATAPATHONLY;
```

FROM or TO is optional. You can have just a FROM or just a TO.

You are not required to have a FROM, THRU, and TO. You can basically have any combination (FROM-TO, FROM-THRU-TO, THRU-TO, TO, FROM, FROM-THRU-THRUTHRU-TO, FROM-THRU, and so on). There is no restriction on the number of

THRU points. The source, THRU points, and destination can be a net, bel, comp, macro, pin, or timegroup.

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

1. Identify the through points using the Create ... Timing THRU Points button from the Advanced tab.
2. Set a FROM-THRU-TO constraint for groups of elements in the Advanced tab by clicking Specify next to “Slow/Fast Path Exceptions” (to set explicit times) or Specify next to “Multi-Cycle Paths“ (to set times relative to other time specifications).
3. Fill out the FROM/THRU/TO dialog box.

PCF Syntax Example

```
TSname=MAXDELAY FROM TIMEGRP "source" THRU TIMEGRP "thru_pt1" ...THRU
"thru_ptn" TO TIMEGRP "destination" {DATAPATHONLY};
```

You are not required to have a FROM, THRU, and TO. You can have almost any combination (such as FROM-TO, FROM-THRU-TO, THRU-TO, TO, FROM, FROM-THRU-THRU-THRU-TO, and FROM-THRU). There is no restriction on the number of THRU points. The source, THRU points, and destination can be a net, bel, comp, macro, pin, or timegroup.

From To (FROM-TO)

FROM-TO Architecture Support

The FROM-TO constraint applies to all FPGA and CPLD devices.

FROM-TO Applicable Elements

Predefined and user-defined groups

FROM-TO Description

FROM-TO defines a timing constraint between two groups. It is associated with the Period constraint of the high or low time. A group can be user-defined or predefined. From synchronous paths, a FROM-TO constraint controls only the setup path, not the hold path.

For Virtex5, the FROM-TO constraint controls both setup and hold paths.

FROM-TO Propagation Rules

Applies to a path specified between two groups.

FROM-TO Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

TIMESPEC TS*name***=FROM "***group1***" TO "***group2***"** *value* **{DATAPATHONLY};**

where

- TSname must always begin with "TS". Any alphanumeric character or underscore may follow.
- *group1* is the origin path
- *group2* is the destination path
- *value* iS ns by default. Other possible values are MHz or another timing specification such as TS_C2S/2 or TS_C2S*2.

The DATAPATHONLY keyword indicates that the FROM-TO constraint does not take clock skew or phase information into consideration. This keyword results in only the data path between the groups being constrained and analyzed.

```
TIMESPEC TS_MY_PathA = FROM "my_src_grp" TO "my_dst_grp" 23.5 ns
DATAPATHONLY;
```

XCF Syntax Example

XST supports the FROM-TO constraint with the following limitations:

- FROM-THRU-TO is not supported
- Linked Specification is not supported
- Pattern matching for predefined groups is not supported:

```
TIMESPEC TS_1 = FROM FFS(machine/*) TO FFS 2 ns;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Specify next to “Slow/Fast Path Exceptions” (to set explicit times) or Specify next to “Multi-Cycle Paths“ (to set times relative to other time specifications) and then fill out the FROM/THRU/TO dialog box.

PCF Syntax Example

```
TSname=MAXDELAY FROM TIMEGRP "group1" TO TIMEGRP "group2" value
{DATAPATHONLY};
```

You are not required to have a FROM, THRU, and TO. You can have almost any combination (such as FROM-TO, FROM-THRU-TO, THRU-TO, TO, FROM, FROM-THRU-THRU-THRU-TO, and FROM-THRU). There is no restriction on the number of thru points. The source, thru points, and destination can be a net, bel, comp, macro, pin, or timegroup.

Hierarchical Block Name (HBLKNM)

HBLKNM Architecture Support

The HBLKNM constraint applies to FPGA devices only.

HBLKNM Applicable Elements

If "Yes" is shown next to the device name in the Architecture Support table, the constraint may be used with that device in one or more of the following design elements, or categories of design elements. Not all device families support all these elements. To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

1. Registers
2. I/O elements and pads
3. FMAP
4. BUFT
5. PULLUP
6. ACLK, GCLK
7. BUFG
8. BUFGS, BUFGP
9. ROM
10. RAMS and RAMD
11. Carry logic primitives

You can also attach HBLKNM to the net connected to the pad component in a UCF file. NGDBuild transfers the constraint from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following syntax:

```
NET "net_name" HBLKNM=property_value;
```

HBLKNM Description

HBLKNM is an advanced mapping constraint. It assigns hierarchical block names to logic elements and controls grouping in a flattened hierarchical design. When elements on different levels of a hierarchical design carry the same block name, and the design is flattened, NGDBuild prefixes a hierarchical path name to the HBLKNM value.

Like Block Name, HBLKNM forces function generators and flip-flops into the same CLB. Symbols with the same HBLKNM constraint map into the same CLB, if possible.

However, using HBLKNM instead of Block Name has the advantage of adding hierarchy path names during translation, and therefore the same HBLKNM constraint and value can be used on elements within different instances of the same design element.

HBLKNM Propagation Rules

When attached to a design element, HBLKNM is propagated to all applicable elements in the hierarchy within the design element.

HBLKNM Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: HBLKNM
- Attribute Values: *block_name*

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute hblknm: string;
```

Specify the VHDL constraint as follows:

```
attribute hblknm of
{entity_name|component_name|signal_name|label_name}:
{entity|component|signal|label} is "block_name";
```

where

- *block_name* is a valid block name for that type of symbol

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* HBLKNM = "block_name" *)
```

where

- *block_name* is a valid block name for that type of symbol

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Examples

The basic UCF syntax is:

```
NET "net_name" HBLKNM=property_value;
INST "instance_name" HBLKNM=block_name;
```

where

- *block_name* is a valid block name for that type of symbol

The following statement specifies that the element *this_fmap* is put into the block named group1.

```
INST "$I13245/this_fmap" HBLKNM=group1;
```

The following statement attaches HBLKNM to the pad connected to net1.

```
NET "net1" HBLKNM=$COMP_0;
```

Elements with the same HBLKNM are placed in the same logic block if possible. Otherwise an error occurs. Conversely, elements with different block names are not put into the same block.

Hierarchical Lookup Table Name (HLUTNM)

HLUTNM Architecture Support

The HLUTNM constraint applies to Virtex™-5 devices only.

HLUTNM Applicable Elements

The HLUTNM constraint can be applied to two symbols that share a common hierarchy and that are also unique within their level of hierarchy. The constraint can be applied to two 5-input or smaller function generator symbols (LUT, ROM, or RAM) if the total number of unique input pins required for both symbols does not exceed 5 pins. The constraint can be applied to a 6-input read-only function generator symbol (LUT6, ROM64) in conjunction with a 5-input read-only symbol (LUT5, ROM32) if the total number of unique input pins required for both symbols does not exceed 6 inputs and the lower 32 bits of the 6-input symbol programming matches all 32 bits of the 5-input symbol programming.

HLUTNM Description

The HLUTNM constraint provides the ability to control the grouping of logical symbols into the LUT sites of the Virtex-5 FPGA architectures. The HLUTNM constraint is a string value property that is applied to two qualified symbols. The HLUTNM constraint value must be applied uniquely to two symbols within a given level of hierarchy. These two symbols will be implemented in a shared LUT site within a SLICE component.

This constraint is functionally similar to the Hierarchical Block Name (HBLKNM) constraint.

HLUTNM Propagation Rules

The HLUTNM constraint can be applied to two symbols that share a common hierarchy and that are also unique within their level of hierarchy.

HLUTNM Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid element or symbol type
- Attribute Name: HLUTNM
- Attribute Values: <user_defined>

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute hlutnm: string;
```

Specify the VHDL constraint as follows:

```
attribute hlutnm of {instance_name}: label is "string_value";
```

where

- *instance_name* is the instance name of an instantiated LUT, or LUTRAM.
- *string_value* is a value that is applied uniquely to two symbols within a given level of hierarchy. No default value exists. A blank value means the constraint is ignored.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* HLUTNM = "string_value" *)
```

where

- *string_value* is a value that is applied uniquely to two symbols within a given level of hierarchy. No default value exists. A blank value means the constraint is ignored.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

Not supported

UCF and NCF Syntax Example

```
INST "symbol_name" HLUTNM=string_value;
```

where

- *string_value* is a value that is applied uniquely to two symbols within a given level of hierarchy. No default value exists. A blank value means the constraint is ignored.

XCF Syntax Example

```
MODEL "symbol_name" hlutnm = string_value;
```

Constraints Editor Syntax Example

Not supported

PCF Syntax Example

Not supported

Floorplanner Syntax Example

Not supported

PACE Syntax Example

Not supported

Floorplan Editor Syntax Example

Not supported

FPGA Editor Syntax Example

Not supported

Project Navigator Syntax Example

Not supported

HU_SET

HU_SET Architecture Support

The HU_SET constraint applies to FPGA devices only.

HU_SET Applicable Elements

If "Yes" is shown next to the device name in the Architecture Support table, the constraint may be used with that device in one or more of the following design elements, or categories of design elements. Not all device families support all these elements. To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

1. Registers
2. FMAP
3. Macro Instance
4. ROM
5. RAMS, RAMD
6. BUFT
7. MULT18X18S
8. RAMB4_Sm_Sn, RAMB4_Sn
9. RAMB16_*Sm_Sn*, *RAMB16_Sn*
10. RAMB16
11. DSP48

HU_SET Description

HU_SET is an advanced mapping constraint. It is defined by the design hierarchy. However, it also allows you to specify a set name. It is possible to have only one H_SET within a given hierarchical element but by specifying set names, you can specify several HU_SET sets.

NGDBuild hierarchically qualifies the name of the HU_SET as it flattens the design and attaches the hierarchical names as prefixes.

The differences between an HU_SET constraint and an H_SET constraint include:

HU_SET	H_SET
Has an explicit user-defined and hierarchically qualified name for the set	Has only an implicit hierarchically qualified name generated by the design-flattening program
"Starts" with the symbols that are assigned the HU_SET constraint	"Starts" with the instantiating macro one level above the symbols with the RLOC constraints

For background information about using the various set attributes, see "RLOC Description" in the "Relative Location (RLOC)" constraint.

HU_SET Propagation Rules

HU_SET is a design element constraint. Any attachment to a net is illegal.

HU_SET Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: HU_SET
- Attribute Values: *set_name*

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute HU_SET: string;
```

Specify the VHDL constraint as follows:

```
attribute HU_SET of {component_name|entity_name|label_name}:
{component|entity|label} is "set_name";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* HU_SET = "set_name" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" HU_SET=set_name;
```

where

- *set_name* is the identifier for the set

The variable *set_name* must be unique among all the sets in the design.

The following statement assigns an instance of the register FF_1 to a set named heavy_set.

```
INST "$1I3245/FF_1" HU_SET=heavy_set;
```

XCF Syntax Example

```
MODEL "entity_name" hu_set={yes|no};

BEGIN MODEL "entity_name"
 INST "instance_name" hu_set=yes;
END;
```

Input Buffer Delay Value (IBUF_DELAY_VALUE)

IBUF_DELAY_VALUE Architecture Support

The IBUF_DELAY_VALUE constraint applies to the following devices:

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

IBUF_DELAY_VALUE Applicable Elements

Any top-level I/O port.

IBUF_DELAY_VALUE Description

The IBUF_DELAY_VALUE constraint is a mapping constraint that adds additional static delay to the input path of the FPGA array. This constraint can be applied to any input or bi-directional signal that is not directly driving a clock or IOB (Input Output Block) register. For more information regarding the constraint of signals driving clock and IOB registers, see the "IFD_DELAY_VALUE" constraint. The IBUF_DELAY_VALUE constraint can be set to an integer value from 0-16. The value 0 is the default value, and applies no additional delay to the input path. A larger value for this constraint correlates to a larger delay added to input path. These values do not directly correlate to a unit of time but rather additional buffer delay. For more information, see the product data sheets.

IBUF_DELAY_VALUE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach a new property to the top-level port of the schematic
- Attribute Name-IBUF_DELAY_VALUE
- Attribute Values: 0-16

VHDL Syntax Example

Attach a VHDL attribute to the appropriate top-level port

```
attribute IBUF_DELAY_VALUE : string;
attribute IBUF_DELAY_VALUE of top_level_port_name: signal is "value";
```

where

- a valid *value* is from 0 to 16.

The following statement assigns an IBUF_DELAY_VALUE increment of 5 to the net DataIn1

```
attribute IBUF_DELAY_VALUE : string;
attribute IBUF_DELAY_VALUE of DataIn1: label is "5";
```

Verilog Syntax Example

Attach a Verilog attribute to the appropriate top-level port

```
(* IBUF_DELAY_VALUE="value" *) input top_level_port_name;
```

where

- a valid *value* is from 0 to 16.

The following statement assigns an IBUF_DELAY_VALUE increment of 5 to the net DataIn1

```
(* IBUF_DELAY_VALUE="5" *) input DataIn1;
```

UCF and NCF Syntax Example

The basic UCF syntax is:

```
NET "top_level_port_name" IBUF_DELAY_VALUE = value;
```

where

- *value* is the numerical IBUF delay setting. A valid *value* is from 0 to 16.

The following statement assigns an IBUF_DELAY_VALUE increment of 5 to the net DataIn1

```
NET "DataIn1" IBUF_DELAY_VALUE = 5;
```

IFD_DELAY_VALUE

IFD_DELAY_VALUE Architecture Support

The IFD_DELAY_VALUE constraint applies tothe following devices:

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

IFD_DELAY_VALUE Applicable Elements

Any top-level I/O port

IFD_DELAY_VALUE Description

The IFD_DELAY_VALUE constraint is a mapping constraint that adds additional static delay to the input path of the FPGA array. This constraint can be applied to any input or bi-directional signal which drives an IOB (Input Output Block) register. For more information on the constraint of signals which do not drive IOB registers, see the "Input Buffer Delay Value (IBUF_DELAY_VALUE)" constraint.

The IFD_DELAY_VALUE constraint can be set to an integer value from 0-8, and as AUTO. The value AUTO is the default value, and is used to guarantee that the input hold time of the destination register is met by automatically adding the appropriate amount of delay to the data path.

When the IFD_DELAY_VALUE constraint is set to 0, the data path has no additional delay added. The integers 1-8 correspond to increasing amounts of delay added to the data path. These values do not directly correlate to a unit of time but rather additional buffer delay. For more information, see the product data sheets.

IFD_DELAY_VALUE Propagation Rules

Although IFD_DELAY_VALUE is attached to an I/O symbol, it applies to the entire I/O component.

IFD_DELAY_VALUE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name-IFD_DELAY_VALUE
- Attribute Values-0-8, AUTO

VHDL Syntax Example

Attach a VHDL attribute to the appropriate top-level port

```
attribute IFD_DELAY_VALUE : string;
attribute IFD_DELAY_VALUE of top_level_port_name: signal is "value";
```

The following statement assigns an IFD_DELAY_VALUE increment of 5 to the net DataIn1

```
attribute IFD_DELAY_VALUE : string;
attribute IFD_DELAY_VALUE of DataIn1: label is "5";
```

Verilog Syntax Example

Attach a Verilog attribute to the appropriate top-level port

```
(* IFD_DELAY_VALUE="value" *) input top_level_port_name;
```

The following statement assigns an IFD_DELAY_VALUE increment of 5 to the net DataIn1

```
(* IFD_DELAY_VALUE="5" *) input DataIn1;
```

UCF and NCF Syntax Example

The basic UCF syntax is:

```
NET "top_level_port_name" IFD_DELAY_VALUE = value;
```

where

- *value* is the numerical IBUF delay setting

The following statement assigns an IFD_DELAY_VALUE increment of 5 to the net DataIn1

```
NET "DataIn1" IFD_DELAY_VALUE = 5;
```

Input Registers (INREG)

INREG Architecture Support

The INREG constraint applies to Coolrunner™ devices only.

INREG Applicable Elements

Applies to register and latch instances with their D-inputs driven by input pads or to the Q-output nets of such registers or latches.

INREG Description

This constraint applies to register and latch instances with their D-inputs driven by input pads, or to the Q-output nets of such registers and latches. By default, registers and latches in a CoolRunner XPLA3 or CoolRunner-II design that have their D-inputs driven by input pads are automatically implemented using the device's Fast Input path, where possible. If you disable the Project Navigator property Use Fast Input for INREG for the Fit (Implement Design) process, then only register and latches with the INREG attribute are considered for Fast Input optimization.

INREG Propagation Rules

Applies to register or latch to which it is attached or to the Q-output nets of such registers or latches

INREG Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a register, latch, or net
- Attribute Name: INREG
- Attribute Values: None (TRUE by default)

ABEL Syntax Example

```
XILINX PROPERTY 'inreg signal_name';
```

UCF Syntax Example

```
NET "signal_name" INREG;
INST "register_name" INREG;
```

IOB

IOB Architecture Support

The IOB constraint applies to FPGA devices only.

IOB Applicable Elements

- Non-INFF/OUTFF flip-flop and latch primitives
- Registers

IOB Description

IOB is a basic mapping and synthesis constraint. It indicates which flip-flops and latches can be moved into the IOB. The mapper supports a command line option (-pr i | o | b) that allows flip-flop or latch primitives to be pushed into the input IOB (i), output IOB (o), or input/output IOB (b) on a global scale. The IOB constraint, when associated with a flip-flop or latch, tells the mapper to pack that instance into an IOB type component if possible. The IOB constraint has precedence over the mapper **-pr** command line option, however, IOB constraints do not have precedence over LOC constraints.

XST considers the IOB constraint as an implementation constraint, and therefore propagates it in the generated NGC file. XST also duplicates the flip-flops and latches driving the Enable pin of output buffers, so that the corresponding flip-flops and latches can be packed in the IOB.

IOB Propagation Rules

Applies to the design element to which it is attached

IOB Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a flip-flop or latch instance or to a register
- Attribute Name: IOB
- Attribute Values: TRUE, FALSE, AUTO

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute iob: string;
```

Specify the VHDL constraint as follows:

```
attribute iob of {component_name|entity_name|label_name}:
{component|entity|label} is "{TRUE|FALSE|AUTO}";
```

where

- **TRUE** allows the flip-flop or latch to be pulled into an IOB

- **FALSE** indicates not to pull it into an IOB
- **AUTO** is used by XST only. XST takes into account timing constraints and automatically decides to push or not to push flip-flops into IOBs

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* IOB = "{TRUE|FALSE|AUTO}" *)
```

See value definitions in "VHDL Syntax Example" above.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic syntax is:

```
INST "instance_name" IOB={TRUE|FALSE};
```

where

- **TRUE** allows the flip-flop or latch to be pulled into an IOB
- **FALSE** indicates not to pull it into an IOB

The following statement instructs the mapper from placing the foo/bar instance into an IOB component.

```
INST "foo/bar" IOB=TRUE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" iob={true|false|auto};
 INST "instance_name" iob={true|false|auto};
END;
```

Note: For the **AUTO** option, XST takes into account timing constraints and automatically decides to push or not to push flip-flops into IOBs

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Misc tab, click Specify next to "Registers to be placed in IOBs" and move the desired register to the Registers for IOB packing list. This sets the IOB constraint to TRUE.

Project Navigator Syntax Example

Define globally with the Pack I/O Registers into IOBs option in the Xilinx Specific Options tab of the Process Properties dialog box in Project Navigator. YES maps to TRUE. NO maps to FALSE. With a design selected in the Sources window, right-click Synthesize in the Processes window to access the Process Properties dialog box.

Input Output Block Delay (IOBDELAY)

IOBDELAY Architecture Support

The IOBDELAY constraint applies to FPGA devices only.

IOBDELAY Applicable Elements

Any I/O symbol (I/O pads, I/O buffers, or input pad nets)

IOBDELAY Description

IOBDELAY is a basic mapping constraint. It specifies how the input path delay elements in Spartan™-II, Spartan-IIE, Spartan-3, Virtex™, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices are to be programmed.

There are two possible destinations for input signals: the local IOB input FF or a load external to the IOB. Spartan-II, Spartan-3, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices allow a delay element to delay the signal going to one or both of these destinations.

IOBDELAY cannot be used concurrently with "No Delay (NODELAY)."

Note: IOBDELAY_TYPE and IOBDELAY_VALUE are library component attributes and therefor are not documented in this guide. Details on these two attributes can be found within the descriptions of the IDELAY, IODELAY, IDELAYCTRL and ISERDES components in the Libraries Guide for Virtex-4 and Virtex-5.

IOBDELAY Propagation Rules

Although IOBDELAY is attached to an I/O symbol, it applies to the entire I/O component.

IOBDELAY Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an I/O symbol
- Attribute Name: IOBDELAY
- Attribute Values: NONE, BOTH, IBUF, IFD

Note: For the Spartan-3 family, the default is not set to NONE so the device can achieve a zero hold time.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute iobdelay: string;
```

Specify the VHDL constraint as follows:

```
attribute iobdelay of {component_name|label_name}: {component|label} is
"{NONE|BOTH|IBUF|IFD}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* IOBDELAY = {NONE|BOTH|IBUF|IFD} *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" IOBDELAY={NONE|BOTH|IBUF|IFD};
```

where

- NONE sets the delay OFF for both the IBUF and IFD paths.

Note: For the Spartan-3 family, the default is not set to NONE so the device can achieve a zero hold time.

- BOTH sets the delay ON for both the IBUF and IFD paths.
- IBUF sets the delay to OFF for any register inside the I/O component and to ON for the registers outside of the component if the input buffer drives a register D pin outside of the I/O component.
- IFD sets the delay to ON for any register inside the I/O component and to OFF for the registers outside the component if a register occupies the input side of the I/O component, regardless of whether the register has the IOB=TRUE constraint.

The following statement sets the delay OFF for the IBUF and IFD paths.

```
INST "xyzzy" IOBDELAY=NONE.
```

Input Output Standard (IOSTANDARD)

IOSTANDARD Architecture Support

The IOSTANDARD constraint applies to all FPGA and CPLD devices except Coolrunner™ XPLA3.

IOSTANDARD Applicable Elements

If "Yes" is shown next to the device name in the Architecture Support table, the constraint may be used with that device in one or more of the following design elements, or categories of design elements. Not all device families support all these elements. To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

- IBUF, IBUFG, OBUF, OBUFT
- IBUFDS, IBUFGDS, OBUFDS, OBUFTDS
- Output Voltage Banks

IOSTANDARD Description

IOSTANDARD is a basic mapping constraint and synthesis constraint.

IOSTANDARD for FPGA Devices

Use IOSTANDARD to assign an I/O standard to an I/O primitive.

All components with IOSTANDARD must follow the same placement rules (banking rules) as the SelectIO components. See the Xilinx *Libraries Guides* for information on the banking rules for each architecture. For descriptions of the supported I/O standards, see the device data sheet.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the recommended procedure is to attach IOSTANDARD to a buffer component instead of using the SelectIO variants of a component. For example, use an IBUF with the IOSTANDARD=HSTL_III constraint instead of the IBUF_HSTL_III component.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, differential signaling standards apply to IBUFDS, IBUFGDS, OBUFDS, and OBUFTDS only (not IBUF or OBUF).

IOSTANDARD for CPLD Devices

You can apply IOSTANDARD to I/O pads of CoolRunner-II devices to specify both input threshold and output VCCIO voltage. For supported values, see the device data sheet.

You can apply IOSTANDARD to outputs of XC9500XV devices to specify the VCCO voltage. The IOSTANDARD names supported by XC9500XV are:

- LVTTL (VCCO=3.3V)
- LVCMOS2 (VCCO=2.5V)
- X25TO18 (VCCO=1.8V)

The X25TO18 setting is provided for generating 1.8V compatible outputs from a CPLD normally operating in a 2.5V environment.

The CPLD fitter automatically groups outputs with compatible IOSTANDARD settings into the same bank when no location constraints are specified.

IOSTANDARD Propagation Rules

It is illegal to attach IOSTANDARD to a net or signal except when the signal or net is connected to a pad. In this case, IOSTANDARD is treated as attached to an IOB instance (IBUF, OBUF, IOB FF). When attached to a design element, IOSTANDARD propagates to all applicable elements in the hierarchy within the design element.

IOSTANDARD Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an I/O primitive
- Attribute Name: IOSTANDARD
- Attribute Values: *iostandard_name*

For more information, see "UCF and NCF Syntax Example" in this chapter.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute iostandard: string;
```

Specify the VHDL constraint as follows:

```
attribute iostandard of {component_name|label_name}: {component|label}
is "iostandard_name";
```

For more information about *iostandard_name*, see "UCF and NCF Syntax Example" in this chapter.

For CPLD devices you can also apply IOSTANDARD to the pad signal.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* IOSTANDARD = "iostandard_name" *)
```

For a description of *iostandard_name*, see the UCF section.

For CPLD devices you can also apply IOSTANDARD to the pad signal.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'iostandard=iostandard_name mysignal';
```

UCF and NCF Syntax Example

The basic syntax is:

```
INST "instance_name" IOSTANDARD=iostandard_name;
NET "pad_net_name" IOSTANDARD=iostandard_name;
```

where

- *iostandard_name* is an IO Standard name as specified in the device data sheet

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 INST "instance_name" iostandard=string;
 NET "signal_name" iostandard=string;
END;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Ports tab grid with the I/O Configuration Options checked, click the IOSTANDARD column in the row with the desired net name and choose a value from the drop-down list.

PACE Syntax Example

PACE is mainly used to assign location constraints to IOs. It can also be used to assign certain IO properties such as IO Standards. You can access PACE from the Processes window in the Project Navigator.

For more information, see the PACE help, especially the topics within Editing Pins and Areas in the Procedures section.

Keep (KEEP)

KEEP Architecture Support

The KEEP constraint applies to all FPGA and CPLD devices.

KEEP Applicable Elements

Signals

KEEP Description

KEEP is an advanced mapping constraint and synthesis 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.

KEEP is translated into an internal constraint known as NOMERGE when targeting an FPGA. Messaging from the implementation tools therefore refers to the system property NOMERGE, not KEEP.

KEEP Propagation Rules

Applies to the signal to which it is attached

KEEP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: KEEP
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute keep : string;
```

Specify the VHDL constraint as follows:

```
attribute keep of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* KEEP = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
mysignal NODE istype 'keep';
```

UCF and NCF Syntax Example

The following statement ensures that the net $SIG_0 remains visible.

```
NET "$1I3245/$SIG_0" KEEP;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" keep={yes|no|true|false};
END;
```

Keeper (KEEPER)

KEEPER Architecture Support

The KEEPER constraint applies to all FPGA devices and only Coolrunner™-II CPLDs.

KEEPER Applicable Elements

Tristate input/output pad nets

KEEPER Description

KEEPER is a basic mapping constraint. It retains the value of the output net it is attached to. For example, if logic 1 is being driven onto the net, KEEPER drives a weak/resistive 1 onto the net. If the net driver is then 3-stated, KEEPER continues to drive a weak/resistive 1 onto the net.

The KEEPER constraint must follow the same banking rules as the KEEPER component. For more information on banking rules, see the Xilinx *Libraries Guides*.

KEEPER, PULLUP, and PULLDOWN are only valid on pad NETs, not on INSTs of any kind.

For CoolRunner-II devices, the use of KEEPER and the use of PULLUP are mutually exclusive across the whole device.

KEEPER Propagation Rules

KEEPER is illegal when attached to a net or signal except when the net or signal is connected to a pad. In this case, KEEPER is treated as attached to the pad instance.

KEEPER Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic

- Attach to an output pad net
- Attribute Name: KEEPER
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute keeper: string;
```

Specify the VHDL constraint as follows:

```
attribute keeper of signal_name : signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

```
(* KEEPER = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'KEEPER mysignal';
```

UCF and NCF Syntax Example

This statement configures the IO to use KEEPER for a NET.

```
NET "pad_net_name" KEEPER;
```

This statement configures KEEPER to be used globally.

```
DEFAULT KEEPER = TRUE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" keeper={yes|no|true|false};
END;
```

Keep Hierarchy (KEEP_HIERARCHY)

KEEP_HIERARCHY Architecture Support

The KEEP_HIERARCHY constraint applies to all FPGA and CPLD devices.

KEEP_HIERARCHY Applicable Elements

KEEP_HIERARCHY is attached to logical blocks, including blocks of hierarchy or symbols.

KEEP_HIERARCHY Description

KEEP_HIERARCHY is a synthesis and implementation constraint. If hierarchy is maintained during Synthesis, the Implementation tools use this constraint to preserve the hierarchy throughout the implementation process and allow a simulation netlist to be created with the desired hierarchy.

XST may flatten the design to get better results by optimizing entity or module boundaries. You can set KEEP_HIERARCHY to **`true`** so that the generated netlist is hierarchical and respects the hierarchy and interface of any entity or module of your design.

This option is related to the hierarchical blocks (VHDL entities, Verilog modules) specified in the HDL design and does not concern the macros inferred by the HDL synthesizer. Three values are available for this option:

- **`true`**

  Allows the preservation of the design hierarchy, as described in the HDL project. If this value is applied to synthesis, it is also propagated to implementation.

- **`false`**

  Hierarchical blocks are merged in the top level module.

- **`soft`**

  Allows the preservation of the design hierarchy in synthesis, but the KEEP_HIERARCHY constraint is not propagated to implementation.

For CPLD devices, the default is **`true`**. For FPGA devices, the default is **`false`**.

Note: In XST, the KEEP_HIERARCHY constraint can be set to the following values: **`yes`**, **`true`**, **`no`**, **`false`**, and **`soft`**. When used at the command line, only **`yes`**, **`no`**, and **`soft`** are accepted.

In general, an HDL design is a collection of hierarchical blocks, and preserving the hierarchy gives the advantage of fast processing because the optimization is done on separate pieces of reduced complexity. Nevertheless, very often, merging the hierarchy blocks improves the fitting results (fewer PTerms and device macrocells, better frequency) because the optimization processes (collapsing, factorization) are applied globally on the entire logic.

The KEEP_HIERARCHY constraint enables or disables hierarchical flattening of user-defined design units. Allowed values are **`true`** and **`false`**. By default, the user hierarchy is preserved.

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. Also I1, I3, I6, I7 are flattened.

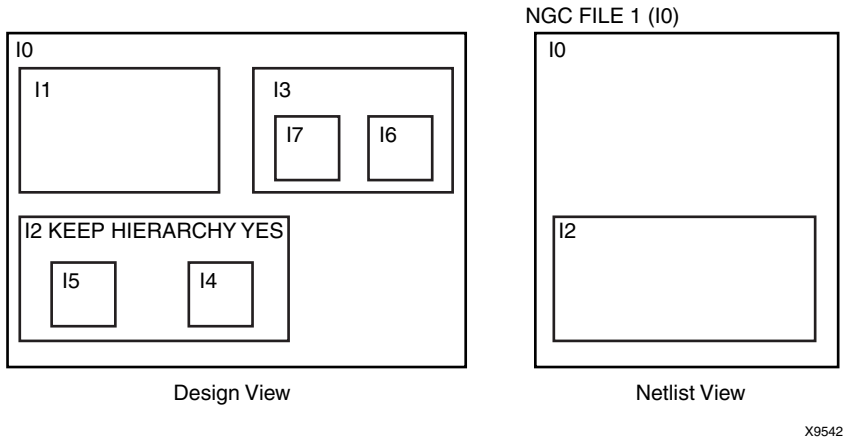

Figure 42-1: **KEEP_HIERARCHY EXAMPLE**

KEEP_HIERARCHY Propagation Rules

Applies to the entity or module to which it is attached

KEEP_HIERARCHY Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to the entity or module symbol
- Attribute Name: KEEP_HIERARCHY
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute keep_hierarchy : string;
```

Specify the VHDL constraint as follows:

```
attribute keep_hierarchy of architecture_name: architecture is
{TRUE|FALSE|SOFT};
```

The default is **false** for FPGA devices and **true** for CPLD devices.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* KEEP_HIERARCHY = "{TRUE|FALSE|SOFT}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

For instances:

```
INST "instance_name" KEEP_HIERARCHY={TRUE|FALSE|SOFT};
```

XCF Syntax Example

In XST, the KEEP_HIERARCHY constraint accepts the following values: **yes**, **true**, **no**, **false**, and **soft**. When KEEP_HIERARCHY is used as a command-line switch, only **yes**, **no**, and **soft** are accepted.

```
MODEL "entity_name" keep_hierarchy={yes|no|soft};
```

Project Navigator Syntax Example

Define globally with the Keep Hierarchy option in the Synthesis Options tab of the Process Properties dialog box within the Project Navigator. With a design selected in the Sources window, right-click Synthesize in the Processes window to access the Process Properties dialog box.

Location (LOC)

This section contains the following:

LOC Architecture Support

The LOC constraint applies to all FPGA and CPLD devices.

LOC Applicable Elements

To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

LOC Description

LOC is a basic placement constraint and a synthesis constraint.

LOC Description for FPGA Devices

LOC defines where a design element can be placed within an FPGA. It specifies the absolute placement of a design element on the FPGA die. It can be a single location, a range of locations, or a list of locations. You can specify LOC from the design file and also direct placement with statements in a constraints file.

To specify multiple locations for the same symbol, separate each location within the field using a comma. The comma specifies that the symbols can be placed in any of the specified locations. You can also specify an area in which to place a design element or group of design elements.

A convenient way to find legal site names is use the FPGA Editor, PACE, or Floorplanner. The legal names are a function of the target part type. To find the correct syntax for specifying a target location, load an empty part into the FPGA Editor (or look in Floorplanner). Place the cursor on any block, then click the block to display its location in the FPGA Editor history area. Do not include the pin name such as .I, .O, or .T as part of the location.

You can use LOC for logic that uses multiple CLBs, IOBs, soft macros, or other symbols. To do this, use LOC on a soft macro symbol, which passes the location information down to the logic on the lower level. The location restrictions are automatically applied to all blocks on the lower level for which LOCs are legal.

Spartan-II, Spartan-IIE, Virtex, and Virtex-E

The physical site specified in the location value is defined by the row and column numbers for the array, with an optional extension to define the slice for a given row/column location. A Spartan-II, Spartan-IIE, Virtex, Virtex-E slice is composed of:

- Two LUTs (which can be configured as RAM or shift registers)
- Two flip-flops (which can also be configured as latches)
- Two XORCYs
- Two MULT_ANDs
- One MUXF5
- One MUXF6
- One MUXCY

Only one MUXF6 can be used between the two adjacent slices in a specific row/column location. The two slices at a specific row/column location are adjacent to one another.

The block RAMs (RAMB4s) have a different row/column grid specification than the CLB and TBUFs. A block RAM located at RAMB4_R3C1 is not located at the same site as a flip-flop located at CLB_R3C1. Therefore, the location value must start with "CLB," "TBUF," or "RAMB4." The location cannot be shortened to reference only the row, column, and extension. The optional extension specifies the left-most or right-most slice for the row/column.

The location value for global buffers and DLL elements is the specific physical site name for available locations

Spartan-3 and Higher Devices

In the Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 CLBs, there are four slices, arranged vertically, per CLB with the bottom two slices on the left side of the CLB and the top two slices on the right side of the CLB. Each slice is equivalent and contains two function generators (F and G), two storage elements, arithmetic logic gates, large multiplexers, wide function capability, and two fast carry look-ahead chains.

The Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 architectures diverge from the traditional Row/Column/Slice designators on the CLB. Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices use a Cartesian-based XY designator at the slice level. The slice-based location specification uses the form: SLICE_X*m*Y*n*. The XY slice grid starts as X0Y0 in the lower left CLB tile of the chip. The X values start at 0 and increase horizontally to the right in the CLB row, with two different X values per CLB. The Y values start at 0 and

increase vertically up in the CLB column, with two different Y values per CLB. The XY slice numbering scheme is shown in the following figure.

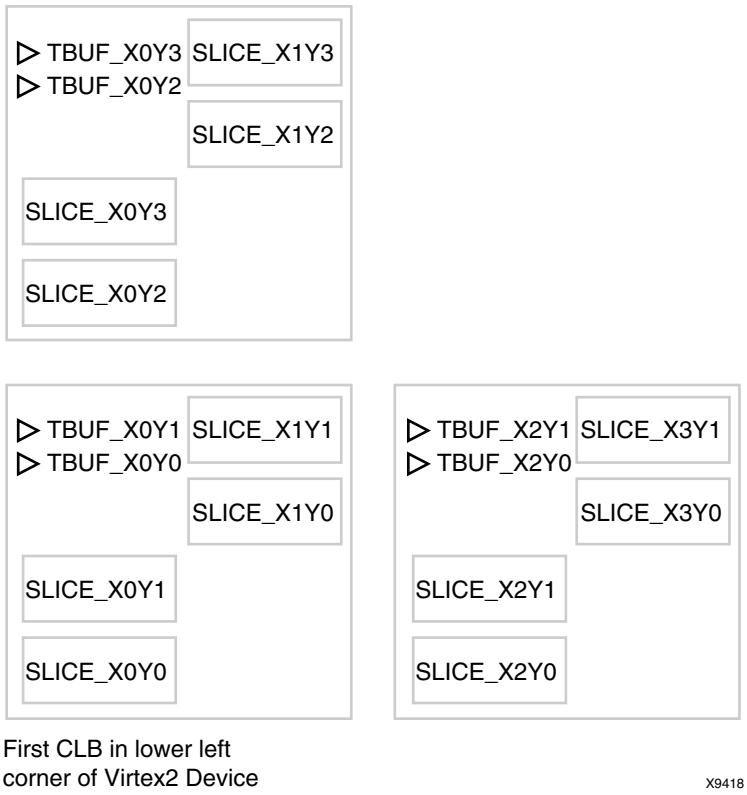

Figure 43-1: **Slice and TBUF Numbering in Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5**

Following are examples of how to specify the slices in the XY coordinate system.

SLICE_X0Y0	First (bottom) slice of the CLB in the lower left corner of the chip
SLICE_X0Y1	Second slice of the CLB in the lower left corner of the chip
SLICE_X1Y0	Third slice of the CLB in the lower left corner of the chip
SLICE_X1Y1	Fourth (top) slice of the CLB in the lower left corner of the chip
SLICE_X0Y2	First slice of the second CLB in CLB column 1
SLICE_X2Y0	First (bottom) slice of the bottom CLB in CLB column 2
SLICE_X2Y1	Second slice of the bottom CLB in CLB column 2
SLICE _X50Y125	Slice located 125 slices up from and 50 slices to the right of SLICE_X0Y0

The Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 block RAMs, TBUFs, and multipliers have their own specification different from the SLICE specifications. Therefore, the location value must start with "SLICE," "RAMB," "TBUF," or "MULT." The Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 block RAMs and multipliers have their own XY

grids different from the SLICE XY grid. A block RAM located at RAMB16_X2Y3 is not located at the same site as a flip-flop located at SLICE_X2Y3. A multiplier located at MULT18X18_X2Y3 is not located at the same site as a flip-flop located at SLICE_X2Y3 or at the same site as a block RAM located at RAMB16_X2Y3. However, the two TBUFs in each CLB follow the same XY grid as the SLICEs. A TBUF located at TBUF_X2Y3 is in the same CLB as a flip-flop located at SLICE_X2Y3.

Because there are two TBUFs per CLB and four slices per CLB, the X value for a TBUF is always an even integer or zero (for example, TBUF_X1Y1 is illegal).

The location values for global buffers and DLL elements is the specific physical site names for available locations.

LOC Description for CPLD Devices

For CPLD devices, use the LOC=*pin_name* constraint on a PAD symbol or pad net to assign the signal to a specific pin. The PAD symbols are IPAD, OPAD, IOPAD, and UPAD. You can use the LOC=FBnn constraint on any instance or its output net to assign the logic or register to a specific function block or macrocell, provided the instance is not collapsed.

The LOC=FB*nn_mm* constraint on any internal instance or output pad assigns the corresponding logic to a specific function block or macrocell within the CPLD. If a LOC is placed on a symbol that does not get mapped to a macrocell or is otherwise removed through optimization, the LOC is ignored.

Pin assignment using the LOC constraint is not supported for bus pad symbols such as OPAD8.

Location Specification Types for FPGA Devices

Use the following location types to define the physical location of an element.

Table 43-1: **Location Specification Types for FPGA Devices**

Element Types	Location Examples	Meaning
IOBs		
	P12	IOB location (chip carrier)
	A12	IOB location (pin grid)
	B, L, T, R	Applies to IOBs and indicates edge locations (bottom, left, top, right) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
	LB, RB, LT, RT, BR, TR, BL, TL	Applies to IOBs and indicates half edges (for example, left bottom, right bottom) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices

Table 43-1: **Location Specification Types for FPGA Devices**

Element Types	Location Examples	Meaning
	Bank0, Bank1, Bank2, Bank3, Bank4, Bank5, Bank6, Bank7	Applies to IOBs and indicates half edges (banks) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
CLBs		
	CLB_R4C3 (or .S0 or .S1)	CLB location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
	CLB_R6C8.S0 (or .S1)	Function generator or register slice for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
Slices		
	SLICE_X22Y3	SLICE_X22Y3 Slice location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
TBUFs		
	TBUF_R6C7 (or .0 or .1)	TBUF location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
Block RAMs		
	RAMB4_R3C1	Block RAM location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
	RAMB16_X2Y56	Block RAM location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Multipliers		
	MULT18X18_X55Y82	Multiplier location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Global Clocks		
	GCLKBUF0 (or 1, 2, or 3)	Global clock buffer location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
	GCLKPAD0 (or 1, 2, or 3)	Global clock pad location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
Delay Locked Loops		
	DLL0P(or S) (or 1, 2, or 3)	Delay Locked Loop element location for Spartan-II, Spartan-IIE, Virtex, Virtex-E devices
Digital Clock Manager		

Table 43-1: **Location Specification Types for FPGA Devices**

Element Types	Location Examples	Meaning
	DCM_X[A]Y[B]	Digital Clock Manager for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Phase Lock Loop		
	PLL_X[A]Y[B]	Phase Lock Loop for Virtex-5

The wildcard character (*) can be used to replace a single location with a range as shown in the following example:

CLB_R*C5	Any CLB in column 5 of a Spartan-II, Spartan-IIE, Virtex, or Virtex-E device
SLICE_X*Y5	Any slice of a Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, or Virtex-5 device whose Y coordinate is 5

The following are *not* supported.

- Dot extensions on ranges. For example, LOC=CLB_R0C0:CLB_R5C5.G.
- Wildcard character for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, or Virtex-5 global buffer, global pad, or DLL locations.

LOC Priority

When specifying two adjacent LOCs on an input pad and its adjoining net, the LOC attached to the net has priority. In the following diagram, LOC=11 takes priority over LOC=38.

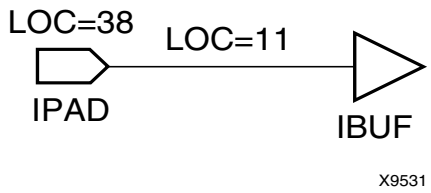

Figure 43-2: **LOC Priority Example**

LOC Propagation Rules

For all nets, LOC is illegal when attached to a net or signal except when the net or signal is connected to a pad. In this case, LOC is treated as attached to the pad instance.

For CPLD nets, LOC attaches to all applicable elements that drive the net or signal.

When attached to a design element, LOC propagates to all applicable elements in the hierarchy within the design element.

LOC Syntax for FPGA Devices

This section discusses LOC syntax for FPGA devices in:

- "Single Location"
- "Multiple Locations"
- "Range of Locations"

Single Location

The basic UCF syntax is:

```
INST "instance_name" LOC=location;
```

where

- *location* is a legal location for the part type

Examples of the syntax for single LOC constraints are given in the following table.

Table 43-2: **Single LOC Constraint Examples**

Constraint (UCF Syntax)	Devices	Description
INST "*instance_name*"LOC=P12;		Place I/O at location P12.
INST "*instance_name*"LOC=CLB_R3C5;	Spartan-II, Spartan-IIE, Virtex, and Virtex-E	Place logic in either slice of the CLB in row3, column 5.
INST "*instance_name*"LOC=CLB_R3C5.S0;	Spartan-II, Spartan-IIE, Virtex, and Virtex-E	Place logic in the left slice of the CLB in row 3, column 5.
INST "*instance_name*" LOC=SLICE_X3Y2;	Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5	Place logic in slice X3Y2 on the XY SLICE grid.
INST "*instance_name*" LOC=TBUF_R1C2.*;	Spartan-II, Spartan-IIE, Virtex, and Virtex-E	Place both TBUFs in row 1, column 2.
INST "*instance_name*" LOC=TBUF_X0Y6;	Virtex-II, Virtex-II Pro, and Virtex-II Pro X, Virtex-4, and Virtex-5	Place logic in the BUFT located at TBUF_ X0Y6 on the XY SLICE grid
INST "*instance_name*" LOC=RAMB4_R*C1;	Spartan-II, Spartan-IIE, Virtex, and Virtex-E	Specifies any block RAM in column 1 of the block RAM array
INST "*instance_name*" LOC=RAMB16_X0Y6;	Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5	Place the logic in the block RAM located at RAMB16_X0Y6 on the XY RAMB grid.
INST "*instance_name*" LOC=MULT18X18_X0Y6;	Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5	Place the logic in the multiplier located at MULT18X18_X0Y6 on the XY MULT grid.
INST "*instance_name*" LOC=FIFO16_X0Y15;	Virtex-4, and Virtex-5	Place the logic in the FIFO located at FIFO16_X0Y15 on the XY FIFO grid.
INST "*instance_name*" LOC=IDELAYCTRL_X0Y3;	Virtex-4, and Virtex-5	Place the logic in the IDELAYCTRL located at the IDELAYCTRL_X0Y3 on the XY IDELAYCTRL grid.

Multiple Locations

LOC=*location1*,*location2*,...,*locationx*

Separating each such constraint by a comma specifies multiple locations for an element. When you specify multiple locations, PAR can use any of the specified locations. Examples of multiple LOC constraints are provided in the following table.

Table 43-3: **Multiple LOC Constraint Examples**

Constraint	Devices	Description
`INST "instance_name" LOC=clb_r4c5.s1, clb_r4c6.*;`	Spartan-II, Spartan-IIE, Virtex, and Virtex-E	Place the flip-flop in the right-most slice of CLB R4C5 or in either slice of CLB R4C6.
`INST "instance_name" LOC=SLICE_X2Y10, SLICE_X1Y10;`	Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5	Place the logic in SLICE_X2Y10 or in SLICE_X1Y10 on the XY SLICE grid.

Currently, using a single constraint there is no way to constrain multiple elements to a single location or multiple elements to multiple locations.

Range of Locations

The basic UCF syntax is:

INST "*instance_name***" LOC=***location***:***location* {**SOFT**};

You can define a range by specifying the two corners of a bounding box. Except for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, specify the upper left and lower right corners of an area in which logic is to be placed. For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, specify the lower left and upper right corners. Use a colon (:) to separate the two boundaries.

The logic represented by the symbol is placed somewhere inside the bounding box. The default is to interpret the constraint as a "hard" requirement and to place it within the box. If SOFT is specified, PAR may place the constraint elsewhere if better results can be obtained at a location outside the bounding box. Examples of LOC constraints used to specify a range are given in the following table.

Table 43-4: **LOC Range Constraint Examples**

Constraint	Devices	Description
`INST "instance_name" LOC=CLB_R1C1:CLB_R4C4;`	Spartan-II, Spartan-IIE, Virtex, Virtex-E	Place logic in either slice in the top left corner of the CLB bounded by row 4, column 4.
`INST "instance_name" LOC=SLICE_X3Y5:SLICE_X5Y20;`	Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5	Place logic in any slice within the rectangular area bounded by SLICE_X3Y5 (the lower left corner) and SLICE_X5Y20 (the upper right corner) on the XY SLICE grid.

LOC ranges can be supplemented with the keyword SOFT. Unlike AREA_GROUP, LOC ranges do not influence the packing of symbols. LOC range is strictly a placement constraint used by PAR.

LOC Syntax for CPLD Devices

The basic UCF syntax is:

INST "*instance_name***" LOC=***pin_name***;**

or

INST "*instance_name***" LOC=FB***ff***;**

or

INST "*instance_name***" LOC=FB***ff_mm***;**

where

- *pin_name* is P*nn* for numeric pin names or *rc* for row-column pin names
- *ff* is a function block number
- *mm* is a macrocell number within a function block

LOC Syntax Examples

For examples of legal placement constraints for each type of logic element in FPGA designs, see "LOC Syntax for FPGA Devices" in this chapter, and the "Relative Location (RLOC)" constraint. Logic elements include flip-flops, ROMs and RAMs, block RAMS, FMAPs, BUFTs, CLBs, IOBs, I/Os, edge decoders, and global buffers.

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an instance
- Attribute Name: LOC
- Attribute Values: *value*

For valid values, see "LOC Syntax for FPGA Devices" and "LOC Syntax for CPLD Devices" in this chapter.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute loc: string;
```

Specify the VHDL constraint as follows:

```
attribute loc of {signal_name|label_name}: {signal|label} is
"location";
```

Set the LOC constraint on a bus as follows:

```
attribute loc of bus_name : signal is "location_1 location_2
location_3...";
```

To constrain only a portion of a bus (CPLD devices only), use the following syntax:

```
attribute loc of bus_name : signal is "* * location_1 * location_2...";
```

For more information about *location*, see "LOC Syntax for FPGA Devices" and "LOC Syntax for CPLD Devices"in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Examples

Specify the Verilog constraint as follows:

```
(* LOC = "location" *)
```

Set the LOC constraint on a bus as follows:

```
(* LOC = "location_1 location_2 location_3..." *)
```

To constrain only a portion of a bus (CPLD devices only), use the following syntax:

```
(* LOC = "* *location_1 location_2..." *)
```

For more information about *location*, see "LOC Syntax for FPGA Devices" and "LOC Syntax for CPLD Devices" in this chapter.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Examples

Pin Assignment

```
mysignal PIN 12;
```

Internal Location Constraint

```
XILINX PROPERTY 'loc=fb1 mysignal';
```

UCF and NCF Syntax Examples

The following statement specifies that each instance found under "FLIP_FLOPS" is to be placed in any CLB in column 8.

```
INST "/FLIP_FLOPS/*" LOC=CLB_R*C8;
```

The following statement specifies that an instantiation of MUXBUF_D0_OUT be placed in IOB location P110.

```
INST "MUXBUF_D0_OUT" LOC=P110;
```

The following statement specifies that the net DATA<1> be connected to the pad from IOB location P111.

```
NET "DATA<1>" LOC=P111
```

XCF Syntax Examples

```
BEGIN MODEL "entity_name"
 PIN "signal_name"  loc=string;
 INST "instance_name" loc=string;
END;
```

Constraints Editor Syntax Examples

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Ports tab grid, double-click the Location column in the row with the desired port name and fill out the Location dialog box. This locks the selected signal to the specified pin. You cannot set any other location constraints in the Constraints Editor.

PCF Syntax Examples

LOC writes out a LOCATE constraint to the PCF file. For more information, see the "Locate (LOCATE)" constraint.

Floorplanner Syntax Examples

After you place your logic within Floorplanner, save the file as a UCF file to create a LOC constraint. For more information, see the following topics in the Floorplanner help:

- Creating and Editing Area Constraints
- Using a Floorplanner UCF File in Project Navigator

PACE Syntax Examples

The Pin Assignments Editor is mainly used for assigning location constraints to IOs in designs. You can access PACE from the Processes window in the Project Navigator. Double-click Assign Package Pins or Create Area Constraints under User Constraints.

For more information, see the PACE help, especially the topics within Editing Pins and Areas in the Procedures section.

BUFT Examples

You can constrain internal 3-state buffers (BUFTs) to an individual BUFT location, a list of BUFT locations, or a rectangular block of BUFT locations. BUFT constraints all refer to locations with a prefix of TBUF, which is the name of the physical element on the device.

BUFT constraints can be assigned from the schematic or through the UCF file. From the schematic, LOC constraints are attached to the target BUFT. The constraints are then passed into the EDIF netlist file and after mapping are read by PAR. Alternatively, in a constraints file a BUFT is identified by a unique instance name.

Fixed Locations

This section describes fixed locations for:

- "Virtex, Virtex-E, Spartan-II, and Spartan-IIE"
- "Spartan-3 and Higher Devices"

Virtex, Virtex-E, Spartan-II, and Spartan-IIE

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE use the following syntax to denote fixed locations.

TBUF_R*row***C***col*{**.0**|**.1**}

where

- *row* is the row location
- *col* is the column location

They can be any number between 0 and 99, inclusive. They must be less than or equal to the number of CLB rows or columns in the target device.

A suffix of .0 or .1 is required.

The suffixes have the following meanings:

- 0 indicates at least one TBUF at the specific row/column
- 1 indicates the second TBUF at the specific row/column

Spartan-3 and Higher Devices

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, use the following syntax to denote fixed locations:

TBUF_X*m***Y***n*

where

- *m* and *n* represent XY values on the slice-based X0Y0 grid

The TBUFs are associated with the SLICE grid. Because there are two TBUFs per CLB and four slices per CLB, the X value for a TBUF location can only be an even integer or zero. The values must be less than or equal to the number of slices in the target device.

Range of Locations

For Spartan-II, Spartan-IIE, Virtex, or Virtex-E, use the following syntax to denote a range of locations from the lowest to the highest.

TBUF_R*row***C***col***:TBUF_R***row***C***col*

Range of locations does not apply to Spartan-3 and higher devices.

Format of BUFT LOC Constraints

The following examples illustrate the format of BUFT LOC constraints. Specify LOC= and the BUFT location.

LOC=TBUF_R1C1.0 (or .1)	Spartan-II, Spartan-IIE, Virtex, and Virtex-E
LOC=TBUF_X2Y1	Virtex-II, Virtex-II Pro, and Virtex-II Pro X

The next statements place BUFTs at any location in the first column of BUFTs. The asterisk (*) is a wildcard character.

LOC=TBUF_R*C0	Spartan-II, Spartan-IIE, Virtex, and Virtex-E
LOC=TBUF_X0Y*	Virtex-II, Virtex-II Pro, and Virtex-II Pro X

The following statements place BUFTs within the rectangular block defined by the two TBUFs/LOCs. For all architectures except Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the first specified BUFT is in the upper left corner and the second specified BUFT is in the lower right corner. For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the first BUFT is the lower left corner and the second is the upper right corner.

LOC=TBUF_R1C1:TBUF_R2C8	Spartan-II, Spartan-IIE, Virtex, and Virtex-E
LOC=TBUF_X0Y1:TBUF_X2Y8	Virtex-II, Virtex-II Pro, and Virtex-II Pro X

CLB-Based Row/Column/Slice Designations

The examples in this section apply to Spartan-II, Spartan-IIE, Virtex, and Virtex-E architectures.

In the following examples, the instance names of two BUFTs are /top-72/rd0 and /top-79/ed7. The examples are:

- "Example One: BUFT Adjacent to a Specific CLB"
- "Example Two: BUFT in a Specific Location"
- "Example Three: Column of BUFTs"
- "Example Four: Row of BUFTs"

Example One: BUFT Adjacent to a Specific CLB

The following example specifies a BUFT adjacent to a specific CLB.

Schematic	LOC=TBUF_R1C5
UCF	INST "/top-72/rd0" LOC=TBUF_R1C5;

Place the BUFT adjacent to CLB R1C5. In Spartan-II, Spartan-IIE, Virtex, and Virtex-E, PAR places the BUFT in one of two slices of the CLB at row 1, column 5.

Example Two: BUFT in a Specific Location

The following example places a BUFT in a specific location.

Schematic	LOC=TBUF_r1c5.1
UCF	INST "/top-72/rd0" LOC=TBUF_r1c5.1;

Place the BUFT adjacent to CLB R1C5. In Spartan-II, Spartan-IIE, Virtex, and Virtex-E, the .1 tag specifies the second TBUF in CLB R1C5.

BUFTs that drive the same signal must carry consistent constraints. If you specify .1 or .2 for one of the BUFTs that drives a given signal, you must also specify .1 or .2 on the other BUFTs on that signal; otherwise, do not specify any constraints at all.

Example Three: Column of BUFTs

The following example specifies a column of BUFTs.

Schematic	LOC=TBUF_r*c3
UCF	INST "/top-72/rd0 /top-79/ed7" LOC=TBUF_r*c3;

Place BUFTs in column 3 on any row. This constraint might be used to align BUFTs with a common enable signal. You can use the wildcard (*) character in place of either the row or column number to specify an entire row or column of BUFTs.

Example Four: Row of BUFTs

The following example specifies a row of BUFTs.

Schematic	LOC=TBUF_r7c*
UCF	INST "/top-79/ed7" LOC=TBUF_r7c*;

Place the BUFT on one of the longlines in row 7 for any column. You can use the wildcard (*) character in place of either the row or column number to specify an entire row or column of BUFTs.

Sliced-Based XY Coordinate Designations

The examples in this section apply to the Virtex-II, Virtex-II Pro, and Virtex-II Pro X devices.

The examples are:

- "Example One: BUFT in a Specific Location"
- "Example Two: Column of BUFTs"
- "Example Three: Row of BUFTs"

Example One: BUFT in a Specific Location

The following example places a BUFT in a specific location.

Schematic	LOC=TBUF_X4Y5
UCF	INST "/top-72/rd0" LOC=TBUF_X4Y5;

Place the BUFT in TBUF_X4Y5 in the CLB containing SLICE_X4Y5.

BUFTs that drive the same signal must carry consistent constraints.

Example Two: Column of BUFTs

The following example specifies a column of BUFTs.

Schematic	LOC=TBUF_X6Y*
UCF	INST "/top-72/rd0 /top-79/ed7" LOC=TBUF_X6Y*;

Place BUFTs in the column of CLBs that contains the TBUFs whose X coordinate is 6. This constraint might be used to align BUFTs with a common enable signal. You can use the wildcard (*) character in place of either the X or Y coordinate to specify an entire row (X*) or column (Y*) of BUFTs.

Example Three: Row of BUFTs

The following example specifies a row of BUFTs.

Schematic	LOC=TBUF_X*Y6
UCF	INST "/top-79/ed7" LOC=TBUF_X*Y6;

Place the BUFT on one of the longlines in the row of CLBs that contains TBUFs whose Y coordinate is 6. You can use the wildcard (*) character in place of either the X or Y coordinate to specify an entire row (X*) or column (Y*) of TBUFs.

CLB Examples (CLB-Based Row/Column/Slice Architectures Only)

Note: This section applies only to the architecture that uses the CLB-based Row/Column/Slice designations:

You can assign soft macros and flip-flops to a single CLB location, a list of CLB locations, or a rectangular block of CLB locations. You can also specify the exact function generator or flip-flop within a CLB. CLB locations are identified as CLB_R*row*C*col* for Spartan-II, Spartan-IIE, Virtex, and Virtex-E. The upper left CLB is CLB_R1C1.

CLB Locations

CLB locations can be a fixed location or a range of locations.

Fixed Locations

Use the following syntax to denote fixed locations.

For Spartan-II, Spartan-IIE, Virtex, and Virtex-E:

```
CLB_Rrow Ccol{.S0 | .S1}
```

where

- *row* is the row location
- *col* is the column location

They can be any number between 0 and 99, inclusive, or *.

They must be less than or equal to the number of CLB rows or columns in the target device.

The suffixes have the following meanings.

- .S0 means the right-most slice in the Spartan-II, Spartan-IIE, Virtex, and Virtex-E CLB
- .S1 means the left-most slice in the Spartan-II, Spartan-IIE, Virtex, and Virtex-E CLB

Range of Locations

Use the following syntax to denote a range of locations from the highest to the lowest.

CLB_R*row1***C***col***:CLB_R***row2***C***col2*

Format of CLB Constraints

The following examples illustrate the format of CLB constraints. Enter LOC= and the pin or CLB location. If the target symbol represents a soft macro, the LOC constraint is applied to all appropriate symbols (flip-flops, maps) contained in that macro. If the indicated logic does not fit into the specified blocks, an error is generated.

- The following UCF statement places logic in the designated CLB.

  INST "*instance_name***" LOC=CLB_R1C1.S0;**

  (Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices)

- The following UCF statement places logic within the first column of CLBs. The asterisk (*) is a wildcard character.

  INST "*instance_name***" LOC=CLB_R*C1.S0;**

  (Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices)

- The next two UCF statements place logic in any of the three designated CLBs. There is no significance to the order of the LOC statements.

  INST "*instance_name***" LOC=CLB_R1C1,CLB_R1C2,CLB_R1C3;**

  (Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices)

- The following statement places logic within the rectangular block defined by the first specified CLB in the upper left corner and the second specified CLB towards the lower right corner.

  INST "*instance_name***" LOC=CLB_R1C1:CLB_R8C5;**

  (Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices)

You can prohibit PAR from using a specific CLB, a range of CLBs, or a row or column of CLBs. Such PROHIBIT constraints can be assigned only through the User Constraints File (UCF). CLBs are prohibited by specifying a PROHIBIT constraint at the design level, as shown in the following examples.

Example One

Do not place any logic in the CLB in row 1, column 5. CLB R1C1 is in the upper left corner of the device.

Schematic	None
UCF	CONFIG PROHIBIT=clb_r1c5;

Example Two

Do not place any logic in the rectangular area bounded by the CLB R1C1 in the upper left corner and CLB R5C7 in the lower right.

Schematic	None
UCF	CONFIG PROHIBIT=clb_r1c1:clb_r5c7;

Example Three

Do not place any logic in any row of column 3. You can use the wildcard (*) character in place of either the row or column number to specify an entire row or column of CLBs.

Schematic	None
UCF	CONFIG PROHIBIT=clb_r*c3;

Example Four

Do not place any logic in either CLB R2C4 or CLB R7C9.

Schematic	None
UCF	CONFIG PROHIBIT=clb_r2c4, clb_r7c9;

Delay Locked Loop (DLL) Constraint Examples

This section applies to Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices only.

You can constrain DLL elements—CLKDLL, CLKDLLE, and CLKDLLHF—to a specific physical site name. Specify LOC=DLL and a numeric value (0 through 3) to identify the location.

Following is an example.

Schematic	LOC=DLL1P
UCF	INST “buf1” LOC=DLL1P;

Digital Clock Manager (DCM) Constraint Examples

This section applies to Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only.

You can lock the DCM in the UCF file. The syntax is as follows:

```
INST "instance_name" LOC = DCM_XAYB;
```

A is the X coordinate, starting with 0 at the left-hand bottom corner. *A* increases in value as you move across the device to the right.

B is the Y coordinate, starting with 0 at the left-hand bottom corner. *B* increases in value as you move up the device.

For example:

```
INST "myinstance" LOC = DCM_X0Y0;
```

Flip-Flop Constraint Examples

Flip-flop constraints can be assigned from the schematic or through the UCF file.

From the schematic, attach LOC constraints to the target flip-flop. The constraints are then passed into the EDIF netlist and are read by PAR after the design is mapped.

The following examples show how the LOC constraint is applied to a schematic and to a UCF (User Constraints File). The instance names of two flip-flops, /top-12/fdrd and /top-54/fdsd, are used to show how you would enter the constraints in the UCF.

CLB-Based Row/Column/Slice Designations

The Virtex architecture uses CLB-based Row/Column/Slice designations.

Flip-flops can be constrained to a specific CLB, a range of CLBs, a row or column of CLBs, or a specific half-CLB.

Example One

Place the flip-flop in the CLB in row 1, column 5. CLB R1C1 is in the upper left corner of the device.

Schematic	LOC=CLB_RlC5
UCF	INST "/top-12/fdrd" LOC=CLB_R1C5;

Example Two

Place the flip-flop in the rectangular area bounded by the CLB R1C1 in the upper left corner and CLB R5C7 in the lower right corner.

Schematic	LOC=CLB_R1C1:CLB_R5C7
UCF	INST "/top-12/fdrd" LOC=CLB_R1C1:CLB_R5C7;

Example Three

Place the flip-flops in any row of column 3. You can use the wildcard (*) character in place of either the row or column number to specify an entire row or column of CLBs.

Schematic	LOC=CLB_R*C3
UCF	INST "/top-12/fdrd/top-54/fdsd" LOC=CLB_R*C3;

Example Four

Place the flip-flop in either CLB R2C4 or CLB R7C9.

Schematic	LOC=CLB_R2C4,CLB_R7C9
UCF	INST "/top-54/fdsd" LOC=CLB_R2C4,CLB_R7C9;

In Example Four, repeating the LOC constraint and separating each such constraint by a comma specifies multiple locations for an element. When you specify multiple locations, PAR can use any of the specified locations.

Example Five

Do not place the flip-flop in any column of row 5. You can use the wildcard (*) character in place of either the row or column number to specify an entire row or column of CLBs.

Schematic	PROHIBIT=CLB_R5C*
UCF	CONFIG PROHIBIT=CLB_R5C*;

Slice-Based XY Grid Designations

Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 are the only architectures that use slice-based XY grid designations.

Flip-flops can be constrained to a specific slice, a range of slices, a row or column of slices.

Example One

Place the flip-flop in SLICE_X1Y5. SLICE_X0Y0 is in the lower left corner of the device.

Schematic	LOC=SLICE_XlY5
UCF	INST “/top-12/fdrd” LOC=SLICE_X1Y5;

Example Two

Place the flip-flop in the rectangular area bounded by the SLICE_X1Y1 in the lower left corner and SLICE_X5Y7 in the upper right corner.

Schematic	LOC=SLICE_R1C1:SLICE_R5C7
UCF	INST “/top-12/fdrd” LOC=SLICE_X1Y1:SLICE_X5Y7;

Example Three

Place the flip-flops anywhere in the row of slices whose Y coordinate is 3. Use the wildcard (*) character in place of either the X or Y value to specify an entire row (Y*) or column (X*) of slices.

Schematic	LOC=SLICE_X*Y3
UCF	INST “/top-12/fdrd/top-54/fdsd” LOC=SLICE_X*Y3;

Example Four

Place the flip-flop in either SLICE_X2Y4 or SLICE_X7Y9.

Schematic	LOC=SLICE_X2Y4,SLICE_X7Y9
UCF	INST “/top-54/fdsd” LOC=SLICE_X2Y4, SLICE_X7Y9;

In Example Four, repeating the LOC constraint and separating each such constraint by a comma specifies multiple locations for an element. When you specify multiple locations, PAR can use any of the specified locations.

Example Five

Do not place the flip-flop in the column of slices whose X coordinate is 5. Use the wildcard (*) character in place of either the X or Y value to specify an entire row (Y*) or column (X*) of slices.

Schematic	PROHIBIT=SLICE_X5Y*
UCF	CONFIG PROHIBIT=SLICE_X5Y*;

Global Buffer Constraint Examples

This section applies to Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices only.

You can constrain a Spartan-II, Spartan-IIE, Virtex, and Virtex-E global buffer (BUFGP and IBUFG_SelectIO variants) to a specific buffer site name or dedicated global clock pad in the device model.

From the schematic, attach LOC constraints to the global buffer symbols. Specify LOC= and GCLKBUF plus a number (0 through 3) to create a specific buffer site name in the device model. Or, specify LOC= and GCLKPAD plus a number (0 through 3) to create a specific dedicated global clock pad in the device model. The constraints are then passed into the EDIF netlist and after mapping are read by PAR.

Example

Schematic	LOC=GCLKBUF1
UCF	INST “buf1” LOC=GCLKBUF1;
Schematic	LOC=GCLKPAD1
UCF	INST “buf1” LOC=GCLKPAD1;

I/O Constraint Examples

You can constrain I/Os to a specific IOB. You can assign I/O constraints from the schematic or through the UCF file.

From the schematic, attach LOC constraints to the target PAD symbol. The constraints are then passed into the netlist file and read by PAR after mapping.

Alternatively, in the UCF file a pad is identified by a unique instance name. The following example shows how the LOC constraint is applied to a schematic and to a UCF (User Constraints File). In the examples, the instance names of the I/Os are /top-102/data0_pad and /top-117/q13_pad. The example uses a pin number to lock to one pin.

Schematic	LOC=P17
UCF	INST “/top-102/data0_pad” LOC=P17;

Place the I/O in the IOB at pin 17. For pin grid arrays, a pin name such as B3 or T1 is used.

IOB Constraint Examples

You can assign I/O pads, buffers, and registers to an individual IOB location. IOB locations are identified by the corresponding package pin designation.

The following examples illustrate the format of IOB constraints. Specify LOC= and the pin location. If the target symbol represents a soft macro containing only I/O elements, for example, INFF8, the LOC constraint is applied to all I/O elements contained in that macro. If the indicated I/O elements do not fit into the specified locations, an error is generated.

The following UCF statement places the I/O element in location P13. For PGA packages, the letter-number designation is used, for example, B3.

```
INST "instance_name" LOC=P13;
```

You can prohibit the mapper from using a specific IOB. You might take this step to keep user I/O signals away from semi-dedicated configuration pins. Such PROHIBIT constraints can be assigned only through the UCF file.

IOBs are prohibited by specifying a PROHIBIT constraint preceded by the CONFIG keyword, as shown in the following example.

Schematic	None
UCF	CONFIG PROHIBIT=p36, p37, p41;

Do not place user I/Os in the IOBs at pins 36, 37, or 41. For pin grid arrays, pin names such as D14, C16, or H15 are used.

Mapping Constraint Examples (FMAP)

Mapping constraints control the mapping of logic into CLBs. They have two parts. The first part is an FMAP component placed on the schematic. The second is a LOC constraint that can be placed on the schematic or in the constraints file.

FMAP controls the mapping of logic into function generators. This symbol does not define logic on the schematic; instead, it specifies how portions of logic shown elsewhere on the schematic should be mapped into a function generator.

The FMAP symbol defines mapping into a four-input (F) function generator. For Spartan-II, Spartan-IIE, Virtex, and Virtex-E, the four-input function generator defined by the FMAP is assigned to one of the two slices of the CLB.

For the FMAP symbol as with the CLBMAP primitive, MAP=PUC or PUO is supported, as well as the LOC constraint. (Currently, pin locking is not supported. MAP=PLC or PLO is translated into PUC and PUO, respectively.)

Example One

Place the FMAP symbol in the CLB at row 7, column 3.

Schematic	LOC=CLB_R7C3
UCF	INST “$1I323” LOC=CLB_R7C3;

Example Two

Place the FMAP symbol in either the CLB at row 2, column 4 or the CLB at row 3, column 4.

Schematic	LOC=CLB_R2C4,CLB_R3C4
UCF	INST “top/dec0011” LOC=CLB_R2C4,CLB_R3C4;

Example Three

Place the FMAP symbol in the area bounded by CLB R5C5 in the upper left corner and CLB R10C8 in the lower right

Schematic	LOC=CLB_R5C5:CLB_R10C8
UCF	INST “$3I27” LOC=CLB_R5C5:CLB_R10C8;

Example Four (Virtex, Virtex-E, Spartan-II, and Spartan-IIE)

Place the FMAP in the right-most slice of the CLB in row 10, column 11.

Schematic	LOC=CLB_R10C11.S0
UCF	INST “/top/done” LOC=CLB_R10C11.S0;

Multiplier Constraint Examples

This section applies to Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only.

Multiplier constraints can be assigned from the schematic or through the UCF file. From the schematic, attach the LOC constraints to a multiplier symbol. The constraints are then passed into the netlist file and after mapping they are read by PAR. For more information on attaching LOC constraints, see the application user guide. Alternatively, in the constraints file a multiplier is identified by a unique instance name.

A Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 multiplier has a different XY grid specification than slices, block RAMs, and TBUFs. It is specified using MULT18X18_*X*m*Y*n where the X and Y coordinate values correspond to the multiplier grid array. A multiplier located at MULT18X18_X0Y1 is not located at the same site as a flip-flop located at SLICE_X0Y1 or a block RAM located at RAMB16_X0Y1.

For example, assume you have a device with two columns of multipliers, each column containing two multipliers, where one column is on the right side of the chip and the other is on the left. The multiplier located in the lower left corner is MULT18X18_X0Y0. Because there are only two columns of multipliers, the multiplier located in the upper right corner is MULT18X18_X1Y1.

Schematic	LOC=MULT18X18_X0Y0
UCF	INST “/top-7/rq” LOC=MULT18X18_X0Y0;

ROM Constraint Examples

Memory constraints can be assigned from the schematic or through the UCF file.

From the schematic, attach the LOC constraints to the memory symbol. The constraints are then passed into the netlist file and after mapping they are read by PAR. For more information on attaching LOC constraints, see the application user guide.

Alternatively, in the constraints file memory is identified by a unique instance name. One or more memory instances of type ROM can be found in the input file. All memory macros larger than 16 x 1 or 32 x 1 are broken down into these basic elements in the netlist file.

In the following examples, the instance name of the ROM primitive is /top-7/rq.

CLB-Based Row/Column/Slice Designations

The Virtex architecture uses CLB-based Row/Column/Slice designations. You can constrain a ROM to a specific CLB, a range of CLBs, a row or column of CLBs.

Example One

Place the memory in the CLB in row 1, column 5. CLB R1C1 is in the upper left corner of the device. You can only apply a single-CLB constraint such as this to a 16 x 1 or 32 x 1 memory.

Schematic	LOC=clb_r1c5
UCF	INST “/top-7/rq” LOC=clb_r1c5;

Example Two

Place the memory in either CLB R2C4 or CLB R7C9.

Schematic	LOC=clb_r2c4, clb_r7c9
UCF	INST “/top-7/rq” LOC=clb_r2c4, clb_r7c9;

Example Three

Do not place the memory in any column of row 5. You can use the wildcard (*) character in place of either the row or column number in the CLB name to specify an entire row or column of CLBs.

Schematic	PROHIBIT clb_r5c*
UCF	CONFIG PROHIBIT=clb_r5c*;

Slice-Based XY Designations

Only Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices use slice-based XY grid designations. You can constrain a ROM to a specific slice, a range of slices, or a row or column of slices.

Example One

Place the memory in the SLICE_X1Y1. SLICE_X1Y1 is in the lower left corner of the device. You can apply a single-SLICE constraint such as this only to a 16 x 1 or 32 x 1 memory.

Schematic	LOC=SLICE_X1Y1
UCF	INST "/top-7/rq" LOC=SLICE_X1Y1;

Example Two

Place the memory in either SLICE_X2Y4 or SLICE_X7Y9.

Schematic	LOC=SLICE_X2Y4, SLICE_X7Y9
UCF	INST "/top-7/rq" LOC=SLICE_X2Y4, SLICE_X7Y9;

Example Three

Do not place the memory in column of slices whose X coordinate is 5. You can use the wildcard (*) character in place of either the X or Y coordinate value in the SLICE name to specify an entire row (Y*) or column (X*) of slices.

Schematic	PROHIBIT SLICE_X5Y*
UCF	CONFIG PROHIBIT=SLICE_X5Y*;

Block RAM (RAMBs) Constraint Examples

This section applies to Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only.

Block RAM constraints can be assigned from the schematic or through the UCF file. From the schematic, attach the LOC constraints to the block RAM symbol. The constraints are then passed into the netlist file. After mapping they are read by PAR. For more information on attaching LOC constraints, see the application user guide. Alternatively, in the constraints file a memory is identified by a unique instance name.

Spartan-II, Spartan-IIE, Virtex, and Virtex-E Devices

A Spartan-II, Spartan-IIE, Virtex, and Virtex-E block RAM has a different row/column grid specification than CLBs and TBUFs. It is specified using RAMB4_RnCn where the numeric row and column numbers refer to the block RAM grid array. A block RAM located at RAMB4_R3C1 is not located at the same site as a flip-flop located at CLB_R3C1.

For example, assume you have a device with two columns of block RAM, each column containing four blocks, where one column is on the right side of the chip and the other is on the left. The block RAM located in the upper left corner is RAMB4_R0C0. Because there are only two columns of block RAM, the block located in the upper right corner is RAMB4_R0C1.

Schematic	LOC=RAMB4_R0C0
UCF	INST "/top-7/rq" LOC=RAMB4_R0C0;

Spartan-3 and Higher Devices

A Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 block RAM has a different XY grid specification than a slice, multiplier, or TBUF. It is specified using RAMB16_X*m*Y*n* where the X and Y coordinate values correspond to the block RAM grid array. A block RAM located at RAMB16_X0Y1 is not located at the same site as a flip-flop located at SLICE_X0Y1.

For example, assume you have a device with two columns of block RAM, each column containing two blocks, where one column is on the right side of the chip and the other is on the left. The block RAM located in the lower left corner is RAMB16_X0Y0. Because there are only two columns of block RAM, the block located in the upper right corner is RAMB16_X1Y1.

Schematic	LOC=RAMB16_X0Y0
UCF	INST "/top-7/rq" LOC=RAMB16_X0Y0;

Slice Constraint Examples

This section applies only to Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices. These are currently the only architectures that use the slice-based XY grid designations.

You can assign soft macros and flip-flops to a single slice location, a list of slice locations, or a rectangular block of slice locations.

Slice locations can be a fixed location or a range of locations. Use the following syntax to denote fixed locations.

SLICE_X*m*Y*n*

where

- *m* and *n* are the X and Y coordinate values, respectively

They must be less than or equal to the number of slices in the target device. Use the following syntax to denote a range of locations from the highest to the lowest.

SLICE_X*m*Y*n*:SLICE_X*m*Y*n*

Format of Slice Constraints

The following examples illustrate the format of slice constraints: LOC= and the slice location. If the target symbol represents a soft macro, the LOC constraint is applied to all appropriate symbols (flip-flops, maps) contained in that macro. If the indicated logic does not fit into the specified blocks, an error is generated.

Slice Constraints Example One

The following UCF statement places logic in the designated slice for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices.

```
INST "instance_name" LOC=SLICE_X133Y10;
```

Slice Constraints Example Two

The following UCF statement places logic within the first column of slices for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices. The asterisk (*) is a wildcard character

```
INST "instance_name" LOC=SLICE_X0Y*;
```

Slice Constraints Example Three

The following UCF statement places logic in any of the three designated slices for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices. There is no significance to the order of the LOC statements.

```
INST "instance_name" LOC=SLICE_X0Y3, SLICE_X67Y120, SLICE_X3Y0;
```

Slice Constraints Example Four

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the following UCF statement places logic within the rectangular block defined by the first specified slice in the lower left corner and the second specified slice towards the upper right corner.

```
INST "instance_name" LOC=SLICE_X3Y22:SLICE_X10Y55;
```

Slices Prohibited

You can prohibit PAR from using a specific slice, a range of slices, or a row or column of slices. Such prohibit constraints can be assigned only through the User Constraints File (UCF). Slices are prohibited by specifying a PROHIBIT constraint at the design level, as shown in the following examples.

Slices Prohibited Example One

Do not place any logic in the SLICE_X0Y0. SLICE_X0Y0 is at the lower left corner of the device.

Schematic	None
UCF	CONFIG PROHIBIT=SLICE_X0Y0;

Slices Prohibited Example Two

Do not place any logic in the rectangular area bounded by SLICE_X2Y3 in the lower left corner and SLICE_X10Y10 in the upper right.

Schematic	None
UCF	CONFIG PROHIBIT=SLICE_X2Y3:SLICE_X10Y10;

Slices Prohibited Example Three

Do not place any logic in a slice whose location has 3 as the X coordinate. This designates a column of prohibited slices. You can use the wildcard (*) character in place of either the X or Y coordinate to specify an entire row (X*) or column (Y*) of slices.

Schematic	None
UCF	CONFIG PROHIBIT=SLICE_X3Y*;

Example Four

Do not place any logic in either SLICE_X2Y4 or SLICE_ X7Y9.

Schematic	None
UCF	CONFIG PROHIBIT=SLICE_X2Y4, SLICE_X7Y9;

Locate (LOCATE)

LOCATE Architecture Support

The LOCATE constraint applies to FPGA devices only.

LOCATE Applicable Elements

- CLBs
- IOBs
- TBUFs
- DCMs
- Clock logic
- Macros

LOCATE Description

LOCATE is a basic placement constraint that specifies a single location, multiple single locations, or a location range.

LOCATE Propagation Rules

When attached to a macro, the constraint propagates to all elements of the macro. When attached to a primitive, the constraint applies to the entire primitive.

LOCATE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

PCF Syntax Examples

Single or multiple single locations

COMP "*comp_name*" **LOCATE=**[**SOFT**] "*site_item1*"... "*site_itemn*" [**LEVEL** *n*];
COMPGRP "*group_name*" **LOCATE=**[**SOFT**] "*site_item1*"... "*site_itemn*" [**LEVEL** *n*];
MACRO *name* **LOCATE=**[**SOFT**] "*site_item1*"... "*site_itemn*" [**LEVEL** *n*];

Range of locations

COMP "*comp_name*" **LOCATE=**[**SOFT**] **SITE** "*site_name*" **:** **SITE** "*site_name*" [**LEVEL** *n*];
COMPGRP "*group_name*" **LOCATE=**[**SOFT**] **SITE** "*site_name*" **:** **SITE** "*site_name*" [**LEVEL** *n*];
MACRO "*macro_name*" **LOCATE=**[**SOFT**] **SITE** "*site_name*" **:** **SITE** "*site_name*" [**LEVEL** *n*];

where

- *site_name* is a component site (that is, a CLB or IOB location)
- *site_item* is one of the following:
  - **SITE** "*site_name*"

- **SITEGRP** "*site_group_name*"

- *n* in LEVEL n is 0, 1, 2, 3, or 4

Lock Pins (LOCK_PINS)

LOCK_PINS Architecture Support

The LOCK_PINS constraint applies to FPGA devices only.

LOCK_PINS Applicable Elements

The LOCK_PINS constraint is applied only to specific instances of LUT symbols.

LOCK_PINS Description

The LOCK_PINS constraint instructs the implementation tools to not swap the pins of the LUT symbol to which it is attached. The LOCK_PINS constraint should not be confused with the Lock Pins process in Project Navigator, which is used to preserve the existing pinout of a CPLD design.

LOCK_PINS Propagation Rules

LOCK_PINS is applied only to a single LUT instance.

LOCK_PINS Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute lock_pins: string;
```

Specify the VHDL constraint as follows:

```
attribute lock_pins of {component_name|label_name} : {component|label}
is "all";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

```
(* LOCK_PINS = "all" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Examples

Using No Designator

```
INST "XSYM1" LOCK_PINS;
```

Using the ALL Attribute

```
INST "XSYM1" LOCK_PINS='ALL';
```

Using a PIN Assignment List

```
INST I_589 LOCK_PINS=I0:A2;
INST I_894 LOCK_PINS=I3:A1,I2:A4;
INST tvAgy LOCK_PINS=I0:A4,I1:A3,I2:A2,I3:A1;
```

Lookup Table Name (LUTNM)

LUTNM Architecture Support

The LUTNM constraint applies to Virtex™-5 devices only.

LUTNM Applicable Elements

The LUTNM constraint can be applied to two symbols that are unique within the design. The constraint can be applied to two 5-input or smaller function generator symbols (LUT, ROM, or RAM) if the total number of unique input pins required for both symbols does not exceed 5 pins. The constraint can be applied to a 6-input read-only function generator symbol (LUT6, ROM64) in conjunction with a 5-input read-only symbol (LUT5, ROM32) if the total number of unique input pins required for both symbols does not exceed 6 inputs and the lower 32 bits of the 6-input symbol programming matches all 32 bits of the 5-input symbol programming.

LUTNM Description

The LUTNM constraint provides the ability to control the grouping of logical symbols into the LUT sites of the Virtex-5 FPGA architectures. The LUTNM constraint is a string value property that is applied to two qualified symbols. The LUTNM constraint value must be applied uniquely to two symbols within the design. These two symbols are implemented in a shared LUT site within a SLICE component.

This constraint is functionally similar to the Block Name (BLKNM) constraint.

LUTNM Propagation Rules

The LUTNM constraint can be applied to two symbols that are unique within the design.

LUTNM Syntax Examples

Following are syntax examples using this constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid element or symbol type
- Attribute Name: LUTNM
- Attribute Value: <user_defined>

VHDL Syntax Example

Before using LUTNM, declare it with the following syntax placed after the architecture declaration, but before the begin statement in the top-level VHDL file:

```
attribute LUTNM: string;
```

After LUTNM has been declared, specify the VHDL constraint as follows:

```
attribute LUTNM of {LUT5_instance_name}: label is "value";
```

Where value is any chosen name under which you want to group the two elements.

Example:

```
architecture MY_DESIGN of top is
attribute LUTNM: string;
   attribute LUTNM of LUT5_inst1: label is "logic_group1";
   attribute LUTNM of LUT5_inst2: label is "logic_group1";
begin
-- LUT5: 5-input Look-Up Table
   --       Virtex-5
   -- Xilinx HDL Libraries Guide version 8.2i
LUT5_inst1 : LUT5
   generic map (
      INIT => X"a49b44c1")
   port map (
      O  => aout,  -- LUT output (1-bit)
      I0 => d(0),  -- LUT input (1-bit)
      I1 => d(1),  -- LUT input (1-bit)
      I2 => d(2),  -- LUT input (1-bit)
      I3 => d(3),  -- LUT input (1-bit)
      I4 => d(4)   -- LUT input (1-bit)
   );
-- End of LUT5_inst1 instantiation
-- LUT5: 5-input Look-Up Table
   --       Virtex-5
   -- Xilinx HDL Libraries Guide version 8.2i
LUT5_inst2 : LUT5
   generic map (
      INIT => X"649d610a")
   port map (
      O  => bout,  -- LUT output (1-bit)
      I0 => d(0),  -- LUT input (1-bit)
      I1 => d(1),  -- LUT input (1-bit)
      I2 => d(2),  -- LUT input (1-bit)
      I3 => d(3),  -- LUT input (1-bit)
      I4 => d(4)   -- LUT input (1-bit)
   );
-- End of LUT5_inst2 instantiation
END MY_DESIGN;
```

For a more detailed discussion of the basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Place the following attribute specification before the port declaration in the top-level Verilog code:

```
(* LUTNM = "value" *)
```

Where *value* is any chosen name under which you want to group the two elements.

Example:

```
// LUT5: 5-input Look-Up Table
//       Virtex-5
// Xilinx HDL Libraries Guide version 8.2i
(* LUTNM="logic_group1" *) LUT5 #(
   .INIT(32'ha49b44c1)
) LUT5_inst1 (
   .O(aout),   // LUT output (1-bit)
   .I0(d[0]),  // LUT input (1-bit)
   .I1(d[1]),  // LUT input (1-bit)
   .I2(d[2]),  // LUT input (1-bit)
   .I3(d[3]),  // LUT input (1-bit)
   .I4(d[4])   // LUT input (1-bit)
);
// End of LUT5_inst1 instantiation
// LUT5: 5-input Look-Up Table
//       Virtex-5
// Xilinx HDL Libraries Guide version 8.2i
(* LUTNM="logic_group1" *) LUT5 #(
   .INIT(32'h649d610a)
) LUT5_inst2 (
   .O(bout),   // LUT output (1-bit)
   .I0(d[0]),  // LUT input (1-bit)
   .I1(d[1]),  // LUT input (1-bit)
   .I2(d[2]),  // LUT input (1-bit)
   .I3(d[3]),  // LUT input (1-bit)
   .I4(d[4])   // LUT input (1-bit)
);
// End of LUT5_inst2 instantiation
```

For a more detailed discussion of the basic Verilog syntax, see "Verilog" in Chapter 3.

UCF/NCF Syntax Example

Placed on the output, or bi-directional port:

```
INST "LUT5_instance_name" LUTNM="value";
```

Where *value* is any chosen name under which you want to group the two elements.

Example:

```
INST "LUT5_inst1" LUTNM="logic_group1";
INST "LUT5_inst2" LUTNM="logic_group1";
```

Map (MAP)

MAP Architecture Support

The MAP constraint applies to FPGA devices only.

MAP Applicable Elements

FMAP

MAP Description

MAP is an advanced mapping constraint. Place MAP on an FMAP to specify whether pin swapping and the merging of other functions with the logic in the map are allowed. If merging with other functions is allowed, other logic can also be placed within the CLB, if space allows.

MAP Propagation Rules

Applies to the design element to which it is attached

MAP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Examples

The basic UCF syntax is:

```
INST "instance_name" MAP=[PUC | PUO | PLC | PLO];
```

where

the terms have the following meanings:

- PUC

  The CLB pins are unlocked (U) and the CLB is closed (C).
- PUO

  The CLB pins are unlocked (U) and the CLB is open (O).
- PLC

  The CLB pins are locked (L) and the CLB is closed (C).
- PLO

  The CLB pins are locked (L) and the CLB is open (O).

The default is PUO. Currently, only PUC and PUO are observed. PLC and PLO are translated into PUC and PUO, respectively.

As used in these definitions, the following terms have the meanings indicated.

- Unlocked

  The software *can* swap signals among the pins on the CLB.

- Locked

  The software *cannot* swap signals among the pins on the CLB.

- Open

  The software *can* add or remove logic from the CLB.

- Closed

  The software *cannot* add or remove logic from the function specified by the MAP symbol.

The following statement allows pin swapping, and ensures that no logic other than that defined by the original map is mapped into the function generators.

```
INST "$1I3245/map_of_the_world" map=puc;
```

Maximum Delay (MAXDELAY)

MAXDELAY Architecture Support

The MAXDELAY constraint applies to FPGA devices only.

MAXDELAY Applicable Elements

Nets

MAXDELAY Description

The MAXDELAY attribute defines the maximum allowable delay on a net.

MAXDELAY Propagation Rules

Applies to the net to which it is attached

MAXDELAY Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: MAXDELAY
- Attribute Values: *value units*

  where

  - *value* is the numerical time delay
  - *units* are micro, ms, ns, ps

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute maxdelay: string;
```

Specify the VHDL constraint as follows:

```
attribute maxdelay of signal_name: signal is "value [units]";
```

where

- *value* is a positive integer

Valid units are ps, ns, micro, ms, GHz, MHz, and kHz. The default is ns.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Examples

Specify the Verilog constraint as follows:

```
(*MAXDELAY = "value [units]" *)
```

where

- *value* is a positive integer

Valid units are ps, ns, micro, ms, GHz, MHz, and kHz. The default is ns.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Examples

```
NET "net_name" MAXDELAY=value units;
```

where

- *value* is the numerical time delay.
- *units* are micro, ns, ms, ps.

The following statement assigns a maximum delay of 1 micro to the net $SIG_4.

```
NET "$1I3245/$SIG_4" MAXDELAY=10 ns;
```

PCF Syntax Examples

```
item MAXDELAY = maxvalue [PRIORITY integer];
```

where

- *item* can be:
  - **ALLNETS**
  - **NET** *name*
  - **TIMEGRP** *name*
  - **ALLPATHS**
  - **PATH** *name*
  - *path specification*
- *maxvalue* can be:
  - a numerical time value with units of micro, ms, ps, or ns
  - a numerical frequency value with units of GHz, MHz, or KHz
  - a TSidentifier

FPGA Editor Syntax Examples

To set MAXDELAY to all paths or nets, click Main Properties from the File menu and select the Global Physical Constraints tab.

To set the constraint to a selected path or net, click Properties of Selected Items from the Edit menu with a routed net selected and use the Physical Constraints tab.

Maximum Product Terms (MAXPT)

MAXPT Architecture Support

The MAXPT constraint applies to CPLD devices only.

MAXPT Applicable Elements

Signals

MAXPT Description

MAXPT is an advanced ABEL constraint. It applies to CPLD devices only. MAXPT specifies the maximum number of product terms the fitter is permitted to use when collapsing logic into the node to which MAXPT is applied. MAXPT overrides the Collapsing P-term Limit setting in Project Navigator for the attached node.

MAXPT Propagation Rules

Applies to the signal to which it is attached

MAXPT Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute maxpt: integer;
```

Specify the VHDL constraint as follows:

```
attribute maxpt of signal_name : signal is "integer";
```

where

- *integer* is any positive integer

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* MAXPT = "integer" *)
```

where

- *integer* is any positive integer

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'maxpt=8 mysignal';
```

Valid values are any positive integers.

UCF and NCF Syntax Example

```
Net "signal_name" maxpt=integer;
```

Maximum Skew (MAXSKEW)

MAXSKEW Architecture Support

The MAXSKEW constraint applies to FPGA devices only.

MAXSKEW Applicable Elements

Nets

MAXSKEW Description

MAXSKEW is a timing constraint used to control the amount of skew on a net. Skew is defined as the difference between the delays of all loads driven by the net. You can control the maximum allowable skew on a net by attaching MAXSKEW directly to the net. It is important to understand exactly what MAXSKEW defines. Consider the following example.

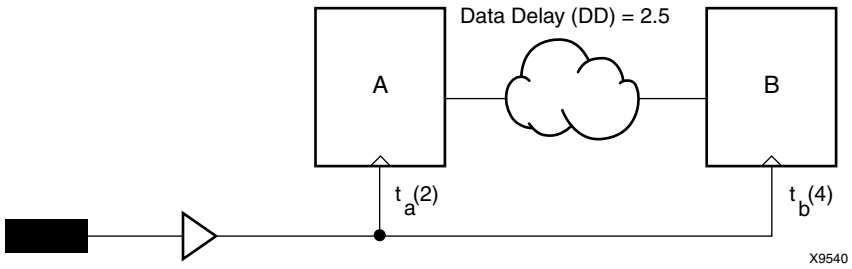

Figure 50-1: **MAXSKEW**

In the preceding diagram, for $t_a(2)$, 2 ns is the maximum delay for the Register A clock. For $t_b(4)$, 4 ns is the maximum delay for the Register B clock. MAXSKEW defines the maximum of t_b minus the maximum of t_a, that is, 4-2=2.

In some cases, relative minimum delays are used on a net for setup and hold timing analysis. When the MAXSKEW constraint is applied to network resources which use relative minimum delays, the MAXSKEW constraint takes relative minimum delays into account in the calculation of skew.

Overuse of this constraint, or too tight of a requirement (value), can cause long PAR runtimes.

MAXSKEW Propagation Rules

Applies to the net to which it is attached

MAXSKEW Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net

- Attribute Name: MAXSKEW
- Attribute Values: *allowable_skew* [*units*]

where

- *allowable_skew* is the timing requirement
- *units* are ms, micro, ns, or ps. The default is ns.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute maxskew: string;
```

Specify the VHDL constraint as follows:

```
attribute maxskew of signal_name : signal is "allowable_skew [units]";
```

where

- *allowable_skew* is the timing requirement
- valid *units* are ms, micro, ns, or ps. The default is ns.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* MAXSKEW = "allowable_skew [units]" *)
```

where

- *allowable_skew* is the timing requirement
- valid *units* are ms, micro, ns, or ps. The default is ns.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

```
NET "net_name" MAXSKEW=allowable_skew [units];
```

where

- *allowable_skew* is the timing requirement
- *units* are ms, micro, ns, or ps. The default is ns.

The following statement specifies a maximum skew of 3 ns on net $SIG_6.

```
NET "$1I3245/$SIG_6" MAXSKEW=3 ns;
```

FPGA Editor Syntax Example

To set constraints in FPGA Editor, select Edit > Properties of Selected Items. With a routed net selected, you can set MAXSKEW from the Physical Constraints tab.

No Delay (NODELAY)

NODELAY Architecture Support

The NODELAY constraint applies to all FPGA devices except Virtex™-4 and Virtex-5.

For Virtex-4 and Virtex-5 devices, the supported constraint is IOBDELAY=NONE. Although NODELAY *can* be used with FPGA devices other than Virtex-4 and Virtex-5 devices, IOBDELAY=NONE is preferable over NODELAY in supported FPGAs. For more information, see "Input Output Block Delay (IOBDELAY)."

NODELAY Applicable Elements

Input register

You can also attach NODELAY to a net connected to a pad component in a UCF file. NGDBuild transfers the constraint from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following UCF syntax:

```
NET "net_name" NODELAY;
```

NODELAY Description

NODELAY is an advanced mapping constraint. The default configuration of IOB flip-flops in designs includes an input delay that results in no external hold time on the input data path. This delay can be removed by placing NODELAY on input flip-flops or latches, resulting in a smaller setup time but a positive hold time.

The input delay element is active in the default configuration for Spartan-II, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-II, Virtex-II Pro, and Virtex-II Pro X.

NODELAY can be attached to the I/O symbols and the special function access symbols TDI, TMS, and TCK.

NODELAY Propagation Rules

NODELAY is illegal when attached to a net or signal except when the net or signal is connected to a pad. In this case, NODELAY is treated as attached to the pad instance.

When attached to a design element, NODELAY is propagated to all applicable elements in the hierarchy within the design element.

NODELAY Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: NODELAY
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute nodelay: string;
```

Specify the VHDL constraint as follows:

```
attribute nodelay of {component_name|signal_name|label_name}:
{component|signal|label} is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* NODELAY = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The following statement specifies that IOB register inreg67 not have an input delay.

```
INST "$1I87/inreg67" NODELAY;
```

The following statement specifies that there be no input delay to the pad that is attached to net1.

```
NET "net1" NODELAY;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" nodelay=true;
 INST "instance_name" nodelay=true;
END;
```

No Reduce (NOREDUCE)

NOREDUCE Architecture Support

The NOREDUCE constraint applies to CPLD devices only.

NOREDUCE Applicable Elements

Any net

NOREDUCE Description

NOREDUCE is a fitter and synthesis constraint. It prevents minimization of redundant logic terms that are typically included in a design to avoid logic hazards or race conditions. NOREDUCE also identifies the output node of a combinatorial feedback loop to ensure correct mapping. When constructing combinatorial feedback latches in a design, always apply NOREDUCE to the latch's output net and include redundant logic terms when necessary to avoid race conditions.

NOREDUCE Propagation Rules

NOREDUCE is a net or signal constraint. Any attachment to a design element is illegal.

NOREDUCE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: NOREDUCE
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute NOREDUCE: string;
```

Specify the VHDL constraint as follows:

```
attribute NOREDUCE of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* NOREDUCE = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
mysignal NODE istype 'retain';
```

UCF and NCF Syntax Example

The following statement specifies that there be no Boolean logic reduction or logic collapse from the net named $SIG_12 forward.

```
NET "$SIG_12" NOREDUCE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" noreduce={true|false};
END;
```

Offset In (OFFSET IN)

OFFSET IN Architecture Support

The OFFSET IN constraint applies to all FPGA and CPLD devices.

OFFSET IN Applicable Elements

- Global
- Net-Specific
- Pad Time Group

OFFSET IN Description

The OFFSET IN constraint is used to specify the timing requirements of an input interface to the FPGA. The constraint specifies the clock and data timing relationship at the external pads of the FPGA. An OFFSET IN constraint specification checks the setup and hold timing requirements of all synchronous elements associated with the constraint. The following image shows the paths covered by the OFFSET IN constraint.

The OFFSET IN constraint is specified using a clock net name. The clock net associated with the OFFSET IN constraint is the external clock pad. Because the constraint specifies the clock and data relationship at the external pads of the FPGA, the OFFSET IN constraint cannot be specified using an internal clock net. However, the OFFSET IN constraint automatically accounts for any phase or delay adjustments on the clock path due to components such as the DCM, PLL, or IDELAY when analyzing the setup and hold timing requirements at the capturing synchronous element. In addition, the constraint propagates through the clock network and automatically applies to all clocks derived from the original external clock.

The OFFSET IN constraint is global in scope by default. In the global OFFSET IN constraint, all synchronous elements that are clocked by the specified clock net, and capture external data, are covered by the constraint. The scope of the synchronous elements covered by the constraint can be restricted by specifying time groups on a subset of input data pads, a subset of the capturing synchronous elements, or both.

OFFSET IN Syntax

Global Method:

The global method is the default OFFSET IN constraint. The global OFFSET IN constraint applies to all synchronous elements that capture incoming data and are triggered by the specified clock signal.

UCF Syntax:

```
OFFSET = IN "offset_time" [units] [VALID <datavalid_time> [UNITS]]
{BEFORE|AFTER} "clk_name" [{RISING | FALLING}];
```

PCF Syntax:

```
OFFSET = IN "offset_time" [units] [VALID <datavalid_time> [UNITS]]
{BEFORE|AFTER} COMP "clk_iob_name" [{RISING | FALLING}];
```

Where:

- *"offset_time" [units]* is the difference in time between the capturing clock edge and the start of the data to be captured. The time can be specified with or without explicitly declaring the units. If no units are specified, the default value is nanoseconds. The valid values for this parameter are: ps, ns, micro, and ms.
- *[VALID <datavalid_time> [UNITS]]* is the valid duration of the data to be captured. This field is required for an accurate hold time verification of the input interface. This value can be specified with or without explicitly declaring the units. If no units are specified, the default value is nanoseconds. The valid values for this field are: ps, ns, micro, and ms.
- *BEFORE|AFTER* defines the timing relationship of the start of data to the clock edge. The best method of defining the clock and data relationship is to use the BEFORE option. BEFORE describes the time the data begins to be valid relative to the capturing clock edge. Positive values of BEFORE indicate the data begins prior to the capturing clock edge. Negative values of BEFORE indicate the data begins following the capturing clock edge.
- *"clk_name"* defines the fully hierarchical name of the input clock pad net. This net name must ne
- [{RISING | FALLING}] are the optional keywords used to define the capturing clock edge in which the clock and data relationship is specified against. In addition, these use of these keywords automatically partition rising and falling edge registers in dual data rate (DDR) interfaces into separate groups for analysis.

Input Group Method:

When a group of inputs captured by the same clock have a shared timing requirement, the inputs can be grouped together to create a single timing constraint. The inputs can be grouped together by input signal names using pad groups, or by the synchronous elements using register groups. By grouping separate signals together into a single time group, the memory and runtime of the implementation tools is reduced. In addition, the timing report will contain bus-based skew and clock centering information.

UCF Syntax:

```
[TIMEGRP "pad_groupname"] OFFSET = IN "offset_time" [units]
[VALID <datavalid_time> [UNITS]] {BEFORE|AFTER} "clk_name"
[TIMEGRP "reg_groupname"] [{RISING | FALLING}];
```

PCF Syntax:

```
[TIMEGRP "inputpad_grpname"] OFFSET = IN "offset_time" [units]
[VALID <datavalid_time> [UNITS]] {BEFORE|AFTER} COMP "clk_iob_name"
[TIMEGRP "reg_groupname"] [{RISING | FALLING}];
```

Where:

- [TIMEGRP *"pad_groupname"*] is the optional input pad time group. This time group can be used to limit the scope of the OFFSET IN constraint to only the synchronous elements fed by the input pad nets contained in the timegroup.
- [TIMEGRP *"reg_groupname"*] is the optional synchronous element time group. This time group can be used to limit the scope of the OFFSET IN constraint to only the synchronous elements which capture input data with the specified clock and are contained in the time group.

Net Specific Method:

OFFSET IN can also be used to specify an input constraint for a specific data net in a schematic, a specific input pad net in the UCF, or a specific input component in the PCF file.

Schematic Syntax When Attached to a Net:

```
OFFSET = IN "offset_time" [units]
[VALID <datavalid_time> [UNITS]] {BEFORE|AFTER} "clk_name"
[TIMEGRP "reg_groupname"] [{RISING | FALLING}];
```

UCF Syntax:

```
NET "pad_net_name" OFFSET = IN "offset_time" [units]
[VALID <datavalid_time> [UNITS]] {BEFORE|AFTER} "clk_name"
[TIMEGRP "reg_groupname"] [{RISING | FALLING}];
```

PCF Syntax:

```
COMP "pad_net_name" OFFSET = IN "offset_time" [units]
[VALID <datavalid_time> [UNITS]] {BEFORE|AFTER} COMP "clk_iob_name"
[TIMEGRP "reg_groupname"] [{RISING | FALLING}];
```

where:

- *"pad_net_name"* is the name of the input data net attached to the pad.
- For the definition of the other variables and keywords, see "Global Method" above.
- The PCF specification uses IO Blocks (COMPs) instead of NETs.
- If the IOB COMP name is omitted in the PCF, or the NET name is omitted in the UCF, the OFFSET IN specification is assumed to be global.

OFFSET IN Syntax Examples

The following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it. While UCF examples are provided, the recommended method of specifying the OFFSET IN constraint is using the Constraint Editor software tool.

UCF Source Synchronous DDR Edge Aligned Example:

The Source Synchronous Dual Data Rate (DDR) Edge aligned case consists of an interface where the clock is sent from the transmitting device edge aligned with the data to the FPGA. In a dual data rate interface, data is captured with both the rising and falling clock edges. In the DDR case, separate OFFSET IN constraints must be defined for the rising and falling clock edge registers capturing the data. The use of the RISING and FALLING keywords with the OFFSET IN constraint simplifies this task.

Example Waveform:

In this example a dual data rate interface is shown with a clock period of 5 ns and 50/50 duty cycle. The rising and falling data is valid for 2 ns and is centered in the high and low portion of the clock waveform. This results in a 250 ps margin before and after data valid window.

Rising Edge Constraints:

The rising edge OFFSET IN constraint defines the time that the data becomes valid prior to rising clock edge used to capture the data. In this example, the data becomes valid 250 ps after the rising clock edge. This results in an OFFSET IN BEFORE value of -250 ps with the value negative because it begins after the clock edge. Once the data begins, it remains valid for 2 ns. This results in a VALID value of 2 ns. The RISING keyword is used with this constraint to indicate that the constraint applies to only the rising edge synchronous elements, and that the OFFSET IN BEFORE value is specified to the rising clock edge.

Falling Edge Constraints:

The falling edge OFFSET IN constraint defines the time that the data becomes valid prior to falling clock edge used to capture the data. In this example, the data becomes valid 250 ps after the falling clock edge. This results in an OFFSET IN BEFORE value of -250 ps with the value negative because it begins after the clock edge. Once the data begins, it remains valid for 2 ns. This results in a VALID value of 2 ns. The FALLING keyword is used with this constraint to indicate that the constraint applies to only the falling edge synchronous elements, and that the OFFSET IN BEFORE value is specified to the falling clock edge.

UCF Syntax Example:

The complete UCF syntax of the clock PERIOD and OFFSET IN constraint for the example is shown below.

```
NET "clock" TNM<_NET = clock;
TIMESPEC TS_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN -250 ps VALID 2 ns BEFORE clock RISING;
OFFSET = IN -250 ps VALID 2 ns BEFORE clock FALLING;
```

UCF Source Synchronous DDR Center Aligned Example:

The Source Synchronous Dual Data Rate (DDR) Center aligned case consists of an interface where the clock is sent from the transmitting device aligned with the center of the data. In a dual data rate interface, data is captured with both the rising and falling clock edges. In the DDR case, separate OFFSET IN constraints must be defined for the rising and falling clock edge registers capturing the data. The use of the RISING and FALLING keywords with the OFFSET IN constraint simplifies this task.

Example Waveform:

In this example a dual data rate interface is shown with a clock period of 5 ns and 50/50 duty cycle. The rising and falling data is valid for 2 ns and is centered over the high and low clock edges. This results in a 250 ps margin before and after data valid window.

Rising Edge Constraints:

The rising edge OFFSET IN constraint defines the time that the data becomes valid prior to rising clock edge used to capture the data. In this example, the data becomes valid 1 ns before the rising clock edge. This results in an OFFSET IN BEFORE value of 1 ns with the value positive because it begins before the clock edge. Once the data begins, it remains valid for 2 ns. This results in a VALID value of 2 ns. The RISING keyword is used with this constraint to indicate that the constraint applies to only the rising edge synchronous elements, and that the OFFSET IN BEFORE value is specified to the rising clock edge.

Falling Edge Constraints:

The falling edge OFFSET IN constraint defines the time that the data becomes valid prior to falling clock edge used to capture the data. In this example, the data becomes valid 1 ns before the falling clock edge. This results in an OFFSET IN BEFORE value of 1 ns with the value positive because it begins before the clock edge. Once the data begins, it remains valid for 2 ns. This results in a VALID value of 2 ns. The FALLING keyword is used with this constraint to indicate that the constraint applies to only the falling edge synchronous elements, and that the OFFSET IN BEFORE value is specified to the falling clock edge.

UCF Syntax Example:

The complete UCF syntax of the clock PERIOD and OFFSET IN constraint for the example is shown below.

```
NET "clock" TNM<_NET = clock;
TIMESPEC TS_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN 1 ns VALID 2 ns BEFORE clock RISING;
OFFSET = IN 1 ns VALID 2 ns BEFORE clock FALLING;
```

UCF System Synchronous SDR Example:

The System Synchronous Single Data Rate (SDR) case consists of an interface where the clock is sent from the transmitting device with one clock edge and captured by the FPGA with the next clock edge. In the single data rate interface data is sent once per clock cycle and requires only one OFFSET IN constraint.

Example Waveform:

In this example a single data rate interface is shown with a clock period of 5 ns and 50/50 duty cycle. The data is valid for 4 ns and begins 500 ps after the transmitting clock edge.

Input Constraints:

The OFFSET IN constraint defines the time that the data becomes valid prior to rising clock edge used to capture the data. In this example, the data becomes valid 500 ps after the transmitting clock edge, or 4.5 ns before the clock edge used to capture the data. This results in an OFFSET IN BEFORE value of 4.5 ns with the value positive because it begins before the clock edge. Once the data begins, it remains valid for 4 ns. This results in a VALID value of 4 ns.

UCF Syntax Example:

The complete UCF syntax of the clock PERIOD and OFFSET IN constraint for the example is shown below.

```
NET "clock" TNM<_NET = clock;
TIMESPEC TS_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = IN 4.5 ns VALID 4 ns BEFORE clock;
```

Schematic Syntax Example

- Attach to a specific net
- Attribute Name: OFFSET
- Attribute Values: IN | OUT *offset_time* BEFORE | AFTER *clk_pad_netname*

VHDL Syntax Example:

Not applicable.

Verilog Syntax Example:

Not applicable.

XCF Syntax Example:

The XCF syntax is the same as the UCF syntax. However, the XCF syntax only supports OFFSET IN BEFORE method.

Offset Out (OFFSET OUT)

OFFSET OUT Architecture Support

The OFFSET OUT constraint applies to all FPGA and CPLD devices.

OFFSET OUT Applicable Elements

- Global
- Nets
- Time groups

OFFSET OUT Description

The OFFSET OUT constraint is used to specify the timing requirements of an output interface from the FPGA. The constraint specifies the time from the clock edge at the input pin of the FPGA until data becomes valid at the outp pin of the FPGA.

The OFFSET OUT constraint is specified using a clock net name. The clock net associated with the OFFSET OUT constraint is the external clock pad. Because the constraint specifies the time from the clock edge at the input pin of the FPGA to the data at the output pin of the FPGA, the OFFSET OUT constraint cannot be specified using an internal clock net. However, the OFFSET OUT constraint automatically accounts for any phase or delay adjustments on the clock path due to components such as the DCM, PLL, or IDELAY when analyzing the output timing requirements. In addition, the constraint propagates through the clock network and automatically applies to all clocks derived from the original external clock.

The OFFSET OUT constraint is global in scope by default. In the global OFFSET OUT constraint, all synchronous elements that are clocked by the specified clock net, and transmit external data, are covered by the constraint. The scope of the synchronous elements covered by the constraint can be restricted by specifying time groups on a subset of output data pads, a subset of the transmitting synchronous elements, or both.

OFFSET OUT Syntax

Global Method:

The global method is the default OFFSET OUT constraint. The global OFFSET OUT constraint applies to all synchronous elements that transmit outgoing data and are triggered by the specified clock signal.

UCF Syntax:

```
OFFSET = OUT "offset_time" [units] {BEFORE|AFTER} "clk_name"
[REFERENCE_PIN "ref_pin"] [{RISING | FALLING}];
```

PCF Syntax:

```
OFFSET = OUT "offset_time" [units] {BEFORE|AFTER} COMP "clk_iob_name"
[REFERENCE_PIN "ref_pin"] [{RISING | FALLING}];
```

Where:

- *["offset_time" [units]]* is and optional parameter that defines the time from the clock edge at the input pin of the FPGA until data first becomes valid at the data output pin of the FPGA. If an "offset_time" value is specified, a timing constraint will be applied to these paths, and errors against that constraint will be reported. If the "offset_time" is omitted, a timing constraint will not be generated, however, the output timing and bus skew of the interface will reported. This report only option is best used in source synchronous interfaces where the clock to output time is of a lesser concern that the skew of the output bus.
- BEFORE|AFTER defines the timing relationship from the clock edge to the start of data. The best method of defining the clock and data requirement is to use the AFTER option. AFTER describes the time the data begins to be valid after the clock edge at the pin of the FPGA.
- *"clk_name"* defines the fully hierarchical name of the input clock pad net.
- [REFERENCE_PIN "ref_pin"] is an optional keyword that is most commonly used in source synchronous output interfaces where the clock is regenerated and sent with the data. The REFERENCE_PIN keyword allows a bus skew analysis of the output signals relative the "ref_pin" signal. If the REFERENCE_PIN keyword is not specified, the bus skew report will be referenced to the signal with the minimum clock to output delay.
- [{RISING | FALLING}] are the optional keywords used to define the transmitting clock edge of the synchronous elements sending the data. In addition, these use of these keywords automatically partitions rising and falling edge registers in dual data rate (DDR) interfaces into separate groups for analysis.

Output Group Method:

When a group of output transmitted by the same clock have a shared timing requirement, the outputs can be grouped together to create a single timing constraint. The outputs can be grouped together by output signal names using pad groups, or by synchronous elements using register groups. By grouping separate signals together into a single time group, the memory and runtime of the implementation tools is reduced. In addition, the timing report will contain bus-based skew and clock centering information.

UCF Syntax:

```
[TIMEGRP "pad_groupname"] OFFSET = OUT "offset_time" [units]
{BEFORE|AFTER} "clk_name" [TIMEGRP "reg_groupname"] [REFERENCE_PIN
"ref_pin"] [{RISING | FALLING}];
```

PCF Syntax:

```
[TIMEGRP "pad_groupname"] OFFSET = OUT "offset_time" [units]
{BEFORE|AFTER} COMP "clk_iob_name" [TIMEGRP "reg_groupname"]
[REFERENCE_PIN "ref_pin"] [{RISING | FALLING}];
```

where:

- The group specific method is identical to the general method with the additions noted below. For the definition of the other variables and keywords, see "Global Method" above.

- [TIMEGRP *"pad_groupname"*] is the optional output pad time group. This time group can be used to limit the scope of the OFFSET OUT constraint to only the synchronous elements feeding the output pad nets contained in the time group.
- [TIMEGRP *"reg_groupname"*] is the optional synchronous element time group. This time group can be used to limit the scope of the OFFSET OUT constraint to only the synchronous elements which transmit output data with the specified clock and are contained in the time group.

Net Specific Method:

OFFSET OUT can also be used to specify an output constraint for a specific data net in a schematic, a specific output pad net in the UCF, or a specific output component in the PCF file.

Schematic Syntax When Attached to a Net:

```
OFFSET = OUT "offset_time" [units]
{BEFORE|AFTER} "clk_name" [TIMEGRP "reg_groupname"] [REFERENCE_PIN
"ref_pin"] [{RISING | FALLING}];
```

UCF Syntax:

```
NET "pad_net_name" OFFSET = OUT "offset_time" [units]
{BEFORE|AFTER} "clk_name" [TIMEGRP "reg_groupname"] [REFERENCE_PIN
"ref_pin"] [{RISING | FALLING}];
```

PCF Syntax:

```
COMP "pad_net_name" OFFSET = OUT "offset_time" [units]
{BEFORE|AFTER} "clk_name" [TIMEGRP "reg_groupname"] [REFERENCE_PIN
"ref_pin"] [{RISING | FALLING}];
```

where:

- The group specific method is identical to the general method with the additions noted below. For the definition of the other variables and keywords, see "Global Method" above.
- *"pad_net_name"* is the name of the output data net attached to the pad.
- The PCF specification uses IO Blocks (COMPs) instead of NETs.
- If the IOB COMP name is omitted in the PCF, or the NET name is omitted in the UCF, the OFFSET OUT specification is assumed to be global.

OFFSET OUT Syntax Examples

The following are syntax examples covering the common use cases of the constraint. While UCF examples are provided, the recommended method of specifying the OFFSET OUT constraint is using the Constraint Editor software tool.

UCF Source Synchronous DDR Example:

The Source Synchronous Dual Data Rate (DDR) case consists of an interface where the clock is regenerated inside the FPGA and sent with the data to the capturing device. In a DDR interface, data is transmitted with both the rising and falling clock edges. In the DDR case, separate OFFSET OUT constraints must be defined for the rising and falling clock edge registers transmitting the data. The use of the RISING and FALLING keywords with

the OFFSET OUT constraint simplifies this task. Also, for a bus skew analysis relative to the regenerated clock, the REFERENCE_PIN keyword is used.

Interface Information:

In this example a clock signal called "clock" enters the FPGA. This clock signal is used to trigger the data output synchronous elements. In addition, a regenerated clock called "TxClock" is created and sent along with the data. Because this is a source synchronous interface, the absolute clock to output time is not required, and the OFFSET OUT AFTER value is omitted to generate a report only constraint.

UCF Syntax Example:

The complete UCF syntax of the clock PERIOD and OFFSET OUT constraint for the example is shown below.

```
NET "clock" TNM_NET = clock;
TIMESPEC TS_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = OUT AFTER clock REFERENCE_PIN "TxClock" RISING;
OFFSET = OUT AFTER clock REFERENCE_PIN "TxClock" FALLING;
```

UCF System Synchronous SDR Example:

The System Synchronous Single Data Rate (SDR) case consists of an interface where the input clock is used to transmit the data to the receiving device. In the SDR interface, data is transmitted once per clock cycle. In this case a single OFFSET OUT requirement is needed to constrain the interface.

Interface Information:

In this example a clock signal called "clock" enters the FPGA. This clock signal is used to trigger the data output synchronous elements. Because this is a system synchronous interface, the absolute clock to output time is required to constraint the interface. In this case, a regenerated clock is not present, and the REFERENCE_PIN keyword is omitted to request the default skew reporting.

UCF Syntax Example:

The complete UCF syntax of the clock PERIOD and OFFSET OUT constraint for the example is shown below.

```
NET "clock" TNM_NET = clock;
TIMESPEC TS_CLK = PERIOD CLK 5.0 ns HIGH 50%;

OFFSET = OUT 5 ns AFTER "clock";
```

Schematic Syntax Example

- Attach to a specific net
- Attribute Name: OFFSET
- Attribute Values: OUT *offset_time* BEFORE | AFTER *clk_pad_netname*

VHDL Syntax Example:

Not applicable.

Verilog Syntax Example:

Not applicable.

XCF Syntax Example:

The XCF syntax is the same as the UCF syntax. However, the XCF syntax only supports OFFSET OUT AFTER method.

Open Drain (OPEN_DRAIN)

OPEN_DRAIN Architecture Support

The OPEN_DRAIN constraint applies to Coolrunner™-II devices only.

OPEN_DRAIN Applicable Elements

- Output pads
- Pad nets

OPEN_DRAIN Description

CoolRunner-II outputs can be configured to drive the primary macrocell output function as an open-drain output signal on the pin. The OPEN_DRAIN constraint applies to non 3-state (always active) outputs in the design. The output structure is configured as open-drain so that a one state on the output signal in the design produces a high-Z on the device pin instead of a driven High voltage.

The high-Z behavior associated with the OPEN_DRAIN constraint is not exhibited during functional simulation, but is represented accurately during post-fit timing simulation.

The logically-equivalent alternative to using the OPEN_DRAIN constraint is to take the original output-pad signal in the design and use it as a 3-state disable for a constant-zero output data value. The CPLD Fitter automatically optimizes all 3-state outputs with constant-zero data value in the design to take advantage of the open-drain capability of the device.

OPEN_DRAIN Propagation Rules

The constraint is a net or signal constraint. Any attachment to a macro, entity, or module is illegal.

OPEN_DRAIN Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an output pad net
- Attribute Name: OPEN_DRAIN
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute OPEN_DRAIN: string;
```

Specify the VHDL constraint as follows:

```
attribute OPEN_DRAIN of signal_name : signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* OPEN_DRAIN = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'OPEN_DRAIN mysignal';
```

UCF and NCF Syntax Example

```
NET "mysignal" OPEN_DRAIN;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" OPEN_DRAIN=true;
END;
```

Optimizer Effort (OPT_EFFORT)

OPT_EFFORT Architecture Support

The OPT_EFFORT constraint applies to FPGA devices only.

OPT_EFFORT Applicable Elements

Any macro or hierarchy level

OPT_EFFORT Description

OPT_EFFORT is a basic placement and routing constraint. It defines an effort level used by the optimizer.

OPT_EFFORT Propagation Rules

OPT_EFFORT is a macro, entity, module constraint. Any attachment to a net or signal is illegal.

OPT_EFFORT Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a macro
- Attribute Name: OPT_EFFORT
- Attribute Values: Default (Low), Lowest, Low, Normal, High, Highest

UCF and NCF Syntax Example

The following statement attaches a High effort of optimization to all of the logic contained within the module defined by instance $1I678/adder.

```
INST "$1I678/adder" OPT_EFFORT=HIGH;
```

Project Navigator Syntax Example

Define globally with the Place and Route Effort Level (Overall) option in the Place and Route Properties tab of the Process Properties dialog box in the Project Navigator. The default is Low.

With a design selected in the Sources window, right-click Implement Design in the Processes window to access the appropriate Process Properties dialog box.

Optimize (OPTIMIZE)

OPTIMIZE Architecture Support

The OPTIMIZE constraint applies to FPGA devices only.

OPTIMIZE Applicable Elements

Any macro, entity, module or hierarchy level

OPTIMIZE Description

OPTIMIZE is a basic mapping constraint. It defines whether optimization is performed on the flagged hierarchical tree. OPTIMIZE has no effect on any symbol that contains no combinatorial logic, such as an input or output buffer.

OPTIMIZE Propagation Rules

Applies to the macro, entity, or module to which it is attached

OPTIMIZE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a design element
- Attribute Name: OPTIMIZE
- Attribute Values: AREA, SPEED, BALANCE, OFF

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute optimize string;
```

Specify the VHDL constraint as follows:

```
attribute optimize of {entity_name:entity} is
"{AREA|SPEED|BALANCE|OFF}"
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify OPTIMIZE as follows:

```
(* OPTIMIZE = "{AREA|SPEED|BALANCE|OFF}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The following statement specifies that no optimization be performed on an instantiation of the macro CTR_MACRO.

```
INST "/$1I678/CTR_MACRO" OPTIMIZE=OFF;
```

Project Navigator Syntax Example

Define globally with the Optimization Strategy (Cover Mode) option in the Map Properties tab of the Process Properties dialog box in the Project Navigator. The default is Area.

With a design selected in the Sources window, right-click Implement Design in the Processes window to access the appropriate Process Properties dialog box.

Period (PERIOD)

PERIOD Architecture Support

The PERIOD constraint applies to FPGA devices only.

PERIOD Applicable Elements

Nets that feed forward to drive flip-flop clock pins

PERIOD Description

PERIOD is a basic timing constraint and synthesis constraint. A clock period specification checks timing between all synchronous elements within the clock domain as defined in the destination element group. The group may contain paths that pass between clock domains if the clocks are defined as a function of one or the other.

The period specification is attached to the clock net. The definition of a clock period is unlike a FROM-TO style specification because the timing analysis tools automatically take into account any inversions of the clock net at register clock pins, lock phase, and includes all synchronous item types in the analysis. It also checks for hold violations.

A PERIOD constraint on the clock net in the following figure would generate a check for delays on all paths that terminate at a pin that has a setup or hold timing constraint relative to the clock net. This could include the data paths CLB1.Q to CLB2.D, as well as the path EN to CLB2.EC (if the enable were synchronous with respect to the clock).

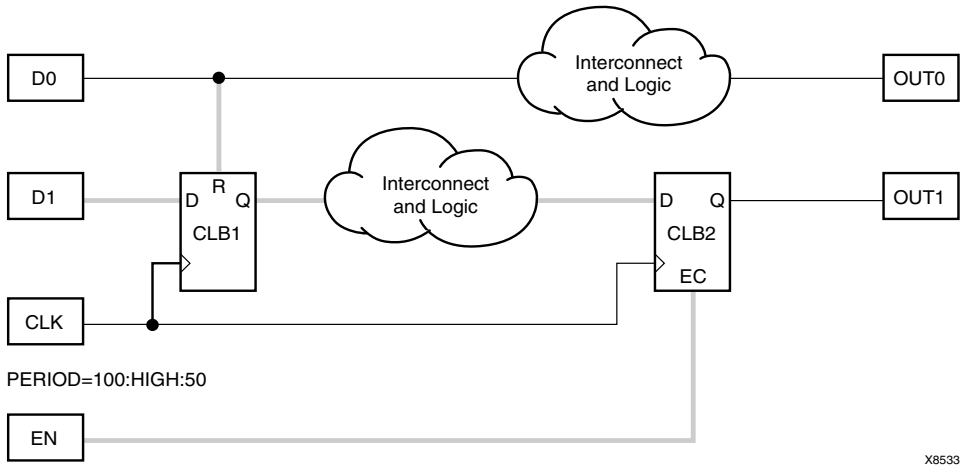

Figure 58-1: **Paths for PERIOD Constraint**

The timing tools do not check pad-to-register paths relative to setup requirements. For example, in the preceding figure, the path from D1 to Pin D of CLB1 is not included in the PERIOD constraint. The same is true for CLOCK_TO_OUT.

Special rules apply when using TNM and TNM_NET with the PERIOD constraint for DLLs, DCMs and PLLs. These rules are explained in "PERIOD Specifications on CLKDLLs, DCMs and PLLs."

Preferred Method

The preferred method for defining a clock period allows more complex derivative relationships to be defined as well as a simple clock period. The following constraint is defined using the TIMESPEC keyword in conjunction with a TNM constraint attached to the relevant clock net.

UCF Syntax Example

```
TIMESPEC "TSidentifier"=PERIOD "TNM_reference" period {HIGH | LOW}
[high_or_low_time] INPUT_JITTER value;
```

where

- *identifier* is a reference identifier that has a unique name
- *TNM_reference* identifies the group of elements to which the period constraint applies. This is typically the name of a TNM_NET that was attached to a clock net, but it can be any TNM group or user group (TIMEGRP) that contains only synchronous elements.

The following rules apply:

- The variable name *period* is the required clock period.
- The default units for *period* are nanoseconds, but the number can be followed by ps, ns, micro, or ms. The *period* can also be specified as a frequency value, using units of MHz, GHz, or kHZ.
- Units may be entered with or without a leading space.
- Units are case-insensitive.
- The **HIGH | LOW** keyword indicates whether the first pulse in the period is high or low, and the optional *high_or_low_time* is the polarity of the first pulse. This defines the initial clock edge and is used in the OFFSET constraint. HIGH is the default logic level if the logic level is not specified.
- If an actual time is specified, it must be less than the period.
- If no *high_or_low_time* is specified the default duty cycle is 50%.
- The default units for *high_or_low_time* is *ns*, but the number can be followed by % or by *ps, ns, micro, or ms* to specify an actual time measurement.
- INPUT_JITTER is the random, peak-to-peak jitter on an input clock. The default units are picoseconds.

Examples

Clock net sys_clk has the constraint `tnm=master_clk` attached to it and the following constraint is attached to TIMESPEC.

UCF Syntax Examples

```
TIMESPEC TS_master = PERIOD "master_clk" 50 HIGH 30 INPUT_JITTER 50;
```

This period constraint applies to the net master_clk, and defines a clock period of 50 nanoseconds, with an initial 30 nanosecond high time, and INPUT_JITTER at 50 ps.

```
TIMESPEC TS_clkinA = PERIOD "clkinA" 21 ns LOW 50% INPUT_JITTER 500 ps;
```

```
TIMESPEC TS_clkinB = PERIOD "clkinB" 21 ns HIGH 50% INPUT_JITTER 500 ps;
```

Another Method

Another method of defining a clock period is to attach the following constraint directly to a net in the path that drives the register clock pins.

Schematic Syntax Example

```
PERIOD = period {HIGH|LOW} [high_or_low_time] INPUT_JITTER value;
```

UCF Syntax Example

```
NET "net_name" PERIOD = period {HIGH|LOW} [high_or_low_time]
INPUT_JITTER value;
```

The following rules apply:

- *period* is the required clock period. The default units are nanoseconds, but the timing number can be followed by ps, ns, micro, or ms. The *period* can also be specified as a frequency value, using units of MHz, GHz, or kHZ.
- Units may be entered with or without a leading space.
- Units are case-insensitive.
- The **HIGH | LOW** keyword indicates whether the first pulse in the period is high or low, and the optional *high_or_low_time* is the duty cycle of the first pulse. HIGH is the default logic level if the logic level is not specified.
- If an actual time is specified, it must be less than the period.
- If no high or low time is specified the default duty cycle is 50%.
- The default unit for *high_or_low_time* is *ns*, but the number can be followed by % or by *ps, ns, micro* or *ms* to specify an actual time measurement.

The PERIOD constraint is forward traced in exactly the same way a TNM would be and attaches itself to all of the synchronous elements that the forward tracing reaches. If a more complex form of tracing behavior is required (for example, where gated clocks are used in the design), you must place the PERIOD on a particular net or use the preferred method described in the next section.

Specifying Derived Clocks

The preferred method of defining a clock period uses an identifier, allowing another clock period specification to reference it. To define the relationship in the case of a derived clock, use the following syntax:

UCF Syntax Example

```
TIMESPEC "TSidentifier"=PERIOD "timegroup_name" "TSidentifier" [* or /]
factor PHASE [+ |-] phase_value [units];
```

where

- *identifier* is a reference identifier that has a unique name
- *factor* is a floating point number

  Note: You can omit the [* or /] factor if the specification being defined has the same value as the one being referenced (that is, they differ only in phase); this is the same as using "* 1".
- *phase_value* is a floating point number
- *units* are ps, ms, micro, or ns. The default is ns.

The following rules apply:

- If an actual time is specified it must be less than the period.
- If no *high_or_low_time* is specified, the default duty cycle is 50%.
- The default units for *high_or_low_time* is *ns*, but the number can be followed by % or by *ps, ns, micro, or ms* to specify an actual time measurement.

Examples of a Primary Clock with Derived Clocks

Period for primary clock:

```
TIMESPEC "TS01" = PERIOD "clk0" 10.0 ns;
```

Period for clock phase-shifted forward by 180 degrees:

```
TIMESPEC "TS02" = PERIOD "clk180" TS01 PHASE + 5.0 ns;
```

Period for clock phase-shifted backward by 90 degrees:

```
TIMESPEC "TS03" = PERIOD "clk90" TS01 PHASE - 2.5 ns;
```

Period for clock doubled and phase-shifted forward by 180 degrees (which is 90 degrees relative to TS01):

```
TIMESPEC "TS04" = PERIOD "clk180" TS01 / 2 PHASE + 2.5 nS;
```

PERIOD Specifications on CLKDLLs, DCMs and PLLs

When a TNM or TNM_NET property traces into an input pin on a DLL, DCM or PLL, it is handled as described in the following paragraphs.

The checking and transformations described are performed by the logical TimeSpec processing code, which is run during NGDBuild, or the translate process. (The checking timing specifications status message indicates that the logical TimeSpec processing is being run.) The modifications are saved in the built NGD, used by the Mapper and the Map phase passed through the PCF file to the place and route (PAR) phase and TRACE.

However, note that the data saved in the built NGD is distinct from the original TimeSpec user-applied properties, which are left unchanged by this process. Therefore, the Constraints Editor does not see these new groups or specifications, but sees (and possibly modifies) the original user-applied ones.

Conditions for Transformation

When a TNM_NET property is traced into the CLKIN pin of a DLL, DCM or PLL, the TNM group and its usage are examined. The TNM is pushed through the CLKDLL, DCM or PLL (as described below) only if the following conditions are met:

- The TNM group is used in exactly *one* PERIOD specification.
- The TNM group is *not* used in any FROM-TO or OFFSET specifications.
- The TNM group is *not* referenced in any user group definition.

If any of the above conditions are not met, the TNM is not be pushed through the CLKDLL/DCM/PLL, and a warning message is issued. This does not prevent the TNM from tracing into other elements in the standard fashion, but if it traces nowhere else, and is used in a specification, an error results.

Definition of New PERIOD Specifications

If the CLK0 output on the CLKDLL, DCM or PLL is the only one being used (and neither CLKIN_DIVIDE_BY_2 nor CLKOUT_PHASE_SHIFT=FIXED are used), the original

PERIOD specification is simply transferred to that clock output.

Otherwise, for each clock output pin used on the CLKDLL, DCM or PLL, a new TNM group is created on the connected net, and a new PERIOD specification is created for that group. The following table shows how the new PERIOD specifications is defined, assuming an original PERIOD specification named TS_CLKIN.

Table 58-1: **New PERIOD Specifications**

Output Pin	New PERIOD Specification		
	Period Value	Phase Shift	Duty Cycle
CLK0	TS_CLKIN * 1	none	Copied from TS_CLKIN if DUTY_CYCLE_CORRECTION is FALSE. Otherwise, 50%.
CLK90		PHASE + (*clk0_period* * 1/4)	
CLK180		PHASE + (*clk0_period* * 1/2)	
CLK270		PHASE + (*clk0_period* * 3/4)	
CLK2X	TS_CLKIN / 2	none	50%
CLK2X180		PHASE + (*clk2X_period* * 1/2)	
CLKDV	TS_CLKIN * *clkdv_divide* where *clkdv_divide* is the value of the CLKDV_DIVIDE property (default 2.0)	none	50% except for non-integer divides in high-frequency mode (CLKDLLHF, or DCM with DLL_FREQUENCY_MODE=HIGH): CLKDV_DIVIDE 1.5 33.33% HIGH 2.5 40.00% HIGH 3.5 42.86% HIGH 4.5 44.44% HIGH 5.5 45.45% HIGH 6.5 46.15% HIGH 7.5 46.67% HIGH
CLKFX	TS_CLKIN / *clkfx_factor* where *clkfx_factor* is the value of the CLKFX_MULTIPLY property (default 4.0) divided by the value of the CLKFX_DIVIDE property (default 1.0).	none	
CLKFX180		PHASE + (*clkfx_period* * 1/2)	50%

The Period Value shown in this table assumes that the original specification, TS_CLKIN, is expressed as a time. If TS_CLKIN is expressed as a frequency, the multiply or divide operation is reversed.

If the DCM attribute FIXED_PHASE_SHIFT or VARIABLE_PHASE_SHIFT is used, the amount of phase specified is also included in the PHASE value.

PERIOD Propagation Rules

Applies to the signal to which it is attached

PERIOD Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it. The following examples are for the "simple method."

Schematic Syntax Example

- Attach to a net. Following is an example of the syntax format.
- Attribute Name: PERIOD
- Attribute Values: *period* [*units*] [{**HIGH**|**LOW**} [*high_or_low_time*[*hi_lo_units*]]

VHDL Syntax Example

For XST, PERIOD applies only to a specific clock signal.

Declare the VHDL constraint as follows:

```
attribute period: string;
```

Specify the VHDL constraint as follows:

```
attribute period of signal_name : signal is "period [units]";
```

- *period* is the required clock period
- *units* is an optional field to indicate the units for a clock period. The default is nanoseconds (ns), but the timing number can be followed by ps, ns, or micro to indicate the intended units.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

For XST, PERIOD applies only to a specific clock signal.

Specify the Verilog constraint as follows:

```
(* PERIOD = "period [units]" *)
```

- *period* is the required clock period
- *units* is an optional field to indicate the units for a clock period. The default is nanoseconds (ns), but the timing number can be followed by ps, ns, or micro to indicate the intended units.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

TIMESPEC PERIOD Method (Primary Method) (Recommended)

UCF Syntax

```
TIMESPEC "TSidentifier"=PERIOD "TNM_reference period" [units] [{HIGH | LOW} [high_or_low_time [hi_lo_units]]] INPUT_JITTER value [units];
```

where

- *identifier* is a reference identifier that has a unique name
- *TNM_reference* is the identifier name that is attached to a clock net (or a net in the clock path) using the TNM or TNM_NET constraint

When a TNM_NET constraint is traced into the CLKIN input of a DLL, DCM or PLL component, new PERIOD specifications may be created at the DLL/DCM/PLL outputs. If new PERIOD specifications are created, new TNM_NET groups to use in those specifications are also created.

Each new TNM_NET group is named the same as the corresponding DLL/DCM/PLL output net (*outputnetname*). The new PERIOD specification becomes "TS_*outputnetname*=PERIOD *outputnetname value units*."

The new TNM_NET groups are then traced forward from the DLL/DCM/PLL output net to tag all synchronous elements controlled by that clock signal. The new groups and specifications are shown in the timing analysis reports.

Rules

The following rules apply:

- *period* is the required clock period.
- *units* is an optional field to indicate the units for a clock period. The default is nanoseconds (ns), but the timing number can be followed by ps, ms, micro, or % to indicate the intended units.
- HIGH or LOW indicates whether the first pulse is to be High or Low.
- *high_or_low_time* is the optional High or Low time, depending on the preceding keyword. If an actual time is specified, it must be less than the period. If no *high_or_low_time* is specified, the default duty cycle is 50 percent.
- *hi_lo_units* is an optional field to indicate the units for the duty cycle. The default is nanoseconds (ns), but the *high_or_low_time* number can be followed by ps, micro, ms, or % if the High or Low time is an actual time measurement.

The following statement assigns a clock period of 40 ns to the net named CLOCK, with the first pulse being High and having a duration of 25 nanoseconds.

```
NET "CLOCK" PERIOD=40 HIGH 25;
```

NET PERIOD Method (Secondary Method) (Not Recommended)

```
NET "net_name" PERIOD=period [units] [{HIGH|LOW}
[high_or_low_time[hi_lo_units]]];
```

where

- *period* is the required clock period

- *units* is an optional field to indicate the units for a clock period. The default is nanoseconds (ns), but the timing number can be followed by ps, ns, or micro to indicate the intended units.
- HIGH or LOW can be optionally specified to indicate whether the first pulse is to be High or Low.
- *hi_lo_units* can be ns, ps, or micro. The default is ns.

The following rules apply:

- *high_or_low_time* is the optional High or Low time, depending on the preceding keyword.
- If an actual time is specified, it must be less than the period.
- If no *high_or_low_time* is specified, the default duty cycle is 50 percent.
- *hi_lo_units* is an optional field to indicate the units for the duty cycle.
- The default is nanoseconds (ns), but the *high_or_low_time* number can be followed by ps, micro, ms, or % if the High or Low time is an actual time measurement.

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Global tab grid, double-click the Period column in the row with the desired clock name and fill out the PERIOD dialog box.

XCF Syntax Example

Same as UCF syntax

Both the simple and preferred are supported with the following limitation: HIGH/LOW values are not taken into account during timing estimation/optimization and only propagated to the final netlist if WRITE_TIMING_CONSTRAINTS = yes.

PCF Syntax Example

```
"TSidentifier"=PERIOD periodtem periodvalue INPUT_JITTER value;
```

perioditem can be:

- NET *name*
- TIMEGRP *name*

periodvalue can be:

- TS*identifier* PHASE [+ | -] *time*
- TS*identifier* PHASE *time*
- TS*identifier* PHASE [+ | -] *time* [LOW | HIGH] *time*
- TS*identifier* PHASE *time* [LOW | HIGH] *time*
- TS*identifier* PHASE [+ | -] *time* [LOW | HIGH] *percent*
- TS*identifier* PHASE *time* [LOW | HIGH] *percent*

FPGA Editor Syntax Example

To set constraints, in the FPGA Editor main window, click Properties of Selected Items from the Edit menu. To set PERIOD constraint, click Properties of Selected Items from the Edit menu with a net selected. You can set the constraint from the Physical Constraints tab.

Pin (PIN)

PIN Architecture Support

The PIN constraint applies to FPGA devices only.

PIN Applicable Elements

Nets

PIN Description

The PIN constraint in conjunction with LOC defines a net location.

The PIN/LOC UCF constraint has the following syntax:

```
PIN "module.pin" LOC="location";
```

This UCF constraint is used in creating design flows. This UCF constraint is translated into a COMP/LOCATE constraint in the PCF file. This constraint has the following syntax in the PCF file:

```
COMP "name" LOCATE = SITE "location";
```

This constraint specifies that the pseudo component that is created for the pin on the module should be located in the site location. Pseudo logic is created only when a net connects from a pin on one module to a pin on another module.

PIN Propagation Rules

Not applicable.

PIN Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
PIN “module.pin” LOC=location;
PIN mod.pin TIG;
```

POST_CRC

POST_CRC Architecture Support

The POST_CRC constraint applies to Virtex™-5 and Spartan™-3A devices only.

POST_CRC Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC Description

The POST_CRC constraint enables or disables the configuration logic CRC error detection feature allowing for notification of any possible change to the configuration memory. In the case of Spartan-3A, it also has the affect of reserving the multi-use INIT pin for signaling of a configuration CRC failure. This also allows the banking rules used by PACE, PAR, and BitGen to refrain from using the IOB that drives the INIT pin. During configuration, the INIT pin operates as normal. After configuration, if POST_CRC analysis is enabled, the INIT pin serves as a CRC status pin. If comparison of the real-time computed CRC differs from the pre-computed CRC, a configuration memory change has been detected and the INIT pin is driven low.

The following table lists the values for POST_CRC:

Value	Description
ENABLE	Enables the Post CRC checking feature.
DISABLE	Disables the Post CRC checking features. (Default)

POST_CRC Propagation Rules

This constraint applies to the entire design/device.

POST_CRC Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG POST_CRC = [ENABLE|DISABLE];
```

PCF Syntax Example

```
CONFIG POST_CRC = [ENABLE|DISABLE];
```

POST_CRC_ACTION

POST_CRC_ACTION Architecture Support

The POST_CRC_ACTION constraint applies to Spartan™-3A devices only.

POST_CRC_ACTION Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_ACTION Description

Spartan-3A devices support a configuration logic CRC error detection mode called POST_CRC in which a pre-computed CRC for the configuration bitstream is compared against a CRC computed by internal logic based on periodic readback of the configuration memory cells. POST_CRC_ACTION determines whether a CRC mismatch detection continues or whether the CRC operation is halted. This constraint is only applicable when POST_CRC is set to ENABLE.

The following table lists the values for POST_CRC_ACTION:

Value	Description
HALT	If a CRC mismatch is detected, cease reading back the bitstream, computing the comparison CRC, and making the comparison against the pre-computed CRC.
CONTINUE	If a CRC mismatch is detected by the CRC comparison, continue reading back the bitstream, computing the comparison CRC, and making the comparison against the pre-computed CRC. (Default)

POST_CRC_ACTION Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_ACTION Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG POST_CRC_ACTION = [HALT|CONTINUE];
```

PCF Syntax Example

```
CONFIG POST_CRC_ACTION = [HALT|CONTINUE];
```

POST_CRC_FREQ

POST_CRC_FREQ Architecture Support

The POST_CRC_FREQ constraint applies to Spartan™-3A devices only.

POST_CRC_FREQ Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_FREQ Description

Spartan-3A devices support a configuration logic CRC error detection mode called POST_CRC in which a pre-computed CRC for the configuration bitstream is compared against a CRC computed by internal logic based on periodic readback of the configuration memory cells. POST_CRC_FREQ controls the frequency with which the configuration CRC check is performed within a Spartan-3A device. This constraint is only applicable when POST_CRC is set to ENABLE.

The frequency range represented by these 10 bits are from 1 to 100 MHz, and the steps are 1, 3, 6, 7, 8, 10, 12, 13, 17, 22, 25, 27, 33, 44, 50 and 100 MHz. The default value is 1 MHz.

POST_CRC_FREQ Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_FREQ Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG POST_CRC_FREQ =
[1|3|6|7|8|10|12|13|17|22|25|27|33|44|50|100];
```

PCF Syntax Example

```
CONFIG POST_CRC_FREQ =
[1|3|6|7|8|10|12|13|17|22|25|27|33|44|50|100];
```

POST_CRC_SIGNAL

POST_CRC_SIGNAL Architecture Support

The POST_CRC_SIGNAL constraint applies to Virtex™-5 devices only.

Virtex™	No
Virtex-E	No
Virtex-II	No
Virtex-II Pro	No
Virtex-II Pro X	No
Virtex-4	No
Virtex-5	Yes
Spartan™-II	No
Spartan-IIE	No
Spartan-3	No
Spartan-3A	No
Spartan-3E	No
XC9500™, XC9500XL, XC9500XV	No
CoolRunner™ XPLA3	No
CoolRunner-II	No

POST_CRC_SIGNAL Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_SIGNAL Description

Virtex-5 devices support a configuration logic CRC error detection mode called POST_CRC in which a pre-computed CRC for the configuration bitstream is compared against a CRC computed by internal logic based on periodic readback of the configuration memory cells. POST_CRC_SIGNAL determines whether the Virtex-5 INIT_B pin is a source of the CRC error signal. POST_CRC allows you to disable the INIT_B pin as the readback CRC error status output pin. The error condition is still available from the FRAME_ECC_VIRTEX5 site. This constraint is only applicable when POST_CRC is set to ENABLE.

The following table lists the values for POST_CRC_SIGNAL:

Value	Description
FRAME_ECC_ONLY	Disables the use of the INIT_B pin, with the FRAME_ECC site as the sole source of the CRC error signal.
INIT_AND_FRAME_ECC	Leaves the INIT_B pin enabled as a source of the CRC error signal. (Default)

POST_CRC_SIGNAL Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_SIGNAL Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG POST_CRC_SIGNAL = [FRAME_ECC_ONLY|INIT_AND_FRAME_ECC];
```

PCF Syntax Example

```
CONFIG POST_CRC_SIGNAL = [FRAME_ECC_ONLY|INIT_AND_FRAME_ECC];
```

Priority (PRIORITY)

PRIORITY Architecture Support

The PRIORITY constraint applies to all FPGA and CPLD devices.

PRIORITY Applicable Elements

TIMESPECs

PRIORITY Description

PRIORITY is an advanced timing constraint keyword. There may be situations where there is a conflict between two timing constraints that cover the same path. The lower the PRIORITY value, the higher the priority. This value does not affect which paths are placed and routed first. It only affects which constraint controls the path when two constraints of equal priority cover the same path.

PRIORITY Propagation Rules

Not applicable

PRIORITY Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

Defines the priority of a timing constraint using the following syntax.

```
normal_timespec_syntax PRIORITY integer;
```

where

- *normal_timespec_syntax* is a legal timing specification
- *integer* represents the priority (the smaller the number, the higher the priority)

The number can be positive, negative, or zero, and the value only has meaning when compared with other PRIORITY values. The lower the value, the higher the priority.

```
TIMESPEC "TS01"=FROM "GROUPA" TO "GROUPB" 40 PRIORITY 4;
```

PCF Syntax Example

Same as UCF

Prohibit (PROHIBIT)

PROHIBIT Architecture Support

The PROHIBIT constraint applies to all FPGA and CPLD devices.

PROHIBIT Applicable Elements

Sites

PROHIBIT Description

PROHIBIT is a basic placement constraint that disallows the use of a site within PAR, FPGA Editor, and the CPLD fitter.

Location Types for FPGA Devices

For an FPGA, use the following location types to define the physical location of an element.

Table 65-1: **Location Types for FPGA Devices**

Element Type	Location Specification	Meaning
IOB	P12	IOB location (chip carrier)
	A12	IOB location (pin grid)
	T, B, L, R	Applies to IOBs and indicates edge locations (bottom, left, top, right) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
	LB, RB, LT, RT, BR, TR, BL, TL	Applies to IOBs and indicates half edges (for example, left bottom, right bottom) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
	Bank 0, Bank 1, Bank 2, Bank 3, Bank 4, Bank 5, Bank 6, Bank 7	Applies to IOBs and indicates half edges (banks) for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex, Virtex-E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
CLB	CLB_R4C3 (or .S0 or .S1)	CLB location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
	CLB_R6C8.S0 (or .S1)	Function generator or register slice for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
Slice	SLICE_X22Y3	Slice location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
TBUF	TBUF_R6C7 (or .0 or .1)	TBUF location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
block RAM	RAMB4_R3C1	Block RAM location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
	RAMB16_X2Y56	Block RAM location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices

Table 65-1: **Location Types for FPGA Devices**

Multiplier	MULT18X18_X55Y82	Multiplier location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Global Clock	GCLKBUF0 (or 1, 2, or 3)	Global clock buffer location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
	GCLKPAD0 (or 1, 2, or 3)	Global clock pad location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
	BUFGMUX0P	Global clock buffer location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Delay Locked Loops (DLL)	DLL0 (or 1, 2, or 3)	Delay Locked Loop element location for Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices
Digital Clock Manager (DCM)	DCM_X[A]Y[B]	Digital Clock Manager for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices
Phase Lock Loop (PLL)	PLL_X[A]Y[B]	Phase Lock Loop for Virtex-5.

You can use the wildcard character (*) to replace a single location with a range as shown in the following examples.

CLB_R*C5	Any CLB in column 5 of a Spartan-II, Spartan-IIE, Virtex, and Virtex-E
SLICE_X*Y5	Any slice of a Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, or Virtex-5 device whose Y-coordinate is 5

The following are *not* supported:

- Dot extensions on ranges. For example, LOC=CLB_R0C0:CLB_R5C5.G.
- The wildcard character for Spartan-II, Spartan-IIE, Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, or Virtex-5 global buffer or DLL locations.

Location Types for CPLD Devices

CPLD devices support only the location type *pin_name*

where

- *pin_name* is P*nn* for numeric pin names or *rc* for row-column pin names

PROHIBIT Propagation Rules

It is illegal to attach PROHIBIT to a net, signal, entity, module, or macro.

PROHIBIT Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

In a UCF file, PROHIBIT must be preceded by the keyword CONFIG.

Single Location

```
CONFIG PROHIBIT=location;
```

Multiple Single Locations

```
CONFIG PROHIBIT=location1, location2, ... ,locationn;
```

Range of Locations

```
CONFIG PROHIBIT=location1:location2;
```

where

- *location* is a legal location type for the part type

For more information, see "Location Types for FPGA Devices" and "Location Types for CPLD Devices" in this chapter. For examples of using the location types, see the "Location (LOC)" constraint. CPLD devices do not support the "Range of locations" form of PROHIBIT.

The following statement prohibits use of the site P45.

```
CONFIG PROHIBIT=P45;
```

For CLB-based Row/Column/Slice Designations

The following statement prohibits use of the CLB located in Row 6, Column 8.

```
CONFIG PROHIBIT=CLB_R6C8;
```

The following statement prohibits use of the site TBUF_R5C2.2.

```
CONFIG PROHIBIT=TBUF_R5C2.2;
```

For Slice-based XY Coordinate Designations

The following statement prohibits use of the slice at the SLICE_X6Y8 site.

```
CONFIG PROHIBIT=SLICE_X6Y8;
```

The following statement prohibits use of the TBUF at the TBUF_X6Y2 site.

```
CONFIG PROHIBIT=TBUF_X6Y2;
```

PCF Syntax Example

For single or multiple single locations:

```
COMP "comp_name" PROHIBIT = [SOFT] "site_group"..."site_group";
COMPGRP "group_name" PROHIBIT = [SOFT] "site_group"..."site_group";
MACRO "name" PROHIBIT = [SOFT] "site_group"..."site_group";
```

For a range of locations:

```
COMP "comp_name" PROHIBIT = [SOFT] "site_group"... "site_group";
```

```
COMPGRP "group_name" PROHIBIT = [SOFT] "site_group"... "site_group";
MACRO "name" PROHIBIT = [SOFT] "site_group"..."site_group";
```

where

- *site_group* is one of the following
  - **SITE "***site_name***"**
  - **SITEGRP "***site_group_name***"**
- *site_name* is a component site (that is, a CLB or IOB location)

Floorplanner Syntax Example

The Floorplanner supports PROHIBIT. For more information, see the Prohibit command section in the Floorplanner help.

PACE Syntax Example

The Pin Assignments Editor (PACE) can be used to set PROHIBIT. For more information, see the Prohibit Mode command section in the PACE help.

FPGA Editor Syntax Example

FPGA Editor supports PROHIBIT. For more information, see the Prohibit Constraint topic in the FPGA Editor help. The constraint is written to the PCF file by the Editor.

Pulldown (PULLDOWN)

PULLDOWN Architecture Support

The PULLDOWN constraint applies to all FPGA devices and the Coolrunner™-II CPLD only.

PULLDOWN Applicable Elements

- Input
- Tristate outputs
- Bidirectional pad nets

PULLDOWN Description

PULLDOWN is a basic mapping constraint. It guarantees a logic Low level to allow 3-stated nets to avoid floating when not being driven.

KEEPER, PULLUP, and PULLDOWN are only valid on pad NETs, not on INSTs of any kind.

PULLDOWN Propagation Rules

PULLDOWN is a net constraint. Any attachment to a design element is illegal.

PULLDOWN Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a pad net
- Attribute Name: PULLDOWN
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute PULLDOWN: string;
```

Specify the VHDL constraint as follows:

```
attribute PULLDOWN of signal_name: signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* PULLDOWN = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The following statement configures the IO to use a PULLDOWN.

```
NET "pad_net_name" PULLDOWN;
```

This statement configures PULLDOWN to be used globally.

```
DEFAULT PULLDOWN = TRUE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" pulldown=true;
END;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Ports tab grid with I/O Configuration Options checked, click the PULLUP/PULLDOWN column in the row with the desired port name and choose PULLDOWN from the drop-down list.

Pullup (PULLUP)

PULLUP Architecture Support

The PULLUP constraint applies to all FPGA devices and the Coolrunner™ XPLA3 and Coolrunner-II CPLDs.

PULLUP Applicable Elements

- Input
- Tristate outputs
- Bidirectional pad nets

PULLUP Description

PULLUP is a basic mapping constraint. It guarantees a logic High level to allow 3-stated nets to avoid floating when not being driven.

KEEPER, PULLUP, and PULLDOWN are only valid on pad NETs, not on INSTs of any kind.

For CoolRunner-II designs, the use of KEEPER and the use of PULLUP are mutually exclusive across the whole device.

PULLUP Propagation Rules

PULLUP is a net constraint. Any attachment to a design element is illegal.

PULLUP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a pad net
- Attribute Name: PULLUP
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute PULLUP: string;
```

Specify the VHDL constraint as follows:

```
attribute PULLUP of signal_name: signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* PULLUP = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'PULLUP mysignal';
```

UCF and NCF Syntax Example

The following statement configures the IO to use a PULLUP.

```
NET "pad_net_name" PULLUP;
```

This statement configures PULLUP to be used globally.

```
DEFAULT PULLUP = TRUE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" pullup=true;
END;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Ports tab grid with I/O Configuration Options checked, click the PULLUP/PULLDOWN column in the row with the desired port name and choose PULLUP from the drop-down list.

Power Mode (PWR_MODE)

PWR_MODE Architecture Support

The PWR_MODE constraint applies to the following devices:

- XC9500™
- XC9500XL
- XC9500XV

PWR_MODE Applicable Elements

- Nets
- Any instance

PWR_MODE Description

PWR_MODE is an advanced fitter constraint. It defines the mode, Low power or High performance (standard power), of the macrocell that implements the tagged element.

If the tagged function is collapsed forward into its fanouts, PWR_MODE is not applied.

PWR_MODE Propagation Rules

When attached to a net, PWR_MODE attaches to all applicable elements that drive the net.

When attached to a design element, PWR_MODE propagates to all applicable elements in the hierarchy within the design element.

PWR_MODE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net or an instance
- Attribute Name: PWR_MODE
- Attribute Values: LOW, STD

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute PWR_MODE: string;
```

Specify the VHDL constraint as follows:

```
attribute PWR_MODE of {signal_name|component_name|label_name}:
{signal|component|label} is "{LOW|STD}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* PWR_MODE = "{LOW|STD} *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'pwr_mode={low|std} mysignal';
```

UCF and NCF Syntax Example

The following statement specifies that the macrocell that implements the net $SIG_0 is in Low power mode.

```
NET "$1187/$SIG_0" PWR_MODE=LOW;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" PWR_MODE={LOW|STD};
 INST "instance_name" PWR_MODE={LOW|STD};
END;
```

Registers (REG)

REG Architecture Support

The REG constraint applies to CPLD devices only.

REG Applicable Elements

Registers

REG Description

REG is a basic fitter constraint. It specifies how a register is to be implemented in the CPLD macrocell.

REG Propagation Rules

When attached to a design element, REG propagates to all applicable elements in the hierarchy within the design element.

REG Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a flip-flop instance or macro containing flip-flops
- Attribute Name: REG
- Attribute Values: CE, TFF

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute REG: string;
```

Specify the VHDL constraint as follows:

```
attribute REG of signal_name: signal is "{CE|TFF}";
```

For more information on CE and TFF, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* REG = {CE|TFF} *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'REG={CE|TFF} mysignal';
```

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" REG = {CE | TFF};
```

where

- CE, when applied to a flip-flop primitive with a CE input, forces the CE input to be implemented using a clock enable product term in the macrocell. Normally the fitter uses the register CE input only if all logic on the CE input can be implemented using the single CE product term. Otherwise the fitter decomposes the CE input into the D (or T) logic expression unless REG=CE is applied. CE product terms are not available in XC9500 devices (REG=CE is ignored). In XC9500XL and XC9500XV devices, the CE product term is available only for registers that do not use both the CLR and PRE inputs.
- TFF indicates that the register is to be implemented as a T-type flip-flop in the CPLD macrocell. If applied to a D-flip-flop primitive, the D-input expression is transformed to T-input form and implemented with a T-flip-flop. Automatic transformation between D and T flip-flops is normally performed by the CPLD fitter.

The following statement specifies that the CE pin input be implemented using the clock enable product term of the XC9500XL or XC9500XV macrocell.

```
INST "Q1" REG=CE;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" REG={CE|TFF};
END;
```

Relative Location (RLOC)

RLOC Architecture Support

The RLOC constraint applies to FPGA devices only.

RLOC Applicable Elements

To see which design elements can be used with which device families, see the Xilinx *Libraries Guide*s. For more information, see the device data sheet.

1. Registers
2. ROM
3. RAMS, RAMD
4. BUFT
   Can be used only if the associated RPM has an RLOC_ORIGIN that causes the RLOC values in the RPM to be changed to LOC values.
5. LUTs, MUXF5, MUXF6, MUXCY, XORCY, MULT_AND, SRL16, SRL16E, MUXF7
   Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X only
6. MUXF8
   Spartan™-3, Spartan-3A, Spartan-3E, Virtex™-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 only
7. Block RAMs
8. Multipliers
9. DSP48

RLOC Description

Relative location (RLOC) is a basic mapping and placement constraint. It is also a synthesis constraint. RLOC constraints group logic elements into discrete sets and allow you to define the location of any element within the set relative to other elements in the set, regardless of eventual placement in the overall design.

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, the RLOC constraint must include the extension that defines in which of the two slices of a CLB the element is placed (.S0, .S1).

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the RLOC constraint is specified using the slice-based XY coordinate system.

Benefits and Limitations of RLOC Constraints

RLOC constraints allow you to place logic blocks relative to each other to increase speed and use die resources efficiently. They provide an order and structure to related design elements without requiring you to specify their absolute placement on the FPGA die. They allow you to replace any existing hard macro with an equivalent that can be directly simulated.

In the Unified Libraries, you can use RLOC constraints with BUFT- and CLB-related primitives, that is, FMAP. You can also use them on non-primitive macro symbols. There are some restrictions on the use of RLOC constraints on BUFT symbols. For more

information, see "Set Modifiers" in this chapter. You cannot use RLOC constraints with decoders or clocks. LOC constraints, on the other hand, can be used on all primitives: BUFTs, CLBs, IOBs, decoders, and clocks.

The following symbols (primitives) accept RLOCs.

- Registers
- ROM
- RAMS, RAMD
- BUFT
- LUTs, MUXCY, XORCY, MULT_AND, SRL16, SRL16E
- DSP48
- MULT18x18

Guidelines for Specifying Relative Locations

There are two different coordinate designations:

- Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices use the CLB-based coordinate system.
- Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices use the slice-based coordinate system.

CLB-based Row/Column/Slice Designations

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, the general syntax for assigning elements to relative locations is

RLOC=[*element*]**R***m***C***n*[**.***extension*]

where

- *m* and *n* are relative row numbers and column numbers, respectively
- *extension* uses the LOC extension syntax and can take all the values that are available with the absolute LOC syntax: S0, S1, 0, 1, 2, and 3 as appropriate for the architecture and element the RLOC is attached to.

The extension is required for Virtex, Virtex-E, Spartan-II, and Spartan-IIE designs to specify the spatial relationship of the objects in the RPM (.S0, .S1).

The row and column numbers can be any positive or negative integer including zero. Absolute die locations, in contrast, cannot have zero as a row or column number. Because row and column numbers in RLOC constraints define only the order and relationship between design elements and not their absolute die locations, their numbering can include zero or negative numbers. Even though you can use any integer in numbering rows and columns for RLOC constraints, it is recommended that you use small integers for clarity and ease of use.

It is not the absolute values of the row and column numbers that is important in RLOC specifications but their relative values or differences. For example, if design element A has an RLOC=R3C4 constraint and design element B has an RLOC=R6C7 constraint, the absolute values of the row numbers (3 and 6) are not important in themselves. However, the difference between them is important; in this case, 3 (6 -3) specifies that the location of design element B is three rows down from the location of design element A.

To capture this information, a normalization process is used and column-wise the design element B is 3 (7-4) columns on the right of element A. In the example just given,

normalization would reduce the RLOC on design element A to R0C0, and the RLOC on design element B to R3C3.

In CLB-based programs, row/column rows are numbered in increasing order from top to bottom, and columns are numbered in increasing order from left to right. RLOC constraints follow this numbering convention.

Figure 70-1 demonstrates the use of RLOC constraints. Figure 70-1 applies only to Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices.

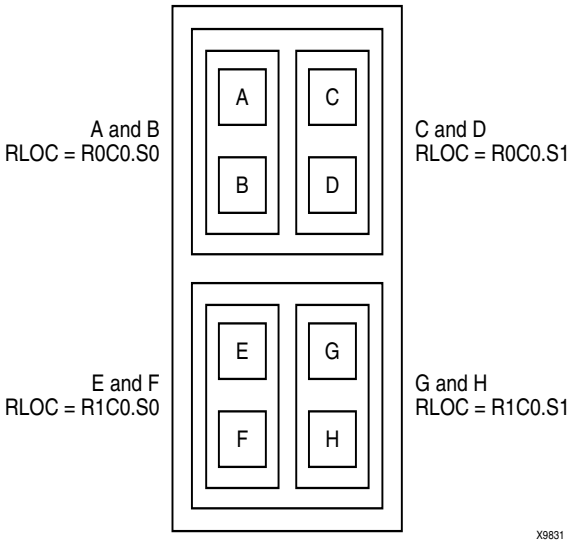

Figure 70-1: **RLOC Specifications for Eight Flip-Flop Primitives**

Slice-Based XY Coordinate Designations

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the general syntax for assigning elements to relative locations is

RLOC=X*m***Y***n*

where

- *m* and *n* are the relative X-axis (left/right) value and the relative Y-axis (up/down) value, respectively
- the X and Y numbers can be any positive or negative integer including zero

Because the X and Y numbers in RLOC constraints define only the order and relationship between design elements and not their absolute die locations, their numbering can include negative numbers. Even though you can use any integer for RLOC constraints, it is recommended that you use small integers for clarity and ease of use.

It is not the absolute values of the X and Y numbers that is important in RLOC specifications but their relative values or differences. For example, if design element A has an RLOC=X3Y4 constraint and design element B has an RLOC=X6Y7 constraint, the absolute values of the X numbers (3 and 6) are not important in themselves. However, the difference between them is important; in this case, 3 (6 -3) specifies that the location of design element B is three slices away from the location of design element A.

To capture this information, a normalization process is used and y coordinate-wise, element B is 3 (7-4) slices above element A. In the example just given, normalization would reduce the RLOC on design element A to X0Y0, and the RLOC on design element B to X3Y3.

In Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, slices are numbered on an XY grid beginning in the lower left corner of the chip. X ascends in value horizontally to the right. Y ascends in value vertically up. RLOC constraints follow the cartesian-based convention.

Figure 70-2 demonstrates the use of RLOC constraints. In (a) in Figure 70-2 four flip-flop primitives named A, B, C, and D are assigned RLOC constraints as shown. These RLOC constraints require each flip-flop to be placed in a different slice with the slices stacked in the order shown: A below B, C, and D.

If you want to place more than one of these flip-flop primitives per slice, you can specify the RLOCs as shown in (b) in Figure 70-2. The arrangement in the figure requires that A and B be placed in a single slice and that C and D be placed in another slice immediately to the right of the AB slice. Figure 70-2 applies only to Virtex-II, Virtex-II Pro, and Virtex-II Pro X, Virtex-4, Virtex-5, Spartan-3, Spartan-3A, and Spartan-3E devices.

Figure 70-2: **Different RLOC Specifications for Four Flip-Flop Primitives**

RLOC Sets

RLOC constraints give order and structure to related design elements. This section describes RLOC sets, which are groups of related design elements to which RLOC constraints have been applied. For example, the eight flip-flops in Figure 70-1 and the four flip-flops in Figure 70-2 are related by RLOC constraints and form a set. Elements in a set are related by RLOC constraints to other elements in the same set. Each member of a set must have an RLOC constraint, which relates it to other elements in the same set. You can create multiple sets, but a design element can belong to one set only.

Sets can be defined explicitly through the use of a set parameter or implicitly through the structure of the design hierarchy.

Four distinct types of rules are associated with each set.

- Definition rules define the requirements for membership in a set.
- Linkage rules specify how elements can be linked to other elements to form a single set.
- Modification rules dictate how to specify parameters that modify RLOC values of all the members of the set.
- Naming rules specify the nomenclature of sets.

These rules are discussed in the sections that follow.

The following sections discuss three different set constraints: U_SET, H_SET, and HU_SET. Elements must be tagged with both the RLOC constraint and one of these set constraints to belong to a set.

U_SET

U_SET constraints enable you to group into a single set design elements with attached RLOC constraints that are distributed throughout the design hierarchy. The letter U in the name U_SET indicates that the set is user-defined.

U_SET constraints allow you to group elements, even though they are not directly related by the design hierarchy. By attaching a U_SET constraint to design elements, you can explicitly define the members of a set.

The design elements tagged with a U_SET constraint can exist anywhere in the design hierarchy; they can be primitive or non-primitive symbols. When attached to non-primitive symbols, the U_SET constraint propagates to all the primitive symbols with RLOC constraints that are below it in the hierarchy.

The syntax of the U_SET constraint is:

```
U_SET=set_name
```

where

- *set_name* is the user-specified identifier of the set

All design elements with RLOC constraints tagged with the same U_SET constraint name belong to the same set. Names therefore must be unique among all the sets in the design.

H_SET

In contrast to the U_SET constraint, which you explicitly define by tagging design elements, the H_SET (hierarchy set) is defined implicitly through the design hierarchy. The combination of the design hierarchy and the presence of RLOC constraints on elements defines a hierarchical set, or H_SET set.

You are *not* able to use an H_SET constraint to tag the design elements to indicate their set membership. The set is defined automatically by the design hierarchy.

All design elements with RLOC constraints at a single node of the design hierarchy are considered to be in the same H_SET set unless they are tagged with another type of set constraint such as RLOC_ORIGIN or RLOC_RANGE. If you explicitly tag any element with an RLOC_ORIGIN, RLOC_RANGE, U_SET, or HU_SET constraint, it is removed from an H_SET set.

Most designs contain only H_SET constraints, since they are the underlying mechanism for relationally placed macros. The RLOC_ORIGIN or RLOC_RANGE constraints are discussed further in "Set Modifiers"in this chapter.

NGDBuild recognizes the implicit H_SET set, derives its name, or identifier, attaches the H_SET constraint to the correct members of the set, and writes them to the output file.

HU_SET

The HU_SET constraint is a variation of the implicit H_SET (hierarchy set). Like H_SET, HU_SET is defined by the design hierarchy. However, you can use the HU_SET constraint to assign a user-defined name to the HU_SET.

The syntax of the HU_SET constraint is:

```
HU_SET=set_name
```

where

- *set_name* is the identifier of the set. It must be unique among all the sets in the design

This user-defined name is the base name of the HU_SET set. Like the H_SET set, in which the base name of "h_set" is prefixed by the hierarchical name of the lowest common ancestor of the set elements, the user-defined base name of an HU_SET set is prefixed by the hierarchical name of the lowest common ancestor of the set elements.

You must define the base names to ensure unique hierarchically qualified names for the sets before the mapper resolves the design and attaches the hierarchical names as prefixes.

The HU_SET constraint defines the start of a new set. All design elements at the same node that have the same user-defined value for the HU_SET constraint are members of the same HU_SET set. Along with the HU_SET constraint, elements can also have an RLOC constraint.

The presence of an RLOC constraint in an H_SET constraint links the element to all elements tagged with RLOCs above and below in the hierarchy. However, in the case of an HU_SET constraint, the presence of an RLOC constraint along with the HU_SET constraint on a design element does not automatically link the element to other elements with RLOC constraints at the same hierarchy level or above.

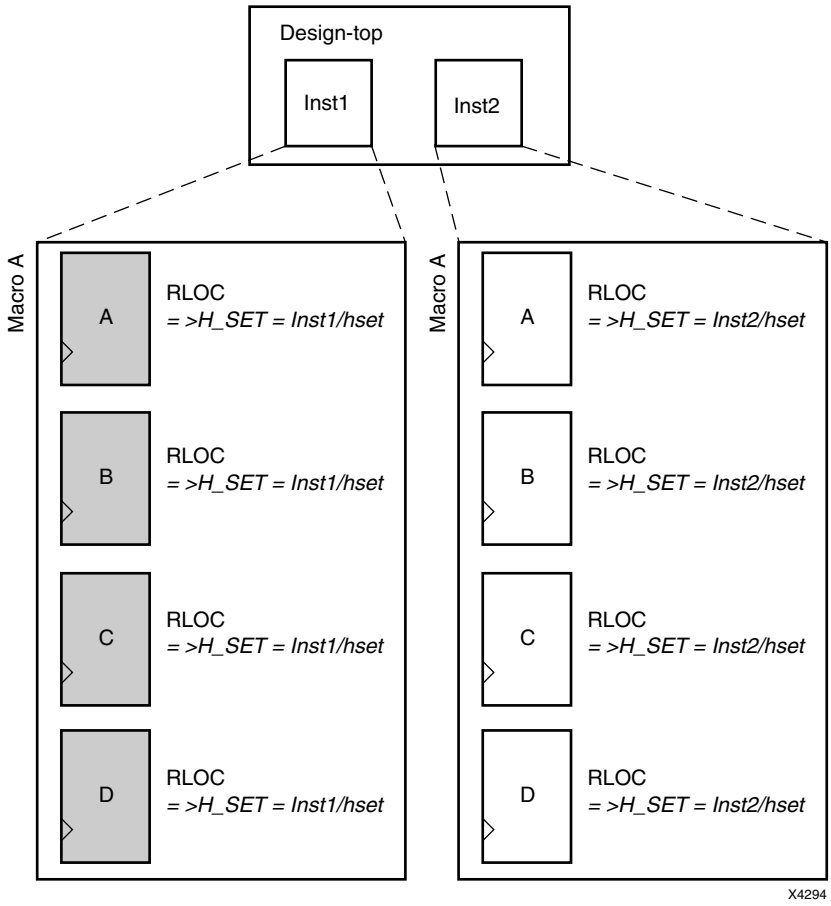

Figure 70-3: **Macro A Instantiated Twice**

Note: In Figure 70-3 and the other related figures shown in the subsequent sections, the italicized text prefixed by => is added by NGDBuild during the design flattening process. You add all other text.

Figure 70-3 demonstrates a typical use of the implicit H_SET (hierarchy set). The figure shows only the first "RLOC" portion of the constraint. In a real design, the RLOC constraint must be specified completely with RLOC=R*m*C*n* or, for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 RLOC=X*m*Y*n*. In this example, macro A is originally designed with RLOC constraints on four flip-flops: A, B, C, and D. The macro is then instantiated twice in the design: Inst1 and Inst2.

When the design is flattened, two different H_SET sets are recognized because two distinct levels of hierarchy contain elements with RLOC constraints. NGDBuild creates and attaches the appropriate H_SET constraint to the set members: H_SET=Inst1/h_set for the macro instantiated in Inst1, and H_SET=Inst2/h_set for the macro instantiated in Inst2. The design implementation programs place each of the two sets individually as a unit with relative ordering within each set specified by the RLOC constraints. However, the two sets are regarded to be completely independent of each other.

The name of the H_SET set is derived from the symbol or node in the hierarchy that includes all the RLOC elements. In Figure 70-3, Inst1 is the node (instantiating macro) that includes the four flip-flop elements with RLOCs shown on the left of the figure. Therefore, the name of this H_SET set is the hierarchically qualified name of "Inst1" followed by "h_set."

The Inst1 symbol is considered the "start" of the H_SET, which gives a convenient handle to the entire H_SET and attaches constraints that modify the entire H_SET. Constraints that modify sets are discussed in the "Save Net Flag (SAVE NET FLAG)" constraint.

Figure 70-3, page 259 demonstrates the simplest use of a set that is defined and confined to a single level of hierarchy. Through linkage and modification, you can also create an H_SET set that is linked through two or more levels of hierarchy.

Linkage allows you to link elements through the hierarchy into a single set. On the other hand, modification allows you to modify RLOC values of the members of a set through the hierarchy.

RLOC Set Summary

The following table summarizes the RLOC set types and the constraints that identify members of these sets.

Table 70-1: **Summary of Set Types**

Type	Definition	Naming	Linkage	Modification
U_SET= name	All elements with the same user-tagged U_SET constraint value are members of the same U_SET set.	The name of the set is the same as the user-defined name without any hierarchical qualification.	U_SET links elements to all other elements with the same value for the U_SET constraint.	U_SET is modified by applying RLOC_ORIGIN or RLOC_RANGE constraints on, at most, one of the U_SET constraint-tagged elements.
HU_SET= name	All elements with the same hierarchically qualified name are members of the same set.	The lowest common ancestor of the members is prefixed to the user-defined name to obtain the name of the set.	HU_SET links to other elements at the same node with the same HU_SET constraint value. It links to elements with RLOC constraints below.	The start of the set is made up of the elements on the same node that are tagged with the same HU_SET constraint value. An RLOC_ORIGIN or an RLOC_RANGE can be applied to, at most, one of these start elements of an HU_SET set.

RLOC Propagation Rules

RLOC is a design element constraint and any attachment to a net is illegal. When attached to a design element, RLOC propagates to all applicable elements in the hierarchy within the design element.

RLOC Syntax

For Architectures Using CLB-based Row/Column/Slice Specifications

This section applies to Virtex, and Virtex-E, Spartan-II, and Spartan-IIE devices only.

RLOC=R*m***C***n*.*extension*

where

- *m* and *n* are integers (positive, negative, or zero) representing relative row numbers and column numbers, respectively

- *extension* uses the LOC extension syntax as appropriate. It can take all the values that are available with the current absolute LOC syntax

For Virtex, Virtex-E, Spartan-II and Spartan-IIE, *extension* is required to define the spatial relationships (.S0 is the right-most slice; .S1 is the left-most slice) of the objects in the RPM.

The RLOC value cannot specify a range or a list of several locations; it must specify a single location. For more information, see "RLOC Description" in this chapter.

For Architectures Using a Slice-Based XY Coordinate System

This section applies to Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only.

RLOC=X*m***Y***n*

where

- *m* and *n* are integers (positive, negative, or zero) representing relative X and Y coordinates, respectively

Set Linkage

The example Figure 70-4, page 262 explains and illustrates the process of linking together elements through the design hierarchy. Again, the complete RLOC specification, RLOC=R*m*C*n* or RLOC=X*m*X*n*, is required for a real design.

Note: In this and other illustrations in this section, the sets are shaded differently to distinguish one set from another.

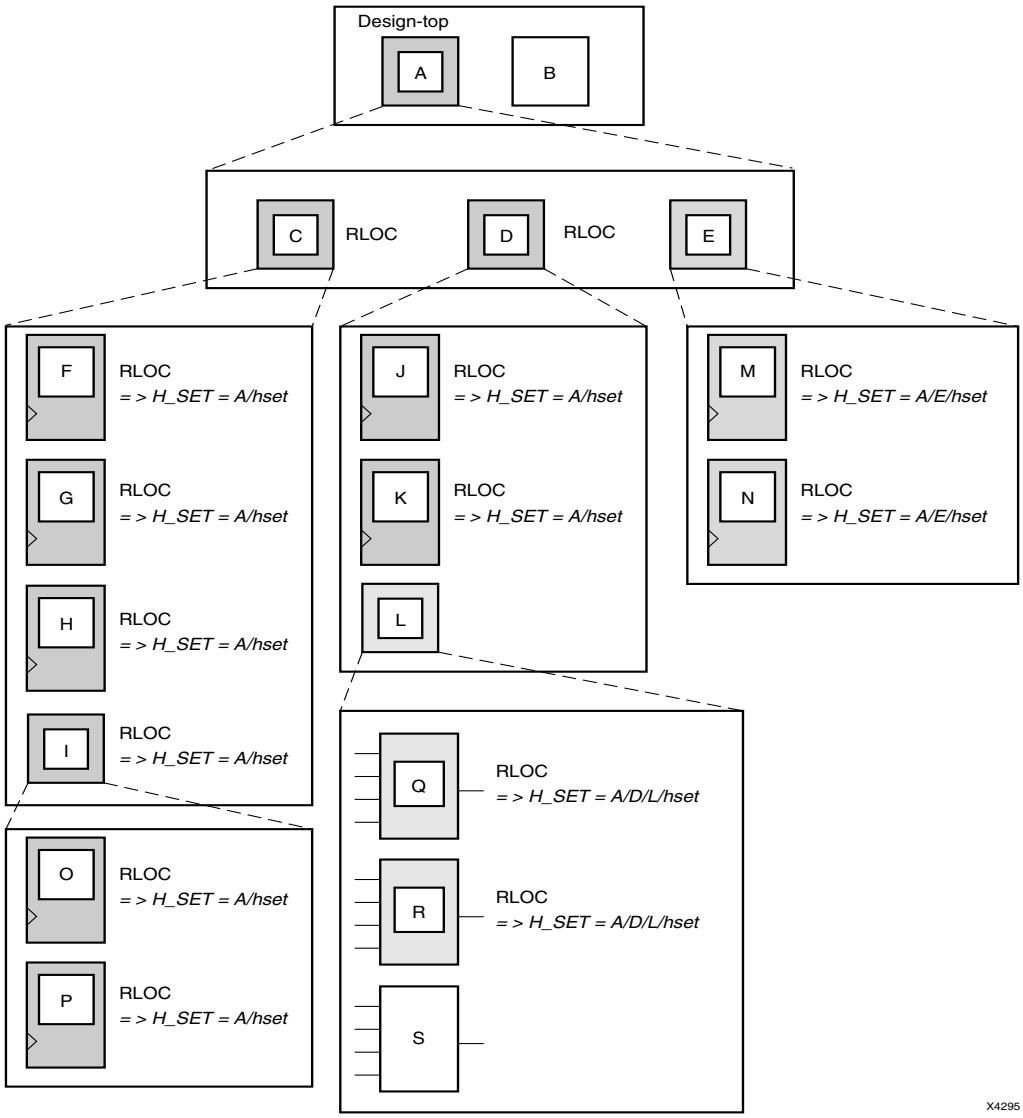

Figure 70-4: **Three H_SET Sets**

As noted previously, all design elements with RLOC constraints at a single node of the design hierarchy are considered to be in the same H_SET set unless they are assigned another type of set constraint, an RLOC_ORIGIN constraint, or an RLOC_RANGE constraint. In Figure 70-4, page 262, RLOC constraints have been added on primitives and non-primitives C, D, F, G, H, I, J, K, M, N, O, P, Q, and R. No RLOC constraints were placed on B, E, L, or S. Macros C and D have an RLOC constraint at node A, so all the primitives below C and D that have RLOCs are members of a single H_SET set.

The name of this H_SET set is "A/h_set" because it is at node A that the set starts. The start of an H_SET set is the lowest common ancestor of all the RLOC-tagged constraints that constitute the elements of that H_SET set.

Because element E does not have an RLOC constraint, it is not linked to the A/h_set set. The RLOC-tagged elements M and N, which lie below element E, are therefore in their own H_SET set. The start of that H_SET set is A/E, giving it the name "A/E/h_set."

Similarly, the Q and R primitives are in their own H_SET set because they are not linked through element L to any other design elements. The lowest common ancestor for their H_SET set is L, which gives it the name "A/D/L/h_set." After the flattening, NGDBuild attaches H_SET=A/h_set to the F, G, H, O, P, J, and K primitives; H_SET=A/D/L/h_set to the Q and R primitives; and H_SET=A/E/h_set to the M and N primitives.

Consider a situation in which a set is created at the top of the design. In Figure 70-4, page 262, there would be no lowest common ancestor if macro A also had an RLOC constraint, since A is at the top of the design and has no ancestor. In this case, the base name "h_set" would have no hierarchically qualified prefix, and the name of the H_SET set would simply be "h_set."

Set Modification

The RLOC constraint assigns a primitive an RLOC value (the row and column numbers with the optional extensions), specifies its membership in a set, and links together elements at different levels of the hierarchy. In Figure 70-4, page 262, the RLOC constraint on macros C and D links together all the objects with RLOC constraints below them. An RLOC constraint is also used to modify the RLOC values of constraints below it in the hierarchy. In other words, RLOC values of elements affect the RLOC values of all other member elements of the same H_SET set that lie below the given element in the design hierarchy.

The Effect of the Hierarchy on Set Modification

The following sections describe the effect of the hierarchy on set modification for the CLB-based Row/Column/Slice designations and for the slice-based XY coordinate designations (Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices).

CLB-Based Row/Column/Slice Designations

When the design is flattened, the row and column numbers of an RLOC constraint on an element are added to the row and column numbers of the RLOC constraints of the set members below it in the hierarchy. This feature gives you the ability to modify existing RLOC values in submodules and macros without changing the previously assigned RLOC values on the primitive symbols.

This modification process also applies to the optional extension field. However, when using extensions for modifications, you must ensure that inconsistent extensions are not attached to the RLOC value of a design element that may conflict with RLOC extensions placed on underlying elements.

For example, if an element has an RLOC constraint with the S0 extension, all the underlying elements with RLOC constraints must either have the same extension, in this case S0, or no extension at all; any underlying element with an RLOC constraint and an extension different from S0, such as S1, is flagged as an error.

After resolving all the RLOC constraints, extensions that are not valid on primitives are removed from those primitives. For example, if NGDBuild generates an S0 extension to be applied on a primitive after propagating the RLOC constraints, it applies the extension if and only if the primitive is a flip-flop. If the primitive is an element other than a flip-flop, the extension is ignored. Only the extension is ignored in this case, not the entire RLOC constraint.

Figure 70-5, page 265 illustrates the process of adding RLOC values down the hierarchy. The row and column values between the parentheses show the addition function performed by the mapper. The italicized text prefixed by => is added by MAP during the design resolution process and replaces the original RLOC constraint that you added. For S*n*, the value *n* is either a 1 or a 0.

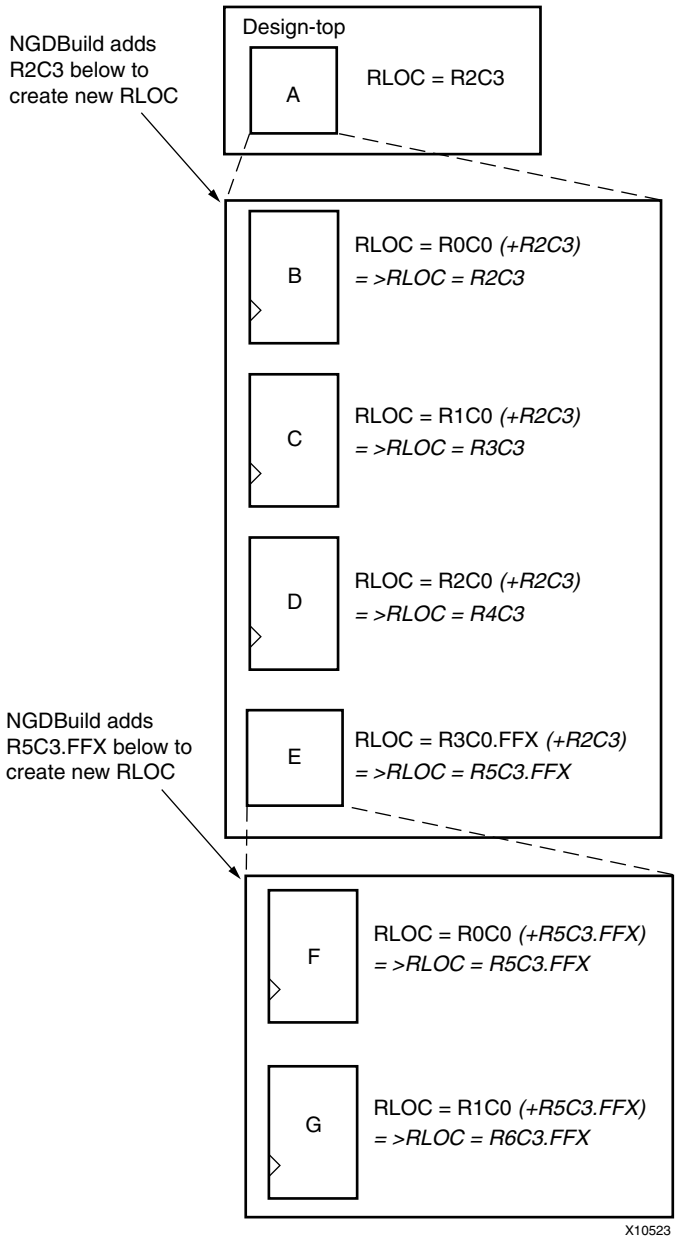

Figure 70-5: **Adding RLOC Values Down the Hierarchy (CLB-based Row/Column/Slice)**

The ability to modify RLOC values down the hierarchy is particularly valuable when instantiating the same macro more than once. Typically, macros are designed with RLOC constraints that are modified when the macro is instantiated. Figure 70-6, page 266 is a variation of the sample design in Figure 70-3, page 259. The RLOC constraint on Inst1 and Inst2 now link all the objects in one H_SET set.

Because the RLOC=R0C0 modifier on the Inst1 macro does not affect the objects below it, the mapper adds only the H_SET tag to the objects and leaves the RLOC values as they are.

However, the RLOC=R0C1 modifier on the Inst2 macro causes MAP to change the RLOC values on objects below it, as well as to add the H_SET tag, as shown in the italicized text.

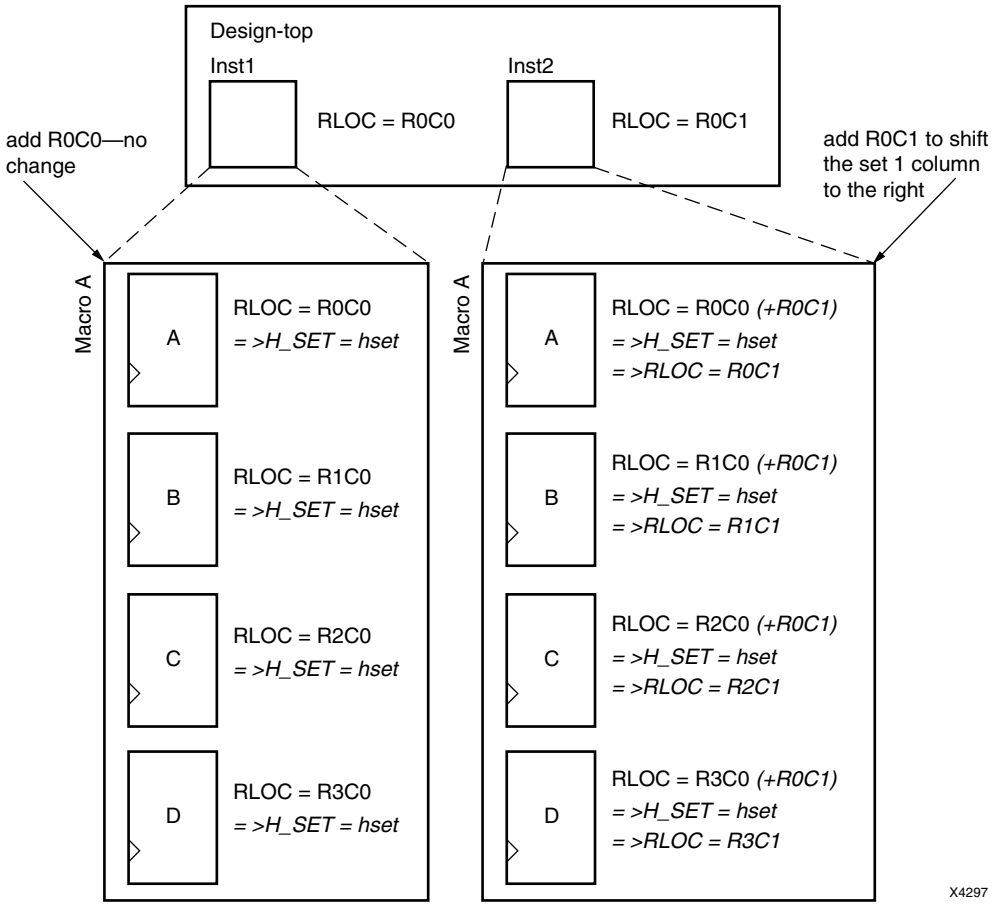

Figure 70-6: **Modifying RLOC Values of Same Macro and Linking Together as One Set (CLB-based Row/Column/Slice)**

Slice-Based XY Designations

When the design is flattened, the XY values of an RLOC constraint on an element are added to the XY values of the RLOC constraints of the set members below it in the hierarchy. This feature gives you the ability to modify existing RLOC values in submodules and macros without changing the previously assigned RLOC values on the primitive symbols.

Figure 70-7, page 267 illustrates the process of adding RLOC values down the hierarchy. The row and column values between the parentheses show the addition function performed by the mapper. The italicized text prefixed by => is added by MAP during the design resolution process and replaces the original RLOC constraint that you added.

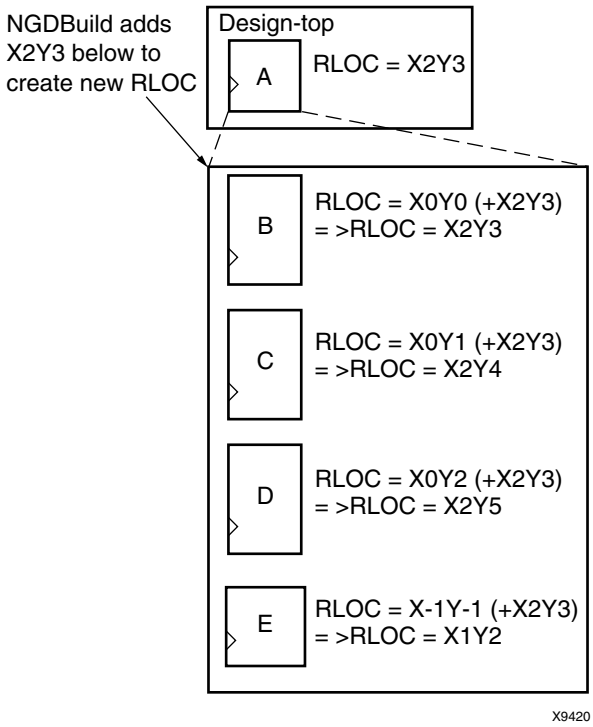

Figure 70-7: **Adding RLOC Values Down the Hierarchy Example (Slice-based XY Designations)**

The ability to modify RLOC values down the hierarchy is particularly valuable when instantiating the same macro more than once. Typically, macros are designed with RLOC constraints that are modified when the macro is instantiated. Figure 70-8, page 268 is a variation of the sample design in Figure 70-7, page 267. The RLOC constraint on Inst1 and Inst2 now link all the objects in one H_SET set.

Because the RLOC=X0Y0 modifier on the Inst1 macro does not affect the objects below it, the mapper adds only the H_SET tag to the objects and leaves the RLOC values as they are. However, the RLOC=X1Y0 modifier on the Inst2 macro causes MAP to change the RLOC values on objects below it, as well as to add the H_SET tag, as shown in the italicized text.

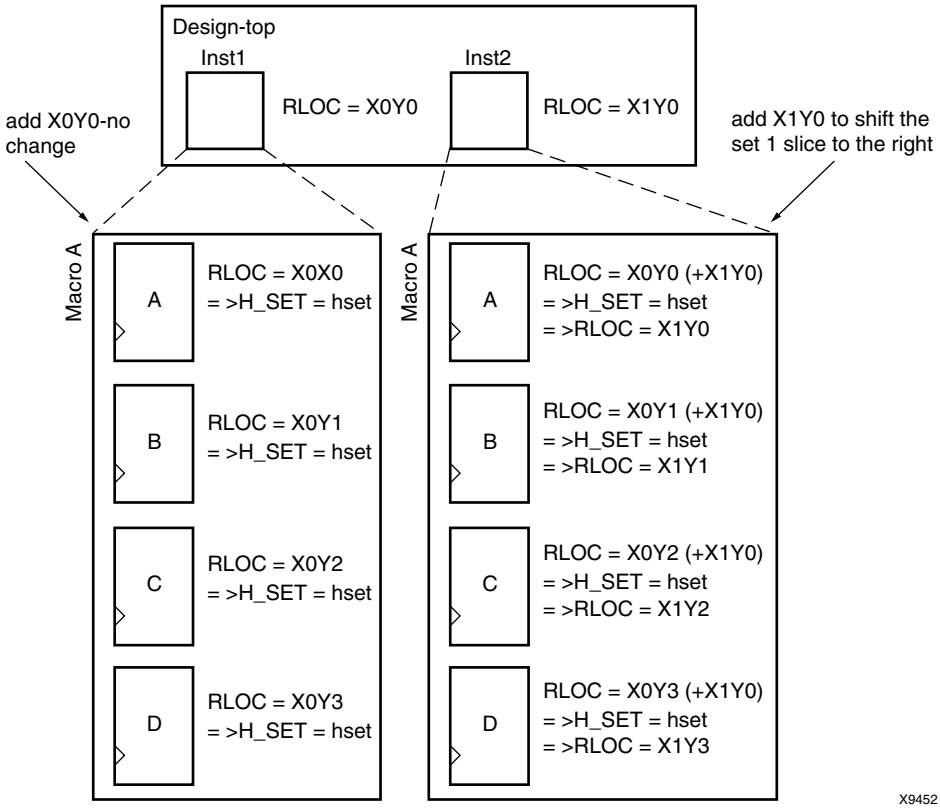

Figure 70-8: **Modifying RLOC Values of Same Macro and Linking Together as One Set (Slice-based XY Designations)**

Separating Elements from H_SET Sets

The HU_SET constraint is a variation of the implicit H_SET (hierarchy set). The HU_SET constraint defines the start of a new set. Like H_SET, HU_SET is defined by the design hierarchy. However, you can use the HU_SET constraint to assign a user-defined name to the HU_SET.

Figure 70-9, page 269 demonstrates how HU_SET constraints designate elements as set members, break links between elements tagged with RLOC constraints in the hierarchy to separate them from H_SET sets, and generate names as identifiers of these sets.

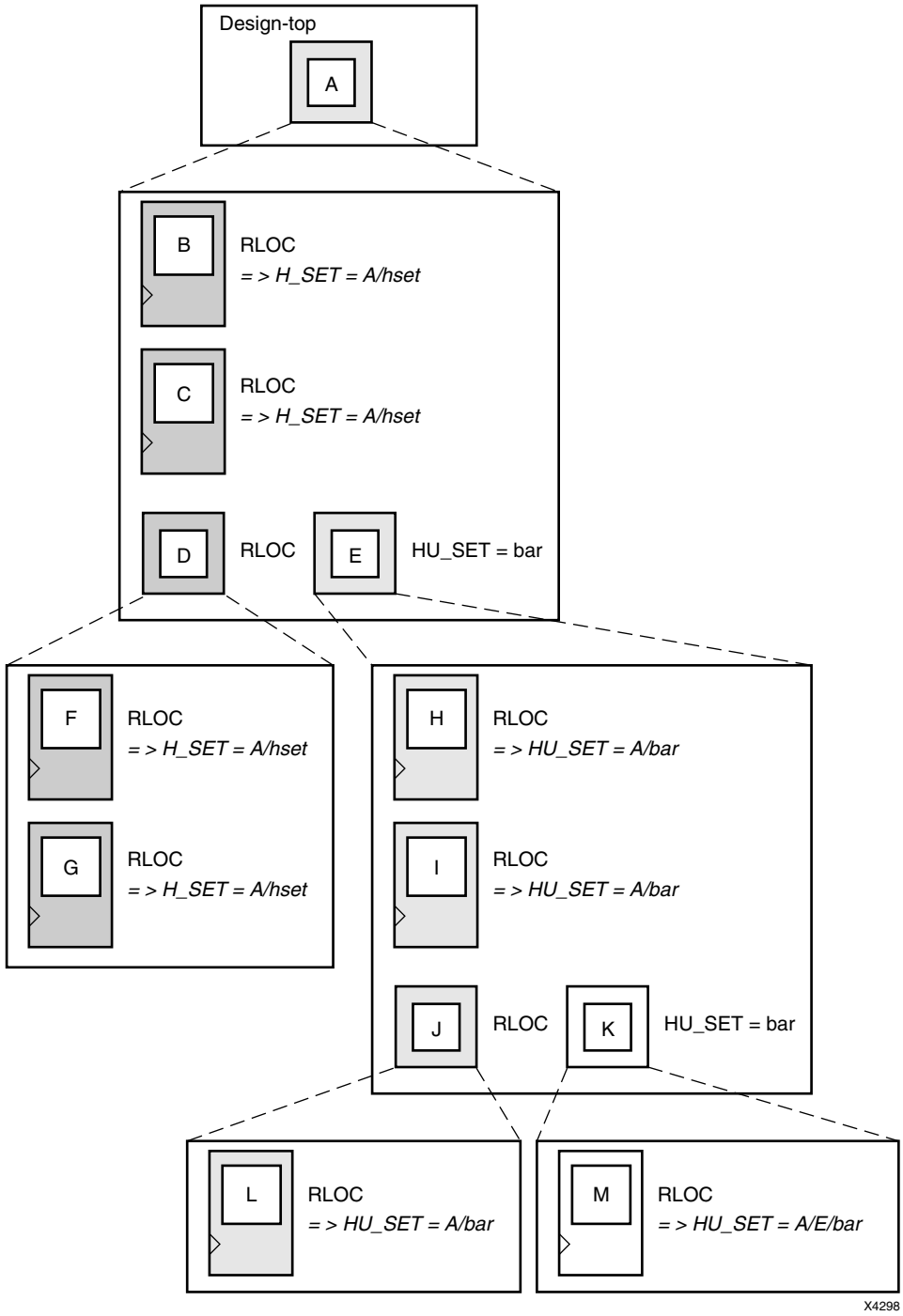

Figure 70-9: **HU_SET Constraint Linking and Separating Elements from H_SET Sets**

The user-defined HU_SET constraint on E separates its underlying design elements, namely H, I, J, K, L, and M from the implicit H_SET=A/h_set that contains primitive

members B, C, F, and G. The HU_SET set that is defined at E includes H, I, and L (through the element J).

The mapper hierarchically qualifies the name value “bar” on element E to be A/bar, since A is the lowest common ancestor for all the elements of the HU_SET set, and attaches it to the set member primitives H, I, and L. An HU_SET constraint on K starts another set that includes M, which receives the HU_SET=A/E/bar constraint after processing by the mapper.

In Figure 70-9, page 269, the same name field is used for the two HU_SET constraints, but because they are attached to symbols at different levels of the hierarchy, they define two different sets.

Figure 70-10, page 270 shows how HU_SET constraints link elements in the same node together by naming them with the same identifier. Because of the same name, “bar,” on two elements, D and E, the elements tagged with RLOC constraints below D and E become part of the same HU_SET.

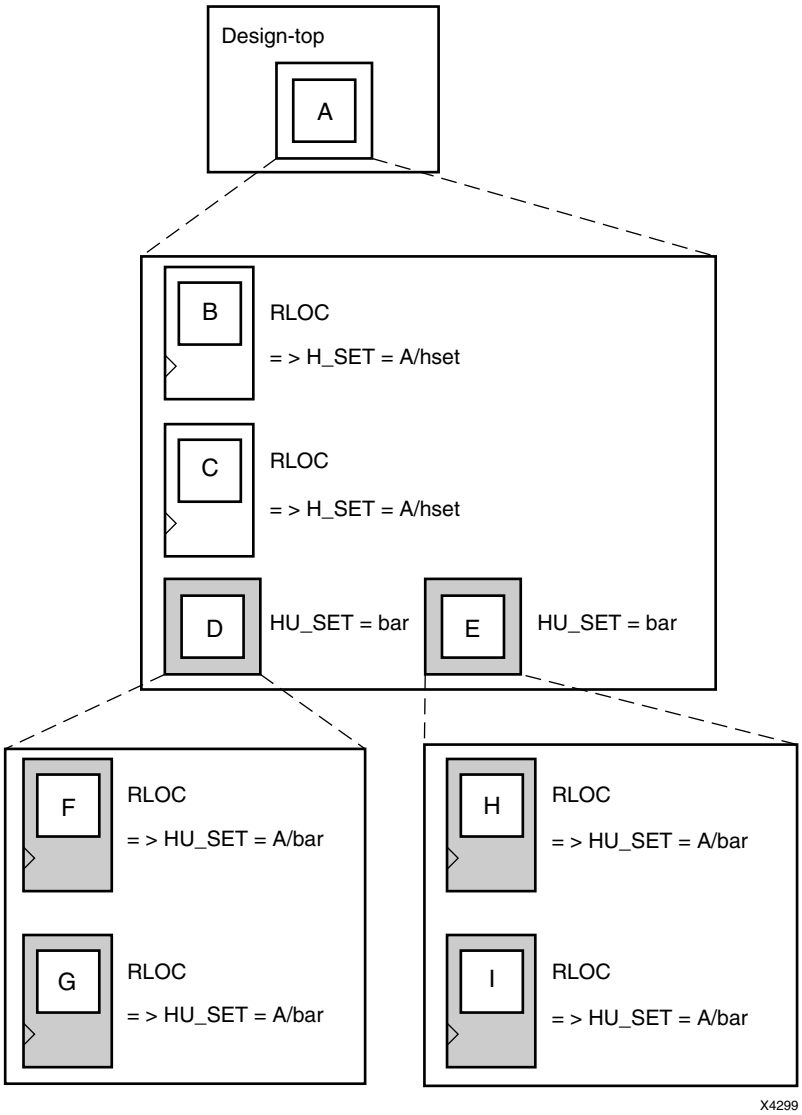

Figure 70-10: **Linking Two HU_SET Sets**

Set Modifiers

A modifier, as its name suggests, modifies the RLOC constraints associated with design elements. Since it modifies the RLOC constraints of all the members of a set, it must be applied in a way that propagates it to all the members of the set easily and intuitively. For this reason, the RLOC modifiers of a set are placed at the start of that set. The following set modifiers apply to RLOC constraints.

- RLOC

  The RLOC constraint associated with a design element modifies the values of other RLOC constraints below the element in the hierarchy of the set. Regardless of the set type, RLOC values (row, column, extension or XY values) on an element always propagate down the hierarchy and are added at lower levels of the hierarchy to RLOC constraints on elements in the same set.

- "Relative Location Origin (RLOC_ORIGIN)"
- "Relative Location Range (RLOC_RANGE)"

Using RLOCs with Xilinx Macros

Xilinx-supplied flip-flop macros include an RLOC=R0C0 constraint on the underlying primitive, which allows you to attach an RLOC to the macro symbol. (For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the macros include an RLOC=X0Y0 constraint.) This symbol links the underlying primitive to the set that contains the macro symbol.

Simply attach an appropriate RLOC constraint to the instantiation of the actual Xilinx flip-flop macro. The mapper adds the RLOC value that you specified to the underlying primitive so that it has the desired value.

For example, in Figure 70-11, page 272, the RLOC = R1C1 constraint is attached to the instantiation (Inst1) of an example macro. It is added to the R0C0 value of the RLOC constraint on the flip-flop within the macro to obtain the new RLOC values. This functions the same for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 macros except that the RLOC constraint uses XY designations.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, if the RLOC=X1Y1 constraint is attached to Inst1 of a macro, the X0Y0 value of the RLOC constraint on the flip-flop within the macro would be used to obtain the new RLOC values.

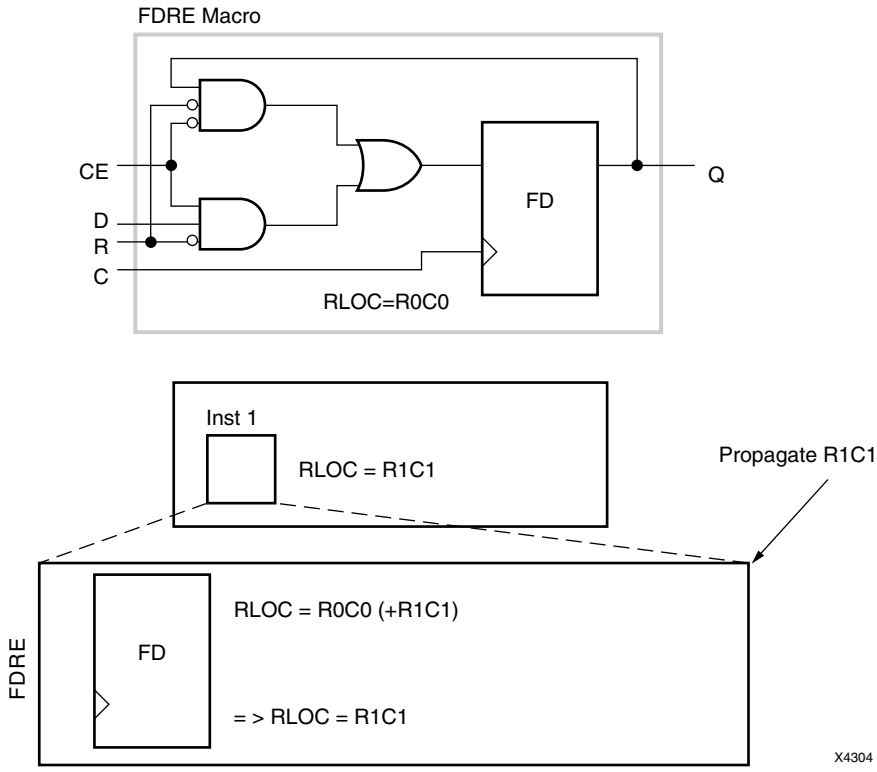

Figure 70-11: **Typical Use of a Xilinx Macro**

If you do not put an RLOC constraint on the flip-flop macro symbol, the underlying primitive symbol is the lone member of a set. The mapper removes RLOC constraints from a primitive that is the only member of a set or from a macro that has no RLOC objects below it.

LOC and RLOC Propagation through Design Flattening

NGDBuild continues to propagate LOC constraints down the design hierarchy. It adds this constraint to appropriate objects that are not members of a set. While RLOC constraint propagation is limited to sets, the LOC constraint is applied from its start point all the way down the hierarchy.

When the design is flattened, the row and column numbers of an RLOC constraint on an element are added to the row and column numbers of the RLOC constraints of the set members below it in the hierarchy. This feature gives you the ability to modify existing RLOC values in submodules and macros without changing the previously assigned RLOC values on the primitive symbols.

Specifying RLOC constraints to describe the spatial relationship of the set members to themselves allows the members of the set to float anywhere on the die as a unit. You can, however, fix the exact die location of the set members. The RLOC_ORIGIN constraint allows you to change the RLOC values into absolute LOC constraints that respect the structure of the set.

The design resolution program, NGDBuild, translates the RLOC_ORIGIN constraint into LOC constraints. The row and column values of the RLOC_ORIGIN are added

individually to the members of the set after all RLOC modifications have been made to their row and column values by addition through the hierarchy. The final values are then turned into LOC constraints on individual primitives.

Fixing Members of a Set at Exact Die Locations

As noted in the previous section, you can fix the members of a set at exact die locations with the RLOC_ORIGIN constraint. You must use the RLOC_ORIGIN constraint with sets that include BUFT symbols. However, for sets that do not include BUFT symbols, you can limit the members of a set to a certain range on the die.

In this case, the set could "float" as a unit within the range until a final placement. Since every member of the set must fit within the range, it is important that you specify a range that defines an area large enough to respect the spatial structure of the set.

CLB-Based Row/Column/Slice Designations

The syntax of this constraint is:

RLOC_RANGE=R*m1***C***n1***:R***m2***C***n2*

where

- the relative row numbers (*m1*, *m2*) and column numbers (*n1*, *n2*) can be:
  - non-zero positive numbers
  - the wildcard (*) character

This syntax allows for three kinds of range specifications as follows.

- R*r1*C*c1*:R*r2*C*c2*
  A rectangular region enclosed by rows *r1*, *r2*, and columns *c1*, *c2*
- R*C*c1*:R*C*c2*
  A region enclosed by the columns *c1* and *c2* (any row number)
- R*r1*C*:R*r2*C*
  A region enclosed by the rows *r1* and *r2* (any column number)

For the second and third kinds of specifications with wildcards, applying the wildcard character (*) differently on either side of the separator colon creates an error. For example, specifying R*C1:R2C* is an error since the wildcard asterisk is applied to rows on one side and to columns on the other side of the separator colon.

Slice-Based XY Designations

The syntax of this constraint is:

RLOC_RANGE=X*m1***Y***n1***:X***m2***Y***n2*

where

- the relative X values (*m1*, *m2*) and Y values (*n1*, *n2*) can be:
  - non-zero positive numbers
  - the wildcard (*) character

This syntax allows for three kinds of range specifications:

- X*m1*Y*n1*:X*m2*Y*n2*
  A rectangular region bounded by the corners X*m1*Y*n1* and X*m2*Y*n2*
- X*Y*n1*:X*Y*m2*
  The region on the Y-axis between *n1* and *n2* (any X value)

- *X**m1**Y*:*X**m2**Y**
  A region on the X-axis between *m1* and *m2* (any Y value)

For the second and third kinds of specifications with wildcards, applying the wildcard character (*) differently on either side of the separator colon creates an error. For example, specifying X*Y1:X2Y* is an error since the wildcard asterisk is applied to the X value on one side and to the Y value on the other side of the separator colon.

Specifying a Range

To specify a range, use the following syntax, which is equivalent to placing an RLOC_RANGE constraint on the schematic.

- For CLB-based Row/Column/Slice Designations

  set_name **RLOC_RANGE=R***m1***C***n1***:R***m2***C***n2*

  The range identifies a rectangular area. You can substitute a wildcard (*) character for either the row number or the column number of both corners of the range.

- For Slice-based XY Designations

  set_name **RLOC_RANGE=X***m1***Y***n1***:X***m2***Y***n2*

  The range identifies a rectangular area. You can substitute a wildcard (*) character for either the X value or the Y value of both corners of the range.

The bounding rectangle applies to all elements in a relationally placed macro, not just to the origin of the set.

The values of the RLOC_RANGE constraint are not simply added to the RLOC values of the elements. In fact, the RLOC_RANGE constraint does not change the values of the RLOC constraints on underlying elements. It is an additional constraint that is attached automatically by the mapper to every member of a set.

The RLOC_RANGE constraint is attached to design elements in exactly the same way as the RLOC_ORIGIN constraint. The values of the RLOC_RANGE constraint, like RLOC_ORIGIN values, must be non-zero positive numbers since they directly correspond to die locations.

If a particular RLOC set is constrained by an RLOC_ORIGIN or an RLOC_RANGE constraint in the design netlist and is also constrained in the UCF file, the UCF file constraint overrides the netlist constraint.

Toggling the Status of RLOC Constraints

Another important set modifier is the USE_RLOC constraint. It turns the RLOC constraints on and off for a specific element or section of a set. USE_RLOC can be either TRUE or FALSE.

The application of the USE_RLOC constraint is strictly based on hierarchy. A USE_RLOC constraint attached to an element applies to all its underlying elements that are members of the same set. If it is attached to a symbol that defines the start of a set, the constraint is applied to all the underlying member elements, which represent the entire set.

However, if it is applied to an element below the start of the set (for example, E in Figure 70-12, page 275), only the members of the set (H and I) below the specified element are affected. You can also attach the USE_RLOC constraint directly to a primitive symbol so that it affects only that symbol.

When the USE_RLOC=FALSE constraint is applied, the RLOC and set constraints are removed from the affected symbols in the NCD file. This process is different than that followed for the RLOC_ORIGIN constraint. For RLOC_ORIGIN, the mapper generates

and outputs a LOC constraint in addition to all the set and RLOC constraints in the PCF file. The mapper does not retain the original constraints in the presence of a USE_RLOC=FALSE constraint because these cannot be turned on again in later programs.

Figure 70-12, page 275 illustrates the use of the USE_RLOC constraint to mask an entire set as well as portions of a set.

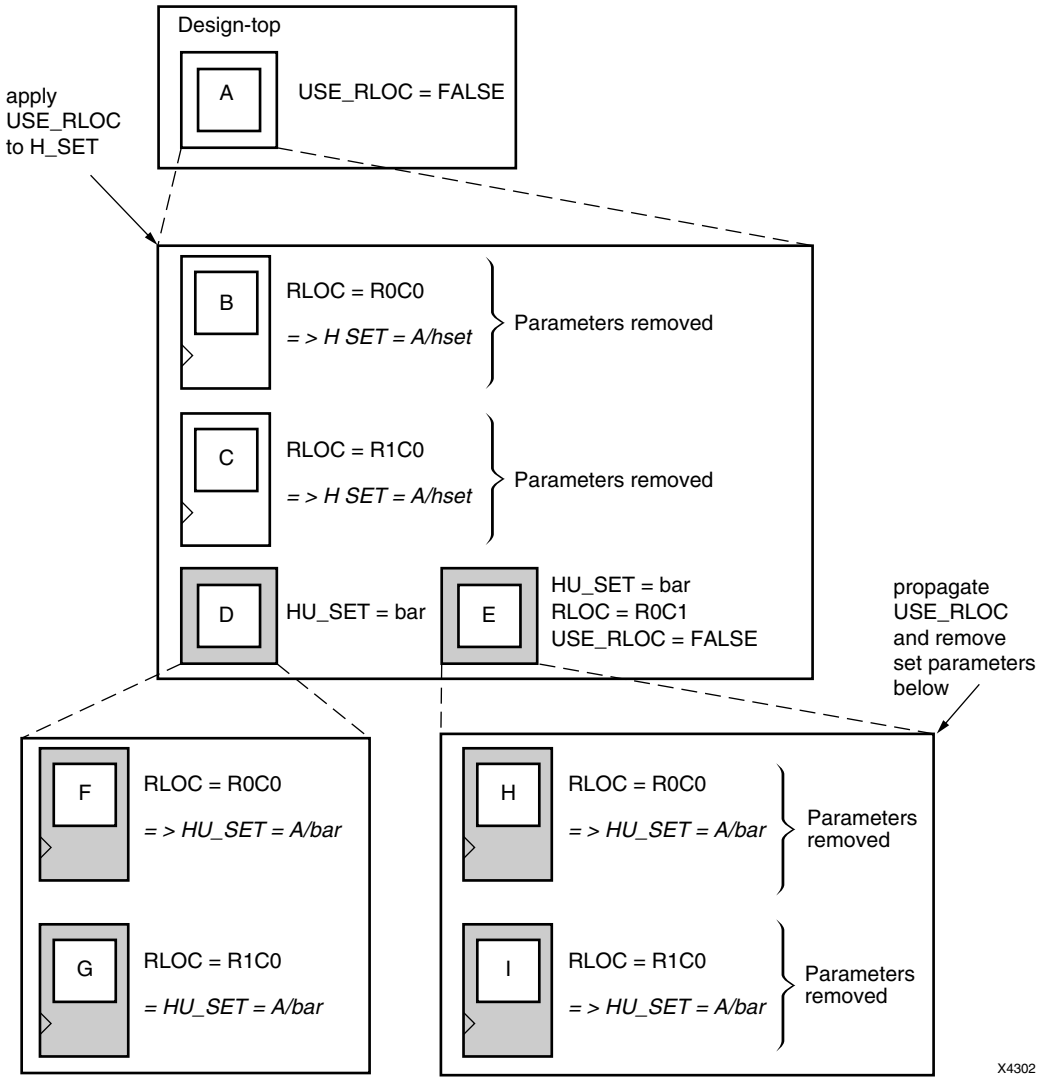

Figure 70-12: **Using the USE_RLOC Constraint to Control RLOC Application on H_SET and HU_SET Sets**

Applying the USE_RLOC constraint on U_SET sets is a special case because of the lack of hierarchy in the U_SET set. Because the USE_RLOC constraint propagates strictly in a hierarchical manner, the members of a U_SET set that are in different parts of the design hierarchy must be tagged separately with USE_RLOC constraints; no single USE_RLOC constraint is propagated to all the members of the set that lie in different parts of the hierarchy.

If you create a U_SET set through an instantiating macro, you can attach the USE_RLOC constraint to the instantiating macro to allow it to propagate hierarchically to all the members of the set.

You can create this instantiating macro by placing a U_SET constraint on a macro and letting the mapper propagate that constraint to every symbol with an RLOC constraint below it in the hierarchy.

Figure 70-13, page 276 illustrates an example of the use of the USE_RLOC=FALSE constraint. The USE_RLOC=FALSE on primitive E removes it from the U_SET set, and USE_RLOC=FALSE on element F propagates to primitive G and removes it from the U_SET set.

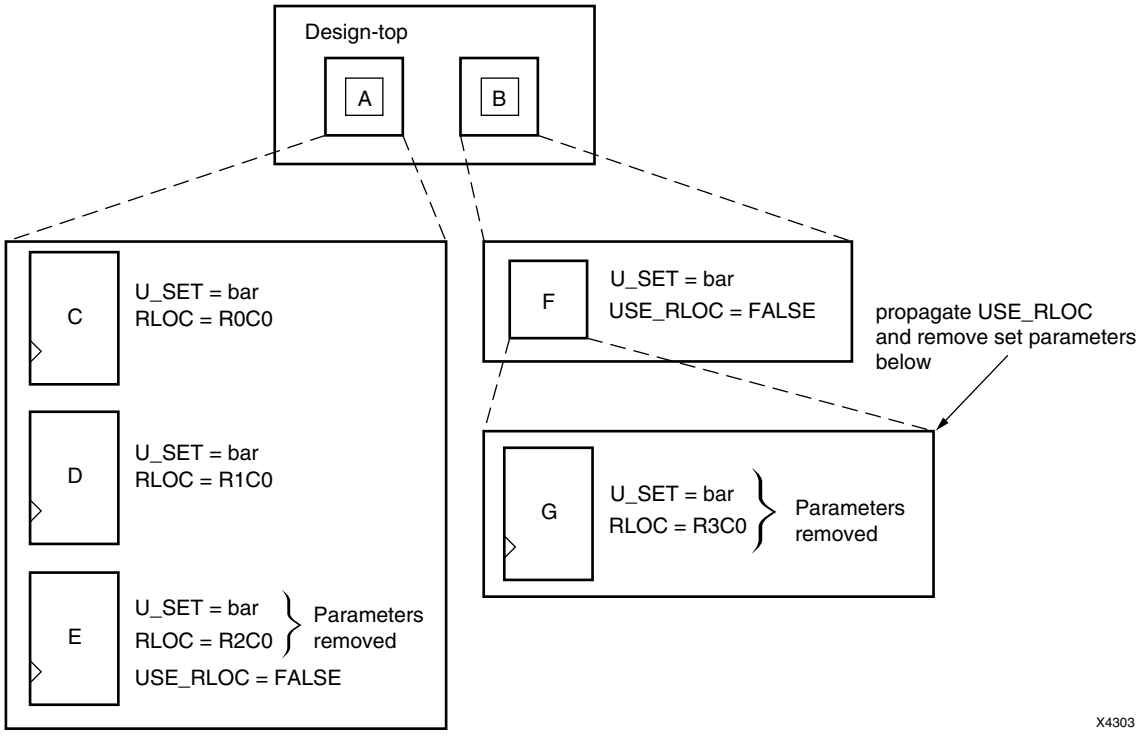

Figure 70-13: **Using the USE_RLOC Constraint to Control RLOC Application on U_SET Sets**

While propagating the USE_RLOC constraint, the mapper ignores underlying USE_RLOC constraints if it encounters elements higher in the hierarchy that already have USE_RLOC constraints. For example, if the mapper encounters an underlying element with a USE_RLOC=TRUE constraint during the propagation of a USE_RLOC=FALSE constraint, it ignores the newly encountered TRUE constraint.

Choosing an RLOC Origin when Using Hierarchy Sets

To specify a single origin for an RLOC set, use the following syntax, which is equivalent to placing an RLOC_ORIGIN constraint on the schematic.

- For CLB-based Row/Column/Slice Designations

  set_name **RLOC_ORIGIN=R***m***C***n*

  where

  - *set_name* can be the name of any type of RLOC set: a U_SET, an HU_SET, or a system-generated H_SET

- The origin itself is expressed as a row number and a column number representing the location of the elements at RLOC=R0C0

- For Slice-based XY Designations

  set_name **RLOC_ORIGIN=X***m***Y***n*

  where

  - *set_name* can be the name of any type of RLOC set: a U_SET, an HU_SET, or a system-generated H_SET
  - The origin itself is expressed as an X and Y value representing the location of the elements at RLOC=X0Y0

When RLOC_ORIGIN is used in conjunction with an implicit H_SET (hierarchy set), it must be placed on the element that is the start of the H_SET set, that is, on the lowest common ancestor of all the members of the set.

If you apply RLOC_ORIGIN to an HU_SET constraint, place it on the element at the start of the HU_SET set, that is, on an element with the HU_SET constraint.

However, since there could be several elements linked together with the HU_SET constraint at the same node, the RLOC_ORIGIN constraint can be applied to only one of these elements to prevent more than one RLOC_ORIGIN constraint from being applied to the HU_SET set.

Similarly, when used with a U_SET constraint, the RLOC_ORIGIN constraint can be placed on only one element with the U_SET constraint. If you attach the RLOC_ORIGIN constraint to an element that has only an RLOC constraint, the membership of that element in any set is removed, and the element is considered the start of a new H_SET set with the specified RLOC_ORIGIN constraint attached to the newly created set.

In Figure 70-14, page 278, the elements B, C, D, F, and G are members of an H_SET set with the name A/h_set. This figure is the same as Figure 70-5, page 265 except for the presence of an RLOC_ORIGIN constraint at the start of the H_SET set (at A).

The RLOC_ORIGIN values are added to the resultant RLOC values at each of the member elements to obtain the values that are then converted by the mapper to LOC constraints. For example, the RLOC value of F, given by adding the RLOC value at E (R0C1) and that at F (R0C0), is added to the RLOC_ORIGIN value (R2C3) to obtain the value of (R2C4), which is then converted to a LOC constraint, LOC = CLB_R2C4.

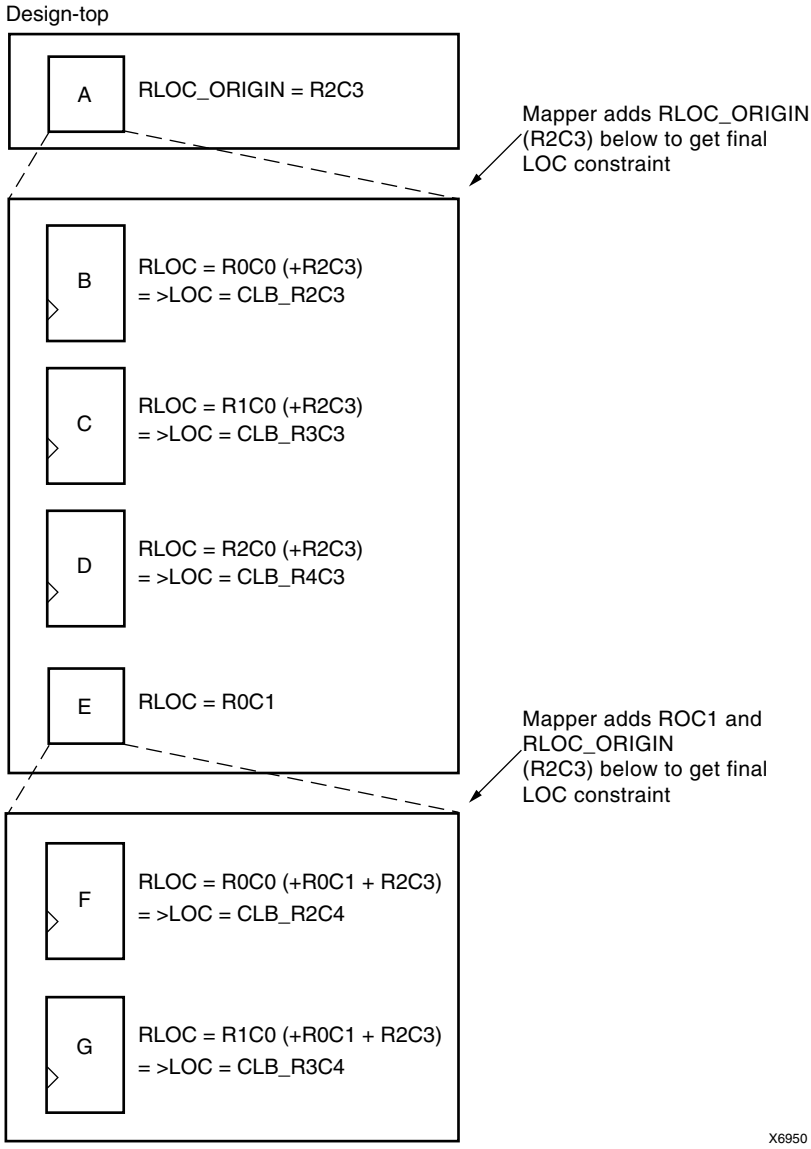

Figure 70-14: **Using an RLOC_ORIGIN Constraint to Modify an H_SET Set**

Figure 70-15, page 279 shows an example of an RLOC_ORIGIN constraint modifying an HU_SET constraint. The start of the HU_SET A/bar is given by element D or E. The RLOC_ORIGIN attached to E, therefore, applies to this HU_SET set. On the other hand, the RLOC_ORIGIN at A, which is the start of the H_SET set A/h_set, applies to elements B and C, which are members of the H_SET set.

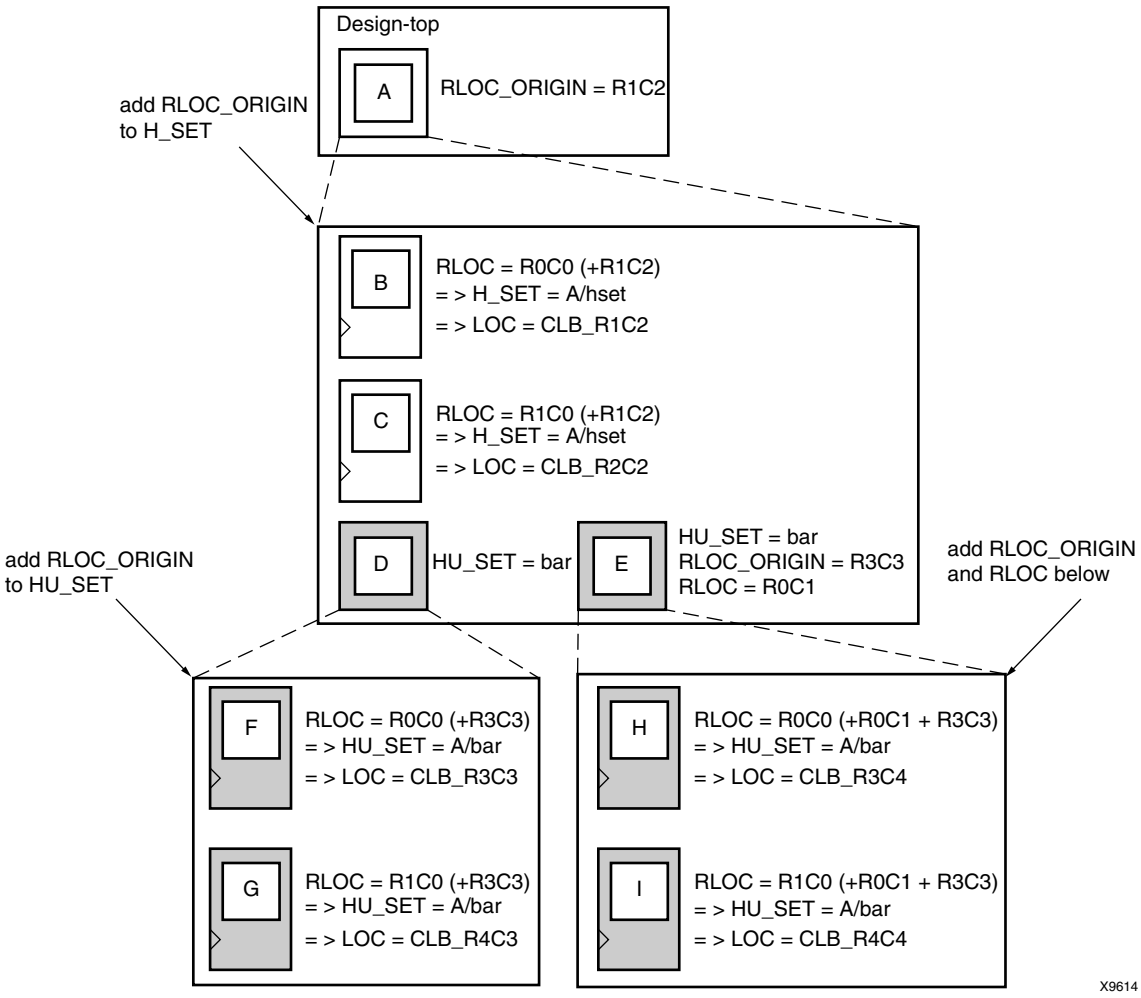

Figure 70-15: **Using an RLOC_ORIGIN to Modify H_SET and HU_SET Sets**

RLOC Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an instance
- Attribute Name: RLOC
- Attribute Values: See "RLOC Syntax" in this chapter.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute rloc: string;
```

Specify the VHDL constraint as follows for Virtex, Virtex-E, Spartan-II, and Spartan-IIE:

```
attribute rloc of {component_name|entity_name|label_name}:
{component|entity|label} is "[element]RmCn[.extension]";
```

Specify the VHDL constraint as follows for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, and Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

```
attribute rloc of {component_name|entity_name|label_name}:
{component|entity|label} is "[element]XmYn[.extension]";
```

For descriptions of valid values, see "Guidelines for Specifying Relative Locations" in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

The following code sample shows how to use RLOCs with a VHDL generate statement. The code is a simple example showing how to auto-generate the RLOCs for several instantiated FDEs. This methodology can be used with virtually any primitive.

```
LEN:for i in 0 to bits-1 generate
  constant row :natural:=((width-1)/2)-(i/2);
  constant column:natural:=0;
  constant slice:natural:=0;
  constant rloc_str : string := "R" & itoa(row) & "C" & itoa(column) &
".S" & itoa(slice);
  attribute RLOC of U1: label is rloc_str;
begin
  U1: FDE port map (
      Q  => dd(j),
      D  => ff_d,
      C  => clk,
      CE =>lcl_en(en_idx));
end generate LEN;
```

Verilog Syntax Example

Specify as follows for Virtex, Virtex-E, Spartan-II and Spartan-IIE:

```
(* RLOC = "[element]RmCn[.extension]" *)
```

Specify as follows for Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

```
(* RLOC = "[element]XmYn[.extension]" *)
```

For descriptions of valid value, see "Guidelines for Specifying Relative Locations" in this chapter. For more information about Verilog syntax, see"Verilog" in this chapter.

UCF and NCF Syntax Example

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, the following statement specifies that an instantiation of FF1 be placed in the CLB at row 4, column 4.

```
INST "/Virtex/design/FF1" RLOC=R4C4;
```

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, the following statement specifies that an instantiation of elemA be placed in the X flip-flop in the CLB at row 0, column 1.

```
INST "/$1I87/elemA" RLOC=r0c1.S0;
```

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, the following statement specifies that an instantiation of FF1 be placed in a slice that is +4 X coordinates and +4 Y coordinates relative to the origin slice.

```
INST "/V2/design/FF1" RLOC=X4Y4;
```

XCF Syntax Example

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices:

```
BEGIN MODEL "entity_name"
 INST "instance_name" rloc=[element]RmCn[.extension];
END;
```

For Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices:

```
BEGIN MODEL "entity_name"
 INST "instance_name" rloc=[element]XmYn[.extension];
END;
```

Floorplanner Syntax Example

D rag logic to locations on the Floorplan view. To write out RLOCs, save the constraints to an NCF file via the Write RPM to NCF... command on the File pulldown menu. For more information, see "Write RPM to NCF Command" in the Floorplanner help.

Relative Location Origin (RLOC_ORIGIN)

RLOC_ORIGIN Architecture Support

The RLOC_ORIGIN constraint applies to FPGA devices only.

RLOC_ORIGIN Applicable Elements

Instances or macros that are members of sets

RLOC_ORIGIN Propagation Rules

RLOC_ORIGIN is a macro constraint and any attachment to a net is illegal.

RLOC_ORIGIN Description

RLOC_ORIGIN is a placement constraint. It fixes the members of a set at exact die locations. RLOC_ORIGIN must specify a single location, not a range or a list of several locations. For more information, see "Set Modifiers" in the "Relative Location (RLOC)" constraint.

RLOC_ORIGIN is required for a set that includes BUFT symbols. RLOC_ORIGIN cannot be attached to a BUFT instance.

RLOC_ORIGIN Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an instance that is a member of a set
- Attribute Name: RLOC_ORIGIN
- Attribute Values: See "UCF and NCF Syntax Example" in this chapter.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute rloc_origin: string;
```

Specify the VHDL constraint as follows:

```
attribute rloc_origin of {component_name|entity_name|label_name}:
{component|entity|label} is "value";
```

For Virtex, Virtex-E, Spartan-II, and Spartan-II E devices, *value* is **R***m***C***n*.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, *value* is **X***m***Y***n*.

For a description of valid values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* RLOC_ORIGIN = "value" *)
```

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, *value* is **R***m***C***n*.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, *value* is **X***m***Y***n*.

For a description of valid values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

RLOC_ORIGIN Syntax for Architectures Using CLB-based Row/Column/Slice Specifications

```
RLOC_ORIGIN=RmCn
```

where

- *m* and *n* are positive or negative integers (including zero) representing relative row and column numbers, respectively

The following statement specifies that any RLOC statement applied to FF1 uses the CLB at R4C4 as its reference point. For example, if RLOC=R0C2 for FF1, then the instantiation of FF1 is placed in the CLB that occupies row 4 (R0 + R4), column 6 (C2 + C4).

```
INST "/archive/designs/FF1" RLOC_ORIGIN=R4C4;
```

RLOC_ORIGIN Syntax for Architectures Using Slice-Based XY Coordinates

This section applies to Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices only.

```
RLOC_ORIGIN=XmYn
```

where

- *m* and *n* are positive or negative integers (including zero) representing relative X and Y coordinates, respectively

The following statement specifies that an instantiation of FF1, which is a member of a set, be placed in the slice at X4Y4 relative to FF1. For example, if RLOC=X0Y2 for FF1, then the instantiation of FF1 is placed in the slice that is 0 rows to the right of X4 and 2 rows up from Y4 (X4Y6).

```
INST "/archive/designs/FF1" RLOC_ORIGIN=X4Y4;
```

Floorplanner Syntax Example

See "Writing RPM to UCF" in the Floorplanner help.

Relative Location Range (RLOC_RANGE)

RLOC_RANGE Architecture Support

The RLOC_RANGE constraint applies to FPGA devices only.

RLOC_RANGE Applicable Elements

Instances or macros that are members of sets

RLOC_RANGE Description

RLOC_RANGE is a placement constraint. It is similar to RLOC_ORIGIN except that it limits the members of a set to a certain range on the die. The range or list of locations is meant to apply to all applicable elements with RLOCs, not just to the origin of the set.

RLOC_RANGE Propagation Rules

RLOC_RANGE is a macro constraint and any attachment to a net is illegal.

RLOC_RANGE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an instance that is a member of a set
- Attribute Name: RLOC_RANGE
- Attribute Values: See "UCF and NCF Syntax Example" in this chapter.

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute rloc_range: string;
```

Specify the VHDL constraint as follows:

```
attribute rloc_range of {component_name|entity_name|label_name}:
{component|entity|label} is "value";
```

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE, *value* is **R***m1***C***n1*:**R***m2***C***n2*.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices *value* is **X***m1***Y***n1*:**X***m2***Y***n2*.

For a description of valid values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* RLOC_RANGE = "value" *)
```

For Virtex, Virtex-E, Spartan-II, and Spartan-IIE devices, *value* is **R***m1***C***n1*:**R***m2***C***n2*.

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 devices, *value* is **X***m1***Y***n1*:**X***m2***Y***n2*.

For a description of valid values, see "UCF and NCF Syntax Example" in this chapter.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

For Architectures Using CLB-Based Row/Column/Slice Specifications

This section is applicable to Virtex, Virtex-E, Spartan-II and Spartan-IIE devices only.

RLOC_RANGE=R*m1***C***n1***:R***m2***C***n2*

where

- the relative row numbers (*m1* and *m2*) and column numbers (*n1* and *n2*) can be positive integers (including zero)
- the wildcard (*) character

This syntax allows three kinds of range specifications, which are defined in "Set Modifiers."

The following statement specifies that an instantiation of the macro MACRO4 be placed within a region that is enclosed by the rows R4-R10 and the columns C4-C10.

```
INST "/archive/designs/MACRO4" RLOC_RANGE=R4C4:R10C10;
```

For Architectures Using Slice-Based XY Specifications

This section is applicable to Spartan-3 devices and up, and Virtex-II devices and up.

RLOC_RANGE=X*m1***Y***n1***:X***m2***Y***n2*

where

- the relative X values (*m1* and *m2*) and Y values (*n1* and *n2*) can be:
  - positive integers (including zero)
  - the wildcard (*) character

This syntax allows three kinds of range specifications, which are defined in "Set Modifiers."

The following statement specifies that an instantiation of the macro MACRO4 be placed relative to other members of the set within a region that is bounded by X4Y4 in the lower left corner and by X10Y10 in the upper right corner.

```
INST "/archive/designs/MACRO4" RLOC_RANGE=X4Y4:X10Y10;
```

XCF Syntax Example

```
MODEL "entity_name" rloc_range=value;

BEGIN MODEL "entity_name"
 INST "instance_name" rloc_range=value;
END;
```

PCF Syntax Example

RLOC_RANGE translates to a LOCATE constraint that has a range of sites. For example,
locate CLB_R1C1:CLB_R10C2

Save Net Flag (SAVE NET FLAG)

SAVE NET FLAG Architecture Support

The SAVE NET FLAG constraint applies to FPGA devices only.

SAVE NET FLAG Applicable Elements

- Nets
- Signals

SAVE NET FLAG Description

SAVE NET FLAG is a basic mapping constraint. Attaching the Save Net flag to nets or signals affects the mapping, placement, and routing of the design by preventing the removal of unconnected signals.

The flag prevents the removal of loadless or driverless signals. For loadless signals, the S constraint acts as a dummy OBUF load connected to the signal. For driverless signals the S constraint acts as a dummy IBUF driver connected to the signal.

If you do not have the S constraint on a net, any signal that cannot be observed or controlled via a path to an I/O primitive is removed.

The S constraint may prevent the trimming of logic connected to the signal. SAVE NET FLAG can be abbreviated S NET FLAG.

SAVE NET FLAG Propagation Rules

SAVE NET FLAG is a net or signal constraint. Any attachment to a design element is illegal.

SAVE NET FLAG prevents the removal of unconnected signals. If you do not have the S constraint on a net, any signal not connected to logic or an I/O primitive is removed.

SAVE NET FLAG Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net or signal
- Attribute Name: S
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute S: string;
```

Specify the VHDL constraint as follows:

```
attribute S of signal_name: signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* S = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The following statement specifies that the net or signal named $SIG_9 should not be removed.

```
NET "$SIG_9" S;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET  "signal_name" s=true;
END;
```

Schmitt Trigger (SCHMITT_TRIGGER)

SCHMITT_TRIGGER Architecture Support

The SCHMITT_TRIGGER constraint applies to the Coolrunner™-II CPLD only.

SCHMITT_TRIGGER Applicable Elements

All input pads and pad nets

SCHMITT_TRIGGER Description

This constraint causes the attached input pad to be configured with Schmitt Trigger (hysteresis). This constraint applies to any input pad in the design.

SCHMITT_TRIGGER Propagation Rules

The constraint is a net or signal constraint. Any attachment to a macro, entity, or module is illegal.

SCHMITT_TRIGGER Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: SCHMITT_TRIGGER
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute SCHMITT_TRIGGER: string;
```

Specify the VHDL constraint as follows:

```
attribute SCHMITT_TRIGGER of signal_name: signal is "{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* SCHMITT_TRIGGER = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'SCHMITT_TRIGGER mysignal';
```

UCF and NCF Syntax Example

```
NET “mysignal” SCHMITT_TRIGGER;
```

XCF Syntax Example

```
BEGIN MODEL “entity_name”
 NET  "signal_name" SCHMITT_TRIGGER=true;
END;
```

Slew (SLEW)

SLEW Architecture Support

The SLEW constraint applies to all FPGA and CPLD devices.

SLEW Applicable Elements

The SLEW attribute should only be placed on a top-level output or bi-directional port.

SLEW Description

The SLEW constraint is used to define the slew rate (rate of transition) behavior of each individual output to the device. This attribute may be placed on any output or bi-directional port to specify the port slew rate to be SLOW (default), FAST, or QUIETIO (Spartan-3A and Spartan-3A DSP). Use the slowest SLEW attribute available to the device while still allowing applicable I/O timing to be met in order to minimize any possible signal integrity issues.

The LVCMOS SLEW cannot be changed for Virtex-E and Spartan-IIE devices.

SLEW Propagation Rules

The SLEW attribute should only be placed on a top-level output or bi-directional port.

SLEW Syntax Examples

Following are syntax examples using this constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

Specify a new attribute to an output port, or bi-directional port:

- Attribute Name: SLEW
- Attribute Values: FAST, SLOW, QUIETIO (Spartan-3A only)

VHDL Syntax Example

Before using SLEW, declare it with the following syntax placed after the architecture declaration, but before the begin statement in the top-level VHDL file:

```
attribute SLEW: string;
```

After SLEW has been declared, specify the VHDL constraint as follows:

```
attribute SLEW of {top_level_port_name}: signal is "value";
```

Where *value* is SLOW, FAST, QUIETIO (Spartan-3A only)

Example:

```
entity top is
   port (FAST_OUT: out std_logic);
end top;
architecture MY_DESIGN of top is
```

```
attribute SLEW: string;
   attribute SLEW of FAST_OUT: signal is "FAST";
begin
```

For a more detailed discussion of the basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Place the following attribute specification before the port declaration in the top-level Verilog code:

```
(* SLEW="value" *)
```

Where *value* is SLOW, FAST, QUIETIO (Spartan-3A only)

Example:

```
module top (
         (* SLEW="FAST" *) output FAST_OUT
);
```

For a more detailed discussion of the basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

Placed on output or bi-directional port:

```
NET "top_level_port_name" SLEW="value";
```

Where *value* is SLOW, FAST, QUIETIO (Spartan-3A only).

Example:

```
NET "FAST_OUT" SLEW="FAST";
```

PACE Syntax Example

The SLEW attribute can be set from within the PACE (Assign Package Pins) tool by selecting the appropriate value for the desired pin from the Design Objects window.

Slow (SLOW)

SLOW Architecture Support

The SLOW constraint applies to all FPGA and CPLD devices.

SLOW Applicable Elements

- Output primitives
- Output pads
- Bidirectional pads

You can also attach SLOW to the net connected to the pad component in a UCF file. NGDBuild transfers SLOW from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following UCF syntax:

```
NET "net_name" SLOW;
```

SLOW Description

SLOW is a basic fitter constraint. It stipulates that the slew rate limited control should be enabled.

SLOW Propagation Rules

SLOW is illegal when attached to a net except when the net is connected to a pad. In this case, SLOW is treated as attached to the pad instance.

When attached to a design element, SLOW propagates to all applicable elements in the hierarchy within the design element.

SLOW Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: SLOW
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute SLOW : string;
```

Specify the VHDL constraint as follows:

```
attribute SLOW of {signal_name|entity_name}: {signal|entity} is
"{TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* SLOW = "{TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

ABEL Syntax Example

```
XILINX PROPERTY 'SLOW mysignal';
```

UCF and NCF Syntax Example

The following statement establishes a slow slew rate for an instantiation of the element y2.

```
INST "$1I87/y2" SLOW;
```

The following statement establishes a slow slew rate for the pad to which net1 is connected.

```
NET "net1" SLOW;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Ports tab grid with I/O Configuration Options checked, click the FAST/SLOW column in the row with the desired output port name and choose SLOW from the drop-down list.

Stepping (STEPPING)

STEPPING Architecture Support

The STEPPING constraint applies to the following devices:

- Virtex™-II
- Virtex-II Pro
- Virtex-II Pro X
- Virtex-4
- Virtex-5
- Spartan™-3A
- Spartan-3E
- CoolRunner™-II

STEPPING Applicable Elements

The STEPPING attribute is a global CONFIG constraint and is not attached to any instance or signal name.

STEPPING Description

The STEPPING constraint is assigned a value that matches the step level marking on the silicon; the step level identifies specific device capabilities. Xilinx recommends that the step level be set for the design using the STEPPING constraint, otherwise, the software uses a default target device.

For more information on STEPPING, see Xilinx Answer Record 20947, *"Stepping FAQs."*

STEPPING Propagation Rules

The CONFIG STEPPING constraints applies to an entire design.

STEPPING Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

```
CONFIG STEPPING="n";
```

where

- *n* is the target stepping level (ES, SCD1, 1, 2, 3, ...)

For example:

```
CONFIG STEPPING="1";
```

Suspend (SUSPEND)

SUSPEND Architecture Support

The SUSPEND constraint applies to Spartan™-3A devices only.

SUSPEND Applicable Elements

The SUSPEND attribute should only be placed on a top-level output or bi-directional port targeting a Spartan-3A device.

SUSPEND Description

The SUSPEND constraint is used to define the behavior of each individual output to the device when the FPGA is placed in the SUSPEND power-reduction mode. This attribute may be placed on any output or bi-directional port to specify the port to be 3-stated (3STATE), pulled high (3STATE_PULLUP), or low (3STATE_PULLDOWN), or driven to the last value (3STATE_KEEPER or DRIVE_LAST_VALUE). The default value is 3STATE.

SUSPEND Propagation Rules

The SUSPEND attribute should only be placed on a top-level output or bi-directional port.

SUSPEND Syntax Examples

Following are syntax examples using this constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

Specify a new attribute to an output port or bidirectional port:

- Attribute Name: SUSPEND
- Attribute Values: DRIVE_LAST_VALUE, 3STATE, 3STATE_PULLUP, 3STATE_PULLDOWN or 3STATE_KEEPER

VHDL Syntax Example

Before using SUSPEND, declare it with the following syntax placed after the architecture declaration but before the begin statement in the top-level VHDL file:

```
attribute SUSPEND: string;
```

After SUSPEND has been declared, specify the VHDL constraint as follows:

```
attribute SUSPEND of {top_level_port_name}: signal is "value";
```

Where *value* is DRIVE_LAST_VALUE, 3STATE, 3STATE_PULLUP, 3STATE_PULLDOWN or 3STATE_KEEPER

Example:

```
entity top is
   port (STATUS: out std_logic);
end top;
```

```
architecture MY_DESIGN of top is
attribute SUSPEND: string;
   attribute SUSPEND of STATUS: signal is "DRIVE_LAST_VALUE";
begin
```

For a more detailed discussion of the basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Place the following attribute specification before the port declaration in the top-level Verilog code:

```
(* SUSPEND="value" *)
```

Where *value* is DRIVE_LAST_VALUE, 3STATE, 3STATE_PULLUP, 3STATE_PULLDOWN or 3STATE_KEEPER

Example:

```
module top (
(* SUSPEND="DRIVE_LAST_VALUE" *) output STATUS
);
```

For a more detailed discussion of the basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

Placed on output or bi-directional port:

```
NET "top_level_port_name" SUSPEND="value";
```

Where *value* is DRIVE_LAST_VALUE, 3STATE, 3STATE_PULLUP, 3STATE_PULLDOWN or 3STATE_KEEPER

Example:

```
NET "STATUS" SUSPEND="DRIVE_LAST_VALUE";
```

Pace Syntax Example

The SUSPEND attribute can be set from within the Pace (Assign Package Pins) tool by selecting the appropriate value for the desired pin from the Design Objects window.

System Jitter (SYSTEM_JITTER)

SYSTEM_JITTER Architecture Support

The SYSTEM_JITTER constraint applies to FPGA devices only.

SYSTEM_JITTER Applicable Elements

Applies globally to the entire design

SYSTEM_JITTER Description

This constraint specifies the system jitter of the design. SYSTEM_JITTER depends on various design conditions -- for example, the number of flip-flops changing at one time and the number of I/Os changing. The SYSTEM_JITTER constraint applies to all of the clocks within a design. It can be combined with the INPUT_JITTER keyword on the PERIOD constraint to generate the Clock Uncertainty value that is shown in the timing report.

SYSTEM_JITTER Propagation Rules

Not applicable

SYSTEM_JITTER Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: SYSTEM_JITTER

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute SYSTEM_JITTER: string;
```

Specify the VHDL constraint as follows:

```
attribute SYSTEM_JITTER of
{component_name|signal_name|entity_name|label_name}:
{component|signal|entity|label} is "value ps";
```

where

- *value* is a numerical value. The default is ps.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* SYSTEM_JITTER = "value ps" *)
```

where

- *value* is a numerical value. The default is ps.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
SYSTEM_JITTER= value ps;
```

where

- *value* is a numerical value. The default is ps.

XCF Syntax Example

```
MODEL "entity_name" SYSTEM_JITTER = value ps;
```

Temperature (TEMPERATURE)

TEMPERATURE Architecture Support

The TEMPERATURE constraint applies to FPGA devices only.

TEMPERATURE Applicable Elements

Global

TEMPERATURE Description

TEMPERATURE is an advanced timing constraint. It allows the specification of the operating junction temperature. TEMPERATURE provides a means of device delay characteristics based on the specified temperature. Prorating is a scaling operation on existing speed file delays and is applied globally to all delays.

Note: Newer devices may not support Temperature prorating until the timing information (speed files) are marked as production status.

Each architecture has its own specific range of valid operating temperatures. If the entered temperature does not fall within the supported range, TEMPERATURE is ignored and an architecture-specific worst-case value is used instead. Also note that the error message for this condition does not appear until static timing.

TEMPERATURE Propagation Rules

It is illegal to attach TEMPERATURE to a net.

TEMPERATURE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
TEMPERATURE=value [C |F| K];
```

where

- *value* is a real number specifying the temperature
- C, K, and F are the temperature units
  - F is degrees Fahrenheit
  - K is degrees Kelvin
  - C is degrees Celsius, the default

The following statement specifies that the analysis for everything relating to speed file delays assumes a junction temperature of 25 degrees Celsius.

```
TEMPERATURE=25 C;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Misc tab, click Specify next to Temperature and fill out the temperature dialog box.

Timing Ignore (TIG)

TIG Architecture Support

The TIG constraint applies to FPGA devices only.

TIG Applicable Elements

- Nets
- Pins
- Instances

TIG Description

TIG (Timing IGnore) is a basic timing constraint and a synthesis constraint. It causes paths that fan forward from the point of application (of TIG) to be treated as if they do not exist (for the purposes of the timing model) during implementation.

You may apply a TIG relative to a specific timing specification.

The value of TIG may be any of the following:

- Empty (global TIG that blocks all paths)
- A single TSid to block
- A comma separated list of TSids to block, for example

```
NET "RESET" TIG=TS_fast, TS_even_faster;
```

XST fully supports the TIG constraint.

TIG Propagation Rules

If TIG is attached to a net, primitive pin, or macro pin, all paths that fan forward from the point of application of the constraint are treated as if they do not exist for the purposes of timing analysis during implementation. In the following figure, NET C is ignored.

However, note that the lower path of NET B that runs through the two OR gates would not be ignored.

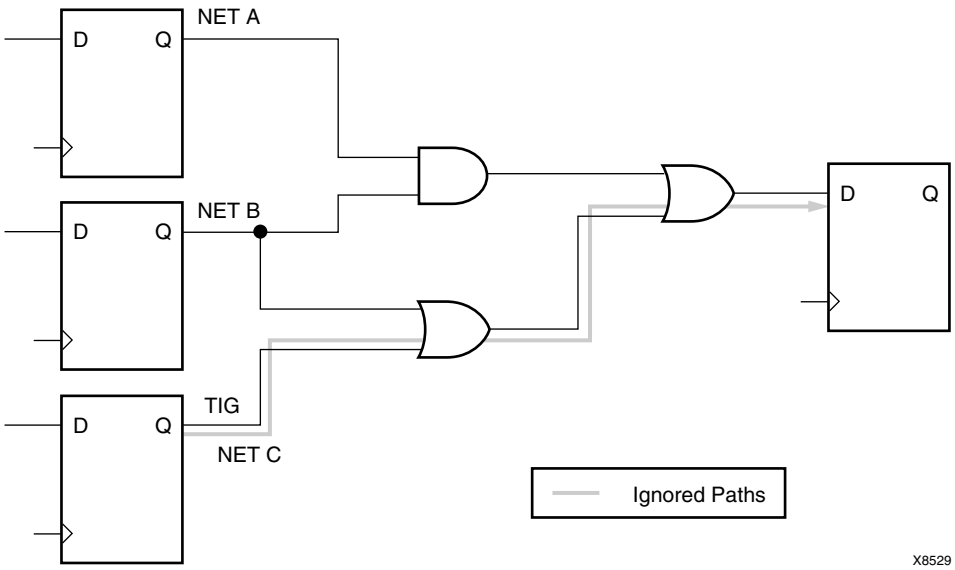

Figure 81-1: **TIG Example**

The following constraint would be attached to a net to inform the timing analysis tools that it should ignore paths through the net for specification TS43:

Schematic syntax

```
TIG = TS43
```

UCF syntax

```
NET "net_name" TIG = TS43;
```

You cannot perform path analysis in the presence of combinatorial loops. Therefore, the timing tools ignore certain connections to break combinatorial loops. You can use the TIG constraint to direct the timing tools to ignore specified nets or load pins, consequently controlling how loops are broken.

TIG Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Note: The TIG constraint does not have any affect on the timing reported at the bottom of the XST report. TIG only applies to the timing reported by Timing Analyzer.

Schematic Syntax Example

- Attach to a net or pin.
- Attribute Name: TIG
- Attribute Values: *value*

UCF and NCF Syntax Example

The basic UCF syntax is:

```
NET "net_name" TIG;
PIN "ff_inst.RST" TIG=TS_1;
INST "instance_name" TIG=TS_2;
TIG=TSidentifier1,..., TSidentifiern
```

where

- *identifier* refers to a timing specification that should be ignored

When attached to an instance, TIG is pushed to the output pins of that instance. When attached to a net, TIG pushes to the drive pin of the net. When attached to a pin, TIG applies to the pin.

The following statement specifies that the timing specifications TS_fast and TS_even_faster is ignored on all paths fanning forward from the net RESET.

```
NET "RESET" TIG=TS_fast, TS_even_faster;
```

XCF Syntax Example

Same as UCF syntax

XST fully supports the TIG constraint. TIG can be applied to the nets, situated in the CORE files (EDIF, NGC) as well.

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Specify next to "False Paths (FROM TO TIG)" and fill out the FROM/THRU/TO dialog box or click Specify next to "False Paths by Net (NET TIG)" and fill out the Timing Ignore dialog box.

PCF Syntax Example

item **TIG;**

item **TIG = ;**

item **TIG = TS***identifier***;**

where

- *item* is:
  - **PIN** *name*
  - **PATH** *name*
  - *path specification*
  - **NET** *name*
  - **TIMEGRP** *name*
  - **BEL** *name*
  - **COMP** *name*
  - **MACRO** *name*

Timing Group (TIMEGRP)

TIMEGRP Architecture Support

The TIMEGRP constraint applies to all FPGA and CPLD devices.

TIMEGRP Applicable Elements

- Design elements
- Nets

TIMEGRP Description

TIMEGRP is a basic grouping constraint. In addition to naming groups using the TNM identifier, you can also define groups in terms of other groups. You can create a group that is a combination of existing groups by defining a TIMEGRP constraint.

You can place TIMEGRP constraints in a constraints file (UCF and NCF).

You can use TIMEGRP attributes to create groups using the following methods.

- "Combining Multiple Groups into One"
- "Creating Groups by Exclusion"
- "Defining Flip-Flop Subgroups by Clock Sense"

Combining Multiple Groups into One

You can define a group by combining other groups. The following syntax example illustrates the simple combining of two groups:

UCF Syntax Example One

```
TIMEGRP "big_group"="small_group" "medium_group";
```

In this syntax example, *small_group* and *medium_group* are existing groups defined using a TNM or TIMEGRP attribute.

UCF Syntax Example Two

A circular definition, as shown below, causes an error when you run your design through NGDBuild:

```
TIMEGRP "many_ffs"="ffs1" "ffs2";
TIMEGRP "ffs1"="many_ffs" "ffs3";
```

Creating Groups by Exclusion

You can define a group that includes all elements of one group except the elements that belong to another group, as illustrated by the following syntax examples:

UCF Syntax Example One

```
TIMEGRP "group1"="group2" EXCEPT "group3";
```

where

- *group1* represents the group being defined. It contains all of the elements in *group2* except those that are also in *group3.*
- *group2* and *group3* can be a:
  - valid TNM
  - predefined group
  - TIMEGRP attribute

UCF Syntax Example Two

As illustrated by the following example, you can specify multiple groups to include or exclude when creating the new group.

```
TIMEGRP "group1"="group2" "group3" EXCEPT "group4" "group5";
```

The example defines a *group1* that includes the members of *group2* and *group3*, except for those members that are part of *group4* or *group5*. All of the groups before the keyword EXCEPT are included, and all of the groups after the keyword are excluded.

Defining Flip-Flop Subgroups by Clock Sense

You can create subgroups using the keywords RISING and FALLING to group flip-flops triggered by rising and falling edges.

UCF Syntax Example One

```
TIMEGRP "group1"=RISING FFS;
TIMEGRP "group2"=RISING "ffs_group";
TIMEGRP "group3"=FALLING FFS;
TIMEGRP "group4"=FALLING "ffs_group";
```

where

- *group1* to *group4* are new groups being defined. The *ffs_group* must be a group that includes only flip-flops.

Keywords, such as EXCEPT, RISING, and FALLING, appear in the documentation in upper case; however, you can enter them in either lower or upper case. You cannot enter them in a combination of lower and upper case.

UCF Syntax Example Two

The following example defines a group of flip-flops that switch on the falling edge of the clock.

```
TIMEGRP "falling_ffs"=FALLING FFS;
```

Defining Latch Subgroups by Gate Sense

Groups of type LATCHES (no matter how these groups are defined) can be easily separated into transparent high and transparent low subgroups. The TRANSHI and TRANSLO keywords are provided for this purpose and are used in TIMEGRP statements like the RISING and FALLING keywords for flip-flop groups.

UCF Syntax Example

```
TIMEGRP "lowgroup"=TRANSLO "latchgroup";
TIMEGRP "highgroup"=TRANSHI "latchgroup";
```

Creating Groups by Pattern Matching

When creating groups, you can use wildcard characters to define groups of symbols whose associated net names match a specific pattern. This is typically used in schematic designs where net names are specified, not instance names. Synthesis plans typically use INST/TNM syntax. For more information, see the "Timing Name (TNM)" constraint.

How to Use Wildcards to Specify Net Names

The wildcard characters, asterisk (*) and question mark (?), enable you to select a group of symbols whose output net names match a specific string or pattern. The asterisk (*) represents any string of zero or more characters. The question mark (?) indicates a single character.

For example, DATA* indicates any net name that begins with "DATA," such as DATA, DATA1, DATA22, and DATABASE. The string NUMBER? specifies any net names that begin with "NUMBER" and end with one single character, for example, NUMBER1 or NUMBERS, but not NUMBER or NUMBER12.

You can also specify more than one wildcard character. For example, *AT? specifies any net names that begin with any series of characters followed by "AT" and end with any one character such as BAT1, CAT2, and THAT5. If you specify *AT*, you would match BAT11, CAT26, and THAT50.

Pattern Matching Syntax

UCF Syntax Example

The syntax for creating a group using pattern matching is:

```
TIMEGRP "group_name"=predefined_group("pattern");
```

where

- *predefined_group* can be one of the following predefined groups only: FFS, LATCHES, PADS, RAMS, CPUS, HSIOS, DSPS, BRAM_PORTA, BRAM_PORTB, or MULTS. For information on the definition of these groups, see "UCF and NCF Syntax Example" in the "Timing Name Net (TNM_NET)" constraint.
- *pattern* is any string of characters used in conjunction with one or more wildcard characters.

When specifying a net name, you must use its full hierarchical path name so PAR can find the net in the flattened design.

For FFS, RAMs, LATCHES, PADS, CPUS, DSPS, HSIOS, and MULTS, specify the output net name. For pads, specify the external net name.

UCF Syntax Example

The following example illustrates a group that includes the flip-flops that source nets whose names begin with $1I3/FRED.

```
TIMEGRP "group1"=FFS("$1I3/FRED*");
```

UCF Syntax Example

The following example illustrates a group that excludes certain flip-flops whose output net names match the specified pattern.

```
TIMEGRP "this_group"=FFS EXCEPT FFS("a*");
```

where

- *this_group* includes all flip-flops except those whose output net names begin with the letter "a"

UCF Syntax Example

The following example defines a group named "some_latches."

```
TIMEGRP "some_latches"=latches("$113/xyz*");
```

where

- the group *some_latches* contains all input latches whose output net names start with "$1I3/xyz"

Additional Pattern Matching Details

In addition to using pattern matching when you create timing groups, you can specify a predefined group qualified by a pattern any place you specify a predefined group. The syntax below illustrates how pattern matching can be used within a timing specification.

UCF Syntax Example

```
TIMESPEC "TSidentifier"=FROM predefined_group("pattern") TO
predefined_group
("pattern") value;
```

Instead of specifying one pattern, you can specify a list of patterns separated by a colon.

UCF Syntax Example

```
TIMEGRP "some_ffs"=FFS("a*:b?:c*d");
```

where

- The group *some_ffs* contains flip-flops whose output net names adhere to one of the following rules.
  - Start with the letter "a"
  - Contain two characters; the first character is "b"
  - Start with "c" and end with "d"

Defining Area Groups Using Timing Groups

For more information, see "Defining From Timing Groups" in the "Area Group (AREA_GROUP)" constraint.

TIMEGRP Propagation Rules

Applies to all elements or nets within the group

TIMEGRP Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF Syntax Example

Following are TIMEGRP UCF Syntax Examples.

Example One

```
TIMEGRP "newgroup"="existing_grp1" "existing_grp2" ["existing_grp3" .
. .];
```

where

- *newgroup* is a newly created group that consists of:
  - existing groups created via TNMs
  - predefined groups
  - other TIMEGRP attributes

Example Two

```
TIMEGRP "GROUP1" = "gr2" "GROUP3";
TIMEGRP "GROUP3" = FFS except "grp5";
```

XCF Syntax Example

XST supports TIMEGRP with the following limitations:

- Groups Creation by Exclusion is not supported
- When a group is defined on the basis of another user group with pattern matching;

  TIMEGRP TG1 = FFS (machine*); # Supported

  TIMEGRP TG2 = TG1 (machine_clk1*); # Not supported

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Create next to "Group elements by element output net name" and fill out the Time Group dialog box.

PCF Syntax Example

TIMEGRP *name*;

TIMEGRP *name* = *list of elements*;

Timing Specifications (TIMESPEC)

TIMESPEC Architecture Support

The TIMESPEC constraint applies to all FPGA and CPLD devices.

TIMESPEC Applicable Elements

TS identifiers

TIMESPEC Description

TIMESPEC is a basic timing related constraint. TIMESPEC serves as a placeholder for timing specifications, which are called TS attribute definitions. Every TS attribute begins with the letters "TS" and ends with a unique identifier that can consist of letters, numbers, or the underscore character (_).

TIMESPEC Propagation Rules

Not applicable

TIMESPEC Syntax Examples

UCF Syntax Example

A TS attribute defines the allowable delay for paths in your design. The basic syntax for a TS attribute is:

```
TIMESPEC "TSidentifier"=PERIOD "timegroup_name" value [units];
```

where

- **TS***identifier* is a unique name for the TS attribute
- *value* is a numerical value
- *units* can be ms, micro, ps, ns

```
TIMESPEC "TSidentifier"=PERIOD "timegroup_name" "TSidentifier" [* or /]
factor PHASE [+ |-] phase_value [units];
```

Syntax Rules

The following syntax rules apply.

Value Parameter

The *value* parameter defines the maximum delay for the attribute. Nanoseconds are the default units for specifying delay time in TS attributes. You can also specify delay using other units, such as picoseconds or megahertz.

Keywords

Keywords, such as FROM, TO, and TS, appear in the documentation in upper case. However, you can enter them in the TIMESPEC primitive in either upper or lower case. The characters in the keywords must be all upper case or all lower case. Examples of acceptable keywords are:

- FROM
- PERIOD
- TO
- from
- to

Examples of unacceptable keywords are:

- From
- To
- fRoM
- tO

TSidentifier Name

If a TSidentifier name is referenced in a property value, it must be entered in upper case letters. For example, the TSID1 in the second constraint below must be entered in upper case letters to match the TSID1 name in the first constraint.

```
TIMESPEC "TSID1" = FROM "gr1" TO "gr2" 50;
TIMESPEC "TSMAIN" = FROM "here" TO "there" TSID1 /2
```

Separators

A colon may be used as a separator instead of a space in all timing specifications.

TIMESPEC FROM-TO Syntax

Within TIMESPEC, you use the following UCF syntax to specify timing requirements between specific end points.

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group" value
units;
TIMESPEC "TSidentifier"=FROM "source_group" value units;
TIMESPEC "TSidentifier"=TO "dest_group" value units;
```

Unspecified FROM or TO, as in the second and third syntax statements, implies all points.

Note: Although you can use a FROM or TO statement to imply all points, you cannot use an unspecified THRU statement by itself to imply all points.

The From-To statements are TS attributes that reside in the TIMESPEC primitive. The parameters *source_group* and *dest_group* must be one of the following:

- Predefined groups
- Previously created TNM identifiers
- Groups defined in TIMEGRP symbols
- TPSYNC groups

Predefined groups consist of FFS, LATCHES, RAMS, PADS, CPUS, DSPS, HSIOS, BRAMS_PORTA, BRAMS_PORTB, and MULTS. These groups are defined in the section entitled "UCF and NCF Syntax Example," in the discussion of TNM_NET, and are discussed in "Grouping Constraints" of the Constraints Type chapter.

Keywords, such as FROM, TO, and TS appear in the documentation in upper case. However, you use them TIMESPEC in either upper or lower case. You cannot enter them in a combination of lower and upper case.

The *value* parameter defines the maximum delay for the attribute. Nanoseconds are the default units for specifying delay time in TS attributes. You can also specify delay using other units, such as picoseconds or megahertz.

TIMESPEC Examples of FROM-TO TS Attributes

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Examples

```
TIMESPEC "TS_master"=PERIOD "master_clk" 50 HIGH 30;
TIMESPEC "TS_THIS"=FROM FFS TO RAMS 35;
TIMESPEC "TS_THAT"=FROM PADS TO LATCHES 35;
```

Constraints Editor Syntax Examples

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. In the help index for the Constraints Editor, double-click "TIMESPEC."

Timing Name (TNM)

TNM Architecture Support

The TNM constraint applies to all FPGA and CPLD devices.

TNM Applicable Elements

You can attach TNM constraints to a net, an element pin, a primitive, or a macro.

You can attach the TNM constraint to the net connected to the pad component in a UCF file. NGDBuild transfers the constraint from the net to the pad instance in the NGD file so that it can be processed by the mapper. Use the following UCF syntax:

```
NET "net_name" TNM="property_value";
```

TNM Description

TNM is a basic grouping constraint. Use TNM (Timing Name) to identify the elements that make up a group which you can then use in a timing specification.

TNM tags specific FFS, RAMs, LATCHES, PADS, CPUS, HSIOS, and MULTS as members of a group to simplify the application of timing specifications to the group.

The RISING and FALLING keywords may also be used with TNMs.

TNM Propagation Rules

When attached to a net or signal, TNM propagates to all synchronous elements driven by that net. No special propagation is required.

When attached to a design element, TNM propagates to all applicable elements in the hierarchy within the design element.

The following rules apply to TNMs.

- TNMs applied to pad nets *do not* propagate forward through IBUFs. The TNM is applied to the external pad. This case includes the net attached to the D input of an IFD. See "Timing Name Net (TNM_NET)" if you want the TNM to trace forward from an input pad net.
- TNMs applied to an IBUF instance are illegal.
- TNMs applied to the output pin of an IBUF propagate the TNM to the next appropriate element.
- TNMs applied to an IBUF element stay attached to that element.
- TNMs applied to a clock-pad-net does not propagate forward through the clock buffer.
- When TNM is applied to a macro, all the elements in the macro have that timing name.

Special rules apply when using TNM with the PERIOD constraint for Virtex, Virtex-II, Spartan-II CLKDLLs and CLKDLLHFs, and related architectures.

Placing TNMs on Nets

You can place TNM on any net in the design. The constraint indicates that the TNM value should be attached to all valid elements fed by all paths that fan forward from the tagged net. Forward tracing stops at FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, and MULTS. TNMs do not propagate across IBUFs if they are attached to the input pad net.

Placing TNMs on Macro or Primitive Pins

You can place TNM on any macro or component pin in the design if the design entry package allows placement of constraints on macro or primitive pins. The constraint indicates that the TNM value should be attached to all valid elements fed by all paths that fan forward from the tagged pin. Forward tracing stops at FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, and MULTS. The following illustration shows the valid elements for a TNM attached to the schematic of a macro pin.

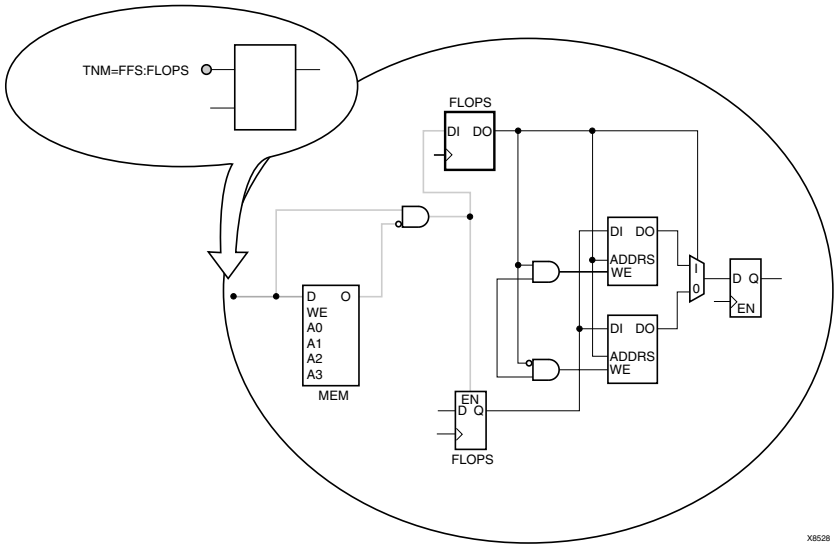

Figure 84-1: **TNM Placed on a Macro Pin**

The syntax for the UCF file is:

```
PIN "pin_name" TNM="FLOPS";
```

Placing TNMs on Primitive Symbols

You can group individual logic primitives explicitly by flagging each symbol, as illustrated by the following figure.

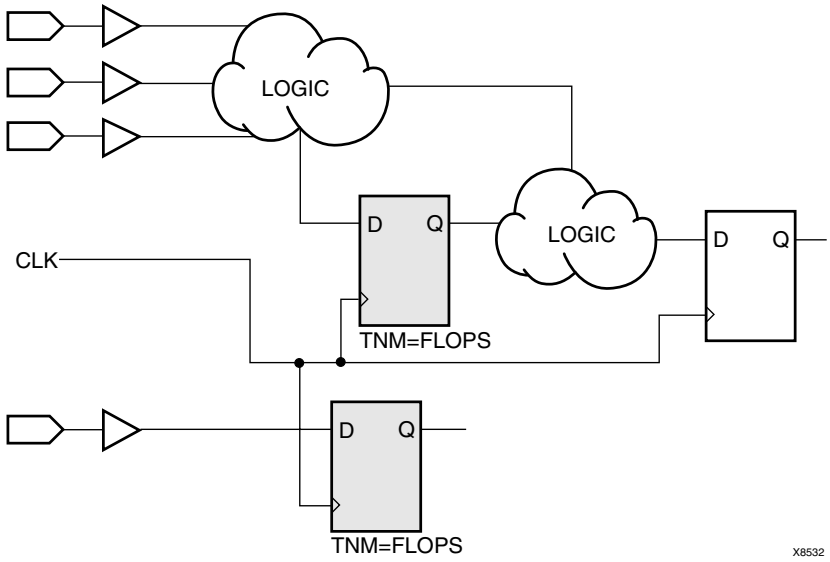

Figure 84-2: **TNM on Primitive Symbols**

In the figure, the flip-flops tagged with the TNM form a group called "'FLOPS." The untagged flip-flop on the right side of the drawing is not part of the group.

Place only one TNM on each symbol, driver pin, or macro driver pin.

Schematic Syntax Example

```
TNM=FLOPS;
```

UCF Syntax Example

```
INST "instance_name" TNM=FLOPS;
```

Placing TNMs on Macro Symbols

A macro is an element that performs some general purpose higher level function. It typically has a lower level design that consists of primitives, other macros, or both, connected together to implement the higher level function. An example of a macro function is a 16-bit counter.

A TNM constraint attached to a macro indicates that all elements inside the macro (at all levels of hierarchy below the tagged macro) are part of the named group.

When a macro contains more than one symbol type and you want to group only a single type, use the TNM identifier in conjunction with one of the predefined groups: FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, DSPS, BRAM_PORTA, BRAM_PORTB, and MULTS as indicated by the following syntax examples.

UCF Syntax Example

```
INST "instance_name" TNM=FFS identifier;
INST "instance_name" TNM=RAMS identifier;
INST "instance_name" TNM=LATCHES identifier;
INST "instance_name" TNM=PADS identifier;
INST "instance_name" TNM=CPUS identifier;
INST "instance_name" TNM=HSIOS identifier;
INST "instance_name" TNM=MULTS identifier;
```

If multiple symbols of the same type are contained in the same hierarchical block, you can simply flag that hierarchical symbol, as illustrated by the following figure. In the figure, all flip-flops included in the macro are tagged with the TNM "FLOPS." By tagging the macro symbol, you need not tag each underlying symbol individually.

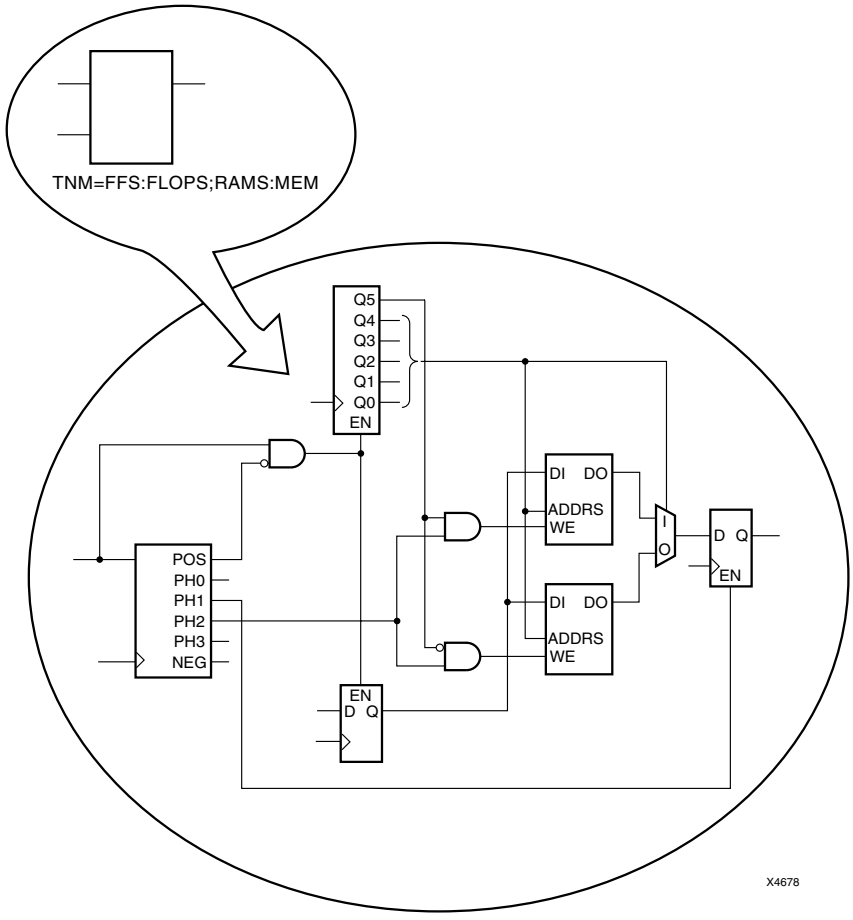

Figure 84-3: **TNM on Macro Symbol**

Placing TNMs on Nets or Pins to Group Flip-Flops and Latches

You can easily group flip-flops, latches, or both by flagging a common input net, typically either a clock net or an enable net. If you attach a TNM to a net or driver pin, that TNM applies to all flip-flops and input latches that are reached through the net or pin. That is, that path is traced forward, through any number of gates or buffers, until it reaches a flip-flop or input latch. That element is added to the specified TNM group.

The following figure illustrates the use of a TNM on a net that traces forward to create a group of flip-flops. In the figure, the constraint TNM=FLOPS traces forward to the first two flip-flops, which form a group called FLOPS. The bottom flip-flop is not part of the group FLOPS.

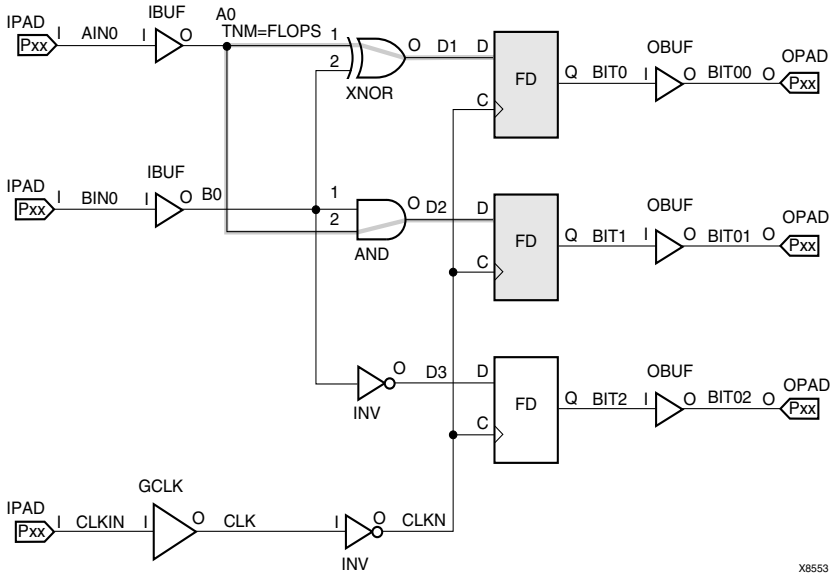

Figure 84-4: **TNM on Net Used to Group Flip-Flops**

The following figure illustrates placing a TNM on a clock net, which traces forward to all three flip-flops and forms the group Q_FLOPS.

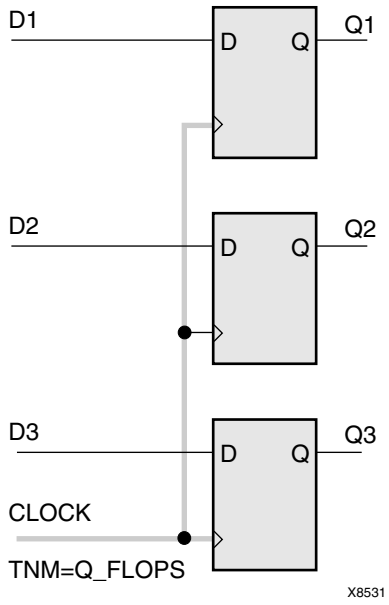

Figure 84-5: **TNM on Clock Pin Used to Group Flip-Flops**

The TNM parameter on nets or pins is allowed to have a qualifier. For example, on schematics:

```
TNM=FFS data;
TNM=RAMS fifo;
TNM=LATCHES capture;
```

In UCF files:

```
{NET|PIN} "net_or_pin_name" TNM=FFS data;
{NET|PIN} "net_or_pin_name" TNM=RAMS fifo;
{NET|PIN} "net_or_pin_name" TNM=LATCHES capture;
```

A qualified TNM is traced forward until it reaches the first storage element (FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, and MULTS). If that type of storage element matches the qualifier, the storage element is given that TNM value. Whether or not there is a match, the TNM is *not* traced through that storage element.

TNM parameters on nets or pins are never traced through a storage element (FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, and MULTS).

TNM Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net or a macro

- Attribute Name: TNM
- Attribute Values: *identifier*

For a discussion of *identifier*, see "UCF and NCF Syntax Example" in this chapter.

ABEL Syntax Example

```
XILINX PROPERTY 'TNM=identifier mysignal';
```

UCF and NCF Syntax Example

```
{NET|INST|PIN} "net_or_pin_or_inst_name" TNM=[predefined_group]
identifier;
```

where

- *predefined_group* can be:
  - All of the members of a predefined group using the keywords FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, and MULTS as follows:
    - FFS refers to all CLB and IOB flip-flops. Flip-flops built from function generators are not included.
    - RAMS refers to all RAMs for architectures with RAMS. This includes LUT RAMS and BLOCK RAMS.
    - PADS refers to all I/O pads.
    - LATCHES refers to all CLB or IOB latches. Latches built from function generators are not included.
    - MULTS group the Spartan-3, Spartan-3A, and Spartan-3E and Virtex-II registered multiplier.
    - CPUS group the Virtex-II Pro or Virtex-II Pro X processor.
    - HSIOS to group the Virtex-II Pro or Virtex-II Pro X gigabit transceiver.
  - A subset of elements in a *predefined_group* can be defined as follows:

    ```
    predefined_group (name_qualifier1... name_qualifiern)
    ```

    where
    - *name_qualifiern* can be any combination of letters, numbers, or underscores. The *name_qualifier* type (net or instance) is based on the element type that TNM is placed on. If the TNM is on a NET, the *name_qualifier* is a net name. If the TNM is an instance (INST), the *name_qualifier* is an instance name.

    For example:

    ```
    NET clk TNM = FFS (my_flop) Grp1;
    INST clk TNM = FFS (my_macro) Grp2;
    ```
- *identifier* can be any combination of letters, numbers, or underscores.

The *identifier* cannot be any the following reserved words: FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, MULTS, RISING, FALLING, TRANSHI, TRANSLO, or EXCEPT.

In addition, do not use the constraints in the table below, which are also reserved words, as *identifiers*.

Table 84-1: **Reserved Words (Constraints)**

ADD	ALU	ASSIGN
BEL	BLKNM	CAP
CLKDV_DIVIDE	CLBNM	CMOS
CYMODE	DECODE	DEF
DIVIDE1_BY	DIVIDE2_BY	DOUBLE
DRIVE	DUTY_CYCLE_ CORRECTION	FAST
FBKINV	FILE	F_SET
HBLKNM	HU_SET	H_SET
INIT	INIT OX	INTERNAL
IOB	IOSTANDARD	LIBVER
LOC	LOWPWR	MAP
MEDFAST	MEDSLOW	MINIM
NODELAY	OPT	OSC
RES	RLOC	RLOC_ORIGIN
RLOC_RANGE	SCHNM	SLOW
STARTUP_WAIT	SYSTEM	TNM
TRIM	TS	TTL
TYPE	USE_RLOC	U_SET

You can specify as many groups of end points as are necessary to describe the performance requirements of your design. However, to simplify the specification process and reduce the place and route time, use as few groups as possible.

XCF Syntax Example

See "UCF and NCF Syntax Example" in this chapter.

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Create next to "Group elements by instance name" or Create next to "Group elements by hierarchy" and fill out the Time Name dialog box.

Timing Name Net (TNM_NET)

TNM_NET Architecture Support

The TNM_NET constraint applies to FPGA devices only.

TNM_NET Applicable Elements

Nets

TNM_NET Description

TNM_NET is a basic grouping constraint. TNM_NET (timing name for nets) identifies the elements that make up a group, which can then be used in a timing specification. TNM_NET is essentially equivalent to TNM on a net *except* for input pad nets.

Special rules apply when using TNM_NET with the PERIOD constraint for DLL/DCM/PLLs. For more information, see "PERIOD Specifications on CLKDLLs, DCMs and PLLs" in the "Period (PERIOD)" constraint.

A TNM_NET is a property that you normally use in conjunction with an HDL design to tag a specific net. All downstream synchronous elements and pads tagged with the TNM_NET identifier are considered a group.

TNM_NET (Timing Name - Net) tags specific synchronous elements, pads, and latches as members of a group to simplify the application of timing specifications to the group. NGDBuild never transfers a TNM_NET constraint from the attached net to an input pad, as it does with the TNM constraint.

TNM_NET Rules

The following rules apply to TNM_NET:

- TNM_NETs applied to pad nets propagate forward through the IBUF or OBUF and any other combinatorial logic to synchronous logic or pads.
- TNM_NETs applied to a clock-pad-net propagate forward through the clock buffer.
- Special rules apply when using TNM_NET with the PERIOD constraint for Virtex™, Spartan™-II, Virtex-II, Virtex-II Pro, Virtex-II Pro X, Virtex-4, and Virtex-5 DLLs, DCMs, and PLLs.

Use TNM_NET to define certain types of nets that cannot be adequately described by the TNM constraint.

For example, consider the following design

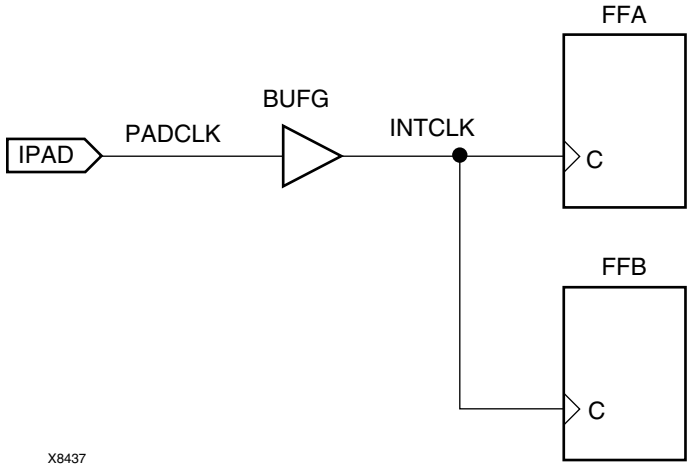

Figure 85-1: **TNM Associated with the IPAD**

In the preceding design, a TNM associated with the IPAD symbol includes only the PAD symbol as a member in a timing analysis group. For example, the following UCF file entry creates a time group that includes the IPAD symbol only.

```
NET "PADCLK" TNM= "PADGRP"; (UCF file example)
```

However, using TNM to define a time group for the net PADCLK creates an empty time group.

```
NET "PADCLK" TNM=FFS "FFGRP";(UCF file example)
```

All properties that apply to a pad are transferred from the net to the PAD symbol. Since the TNM is transferred from the net to the PAD symbol, the qualifier, "FFS" does not match the PAD symbol.

To overcome this obstacle for schematic designs using TNM, you can create a time group for the INTCLK net.

```
NET "INTCLK" TNM=FFS FFGRP;(UCF file example)
```

However, for HDL designs, the only meaningful net names are the ones connected directly to pads. Then, use TNM_NET to create the FFGRP time group.

```
NET PADCLK TNM_NET=FFS FFGRP;(UCF file example)
```

NGDBuild does not transfer a TNM_NET constraint from a net to an IPAD as it does with TNM.

You can use TNM_NET in NCF or UCF files as a property attached to a net in an input netlist (EDIF or NGC). TNM_NET is not supported in PCF files.

You can use TNM_NET with nets or instances. If TNM_NET is used with any other object such as a pin or symbol, a warning is generated and the TNM_NET definition is ignored.

TNM_NET Propagation Rules

It is illegal to attach TNM_NET to a design element.

TNM_NET Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: TNM_NET
- Attribute Values: *identifier*

For a discussion of *identifier*, see "UCF and NCF Syntax Example" in this chapter.

UCF and NCF Syntax Example

```
{NET|INST} "net_name" TNM_NET=[predefined_group:]identifier;
```

where

- *predefined_group* can be:
  - All of the members of a predefined group using the keywords FFS, RAMS, PADS, MULTS, HSIOS, CPUS, DSPS, BRAMS_PORTA, BRAMS_PORTB or LATCHES as follows:
    - FFS refers to all CLB and IOB flip-flops. Flip-flops built from function generators are not included.
    - RAMS refers to all RAMs for architectures with RAMS. This includes LUT RAMS and BLOCK RAMS.
    - PADS refers to all I/O pads.
    - MULTS group the Spartan-3, Spartan-3A, and Spartan-3E and Virtex-II registered multiplier.
    - CPUS group the Virtex-II Pro or Virtex-II Pro X processor.
    - DSPS is used to group DSP elements like the Virtex-4 DSP48.
    - HSIOS group the Virtex-II Pro or Virtex-II Pro X gigabit transceiver.
    - LATCHES refers to all CLB or IOB latches. Latches built from function generators are not included.
  - A subset of elements in a *predefined_group* can be defined as follows:

    ```
    predefined_group (name_qualifier1... name_qualifiern)
    ```

    where
    - *name_qualifiern* can be any combination of letters, numbers, or underscores. The *name_qualifier* type (net or instance) is based on the element type that TNM_NET is placed on. If the TNM_NET is on a NET, the *name_qualifier* is a net name. If the TNM_NET is an instance (INST), the *name_qualifier* is an instance name.

    For example:

    ```
    NET clk TNM_NET = FFS (my_flop) Grp1;
    INST clk TNM_NET = FFS (my_macro) Grp2;
    ```
- *identifier* can be any combination of letters, numbers, or underscores.

The *identifier* cannot be any the following reserved words: FFS, RAMS, LATCHES, PADS, CPUS, HSIOS, MULTS, RISING, FALLING, TRANSHI, TRANSLO, or EXCEPT.

In addition, do not use the constraints/reserved words located in Table 84-1, page 321 as *identifiers*.

The following statement identifies all flip-flops fanning out from the PADCLK net as a member of the timing group GRP1.

```
NET "PADCLK" TNM_NET=FFS "GRP1";
```

XCF Syntax Example

XST supports TNM_NET with the following limitation: only a single pattern supported for predefined groups.

The following command syntax is supported:

```
NET "PADCLK" TNM_NET=FFS "GRP1";
```

The following command syntax is *not* supported:

```
NET "PADCLK" TNM_NET = FFS(machine/*:xcounter/*) TG1;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Create next to "Group elements associated by Nets" and fill out the Time Name dialog box.

Timing Point Synchronization (TPSYNC)

TPSYNC Architecture Support

The TPSYNC constraint applies to FPGA devices only.

TPSYNC Applicable Elements

- Nets
- Instances
- Pins

TPSYNC Description

TPSYNC is an advanced grouping constraint. It flags a particular point or a set of points with an identifier for reference in subsequent timing specifications. You can use the same identifier on several points, in which case timing analysis treats the points as a group.

When the timing of a design must be designed from or to a point that is not a synchronous element or I/O pad, the following rules apply if a TPSYNC timing point is attached to a net, macro pin, output or input pin of a primitive, or an instance.

- A net: the source of the net is identified as a potential source or destination for timing specifications.
- A macro pin: all of the sources inside the macro that drive the pin to which the constraint is attached are identified as potential sources or destinations for timing specifications. If the macro pin is an input pin (that is, if there are no sources for the pin in the macro), then all of the load pins in the macro are flagged as synchronous points.

  In the following diagram, POINTY applies to the inverter.

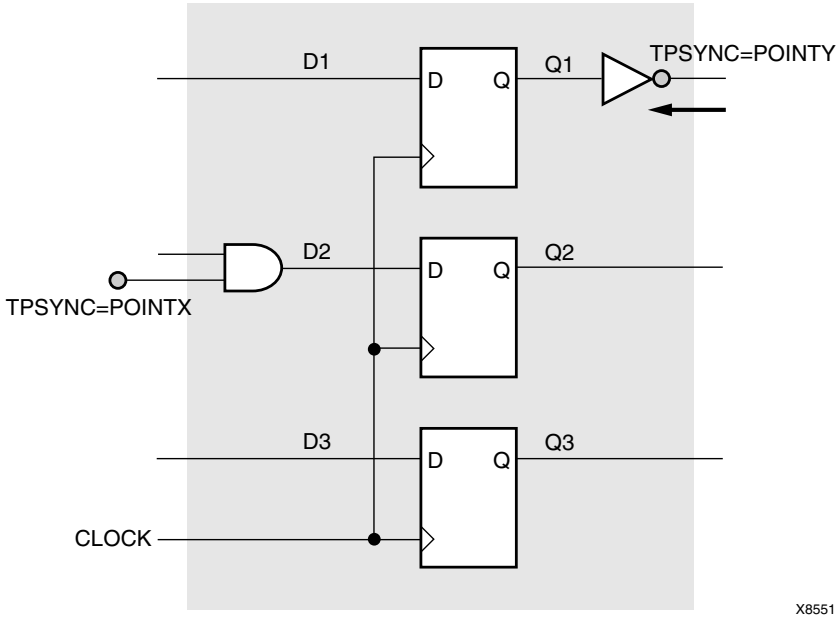

Figure 86-1: **TPSYNCs Attached to Macro Pins**

- The output pin of a primitive — the primitive's output is flagged as a potential source or destination for timing specifications.
- The input pin of a primitive — the primitive's input is flagged as a potential source or destination for timing specifications.
- An instance — the output of that element is identified as a potential source or destination for timing specifications.
- A primitive symbol—Attached to a primitive symbol, TPSYNC identifies the outputs of that element as a potential source or destination for timing specifications. See the following figure.

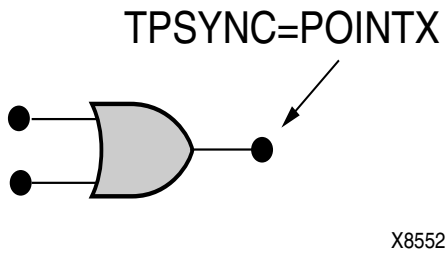

Figure 86-2: **TPSYNC Attached to a Primitive Symbol**

The use of a TPSYNC timing point to define a synchronous point in a design implies that the flagged point cannot be merged into a function generator. For example, consider the following diagram.

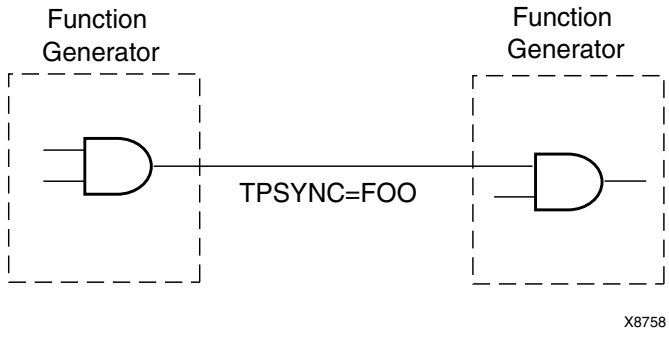

Figure 86-3: **Working with Two Gates**

In this example, because of the TPSYNC definition, the two gates cannot be merged into a single function generator.

TPSYNC Propagation Rules

See "TPSYNC Description."

TPSYNC Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attached to a net, instance, or pin
- Attribute Name: TPSYNC
- Attribute Values: *identifier*

where

- *identifier* is a name that is used in timing specifications in the same way that groups are used

UCF and NCF Syntax Example

```
NET "net_name" TPSYNC=identifier;
INST "instance_name" TPSYNC=identifier;
PIN "pin_name" TPSYNC=identifier;
```

where

- *identifier* is a name that is used in timing specifications in the same way that groups are used

All flagged points are used as a source or destination or both for the specification where the TPSYNC identifier is used.

The name for the identifier must be unique to any identifier used for a TNM or TNM_NET grouping constraint.

The following statement identifies latch as a potential source or destination for timing specifications for the net logic_latch.

```
NET "logic_latch" TPSYNC=latch;
```

Timing Thru Points (TPTHRU)

TPTHRU Architecture Support

The TPTHRU constraint applies to FPGA devices only.

TPTHRU Applicable Elements

- Nets
- Pins
- Instances

TPTHRU Description

TPTHRU is an advanced grouping constraint. It flags a particular point or a set of points with an identifier for reference in subsequent timing specifications. If you use the same identifier on several points, timing analysis treats the points as a group. For more information, see the "Timing Specifications (TIMESPEC)" constraint.

Use the TPTHRU constraint when it is necessary to define intermediate points on a path to which a specification applies. For more information, see the "Timing Specification Identifier (TSidentifier)" constraint.

TPTHRU Propagation Rules

Not applicable

TPTHRU Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net, instance, or pin
- Attribute Name: TPTHRU
- Attribute Values: *identifier*

  For a discussion of *identifier*, see "UCF and NCF Syntax Example" in this chapter.

UCF and NCF Syntax Example

The basic UCF syntax is as follows:

```
NET "net_name" TPTHRU=identifier;
INST "instance_name" TPTHRU=identifier;
PIN "instance_name.pin_name" TPTHRU="thru_group_name";
```

where

- *identifier* is a name used in timing specifications for further qualifying timing paths within a design

The name for the identifier must be different from any identifier used for a TNM constraint.

Using TPTHRU in a FROM TO Constraint

It is sometimes convenient to define intermediate points on a path to which a specification applies. This defines the maximum allowable delay and has the syntax shown in the following sections.

UCF Syntax with TIMESPEC

```
TIMESPEC "TSidentifier"=FROM "source_group" THRU "thru_point" [THRU
"thru_point"] TO "dest_group" allowable_delay [units];
TIMESPEC "TSidentifier"=FROM "source_group" THRU "thru_point" [THRU
"thru_point"] allowable_delay [units];
```

where

- *identifier* is an ASCII string made up of the characters A..Z, a..z, 0..9, and underscore (_)
- *source_group* and *dest_group* are user-defined groups, predefined groups or TPSYNCs
- *thru_point* is an intermediate point used to qualify the path, defined using a TPTHRU constraint
- *allowable_delay* is the timing requirement
- *units* is an optional field to indicate the units for the allowable delay. Default units are nanoseconds, but the timing number can be followed by ps, ns, micro, ms, GHz, MHz, or KHz to indicate the intended units.

The example shows how to use the TPTHRU constraint with the THRU constraint on a schematic. The UCF syntax is as follows.

```
INST "FLOPA" TNM="A";
INST "FLOPB" TNM="B";

NET "MYNET" TPTHRU="ABC";
TIMESPEC "TSpath1"=FROM "A" THRU "ABC" TO "B" 30;
```

The following statement identifies the net on_the_way as an intermediate point on a path to which the timing specification named "here" applies.

```
NET "on_the_way" TPTHRU="here";
```

Note: The following NCF construct is not supported.

```
TIMESPECT "TS_1"=THRU "Thru_grp" 30.0
```

XCF Syntax Example

Not supported

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Advanced tab, click Create next to "Timing THRU Points (TPTHRU)" and then fill out the Timing THRU Point dialog box.

PCF Syntax Example

```
PATH "name"=FROM "source" THRU "thru_pt1" ...THRU "thru_ptn" TO
"destination";
```

You are not required to have a FROM, THRU, and TO. You can have almost any combination (such as FROM-TO, FROM-THRU-TO, THRU-TO, TO, FROM, FROM-THRU-THRU-THRU-TO, and FROM-THRU). There is no restriction on the number of THRU points. The source, thru points, and destination can be a net, bel, comp, macro, pin, or timegroup.

Timing Specification Identifier (TSidentifier)

TSidentifier Architecture Support

The TSindentifier constraint applies to all FPGA and CPLD devices.

TSidentifier Applicable Elements

TIMESPEC keywords

TSidentifier Description

TS*identifier* is a basic timing constraint. TS*identifier* properties beginning with the letters "TS" are used with the TIMESPEC keyword in a UCF file. The value of TS*identifier* corresponds to a specific timing specification that can then be applied to paths in the design.

TSidentifier Propagation Rules

It is illegal to attach TS*identifier* to a net, signal, or design element.

All the following syntax definitions use a space as a separator. The use of a colon (:) as a separator is optional.

TSidentifier Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Examples

Following are the UCF and NCF syntax examples:

- "Defining a Maximum Allowable Delay"
- "Defining Intermediate Points (UCF)"
- "Defining a Linked Specification"
- "Defining a Clock Period"
- "Specifying Derived Clocks"
- "Ignoring Paths"

Defining a Maximum Allowable Delay

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group"
allowable_delay [units];
```

Defining Intermediate Points (UCF)

```
TIMESPEC "TSidentifier"=FROM "source_group" THRU "thru_point" [THRU
"thru_point1"... "thru_pointn"] TO "dest_group" allowable_delay
[units];
```

where

- *identifier* is an ASCII string made up of the characters A-Z, a-z, 0-9, and _

- *source_group* and *dest_group* are user-defined or predefined groups
- *thru_point* is an intermediate point used to qualify the path, defined using a TPTHRU constraint
- *allowable_delay* is the timing requirement value
- *units* is an optional field to indicate the units for the allowable delay. The default units are nanoseconds (ns), but the timing number can be followed by ps, ns, micro, ms, GHz, MHz, or kHz to indicate the intended units.

Defining a Linked Specification

This allows you to link the timing number used in one specification to another specification.

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group"
another_TSid [/ | *] number;
```

where

- *identifier* is an ASCII string made up of the characters A-Z, a-z, 0-9, and _
- *source_group* and *dest_group* are user-defined or predefined groups
- *another_Tsid* is the name of another timespec
- *number* is a floating point number

Defining a Clock Period

This allows more complex derivative relationships to be defined as well as a simple clock period.

```
TIMESPEC "TSidentifier"=PERIOD "TNM_reference" value [units] [{HIGH |
LOW} [high_or_low_time [hi_lo_units]]] INPUT_JITTER value;
```

where

- *identifier* is a reference identifier with a unique name
- *TNM_reference* is the identifier name attached to a clock net (or a net in the clock path) using a TNM constraint
- *value* is the required clock period
- *units* is an optional field to indicate the units for the allowable delay. The default units are nanoseconds (ns), but the timing number can be followed by micro, ms, ps, ns, GHz, MHz, or kHz to indicate the intended units
- HIGH or LOW can be optionally specified to indicate whether the first pulse is to be High or Low
- *high_or_low_time* is the optional High or Low time, depending on the preceding keyword. If an actual time is specified, it must be less than the period. If no High or Low time is specified, the default duty cycle is 50 percent.
- *hi_lo_units* is an optional field to indicate the units for the duty cycle. The default is nanoseconds (ns), but the High or Low time number can be followed by ps, micro, ms, ns or % if the High or Low time is an actual time measurement.

Specifying Derived Clocks

```
TIMESPEC "TSidentifier"=PERIOD "TNM_reference"
"another_PERIOD_identifier" [/ | *] number [{HIGH | LOW}
[high_or_low_time [hi_lo_units]]] INPUT_JITTER value;
```

where

- *TNM_reference* is the identifier name attached to a clock net (or a net in the clock path) using a TNM constraint
- *another_PERIOD_identifier* is the name of the identifier used on another period specification
- *number* is a floating point number
- HIGH or LOW can be optionally specified to indicate whether the first pulse is to be High or Low
- *high_or_low_time* is the optional High or Low time, depending on the preceding keyword. If an actual time is specified, it must be less than the period. If no High or Low time is specified, the default duty cycle is 50 percent.
- *hi_lo_units* is an optional field to indicate the units for the duty cycle. The default is nanoseconds (ns), but the High or Low time number can be followed by ps, micro, ms, or % if the High or Low time is an actual time measurement.

Ignoring Paths

Note: This form is not supported for CPLD devices.

There are situations in which a path that exercises a certain net should be ignored because all paths through the net, instance, or instance pin are not important from a timing specification point of view.

```
TIMESPEC "TSidentifier"=FROM "source_group" TO "dest_group" TIG;
```

or

```
TIMESPEC "TSidentifier"=FROM "source_group" THRU "thru_point" [THRU
"thru_point1"... "thru_pointn"] TO "dest_group" TIG;
```

where

- *identifier* is an ASCII string made up of the characters A-Z, a-z 0-9, and _
- *source_group* and *dest_group* are user-defined or predefined groups
- *thru_point* is an intermediate point used to qualify the path, defined using a TPTHRU constraint

The following statement says that the timing specification TS_35 calls for a maximum allowable delay of 50 ns between the groups “here” and “there”.

```
TIMESPEC "TS_35"=FROM "here" TO "there" 50;
```

The following statement says that the timing specification TS_70 calls for a 25 ns clock period for clock_a, with the first pulse being High for a duration of 15 ns.

```
TIMESPEC "TS_70"=PERIOD "clock_a" 25 high 15;
```

For more information, see “Logical Constraints” and “Physical Constraints” in Chapter 2, “Constraint Types.”

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

You can enter clock period timing constraints in the Global tab. Input setup time and clock-to-output delay can be entered for specific pads in the Ports tab, or for all pads related to a given clock in the Global tab. Combinatorial pad-to-pad delays can be entered in the Advanced tab, or for all pad-to-pad paths in the Global tab.

PCF Syntax Example

The same as the UCF syntax without the TIMESPEC keyword.

FPGA Editor Syntax Example

To set constraints, in the FPGA Editor main window, click Properties of Selected Items from the Edit menu. With a component, net, path, or pin selected, you can set a TSid from the Physical Constraints tab.

U_SET

U_SET Architecture Support

The U_SET constraint applies to FPGA devices only.

U_SET Applicable Elements

To see which design elements can be used with which device families, see the Xilinx® *Libraries Guide*s. For more information, see the device data sheet.

1. Registers
2. FMAP
3. Macro Instance
4. ROM
5. RAMS, RAMD
6. BUFT
7. MULT18X18S
8. RAMB4_Sm_Sn, RAMB4_Sn
9. RAMB16_*Sm_Sn*, *RAMB16_Sn*
10. RAMB16
11. DSP48

U_SET Description

U_SET is an advanced mapping constraint. It groups design elements with attached RLOC constraints that are distributed throughout the design hierarchy into a single set. The elements that are members of a U_SET can cross the design hierarchy. You can arbitrarily select objects without regard to the design hierarchy and tag them as members of a U_SET. For more information, see "U_SET" in this chapter.

U_SET Propagation Rules

U_SET is a macro constraint and any attachment to a net is illegal.

U_SET Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: U_SET
- Attribute Values: *name*

where

- *name* is the identifier of the set

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute U_SET: string;
```

Specify the VHDL constraint as follows:

```
attribute U_SET of {component_name|label_name}: {component|label} is
"name";
```

where

- *name* is the identifier of the set

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* U_SET = "name" *)
```

where

- *name* is the identifier of the set

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" U_SET=name;
```

where

- *name* is the identifier of the set

This name is absolute. It is not prefixed by a hierarchical qualifier.

The following statement specifies that the design element ELEM_1 be in a set called JET_SET.

```
INST "$1I3245/ELEM_1" U_SET=JET_SET;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 INST "instance_name" U_SET=uset_name;
END;
```

Use Relative Location (USE_RLOC)

USE_RLOC Architecture Support

The USE_RLOC constraint applies to FPGA devices only.

USE_RLOC Applicable Elements

Instances or macros that are members of sets

USE_RLOC Description

USE_RLOC is an advanced mapping and placement constraint. It turns RLOC on or off for a specific element or section of a set. For more information about USE_RLOC, see "Toggling the Status of RLOC Constraints" in the "Relative Location (RLOC)" constraint.

USE_RLOC Propagation Rules

It is illegal to attach USE_RLOC to a net. When attached to a design element, U_SET propagates to all applicable elements in the hierarchy within the design element.

USE_RLOC Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a member of a set
- Attribute Name: USE_RLOC
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute USE_RLOC: string;
```

Specify the VHDL constraint as follows:

```
attribute USE_RLOC of entity_name: entity is "{TRUE|FALSE}";
```

where

- **TRUE** turns on the RLOC constraint for a specific element
- **FALSE** turns it off

The default is **TRUE**.

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* USE_RLOC = "{TRUE|FALSE}" *)
```

where

- **TRUE** turns on the RLOC constraint for a specific element
- **FALSE** turns it off

The default is **TRUE**.

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" USE_RLOC={TRUE|FALSE};
```

where

- **TRUE** turns on the RLOC constraint for a specific element
- **FALSE** turns it off

The default is **TRUE**.

XCF Syntax Example

```
MODEL "entity_name" use_rloc={true|false};
```

Use Low Skew Lines (USELOWSKEWLINES)

USELOWSKEWLINES Architecture Support

The USELOWSKEWLINES constraint applies to the following devices:

- Virtex™
- Virtex-E
- Spartan™-II
- Spartan-IIE

USELOWSKEWLINES Applicable Elements

Nets

USELOWSKEWLINES Description

USELOWSKEWLINES is a PAR routing constraint.

The Spartan-II, Spartan-IIE, Virtex, and Virtex-E devices have 24 horizontal low skew resources which are intended to drive slower secondary clocks and may be used for high fanout nets. These 24 horizontal resources connect to the 12 vertical longlines in the column. The USELOWSKEWLINES constraint specifies the use of low skew routing resources for any net. You can use these resources for both internally generated and externally generated signals. Externally generated signals are those driven by IOBs.

USELOWSKEWLINES on a net directs PAR to route the net on one of the low skew resources. When this constraint is used, the timing tool automatically accounts for and reports skew on register-to-register paths that utilize those low skew resources. Specify USELOWSKEWLINES only when all four primary global clocks have been used.

USELOWSKEWLINES Propagation Rules

Applies to attached net

USELOWSKEWLINES Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to an output net
- Attribute Name: USELOWSKEWLINES
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute USELOWSKEWLINES: string;
```

Specify the VHDL constraint as follows:

```
attribute USELOWSKEWLINES of signal_name : signal is
"{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* USELOWSKEWLINES = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

This statement forces net $1I87/1N6745 to be routed on one of the device's low skew resources.

```
NET "$1I87/$1N6745" USELOWSKEWLINES;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 NET "signal_name" uselowskewlines={yes|true};
END;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Misc tab, click Identify next to "Nets to use low Skew resources". Complete the Low Skew Resource dialog box.

PCF Syntax Example

Same as UCF syntax

VCCAUX

VCCAUX Architecture Support

The VCCAUX constraint applies to Spartan™-3A devices only.

VCCAUX Applicable Elements

The VCCAUX attribute is a global attribute for the Spartan-3A device and is not attached to any particular element.

VCCAUX Description

The VCCAUX constraint is used to define the voltage value of the VCCAUX pin for the device. The valid values for this attribute is 2.5 (default) or 3.3. This attribute affects the banking rules for I/O placement within the automated placer, as well as in the Pace pin assignments tool. It also affects the end-generated bitstream for the device.

VCCAUX Syntax Examples

Following are syntax examples using this constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

Placed on output or bi-directional port:

```
CONFIG VCCAUX="value";
```

Where value is 2.5 or 3.3

Example:

```
CONFIG VCCAUX=3.3;
```

Voltage (VOLTAGE)

VOLTAGE Architecture Support

The VOLTAGE constraint applies to FPGA devices only.

VOLTAGE Applicable Elements

Global

VOLTAGE Description

VOLTAGE is an advanced timing constraint. It allows the specification of the operating voltage, which provides a means of prorating delay characteristics based on the specified voltage. Prorating is a scaling operation on existing speed file delays and is applied globally to all delays.

Note: Newer devices may not support Voltage prorating until the timing information (speed files) are marked as production status.

Each architecture has its own specific range of supported voltages. If the entered voltage does not fall within the supported range, the constraint is ignored and an architecture-specific default value is used instead. Also note that the error message for this condition appears during static timing.

VOLTAGE Propagation Rules

It is illegal to attach VOLTAGE to a net, signal, or design element.

VOLTAGE Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

UCF and NCF Syntax Example

```
VOLTAGE=value [V];
```

where

- *value* is real number specifying the voltage
- V indicates volts, the default voltage unit

The following statement specifies that the analysis for everything relating to speed file delays assumes an operating power of 5 volts.

```
VOLTAGE=5;
```

Constraints Editor Syntax Example

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

In the Misc tab, click Specify next to “Voltage” and then fill out the Voltage dialog box.

PCF Syntax Example

Same as UCF

VREF

VREF Architecture Support

The VREF constraint applies to CoolRunner™-II devices with 128 macrocells and larger.

VREF Applicable Elements

Global

VREF Description

VREF applies to the design as a global attribute (not directly applicable to any element in the design). The constraint configures listed pins as VREF supply pins to be used in conjunction with other I/O pins designated with one of the SSTL or HSTL I/O Standards.

Because VREF is selectable on any I/O in CoolRunner-II designs, this constraint allows you to select which pins are VREF pins. Make sure you double-check pin assignment in the report (RPT) file. If you do not specify any VREF pins for the differential I/O standards, HSTL and SSTL, or if you do not specify sufficient VREF pins within the required proximity of differential I/O pins, the fitter automatically assigns sufficient VREF.

VREF Propagation Rules

Configures listed pins as VREF supply pins to be used in conjunction with other I/O pins designated with one of the SSTL or HSTL I/O Standards.

VREF Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

VREF=*value_list* (on CONFIG symbol)

The legal values are:

- P*nn*

  where

  - *nn* is a numeric pin number
- *rc*

  where

  - *r*=alphabetic row
  - *c*=numeric column

UCF and NCF Syntax Example

```
CONFIG VREF=value_list;
```

The legal values are:

- P*nn*

where

- *nn* is a numeric pin number

- *rc*

  where

  - *r*=alphabetic row

  c=numeric column

```
CONFIG VREF=P12,P13;
```

Wire And (WIREAND)

WIREAND Architecture Support

The WIREAND constraint applies the following devices:

- XC9500™
- XC9500XL
- XC9500XV

WIREAND Applicable Elements

Any net

WIREAND Description

WIREAND is an advanced fitter constraint. It forces a tagged node to be implemented as a wired AND function in the interconnect (UIM and Fastconnect).

WIREAND Propagation Rules

WIREAND is a net constraint. Any attachment to a design element is illegal.

WIREAND Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a net
- Attribute Name: WIREAND
- Attribute Values: TRUE, FALSE

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute WIREAND: string;
```

Specify the VHDL constraint as follows:

```
attribute WIREAND of signal_name : signal is "{YES|NO|TRUE|FALSE}";
```

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* WIREAND = "{YES|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The following statement specifies that the net named SIG_11 be implemented as a wired AND when optimized.

```
NET "$I16789/SIG_11" WIREAND;
```

XBLKNM

XBLKNM Architecture Support

The XBLKNM constraint applies to FPGA devices only.

XBLKNM Applicable Elements

To see which design elements can be used with which device families, see the Xilinx *Libraries Guides*. For more information, see the device data sheet.

XBLKNM Description

XBLKNM is an advanced mapping constraint. It assigns block names to qualifying primitives and logic elements. If the same XBLKNM attribute is assigned to more than one instance, the software attempts to pack logic with the same block name into one or more CLBs. Conversely, two symbols with different XBLKNM names are not mapped into the same block. Placing the same XBLKNMs on instances that do not fit within one block creates an error.

Specifying identical XBLKNM attributes on FMAP symbols tells the software to group the associated function generators into a single CLB. Using XBLKNM, you can partition a complete CLB without constraining the CLB to a physical location on the device.

Hierarchical paths are not prefixed to XBLKNM attributes, so XBLKNM attributes for different CLBs must be unique throughout the entire design.

The BLKNM attribute allows any elements except those with a different BLKNM to be mapped into the same physical component. XBLKNM, however, allows only elements with the same XBLKNM to be mapped into the same physical component. Elements without an XBLKNM cannot be not mapped into the same physical component as those with an XBLKNM.

XBLKNM Propagation Rules

Applies to the design element to which it is attached

XBLKNM Syntax Examples

Following are syntax examples using the constraint with particular tools or methods. If a tool or method is not listed, the constraint may not be used with it.

Schematic Syntax Example

- Attach to a valid instance
- Attribute Name: XBLKNM
- Attribute Values: *block_name*

where

- *block_name* is a valid block name for that type of symbol

VHDL Syntax Example

Declare the VHDL constraint as follows:

```
attribute XBLKNM: string;
```

Specify the VHDL constraint as follows:

```
attribute XBLKNM of {component_name|label_name}: {component|label} is
"block_name";
```

where

- *block_name* is a valid block name for that type of symbol

For more information on basic VHDL syntax, see "VHDL" in Chapter 3.

Verilog Syntax Example

Specify the Verilog constraint as follows:

```
(* XBLKNM = "block_name" *)
```

where

- *block_name* is a valid block name for that type of symbol

For more information on basic Verilog syntax, see "Verilog" in Chapter 3.

UCF and NCF Syntax Example

The basic UCF syntax is:

```
INST "instance_name" XBLKNM=block_name;
```

where

- *block_name* is a valid block name for that type of symbol

The following statement assigns an instantiation of an element named flip_flop2 to a block named U1358.

```
INST "$1I87/flip_flop2" XBLKNM=U1358;
```

XCF Syntax Example

```
BEGIN MODEL "entity_name"
 INST "instance_name" xblknm=xblknm_name;
END;
```